

NWCRG  
Internet-Draft  
Intended status: Experimental  
Expires: March 12, 2020

J. Heide  
Steinwurf Aps  
S. Shi  
K. Fouli  
M. Medard  
Code On Network Coding LLC  
V. Chook  
Inmarsat PLC  
September 09, 2019

Random Linear Network Coding (RLNC)-Based Symbol Representation  
draft-heide-nwcrg-rlnc-03

Abstract

This document describes a symbol representation for Random Linear Network Coding (RLNC) schemes used for reliable data transfer. Specifically, the following features are discussed and incorporated: both block RLNC and a sliding window RLNC, varying data frame sizes, and one or multiple symbols associated with a single symbol representation header.

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 March 12, 2020.

## Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
2. Symbol Representation . . . . .	3
2.1. Representation Setup . . . . .	4
2.2. Field Types and Formats . . . . .	5
2.3. Externally Specified Parameters Required . . . . .	7
2.4. Small Encoding Window . . . . .	7
2.4.1. Examples . . . . .	9
2.5. Large Encoding Window . . . . .	10
3. Security Considerations . . . . .	11
4. IANA Considerations . . . . .	11
5. References . . . . .	11
5.1. Normative References . . . . .	11
5.2. Informative References . . . . .	12
Authors' Addresses . . . . .	12

## 1. Introduction

Symbol representation specifies the format of the symbol-carrying data unit that is to be used in network coding operations, including header format and symbol concatenation. This document describes a symbol representation format intended to be used for Network Coding in general, and for Random Linear Network Coding (RLNC) in particular [HK03].

Owing to its dynamic structure, network coding has requirements that are distinct from conventional point-to-point codes, leading to a highly reconfigurable symbol set. Consequently, the design choices related to symbol representation are particularly important in network coding as they have a direct impact on the viability of network protocols, topologies, and architecture [RLNC-Background]. For example, recoding [RLNC-Background] requires the coefficients to

be accessible at the recoding nodes. Hence, architectures and protocols requiring recoding must specify coefficient location in their symbol representation.

In addition to providing background on RLNC, [RLNC-Background] argues that careful design and specification of a symbol representation is a requirement for any viable network coding protocol, architecture, or topology.

## 2. Symbol Representation

This section provides a symbol representation design for implementing RLNC-based erasure correction schemes. In the described 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.

## 2.1. Representation Setup

This section specifies a symbol representation that enables both a general form with coding coefficient vectors attached, and a compact form where a seed is attached which is used to generate one or multiple coding coefficient vectors. Different maximum GENERATION and WINDOW SIZE are supported for RLNC encoding, recoding, and decoding.

To encode over a set of data symbols, a coding coefficient vector is first generated, comprising a number of finite field elements as specified by a GENERATION SIZE or WINDOW SIZE variable. For a generation based code the GENERATION SIZE defines the number of original symbols in each generation. For a window based code the WINDOW SIZE specifies the maximal number of symbols in the window over which coding can be performed. In the case of systematic codes, systematic symbols correspond to unit coding coefficient vectors.

Figure 1 illustrates the general symbol representation design. Four header fields precede the symbol data: TYPE flag (T), SYMBOLS, ENCODER RANK, and SEED or CODING COEFFICIENTS. The TYPE Flag (T) indicates if the symbol is systematic, coded, or recoded. SYMBOLS indicates the number of symbols in the SYMBOLS(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. The CODING COEFFICIENTS field is a list of SYMBOLS number of coding coefficient vectors used to generate the SYMBOL(S) DATA, and used in the case where no random number generator is available or practical

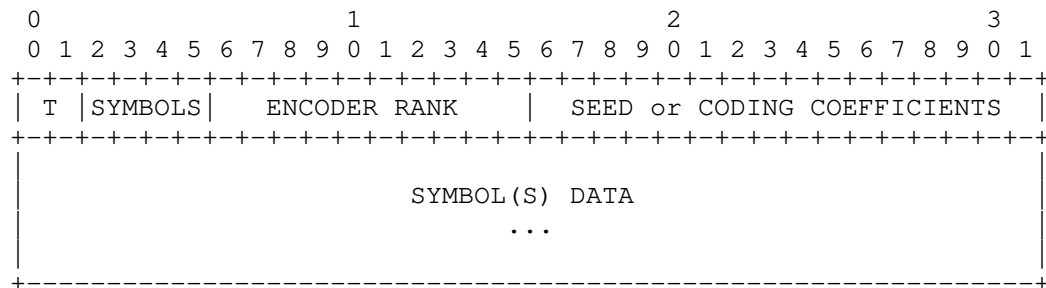


Figure 1: A general symbol representation design.

## 2.2. Field Types and Formats

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

- o 2 bits long.
- o 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.
- o If the TYPE is '2', all symbols included in this symbol representation are coded, with coding coefficient vectors generated using the included SEED and the ENCODER RANK. Consequently, only the first ENCODER RANK elements in the coding coefficient vector can be non-zero, whereas the remaining elements (e.g. GENERATION SIZE - ENCODER RANK) in the coding coefficient vector are zeros. This option allows for compact and efficient representation of coded symbols, which may also subsequently be recoded.
- o If the TYPE is '3', all symbols included in this symbol representation are either uncoded, coded or recoded. Each coding vector included is composed of GENERATION SIZE or WINDOW SIZE coefficients.

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

- o 4 bits long. A maximum number of 15 symbols are concatenated within each symbol representation.
- o The special case of SYMBOLS = 0 indicates that zero symbols are included, and consequently the size of SYMBOLS(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, that is, the number of original symbols used to compute the coded symbols(s). It has the following properties:

- o MUST be no larger than GENERATION/WINDOW SIZE.
- o If TYPE flag is '1', ENCODER RANK is the symbol index of the first data symbol in this symbol representation.

- o 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.

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:

- o The SEED field is only present when TYPE flag is '2'. If TYPE is '1' or '3', this field is absent.
- o 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 the coding coefficient 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 coefficient 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 coefficient vector are zero.
- o 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.
- o 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 coefficient 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 coefficient 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  $(2^8) * (2^4 - 1) * (2^{10}) \sim 2^{22}$ , 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 coefficient vectors is included can be sent instead.

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:

- o The CODING COEFFICIENT field is only present when TYPE flag is '3'. If TYPE is '1' or '2', this field is absent.
- o Each coding coefficient vector includes ENCODER RANK number of coding coefficients.

### 2.3. Externally Specified Parameters Required

This section specifies parameters that are REQUIRED for the use of this symbol representation but which are not included in the symbol representation and therefore MUST be communicated by means of some outer mechanism. Typically these parameters will be static throughout a protocol session. Consequently, there is little to gain by incorporating these parameters into the representation but conversely it would add additional overhead.

- o Field polynomial, the underlying field over which coding is performed.
- o Pseudo Random Generator, used to generate coding coefficient vectors.
- o Symbol Size, used to divide the original data into symbols.
- o GENERATION SIZE or WINDOW SIZE, for block and sliding window codes, respectively.
- o Small or large encoding window, this symbol representation supports both a small and a large coding window, but the variant used is not communicated.

### 2.4. 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 2 to 4 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 comprise a seed for coding coefficient generation. Recoded symbols are general in form and comprise the coding coefficient vectors explicitly.

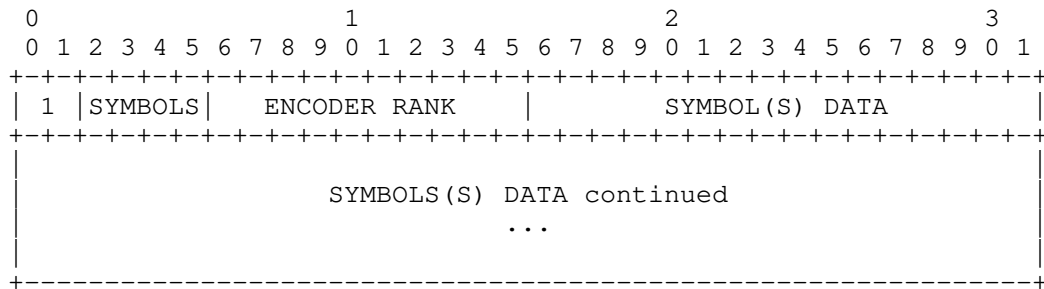


Figure 2: A systematic symbol representation.

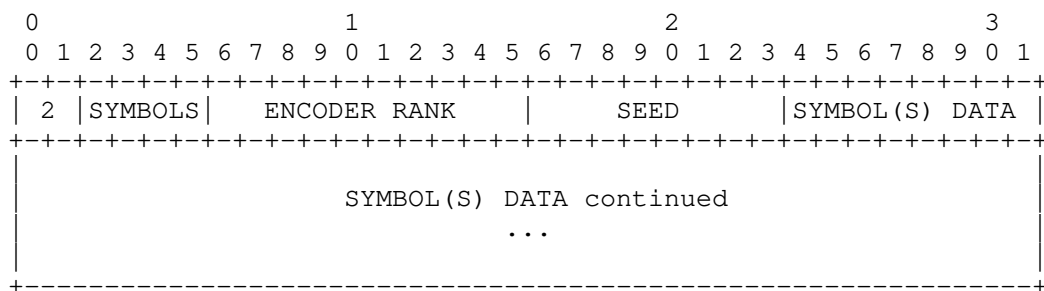


Figure 3: A compact, coded symbol representation.

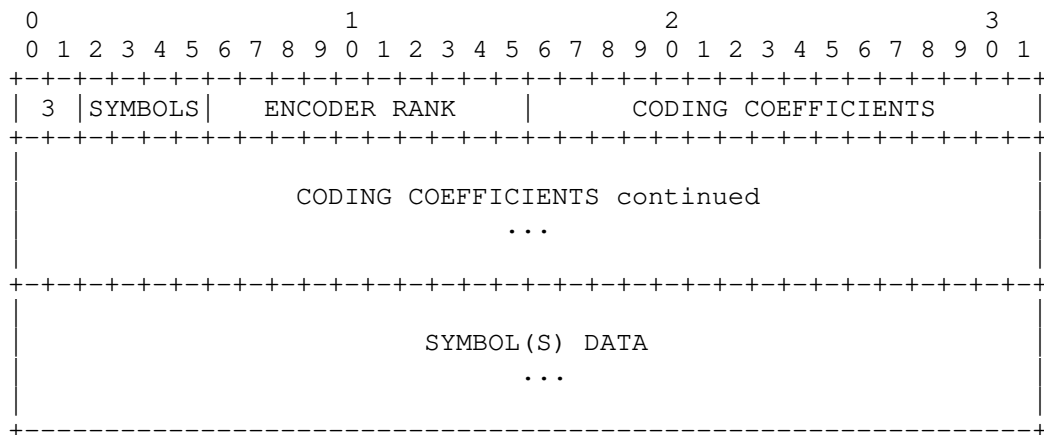


Figure 4: A recoded symbol representation.





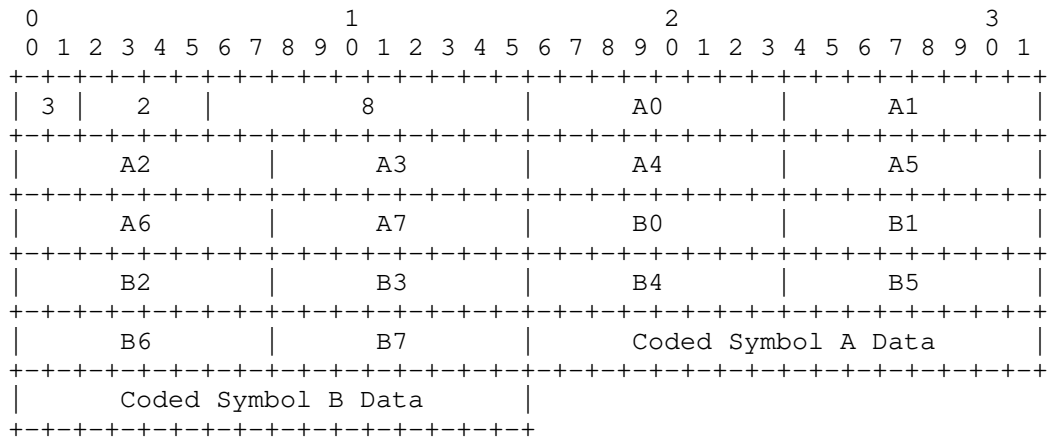


Figure 7: A symbol representation with 2 recoded symbols having coding coefficients attached.

## 2.5. 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 8 to 10 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 comprise a seed for coding coefficient generation. Recoded symbols are general in form and comprise the coding coefficient vectors explicitly (CODING COEFFICIENTS or CODING COEFFS).

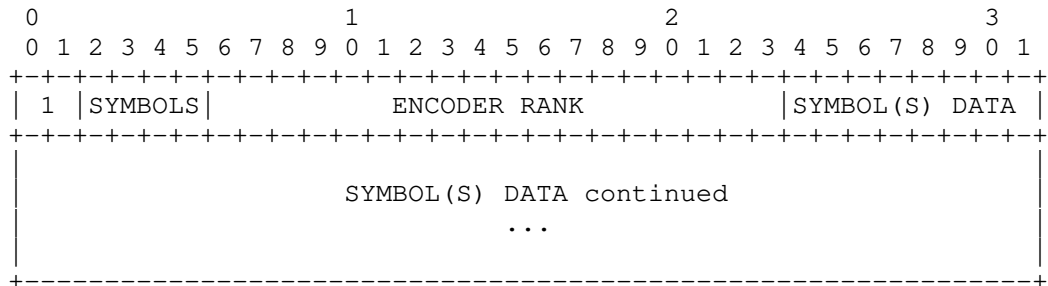


Figure 8: A systematic symbol representation.

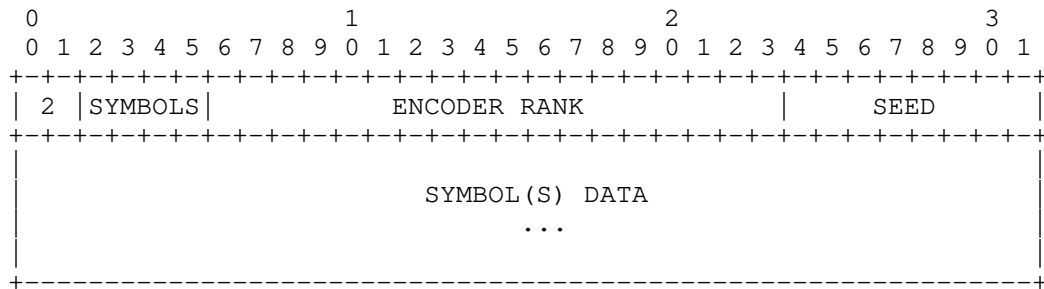


Figure 9: A coded symbol representation.

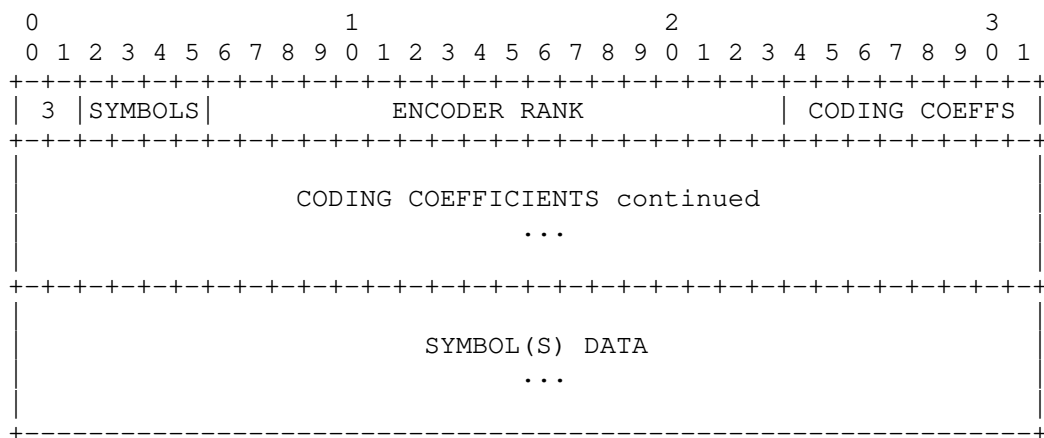


Figure 10: A recoded symbol representation.

### 3. Security Considerations

This document does not present new security considerations.

### 4. IANA Considerations

This document has no actions for IANA.

### 5. References

#### 5.1. 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>>.

[RLNC-Background]

Heide, J., Shi, S., Fouli, K., Medard, M., and V. Chook,  
"Random Linear Network Coding (RLNC): Background and  
Practical Considerations", February 2018,  
<[https://datatracker.ietf.org/doc/  
draft-heide-nwcrg-rlnc-background/](https://datatracker.ietf.org/doc/draft-heide-nwcrg-rlnc-background/)>.

## 5.2. 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/>>.

## Authors' Addresses

Janus Heide  
Steinwurf Aps  
Aalborg  
Denmark

Email: [janus@steinwurf.com](mailto:janus@steinwurf.com)

Shirley Shi  
Code On Network Coding LLC  
Cambridge  
USA

Email: [xshi@alum.mit.edu](mailto:xshi@alum.mit.edu)

Kerim Fouli  
Code On Network Coding LLC  
Cambridge  
USA

Email: [fouli@codeontechnologies.com](mailto:fouli@codeontechnologies.com)

Muriel Medard  
Code On Network Coding LLC  
Cambridge  
USA

Email: [muriel.medard@codeontechnologies.com](mailto:muriel.medard@codeontechnologies.com)

Vince Chook  
Inmarsat PLC  
London  
United Kingdom

Email: Vince.Chook@inmarsat.com

TSVWG  
Internet-Draft  
Updates: 6363 (if approved)  
Intended status: Standards Track  
Expires: July 15, 2019

V. Roca  
INRIA  
A. Begen  
Networked Media  
January 11, 2019

Forward Error Correction (FEC) Framework Extension to Sliding Window  
Codes  
draft-ietf-tsvwg-fecframe-ext-08

Abstract

RFC 6363 describes a framework for using Forward Error Correction (FEC) codes to provide protection against packet loss. The framework supports applying FEC to arbitrary packet flows over unreliable transport and is primarily intended for real-time, or streaming, media. However, FECFRAME as per RFC 6363 is restricted to block FEC codes. This document updates RFC 6363 to support FEC Codes based on a sliding encoding window, in addition to Block FEC Codes, in a backward-compatible way. During multicast/broadcast real-time content delivery, the use of sliding window codes significantly improves robustness in harsh environments, with less repair traffic and lower FEC-related added latency.

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 July 15, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
2. Definitions and Abbreviations . . . . .	4
3. Summary of Architecture Overview . . . . .	7
4. Procedural Overview . . . . .	10
4.1. General . . . . .	10
4.2. Sender Operation with Sliding Window FEC Codes . . . . .	10
4.3. Receiver Operation with Sliding Window FEC Codes . . . . .	13
5. Protocol Specification . . . . .	15
5.1. General . . . . .	15
5.2. FEC Framework Configuration Information . . . . .	16
5.3. FEC Scheme Requirements . . . . .	16
6. Feedback . . . . .	16
7. Transport Protocols . . . . .	17
8. Congestion Control . . . . .	17
9. Implementation Status . . . . .	17
10. Security Considerations . . . . .	17
11. Operations and Management Considerations . . . . .	18
12. IANA Considerations . . . . .	18
13. Acknowledgments . . . . .	18
14. References . . . . .	18
14.1. Normative References . . . . .	18
14.2. Informative References . . . . .	19
Appendix A. About Sliding Encoding Window Management (informational) . . . . .	20
Authors' Addresses . . . . .	21

## 1. Introduction

Many applications need to transport a continuous stream of packetized data from a source (sender) to one or more destinations (receivers) over networks that do not provide guaranteed packet delivery. In particular packets may be lost, which is strictly the focus of this document: we assume that transmitted packets are either lost (e.g., because of a congested router, of a poor signal-to-noise ratio in a wireless network, or because the number of bit errors exceeds the correction capabilities of the physical-layer error correcting code)

or received by the transport protocol without any corruption (i.e., the bit-errors, if any, have been fixed by the physical-layer error correcting code and therefore are hidden to the upper layers).

For these use-cases, Forward Error Correction (FEC) applied within the transport or application layer is an efficient technique to improve packet transmission robustness in presence of packet losses (or "erasures"), without going through packet retransmissions that create a delay often incompatible with real-time constraints. The FEC Building Block defined in [RFC5052] provides a framework for the definition of Content Delivery Protocols (CDPs) that make use of separately-defined FEC schemes. Any CDP defined according to the requirements of the FEC Building Block can then easily be used with any FEC Scheme that is also defined according to the requirements of the FEC Building Block.

Then FECFRAME [RFC6363] provides a framework to define Content Delivery Protocols (CDPs) that provide FEC protection for arbitrary packet flows over an unreliable datagram service transport such as UDP. It is primarily intended for real-time or streaming media applications, using broadcast, multicast, or on-demand delivery.

However, [RFC6363] only considers block FEC schemes defined in accordance with the FEC Building Block [RFC5052] (e.g., [RFC6681], [RFC6816] or [RFC6865]). These codes require the input flow(s) to be segmented into a sequence of blocks. Then FEC encoding (at a sender or an encoding middlebox) and decoding (at a receiver or a decoding middlebox) are both performed on a per-block basis. For instance, if the current block encompasses the 100's to 119's source symbols (i.e., a block of size 20 symbols) of an input flow, encoding (and decoding) will be performed on this block independently of other blocks. This approach has major impacts on FEC encoding and decoding delays. The data packets of continuous media flow(s) may be passed to the transport layer immediately, without delay. But the block creation time, that depends on the number of source symbols in this block, impacts both the FEC encoding delay (since encoding requires that all source symbols be known), and mechanically the packet loss recovery delay at a receiver (since no repair symbol for the current block can be generated and therefore received before that time). Therefore a good value for the block size is necessarily a balance between the maximum FEC decoding latency at the receivers (which must be in line with the most stringent real-time requirement of the protected flow(s), hence an incentive to reduce the block size), and the desired robustness against long loss bursts (which increases with the block size, hence an incentive to increase this size).

This document updates [RFC6363] in order to also support FEC codes based on a sliding encoding window (A.K.A. convolutional codes)



[RFC8406]. This encoding window, either of fixed or variable size, slides over the set of source symbols. FEC encoding is launched whenever needed, from the set of source symbols present in the sliding encoding window at that time. This approach significantly reduces FEC-related latency, since repair symbols can be generated and passed to the transport layer on-the-fly, at any time, and can be regularly received by receivers to quickly recover packet losses. Using sliding window FEC codes is therefore highly beneficial to real-time flows, one of the primary targets of FECFRAME. [RLC-ID] provides an example of such FEC Scheme for FECFRAME, built upon the simple sliding window Random Linear Codes (RLC).

This document is fully backward compatible with [RFC6363]. Indeed:

- o this FECFRAME update does not prevent nor compromise in any way the support of block FEC codes. Both types of codes can nicely co-exist, just like different block FEC schemes can co-exist;
- o each sliding window FEC Scheme is associated to a specific FEC Encoding ID subject to IANA registration, just like block FEC Schemes;
- o any receiver, for instance a legacy receiver that only supports block FEC schemes, can easily identify the FEC Scheme used in a FECFRAME session. Indeed, the FEC Encoding ID that identifies the FEC Scheme is carried in the FEC Framework Configuration Information (see section 5.5 of [RFC6363]). For instance, when the Session Description Protocol (SDP) is used to carry the FEC Framework Configuration Information, the FEC Encoding ID can be communicated in the "encoding-id=" parameter of a "fec-repair-flow" attribute [RFC6364]. This mechanism is the basic approach for a FECFRAME receiver to determine whether or not it supports the FEC Scheme used in a given FECFRAME session;

This document leverages on [RFC6363] and re-uses its structure. It proposes new sections specific to sliding window FEC codes whenever required. The only exception is Section 3 that provides a quick summary of FECFRAME in order to facilitate the understanding of this document to readers not familiar with the concepts and terminology.

## 2. Definitions and Abbreviations

The following list of definitions and abbreviations is copied from [RFC6363], adding only the Block/sliding window FEC Code and Encoding/Decoding Window definitions (tagged with "ADDED"):

**Application Data Unit (ADU):** The unit of source data provided as payload to the transport layer. For instance, it can be a

payload containing the result of the RTP packetization of a compressed video frame.

**ADU Flow:** A sequence of ADUs associated with a transport-layer flow identifier (such as the standard 5-tuple {source IP address, source port, destination IP address, destination port, transport protocol}).

**AL-FEC:** Application-layer Forward Error Correction.

**Application Protocol:** Control protocol used to establish and control the source flow being protected, e.g., the Real-Time Streaming Protocol (RTSP).

**Content Delivery Protocol (CDP):** A complete application protocol specification that, through the use of the framework defined in this document, is able to make use of FEC schemes to provide FEC capabilities.

**FEC Code:** An algorithm for encoding data such that the encoded data flow is resilient to data loss. Note that, in general, FEC codes may also be used to make a data flow resilient to corruption, but that is not considered in this document.

**Block FEC Code: (ADDED)** An FEC Code that operates on blocks, i.e., for which the input flow MUST be segmented into a sequence of blocks, FEC encoding and decoding being performed independently on a per-block basis.

**Sliding Window FEC Code: (ADDED)** An FEC Code that can generate repair symbols on-the-fly, at any time, from the set of source symbols present in the sliding encoding window at that time. These codes are also known as convolutional codes.

**FEC Framework:** A protocol framework for the definition of Content Delivery Protocols using FEC, such as the framework defined in this document.

**FEC Framework Configuration Information:** Information that controls the operation of the FEC Framework.

**FEC Payload ID:** Information that identifies the contents and provides positional information of a packet with respect to the FEC Scheme.

**FEC Repair Packet:** At a sender (respectively, at a receiver), a payload submitted to (respectively, received from) the transport

protocol containing one or more repair symbols along with a Repair FEC Payload ID and possibly an RTP header.

**FEC Scheme:** A specification that defines the additional protocol aspects required to use a particular FEC code with the FEC Framework.

**FEC Source Packet:** At a sender (respectively, at a receiver), a payload submitted to (respectively, received from) the transport protocol containing an ADU along with an optional Explicit Source FEC Payload ID.

**Repair Flow:** The packet flow carrying FEC data.

**Repair FEC Payload ID:** A FEC Payload ID specifically for use with repair packets.

**Source Flow:** The packet flow to which FEC protection is to be applied. A source flow consists of ADUs.

**Source FEC Payload ID:** A FEC Payload ID specifically for use with source packets.

**Source Protocol:** A protocol used for the source flow being protected, e.g., RTP.

**Transport Protocol:** The protocol used for the transport of the source and repair flows, using an unreliable datagram service such as UDP.

**Encoding Window:** (ADDED) Set of Source Symbols available at the sender/coding node that are used to generate a repair symbol, with a Sliding Window FEC Code.

**Decoding Window:** (ADDED) Set of received or decoded source and repair symbols available at a receiver that are used to decode erased source symbols, with a Sliding Window FEC Code.

**Code Rate:** The ratio between the number of source symbols and the number of encoding symbols. By definition, the code rate is such that  $0 < \text{code rate} \leq 1$ . A code rate close to 1 indicates that a small number of repair symbols have been produced during the encoding process.

**Encoding Symbol:** Unit of data generated by the encoding process. With systematic codes, source symbols are part of the encoding symbols.

**Packet Erasure Channel:** A communication path where packets are either lost (e.g., in our case, by a congested router, or because the number of transmission errors exceeds the correction capabilities of the physical-layer code) or received. When a packet is received, it is assumed that this packet is not corrupted (i.e., in our case, the bit-errors, if any, are fixed by the physical-layer code and therefore hidden to the upper layers).

**Repair Symbol:** Encoding symbol that is not a source symbol.

**Source Block:** Group of ADUs that are to be FEC protected as a single block. This notion is restricted to Block FEC Codes.

**Source Symbol:** Unit of data used during the encoding process.

**Systematic Code:** FEC code in which the source symbols are part of the encoding symbols.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

### 3. Summary of Architecture Overview

The architecture of [RFC6363], Section 3, equally applies to this FECFRAME extension and is not repeated here. However, we provide hereafter a quick summary to facilitate the understanding of this document to readers not familiar with the concepts and terminology.

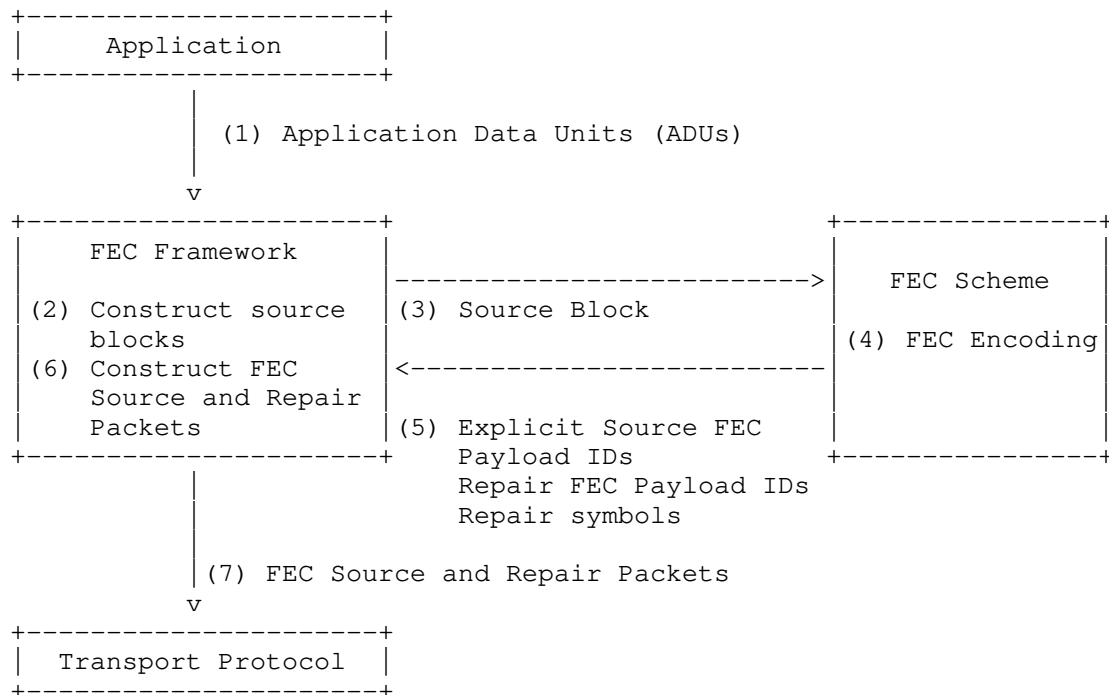


Figure 1: FECFRAME architecture at a sender.

The FECFRAME architecture is illustrated in Figure 1 from the sender's point of view, in case of a block FEC Scheme. It shows an application generating an ADU flow (other flows, from other applications, may co-exist). These ADUs, of variable size, must be somehow mapped to source symbols of fixed size (this fixed size is a requirement of all FEC Schemes that comes from the way mathematical operations are applied to symbols content). This is the goal of an ADU-to-symbols mapping process that is FEC-Scheme specific (see below). Once the source block is built, taking into account both the FEC Scheme constraints (e.g., in terms of maximum source block size) and the application's flow constraints (e.g., in terms of real-time constraints), the associated source symbols are handed to the FEC Scheme in order to produce an appropriate number of repair symbols. FEC Source Packets (containing ADUs) and FEC Repair Packets (containing one or more repair symbols each) are then generated and sent using an appropriate transport protocol (more precisely [RFC6363], Section 7, requires a transport protocol providing an unreliable datagram service, such as UDP). In practice FEC Source Packets may be passed to the transport layer as soon as available, without having to wait for FEC encoding to take place. In that case

a copy of the associated source symbols needs to be kept within FECFRAME for future FEC encoding purposes.

At a receiver (not shown), FECFRAME processing operates in a similar way, taking as input the incoming FEC Source and Repair Packets received. In case of FEC Source Packet losses, the FEC decoding of the associated block may recover all (in case of successful decoding) or a subset potentially empty (otherwise) of the missing source symbols. After source-symbol-to-ADU mapping, when lost ADUs are recovered, they are then assigned to their respective flow (see below). ADUs are returned to the application(s), either in their initial transmission order (in that case ADUs received after an erased one will be delayed until FEC decoding has taken place) or not (in that case each ADU is returned as soon as it is received or recovered), depending on the application requirements.

FECFRAME features two subtle mechanisms:

- o ADUs-to-source-symbols mapping: in order to manage variable size ADUs, FECFRAME and FEC Schemes can use small, fixed size symbols and create a mapping between ADUs and symbols. To each ADU this mechanism prepends a length field (plus a flow identifier, see below) and pads the result to a multiple of the symbol size. A small ADU may be mapped to a single source symbol while a large one may be mapped to multiple symbols. The mapping details are FEC-Scheme-dependent and must be defined in the associated document;
- o Assignment of decoded ADUs to flows in multi-flow configurations: when multiple flows are multiplexed over the same FECFRAME instance, a problem is to assign a decoded ADU to the right flow (UDP port numbers and IP addresses traditionally used to map incoming ADUs to flows are not recovered during FEC decoding). To make it possible, at the FECFRAME sending instance, each ADU is prepended with a flow identifier (1 byte) during the ADU-to-source-symbols mapping (see above). The flow identifiers are also shared between all FECFRAME instances as part of the FEC Framework Configuration Information. This (flow identifier + length + application payload + padding), called ADUI, is then FEC protected. Therefore a decoded ADUI contains enough information to assign the ADU to the right flow.

A few aspects are not covered by FECFRAME, namely:

- o [RFC6363] section 8 does not detail any congestion control mechanism, but only provides high level normative requirements;

- o the possibility of having feedbacks from receiver(s) is considered out of scope, although such a mechanism may exist within the application (e.g., through RTCP control messages);
- o flow adaptation at a FECFRAME sender (e.g., how to set the FEC code rate based on transmission conditions) is not detailed, but it needs to comply with the congestion control normative requirements (see above).

#### 4. Procedural Overview

##### 4.1. General

The general considerations of [RFC6363], Section 4.1, that are specific to block FEC codes are not repeated here.

With a Sliding Window FEC Code, the FEC Source Packet MUST contain information to identify the position occupied by the ADU within the source flow, in terms specific to the FEC Scheme. This information is known as the Source FEC Payload ID, and the FEC Scheme is responsible for defining and interpreting it.

With a Sliding Window FEC Code, the FEC Repair Packets MUST contain information that identifies the relationship between the contained repair payloads and the original source symbols used during encoding. This information is known as the Repair FEC Payload ID, and the FEC Scheme is responsible for defining and interpreting it.

The Sender Operation ([RFC6363], Section 4.2.) and Receiver Operation ([RFC6363], Section 4.3) are both specific to block FEC codes and therefore omitted below. The following two sections detail similar operations for Sliding Window FEC codes.

##### 4.2. Sender Operation with Sliding Window FEC Codes

With a Sliding Window FEC Scheme, the following operations, illustrated in Figure 2 for the generic case (non-RTP repair flows), and in Figure 3 for the case of RTP repair flows, describe a possible way to generate compliant source and repair flows:

1. A new ADU is provided by the application.
2. The FEC Framework communicates this ADU to the FEC Scheme.
3. The sliding encoding window is updated by the FEC Scheme. The ADU-to-source-symbols mapping as well as the encoding window management details are both the responsibility of the FEC Scheme

and MUST be detailed there. Appendix A provides non-normative hints about what FEC Scheme designers need to consider;

4. The Source FEC Payload ID information of the source packet is determined by the FEC Scheme. If required by the FEC Scheme, the Source FEC Payload ID is encoded into the Explicit Source FEC Payload ID field and returned to the FEC Framework.
5. The FEC Framework constructs the FEC Source Packet according to [RFC6363] Figure 6, using the Explicit Source FEC Payload ID provided by the FEC Scheme if applicable.
6. The FEC Source Packet is sent using normal transport-layer procedures. This packet is sent using the same ADU flow identification information as would have been used for the original source packet if the FEC Framework were not present (e.g., the source and destination addresses and UDP port numbers on the IP datagram carrying the source packet will be the same whether or not the FEC Framework is applied).
7. When the FEC Framework needs to send one or several FEC Repair Packets (e.g., according to the target Code Rate), it asks the FEC Scheme to create one or several repair packet payloads from the current sliding encoding window along with their Repair FEC Payload ID.
8. The Repair FEC Payload IDs and repair packet payloads are provided back by the FEC Scheme to the FEC Framework.
9. The FEC Framework constructs FEC Repair Packets according to [RFC6363] Figure 7, using the FEC Payload IDs and repair packet payloads provided by the FEC Scheme.
10. The FEC Repair Packets are sent using normal transport-layer procedures. The port(s) and multicast group(s) to be used for FEC Repair Packets are defined in the FEC Framework Configuration Information.



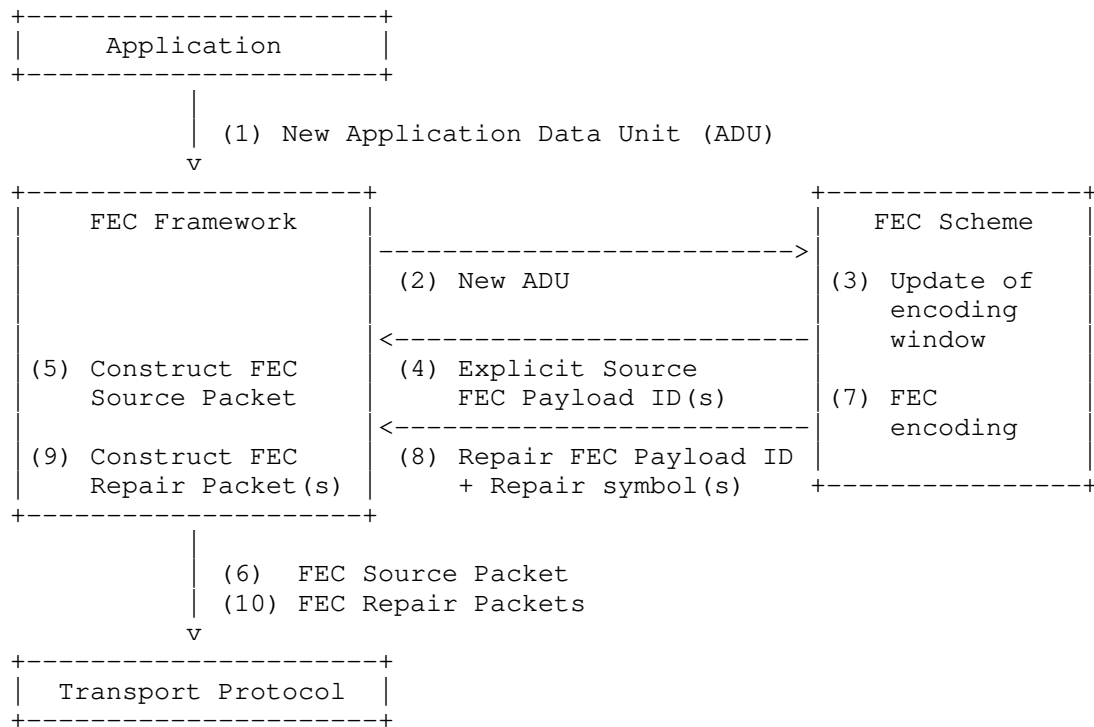


Figure 2: Sender Operation with Sliding Window FEC Codes

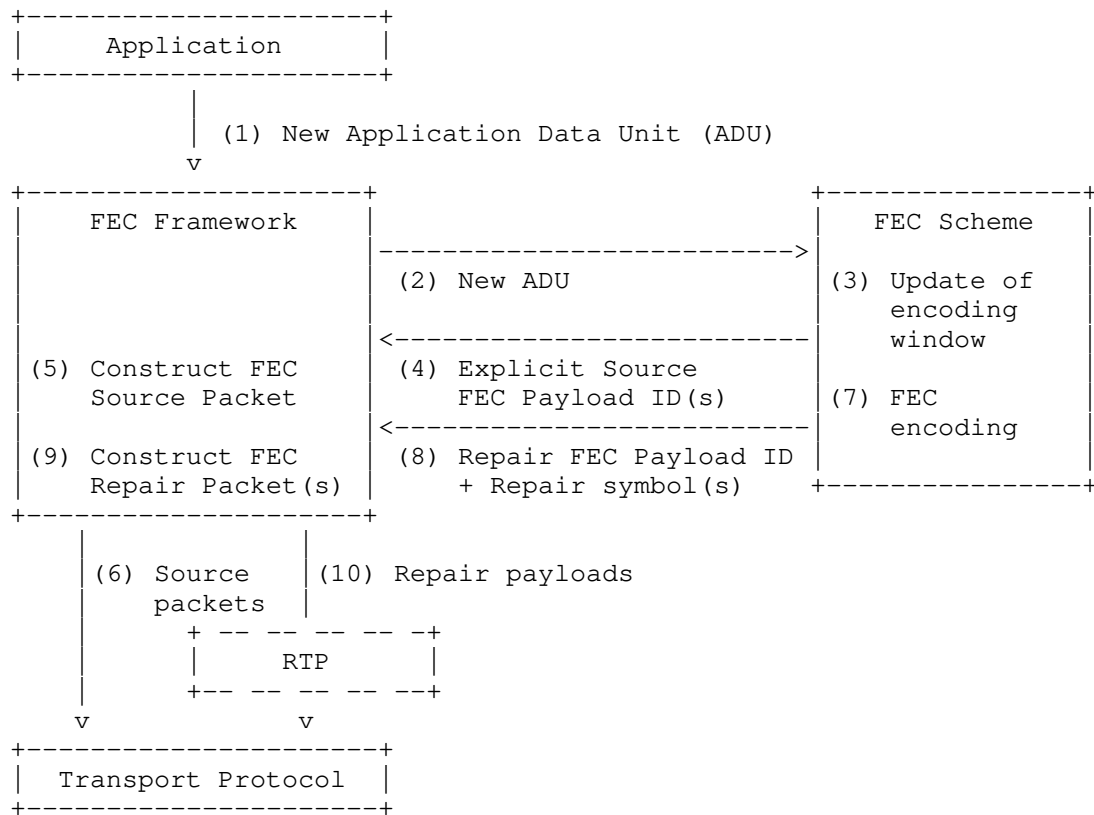


Figure 3: Sender Operation with Sliding Window FEC Codes and RTP Repair Flows

#### 4.3. Receiver Operation with Sliding Window FEC Codes

With a Sliding Window FEC Scheme, the following operations, illustrated in Figure 4 for the generic case (non-RTP repair flows), and in Figure 5 for the case of RTP repair flows. The only differences with respect to block FEC codes lie in steps (4) and (5). Therefore this section does not repeat the other steps of [RFC6363], Section 4.3, "Receiver Operation". The new steps (4) and (5) are:

4. The FEC Scheme uses the received FEC Payload IDs (and derived FEC Source Payload IDs when the Explicit Source FEC Payload ID field is not used) to insert source and repair packets into the decoding window in the right way. If at least one source packet is missing and at least one repair packet has been received, then FEC decoding is attempted to recover missing source payloads. The FEC Scheme determines whether source packets have been lost

and whether enough repair packets have been received to decode any or all of the missing source payloads.

5. The FEC Scheme returns the received and decoded ADUs to the FEC Framework, along with indications of any ADUs that were missing and could not be decoded.

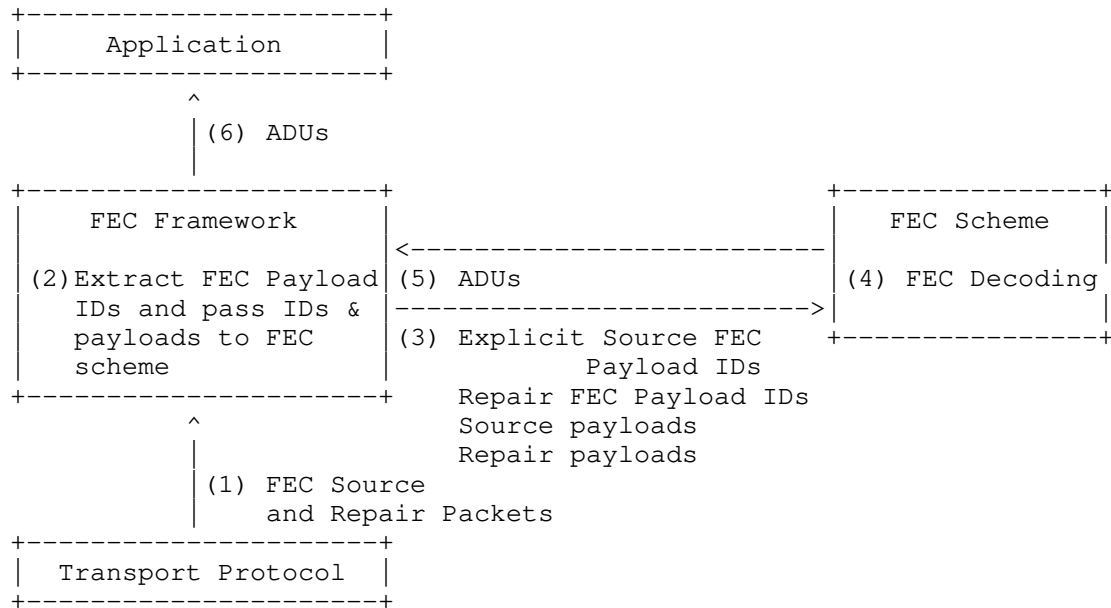


Figure 4: Receiver Operation with Sliding Window FEC Codes

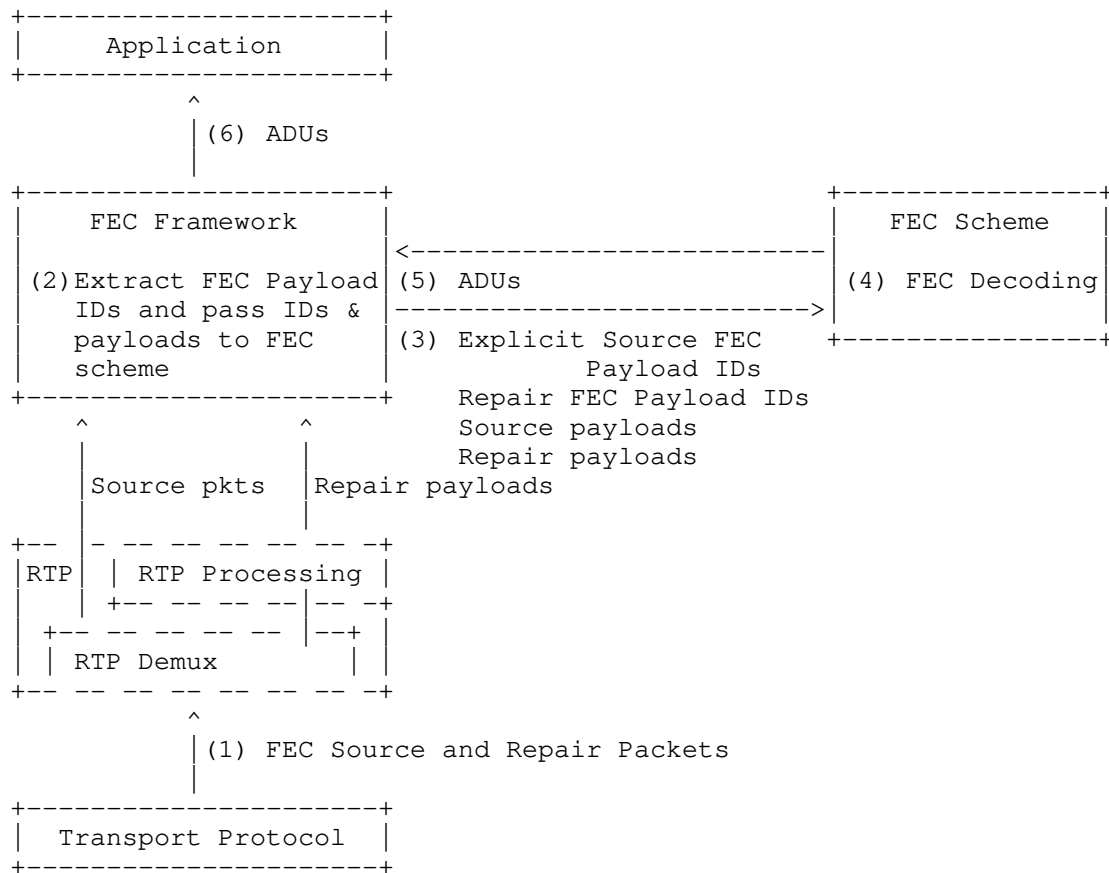


Figure 5: Receiver Operation with Sliding Window FEC Codes and RTP Repair Flows

## 5. Protocol Specification

### 5.1. General

This section discusses the protocol elements for the FEC Framework specific to Sliding Window FEC schemes. The global formats of source data packets (i.e., [RFC6363], Figure 6) and repair data packets (i.e., [RFC6363], Figures 7 and 8) remain the same with Sliding Window FEC codes. They are not repeated here.

## 5.2. FEC Framework Configuration Information

The FEC Framework Configuration Information considerations of [RFC6363], Section 5.5, equally applies to this FECFRAME extension and is not repeated here.

## 5.3. FEC Scheme Requirements

The FEC Scheme requirements of [RFC6363], Section 5.6, mostly apply to this FECFRAME extension and are not repeated here. An exception though is the "full specification of the FEC code", item (4), that is specific to block FEC codes. The following item (4-bis) applies in case of Sliding Window FEC schemes:

### 4-bis. A full specification of the Sliding Window FEC code

This specification MUST precisely define the valid FEC-Scheme-Specific Information values, the valid FEC Payload ID values, and the valid packet payload sizes (where packet payload refers to the space within a packet dedicated to carrying encoding symbols).

Furthermore, given valid values of the FEC-Scheme-Specific Information, a valid Repair FEC Payload ID value, a valid packet payload size, and a valid encoding window (i.e., a set of source symbols), the specification MUST uniquely define the values of the encoding symbol (or symbols) to be included in the repair packet payload with the given Repair FEC Payload ID value.

Additionally, the FEC Scheme associated to a Sliding Window FEC Code:

- o MUST define the relationships between ADUs and the associated source symbols (mapping);
- o MUST define the management of the encoding window that slides over the set of ADUs. Appendix A provides non normative hints about what FEC Scheme designers need to consider;
- o MUST define the management of the decoding window. This usually consists in managing a system of linear equations (in case of a linear FEC code);

## 6. Feedback

The discussion of [RFC6363], Section 6, equally applies to this FECFRAME extension and is not repeated here.

## 7. Transport Protocols

The discussion of [RFC6363], Section 7, equally applies to this FECFRAME extension and is not repeated here.

## 8. Congestion Control

The discussion of [RFC6363], Section 8, equally applies to this FECFRAME extension and is not repeated here.

## 9. Implementation Status

Editor's notes: RFC Editor, please remove this section motivated by RFC 7942 before publishing the RFC. Thanks!

An implementation of FECFRAME extended to Sliding Window codes exists:

- o Organisation: Inria
- o Description: This is an implementation of FECFRAME extended to Sliding Window codes and supporting the RLC FEC Scheme [RLC-ID]. It is based on: (1) a proprietary implementation of FECFRAME, made by Inria and Expway for which interoperability tests have been conducted; and (2) a proprietary implementation of RLC Sliding Window FEC Codes.
- o Maturity: the basic FECFRAME maturity is "production", the FECFRAME extension maturity is "under progress".
- o Coverage: the software implements a subset of [RFC6363], as specialized by the 3GPP eMBMS standard [MBMSTS]. This software also covers the additional features of FECFRAME extended to Sliding Window codes, in particular the RLC FEC Scheme.
- o Licensing: proprietary.
- o Implementation experience: maximum.
- o Information update date: March 2018.
- o Contact: vincent.roca@inria.fr

## 10. Security Considerations

This FECFRAME extension does not add any new security consideration. All the considerations of [RFC6363], Section 9, apply to this document as well. However, for the sake of completeness, the

following goal can be added to the list provided in Section 9.1 "Problem Statement" of [RFC6363]:

- o Attacks can try to corrupt source flows in order to modify the receiver application's behavior (as opposed to just denying service).

## 11. Operations and Management Considerations

This FECFRAME extension does not add any new Operations and Management Consideration. All the considerations of [RFC6363], Section 10, apply to this document as well.

## 12. IANA Considerations

No IANA actions are required for this document.

A FEC Scheme for use with this FEC Framework is identified via its FEC Encoding ID. It is subject to IANA registration in the "FEC Framework (FECFRAME) FEC Encoding IDs" registry. All the rules of [RFC6363], Section 11, apply and are not repeated here.

## 13. Acknowledgments

The authors would like to thank Christer Holmberg, David Black, Gorrry Fairhurst, and Emmanuel Lochin, Spencer Dawkins, Ben Campbell, Benjamin Kaduk, Eric Rescorla, Adam Roach, and Greg Skinner for their valuable feedback on this document. This document being an extension to [RFC6363], the authors would also like to thank Mark Watson as the main author of that RFC.

## 14. References

### 14.1. 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>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

## 14.2. Informative References

- [MBMSTS] 3GPP, "Multimedia Broadcast/Multicast Service (MBMS); Protocols and codecs", 3GPP TS 26.346, March 2009, <<http://ftp.3gpp.org/specs/html-info/26346.htm>>.
- [RFC5052] Watson, M., Luby, M., and L. Vicisano, "Forward Error Correction (FEC) Building Block", RFC 5052, DOI 10.17487/RFC5052, August 2007, <<https://www.rfc-editor.org/info/rfc5052>>.
- [RFC6364] Begen, A., "Session Description Protocol Elements for the Forward Error Correction (FEC) Framework", RFC 6364, DOI 10.17487/RFC6364, October 2011, <<https://www.rfc-editor.org/info/rfc6364>>.
- [RFC6681] Watson, M., Stockhammer, T., and M. Luby, "Raptor Forward Error Correction (FEC) Schemes for FECFRAME", RFC 6681, DOI 10.17487/RFC6681, August 2012, <<https://www.rfc-editor.org/info/rfc6681>>.
- [RFC6816] Roca, V., Cunche, M., and J. Lacan, "Simple Low-Density Parity Check (LDPC) Staircase Forward Error Correction (FEC) Scheme for FECFRAME", RFC 6816, DOI 10.17487/RFC6816, December 2012, <<https://www.rfc-editor.org/info/rfc6816>>.
- [RFC6865] Roca, V., Cunche, M., Lacan, J., Bouabdallah, A., and K. Matsuzono, "Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME", RFC 6865, DOI 10.17487/RFC6865, February 2013, <<https://www.rfc-editor.org/info/rfc6865>>.
- [RFC8406] Adamson, B., Adjih, C., Bilbao, J., Firoiu, V., Fitzek, F., Ghanem, S., Lochin, E., Masucci, A., Montpetit, M-J., Pedersen, M., Peralta, G., Roca, V., Ed., Saxena, P., and S. Sivakumar, "Taxonomy of Coding Techniques for Efficient Network Communications", RFC 8406, DOI 10.17487/RFC8406, June 2018, <<https://www.rfc-editor.org/info/rfc8406>>.
- [RLC-ID] Roca, V. and B. Teibi, "Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Scheme for FECFRAME", Work in Progress, Transport Area Working Group (TSVWG) draft-ietf-tsvwg-rlc-fec-scheme (Work in Progress), September 2018, <<https://tools.ietf.org/html/draft-ietf-tsvwg-rlc-fec-scheme>>.



## Appendix A. About Sliding Encoding Window Management (informational)

The FEC Framework does not specify the management of the sliding encoding window which is the responsibility of the FEC Scheme. This annex only provides a few informational hints.

Source symbols are added to the sliding encoding window each time a new ADU is available at the sender, after the ADU-to-source-symbol mapping specific to the FEC Scheme.

Source symbols are removed from the sliding encoding window, for instance:

- o after a certain delay, when an "old" ADU of a real-time flow times out. The source symbol retention delay in the sliding encoding window should therefore be initialized according to the real-time features of incoming flow(s) when applicable;
- o once the sliding encoding window has reached its maximum size (there is usually an upper limit to the sliding encoding window size). In that case the oldest symbol is removed each time a new source symbol is added.

Several considerations can impact the management of this sliding encoding window:

- o at the source flows level: real-time constraints can limit the total time source symbols can remain in the encoding window;
- o at the FEC code level: theoretical or practical limitations (e.g., because of computational complexity) can limit the number of source symbols in the encoding window;
- o at the FEC Scheme level: signaling and window management are intrinsically related. For instance, an encoding window composed of a non-sequential set of source symbols requires an appropriate signaling to inform a receiver of the composition of the encoding window, and the associated transmission overhead can limit the maximum encoding window size. On the opposite, an encoding window always composed of a sequential set of source symbols simplifies signaling: providing the identity of the first source symbol plus their number is sufficient, which creates a fixed and relatively small transmission overhead.

Authors' Addresses

Vincent Roca  
INRIA  
Univ. Grenoble Alpes  
France

EMail: [vincent.roca@inria.fr](mailto:vincent.roca@inria.fr)

Ali Begen  
Networked Media  
Konya  
Turkey

EMail: [ali.begen@networked.media](mailto:ali.begen@networked.media)

TSVWG  
Internet-Draft  
Intended status: Standards Track  
Expires: December 20, 2019

V. Roca  
B. Teibi  
INRIA  
June 18, 2019

Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC)  
Schemes for FECFRAME  
draft-ietf-tsvwg-rlc-fec-scheme-16

Abstract

This document describes two fully-specified Forward Erasure Correction (FEC) Schemes for Sliding Window Random Linear Codes (RLC), one for RLC over the Galois Field (A.K.A. Finite Field)  $GF(2)$ , a second one for RLC over the Galois Field  $GF(2^{8})$ , each time with the possibility of controlling the code density. They can protect arbitrary media streams along the lines defined by FECFRAME extended to sliding window FEC codes. These sliding window FEC codes rely on an encoding window that slides over the source symbols, generating new repair symbols whenever needed. Compared to block FEC codes, these sliding window FEC codes offer key advantages with real-time flows in terms of reduced FEC-related latency while often providing improved packet erasure recovery capabilities.

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 December 20, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1.	Introduction . . . . .	3
1.1.	Limits of Block Codes with Real-Time Flows . . . . .	4
1.2.	Lower Latency and Better Protection of Real-Time Flows with the Sliding Window RLC Codes . . . . .	4
1.3.	Small Transmission Overheads with the Sliding Window RLC FEC Scheme . . . . .	5
1.4.	Document Organization . . . . .	6
2.	Definitions and Abbreviations . . . . .	6
3.	Common Procedures . . . . .	7
3.1.	Codec Parameters . . . . .	7
3.2.	ADU, ADUI and Source Symbols Mappings . . . . .	9
3.3.	Encoding Window Management . . . . .	10
3.4.	Source Symbol Identification . . . . .	11
3.5.	Pseudo-Random Number Generator (PRNG) . . . . .	11
3.6.	Coding Coefficients Generation Function . . . . .	13
3.7.	Finite Fields Operations . . . . .	15
3.7.1.	Finite Field Definitions . . . . .	15
3.7.2.	Linear Combination of Source Symbols Computation . . . . .	15
4.	Sliding Window RLC FEC Scheme over $GF(2^{^8})$ for Arbitrary Packet Flows . . . . .	16
4.1.	Formats and Codes . . . . .	16
4.1.1.	FEC Framework Configuration Information . . . . .	16
4.1.2.	Explicit Source FEC Payload ID . . . . .	18
4.1.3.	Repair FEC Payload ID . . . . .	18
4.2.	Procedures . . . . .	20
5.	Sliding Window RLC FEC Scheme over $GF(2)$ for Arbitrary Packet Flows . . . . .	20
5.1.	Formats and Codes . . . . .	20
5.1.1.	FEC Framework Configuration Information . . . . .	20
5.1.2.	Explicit Source FEC Payload ID . . . . .	20
5.1.3.	Repair FEC Payload ID . . . . .	20
5.2.	Procedures . . . . .	21
6.	FEC Code Specification . . . . .	21
6.1.	Encoding Side . . . . .	21
6.2.	Decoding Side . . . . .	22
7.	Implementation Status . . . . .	22

8.	Security Considerations . . . . .	23
8.1.	Attacks Against the Data Flow . . . . .	23
8.1.1.	Access to Confidential Content . . . . .	23
8.1.2.	Content Corruption . . . . .	23
8.2.	Attacks Against the FEC Parameters . . . . .	23
8.3.	When Several Source Flows are to be Protected Together .	25
8.4.	Baseline Secure FEC Framework Operation . . . . .	25
8.5.	Additional Security Considerations for Numerical Computations . . . . .	25
9.	Operations and Management Considerations . . . . .	26
9.1.	Operational Recommendations: Finite Field GF(2) Versus GF(2 <sup>8</sup> ) . . . . .	26
9.2.	Operational Recommendations: Coding Coefficients Density Threshold . . . . .	26
10.	IANA Considerations . . . . .	27
11.	Acknowledgments . . . . .	27
12.	References . . . . .	27
12.1.	Normative References . . . . .	27
12.2.	Informative References . . . . .	28
Appendix A.	TinyMT32 Validation Criteria (Normative) . . . . .	30
Appendix B.	Assessing the PRNG Adequacy (Informational) . . . . .	31
Appendix C.	Possible Parameter Derivation (Informational) . . . . .	33
C.1.	Case of a CBR Real-Time Flow . . . . .	34
C.2.	Other Types of Real-Time Flow . . . . .	36
C.3.	Case of a Non Real-Time Flow . . . . .	37
Appendix D.	Decoding Beyond Maximum Latency Optimization (Informational) . . . . .	37
Authors' Addresses	. . . . .	38

## 1. Introduction

Application-Level Forward Erasure Correction (AL-FEC) codes, or simply FEC codes, are a key element of communication systems. They are used to recover from packet losses (or erasures) during content delivery sessions to a potentially large number of receivers (multicast/broadcast transmissions). This is the case with the FLUTE/ALC protocol [RFC6726] when used for reliable file transfers over lossy networks, and the FECFRAME protocol [RFC6363] when used for reliable continuous media transfers over lossy networks.

The present document only focuses on the FECFRAME protocol, used in multicast/broadcast delivery mode, in particular for contents that feature stringent real-time constraints: each source packet has a maximum validity period after which it will not be considered by the destination application.

### 1.1. Limits of Block Codes with Real-Time Flows

With FECFRAME, there is a single FEC encoding point (either an end-host/server (source) or a middlebox) and a single FEC decoding point per receiver (either an end-host (receiver) or middlebox). In this context, currently standardized AL-FEC codes for FECFRAME like Reed-Solomon [RFC6865], LDPC-Staircase [RFC6816], or Raptor/RaptorQ [RFC6681], are all linear block codes: they require the data flow to be segmented into blocks of a predefined maximum size.

To define this block size, it is required to find an appropriate balance between robustness and decoding latency: the larger the block size, the higher the robustness (e.g., in case of long packet erasure bursts), but also the higher the maximum decoding latency (i.e., the maximum time required to recover a lost (erased) packet thanks to FEC protection). Therefore, with a multicast/broadcast session where different receivers experience different packet loss rates, the block size should be chosen by considering the worst communication conditions one wants to support, but without exceeding the desired maximum decoding latency. This choice then impacts the FEC-related latency of all receivers, even those experiencing a good communication quality, since no FEC encoding can happen until all the source data of the block is available at the sender, which directly depends on the block size.

### 1.2. Lower Latency and Better Protection of Real-Time Flows with the Sliding Window RLC Codes

This document introduces two fully-specified FEC Schemes that do not follow the block code approach: the Sliding Window Random Linear Codes (RLC) over either Galois Fields (A.K.A. Finite Fields)  $GF(2)$  (the "binary case") or  $GF(2^{255-8})$ , each time with the possibility of controlling the code density. These FEC Schemes are used to protect arbitrary media streams along the lines defined by FECFRAME extended to sliding window FEC codes [fecframe-ext]. These FEC Schemes, and more generally Sliding Window FEC codes, are recommended for instance, with media that feature real-time constraints sent within a multicast/broadcast session [Roca17].

The RLC codes belong to the broad class of sliding-window AL-FEC codes (A.K.A. convolutional codes) [RFC8406]. The encoding process is based on an encoding window that slides over the set of source packets (in fact source symbols as we will see in Section 3.2), this window being either of fixed size or variable size (A.K.A. an elastic window). Repair symbols are generated on-the-fly, by computing a random linear combination of the source symbols present in the current encoding window, and passed to the transport layer.

At the receiver, a linear system is managed from the set of received source and repair packets. New variables (representing source symbols) and equations (representing the linear combination carried by each repair symbol received) are added upon receiving new packets. Variables and the equations they are involved in are removed when they are too old with respect to their validity period (real-time constraints). Lost source symbols are then recovered thanks to this linear system whenever its rank permits to solve it (at least partially).

The protection of any multicast/broadcast session needs to be dimensioned by considering the worst communication conditions one wants to support. This is also true with RLC (more generally any sliding window) code. However, the receivers experiencing a good to medium communication quality will observe a reduced FEC-related latency compared to block codes [Roca17] since an isolated lost source packet is quickly recovered with the following repair packet. On the opposite, with a block code, recovering an isolated lost source packet always requires waiting for the first repair packet to arrive after the end of the block. Additionally, under certain situations (e.g., with a limited FEC-related latency budget and with constant bitrate transmissions after FECFRAME encoding), sliding window codes can more efficiently achieve a target transmission quality (e.g., measured by the residual loss after FEC decoding) by sending fewer repair packets (i.e., higher code rate) than block codes.

### 1.3. Small Transmission Overheads with the Sliding Window RLC FEC Scheme

The Sliding Window RLC FEC Scheme is designed to limit the packet header overhead. The main requirement is that each repair packet header must enable a receiver to reconstruct the set of source symbols plus the associated coefficients used during the encoding process. In order to minimize packet overhead, the set of source symbols in the encoding window as well as the set of coefficients over  $GF(2^m)$  (where  $m$  is 1 or 8, depending on the FEC Scheme) used in the linear combination are not individually listed in the repair packet header. Instead, each FEC Repair Packet header contains:

- o the Encoding Symbol Identifier (ESI) of the first source symbol in the encoding window as well as the number of symbols (since this number may vary with a variable size, elastic window). These two pieces of information enable each receiver to reconstruct the set of source symbols considered during encoding, the only constraint being that there cannot be any gap;
- o the seed and density threshold parameters used by a coding coefficients generation function (Section 3.6). These two pieces

of information enable each receiver to generate the same set of coding coefficients over  $GF(2^m)$  as the sender;

Therefore, no matter the number of source symbols present in the encoding window, each FEC Repair Packet features a fixed 64-bit long header, called Repair FEC Payload ID (Figure 8). Similarly, each FEC Source Packet features a fixed 32-bit long trailer, called Explicit Source FEC Payload ID (Figure 6), that contains the ESI of the first source symbol (Section 3.2).

#### 1.4. Document Organization

This fully-specified FEC Scheme follows the structure required by [RFC6363], section 5.6. "FEC Scheme Requirements", namely:

3. Procedures: This section describes procedures specific to this FEC Scheme, namely: RLC parameters derivation, ADUI and source symbols mapping, pseudo-random number generator, and coding coefficients generation function;
4. Formats and Codes: This section defines the Source FEC Payload ID and Repair FEC Payload ID formats, carrying the signaling information associated to each source or repair symbol. It also defines the FEC Framework Configuration Information (FFCI) carrying signaling information for the session;
5. FEC Code Specification: Finally this section provides the code specification.

#### 2. Definitions and Abbreviations

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the following definitions and abbreviations:

$a^b$   $a$  to the power of  $b$

$GF(q)$  denotes a finite field (also known as the Galois Field) with  $q$  elements. We assume that  $q = 2^m$  in this document

$m$  defines the length of the elements in the finite field, in bits.

In this document,  $m$  is equal to 1 or 8

ADU: Application Data Unit

ADUI: Application Data Unit Information (includes the F, L and padding fields in addition to the ADU)

E: size of an encoding symbol (i.e., source or repair symbol), assumed fixed (in bytes)



br\_in: transmission bitrate at the input of the FECFRAME sender, assumed fixed (in bits/s)  
br\_out: transmission bitrate at the output of the FECFRAME sender, assumed fixed (in bits/s)  
max\_lat: maximum FEC-related latency within FECFRAME (a decimal number expressed in seconds)  
cr: RLC coding rate, ratio between the total number of source symbols and the total number of source plus repair symbols  
ew\_size: encoding window current size at a sender (in symbols)  
ew\_max\_size: encoding window maximum size at a sender (in symbols)  
dw\_max\_size: decoding window maximum size at a receiver (in symbols)  
ls\_max\_size: linear system maximum size (or width) at a receiver (in symbols)  
WSR: window size ratio parameter used to derive ew\_max\_size (encoder) and ls\_max\_size (decoder).  
PRNG: pseudo-random number generator  
TinyMT32: PRNG used in this specification.  
DT: coding coefficients density threshold, an integer between 0 and 15 (inclusive) the controls the fraction of coefficients that are non zero

### 3. Common Procedures

This section introduces the procedures that are used by these FEC Schemes.

#### 3.1. Codec Parameters

A codec implementing the Sliding Window RLC FEC Scheme relies on several parameters:

Maximum FEC-related latency budget, max\_lat (a decimal number expressed in seconds) with real-time flows:  
a source ADU flow can have real-time constraints, and therefore any FECFRAME related operation should take place within the validity period of each ADU (Appendix D describes an exception to this rule). When there are multiple flows with different real-time constraints, we consider the most stringent constraints (see [RFC6363], Section 10.2, item 6, for recommendations when several flows are globally protected). The maximum FEC-related latency budget, max\_lat, accounts for all sources of latency added by FEC encoding (at a sender) and FEC decoding (at a receiver). Other sources of latency (e.g., added by network communications) are out of scope and must be considered separately (said differently, they have already been deducted from max\_lat). max\_lat can be regarded as the latency budget permitted for all FEC-related operations. This is an input parameter that enables a FECFRAME sender to derive other internal parameters (see Appendix C);

Encoding window current (resp. maximum) size, `ew_size` (resp. `ew_max_size`) (in symbols):

at a FECFRAME sender, during FEC encoding, a repair symbol is computed as a linear combination of the `ew_size` source symbols present in the encoding window. The `ew_max_size` is the maximum size of this window, while `ew_size` is the current size. For example, in the common case at session start, upon receiving new source ADUs, the `ew_size` progressively increases until it reaches its maximum value, `ew_max_size`. We have:

$0 < \text{ew\_size} \leq \text{ew\_max\_size}$

Decoding window maximum size, `dw_max_size` (in symbols): at a FECFRAME receiver, `dw_max_size` is the maximum number of received or lost source symbols that are still within their latency budget;

Linear system maximum size, `ls_max_size` (in symbols): at a FECFRAME receiver, the linear system maximum size, `ls_max_size`, is the maximum number of received or lost source symbols in the linear system (i.e., the variables). It SHOULD NOT be smaller than `dw_max_size` since it would mean that, even after receiving a sufficient number of FEC Repair Packets, a lost ADU may not be recovered just because the associated source symbols have been prematurely removed from the linear system, which is usually counter-productive. On the opposite, the linear system MAY grow beyond the `dw_max_size` (Appendix D);

Symbol size, `E` (in bytes): the `E` parameter determines the source and repair symbol sizes (necessarily equal). This is an input parameter that enables a FECFRAME sender to derive other internal parameters, as explained below. An implementation at a sender MUST fix the `E` parameter and MUST communicate it as part of the FEC Scheme-Specific Information (Section 4.1.1.2).

Code rate, `cr`: The code rate parameter determines the amount of redundancy added to the flow. More precisely the `cr` is the ratio between the total number of source symbols and the total number of source plus repair symbols and by definition:  $0 < \text{cr} \leq 1$ . This is an input parameter that enables a FECFRAME sender to derive other internal parameters, as explained below. However, there is no need to communicate the `cr` parameter per se (it's not required to process a repair symbol at a receiver). This code rate parameter can be static. However, in specific use-cases (e.g., with unicast transmissions in presence of a feedback mechanism that estimates the communication quality, out of scope of FECFRAME), the code rate may be adjusted dynamically.

Appendix C proposes non normative techniques to derive those parameters, depending on the use-case specificities.

### 3.2. ADU, ADUI and Source Symbols Mappings

At a sender, an ADU coming from the application is not directly mapped to source symbols. When multiple source flows (e.g., media streams) are mapped onto the same FECFRAME instance, each flow is assigned its own Flow ID value (see below). This Flow ID is then prepended to each ADU before FEC encoding. This way, FEC decoding at a receiver also recovers this Flow ID and the recovered ADU can be assigned to the right source flow (note that the 5-tuple used to identify the right source flow of a received ADU is absent with a recovered ADU since it is not FEC protected).

Additionally, since ADUs are of variable size, padding is needed so that each ADU (with its flow identifier) contribute to an integral number of source symbols. This requires adding the original ADU length to each ADU before doing FEC encoding. Because of these requirements, an intermediate format, the ADUI, or ADU Information, is considered [RFC6363].

For each incoming ADU, an ADUI MUST be created as follows. First of all, 3 bytes are prepended (Figure 1):

Flow ID (F) (8-bit field): this unsigned byte contains the integer identifier associated to the source ADU flow to which this ADU belongs. It is assumed that a single byte is sufficient, which implies that no more than 256 flows will be protected by a single FECFRAME session instance.

Length (L) (16-bit field): this unsigned integer contains the length of this ADU, in network byte order (i.e., big endian). This length is for the ADU itself and does not include the F, L, or Pad fields.

Then, zero padding is added to the ADU if needed:

Padding (Pad) (variable size field): this field contains zero padding to align the F, L, ADU and padding up to a size that is multiple of E bytes (i.e., the source and repair symbol length).

The data unit resulting from the ADU and the F, L, and Pad fields is called ADUI. Since ADUs can have different sizes, this is also the case for ADUIs. However, an ADUI always contributes to an integral number of source symbols.

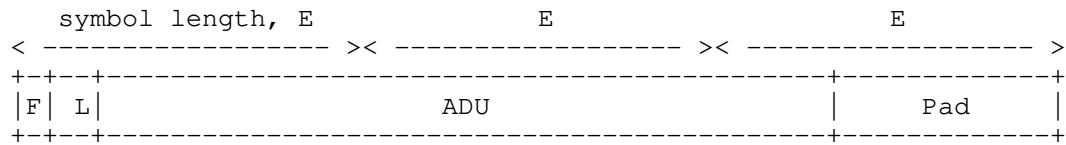


Figure 1: ADUI Creation example (here 3 source symbols are created for this ADUI).

Note that neither the initial 3 bytes nor the optional padding are sent over the network. However, they are considered during FEC encoding, and a receiver who lost a certain FEC Source Packet (e.g., the UDP datagram containing this FEC Source Packet when UDP is used as the transport protocol) will be able to recover the ADUI if FEC decoding succeeds. Thanks to the initial 3 bytes, this receiver will get rid of the padding (if any) and identify the corresponding ADU flow.

### 3.3. Encoding Window Management

Source symbols and the corresponding ADUs are removed from the encoding window:

- o when the sliding encoding window has reached its maximum size, `ew_max_size`. In that case the oldest symbol MUST be removed before adding a new symbol, so that the current encoding window size always remains inferior or equal to the maximum size: `ew_size <= ew_max_size`;
- o when an ADU has reached its maximum validity duration in case of a real-time flow. When this happens, all source symbols corresponding to the ADUI that expired SHOULD be removed from the encoding window;

Source symbols are added to the sliding encoding window each time a new ADU arrives, once the ADU-to-source symbols mapping has been performed (Section 3.2). The current size of the encoding window, `ew_size`, is updated after adding new source symbols. This process may require to remove old source symbols so that: `ew_size <= ew_max_size`.

Note that a FEC codec may feature practical limits in the number of source symbols in the encoding window (e.g., for computational complexity reasons). This factor may further limit the `ew_max_size` value, in addition to the maximum FEC-related latency budget (Section 3.1).

### 3.4. Source Symbol Identification

Each source symbol is identified by an Encoding Symbol ID (ESI), an unsigned integer. The ESI of source symbols MUST start with value 0 for the first source symbol and MUST be managed sequentially. Wrapping to zero happens after reaching the maximum value made possible by the ESI field size (this maximum value is FEC Scheme dependant, for instance,  $2^{32}-1$  with FEC Schemes XXX and YYY).

No such consideration applies to repair symbols.

### 3.5. Pseudo-Random Number Generator (PRNG)

In order to compute coding coefficients (see Section 3.6), the RLC FEC Schemes rely on the TinyMT32 PRNG defined in [tinymt32] with two additional functions defined in this section.

This PRNG MUST first be initialized with a 32-bit unsigned integer, used as a seed, with:

```
void tinymt32_init (tinymt32_t * s, uint32_t seed);
```

With the FEC Schemes defined in this document, the seed is in practice restricted to a value between 0 and 0xFFFF inclusive (note that this PRNG accepts a seed value equal to 0), since this is the Repair\_Key 16-bit field value of the Repair FEC Payload ID (Section 4.1.3). In practice, how to manage the seed and Repair\_Key values (both are equal) is left to the implementer, using a monotonically increasing counter being one possibility (Section 6.1). In addition to the seed, this function takes as parameter a pointer to an instance of a tinymt32\_t structure that is used to keep the internal state of the PRNG.

Then, each time a new pseudo-random integer between 0 and 15 inclusive (4-bit pseudo-random integer) is needed, the following function is used:

```
uint32_t tinymt32_rand16 (tinymt32_t * s);
```

This function takes as parameter a pointer to the same tinymt32\_t structure (that is left unchanged between successive calls to the function).

Similarly, each time a new pseudo-random integer between 0 and 255 inclusive (8-bit pseudo-random integer) is needed, the following function is used:

```
uint32_t tinymt32_rand256 (tinymt32_t * s);
```

These two functions keep respectively the 4 or 8 less significant bits of the 32-bit pseudo-random number generated by the `tinymt32_generate_uint32()` function of [tinymt32]. This is done by computing the result of a binary AND between the `tinymt32_generate_uint32()` output and respectively the 0xF or 0xFF constants, using 32-bit unsigned integer operations. Figure 2 shows a possible implementation. This is a C language implementation, written for C99 [C99]. Test results discussed in Appendix B show that this simple technique, applied to this PRNG, is in line with the RLC FEC Schemes needs.

```
<CODE BEGINS>
/**
 * This function outputs a pseudo-random integer in [0 .. 15] range.
 *
 * @param s      pointer to tinymt internal state.
 * @return       unsigned integer between 0 and 15 inclusive.
 */
uint32_t tinymt32_rand16(tinymt32_t *s)
{
    return (tinymt32_generate_uint32(s) & 0xF);
}

/**
 * This function outputs a pseudo-random integer in [0 .. 255] range.
 *
 * @param s      pointer to tinymt internal state.
 * @return       unsigned integer between 0 and 255 inclusive.
 */
uint32_t tinymt32_rand256(tinymt32_t *s)
{
    return (tinymt32_generate_uint32(s) & 0xFF);
}
<CODE ENDS>
```

Figure 2: 4-bit and 8-bit mapping functions for TinyMT32

Any implementation of this PRNG MUST have the same output as that provided by the reference implementation of [tinymt32]. In order to increase the compliancy confidence, three criteria are proposed: the one described in [tinymt32] (for the TinyMT32 32-bit unsigned integer generator), and the two others detailed in Appendix A (for the mapping to 4-bit and 8-bit intervals). Because of the way the mapping functions work, it is unlikely that an implementation that fulfills the first criterion fails to fulfill the two others.

### 3.6. Coding Coefficients Generation Function

The coding coefficients, used during the encoding process, are generated at the RLC encoder by the `generate_coding_coefficients()` function each time a new repair symbol needs to be produced. The fraction of coefficients that are non zero (i.e., the density) is controlled by the DT (Density Threshold) parameter. DT has values between 0 (the minimum value) and 15 (the maximum value), and the average probability of having a non zero coefficient equals  $(DT + 1) / 16$ . In particular, when DT equals 15 the function guaranties that all coefficients are non zero (i.e., maximum density).

These considerations apply to both the RLC over GF(2) and RLC over GF( $2^{^8}$ ), the only difference being the value of the m parameter. With the RLC over GF(2) FEC Scheme (Section 5), m is equal to 1. With RLC over GF( $2^{^8}$ ) FEC Scheme (Section 4), m is equal to 8.

Figure 3 shows the reference `generate_coding_coefficients()` implementation. This is a C language implementation, written for C99 [C99].

```
<CODE BEGINS>
#include <string.h>

/*
 * Fills in the table of coding coefficients (of the right size)
 * provided with the appropriate number of coding coefficients to
 * use for the repair symbol key provided.
 *
 * (in) repair_key    key associated to this repair symbol. This
 *                    parameter is ignored (useless) if m=1 and dt=15
 * (in/out) cc_tab    pointer to a table of the right size to store
 *                    coding coefficients. All coefficients are
 *                    stored as bytes, regardless of the m parameter,
 *                    upon return of this function.
 * (in) cc_nb          number of entries in the cc_tab table. This
 *                    value is equal to the current encoding window
 *                    size.
 * (in) dt             integer between 0 and 15 (inclusive) that
 *                    controls the density. With value 15, all
 *                    coefficients are guaranteed to be non zero
 *                    (i.e. equal to 1 with GF(2) and equal to a
 *                    value in {1,... 255} with GF( $2^{^8}$ )), otherwise
 *                    a fraction of them will be 0.
 * (in) m              Finite Field GF( $2^{^m}$ ) parameter. In this
 *                    document only values 1 and 8 are considered.
 * (out)               returns 0 in case of success, an error code
 *                    different than 0 otherwise.
```

```
*/
int generate_coding_coefficients (uint16_t  repair_key,
                                uint8_t*   cc_tab,
                                uint16_t   cc_nb,
                                uint8_t    dt,
                                uint8_t    m)
{
    uint32_t    i;
    tinymt32_t  s;    /* PRNG internal state */

    if (dt > 15) {
        return -1; /* error, bad dt parameter */
    }
    switch (m) {
    case 1:
        if (dt == 15) {
            /* all coefficients are 1 */
            memset(cc_tab, 1, cc_nb);
        } else {
            /* here coefficients are either 0 or 1 */
            tinymt32_init(&s, repair_key);
            for (i = 0 ; i < cc_nb ; i++) {
                cc_tab[i] = (tinymt32_rand16(&s) <= dt) ? 1 : 0;
            }
        }
        break;

    case 8:
        tinymt32_init(&s, repair_key);
        if (dt == 15) {
            /* coefficient 0 is avoided here in order to include
             * all the source symbols */
            for (i = 0 ; i < cc_nb ; i++) {
                do {
                    cc_tab[i] = (uint8_t) tinymt32_rand256(&s);
                } while (cc_tab[i] == 0);
            }
        } else {
            /* here a certain number of coefficients should be 0 */
            for (i = 0 ; i < cc_nb ; i++) {
                if (tinymt32_rand16(&s) <= dt) {
                    do {
                        cc_tab[i] = (uint8_t) tinymt32_rand256(&s);
                    } while (cc_tab[i] == 0);
                } else {
                    cc_tab[i] = 0;
                }
            }
        }
    }
}
```



```

        }
        break;

    default:
        return -2; /* error, bad parameter m */
    }
    return 0; /* success */
}
<CODE ENDS>

```

Figure 3: Coding Coefficients Generation Function Reference Implementation

### 3.7. Finite Fields Operations

#### 3.7.1. Finite Field Definitions

The two RLC FEC Schemes specified in this document reuse the Finite Fields defined in [RFC5510], section 8.1. More specifically, the elements of the field  $GF(2^m)$  are represented by polynomials with binary coefficients (i.e., over  $GF(2)$ ) and degree lower or equal to  $m-1$ . The addition between two elements is defined as the addition of binary polynomials in  $GF(2)$ , which is equivalent to a bitwise XOR operation on the binary representation of these elements.

With  $GF(2^8)$ , multiplication between two elements is the multiplication modulo a given irreducible polynomial of degree 8. The following irreducible polynomial is used for  $GF(2^8)$ :

$$x^8 + x^4 + x^3 + x^2 + 1$$

With  $GF(2)$ , multiplication corresponds to a logical AND operation.

#### 3.7.2. Linear Combination of Source Symbols Computation

The two RLC FEC Schemes require the computation of a linear combination of source symbols, using the coding coefficients produced by the `generate_coding_coefficients()` function and stored in the `cc_tab[]` array.

With the RLC over  $GF(2^8)$  FEC Scheme, a linear combination of the `ew_size` source symbol present in the encoding window, say `src_0` to `src_ew_size_1`, in order to generate a repair symbol, is computed as follows. For each byte of position `i` in each source and the repair symbol, where `i` belongs to `[0; E-1]`, compute:

$$\text{repair}[i] = \text{cc\_tab}[0] * \text{src\_0}[i] \text{ XOR } \text{cc\_tab}[1] * \text{src\_1}[i] \text{ XOR } \dots \text{ XOR } \text{cc\_tab}[\text{ew\_size} - 1] * \text{src\_ew\_size\_1}[i]$$

where  $*$  is the multiplication over  $GF(2^{^8})$ . In practice various optimizations need to be used in order to make this computation efficient (see in particular [PGM13]).

With the RLC over  $GF(2)$  FEC Scheme (binary case), a linear combination is computed as follows. The repair symbol is the XOR sum of all the source symbols corresponding to a coding coefficient  $cc\_tab[j]$  equal to 1 (i.e., the source symbols corresponding to zero coding coefficients are ignored). The XOR sum of the byte of position  $i$  in each source is computed and stored in the corresponding byte of the repair symbol, where  $i$  belongs to  $[0; E-1]$ . In practice, the XOR sums will be computed several bytes at a time (e.g., on 64 bit words, or on arrays of 16 or more bytes when using SIMD CPU extensions).

With both FEC Schemes, the details of how to optimize the computation of these linear combinations are of high practical importance but out of scope of this document.

#### 4. Sliding Window RLC FEC Scheme over $GF(2^{^8})$ for Arbitrary Packet Flows

This fully-specified FEC Scheme defines the Sliding Window Random Linear Codes (RLC) over  $GF(2^{^8})$ .

##### 4.1. Formats and Codes

###### 4.1.1. FEC Framework Configuration Information

Following the guidelines of [RFC6363], section 5.6, this section provides the FEC Framework Configuration Information (or FFCI). This FFCI needs to be shared (e.g., using SDP) between the FECFRAME sender and receiver instances in order to synchronize them. It includes a FEC Encoding ID, mandatory for any FEC Scheme specification, plus scheme-specific elements.

###### 4.1.1.1. FEC Encoding ID

- o FEC Encoding ID: the value assigned to this fully specified FEC Scheme MUST be XXXX, as assigned by IANA (Section 10).

When SDP is used to communicate the FFCI, this FEC Encoding ID is carried in the 'encoding-id' parameter.

## 4.1.1.2. FEC Scheme-Specific Information

The FEC Scheme-Specific Information (FSSI) includes elements that are specific to the present FEC Scheme. More precisely:

Encoding symbol size (E): a non-negative integer that indicates the size of each encoding symbol in bytes;

Window Size Ratio (WSR) parameter: a non-negative integer between 0 and 255 (both inclusive) used to initialize window sizes. A value of 0 indicates this parameter is not considered (e.g., a fixed encoding window size may be chosen). A value between 1 and 255 inclusive is required by certain of the parameter derivation techniques described in Appendix C;

This element is required both by the sender (RLC encoder) and the receiver(s) (RLC decoder).

When SDP is used to communicate the FFCI, this FEC Scheme-specific information is carried in the 'fssi' parameter in textual representation as specified in [RFC6364]. For instance:

```
fssi=E:1400,WSR:191
```

In that case the name values "E" and "WSR" are used to convey the E and WSR parameters respectively.

If another mechanism requires the FSSI to be carried as an opaque octet string, the encoding format consists of the following three octets, where the E field is carried in "big-endian" or "network order" format, that is, most significant byte first:

Encoding symbol length (E): 16-bit field;  
Window Size Ratio Parameter (WSR): 8-bit field.

These three octets can be communicated as such, or for instance, be subject to an additional Base64 encoding.

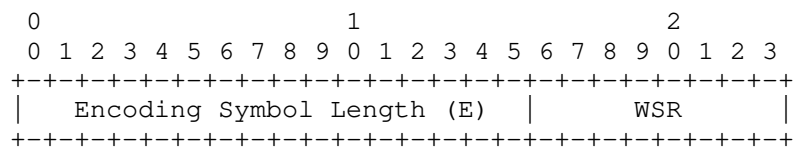


Figure 4: FSSI Encoding Format

#### 4.1.2. Explicit Source FEC Payload ID

A FEC Source Packet MUST contain an Explicit Source FEC Payload ID that is appended to the end of the packet as illustrated in Figure 5.

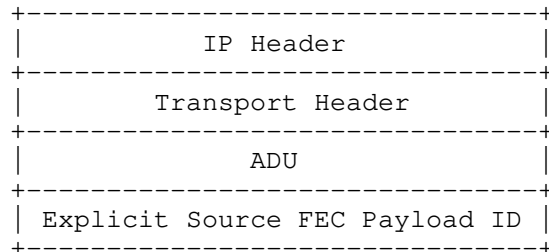


Figure 5: Structure of an FEC Source Packet with the Explicit Source FEC Payload ID

More precisely, the Explicit Source FEC Payload ID is composed of the following field, carried in "big-endian" or "network order" format, that is, most significant byte first (Figure 6):

Encoding Symbol ID (ESI) (32-bit field): this unsigned integer identifies the first source symbol of the ADUI corresponding to this FEC Source Packet. The ESI is incremented for each new source symbol, and after reaching the maximum value ( $2^{32}-1$ ), wrapping to zero occurs.

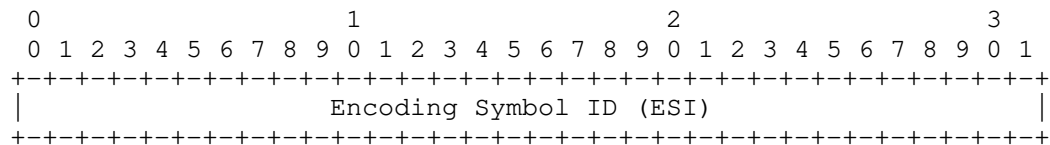


Figure 6: Source FEC Payload ID Encoding Format

#### 4.1.3. Repair FEC Payload ID

A FEC Repair Packet MAY contain one or more repair symbols. When there are several repair symbols, all of them MUST have been generated from the same encoding window, using Repair\_Key values that are managed as explained below. A receiver can easily deduce the number of repair symbols within a FEC Repair Packet by comparing the received FEC Repair Packet size (equal to the UDP payload size when UDP is the underlying transport protocol) and the symbol size, E, communicated in the FFCI.

A FEC Repair Packet MUST contain a Repair FEC Payload ID that is prepended to the repair symbol as illustrated in Figure 7.

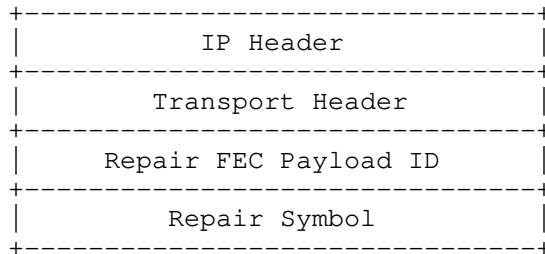


Figure 7: Structure of an FEC Repair Packet with the Repair FEC Payload ID

More precisely, the Repair FEC Payload ID is composed of the following fields where all integer fields are carried in "big-endian" or "network order" format, that is, most significant byte first (Figure 8):

**Repair\_Key (16-bit field):** this unsigned integer is used as a seed by the coefficient generation function (Section 3.6) in order to generate the desired number of coding coefficients. This repair key may be a monotonically increasing integer value that loops back to 0 after reaching 65535 (see Section 6.1). When a FEC Repair Packet contains several repair symbols, this repair key value is that of the first repair symbol. The remaining repair keys can be deduced by incrementing by 1 this value, up to a maximum value of 65535 after which it loops back to 0.

**Density Threshold for the coding coefficients, DT (4-bit field):** this unsigned integer carries the Density Threshold (DT) used by the coding coefficient generation function Section 3.6. More precisely, it controls the probability of having a non zero coding coefficient, which equals  $(DT+1) / 16$ . When a FEC Repair Packet contains several repair symbols, the DT value applies to all of them;

**Number of Source Symbols in the encoding window, NSS (12-bit field):**

this unsigned integer indicates the number of source symbols in the encoding window when this repair symbol was generated. When a FEC Repair Packet contains several repair symbols, this NSS value applies to all of them;

**ESI of First Source Symbol in the encoding window, FSS\_ESI (32-bit field):**

this unsigned integer indicates the ESI of the first source symbol in the encoding window when this repair symbol was generated.

When a FEC Repair Packet contains several repair symbols, this FSS\_ESI value applies to all of them;

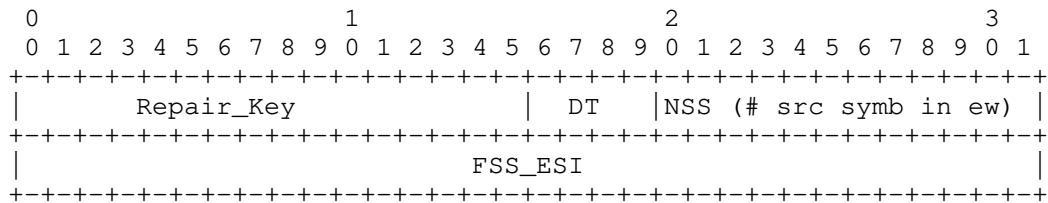


Figure 8: Repair FEC Payload ID Encoding Format

#### 4.2. Procedures

All the procedures of Section 3 apply to this FEC Scheme.

#### 5. Sliding Window RLC FEC Scheme over GF(2) for Arbitrary Packet Flows

This fully-specified FEC Scheme defines the Sliding Window Random Linear Codes (RLC) over GF(2) (binary case).

##### 5.1. Formats and Codes

###### 5.1.1. FEC Framework Configuration Information

###### 5.1.1.1. FEC Encoding ID

- o FEC Encoding ID: the value assigned to this fully specified FEC Scheme MUST be YYYY, as assigned by IANA (Section 10).

When SDP is used to communicate the FFCI, this FEC Encoding ID is carried in the 'encoding-id' parameter.

###### 5.1.1.2. FEC Scheme-Specific Information

All the considerations of Section 4.1.1.2 apply here.

###### 5.1.2. Explicit Source FEC Payload ID

All the considerations of Section 4.1.2 apply here.

###### 5.1.3. Repair FEC Payload ID

All the considerations of Section 4.1.3 apply here, with the only exception that the Repair\_Key field is useless if DT = 15 (indeed, in that case all the coefficients are necessarily equal to 1 and the coefficient generation function does not use any PRNG). When DT = 15

the FECFRAME sender MUST set the Repair\_Key field to zero on transmission and a receiver MUST ignore it on receipt.

## 5.2. Procedures

All the procedures of Section 3 apply to this FEC Scheme.

## 6. FEC Code Specification

### 6.1. Encoding Side

This section provides a high level description of a Sliding Window RLC encoder.

Whenever a new FEC Repair Packet is needed, the RLC encoder instance first gathers the ew\_size source symbols currently in the sliding encoding window. Then it chooses a repair key, which can be a monotonically increasing integer value, incremented for each repair symbol up to a maximum value of 65535 (as it is carried within a 16-bit field) after which it loops back to 0. This repair key is communicated to the coefficient generation function (Section 3.6) in order to generate ew\_size coding coefficients. Finally, the FECFRAME sender computes the repair symbol as a linear combination of the ew\_size source symbols using the ew\_size coding coefficients (Section 3.7). When E is small and when there is an incentive to pack several repair symbols within the same FEC Repair Packet, the appropriate number of repair symbols are computed. In that case the repair key for each of them MUST be incremented by 1, keeping the same ew\_size source symbols, since only the first repair key will be carried in the Repair FEC Payload ID. The FEC Repair Packet can then be passed to the transport layer for transmission. The source versus repair FEC packet transmission order is out of scope of this document and several approaches exist that are implementation-specific.

Other solutions are possible to select a repair key value when a new FEC Repair Packet is needed, for instance, by choosing a random integer between 0 and 65535. However, selecting the same repair key as before (which may happen in case of a random process) is only meaningful if the encoding window has changed, otherwise the same FEC Repair Packet will be generated. In any case, choosing the repair key is entirely at the discretion of the sender, since it is communicated to the receiver(s) in each Repair FEC Payload ID. A receiver should not make any assumption on the way the repair key is managed.

## 6.2. Decoding Side

This section provides a high level description of a Sliding Window RLC decoder.

A FECFRAME receiver needs to maintain a linear system whose variables are the received and lost source symbols. Upon receiving a FEC Repair Packet, a receiver first extracts all the repair symbols it contains (in case several repair symbols are packed together). For each repair symbol, when at least one of the corresponding source symbols it protects has been lost, the receiver adds an equation to the linear system (or no equation if this repair packet does not change the linear system rank). This equation of course re-uses the `ew_size` coding coefficients that are computed by the same coefficient generation function (Section 3.6), using the repair key and encoding window descriptions carried in the Repair FEC Payload ID. Whenever possible (i.e., when a sub-system covering one or more lost source symbols is of full rank), decoding is performed in order to recover lost source symbols. Gaussian elimination is one possible algorithm to solve this linear system. Each time an ADUI can be totally recovered, padding is removed (thanks to the Length field, `L`, of the ADUI) and the ADU is assigned to the corresponding application flow (thanks to the Flow ID field, `F`, of the ADUI). This ADU is finally passed to the corresponding upper application. Received FEC Source Packets, containing an ADU, MAY be passed to the application either immediately or after some time to guaranty an ordered delivery to the application. This document does not mandate any approach as this is an operational and management decision.

With real-time flows, a lost ADU that is decoded after the maximum latency or an ADU received after this delay has no value to the application. This raises the question of deciding whether or not an ADU is late. This decision MAY be taken within the FECFRAME receiver (e.g., using the decoding window, see Section 3.1) or within the application (e.g., using RTP timestamps within the ADU). Deciding which option to follow and whether or not to pass all ADUs, including those assumed late, to the application are operational decisions that depend on the application and are therefore out of scope of this document. Additionally, Appendix D discusses a backward compatible optimization whereby late source symbols MAY still be used within the FECFRAME receiver in order to improve transmission robustness.

## 7. Implementation Status

Editor's notes: RFC Editor, please remove this section motivated by RFC 6982 before publishing the RFC. Thanks.



An implementation of the Sliding Window RLC FEC Scheme for FECFRAME exists:

- o Organisation: Inria
- o Description: This is an implementation of the Sliding Window RLC FEC Scheme limited to  $GF(2^{8})$ . It relies on a modified version of our OpenFEC (<http://openfec.org>) FEC code library. It is integrated in our FECFRAME software (see [fecframe-ext]).
- o Maturity: prototype.
- o Coverage: this software complies with the Sliding Window RLC FEC Scheme.
- o Licensing: proprietary.
- o Contact: [vincent.roca@inria.fr](mailto:vincent.roca@inria.fr)

## 8. Security Considerations

The FEC Framework document [RFC6363] provides a fairly comprehensive analysis of security considerations applicable to FEC Schemes. Therefore, the present section follows the security considerations section of [RFC6363] and only discusses specific topics.

### 8.1. Attacks Against the Data Flow

#### 8.1.1. Access to Confidential Content

The Sliding Window RLC FEC Scheme specified in this document does not change the recommendations of [RFC6363]. To summarize, if confidentiality is a concern, it is RECOMMENDED that one of the solutions mentioned in [RFC6363] is used with special considerations to the way this solution is applied (e.g., is encryption applied before or after FEC protection, within the end-system or in a middlebox), to the operational constraints (e.g., performing FEC decoding in a protected environment may be complicated or even impossible) and to the threat model.

#### 8.1.2. Content Corruption

The Sliding Window RLC FEC Scheme specified in this document does not change the recommendations of [RFC6363]. To summarize, it is RECOMMENDED that one of the solutions mentioned in [RFC6363] is used on both the FEC Source and Repair Packets.

### 8.2. Attacks Against the FEC Parameters

The FEC Scheme specified in this document defines parameters that can be the basis of attacks. More specifically, the following parameters of the FFCI may be modified by an attacker who targets receivers (Section 4.1.1.2):

- o FEC Encoding ID: changing this parameter leads a receiver to consider a different FEC Scheme. The consequences are severe, the format of the Explicit Source FEC Payload ID and Repair FEC Payload ID of received packets will probably differ, leading to various malfunctions. Even if the original and modified FEC Schemes share the same format, FEC decoding will either fail or lead to corrupted decoded symbols. This will happen if an attacker turns value YYYY (i.e., RLC over  $GF(2)$ ) to value XXXX (RLC over  $GF(2^{8})$ ), an additional consequence being a higher processing overhead at the receiver. In any case, the attack results in a form of Denial of Service (DoS) or corrupted content.
- o Encoding symbol length (E): setting this E parameter to a different value will confuse a receiver. If the size of a received FEC Repair Packet is no longer multiple of the modified E value, a receiver quickly detects a problem and SHOULD reject the packet. If the new E value is a sub-multiple of the original E value (e.g., half the original value), then receivers may not detect the problem immediately. For instance, a receiver may think that a received FEC Repair Packet contains more repair symbols (e.g., twice as many if E is reduced by half), leading to malfunctions whose nature depends on implementation details. Here also, the attack always results in a form of DoS or corrupted content.

It is therefore RECOMMENDED that security measures be taken to guarantee the FFCI integrity, as specified in [RFC6363]. How to achieve this depends on the way the FFCI is communicated from the sender to the receiver, which is not specified in this document.

Similarly, attacks are possible against the Explicit Source FEC Payload ID and Repair FEC Payload ID. More specifically, in case of a FEC Source Packet, the following value can be modified by an attacker who targets receivers:

- o Encoding Symbol ID (ESI): changing the ESI leads a receiver to consider a wrong ADU, resulting in severe consequences, including corrupted content passed to the receiving application;

And in case of a FEC Repair Packet:

- o Repair Key: changing this value leads a receiver to generate a wrong coding coefficient sequence, and therefore any source symbol decoded using the repair symbols contained in this packet will be corrupted;
- o DT: changing this value also leads a receiver to generate a wrong coding coefficient sequence, and therefore any source symbol decoded using the repair symbols contained in this packet will be corrupted. In addition, if the DT value is significantly

- increased, it will generate a higher processing overhead at a receiver. In case of very large encoding windows, this may impact the terminal performance;
- o NSS: changing this value leads a receiver to consider a different set of source symbols, and therefore any source symbol decoded using the repair symbols contained in this packet will be corrupted. In addition, if the NSS value is significantly increased, it will generate a higher processing overhead at a receiver, which may impact the terminal performance;
  - o FSS\_ESI: changing this value also leads a receiver to consider a different set of source symbols and therefore any source symbol decoded using the repair symbols contained in this packet will be corrupted.

It is therefore RECOMMENDED that security measures are taken to guarantee the FEC Source and Repair Packets as stated in [RFC6363].

### 8.3. When Several Source Flows are to be Protected Together

The Sliding Window RLC FEC Scheme specified in this document does not change the recommendations of [RFC6363].

### 8.4. Baseline Secure FEC Framework Operation

The Sliding Window RLC FEC Scheme specified in this document does not change the recommendations of [RFC6363] concerning the use of the IPsec/ESP security protocol as a mandatory to implement (but not mandatory to use) security scheme. This is well suited to situations where the only insecure domain is the one over which the FEC Framework operates.

### 8.5. Additional Security Considerations for Numerical Computations

In addition to the above security considerations, inherited from [RFC6363], the present document introduces several formulae, in particular in Appendix C.1. It is RECOMMENDED to check that the computed values stay within reasonable bounds since numerical overflows, caused by an erroneous implementation or an erroneous input value, may lead to hazardous behaviours. However, what "reasonable bounds" means is use-case and implementation dependent and is not detailed in this document.

Appendix C.2 also mentions the possibility of "using the timestamp field of an RTP packet header" when applicable. A malicious attacker may deliberately corrupt this header field in order to trigger hazardous behaviours at a FECFRAME receiver. Protection against this type of content corruption can be addressed with the above recommendations on a baseline secure operation. In addition, it is

also RECOMMENDED to check that the timestamp value be within reasonable bounds.

## 9. Operations and Management Considerations

The FEC Framework document [RFC6363] provides a fairly comprehensive analysis of operations and management considerations applicable to FEC Schemes. Therefore, the present section only discusses specific topics.

### 9.1. Operational Recommendations: Finite Field GF(2) Versus GF(2<sup>8</sup>)

The present document specifies two FEC Schemes that differ on the Finite Field used for the coding coefficients. It is expected that the RLC over GF(2<sup>8</sup>) FEC Scheme will be mostly used since it warrants a higher packet loss protection. In case of small encoding windows, the associated processing overhead is not an issue (e.g., we measured decoding speeds between 745 Mbps and 2.8 Gbps on an ARM Cortex-A15 embedded board in [Roca17] depending on the code rate and the channel conditions, using an encoding window of size 18 or 23 symbols; see the above article for the details). Of course the CPU overhead will increase with the encoding window size, because more operations in the GF(2<sup>8</sup>) finite field will be needed.

The RLC over GF(2) FEC Scheme offers an alternative. In that case operations symbols can be directly XOR-ed together which warrants high bitrate encoding and decoding operations, and can be an advantage with large encoding windows. However, packet loss protection is significantly reduced by using this FEC Scheme.

### 9.2. Operational Recommendations: Coding Coefficients Density Threshold

In addition to the choice of the Finite Field, the two FEC Schemes define a coding coefficient density threshold (DT) parameter. This parameter enables a sender to control the code density, i.e., the proportion of coefficients that are non zero on average. With RLC over GF(2<sup>8</sup>), it is usually appropriate that small encoding windows be associated to a density threshold equal to 15, the maximum value, in order to warrant a high loss protection.

On the opposite, with larger encoding windows, it is usually appropriate that the density threshold be reduced. With large encoding windows, an alternative can be to use RLC over GF(2) and a density threshold equal to 7 (i.e., an average density equal to 1/2) or smaller.

Note that using a density threshold equal to 15 with RLC over GF(2) is equivalent to using an XOR code that computes the XOR sum of all

the source symbols in the encoding window. In that case: (1) only a single repair symbol can be produced for any encoding window, and (2) the `repair_key` parameter becomes useless (the coding coefficients generation function does not rely on the PRNG).

## 10. IANA Considerations

This document registers two values in the "FEC Framework (FECFRAME) FEC Encoding IDs" registry [RFC6363] as follows:

- o YYYY refers to the Sliding Window Random Linear Codes (RLC) over GF(2) FEC Scheme for Arbitrary Packet Flows, as defined in Section 5 of this document.
- o XXXX refers to the Sliding Window Random Linear Codes (RLC) over GF(2<sup>8</sup>) FEC Scheme for Arbitrary Packet Flows, as defined in Section 4 of this document.

## 11. Acknowledgments

The authors would like to thank the three TSVWG chairs, Wesley Eddy, our shepherd, David Black and Gorrry Fairhurst, as well as Spencer Dawkins, our responsible AD, and all those who provided comments, namely (alphabetical order) Alan DeKok, Jonathan Detchart, Russ Housley, Emmanuel Lochin, Marie-Jose Montpetit, and Greg Skinner. Last but not least, the authors are really grateful to the IESG members, in particular Benjamin Kaduk, Mirja Kuhlewind, Eric Rescorla, Adam Roach, and Roman Danyliw for their highly valuable feedbacks that greatly contributed to improve this specification.

## 12. References

### 12.1. Normative References

- [C99] "Programming languages - C: C99, correction 3:2007", International Organization for Standardization, ISO/IEC 9899:1999/Cor 3:2007, November 2007.
- [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), January 2019, <<https://tools.ietf.org/html/draft-ietf-tsvwg-fecframe-ext>>.

- [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>>.
- [RFC6364] Begen, A., "Session Description Protocol Elements for the Forward Error Correction (FEC) Framework", RFC 6364, DOI 10.17487/RFC6364, October 2011, <<https://www.rfc-editor.org/info/rfc6364>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [tinymt32] Saito, M., Matsumoto, M., Roca, V., and E. Baccelli, "TinyMT32 Pseudo Random Number Generator (PRNG)", Transport Area Working Group (TSVWG) draft-roca-tsvwg-tinymt32 (Work in Progress), February 2019, <<https://tools.ietf.org/html/draft-roca-tsvwg-tinymt32>>.

## 12.2. Informative References

- [PGM13] Plank, J., Greenan, K., and E. Miller, "A Complete Treatment of Software Implementations of Finite Field Arithmetic for Erasure Coding Applications", University of Tennessee Technical Report UT-CS-13-717, <http://web.eecs.utk.edu/~plank/plank/papers/UT-CS-13-717.html>, October 2013, <<http://web.eecs.utk.edu/~plank/plank/papers/UT-CS-13-717.html>>.
- [RFC5170] Roca, V., Neumann, C., and D. Furodet, "Low Density Parity Check (LDPC) Staircase and Triangle Forward Error Correction (FEC) Schemes", RFC 5170, DOI 10.17487/RFC5170, June 2008, <<https://www.rfc-editor.org/info/rfc5170>>.
- [RFC5510] Lacan, J., Roca, V., Peltotalo, J., and S. Peltotalo, "Reed-Solomon Forward Error Correction (FEC) Schemes", RFC 5510, DOI 10.17487/RFC5510, April 2009, <<https://www.rfc-editor.org/info/rfc5510>>.

- [RFC6681] Watson, M., Stockhammer, T., and M. Luby, "Raptor Forward Error Correction (FEC) Schemes for FECFRAME", RFC 6681, DOI 10.17487/RFC6681, August 2012, <<https://www.rfc-editor.org/info/rfc6681>>.
- [RFC6726] Paila, T., Walsh, R., Luby, M., Roca, V., and R. Lehtonen, "FLUTE - File Delivery over Unidirectional Transport", RFC 6726, DOI 10.17487/RFC6726, November 2012, <<https://www.rfc-editor.org/info/rfc6726>>.
- [RFC6816] Roca, V., Cunche, M., and J. Lacan, "Simple Low-Density Parity Check (LDPC) Staircase Forward Error Correction (FEC) Scheme for FECFRAME", RFC 6816, DOI 10.17487/RFC6816, December 2012, <<https://www.rfc-editor.org/info/rfc6816>>.
- [RFC6865] Roca, V., Cunche, M., Lacan, J., Bouabdallah, A., and K. Matsuzono, "Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME", RFC 6865, DOI 10.17487/RFC6865, February 2013, <<https://www.rfc-editor.org/info/rfc6865>>.
- [RFC8406] Adamson, B., Adjih, C., Bilbao, J., Firoiu, V., Fitzek, F., Ghanem, S., Lochin, E., Masucci, A., Montpetit, M-J., Pedersen, M., Peralta, G., Roca, V., Ed., Saxena, P., and S. Sivakumar, "Taxonomy of Coding Techniques for Efficient Network Communications", RFC 8406, DOI 10.17487/RFC8406, June 2018, <<https://www.rfc-editor.org/info/rfc8406>>.
- [Roca16] Roca, V., Teibi, B., Burdinat, C., Tran, T., and C. Thienot, "Block or Convolutional AL-FEC Codes? A Performance Comparison for Robust Low-Latency Communications", HAL open-archive document, hal-01395937 <https://hal.inria.fr/hal-01395937/en/>, November 2016, <<https://hal.inria.fr/hal-01395937/en/>>.
- [Roca17] Roca, V., Teibi, B., Burdinat, C., Tran, T., and C. Thienot, "Less Latency and Better Protection with AL-FEC Sliding Window Codes: a Robust Multimedia CBR Broadcast Case Study", 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob17), October 2017 <https://hal.inria.fr/hal-01571609v1/en/>, October 2017, <<https://hal.inria.fr/hal-01571609v1/en/>>.

## Appendix A. TinyMT32 Validation Criteria (Normative)

PRNG determinism, for a given seed, is a requirement. Consequently, in order to validate an implementation of the TinyMT32 PRNG, the following criteria MUST be met.

The first criterion focusses on the `tinymt32_rand256()`, where the 32-bit integer of the core TinyMT32 PRNG is scaled down to an 8-bit integer. Using a seed value of 1, the first 50 values returned by: `tinymt32_rand256()` as 8-bit unsigned integers MUST be equal to values provided in Figure 9, to be read line by line.

37	225	177	176	21
246	54	139	168	237
211	187	62	190	104
135	210	99	176	11
207	35	40	113	179
214	254	101	212	211
226	41	234	232	203
29	194	211	112	107
217	104	197	135	23
89	210	252	109	166

Figure 9: First 50 decimal values (to be read per line) returned by `tinymt32_rand256()` as 8-bit unsigned integers, with a seed value of 1.

The second criterion focusses on the `tinymt32_rand16()`, where the 32-bit integer of the core TinyMT32 PRNG is scaled down to a 4-bit integer. Using a seed value of 1, the first 50 values returned by: `tinymt32_rand16()` as 4-bit unsigned integers MUST be equal to values provided in Figure 10, to be read line by line.

5	1	1	0	5
6	6	11	8	13
3	11	14	14	8
7	2	3	0	11
15	3	8	1	3
6	14	5	4	3
2	9	10	8	11
13	2	3	0	11
9	8	5	7	7
9	2	12	13	6

Figure 10: First 50 decimal values (to be read per line) returned by `tinymt32_rand16()` as 4-bit unsigned integers, with a seed value of 1.



## Appendix B. Assessing the PRNG Adequacy (Informational)

This annex discusses the adequacy of the TinyMT32 PRNG and the `tinymt32_rand16()` and `tinymt32_rand256()` functions, to the RLC FEC Schemes. The goal is to assess the adequacy of these two functions in producing coding coefficients that are sufficiently different from one another, across various repair symbols with repair key values in sequence (we can expect this approach to be commonly used by implementers, see Section 6.1). This section is purely informational and does not claim to be a solid evaluation.

The two RLC FEC Schemes use the PRNG to produce pseudo-random coding coefficients (Section 3.6), each time a new repair symbol is needed. A different repair key is used for each repair symbol, usually by incrementing the repair key value (Section 6.1). For each repair symbol, a limited number of pseudo-random numbers is needed, depending on the DT and encoding window size (Section 3.6), using either `tinymt32_rand16()` or `tinymt32_rand256()`. Therefore we are more interested in the randomness of small sequences of random numbers mapped to 4-bit or 8-bit integers, than in the randomness of a very large sequence of random numbers which is not representative of the usage of the PRNG.

Evaluation of `tinymt32_rand16()`: We first generate a huge number (1,000,000,000) of small sequences (20 pseudo-random numbers per sequence), increasing the seed value for each sequence, and perform statistics on the number of occurrences of each of the 16 possible values across all sequences. In this first test we consider 32-bit seed values in order to assess the PRNG quality after output truncation to 4 bits.

value	occurrences	percentage (%) (total of 20000000000)
0	1250036799	6.2502
1	1249995831	6.2500
2	1250038674	6.2502
3	1250000881	6.2500
4	1250023929	6.2501
5	1249986320	6.2499
6	1249995587	6.2500
7	1250020363	6.2501
8	1249995276	6.2500
9	1249982856	6.2499
10	1249984111	6.2499
11	1250009551	6.2500
12	1249955768	6.2498
13	1249994654	6.2500
14	1250000569	6.2500
15	1249978831	6.2499

Figure 11: `tinymt32_rand16()`: occurrence statistics across a huge number (1,000,000,000) of small sequences (20 pseudo-random numbers per sequence), with 0 as the first PRNG seed.

The results (Figure 11) show that all possible values are almost equally represented, or said differently, that the `tinymt32_rand16()` output converges to a uniform distribution where each of the 16 possible values would appear exactly  $1 / 16 * 100 = 6.25\%$  of times.

Since the RLC FEC Schemes use of this PRNG will be limited to 16-bit seed values, we carried out the same test for the first  $2^{16}$  seed values only. The distribution (not shown) is of course less uniform, with value occurrences ranging between 6.2121% (i.e., 81,423 occurrences out of a total of  $65536 * 20 = 1,310,720$ ) and 6.2948% (i.e., 82,507 occurrences). However, we do not believe it significantly impacts the RLC FEC Scheme behavior.

Other types of biases may exist that may be visible with smaller tests, for instance to evaluate the convergence speed to a uniform distribution. We therefore perform 200 tests, each of them consisting in producing 200 sequences, keeping only the first value of each sequence. We use non overlapping repair keys for each sequence, starting with value 0 and increasing it after each use.

value	min occurrences	max occurrences	average occurrences
0	4	21	6.3675
1	4	22	6.0200
2	4	20	6.3125
3	5	23	6.1775
4	5	24	6.1000
5	4	21	6.5925
6	5	30	6.3075
7	6	22	6.2225
8	5	26	6.1750
9	3	21	5.9425
10	5	24	6.3175
11	4	22	6.4300
12	5	21	6.1600
13	5	22	6.3100
14	4	26	6.3950
15	4	21	6.1700

Figure 12: `tinymt32_rand16()`: occurrence statistics across 200 tests, each of them consisting in 200 sequences of 1 pseudo-random number each, with non overlapping PRNG seeds in sequence starting from 0.

Figure 12 shows across all 200 tests, for each of the 16 possible pseudo-random number values, the minimum (resp. maximum) number of times it appeared in a test, as well as the average number of occurrences across the 200 tests. Although the distribution is not perfect, there is no major bias. On the opposite, in the same conditions, the Park-Miller linear congruential PRNG of [RFC5170] with a result scaled down to 4-bit values, using seeds in sequence starting from 1, returns systematically 0 as the first value during some time, then after a certain repair key value threshold, it systematically returns 1, etc.

Evaluation of `tinymt32_rand256()`: The same approach is used here. Results (not shown) are similar: occurrences vary between 7,810,3368 (i.e., 0.3905%) and 7,814,7952 (i.e., 0.3907%). Here also we see a convergence to the theoretical uniform distribution where each of the 256 possible values would appear exactly  $1 / 256 * 100 = 0.390625\%$  of times.

#### Appendix C. Possible Parameter Derivation (Informational)

Section 3.1 defines several parameters to control the encoder or decoder. This annex proposes techniques to derive these parameters according to the target use-case. This annex is informational, in the sense that using a different derivation technique will not prevent the encoder and decoder to interoperate: a decoder can still recover an erased source symbol without any error. However, in case

of a real-time flow, an inappropriate parameter derivation may lead to the decoding of erased source packets after their validity period, making them useless to the target application. This annex proposes an approach to reduce this risk, among other things.

The FEC Schemes defined in this document can be used in various manners, depending on the target use-case:

- o the source ADU flow they protect may or may not have real-time constraints;
- o the source ADU flow may be a Constant Bitrate (CBR) or Variable BitRate (VBR) flow;
- o with a VBR source ADU flow, the flow's minimum and maximum bitrates may or may not be known;
- o and the communication path between encoder and decoder may be a CBR communication path (e.g., as with certain LTE-based broadcast channels) or not (general case, e.g., with Internet).

The parameter derivation technique should be suited to the use-case, as described in the following sections.

#### C.1. Case of a CBR Real-Time Flow

In the following, we consider a real-time flow with `max_lat` latency budget. The encoding symbol size, `E`, is constant. The code rate, `cr`, is also constant, its value depending on the expected communication loss model (this choice is out of scope of this document).

In a first configuration, the source ADU flow bitrate at the input of the FECFRAME sender is fixed and equal to `br_in` (in bits/s), and this value is known by the FECFRAME sender. It follows that the transmission bitrate at the output of the FECFRAME sender will be higher, depending on the added repair flow overhead. In order to comply with the maximum FEC-related latency budget, we have:

$$\text{dw\_max\_size} = (\text{max\_lat} * \text{br\_in}) / (8 * E)$$

assuming that the encoding and decoding times are negligible with respect to the target `max_lat`. This is a reasonable assumption in many situations (e.g., see Section 9.1 in case of small window sizes). Otherwise the `max_lat` parameter should be adjusted in order to avoid the problem. In any case, interoperability will never be compromised by choosing a too large value.

In a second configuration, the FECFRAME sender generates a fixed bitrate flow, equal to the CBR communication path bitrate equal to `br_out` (in bits/s), and this value is known by the FECFRAME sender,

as in [Roca17]. The maximum source flow bitrate needs to be such that, with the added repair flow overhead, the total transmission bitrate remains inferior or equal to `br_out`. We have:

$$\text{dw\_max\_size} = (\text{max\_lat} * \text{br\_out} * \text{cr}) / (8 * E)$$

assuming here also that the encoding and decoding times are negligible with respect to the target `max_lat`.

For decoding to be possible within the latency budget, it is required that the encoding window maximum size be smaller than or at most equal to the decoding window maximum size. The `ew_max_size` is the main parameter at a FECFRAME sender, but its exact value has no impact on the the FEC-related latency budget. The `ew_max_size` parameter is computed as follows:

$$\text{ew\_max\_size} = \text{dw\_max\_size} * \text{WSR} / 255$$

In line with [Roca17], `WSR = 191` is considered as a reasonable value (the resulting encoding to decoding window size ratio is then close to 0.75), but other values between 1 and 255 inclusive are possible, depending on the use-case.

The `dw_max_size` is computed by a FECFRAME sender but not explicitly communicated to a FECFRAME receiver. However, a FECFRAME receiver can easily evaluate the `ew_max_size` by observing the maximum Number of Source Symbols (NSS) value contained in the Repair FEC Payload ID of received FEC Repair Packets (Section 4.1.3). A receiver can then easily compute `dw_max_size`:

$$\text{dw\_max\_size} = \text{max\_NSS\_observed} * 255 / \text{WSR}$$

A receiver can then chose an appropriate linear system maximum size:

$$\text{ls\_max\_size} \geq \text{dw\_max\_size}$$

It is good practice to use a larger value for `ls_max_size` as explained in Appendix D, which does not impact maximum latency nor interoperability.

In any case, for a given use-case (i.e., for target encoding and decoding devices and desired protection levels in front of communication impairments) and for the computed `ew_max_size`, `dw_max_size` and `ls_max_size` values, it is RECOMMENDED to check that the maximum encoding time and maximum memory requirements at a FECFRAME sender, and maximum decoding time and maximum memory requirements at a FECFRAME receiver, stay within reasonable bounds. When assuming that the encoding and decoding times are negligible

with respect to the target `max_lat`, this should be verified as well, otherwise the `max_lat` SHOULD be adjusted accordingly.

The particular case of session start needs to be managed appropriately since the `ew_size`, starting at zero, increases each time a new source ADU is received by the FECFRAME sender, until it reaches the `ew_max_size` value. Therefore a FECFRAME receiver SHOULD continuously observe the received FEC Repair Packets, since the NSS value carried in the Repair FEC Payload ID will increase too, and adjust its `ls_max_size` accordingly if need be. With a CBR flow, session start is expected to be the only moment when the encoding window size will increase. Similarly, with a CBR real-time flow, the session end is expected to be the only moment when the encoding window size will progressively decrease. No adjustment of the `ls_max_size` is required at the FECFRAME receiver in that case.

## C.2. Other Types of Real-Time Flow

In the following, we consider a real-time source ADU flow with a `max_lat` latency budget and a variable bitrate (VBR) measured at the entry of the FECFRAME sender. A first approach consists in considering the smallest instantaneous bitrate of the source ADU flow, when this parameter is known, and to reuse the derivation of Appendix C.1. Considering the smallest bitrate means that the encoding and decoding window maximum size estimations are pessimistic: these windows have the smallest size required to enable on-time decoding at a FECFRAME receiver. If the instantaneous bitrate is higher than this smallest bitrate, this approach leads to an encoding window that is unnecessarily small, which reduces robustness in front of long erasure bursts.

Another approach consists in using ADU timing information (e.g., using the timestamp field of an RTP packet header, or registering the time upon receiving a new ADU). From the global FEC-related latency budget, the FECFRAME sender can derive a practical maximum latency budget for encoding operations, `max_lat_for_encoding`. For the FEC Schemes specified in this document, this latency budget SHOULD be computed with:

$$\text{max\_lat\_for\_encoding} = \text{max\_lat} * \text{WSR} / 255$$

It follows that any source symbols associated to an ADU that has timed-out with respect to `max_lat_for_encoding` SHOULD be removed from the encoding window. With this approach there is no pre-determined `ew_size` value: this value fluctuates over the time according to the instantaneous source ADU flow bitrate. For practical reasons, a FECFRAME sender may still require that `ew_size` does not increase beyond a maximum value (Appendix C.3).

With both approaches, and no matter the choice of the FECFRAME sender, a FECFRAME receiver can still easily evaluate the `ew_max_size` by observing the maximum Number of Source Symbols (NSS) value contained in the Repair FEC Payload ID of received FEC Repair Packets. A receiver can then compute `dw_max_size` and derive an appropriate `ls_max_size` as explained in Appendix C.1.

When the observed NSS fluctuates significantly, a FECFRAME receiver may want to adapt its `ls_max_size` accordingly. In particular when the NSS is significantly reduced, a FECFRAME receiver may want to reduce the `ls_max_size` too in order to limit computation complexity. A balance must be found between using an `ls_max_size` "too large" (which increases computation complexity and memory requirements) and the opposite (which reduces recovery performance).

### C.3. Case of a Non Real-Time Flow

Finally there are configurations where a source ADU flow has no real-time constraints. FECFRAME and the FEC Schemes defined in this document can still be used. The choice of appropriate parameter values can be directed by practical considerations. For instance, it can derive from an estimation of the maximum memory amount that could be dedicated to the linear system at a FECFRAME receiver, or the maximum computation complexity at a FECFRAME receiver, both of them depending on the `ls_max_size` parameter. The same considerations also apply to the FECFRAME sender, where the maximum memory amount and computation complexity depend on the `ew_max_size` parameter.

Here also, the NSS value contained in FEC Repair Packets is used by a FECFRAME receiver to determine the current coding window size and `ew_max_size` by observing its maximum value over the time.

## Appendix D. Decoding Beyond Maximum Latency Optimization (Informational)

This annex introduces non normative considerations. It is provided as suggestions, without any impact on interoperability. For more information see [Roca16].

With a real-time source ADU flow, it is possible to improve the decoding performance of sliding window codes without impacting maximum latency, at the cost of extra memory and CPU overhead. The optimization consists, for a FECFRAME receiver, to extend the linear system beyond the decoding window maximum size, by keeping a certain number of old source symbols whereas their associated ADUs timed-out:

$$ls\_max\_size > dw\_max\_size$$

Usually the following choice is a good trade-off between decoding performance and extra CPU overhead:

```
ls_max_size = 2 * dw_max_size
```

When the `dw_max_size` is very small, it may be preferable to keep a minimum `ls_max_size` value (e.g., `LS_MIN_SIZE_DEFAULT = 40` symbols). Going below this threshold will not save a significant amount of memory nor CPU cycles. Therefore:

```
ls_max_size = max(2 * dw_max_size, LS_MIN_SIZE_DEFAULT)
```

Finally, it is worth noting that a receiver that benefits from an FEC protection significantly higher than what is required to recover from packet losses, can choose to reduce the `ls_max_size`. In that case lost ADUs will be recovered without relying on this optimization.

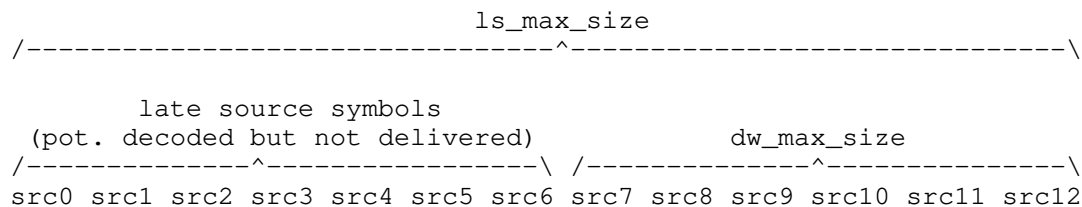


Figure 13: Relationship between parameters to decode beyond maximum latency.

It means that source symbols, and therefore ADUs, may be decoded even if the added latency exceeds the maximum value permitted by the application (the "late source symbols" of Figure 13). It follows that the corresponding ADUs will not be useful to the application. However, decoding these "late symbols" significantly improves the global robustness in bad reception conditions and is therefore recommended for receivers experiencing bad communication conditions [Roca16]. In any case whether or not to use this optimization and what exact value to use for the `ls_max_size` parameter are local decisions made by each receiver independently, without any impact on the other receivers nor on the source.

#### Authors' Addresses

Vincent Roca  
INRIA  
Univ. Grenoble Alpes  
France

EMail: [vincent.roca@inria.fr](mailto:vincent.roca@inria.fr)



Belkacem Teibi  
INRIA  
Univ. Grenoble Alpes  
France

EMail: [belkacem.teibi@gmail.com](mailto:belkacem.teibi@gmail.com)

Internet Engineering Task Force  
Internet-Draft  
Intended status: Informational  
Expires: January 3, 2019

N. Kuhn, Ed.  
CNES  
E. Lochin, Ed.  
ISAE-SUPAERO  
July 2, 2018

Network coding and satellites  
draft-kuhn-nwcrg-network-coding-satellites-05

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 January 3, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
1.1. Glossary . . . . .	3
1.2. Requirements Language . . . . .	3
2. A note on satellite topology . . . . .	4
3. Status of network coding in actually deployed satellite systems . . . . .	6
4. Details on the use cases . . . . .	6
4.1. Two way relay channel mode . . . . .	7
4.2. Reliable multicast . . . . .	7
4.3. Hybrid access . . . . .	8
4.4. Dealing with varying capacity . . . . .	9
4.5. Improving the gateway handovers . . . . .	10
4.6. Delay/Disruption Tolerant Networks . . . . .	10
5. Discussion on the deployability . . . . .	11
6. Conclusion . . . . .	12
7. Acknowledgements . . . . .	12
8. Contributors . . . . .	12
9. IANA Considerations . . . . .	12
10. Security Considerations . . . . .	12
11. References . . . . .	13
11.1. Normative References . . . . .	13
11.2. Informative References . . . . .	13
Authors' Addresses . . . . .	16

## 1. Introduction

Guaranteeing both physical layer robustness and efficient usage of the radio resource has been in the core design of SATellite COMMunication (SATCOM) systems. The trade-off often resided in how much redundancy a system had to add to cope from link impairments, without reducing the good-put when the channel quality is high. Generally speaking, enough redundancy is added so as to guarantee a Quasi-Error Free transmission; however, there are cases where the physical layer could hardly recover the transmission losses (e.g. with a mobile user) and layer 2 (or above) re-transmissions induce an at least 500 ms delay with a geostationary satellite. Further exploiting network coding schemes at higher OSI-layers is an opportunity for releasing constraints on the physical layer and improve the performance of SATCOM systems when the physical layer is challenged. We have noticed an active research activity on how network coding and SATCOM in the past. That being said, not much has actually made it to industrial developments. In this context, this memo aims at:

- o summing up the current deployment of network coding schemes over LEO and GEO satellite systems;

- o 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:

- o BBFRAME: Base-Band FRAME - satellite communication layer 2 encapsulation work as follows: (1) each layer 3 packet is encapsulated with a Generic Stream Encapsulation (GSE) mechanism, (2) GSE packets are gathered to create BBFRAMEs, (3) BBFRAMEs contain information related to how they have to be modulated (4) BBFRAMEs are forwarded to the physical layer;
- o CPE: Customer Premise Equipment;
- o DTN: Delay/Disruption Tolerant Network;
- o EPC: Evolved Packet Core;
- o ETSI: European Telecommunications Standards Institute;
- o PEP: Performance Enhanced Proxy - a typical PEP for satellite communications include compression, caching and TCP acceleration;
- o PLFRAME: Physical Layer FRAME - modulated version of a BBFRAME with additional information (e.g. related to synchronization);
- o SATCOM: generic term related to all kind of SATellite COMMunications systems;
- o UMTRAN: Universal Mobile Terrestrial Radio Access Network;
- o QoS: Quality-of-Service;
- o QoE: Quality-of-Experience;
- o VNF: Virtualized Network Function.

### 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 multi-gateway satellite system. More details on a generic SATCOM ground segment architecture for a bi-directional Internet access can be found in [SAT2017]. We propose a multi-gateway system since some of the use-cases described in this document require multiple gateways. In a multi-gateway system, some elements may be centralized and/or gathered: the relevance of one approach compared to another depends on the deployment scenario. More information on these trade-off discussions can be found in [SAT2017].

It is worth noting that some functional blocks aggregate the traffic coming from multiple users. Even if network coding schemes could be applied to any individual traffic, it could also work on a aggregate.

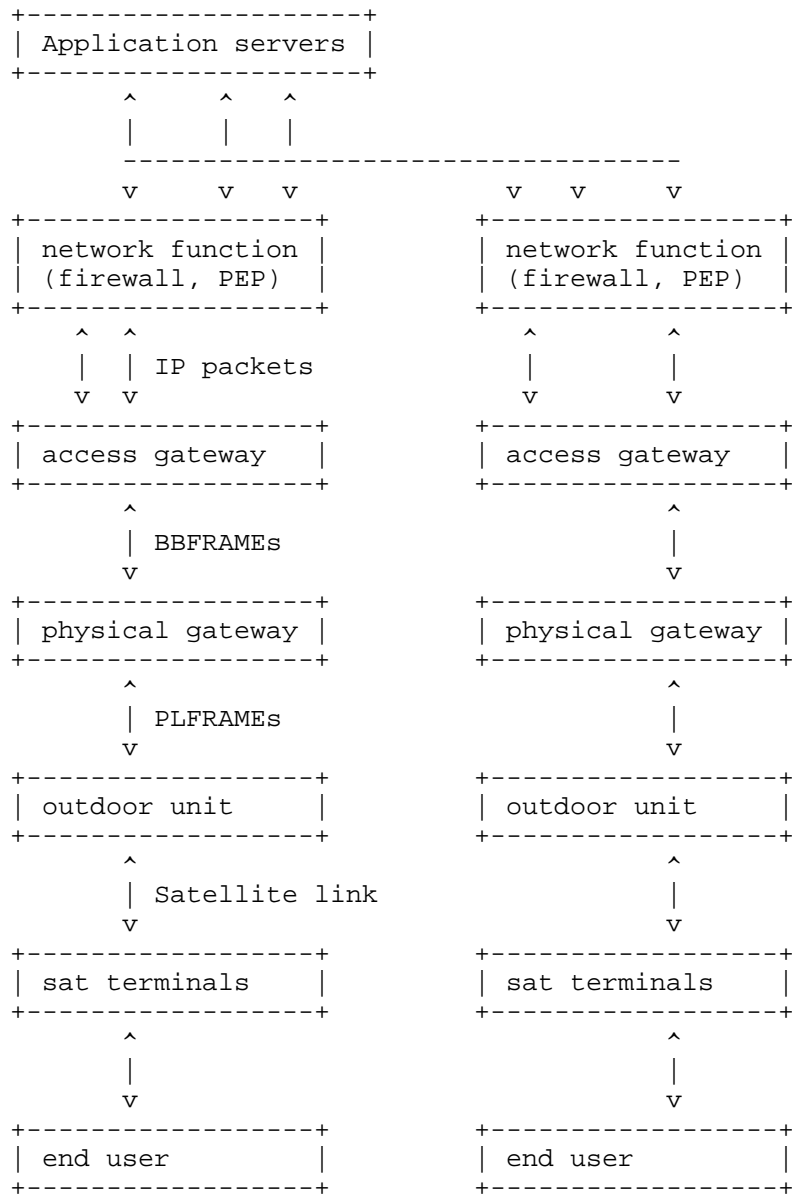


Figure 1: Data plane functions in a generic satellite multi-gateway system

### 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-nwcr-g-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.

	Upper Appl.	Middle ware	Communication layers	
	Source coding	Network AL-FEC	Packetization UDP/IP	PHY layer
E2E	X1			
NC				
IntraF	X1			
InterF				X2
SP	X1			X2
MP				

Figure 2: Network coding in current satellite systems

### 4. 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).

It is worth noting that these use-cases focus more on the middleware (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 multi-gateway 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.





-Li>- : packet indicated the loss of packet i of a multicast flow  
 = {M== : multicast flow including the missing packets

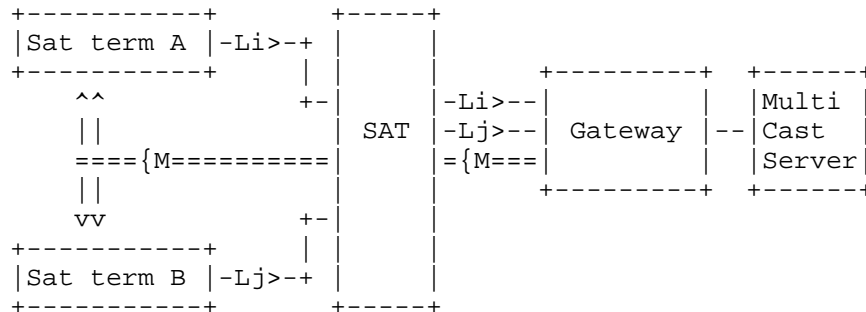


Figure 4: Network architecture for a reliable multicast with NC

#### 4.3. Hybrid access

This use-case considers the use of multiple path management with network coding at the transport level to increase the reliability and/or the total bandwidth (using multiple path does not guarantee an improvement of both the reliability and 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 and has been published as an ETSI Technical Report [ETSITR2017]. It is worth nothing that this kind of 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.

->- : bidirectional link

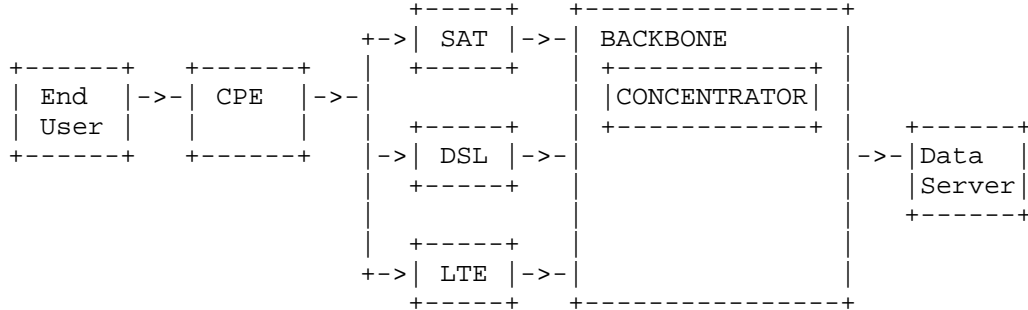


Figure 5: Network architecture for an hybrid access using NC

#### 4.4. 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. This is particularly relevant when end users are moving and the channel shows important variations [IEEEVT2001].

The architecture is recalled in Figure 6. The network coding schemes could be applied at the access gateway or the network function levels to increase the reliability of the transmission. This use-case is mostly relevant for 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.). Depending on the use-case (e.g. very high frequency bands, mobile users) or depending on the deployment use-cases (e.g. performance of the network between each individual block), the relevance of adding network coding is different. Then, depending on the OSI level at which network coding is applied, the impact on the satellite terminal is different: network coding may be applied on IP packets or on layer-2 proprietary format packets.

->- : bidirectional link

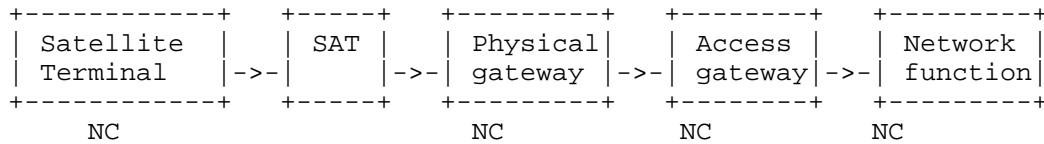


Figure 6: Network architecture for dealing with varying link characteristics with NC

#### 4.5. Improving the gateway handovers

This use-case considers the recovery of packets that may be lost during gateway handovers. Whether this is for off-loading one given equipment or because the transmission quality is not the same on each gateway, changing the transmission gateway may be relevant. However, if gateways are not properly synchronized, this may result in packet loss. During these critical phases, network coding can be added to improve the reliability of the transmission and propose a seamless gateway handover. In this case, the network coding could be applied at either the access gateway or the network function block. The entity responsible for taking the decision to change the communication gateway and changing the routes is the control plane manager; this entity exploits a management interface.

An example architecture for this use-case is showed in Figure 7. It is worth noting that depending on the ground architecture [I-D.chin-nfvrg-cloud-5g-core-structure-yang] [SAT2017], some equipment might be communalised.

->- : bidirectional link  
! : management interface

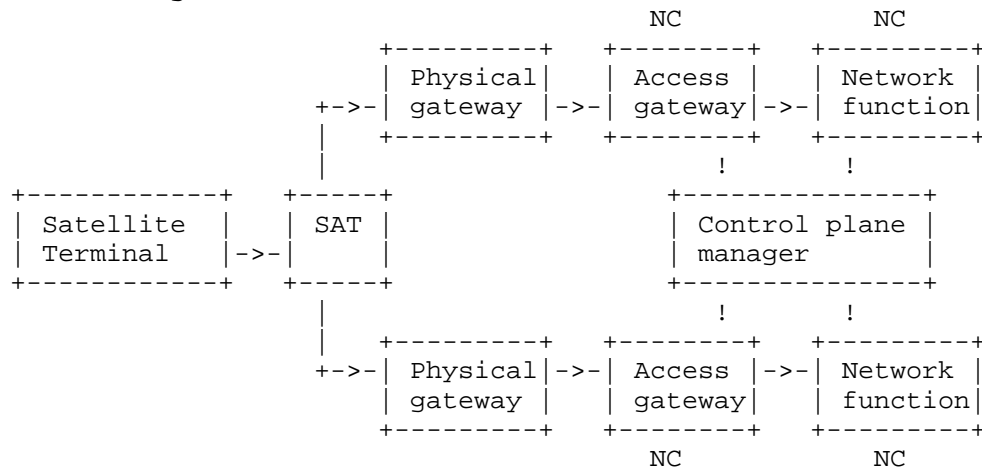


Figure 7: Network architecture for dealing with gateway handover schemes with NC

#### 4.6. Delay/Disruption Tolerant Networks

Establishing communications from terrestrial gateways to aerospace components is a challenge due to the distances involved. As a matter of fact, reliable end-to-end (E2E) communications over such links must cope with long delay and frequent link disruptions. Delay/

Disruption Tolerant Networking [RFC4838] is a solution to enable reliable internetworking space communications where both standard ad-hoc routing and E2E Internet protocols cannot be used. DTN can also be seen as an alternative solution to cope with satellite communications usually managed by PEP. Therefore, the transport of data over such networks requires the use of replication, erasure codes and multipath protocol schemes [WANG05] [ZHANG06] to improve the bundle delivery ratio and/or delivery delay. For instance, transport protocols such as LTP [RFC5326] for long delay links with connectivity disruptions, use Automatic Repeat-reQuest (ARQ) and unequal error protection to reduce the amount of non-mandatory re-transmissions. The work in [TOURNOUX10] proposed upon LTP a robust streaming method based on an on-the-fly coding scheme, where encoding and decoding procedures are done at the source and destination nodes, respectively. However, each link path loss rate may have various order of magnitude and re-encoding at an intermediate node to adapt the redundancy can be mandatory to prevent transmission wasting. This idea has been put forward in [I-D.zinky-dtnrg-random-binary-fec-scheme] and [I-D.zinky-dtnrg-erasure-coding-extension], where the authors proposed an encoding process at intermediate DTN nodes to explore the possibilities of Forward Error Correction (FEC) schemes inside the bundle protocol [RFC5050]. Another proposal is the use of erasure coding inside the CCSDS (Consultative Committee for Space Data Systems) architecture [COLA11]. The objective is to extend the CCSDS File Delivery Protocol (CFDP) [CCSDS-FDP] with erasure coding capabilities where a Low Density Parity Check (LDPC) [RFC6816] code with a large block size is chosen. Recently, on-the-fly erasure coding schemes [LACAN08] [SUNDARARAJAN08] [TOURNOUX11] have shown their benefits in terms of recovery capability and configuration complexity compared to traditional FEC schemes. Using a feedback path when available, on-the-fly schemes can be used to enable E2E reliable communication over DTN with adaptive re-encoding as proposed in [THAI15].

## 5. Discussion on the deployability

This section discusses the deployability of the use-cases detailed in Section 4.

SATCOM systems typically feature Performance Enhancement 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 4. 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].

## 6. Conclusion

This document 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.

Even if this document focuses on satellite systems, it is worth pointing out that the some scenarios proposed may be relevant to other wireless telecommunication systems. As one example, the generic architecture proposed in Figure 1 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.

## 7. Acknowledgements

Many thanks to Tomaso de Cola, Vincent Roca and Marie-Jose Montpetit.

## 8. Contributors

Tomaso de Cola, Marie-Jose Montpetit.

## 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>>.

### 11.2. Informative References

- [ASMS2010] De Cola, T. and et. al., "Demonstration at opening session of ASMS 2010", ASMS , 2010.
- [CCSDS-FDP] "CCSDS File Delivery Protocol, Recommendation for Space Data System Standards", CCSDS 727.0-B-4, Blue Book num. 3, 2007.
- [COLA11] De Cola, T., Paolini, E., Liva, G., and G. Calzolari, "Reliability options for data communications in the future deep-space missions", Proceedings of the IEEE vol. 99 issue 11, 2011.
- [ETSITR2017] "Satellite Earth Stations and Systems (SES); Multi-link routing scheme in hybrid access network with heterogeneous links", ETSI TR 103 351, 2017.
- [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-nwcr-g-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-nwcr-g-network-coding-taxonomy-08 (work in progress), March 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-  
nfvr-g-netcod-function-virtualization-02 (work in  
progress), November 2017.
- [I-D.zinky-dtnrg-erasure-coding-extension]  
Zinky, J., Caro, A., and G. Stein, "Bundle Protocol  
Erasure Coding Extension", draft-zinky-dtnrg-erasure-  
coding-extension-00 (work in progress), August 2012.
- [I-D.zinky-dtnrg-random-binary-fec-scheme]  
Zinky, J., Caro, A., and G. Stein, "Random Binary FEC  
Scheme for Bundle Protocol", draft-zinky-dtnrg-random-  
binary-fec-scheme-00 (work in progress), August 2012.
- [IEEEVT2001]  
Fontan, F., Vazquez-Castro, M., Cabado, C., Garcia, J.,  
and E. Kubista, "Statistical modeling of the LMS channel",  
BEER Transactions on Vehicular Technology vol. 50 issue 6,  
2001.
- [LACAN08] Lacan, J. and E. Lochin, "Rethinking reliability for long-  
delay networks", International Workshop on Satellite and  
Space Communications , October 2008.
- [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>>.
- [RFC4838] Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst,  
R., Scott, K., Fall, K., and H. Weiss, "Delay-Tolerant  
Networking Architecture", RFC 4838, DOI 10.17487/RFC4838,  
April 2007, <<https://www.rfc-editor.org/info/rfc4838>>.
- [RFC5050] Scott, K. and S. Burleigh, "Bundle Protocol  
Specification", RFC 5050, DOI 10.17487/RFC5050, November  
2007, <<https://www.rfc-editor.org/info/rfc5050>>.
- [RFC5326] Ramadas, M., Burleigh, S., and S. Farrell, "Licklider  
Transmission Protocol - Specification", RFC 5326,  
DOI 10.17487/RFC5326, September 2008,  
<<https://www.rfc-editor.org/info/rfc5326>>.

- [RFC5740] Adamson, B., Bormann, C., Handley, M., and J. Macker, "NACK-Oriented Reliable Multicast (NORM) Transport Protocol", RFC 5740, DOI 10.17487/RFC5740, November 2009, <<https://www.rfc-editor.org/info/rfc5740>>.
- [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>>.
- [RFC6816] Roca, V., Cunche, M., and J. Lacan, "Simple Low-Density Parity Check (LDPC) Staircase Forward Error Correction (FEC) Scheme for FECFRAME", RFC 6816, DOI 10.17487/RFC6816, December 2012, <<https://www.rfc-editor.org/info/rfc6816>>.
- [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.
- [SUNDARARAJAN08] Sundararajan, J., Shah, D., and M. Medard, "ARQ for network coding", IEEE Int. Symp. on Information Theory, July 2008.
- [THAI15] Thai, T., Chaganti, V., Lochin, E., Lacan, J., Dubois, E., and P. Gelard, "Enabling E2E reliable communications with adaptive re-encoding over delay tolerant networks", Proceedings of the IEEE International Conference on Communications, June 2015.
- [TOURNOUX10] Tournoux, P., Lochin, E., Leguay, J., and J. Lacan, "On the benefits of random linear coding for unicast applications in disruption tolerant networks", Proceedings of the IEEE International Conference on Communications, 2010.
- [TOURNOUX11] Tournoux, P., Lochin, E., Lacan, J., Bouabdallah, A., and V. Roca, "On-the-fly erasure coding for real-time video applications", IEEE Trans. on Multimedia vol. 13 issue 4, August 2011.
- [WANG05] Wang, Y. and et. al., "Erasure-coding based routing for opportunistic networks", Proceedings of the ACM SIGCOMM workshop on Delay-tolerant networking, 2005.



[ZHANG06] Zhang, X. and et. al., "On the benefits of random linear coding for unicast applications in disruption tolerant networks", Proceedings of the 4th International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks , 2006.

#### 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-nwcr-g-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.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
2. Terminology . . . . .	3
2.1. Definitions . . . . .	3
2.2. NDN/CCN Background . . . . .	5
3. Advantage given by NC and CCN/NDN . . . . .	6
4. Requirements . . . . .	7
4.1. Content Naming . . . . .	7
4.2. Transport . . . . .	8
4.2.1. Scope of Network Coding . . . . .	9
4.2.2. Consumer Operation . . . . .	9
4.2.3. Router Operation . . . . .	10
4.2.4. Publisher Operation . . . . .	11
4.3. In-network Caching . . . . .	11
4.4. Seamless Mobility . . . . .	12
4.5. Security and Privacy . . . . .	12
5. Challenges . . . . .	13
5.1. Adopting Convolutional Coding . . . . .	13
5.2. Rate and Congestion Control . . . . .	13
5.3. Security and Privacy . . . . .	14
5.4. Routing Scalability . . . . .	14
6. Security Considerations . . . . .	14
7. References . . . . .	14
7.1. Normative References . . . . .	14
7.2. Informative References . . . . .	14
Authors' Addresses . . . . .	17

## 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 heterogeneous 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].

- o Random Linear Coding (RLC): Particular case of Linear Coding using a set of random coding coefficients.
- o Generation, or (IETF) Block: With Block Codes, the set of content data that are logically grouped into a Block, before doing encoding.
- o Generation Size: With Block Codes, the number  $k$  of content data belonging to a Block.
- o 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
- o 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.
- o Finite Field size: The number of elements in a finite field. For example the binary extension field  $\{0..2^m-1\}$  has size  $q=2^m$ .
- o 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.
- o 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.
- o 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").
- o 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.

- o Consumer: A node requesting content. It initiates communication by sending an interest packets.
- o Publisher: A node providing content. It originally creates or owns the content.
- o Forwarding Information Base (FIB): A lookup table in a content router containing the name prefix and corresponding destination interface to forward the interest packets.
- o 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.
- o Content Store (CS): A storage space for a router to cache content objects. It is also known as in-network cache.
- o Content Object: A unit of content data delivered through the CCN/NDN network.
- o 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 be 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 sent 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., Bourtsoulatzé, 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. Turetti, "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-nwcrd-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. Multimed. 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](mailto:cedric.westphal@huawei.com)

NWCRG  
Internet-Draft  
Intended status: Informational  
Expires: January 3, 2019

V. Roca (Ed.)  
INRIA  
J. Detchart  
ISAE - Supaero  
C. Adjih  
INRIA  
M. Pedersen  
Steinwurf ApS  
July 2, 2018

Generic Application Programming Interface (API) for Sliding Window FEC  
Codes  
draft-roca-nwcrg-generic-fec-api-02

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). However, it 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 first 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 January 3, 2019.

## Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	3
2. Definitions and Abbreviations . . . . .	3
3. AL-FEC Codes and Mechanisms Considered by the Generic API . .	4
3.1. Mechanisms Considered or Ignored by the API . . . . .	5
4. Generic API for Sliding Window FEC Codes . . . . .	6
4.1. General Definitions Common to the Encoder and Decoder . .	6
4.2. Coding Window Functions at an Encoder and Decoder . . . .	7
4.3. Coding Coefficients Functions at an Encoder and Decoder .	10
4.4. Encoder . . . . .	12
4.5. Decoder . . . . .	15
5. Security Considerations . . . . .	20
6. IANA Considerations . . . . .	20
7. Acknowledgments . . . . .	20
8. References . . . . .	20
8.1. Normative References . . . . .	20
8.2. Informative References . . . . .	21
Appendix A. Existing APIs . . . . .	22
A.1. Morten API proposal . . . . .	22
A.1.1. Encoder . . . . .	22
A.1.2. Decoder . . . . .	25
A.2. Jonathan API proposal . . . . .	28
A.3. Cedric API proposal . . . . .	33
A.4. Vincent API proposal . . . . .	34
A.4.1. General . . . . .	34
A.4.2. Session Management . . . . .	35
A.4.3. Callback Functions . . . . .	37
A.4.4. Coding window functions . . . . .	38
A.4.5. Coding coefficients functions . . . . .	38
A.4.6. Encoder specific functions . . . . .	40
A.4.7. Decoder specific functions . . . . .	40
Authors' Addresses . . . . .	42

## 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 fundamental 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. Additionally, 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:

- o 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;
- o 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;
- o 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;
- o 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;
- o 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);
- o fixed code rate or not FEC codes, including rateless codes where the number of repair symbols that can be generated is huge (in theory unlimited);
- o 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 1 repair symbols may be required to recover all the 1 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) (Figure 1). 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.

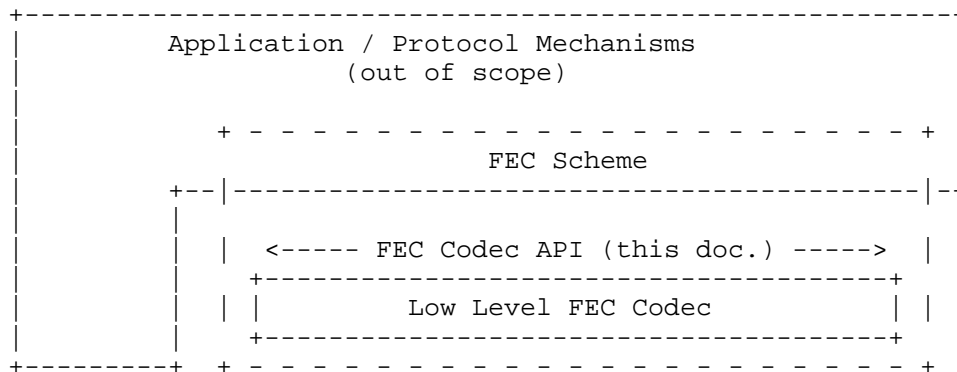


Figure 1: Position of the FEC Codec API with respect to the low level FEC Codec, the FEC Scheme, the protocol and other caller services.

### 3.1. Mechanisms Considered or Ignored by the API

Applying FEC coding, through an FEC Scheme, in a given protocol to improve transmission robustness involves many mechanisms. However, these mechanisms are not all the responsibility of the codec and can be implemented within the application or within the protocol that uses this FEC 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:

- o memory management;
- o packet transmission and reception;
- o signaling header creation / parsing;
- o ADU to source symbol mapping;
- o code rate adjustment, for instance thanks to the knowledge of losses at a receiver via feedbacks;
- o selective ACK creation and parsing;
- o congestion control.

The following mechanisms are **\*\*within scope of the API\*\***:

- o session management (sender and receiver);
- o encoding window management (sender and receiver);
- o set/get/generate coding coefficients (sender and receiver);
- o build coded symbol (sender only);
- o decode with newly received source or repair symbol (receiver only);



#### 4. Generic API for Sliding Window FEC Codes

The following sections describe the generic API, following a C-language formalism. Everything is prefixed by "ga" (Generic API).

##### 4.1. General Definitions Common to the Encoder and Decoder

This section gathers general definitions that are used by both an encoder and decoder.

About FEC Codepoints:

An application first needs to negotiate with its remote side the right FEC Scheme to use. This negotiation usually relies on the FEC Encoding ID associated to this FEC Scheme for this application. The FEC Encoding ID space, associated to an IANA registry, is protocol specific and the same value are usually associated to different FEC Schemes depending on the protocol. The FEC Encoding ID, from the Generic FEC API point of view, cannot be used to uniquely identify the codec. The use of a codepoint to identify locally the right FEC codec requires that the application knows a mapping between the FEC Encoding ID it uses and the local FEC Codepoints corresponding to available codecs. This can be done at development time, after including the Generic FEC API header file, which gives access to the `ga_codepoint_t` enumeration.

<CODE BEGINS>

```
/**
 * Return value of any function.
 *
 * GA_STATUS_OK = 0          Success
 * GA_STATUS_FAILURE        Failure. The function called did not succeed to
 *                           perform its task, however this is not an error
 *                           (e.g., it happens when decoding fails).
 * GA_STATUS_ERROR          Generic error type. The detailed error type is
 *                           stored in the errno variable of ga_encoder_t and
 *                           ga_decoder_t structures.
 */
typedef enum {
    GA_STATUS_OK = 0,
    GA_STATUS_FAILURE,
    GA_STATUS_ERROR
} ga_status_t;

/**
 * Potential errors.
```

```

*/
typedef enum {
    GA_ERRNO_NIL = 0,
    GA_ERRNO_CODEPOINT_NOT_SUPPORTED,
    /* and many more... */
} ga_errno_t;

/**
 * FEC Codepoints.
 * These identifiers are opaque identifiers that fully identify an FEC
 * code locally, including certain parameters like its Galois Field, or
 * the coding coefficient generator (if several exist).
 * These codepoints are codec specific and only have a local meaning.
 * They should not be transmitted as different implementations may use
 * them inconsistently.
 * Note that the same FEC code may be used by several FEC Encoding IDs
 * and therefore share the same codepoint. On the opposite multiple
 * implementations of a given FEC code may exist locally, for instance
 * with different optimizations, and then several codepoints, one per
 * codec, will exist for the same FEC code. The following names are
 * therefore only provided as examples.
 */
typedef enum {
    GA_NULL_CODEPOINT = 0, /* codepoint 0 is reserved */
    /* codepoint for RLC sliding window code, GF(2^8) and variable
     * density (as in FECFRAME FEC Enc. ID XXX). */
    GA_RLC_GF_256_VAR_DENSITY_CODEDC,
    /* codepoint for RLC sliding window code, GF(2) and variable
     * density (as in FECFRAME FEC Enc. ID YYY). */
    GA_RLC_GF_2_VAR_DENSITY_CODEDC,
    /* list here other identifiers for any FEC codec of interest */
} ga_codepoint_t;
<CODE ENDS>

```

General definitions.

#### 4.2. Coding Window Functions at an Encoder and Decoder

This section gathers functions used to manage the coding window, both at an encoder and at a decoder. At an encoder a sliding (of fixed or elastic size) encoding window is managed. Whenever a repair symbol needs to be created, a linear combination (that is code specific) of source symbols currently in the encoding window is performed. This encoding window is managed with the functions below plus, potentially, internal mechanisms that are code specific.

At a decoder, before submitting a new repair symbol to the codec, the application must specify the associated encoding window used at the source. This is done by the reset/add a single or set of symbols/remove a symbol functions. Once this coding window is ready, as well as the coding coefficient list if applicable, the application calls the `decode_with_new_repair_symbol()` function. A coding window may be reused for several repair symbols as long as they are all built from the same set of source symbols. In that case resetting the coding window and setting it from scratch would be a waste of time. The coding window must be viewed as a temporary list used solely by the `decode_with_new_repair_symbol()` function and kept independent from the linear system managed by the codec.

<CODE BEGINS>

```
/**
 * This function resets the current coding window. We assume here that
 * this window is maintained by the FEC codec instance.
 * Encoder:      reset the encoding window for the encoding of future
 *                repair symbols.
 * Decoder:      reset the coding window under preparation associated to
 *                a repair symbol just received.
 *
 * @return
 */
ga_status_t      ga_encoder_reset_coding_window (ga_encoder_t*   enc);
ga_status_t      ga_decoder_reset_coding_window (ga_encoder_t*   dec);

/**
 * Add a sequential set of source symbols (there MUST NOT be any gap) to
 * the coding window. This function may be called several times if there
 * are gaps in the encoding window. Calling this function does not reset
 * the current coding window, but appends these source symbols to it.
 *
 * Encoder:      add this sequential set of source symbols to the encoding
 *                window. The pointers in the table MUST point to the
 *                corresponding source symbol buffers.
 * Decoder:      add this source symbol set to the coding window under
 *                preparation.
 *
 * @param new_src_symbol_buf_tab (Encoder only) table of pointers to
 *                buffers containing each source symbol. The application
 *                MUST NOT free nor modify these buffers as long as the
 *                corresponding source symbol is in the encoding window.
 * @param first_src_symbol_esi   ESI of the first source symbol of the table
 * @param nb_symbols_in_tab      total number of symbols in this table.
 * @return
 */
```

```
*/
ga_status_t      ga_encoder_add_source_symbol_tab_to_coding_window (
                                ga_encoder_t*   enc,
                                void*           new_src_symbol_buf_tab[],
                                UINT32          first_src_symbol_esi,
                                UINT32          nb_symbols_in_tab);

ga_status_t      ga_decoder_add_source_symbol_tab_to_coding_window (
                                ga_decoder_t*   dec,
                                UINT32          first_src_symbol_esi,
                                UINT32          nb_symbols_in_tab);

/**
 * Add this source symbol to the coding window.
 * Encoder:      add a source symbol to the coding window.
 * Decoder:      add a source symbol to the coding window under preparation.
 *
 * @param new_src_symbol_buf      (encoder only) pointer to a buffer
 *                                containing the source symbol. The application MUST NOT
 *                                free nor modify this buffer as long as the source symbol
 *                                is in the coding window.
 * @param new_src_symbol_esi      ESI of the source symbol to add.
 * @return
 */
ga_status_t      ga_encoder_add_source_symbol_to_coding_window (
                                ga_encoder_t*   enc,
                                void*           new_src_symbol_buf,
                                UINT32          new_src_symbol_esi);

ga_status_t      ga_decoder_add_source_symbol_to_coding_window (
                                ga_decoder_t*   dec,
                                UINT32          new_src_symbol_esi);

/**
 * Remove this source symbol from the coding window.
 *
 * Encoder:      remove a source symbol from the encoding window, e.g.
 *                because the application knows that a source symbol has
 *                been acknowledged by the peer (if applicable). Note that
 *                the left side of the sliding window is automatically
 *                managed by the codec and no action is needed from the
 *                application. If needed a callback is available to inform
 *                the application that a source symbol has been removed).
 * Decoder:      remove a source symbol from the coding window under
 *                preparation.
 */
```

```

* @param old_src_symbol_esi    ESI of the source symbol to remove from
*                               the coding window.
* @return
*/
ga_status_t    ga_encoder_remove_source_symbol_from_coding_window (
                                ga_encoder_t*    enc,
                                UINT32           old_src_symbol_esi);

ga_status_t    ga_decoder_remove_source_symbol_from_coding_window (
                                ga_decoder_t*    dec,
                                UINT32           old_src_symbol_esi);

<CODE ENDS>

```

Coding Window Functions at an Encoder and Decoder.

#### 4.3. Coding Coefficients Functions at an Encoder and Decoder

This section gathers functions used to manage the coding coefficients, both at an encoder and at a decoder. Since different FEC codecs will have different requirements, it is important to keep these functions separate from the `build_repair_symbol()` and `decode_with_new_repair_symbol()` functions. Several situations exist:

- o the application provides the list of coding coefficients to use for the next `build_repair_symbol()`;
- o the application provides a key (typically a PRNG seed) that the codec uses to produce the coding coefficients to use for the next `build_repair_symbol()`;
- o the choice of the coding coefficients is totally performed by the codec, in an autonomous manner (e.g., the codec includes an algorithm that produces an appropriate seed based on various criteria, or the codec selects a set of coding coefficients based on various criteria). In that case the application needs to retrieve the list of coding coefficients or the key selected by the codec;

```

<CODE BEGINS>
/**
* The following functions enable an encoder (resp. decoder) to
* initialize the set of coefficients to be used for encoding
* or associated to a received repair symbol.
*
* Encoder: calling one of them MUST be done before calling
*           build_repair_symbol().
* Decoder: calling one of them MUST be done before calling
*           decode_with_new_repair_symbol().
*/

```

```

/**
 * Encoder: this function specifies the coding coefficients chosen by
 * the application if this is the way the codec works.
 * Decoder: communicate with this function the coding coefficients
 * associated to a repair symbol and carried in the packet header.
 *
 * @param coding_coefs_tab
 * (IN) table of coding coefficients associated to each of
 * the source symbols currently in the encoding window.
 * The size (number of bits) of each coefficient depends on
 * the FEC Scheme. The allocation and release of this table
 * is under the responsibility of the application.
 * @param nb_coefs_in_tab
 * (IN) number of entries (i.e., coefficients) in the table.
 * @return
 */
ga_status_t ga_encoder_set_coding_coefs_tab (
                                ga_encoder_t*   enc,
                                void*            coding_coefs_tab,
                                uint32_t         nb_coefs_in_tab);

ga_status_t ga_decoder_set_coding_coefs_tab (
                                ga_decoder_t*   dec,
                                void*            coding_coefs_tab,
                                uint32_t         nb_coefs_in_tab);

/**
 * The coding coefficients may be generated in a deterministic manner,
 * for instance by a PRNG known by the codec and a seed (perhaps with
 * other parameters) provided by the application.
 * The codec may also choose in an autonomous manner these coefficients.
 * This function is used to trigger this process.
 * When the choice is made in an autonomous manner, the actual coding
 * coefficient or key used by the codec can be retrieved with
 * ga_encoder_get_coding_coefs_tab().
 *
 * @param key (IN) Value that can be used as a seed in case of a PRNG
 * for instance, or by a specific coding coefficients
 * function. Set to 0 if not required by a codec.
 * @param add_param
 * (IN) an opaque 32-bit integer that contains a codec
 * specific parameter if needed. Set to 0 if not used.
 * @return
 */
ga_status_t ga_encoder_generate_coding_coefs (
                                ga_encoder_t*   enc,
                                uint32_t         key,

```

```

                                uint32_t      add_param);

ga_status_t      ga_decoder_generate_coding_coefs (
                                ga_decoder_t*   dec,
                                uint32_t        key,
                                uint32_t        add_param);

/**
 * This function enables the application to retrieve the set of coding
 * coefficients generated and used by build_repair_symbol(). This is
 * useful when the choice of coefficients is performed by the codec in
 * an autonomous manner but needs to be sent in the repair packet header.
 * This function is only used by an encoder.
 *
 * @param coding_coefs_tab
 *      (OUT) pointer to a table of coding coefficients.
 *      The size (number of bits) of each coefficient depends on
 *      the FEC scheme. Upon return of this function, this table
 *      is allocated and filled with coefficient values. The
 *      release of this table is under the responsibility of the
 *      application.
 * @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 zero.
 *      Upon return of this function this variable is initialized
 *      with the actual number of entries in the coeffs_tab[].
 * @return
 */
ga_status_t      ga_encoder_get_coding_coefs_tab (
                                ga_encoder_t*   enc,
                                void**          coding_coefs_tab,
                                uint32_t*        nb_coefs_in_tab);

<CODE ENDS>

```

Coding Coefficients Functions at an Encoder and Decoder.

#### 4.4. Encoder

```

<CODE BEGINS>
/**
 * Encoder structure that contains whatever is needed for encoding.
 * The exact content of this structure is FEC code dependent, the
 * structure below being a non normative example.
 */
typedef struct ga_encoder {
    /* desired verbosity: 0 is the minimum verbosity, the maximum

```

```
    /* level being implementation specific. */
    uint32_t      verbosity;

    /* when a function returns with GA_STATUS_ERROR, the errno
    * variable contains a more detailed error type. */
    ga_errno_t    errno;

    /* maximum number of source symbols used for any repair symbol */
    uint32_t      max_coding_window_size;

    /* exact size (in bytes) of any source or repair symbol */
    uint32_t      symbol_size;

    /* add whatever may be needed hereafter... */
} ga_encoder_t;

/**
 * Create and initialize an encoder, providing only key parameters.
 *
 * @param codepoint      opaque identifier that fully identifies the FEC
 *                        code to use.
 * @param verbosity      print information on the codec processing.
 *                        0 is the minimum verbosity, the maximum verbosity
 *                        level being implementation specific.
 * @param symbol_size    source and repair symbol size in bytes. Cannot
 *                        change during the codec instance lifetime.
 * @param max_encoding_window_size
 * @return               pointer to a ga_encoder_t structure if okay, or
 *                        NULL in case of error.
 */
ga_encoder_t* ga_encoder_create (
                                ga_codepoint_t  codepoint,
                                uint32_t        verbosity,
                                uint32_t        symbol_size,
                                uint32_t        max_coding_window_size);

/**
 * Release an encoder and its associated ressources.
 */
ga_status_t ga_encoder_release (ga_encoder_t*  enc);

/**
 * Set the various callback functions for this encoder.
 * All the callback functions require an opaque context parameter, that must be
 * initialized accordingly by the application, since it is application specific.

```



```
*
* @param enc
* @param source_symbol_removed_from_coding_window_callback
*      (IN) Pointer to the function, within the application, that
*      needs to be called each time a source symbol is removed from
*      the left side of the coding window.
*      This callback is called each time the encoding window slides
*      to the right and an old source symbol needs to be removed on
*      the left. The application therefore knows this source symbol
*      will no longer be used by the codec and can free the
*      associated buffer if need be. This function does not return
*      anything.
* @param context_4_callback
*      (IN) Pointer to the application-specific context that will be
*      passed to the callback function (if any). This context is not
*      interpreted by this function.
* @return
*/
ga_status_t ga_encoder_set_callback_functions (
    ga_encoder_t*   enc,
    void (*source_symbol_removed_from_coding_window_callback) (
        void*       context,
        uint32_t     old_symbol_esl),
    void* context_4_callback);

/**
* This function sets one or more FEC codec specific parameters,
* using a type/length/value approach for maximum flexibility.
*
* @param enc
* @param type      (IN) Type of parameter.
* @param length    (IN) length of the pointed value.
* @param value     (IN) Pointer to the value. The exact type of
*                  the object pointed is FEC codec specific.
* @return
*/
ga_status_t ga_encoder_set_parameters (
    ga_encoder_t*   enc,
    uint32_t        type,
    uint32_t        length,
    void*           value);

/**
* This function gets one or more FEC codec specific parameters,
* using a type/length/value approach for maximum flexibility.
*
* @param enc
* @param type      (IN) Type of parameter.
```

```

* @param length      (IN) length of the pointed value.
* @param value       (IN/OUT) Pointer to the value. The exact type of
*                   the object pointed is FEC codec specific.
*                   This function updates the value object
*                   accordingly. The caller, who knows the FEC codec,
*                   is responsible to allocate the appropriate
*                   object buffer.
* @return
*/
ga_status_t ga_encoder_get_parameters (
                                ga_encoder_t*   enc,
                                uint32_t        type,
                                uint32_t        length,
                                void*          value);

/**
 * List here the FEC codec specific control parameters.
 */
enum {
    GA_ENCODER_GET_PARAM_ENCODER_STATISTICS= 1,
    GA_ENCODER_SET_PARAM_RLC_DENSITY_THRESHOLD
};

/**
 * Create a single repair symbol (i.e. perform an encoding).
 *
 * @param new_buf      (IN) The pointer to the buffer for the repair
 *                   symbol to build can either point to a buffer
 *                   allocated by the application, or let to NULL
 *                   meaning that this function will allocate memory.
 * @return
 */
ga_status_t ga_build_repair_symbol (
                                ga_encoder_t*   enc,
                                void*          new_buf);
<CODE ENDS>

```

## Encoder API proposal

## 4.5. Decoder

```

<CODE BEGINS>
/**
 * Decoder structure that contains whatever is needed for decoding.
 * The exact content of this structure is FEC code dependent, the
 * structure below being a non normative example.
 */

```

```

typedef struct ga_decoder {
    /* desired verbosity: 0 is the minimum verbosity, the maximum
     * level being implementation specific. */
    uint32_t      verbosity;

    /* when a function returns with GA_STATUS_ERROR, the errno
     * variable contains a more detailed error type. */
    ga_errno_t    errno;

    /* maximum number of source symbols used for any repair symbol */
    uint32_t      max_coding_window_size;

    /* max. number of source symbols kepts in current linear system.
     * If the linear system grows above this limit, old source
     * symbols in excess are removed and the application callback
     * called. This value should be larger than the
     * max_coding_window_size. */
    uint32_t      max_linear_system_size;

    /* exact size (in bytes) of any source or repair symbol */
    uint32_t      symbol_size;

    /* add whatever may be needed hereafter... */
} ga_decoder_t;

/**
 * Create and initialize a decoder, providing only key parameters.
 *
 * @param codepoint      opaque identifier that fully identifies the FEC
 *                        code to use.
 * @param verbosity      print information on the codec processing.
 *                        0 is the minimum verbosity, the maximum verbosity
 *                        level being implementation specific.
 * @param symbol_size    source and repair symbol size in bytes. Cannot
 *                        change during the codec instance lifetime.
 * @param max_coding_window_size
 * @param max_linear_system_size
 * @return               pointer to a ga_decoder_t structure if okay, or
 *                        NULL in case of error.
 */
ga_decoder_t* ga_decoder_create (
    ga_codepoint_t  codepoint,
    uint32_t        verbosity,
    uint32_t        symbol_size,
    uint32_t        max_coding_window_size,
    uint32_t        max_linear_system_size);

```

```
/**
 * Release a decoder and its associated resources.
 *
 * @param dec    context (i.e., pointer to decoder structure).
 */
ga_status_t ga_decoder_release (ga_decoder_t*    dec);

/**
 * Set the various callback functions for this decoder.
 * All the callback functions require an opaque context parameter, that
 * must be initialized accordingly by the application, since it is
 * application specific.
 *
 * @param dec    context (i.e., pointer to decoder structure).
 * @param source_symbol_removed_from_coding_window_callback
 *      (IN) Pointer to the function, within the application, that
 *      needs to be called each time a source symbol is removed from
 *      the left side of the coding window.
 *      This callback is called each time the decoding window slides
 *      to the right and an old source symbol needs to be removed
 *      on the left. This function does not return anything.
 * @param decoded_source_symbol_callback
 *      (IN) Pointer to the function, within the application, that
 *      needs to be called each time a source symbol is decoded.
 *      What it does is application-dependent, but it MUST return
 *      either a pointer to a data buffer, left uninitialized, of
 *      the appropriate size, or NULL if the application prefers to
 *      let the codec allocate the buffer.
 *      In any case the codec is responsible for storing the actual
 *      symbol value within the data buffer. Also, no matter
 *      whether the data buffer is allocated by the application or
 *      the codec, it is the responsibility of the application to
 *      free this buffer when needed, once decoding is over (but
 *      not before since the codec does not keep any internal copy).
 * @param available_source_symbol_callback
 *      (IN) Pointer to the function, within the application, that
 *      needs to be called each time a source symbol is decoded and
 *      all computations performed (i.e., the buffer does contain the
 *      symbol value).
 *      This callback is called in a second time, when the newly
 *      decoded source symbol is actually ready, i.e. when all the
 *      computations (like XOR and GF(2**8) operations) have been
 *      performed. In any case, it is the responsibility of the
 *      application to free this buffer when needed, once decoding
 *      is over (but not before since the codec does not keep any
 *      internal copy). This function does not return anything.
 * @param context_4_callback
```

```

*          (IN) Pointer to the application-specific context that will be
*          passed to the callback function (if any). This context is not
*          interpreted by this function.
* @return
*/
ga_status_t ga_decoder_set_callback_functions (
    ga_decoder_t* dec,
    void (*source_symbol_removed_from_coding_window_callback) (
        void* context,
        uint32_t old_symbol_esi),
    void* (*decoded_source_symbol_callback) (
        void* context,
        uint32_t esi),
    void (*available_source_symbol_callback) (
        void* context,
        void* new_symbol_buf,
        uint32_t esi),
    void* context_4_callback);

/**
 * This function sets one or more FEC codec specific parameters,
 * using a type/length/value approach for maximum flexibility.
 *
 * @param dec context (i.e., pointer to decoder structure).
 * @param type (IN) Type of parameter.
 * @param length (IN) length of the pointed value.
 * @param value (IN) Pointer to the value. The exact type of
 * the object pointed is FEC codec specific.
 * @return
 */
ga_status_t ga_decoder_set_parameters (
    ga_decoder_t* dec,
    uint32_t type,
    uint32_t length,
    void* value);

/**
 * This function gets one or more FEC codec specific parameters,
 * using a type/length/value approach for maximum flexibility.
 *
 * @param dec context (i.e., pointer to decoder structure).
 * @param type (IN) Type of parameter.
 * @param length (IN) length of the pointed value.
 * @param value (IN/OUT) Pointer to the value. The exact type of
 * the object pointed is FEC codec specific.
 * This function updates the value object
 * accordingly. The caller, who knows the FEC codec,

```

```
*           is responsible to allocate the appropriate
*           object buffer.
* @return
*/
ga_status_t ga_decoder_get_parameters (
                                ga_decoder_t*   dec,
                                uint32_t        type,
                                uint32_t        length,
                                void*           value);

/**
 * List here the FEC codec specific control parameters.
 */
enum {
    GA_DECODER_GET_PARAM_DECODER_STATISTICS = 1,
    GA_DECODER_SET_PARAM_RLC_DENSITY_THRESHOLD
};

/**
 * 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 functions.
 *
 * This function usually returns GA_STATUS_OK, regardless of whether this
 * new symbol enabled the decoding of one or several source symbols, or
 * GA_STATUS_ERROR. It cannot return GA_STATUS_FAILURE.
 *
 * @param dec    context (i.e., pointer to decoder structure).
 * @param new_symbol_buf
 *               (IN) Pointer to the new source symbol now available (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_esl
 *               (IN) encoding symbol ID of the new source symbol.
 * @return
 */
ga_status_t ga_decoder_decode_with_new_source_symbol (
                                ga_decoder_t*   dec,
                                void* const     new_symbol_buf,
                                uint32_t        new_symbol_esl);

/**
 * 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 functions.
 *
```

```

* This function requires that the application has previously initialized
* the coding window and coding coefficients appropriately. The application
* keeps a full control of the repair symbol buffer, i.e., the application
* is in charge of freeing this buffer as soon as it believes appropriate
* (a copy is kept by the codec). This is motivated by the fact that a
* repair symbol may be part of a larger buffer (e.g., if there are
* several repair symbols per packet, or because of a packet header): only
* the application knows when the buffer can be safely freed.
*
* This function usually returns GA_STATUS_OK, regardless of whether this
* new symbol enabled the decoding of one or several source symbols, or
* GA_STATUS_ERROR. It cannot return GA_STATUS_FAILURE.
*
* @param dec    context (i.e., pointer to decoder structure).
* @param new_symbol_buf
*              (IN) Pointer to the new repair symbol now available (i.e.
*              a new symbol received by the application or a decoded
*              symbol in case of a recursive call if it makes sense).
* @return
*/
ga_status_t    ga_decoder_decode_with_new_repair_symbol (
                                ga_decoder_t*    dec,
                                void* const      new_symbol_buf);
<CODE ENDS>

```

#### Decoder API proposal

#### 5. Security Considerations

TBD

#### 6. IANA Considerations

This document has no IANA requirement.

#### 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>>.

## 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), June 2018, <<https://tools.ietf.org/html/draft-ietf-tsvwg-fecframe-ext>>.

## [RLC]

Roca, V. and B. Teibi, "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), June 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

&lt;CODE BEGINS&gt;

```
// 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 encoder 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;

/**
* Init a sliding window encoder by giving a galois field size, a coefficient ge
nerator 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);

/**
* 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);

/**
* 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);

/**
* 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

Roca (Ed.), et al.

Expires January 3, 2019

[Page 30]

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

/**
 * 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);

/**
 * 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;
```

```

/**
 * Init a sliding window decoder by giving a galois field size, a coefficient ge
nerator 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);

/**
 * 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)
 */

/**
 * Decode some source data units by giving to the decoder a new source data 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);

/**
 * Decode some source data units by giving to the decoder a new repair data 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);

/**
 * 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/>>):

- o an API could be globally pretty similar;
- o 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.
- o 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-nwcrp-cisew-00>> for instance).
- o 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.
- o 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. Vincent API proposal

##### A.4.1. 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 specify
 * both the codes and way of using these codes), it is distinct from the FEC Encoding
 * 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 the current
 * codec instance, also known as "session".
 *
 * 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 specific
 * structures MUST begin the same entries as the ones provided in this generic structure, otherwise
 * 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 specific
 * structures MUST begin the same entries as the ones provided in this generic structure, otherwise
 * 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.4.2. 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
located 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'd 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.4.3. 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 contain the
*           symbol value).
*
* @param source_symbol_removed_from_coding_window_callback
*           (IN) Pointer to the function, within the application, that
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 slid 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 will be
ll be
*           passed to the callback function (if any). This context is
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),          /* encoding
symbol ID */
void (*available_source_symbol_callback) (void      *context,
void      *new_symbol_buf, /* symbol
ol buffer */
                                         UINT32      size,          /* size
of decoded source symbol */
                                         UINT32      esi),          /* encoding
symbol ID */
void (*source_symbol_removed_from_coding_window_callback)
                                         (void      *context,
                                         UINT32      old_symbol_esi),
void*          context_4_callback);
<CODE ENDS>

```

Vincent API proposal

#### A.4.4. Coding window functions

TBD

#### A.4.5. Coding coefficients functions

<CODE BEGINS>

```

/**
* SENDER:   this function specifies the coding coefficients chosen by the application 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 and carried in the packet header.
*           This function MUST be called before calling decode_with_new_repair_symbol

```

( ).

Roca (Ed.), et al.

Expires January 3, 2019

[Page 38]

```

*
* @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 containi
ng 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.4.6. 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.4.7. Decoder specific functions



&lt;CODE BEGINS&gt;

```

/**
 * 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 GA_STATUS_OK, regardless of whether this new sy
 * mbol enabled the
 * decoding of one or several source symbols, unless an error occurred. 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  
Univ. Grenoble Alpes  
France

EMail: [vincent.roca@inria.fr](mailto:vincent.roca@inria.fr)

Jonathan Detchart  
ISAE - Supaero  
France

EMail: [jonathan.detchart@isae-supaero.fr](mailto:jonathan.detchart@isae-supaero.fr)

Cedric Adjih  
INRIA  
France

EMail: [cedric.adjih@inria.fr](mailto:cedric.adjih@inria.fr)

Morten V. Pedersen  
Steinwurf ApS  
Denmark

EMail: [morten@steinwurf.com](mailto:morten@steinwurf.com)

NWCRG  
Internet-Draft  
Intended status: Informational  
Expires: May 20, 2020

V. Roca (Ed.)  
INRIA  
J. Detchart  
ISAE - Supaero  
C. Adjih  
INRIA  
M. Pedersen  
Steinwurf ApS  
November 17, 2019

Generic Application Programming Interface (API) for Sliding Window FEC  
Codes  
draft-roca-nwcrg-generic-fec-api-07

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). However, it 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 first 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 May 20, 2020.

## Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
2. Definitions and Abbreviations . . . . .	3
3. AL-FEC Codes and Mechanisms Considered by the Generic API . .	3
3.1. Mechanisms Considered or Ignored by the API . . . . .	5
4. Generic API for Sliding Window FEC Codes . . . . .	6
4.1. General Definitions Common to the Encoder and Decoder . .	6
4.2. Encoder . . . . .	9
4.3. Decoder . . . . .	13
4.4. Coding Window Functions at an Encoder and Decoder . . . .	17
4.5. Coding Coefficients Functions at an Encoder and Decoder .	19
5. Security Considerations . . . . .	22
6. IANA Considerations . . . . .	22
7. Acknowledgments . . . . .	23
8. References . . . . .	23
8.1. Normative References . . . . .	23
8.2. Informative References . . . . .	23
Authors' Addresses . . . . .	23

## 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 fundamental

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. Additionally, 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:

- o 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;
- o 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;
- o 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;
  - o 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;
  - o 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);
  - o fixed code rate or not FEC codes, including rateless codes where the number of repair symbols that can be generated is huge (in theory unlimited);
  - o 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 1 repair symbols may be required to recover all the 1 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) (Figure 1). 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.

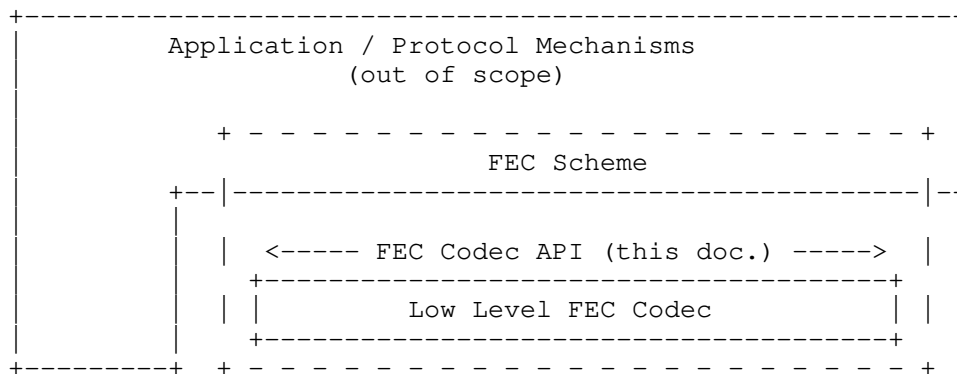


Figure 1: Position of the FEC Codec API with respect to the low level FEC Codec, the FEC Scheme, the protocol and other caller services.

### 3.1. Mechanisms Considered or Ignored by the API

Applying FEC coding, through an FEC Scheme, in a given protocol to improve transmission robustness involves many mechanisms. However, these mechanisms are not all the responsibility of the codec and can be implemented within the application or within the protocol that uses this FEC 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:

- o memory management;
- o packet transmission and reception;
- o signaling header creation / parsing;
- o ADU to source symbol mapping;
- o code rate adjustment, for instance thanks to the knowledge of losses at a receiver via feedbacks;
- o selective ACK creation and parsing;
- o congestion control.

The following mechanisms are **\*\*within scope of the API\*\***:

- o session management (sender and receiver);
- o encoding window management (sender and receiver);
- o set/get/generate coding coefficients (sender and receiver);
- o build coded symbol (sender only);
- o decode with newly received source or repair symbol (receiver only);



#### 4. Generic API for Sliding Window FEC Codes

The following sections describe the generic API, following a C-language formalism. This API tries to adhere to C99 version of C, although it may not strictly be guaranteed. Everything is prefixed by "swif" (sliding window FEC).

##### 4.1. General Definitions Common to the Encoder and Decoder

This section gathers general definitions that are used by both an encoder and decoder.

About FEC Codepoints:

An application first needs to negotiate with its remote side the right FEC Scheme to use. This negotiation usually relies on the FEC Encoding ID associated to this FEC Scheme for this application. A difficulty is that the FEC Encoding ID space, associated to an IANA registry, is protocol specific and the same value are usually associated to different FEC Schemes depending on the protocol. For instance, the FEC Encoding ID value 2 may be used for two totally different FEC Schemes in protocol A and protocol B. Therefore, the FEC Encoding ID, from the Generic FEC API point of view, cannot be used to uniquely identify the target codec.

The use of a codepoint to identify locally the right FEC codec requires that the application knows a mapping between the FEC Encoding ID it uses for a given protocol, and the local FEC Codepoints corresponding to available codecs. This will be done at development time, the FEC API header file giving access to the `swif_codepoint_t` enumeration with the list of all codecs available locally.

<CODE BEGINS>

```
/**
 * Return value of any function.
 *
 * SWIF_STATUS_OK = 0      Success
 * SWIF_STATUS_FAILURE    Failure. The function called did not succeed to
 *                          perform its task, however this is not an error
 *                          (e.g., it happens when decoding fails).
 * SWIF_STATUS_ERROR      Generic error type. The detailed error type is
 *                          stored in the errno variable of swif_encoder_t and
 *                          swif_decoder_t structures.
 */
typedef enum {
    SWIF_STATUS_OK = 0,
    SWIF_STATUS_FAILURE,
```

```
        SWIF_STATUS_ERROR
    } swif_status_t;

/**
 * Potential errors.
 */
typedef enum {
    SWIF_ERRNO_NULL = 0,                /* everything is fine */
    SWIF_ERRNO_UNSUPPORTED_CODEPOINT,
    /* and many more... */
} swif_errno_t;

/**
 * FEC Codepoints.
 * These identifiers are opaque identifiers that fully identify an FEC
 * code locally, including certain parameters like its Galois Field.
 * These codepoints are codec specific and only have a local meaning.
 * They should not be transmitted as different implementations may use
 * them inconsistently.
 * Note that the same FEC code may be used by several FEC Encoding IDs
 * and therefore share the same codepoint. On the opposite multiple
 * implementations of a given FEC code may exist locally, for instance
 * with different optimizations, and then several codepoints, one per
 * codec, will exist for the same FEC code. The following names are
 * therefore only provided as examples.
 */
typedef enum {
    SWIF_CODEPOINT_NULL = 0,            /* codepoint 0 is reserved */

    /* codepoint for sliding window codec AAA. */
    SWIF_CODEPOINT_AAA_CODEC,

    /* codepoint for sliding window codec BBB. */
    SWIF_CODEPOINT_BBB_CODEC,

    /* list here other identifiers for any codec of interest... */
} swif_codepoint_t;

/**
 * Encoding Symbol Identifier (ESI) generic type.
 * With Sliding Window FEC codes, an ESI is in fact a source symbol
 * identifier, unlike block FEC codes.
 */
typedef uint32_t      esi_t;
```

```
/**
 * Throughout the API, a pointer to this structure is used as an
 * identifier of the encoder instance (or "enc").
 *
 * This generic structure is meant to be extended by each codec with
 * new pieces of information that are specific to each codec.
 */
typedef struct swif_encoder {
    swif_codepoint_t    codepoint;

    /* when a function returns with SWIF_STATUS_ERROR, the errno
     * variable contains a more detailed error type. This variable
     * is set by the codec and accessible to the application in
     * READ ONLY mode. Otherwise its value is undefined. */
    swif_errno_t        swif_errno;

    /* pointers to codec specific versions of API functions. */
    swif_status_t    (*set_callback_functions) (
        struct swif_encoder*, void (*) (void*, esi_t), void*);
    swif_status_t    (*set_parameters) (
        struct swif_encoder*, uint32_t, uint32_t, void*);
    swif_status_t    (*get_parameters) (
        struct swif_encoder*, uint32_t, uint32_t, void*);
    swif_status_t    (*build_repair_symbol) (
        struct swif_encoder*, void*);
    swif_status_t    (*reset_coding_window) (struct swif_encoder*);
    swif_status_t    (*add_source_symbol_to_coding_window) (
        struct swif_encoder*, void*, esi_t);
    swif_status_t    (*remove_source_symbol_from_coding_window) (
        struct swif_encoder*, esi_t);
    swif_status_t    (*get_coding_window_information) (
        struct swif_encoder*, esi_t*, esi_t*, uint32_t*);
    swif_status_t    (*set_coding_coefs_tab) (
        struct swif_encoder*, void*, uint32_t);
    swif_status_t    (*generate_coding_coefs) (
        struct swif_encoder*, uint32_t, uint32_t);
    swif_status_t    (*get_coding_coefs_tab) (
        struct swif_encoder*, void**, uint32_t*);
} swif_encoder_t;

/**
 * Decoder structure that contains whatever is needed for decoding.
 * The exact content of this structure is FEC code dependent, the
 * structure below being a non normative example.
 */
typedef struct swif_decoder {
    swif_codepoint_t    codepoint;
```

```

/* when a function returns with SWIF_STATUS_ERROR, the errno
 * variable contains a more detailed error type. This variable
 * is set by the codec and accessible to the application in
 * READ ONLY mode. Otherwise its value is undefined. */
swif_errno_t          swif_errno;

/* pointers to codec specific versions of API functions. */
swif_status_t  (*set_callback_functions) (
    struct swif_decoder*, void (*) (void*, esi_t),
    void* (*) (void*, esi_t),
    void* (*) (void*, void*, esi_t), void*);
swif_status_t  (*set_parameters) (
    struct swif_decoder*, uint32_t, uint32_t, void*);
swif_status_t  (*get_parameters) (
    struct swif_decoder*, uint32_t, uint32_t, void*);
swif_status_t  (*decode_with_new_source_symbol) (
    struct swif_decoder*, void* const, esi_t);
swif_status_t  (*decode_with_new_repair_symbol) (
    struct swif_decoder*, void* const);
swif_status_t  (*reset_coding_window) (swif_encoder_t*);
swif_status_t  (*add_source_symbol_to_coding_window) (
    struct swif_decoder*, esi_t);
swif_status_t  (*remove_source_symbol_from_coding_window) (
    struct swif_decoder*, esi_t);
swif_status_t  (*set_coding_coefs_tab) (
    struct swif_decoder*, void*, uint32_t);
swif_status_t  (*generate_coding_coefs) (
    struct swif_decoder*, uint32_t, uint32_t);
} swif_decoder_t;
<CODE ENDS>

```

General definitions.

#### 4.2. Encoder

```

<CODE BEGINS>
/**
 * Create and initialize an encoder, providing only key parameters.
 *
 * @param codepoint      opaque identifier that fully identifies the FEC
 *                        code to use.
 * @param verbosity      print information on the codec processing.
 *                        0 is the minimum verbosity, the maximum verbosity
 *                        level being implementation specific.
 * @param symbol_size    source and repair symbol size in bytes. Cannot
 *                        change during the codec instance lifetime.
 * @param max_encoding_window_size
 * @return               pointer to a swif_encoder_t structure if okay, or

```

```
*          NULL in case of error.
**/
swif_encoder_t* swif_encoder_create (
                                swif_codepoint_t codepoint,
                                uint32_t          verbosity,
                                uint32_t          symbol_size,
                                uint32_t          max_coding_window_size);

/**
 * Release an encoder and its associated ressources.
 **/
swif_status_t  swif_encoder_release (swif_encoder_t*      enc);

/**
 * Set the various callback functions for this encoder.
 * All the callback functions require an opaque context parameter, that
 * must be initialized accordingly by the application, since it is
 * application specific.
 *
 * @param enc
 * @param source_symbol_removed_from_coding_window_callback
 *      (IN) Pointer to the function, within the application,
 *      that needs to be called each time a source symbol is
 *      removed from the left side of the coding window.
 *      This callback is called each time the encoding window
 *      slides to the right and an old source symbol needs to
 *      be removed on the left. The application therefore knows
 *      this source symbol will no longer be used by the codec
 *      and can free the associated buffer if need be. This
 *      function does not return anything.
 * @param context_4_callback
 *      (IN) Pointer to the application-specific context that
 *      will be passed to the callback function (if any). This
 *      context is not interpreted by this function.
 * @return
 */
swif_status_t  swif_encoder_set_callback_functions (
                                swif_encoder_t*      enc,
                                void (*source_symbol_removed_from_coding_window_callback) (
                                                void*      context,
                                                esi_t      old_symbol_esi),
                                void* context_4_callback);

/**
 * This function sets one or more FEC codec specific parameters,
 * using a type/length/value approach for maximum flexibility.

```

```

*
* @param enc
* @param type      (IN) Type of parameter.
* @param length    (IN) length of the pointed value.
* @param value     (IN) Pointer to the value. The exact type of
*                  the object pointed is FEC codec specific.
* @return
*/
swif_status_t swif_encoder_set_parameters (
                                swif_encoder_t* enc,
                                uint32_t      type,
                                uint32_t      length,
                                void*         value);

/**
* This function gets one or more FEC codec specific parameters,
* using a type/length/value approach for maximum flexibility.
*
* @param enc
* @param type      (IN) Type of parameter.
* @param length    (IN) length of the pointed value.
* @param value     (IN/OUT) Pointer to the value. The exact type of
*                  the object pointed is FEC codec specific.
*                  This function updates the value object
*                  accordingly. The caller, who knows the FEC codec,
*                  is responsible to allocate the appropriate
*                  object buffer.
* @return
*/
swif_status_t swif_encoder_get_parameters (
                                swif_encoder_t* enc,
                                uint32_t      type,
                                uint32_t      length,
                                void*         value);

/**
* List here the FEC codec specific control parameters.
*/
enum {
    swif_ENCODER_GET_PARAM_ENCODER_STATISTICS = 1,
    swif_ENCODER_SET_PARAM_RLC_DENSITY_THRESHOLD
};

/**
* Create a single repair symbol (i.e. perform an encoding).
* Upon return of this function, the application has full control of the
* buffer and is in charge of freeing it when appropriate.

```

```

*
* @param new_buf      (IN) The pointer to the buffer for the repair
*                      symbol to build can either point to a buffer
*                      allocated by the application and initialized to
*                      zero, or let to NULL meaning that this function
*                      will allocate memory.
* @return
*/
swif_status_t  swif_build_repair_symbol (
                                swif_encoder_t* enc,
                                void*          new_buf);
/* FIX ME: must be void** to enable returning a pointer to buffer! */
<CODE ENDS>

```

## Encoder API proposal

```

<CODE BEGINS>
/**
 * Encoder structure that contains whatever is needed for encoding.
 * The exact content of this structure is FEC code dependent, the
 * structure below being a non normative example.
 * However it MUST be aligned with swif_encoder_t (same first items) in
 * order to be able to cast a pointer to one of the two structures,
 * depending on the context.
 */
typedef struct swif_encoder_internal {
    /* generic part of any control block. MUST be first in structure */
    swif_encoder_t  gen;

    /* desired verbosity: 0 is the minimum verbosity, the maximum
     * level being implementation specific. */
    uint32_t        verbosity;

    /* maximum number of source symbols used for any repair symbol */
    uint32_t        max_coding_window_size;

    /* exact size (in bytes) of any source or repair symbol */
    uint32_t        symbol_size;

    /* add whatever may be needed hereafter... */
} swif_encoder_internal_t;

```

Non normative example of internal structure used by an encoder.

## 4.3. Decoder

&lt;CODE BEGINS&gt;

```

/**
 * Create and initialize a decoder, providing only key parameters.
 *
 * @param codepoint      opaque identifier that fully identifies the FEC
 *                        code to use.
 * @param verbosity      print information on the codec processing.
 *                        0 is the minimum verbosity, the maximum verbosity
 *                        level being implementation specific.
 * @param symbol_size    source and repair symbol size in bytes. Cannot
 *                        change during the codec instance lifetime.
 * @param max_coding_window_size
 * @param max_linear_system_size
 * @return               pointer to a swif_decoder_t structure if okay, or
 *                        NULL in case of error.
 */
swif_decoder_t* swif_decoder_create (
                                swif_codepoint_t codepoint,
                                uint32_t         verbosity,
                                uint32_t         symbol_size,
                                uint32_t         max_coding_window_size,
                                uint32_t         max_linear_system_size);

/**
 * Release a decoder and its associated ressources.
 *
 * @param dec            context (i.e., pointer to decoder structure).
 */
swif_status_t  swif_decoder_release (swif_decoder_t*      dec);

/**
 * Set the various callback functions for this decoder.
 * All the callback functions require an opaque context parameter, that
 * must be initialized accordingly by the application, since it is
 * application specific.
 *
 * @param dec            context (i.e., pointer to decoder structure).
 * @param source_symbol_removed_from_linear_system_callback
 *                        (IN) Pointer to the function, within the application, that
 *                        needs to be called each time a source symbol is removed from
 *                        the left side of the linear system.
 *                        This callback is called each time the linear system slides
 *                        to the right and an old source symbol needs to be removed
 *                        on the left. This function does not return anything.

```



```

* @param decodable_source_symbol_callback
*      (IN) Pointer to the function, within the application, that
*      needs to be called each time a source symbol is decodable.
*      What it does is application-dependent, but it MUST return
*      either a pointer to a data buffer, left uninitialized, of
*      the appropriate size, or NULL if the application prefers to
*      let the codec allocate the buffer.
*      In any case the codec is responsible for storing the actual
*      symbol value within the data buffer. Also, no matter
*      whether the data buffer is allocated by the application or
*      the codec, it is the responsibility of the application to
*      free this buffer when needed, once decoding is over (but
*      not before since the codec does not keep any internal copy).
* @param decoded_source_symbol_callback
*      (IN) Pointer to the function, within the application, that
*      needs to be called each time a source symbol is decodable and
*      all computations performed (i.e., the buffer does contain the
*      symbol value).
*      This callback is called in a second time, when the newly
*      decodable source symbol is actually decoded and ready,
*      i.e., when all the computations (like XOR and GF(2**8)
*      operations) have been performed. In any case, it is the
*      responsibility of the application to free this buffer when
*      needed, once decoding is over (but not before since the
*      codec does not keep any internal copy). This function does
*      not return anything.
* @param context_4_callback
*      (IN) Pointer to the application-specific context that will be
*      passed to the callback function (if any). This context is not
*      interpreted by this function.
* @return
*/
swif_status_t swif_decoder_set_callback_functions (
    swif_decoder_t* dec,
    void (*source_symbol_removed_from_linear_system_callback) (
        void* context,
        esi_t old_symbol_esi),
    void* (*decodable_source_symbol_callback) (
        void *context,
        esi_t esi),
    void (*decoded_source_symbol_callback) (
        void *context,
        void *new_symbol_buf,
        esi_t esi),
    void* context_4_callback);

/**

```

```

* This function sets one or more FEC codec specific parameters,
*     using a type/length/value approach for maximum flexibility.
*
* @param dec      context (i.e., pointer to decoder structure).
* @param type     (IN) Type of parameter.
* @param length   (IN) length of the pointed value.
* @param value    (IN) Pointer to the value. The exact type of
*                 the object pointed is FEC codec specific.
* @return
*/
swif_status_t    swif_decoder_set_parameters (
                                swif_decoder_t* dec,
                                uint32_t      type,
                                uint32_t      length,
                                void*         value);

/**
* This function gets one or more FEC codec specific parameters,
* using a type/length/value approach for maximum flexibility.
*
* @param dec      context (i.e., pointer to decoder structure).
* @param type     (IN) Type of parameter.
* @param length   (IN) length of the pointed value.
* @param value    (IN/OUT) Pointer to the value. The exact type of
*                 the object pointed is FEC codec specific.
*                 This function updates the value object
*                 accordingly. The caller, who knows the FEC codec,
*                 is responsible to allocate the appropriate
*                 object buffer.
* @return
*/
swif_status_t    swif_decoder_get_parameters (
                                swif_decoder_t* dec,
                                uint32_t      type,
                                uint32_t      length,
                                void*         value);

/**
* List here the FEC codec specific control parameters.
*/
enum {
    swif_DECODER_GET_PARAM_DECODER_STATISTICS = 1,
    swif_DECODER_SET_PARAM_RLC_DENSITY_THRESHOLD
};

/**
* 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 functions.
*
* This function usually returns SWIF_STATUS_OK, regardless of whether
* this new symbol enabled the decoding of one or several source symbols,
* or SWIF_STATUS_ERROR. It cannot return SWIF_STATUS_FAILURE.
*
* @param dec    context (i.e., pointer to decoder structure).
* @param new_symbol_buf
*             (IN) Pointer to the new source symbol now available (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
*             (IN) encoding symbol ID of the new source symbol.
* @return
*/
swif_status_t    swif_decoder_decode_with_new_source_symbol (
                                swif_decoder_t* dec,
                                void* const    new_symbol_buf,
                                esi_t          new_symbol_esi);

/**
* 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 functions.
*
* This function requires that the application has previously initialized
* the coding window and coding coefficients appropriately. The application
* keeps a full control of the repair symbol buffer, i.e., the application
* is in charge of freeing this buffer as soon as it believes appropriate
* (a copy is kept by the codec). This is motivated by the fact that a
* repair symbol may be part of a larger buffer (e.g., if there are
* several repair symbols per packet, or because of a packet header): only
* the application knows when the buffer can be safely freed.
*
* This function usually returns SWIF_STATUS_OK, regardless of whether
* this new symbol enabled the decoding of one or several source symbols,
* or SWIF_STATUS_ERROR. It cannot return SWIF_STATUS_FAILURE.
*
* @param dec    context (i.e., pointer to decoder structure).
* @param new_symbol_buf
*             (IN) Pointer to the new repair symbol now available (i.e.
*             a new symbol received by the application or a decoded
*             symbol in case of a recursive call if it makes sense).
* @return
*/
swif_status_t    swif_decoder_decode_with_new_repair_symbol (

```

```

                                swif_decoder_t* dec,
                                void* const      new_symbol_buf);
<CODE ENDS>

```

#### Decoder API proposal

```

<CODE BEGINS>
/**
 * Decoder structure that contains whatever is needed for decoding.
 * The exact content of this structure is FEC code dependent, the
 * structure below being a non normative example.
 * However it MUST be aligned with swif_decoder_t (same first items) in
 * order to be able to cast a pointer to one of the two structures,
 * depending on the context.
 */
typedef struct swif_decoder_internal {
    /* generic part of any control block. MUST be first in structure */
    swif_decoder_t  gen;

    /* desired verbosity: 0 is the minimum verbosity, the maximum
     * level being implementation specific. */
    uint32_t        verbosity;

    /* maximum number of source symbols used for any repair symbol */
    uint32_t        max_coding_window_size;

    /* max. number of source symbols kepts in current linear system.
     * If the linear system grows above this limit, old source
     * symbols in excess are removed and the application callback
     * called. This value should be larger than the
     * max_coding_window_size. */
    uint32_t        max_linear_system_size;

    /* exact size (in bytes) of any source or repair symbol */
    uint32_t        symbol_size;

    /* add whatever may be needed hereafter... */
} swif_decoder_internal_t;

```

Non normative example (RLC) of internal structure used by a decoder.

#### 4.4. Coding Window Functions at an Encoder and Decoder

This section gathers functions used to manage the coding window, both at an encoder and at a decoder. At an encoder a sliding (of fixed or elastic size) encoding window is managed. Whenever a repair symbol needs to be created, a linear combination (that is code specific) of source symbols currently in the encoding window is performed. This

encoding window is managed with the functions below plus, potentially, internal mechanisms that are code specific.

At a decoder, before submitting a new repair symbol to the codec, the application must specify the associated encoding window used at the source. This is done by the reset/add a single or set of symbols/remove a symbol functions. Once this coding window is ready, as well as the coding coefficient list if applicable, the application calls the `decode_with_new_repair_symbol()` function. A coding window may be reused for several repair symbols as long as they are all built from the same set of source symbols. In that case resetting the coding window and setting it from scratch would be a waste of time. The coding window must be viewed as a temporary list used solely by the `decode_with_new_repair_symbol()` function and kept independent from the linear system managed by the codec.

<CODE BEGINS>

```
/**
 * This function resets the current coding window. We assume here that
 * this window is maintained by the FEC codec instance.
 * Encoder:      reset the encoding window for the encoding of future
 *                repair symbols.
 * Decoder:      reset the coding window under preparation associated to
 *                a repair symbol just received.
 *
 * @return
 */
swif_status_t    swif_encoder_reset_coding_window (swif_encoder_t*  enc);

swif_status_t    swif_decoder_reset_coding_window (swif_decoder_t*  dec);

/**
 * Add this source symbol to the coding window.
 * Encoder:      add a source symbol to the coding window.
 * Decoder:      add a source symbol to the coding window under preparation.
 *
 * @param new_src_symbol_buf    (encoder only) pointer to a buffer
 *                               containing the source symbol. The application MUST NOT
 *                               free nor modify this buffer as long as the source symbol
 *                               is in the coding window.
 * @param new_src_symbol_esl    ESI of the source symbol to add.
 * @return
 */
swif_status_t    swif_encoder_add_source_symbol_to_coding_window (
                                swif_encoder_t*  enc,
                                void*            new_src_symbol_buf,
                                esi_t            new_src_symbol_esl);
```

```

swif_status_t    swif_decoder_add_source_symbol_to_coding_window (
                                swif_decoder_t* dec,
                                esi_t            new_src_symbol_esi);

/**
 * Remove this source symbol from the coding window.
 *
 * Encoder:    remove a source symbol from the encoding window, e.g.
 *              because the application knows that a source symbol has
 *              been acknowledged by the peer (if applicable). Note that
 *              the left side of the sliding window is automatically
 *              managed by the codec and no action is needed from the
 *              application. If needed a callback is available to inform
 *              the application that a source symbol has been removed).
 * Decoder:    remove a source symbol from the coding window under
 *              preparation.
 *
 * @param old_src_symbol_esi    ESI of the source symbol to remove from
 *                               the coding window.
 * @return
 */
swif_status_t    swif_encoder_remove_source_symbol_from_coding_window (
                                swif_encoder_t* enc,
                                esi_t            old_src_symbol_esi);

swif_status_t    swif_decoder_remove_source_symbol_from_coding_window (
                                swif_decoder_t* dec,
                                esi_t            old_src_symbol_esi);

<CODE ENDS>

```

Coding Window Functions at an Encoder and Decoder.

#### 4.5. Coding Coefficients Functions at an Encoder and Decoder

This section gathers functions used to manage the coding coefficients, both at an encoder and at a decoder. Since different FEC codecs will have different requirements, it is important to keep these functions separate from the `build_repair_symbol()` and `decode_with_new_repair_symbol()` functions. Several situations exist:

- o the application provides the list of coding coefficients to use for the next `build_repair_symbol()`;
- o the application provides a key (typically a PRNG seed) that the codec uses to produce the coding coefficients to use for the next `build_repair_symbol()`;
- o the choice of the coding coefficients is totally performed by the codec, in an autonomous manner (e.g., the codec includes an

algorithm that produces an appropriate seed based on various criteria, or the codec selects a set of coding coefficients based on various criteria). In that case the application needs to retrieve the list of coding coefficients or the key selected by the codec;

<CODE BEGINS>

```
/**
 * The following functions enable an encoder (resp. decoder) to
 * initialize the set of coefficients to be used for encoding
 * or associated to a received repair symbol.
 *
 * Encoder: calling one of them MUST be done before calling
 *          build_repair_symbol().
 * Decoder: calling one of them MUST be done before calling
 *          decode_with_new_repair_symbol().
 */

/**
 * Encoder: this function specifies the coding coefficients chosen by
 * the application if this is the way the codec works.
 * Decoder: communicate with this function the coding coefficients
 * associated to a repair symbol and carried in the packet
 * header.
 *
 * @param coding_coefs_tab
 * (IN) table of coding coefficients associated to each of
 * the source symbols currently in the encoding window.
 * The size (number of bits) of each coefficient depends on
 * the FEC Scheme. The allocation and release of this table
 * is under the responsibility of the application.
 * @param nb_coefs_in_tab
 * (IN) number of entries (i.e., coefficients) in the table.
 * @return
 */
swif_status_t swif_encoder_set_coding_coefs_tab (
                                swif_encoder_t* enc,
                                void*          coding_coefs_tab,
                                uint32_t        nb_coefs_in_tab);

swif_status_t swif_decoder_set_coding_coefs_tab (
                                swif_decoder_t* dec,
                                void*          coding_coefs_tab,
                                uint32_t        nb_coefs_in_tab);

/**
 * The coding coefficients may be generated in a deterministic manner,
```

```

* for instance by a PRNG known by the codec and a seed (perhaps with
* other parameters) provided by the application.
* The codec may also choose in an autonomous manner these coefficients.
* This function is used to trigger this process.
* When the choice is made in an autonomous manner, the actual coding
* coefficient or key used by the codec can be retrieved with
* swif_encoder_get_coding_coefs_tab().
*
* @param key      (IN) Value that can be used as a seed in case of a PRNG
*                  for instance, or by a specific coding coefficients
*                  function. Set to 0 if not required by a codec.
* @param add_param
*                  (IN) an opaque 32-bit integer that contains a codec
*                  specific parameter if needed. Set to 0 if not used.
* @return
*/
swif_status_t    swif_encoder_generate_coding_coefs (
                                swif_encoder_t* enc,
                                uint32_t          key,
                                uint32_t          add_param);

swif_status_t    swif_decoder_generate_coding_coefs (
                                swif_decoder_t* dec,
                                uint32_t          key,
                                uint32_t          add_param);

/**
* This function enables the application to retrieve the set of coding
* coefficients generated and used by build_repair_symbol(). This is
* useful when the choice of coefficients is performed by the codec in
* an autonomous manner but needs to be sent in the repair packet header.
* This function is only used by an encoder.
*
* @param coding_coefs_tab
*        (OUT) pointer to a table of coding coefficients.
*        The size (number of bits) of each coefficient depends on
*        the FEC scheme. Upon return of this function, this table
*        is allocated and filled with coefficient values. The
*        release of this table is under the responsibility of the
*        application.
* @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 zero.
*        Upon return of this function this variable is initialized
*        with the actual number of entries in the coeffs_tab[].
* @return

```



```

*/
swif_status_t    swif_encoder_get_coding_coefs_tab (
                                swif_encoder_t* enc,
                                void**          coding_coefs_tab,
                                uint32_t*        nb_coefs_in_tab);

/**
 * Get information on the current coding window at the encoder.
 * This function stores the ESI of the first source symbol and
 * last source symbol in the coding window, as well as the number
 * of symbols. In theory the application should be able to recover
 * the information (it knows when new symbols are added and old
 * symbols removed), but it's easier to let the SWiF Codec care
 * about it. The number of source symbols is also returned.
 * In situations where there's no gap (i.e., when
 * swif_encoder_remove_source_symbol_from_coding_window() has not
 * been used), nss can also be calculated with first/last. However
 * it is more convenient to use nss directly (in particular in case
 * of wrapping to zero of either first or last).
 *
 * @param enc
 * @param first      (in/out) pointer to ESI of the first source
 *                   symbol in the coding window (inclusive)
 * @param last       (in/out) pointer to ESI of the last source
 *                   symbol in the coding window (inclusive)
 * @param nss        (in/out) pointer to number of source symbols
 *                   in the coding window
 * @return
 */
swif_status_t    swif_encoder_get_coding_window_information (
                                swif_encoder_t* enc,
                                esi_t*         first,
                                esi_t*         last,
                                uint32_t*      nss);
<CODE ENDS>

```

Coding Coefficients Functions at an Encoder and Decoder.

## 5. Security Considerations

TBD

## 6. IANA Considerations

This document has no IANA requirement.

## 7. Acknowledgments

The authors would like to thank Marie-Jose Montpetit, Francois Michel, and Oumaima Attia for their valuable contributions to the IETF Hackathon SWiF-Codec project and their inputs to this document.

## 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>>.

### 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), June 2018, <<https://tools.ietf.org/html/draft-ietf-tsvwg-fecframe-ext>>.
- [RLC] Roca, V. and B. Teibi, "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), June 2018, <<https://tools.ietf.org/html/draft-ietf-tsvwg-rlc-fec-scheme>>.

## Authors' Addresses

Vincent Roca  
INRIA  
Univ. Grenoble Alpes  
France

EMail: [vincent.roca@inria.fr](mailto:vincent.roca@inria.fr)

Jonathan Detchart  
ISAE - Supaero  
France

EMail: [jonathan.detchart@isae-supaero.fr](mailto:jonathan.detchart@isae-supaero.fr)

Cedric Adjih  
INRIA  
France

EMail: [cedric.adjih@inria.fr](mailto:cedric.adjih@inria.fr)

Morten V. Pedersen  
Steinwurf ApS  
Denmark

EMail: [morten@steinwurf.com](mailto:morten@steinwurf.com)

nwcrg  
Internet-Draft  
Intended status: Informational  
Expires: September 10, 2020

V. Roca, Ed.  
INRIA  
F. Michel  
UCLouvain  
I. Swett  
Google  
M-J. Montpetit  
Triangle Video  
March 9, 2020

Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC)  
Schemes for QUIC  
draft-roca-nwcrg-rlc-fec-scheme-for-quic-03

Abstract

This document specifies Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for the QUIC transport protocol, in order to recover from packet losses.

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 10, 2020.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
2. Definitions and Abbreviations . . . . .	3
3. Procedures . . . . .	3
3.1. Source Symbols Mapping . . . . .	3
3.2. Source Symbols identification . . . . .	4
3.3. Pseudo-Random Number Generator (PRNG) . . . . .	4
3.4. Coding Coefficients Generation Function . . . . .	4
4. Sliding Window RLC FEC Scheme over $GF(2^{8})$ . . . . .	5
4.1. Formats and Codes . . . . .	5
4.1.1. Configuration Information . . . . .	5
4.1.2. SRC FPI Frame Format . . . . .	5
4.1.3. REPAIR Frame Format . . . . .	6
4.1.4. Additional Procedures . . . . .	7
4.2. FEC Code Specification . . . . .	7
4.2.1. Encoding Side . . . . .	7
4.2.2. Decoding Side . . . . .	7
5. Security Considerations . . . . .	7
6. IANA Considerations . . . . .	8
7. Acknowledgments . . . . .	8
8. References . . . . .	8
8.1. Normative References . . . . .	8
8.2. Informative References . . . . .	9
Authors' Addresses . . . . .	9

## 1. Introduction

QUIC [QUIC-transport] is a transport protocol that aims at improving network performance by enabling stream multiplexing, partial reliability, and methods of recovery besides retransmission, while also improving security. This document specifies FEC schemes for Sliding Window Random Linear Code (RLC) [RFC8681] to recover from lost packets within a single QUIC stream or across several QUIC streams, compliant with the FEC coding framework for QUIC [Coding4QUIC].

The ability to add FEC coding in QUIC may be beneficial in several situations:

- o for a robust transmission of latency-sensitive traffic, for instance real-time flows, since it enables to recover packet losses independently of the round trip time;

- o for the transmission of contents to a large set of QUIC reception endpoints, since the same repair frame may help recovering several different packet losses at different receivers;
- o for multipath communications, since repair traffic adds diversity.

## 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].

Terms and definitions that apply to coding are available in [nc-taxonomy]. More specifically, this document uses the following definitions:

**Packet versus Symbol:** a Packet is the unit of data that is exchanged over the network while a Symbol is the unit of data that is manipulated during the encoding and decoding operations

**Source Symbol:** a unit of data originating from the source that is used as input to encoding operations

**Repair Symbol:** a unit of data that is the result of a coding operation

This document uses the following abbreviations:

**E:** size of an encoding symbol (i.e., source or repair symbol), assumed fixed (in bytes)

## 3. Procedures

This section introduces the procedures that are used by these FEC Schemes.

### 3.1. Source Symbols Mapping

The present FEC Scheme follows the source symbols mapping specified in [Coding4QUIC]. Figure 1 illustrates this mapping.

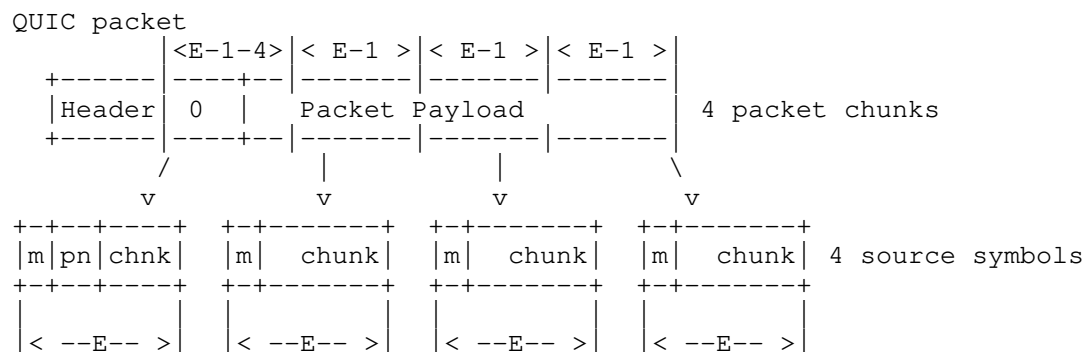


Figure 1: Example of source symbol mapping, when the E value is relatively small.

### 3.2. Source Symbols identification

Similarly to [RFC8681], the present FEC Scheme assigns a unique identifier (ID) to each produced source symbol. The IDs are assigned to the produced source symbols in the ascending order. The IDs start at MUST start 0 and MUST be contiguous. For any symbol with ID  $x$ , the source symbol with ID  $x+1$  is :

- o The source symbol containing the next packet chunk in the same QUIC packet as the source symbol  $X$ , if it exists.
- o The source symbol containing the first packet chunk of the next generated FEC-protected QUIC packet

Do we want to authorize a wrapping of the source symbol ID ? It would be a lot easier if wrapping is not permitted.

### 3.3. Pseudo-Random Number Generator (PRNG)

The RLC FEC Schemes defined in this document rely on the TinyMT32 PRNG defined in [RFC8682] along with the two mapping functions to 4-bit and 8-bit unsigned integers defined in [RFC8681].

### 3.4. Coding Coefficients Generation Function

The coding coefficients, used during the encoding process, are generated at the RLC encoder by the `generate_coding_coefficients()` function each time a new repair symbol needs to be produced. This specification uses the `generate_coding_coefficients()` defined in [RFC8681].

#### 4. Sliding Window RLC FEC Scheme over $GF(2^{8})$

This fully-specified FEC Scheme defines the Sliding Window Random Linear Codes (RLC) over  $GF(2^{8})$ .

##### 4.1. Formats and Codes

###### 4.1.1. Configuration Information

This section provides the RLC configuration information that needs to be shared during QUIC negotiation between the QUIC sender and receiver endpoints in order to synchronize them.

- o FEC Encoding ID (8 bits): the value assigned to this fully specified FEC Scheme MUST be XXXX, as assigned by IANA (Section 6). This FEC Encoding ID is used during the QUIC negotiation to uniquely identify the RLC FEC Scheme for QUIC;
- o Encoding symbol size, E (in bytes) (16 bits): a non-negative integer that indicates the size of each source and repair symbol, in bytes. This element is required both by the QUIC sender endpoint (RLC encoder) and the QUIC receiver endpoint(s) (RLC decoder).

TODO: specify exact format, with binary encoding, to be carried within the opaque 32-bit field during negotiation.

###### 4.1.2. SRC FPI Frame Format

The RLC FEC Scheme requires explicit signaling of the Source Symbols it transmits. The QUIC packets whose payload is protected by FEC MUST contain an SRC FPI frame with the following format.

```

      0               1               2               3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               ID of First Source Symbol in Packet (i)                               ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Figure 2: SRC FPI frame format.

The SRC FPI frame contains the ID of the first source symbol contained in this packet. Each source symbol contains a packet chunk of E-1 bytes long. If the payload to protect is longer than E-1 bytes, this means that the packet contains several packet chunks. In this case the source symbol ID will increase by exactly one for each additional packet chunk contained in the payload to protect.



Note: This frame is not idempotent. In the current version of QUIC, all the frames are idempotent (but this is not especially required). It would be great to preserve this property (a quick fix would be to add the packet number in the frame, but it takes a lot of space and I don't think it is very useful). Another bad property is that the frames are not independant anymore (several SRC FPI frames contained in the same packet have a confusing meaning). I really think that the correct solution would be a (encrypted) header field but I guess it is more complicated to propose (maybe in v2 we'll have a mechanism to define dynamic encrypted header fields during negotiation).

#### 4.1.3. REPAIR Frame Format

The RLC FEC Scheme requires QUIC REPAIR frames to convey enough information. This section specifies the REPAIR frame format specific to the RLC FEC Scheme. Note that the notion of REPAIR frame format is equivalent to the notion of Repair FEC Payload ID in [RFC8681].

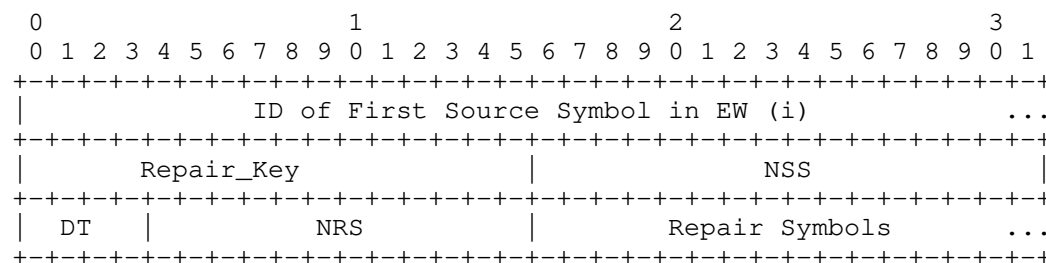


Figure 3: REPAIR frame format when protecting a single QUIC stream.

More precisely, the REPAIR frame format is composed of the following fields (Figure 3):

ID of the first Source Symbol in the Encoding Window (variable-size field):

a variable-length integer specifying the ID of the first source symbol in the encoding window. When a FEC REPAIR frame contains several repair symbols, this value applies to all of them;

Repair\_Key (16-bit field): this unsigned integer is used as a seed by the coefficient generation function (Section 3.4) in order to generate the desired number of coding coefficients. When a FEC Repair Packet contains several repair symbols, this repair key value is that of the first repair symbol. The remaining repair keys can be deduced by incrementing by 1 this value, up to a maximum value of 65535 after which it loops back to 0.

Number of Source Symbols in the encoding window, NSS (16-bit field):

this unsigned integer indicates the number of source symbols in the encoding window when this repair symbol was generated. When a REPAIR frame contains several repair symbols, the NSS value applies to all of them;

Density Threshold for the coding coefficients, DT (4-bit field): this unsigned integer carries the Density Threshold (DT) used by the coding coefficient generation function Section 3.4. More precisely, it controls the probability of having a non zero coding coefficient, which equals  $(DT+1) / 16$ . When a REPAIR frame contains several repair symbols, the DT value applies to all of them;

Number of Repair Symbols contained in the frame, NRS (12-bit field):

this unsigned integer specifies the number of Repair Symbols contained in this REPAIR frame;

Repair Symbols: data for this repair symbol(s). This field is  $NRS \times E$  bytes long.

#### 4.1.4. Additional Procedures

#### 4.2. FEC Code Specification

This RLC FEC Scheme relies on the FEC code specification defined in [RFC8681].

##### 4.2.1. Encoding Side

[RFC8681] high level description of a Sliding Window RLC encoder also applies here to this FEC Scheme.

##### 4.2.2. Decoding Side

[RFC8681] high level description of a Sliding Window RLC decoder also applies here to this FEC Scheme.

#### 5. Security Considerations

TBD

## 6. IANA Considerations

This document registers two values in the "QUIC FEC Encoding IDs" registry as follows:

- o XXXX refers to the Sliding Window Random Linear Codes (RLC) over  $GF(2^{8})$  FEC Scheme for a Single QUIC Stream, as defined in Section 4 of this document.

## 7. Acknowledgments

TBD

## 8. References

### 8.1. Normative References

[Coding4QUIC]

Swett, I., Montpetit, M-J., Roca, V., and F. Michel, "Coding for QUIC", Work in Progress, NWCRG draft-swett-nwcr-g-coding-for-quic (Work in Progress), March 2020, <<https://tools.ietf.org/html/draft-swett-nwcr-g-coding-for-quic>>.

[QUIC-transport]

Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport (Work in Progress) (work in progress), November 2019, <<https://datatracker.ietf.org/doc/draft-ietf-quic-transport/>>.

[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>>.

[RFC8681] Roca, V. and B. Teibi, "Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME", RFC 8681, DOI 10.17487/RFC8681, January 2020, <<https://www.rfc-editor.org/info/rfc8681>>.

[RFC8682] Saito, M., Matsumoto, M., Roca, V., Ed., and E. Baccelli, "TinyMT32 Pseudorandom Number Generator (PRNG)", RFC 8682, DOI 10.17487/RFC8682, January 2020, <<https://www.rfc-editor.org/info/rfc8682>>.

## 8.2. Informative References

[nc-taxonomy]

Roca (Ed.) et al., V., "Taxonomy of Coding Techniques for Efficient Network Communications", Request For Comments RFC 8406, June 2018, <<https://datatracker.ietf.org/doc/draft-irtf-nwcrg-network-coding-taxonomy/>>.

## Authors' Addresses

Vincent Roca (editor)  
INRIA  
Univ. Grenoble Alpes  
France

Email: [vincent.roca@inria.fr](mailto:vincent.roca@inria.fr)

Francois Michel  
UCLouvain  
Louvain-la-Neuve  
Belgium

Email: [francois.michel@uclouvain.be](mailto:francois.michel@uclouvain.be)

Ian Swett  
Google  
Cambridge, MA  
US

Email: [ianswett@google.com](mailto:ianswett@google.com)

Marie-Jose Montpetit  
Triangle Video  
Boston, MA  
US

Email: [marie@mjmontpetit.com](mailto:marie@mjmontpetit.com)

nwcrg  
Internet-Draft  
Intended status: Informational  
Expires: December 23, 2018

I. Swett  
Google  
M-J. Montpetit  
Triangle Video  
V. Roca  
INRIA  
June 21, 2018

Coding for QUIC  
draft-swett-nwcrg-coding-for-quic-01

Abstract

This document focusses on the integration of FEC coding in the QUIC transport protocol, in order to recover from packet losses. This document does not specify any FEC code but defines mechanisms to negotiate and integrate FEC Schemes in QUIC. By using proactive loss recovery, it is expected to improve QUIC performance in sessions impacted by packet losses. More precisely it is expected to improve QUIC performance with real-time sessions (since FEC coding makes packet loss recovery insensitive to the round trip time), with multicast sessions (since the same repair packet can recover several different losses at several receivers), and with multipath sessions (since repair packets add diversity).

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 December 23, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	2
2. Definitions and Abbreviations . . . . .	3
3. General Design Considerations . . . . .	4
3.1. FEC Code versus FEC Scheme, Block Codes versus Sliding Window Codes . . . . .	4
3.2. FEC Scheme Negotiation . . . . .	4
3.3. FEC Protection Within an Encrypted Channel . . . . .	5
3.4. About Middleboxes . . . . .	5
3.5. FEC Protection at the Stream Level . . . . .	5
3.6. About Gaps in the Set of Source Symbols Considered During Encoding . . . . .	5
4. FEC Scheme Negotiation in QUIC . . . . .	6
4.1. FEC Scheme Selection Process . . . . .	7
4.2. FEC Scheme Configuration Information . . . . .	7
5. Procedures when Protecting a Single QUIC Stream . . . . .	8
5.1. Application data, STREAM Frame data and Source Symbols . . . . .	8
5.2. Signaling Considerations within STREAM and REPAIR Frames . . . . .	9
5.3. Management of Silent Periods and End of Stream . . . . .	10
6. Procedures when Protecting Several QUIC Streams . . . . .	11
6.1. Application data, STREAM Frame data and Source Symbols . . . . .	11
6.2. Block or Encoding Window Management . . . . .	11
6.3. Signaling Considerations within STREAM and REPAIR Frames . . . . .	12
7. Security Considerations . . . . .	13
8. IANA Considerations . . . . .	13
9. Acknowledgments . . . . .	13
10. References . . . . .	13
10.1. Normative References . . . . .	13
10.2. Informative References . . . . .	14
Authors' Addresses . . . . .	14

## 1. Introduction

QUIC is a new transport that aims at improving network performance by enabling out of order delivery, partial reliability, and methods of recovery besides retransmission, while also improving security. This document specifies a framework to enable FEC codes to be used to

recover from lost packets within a single QUIC stream or across several QUIC streams.

The ability to add FEC coding in QUIC may be beneficial in several situations:

- o for a robust transmission of latency sensitive traffic, for instance real-time flows, since it enables to recover packet losses independently of the round trip time;
- o for the transmission of contents to a large set of QUIC reception endpoints, since the same repair frame may help recovering several different packet losses at different receivers;
- o for multipath communications, since repair traffic adds diversity.

This framework does not mandate the use of any specific FEC code (i.e., how to encode and decode) nor FEC Scheme (i.e., that specifies both a FEC code and how to use it, in particular in terms of signaling). Instead it allows to negotiate the FEC Scheme to use at session startup, assuming that more than one solution could potentially be offered concurrently. Without loss of generality, we assume that the encoding operations compute a linear combination of QUIC packets, regardless of whether these codes are of block type (as with Reed-Solomon codes [RFC5510]) or sliding window type (as with RLC codes [RLC]).

## 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].

Terms and definitions that apply to coding are available in [nc-taxonomy]. More specifically, this document uses the following definitions:

Packet versus Symbol: a Packet is the unit of data that is exchanged over the network while a Symbol is the unit of data that is manipulated during the encoding and decoding operations

Source Symbol: a unit of data originating from the source that is used as input to encoding operations

Repair Symbol: a unit of data that is the result of a coding operation

This document uses the following abbreviations:

E: size of an encoding symbol (i.e., source or repair symbol), assumed fixed (in bytes)

### 3. General Design Considerations

This section lists a few general considerations that govern the framework for FEC coding support in QUIC.

#### 3.1. FEC Code versus FEC Scheme, Block Codes versus Sliding Window Codes

A FEC code specifies the details of encoding and decoding operations. In addition to that, a FEC Scheme defines the additional protocol aspects required to use a particular FEC code [nc-taxonomy]. In particular the FEC Scheme defines signaling (e.g., information contained in Source and Repair Packet header or trailers) needed to synchronize encoders and decoders.

Block coding (e.g., Reed-Solomon [RFC5510]) and sliding window coding (e.g., RLC [RLC]) are two broad classes of FEC codes [nc-taxonomy]. In the first case, the input flow must be first segmented into a sequence of blocks, FEC encoding and decoding being performed independently on a per-block basis. In the second case rely, a sliding encoding window continuously slides over the input flow. It is envisioned that the two classes of codes could be used to bring FEC protection to QUIC, usually with an advantage for sliding window codes when it comes to low latency communications.

#### 3.2. FEC Scheme Negotiation

There are multiple FEC Scheme candidates. Therefore a negotiation step is needed 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-transport], during the QUIC handshake.

Editor's notes:

- \* It is likely that FEC Scheme negotiation requires the use of a new dedicated Extension Frame Type. To Be Clarified and text updated.
- \* It is not clear whether negotiation is meant to select a **\*\*single\*\*** FEC Scheme or **\*\*multiple\*\*** FEC Schemes. In the second case (multiple FEC) it is required to have a complementary mechanism to indicate which FEC Scheme is used in a given REPAIR frame (which could be done through as many



REPAIR frame type values as potential FEC Scheme negotiated). Is it what we want to achieve? Not sure.

- \* It is not clear whether negotiation is carried out at QUIC level (and therefore for multiple streams) or at a stream level (and therefore multiple streams may use multiple FEC Schemes). The terminology used above should be updated to reflect the choice.

### 3.3. FEC Protection Within an Encrypted Channel

FEC encoding is applied before any QUIC encryption and authentication processing. Source symbols, that constitute the data units used by the FEC codec, contain cleartext application data.

### 3.4. About Middleboxes

The coding approach described in this document does not allow on path elements (middleboxes) to take part in FEC protection. The traffic being encrypted end-to-end, the middleboxes are not in position to perform FEC decoding, nor to add any redundant traffic.

### 3.5. FEC Protection at the Stream Level

Streams in QUIC provide a lightweight, ordered byte-stream abstraction. FEC encoding is applied at the stream level, within a single stream or across two or more streams of the same QUIC session. This is motivated by the fact that FEC protection is not necessarily beneficial to all data streams, but only to a subset of them. For instance FEC protection can be highly beneficial to live video streams to which the proactive erasure correction feature of FEC, independent of the RTT, should be highly beneficial. On the opposite, FEC protection is probably less attractive for latency insensitive bulk unicast flows.

In order to facilitate experiments, and in order to enable backward compatibility, the STREAM frames that carry application data are kept unmodified. On the opposite, frames that carry one or more repair symbols use a dedicated REPAIR frame type, chosen within the type range dedicated to "Extension Frames".

### 3.6. About Gaps in the Set of Source Symbols Considered During Encoding

A given FEC Scheme MAY support or not the presence of gaps in the set of source symbols that constitute a block (for Block codes) or an encoding window (for Sliding Window codes). A potential cause for non contiguous sets of source symbols is the acknowledgment of one of them. When this happens, the QUIC sending endpoint may want to

remove this source symbol from further FEC encodings. This is particularly true with Sliding Window codes because of their flexibility during FEC encoding (i.e., the encoding window can change between two consecutive FEC encodings).

Supporting gaps can be motivated by the desire to reduce encoding and decoding complexity since there are fewer variables. However this choice has major consequences in terms of signaling. Indeed each repair symbol transmitted MUST be accompanied with enough information for the QUIC decoding endpoint to unambiguously identify the exact composition of the block or encoding window. Without any gap, the identity of the first source symbol plus the number of symbols in the block or encoding window is sufficient. With gaps, a more complex encoding needs to be used, perhaps similar to the encoding used for selective acknowledgments.

Whether or not gaps are supported MUST be clarified in each FEC Scheme.

#### 4. FEC Scheme Negotiation in QUIC

FEC Scheme negotiation has two goals:

- o Selecting a FEC Scheme (or FEC Schemes) that can be used by the QUIC transmission and reception endpoints. This process requires an exchange between them;
- o Communicating a certain number of parameters, the "Configuration Information", that are not expected to change over the session lifetime. For instance, this is the case of the symbol size parameter, E (in bytes), that needs either to be agreed between the endpoints, or chosen by the sender and communicated to the receiver(s);

Editor's notes:

- \* It is likely that FEC Scheme negotiation requires the use of a new dedicated Extension Frame Type. The details remain TBD.
- \* The Negotiation Frame Type format remains TBD.
- \* How to communicate the parameters remains TBD.
- \* The present document only provides high level principles, the details are of course the responsibility of the FEC Scheme.

- \* In case negotiation is different when protecting a single versus several streams, this section may be moved to the respective sections.
- \* How does it work in case of a multicast session?
- \* Do we negotiate here a FEC Scheme on a per-Stream basis (or group of Streams to be protected jointly)? Or do we negotiate a FEC Scheme on a QUIC session basis, therefore to be used for all the Streams that need FEC protection?

#### 4.1. FEC Scheme Selection Process

Let us consider the FEC Scheme selection process between the QUIC endpoints. Figure 1 illustrates the principle when a QUIC reception endpoint initiates the exchange.

```

QUIC sender                                QUIC receiver
< - - - - -
    supported_fec_scheme_32b{FS1_Encoding_ID | other}
    supported_fec_scheme_64b{FS1_Encoding_ID | other}

choose FEC Scheme "FS1"
- - - - -
    supported_fec_scheme_32b{FS1_Encoding_ID | other}

```

Figure 1: Example FEC Scheme selection process, during the initial negotiation.

The `supported_fec_scheme_16b` and `supported_fec_scheme_32b` are two new TransportParameterId to be added to the "Table 7: Initial QUIC Transport Parameters Entries" Section 13.1, of [QUIC-transport]. The `supported_fec_scheme_32b` contains a 32-bit data field to carry opaque 32-bit value, while the `supported_fec_scheme_64b` contains a 64-bit data field to carry opaque 64-bit value (see Section 4.2).

#### 4.2. FEC Scheme Configuration Information

Let us now focus on the communication of configuration information specific to the selected FEC Scheme. In Figure 1, the `supported_fec_scheme_32b{FS1_Encoding_ID}` contains a field meant to carry the FEC Encoding ID of the FEC Scheme selected plus additional configuration information if any. If a 32 bit opaque field is not sufficient, the `supported_fec_scheme_64b` can be used instead and proposes a 64 bit opaque field.

## 5. Procedures when Protecting a Single QUIC Stream

This section focusses on the simple case where FEC protection is applied to a single QUIC stream. We consider a unidirectional data flow between a QUIC sending endpoint and one (or more) QUIC reception endpoints.

### 5.1. Application data, STREAM Frame data and Source Symbols

Application data is kept in a transmission buffer at a QUIC sending endpoint, and sent within STREAM frames. Each STREAM frame that carries data contains an Offset field that indicates the offset within the stream of the first byte of the Stream Data field, as well as a Length field that indicates the number of bytes contained in the Stream Data field. Upon receiving a STREAM frame, using the Offset and Length fields, a QUIC reception endpoint can easily store data in its reception buffer. But since a QUIC Packet may be lost during transmission, the reception buffer may have gaps.

Figure 2 illustrates how source symbols are mapped to the QUIC transmission or reception buffers (same principle on either side). Since any source (and repair) symbol is of fixed size (E bytes) for a given stream, since QUIC guarantees that the first byte in the stream has an offset of 0, the position of each source symbol is known by both ends.

```

< -E- > < -E- > < -E- > < -E- >
+-----+-----+-----+-----+
|< -- Frame 1 -- >< ----- Frame | source symbols 0, 1, 2, 3
+-----+-----+-----+-----+
| 2 ----- >< --- Frame 3 -- >< -| source symbols 4, 5, 6, 7
+-----+-----+-----+-----+
| Frame 4 - >< -F5- >| source symbols 8, 9 and 10
+-----+-----+-----+ (incomplete)

```

Figure 2: Example of source symbol mapping, when the E value is relatively small.

Any value for E is possible, from a single byte to several hundreds or thousands of bytes. In general, the source symbols are not aligned with data chunks sent in the STREAM frames. A given STREAM frame may carry all the bytes of a given source symbol. But when a source symbol straddles two or more (e.g., if E is large compared to usual frame size) STREAM frames, a proper reception of these two (or more) STREAM frames is needed for a QUIC reception endpoint to consider that the source symbol is available for FEC decoding operations. The choice of an appropriate value for E may depend on the use case (in particular on the nature of application data). A

reasonably small value reduces the probability that a source symbol straddles two or more STREAM frames, a situation that is considered as potentially harmful (the unit of control, the source symbol, and unit of transmission, the frame, are not aligned). However an overly small value also increases processing complexity (FEC encoding and decoding are performed over a larger linear system) so there is an incentive to use a larger value. An appropriate balance should be found, and this choice is considered as out of scope for this document.

## 5.2. Signaling Considerations within STREAM and REPAIR Frames

Once the initial negotiation succeeded and an appropriate FEC Scheme has been chosen between the QUIC endpoints, data is exchanged as follows. Source data is transmitted within STREAM frames, as would happen without any FEC based loss recovery mechanism (in particular without considering source symbols boundaries). Repair data, computed during FEC encoding, on the opposite, is sent within a dedicated REPAIR frame type, chosen within the type range dedicated to "Extension Frames". In both cases, the same Stream ID is used since both flows relate to the same stream.

The REPAIR frame format is FEC Scheme dependent. The document specifying a FEC Scheme to be used with QUIC MUST define the REPAIR frame format, among other things. The REPAIR frame MUST carry enough information for a QUIC reception endpoint to understand exactly how this repair symbol(s) has(ve) been generated. It implies that each REPAIR symbol MUST communicate the block (with block codes) or encoding window (with Sliding Window codes) composition. This MAY be achieved by communicating in case there is no gap in the source symbol set (see XXX):

- o the offset of the first source symbol of the block or encoding window;
- o the number of source symbols in the block or encoding window, which can be either a number of symbols or a number of bytes since symbols are of fixed size, E.

Note that unlike FEC Schemes for FLUTE/ALC, NORM, and FECFRAME, here there is no notion of Encoding Symbol Id (ESI), an identifier managed in a sequential manner to identify source and repair symbols. The use of an offset within the stream, with the guaranty that no wrapping to zero can occur, provides an alternative mechanism to identify any source symbol.

As explained above, source data is transmitted without any modification, which provides backward compatibility. This is

advantage in situations where the same QUIC stream is delivered to several QUIC reception endpoints (multicast): it may be appropriate to select a given FEC Scheme even if it is known that a subset of the QUIC reception endpoints do not support it.

Editor's notes:

- \* This I-D proposes to define a single generic REPAIR frame type, but an alternative could be to have a one-to-one mapping between a REPAIR frame type and a specific FEC Scheme.
- \* The use of frame type within the Extension Frames range for REPAIR frames is meant to facilitate experimentations. If the use of coding in QUIC is recognized as having benefits, a dedicated (or more, see above) frame type could be selected later on.

### 5.3. Management of Silent Periods and End of Stream

If an application does not submit fresh data for some time, the last source symbol may not be totally filled. It follows that this last source symbol cannot be considered during FEC encoding and therefore the associated bytes of the application stream are not protected. A similar problem arrives when a stream is finished, the application having no more data to submit to QUIC. Here also, the bytes of the last incomplete source symbol are not protected by FEC encoding.

In order to solve this problem, it is RECOMMENDED that a QUIC sending endpoint:

- o Identifies when such a situation is likely to occur, for instance by waiting no more than a certain time during an application silent period;
- o Upon time-out, the application falls back to the alternative re-transmission based loss recovery mechanism for the bytes of the last incomplete source symbol;

Editor's notes: Clearly, the above mechanism requires more thoughts as well as experimental work. The "end of stream" situation may be addressed through zero padding perhaps easily. However the use of zero padding for transitory silent periods may add a lot of specification and implementation complexity...

## 6. Procedures when Protecting Several QUIC Streams

This section focusses on the general case where FEC protection is globally applied across two or more QUIC streams.

Editor's notes: It is not clear whether this use-case is needed. It adds specification and implementation complexity that need to be balanced with the expected benefits.

- \* Receiver: A first complexity comes from the requirement to identify to which stream a decoded source symbol belongs to. This is also one of the main difficulty for FECFRAME (both with block and sliding window codes) which required to distinguish an ADU (submitted by the application) from an ADUI (the same ADU plus an additional FlowID among other things). Do we want this level of complexity?
- \* Sender: Another complexity comes from the encoding window management at a sender. With multiple streams, shifting the encoding window to the right needs to be done based on timestamps associated to source symbols of the various streams: the oldest source symbol across all the streams will be removed.
- \* When two largely different streams are protected together (e.g., a high definition 4K video flow plus the associated relatively low-rate audio stream), is this extra complexity balanced by significant performance improvements compared to an independent protection on each stream (intuition is yes, the low bitrate flow is better protected iff the encoding window is large enough)? And when the various streams have a comparable bitrate? More work (incl. experimental work) is needed to answer this question.

### 6.1. Application data, STREAM Frame data and Source Symbols

Within each stream, the source symbols MUST be defined as in the simple case of a single stream. Figure 2 remains valid.

### 6.2. Block or Encoding Window Management

The details of how to create the block or encoding window are specific to the FEC Scheme. A possible approach is the following.

When creating the block (block FEC code) or encoding window (sliding window FEC code), the source symbols to consider of each stream are appended. All the relevant source symbols of the first stream are appended, followed by all the source symbols of the second stream,

etc. These sequences do not follow any timing consideration in order to simplify signaling.

Figure 3 illustrates, in case of a Sliding Window FEC Scheme, an encoding window with source symbols belonging to two streams, of Stream ID 120 and 51 respectively.

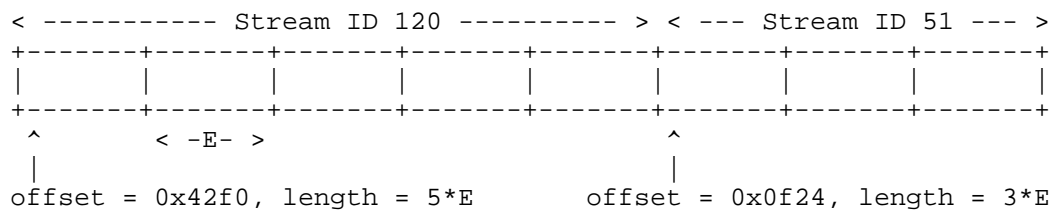


Figure 3: Example of encoding window of a Sliding Window FEC Scheme and FEC protection across two streams.

### 6.3. Signaling Considerations within STREAM and REPAIR Frames

Source data on each stream is transmitted within STREAM frames, as would happen without any FEC based loss recovery mechanism.

Repair symbols, generated during FEC encoding as a linear combination of source symbols that belong to one or more of the streams, are transmitted within REPAIR frames. Each REPAIR frame can be associated to any of the input streams it protects, and therefore associated to any of the associated Stream IDs.

Editor's notes: Check that indeed, with FEC protection across several streams, assigning a REPAIR frame to any of the streams it protects is meaningful. Should an approach for selecting one stream (and Stream ID) be preferred?

The REPAIR frame format is FEC Scheme dependent and MUST be defined by document specifying a FEC Scheme. One of the key information of this REPAIR frame is the composition of the block (with block codes) or encoding window (with sliding window codes) used to perform FEC encoding. Indeed, this is the only manner to convey this information since an application flow is not predictable (e.g., if an application flow is momentarily suspended, the composition of the block or encoding window will be affected). One possibility is to list, in each REPAIR frame header:

- o the actual number of streams considered (the maximum number is known after the negotiation step, but if one of the streams remains silent for some time, it may not contribute during



encoding and therefore be absent from the block or encoding window);

- o for each stream concerned, its Stream ID, the offset of the first source symbol considered as well as the length, i.e., the number of bytes considered.

This approach does not enable to keep track of the source symbol ordering across streams, but enables a non ambiguous description of the encoding window.

The FEC Scheme specification MUST also detail how to manage the block or encoding window. For instance, should the oldest source symbol of any stream be removed from the encoding window when this latter is shifted to the right? This would mean that a timestamp is attached to each source symbol in order to identify the oldest one across all streams.

## 7. Security Considerations

TBD

## 8. IANA Considerations

TBD

## 9. Acknowledgments

TBD

## 10. References

### 10.1. Normative References

[QUIC-transport]

Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-12 (work in progress), May 2018, <<https://datatracker.ietf.org/doc/draft-ietf-quic-transport/>>.

[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>>.

## 10.2. Informative References

## [nc-taxonomy]

Roca et al., V., "Taxonomy of Coding Techniques for Efficient Network Communications", draft-irtf-nwcr-network-coding-taxonomy (Work in Progress) (work in progress), March 2018, <<https://datatracker.ietf.org/doc/draft-irtf-nwcr-network-coding-taxonomy/>>.

## [RFC5510]

Lacan, J., Roca, V., Peltotalo, J., and S. Peltotalo, "Reed-Solomon Forward Error Correction (FEC) Schemes", RFC 5510, DOI 10.17487/RFC5510, April 2009, <<https://www.rfc-editor.org/info/rfc5510>>.

## [RLC]

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

## Authors' Addresses

Ian Swett  
Google  
Cambridge, MA  
US

Email: [ianswett@google.com](mailto:ianswett@google.com)

Marie-Jose Montpetit  
Triangle Video  
Boston, MA  
US

Email: [marie@mjmontpetit.com](mailto:marie@mjmontpetit.com)

Vincent Roca  
INRIA  
Univ. Grenoble Alpes  
France

Email: [vincent.roca@inria.fr](mailto:vincent.roca@inria.fr)

nwcrg  
Internet-Draft  
Intended status: Informational  
Expires: September 10, 2020

I. Swett  
Google  
M-J. Montpetit  
Triangle Video  
V. Roca  
INRIA  
F. Michel  
UCLouvain  
March 9, 2020

Coding for QUIC  
draft-swett-nwcrg-coding-for-quic-04

Abstract

This document focuses on the integration of FEC coding in the QUIC transport protocol, in order to recover from packet losses. This document does not specify any FEC code but defines mechanisms to negotiate and integrate FEC Schemes in QUIC. By using proactive loss recovery, it is expected to improve QUIC performance in sessions impacted by packet losses. More precisely it is expected to improve QUIC performance with real-time sessions (since FEC coding makes packet loss recovery insensitive to the round trip time), with short sessions (since FEC coding can help recovering from tail losses more rapidly than through retransmissions), with multicast sessions (since the same repair packet can recover several different losses at several receivers), and with multipath sessions (since repair packets add diversity and flexibility).

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 10, 2020.

## Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction . . . . .	3
2. Definitions and Abbreviations . . . . .	3
3. General Design Considerations . . . . .	4
3.1. FEC Code versus FEC Scheme, Block Codes versus Sliding Window Codes . . . . .	4
3.2. FEC Scheme Negotiation . . . . .	4
3.3. FEC Protection Within an Encrypted Channel . . . . .	5
3.4. About Middleboxes . . . . .	5
4. FEC Protection Principles . . . . .	5
4.1. Cross Packet Frames FEC Encoding . . . . .	5
4.2. Source Symbol Definition . . . . .	6
4.2.1. Packet Payload to Packet Chunk Mapping . . . . .	6
4.2.2. Packet Chunk to Source Symbol Mapping . . . . .	7
4.2.2.1. Open questions: Content of Source Symbols Metadata? Removing certain frames from FEC protection? . . . . .	9
4.2.3. Source Symbol Size (E) Considerations . . . . .	10
4.3. Source Symbol Signaling . . . . .	11
4.4. Repair Symbol Signaling . . . . .	11
4.5. Signaling a Symbol Recovery . . . . .	11
4.6. About Gaps in the Set of Source Symbols Considered During Encoding . . . . .	12
5. FEC Scheme Negotiation in QUIC . . . . .	12
5.1. FEC Scheme Negotiation . . . . .	13
6. Security Considerations . . . . .	15
7. IANA Considerations . . . . .	15
8. Acknowledgments . . . . .	15
9. References . . . . .	15
9.1. Normative References . . . . .	16
9.2. Informative References . . . . .	16
Authors' Addresses . . . . .	16

## 1. Introduction

QUIC is a new transport that aims at improving network performance by enabling out of order delivery, partial reliability, and methods of recovery besides retransmission, while also improving security. This document specifies a framework to enable FEC codes to be used to recover from lost packets within a single QUIC stream or across several QUIC streams.

The ability to add FEC coding in QUIC may be beneficial in several situations:

- o for a robust transmission of latency sensitive traffic, for instance real-time flows, since it enables to recover packet losses independently of the round trip time;
- o for short sessions, in order to protect the last few packets sent, since it enables to recover from tail losses more rapidly than through retransmissions;
- o for the transmission of contents to a large set of QUIC reception endpoints, since the same repair frame may help recovering several different packet losses at different receivers;
- o for multipath communications, since repair traffic adds diversity and flexibility.

This framework does not mandate the use of any specific FEC code (i.e., how to encode and decode) nor FEC Scheme (i.e., that specifies both a FEC code and how to use it, in particular in terms of signaling). Instead it allows to negotiate the FEC Scheme to use at session startup, assuming that more than one solution could potentially be offered concurrently. Without loss of generality, we assume that the encoding operations compute a linear combination of QUIC packets, regardless of whether these codes are of block type (as with Reed-Solomon codes [RFC5510]) or sliding window type (as with RLC codes [RFC8681]).

## 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].

Terms and definitions that apply to coding are available in [nc-taxonomy]. More specifically, this document uses the following definitions:

Packet versus Symbol: a Packet is the unit of data that is exchanged over the network while a Symbol is the unit of data that is manipulated during the encoding and decoding operations

Source Symbol: a unit of data originating from the source that is used as input to encoding operations

Repair Symbol: a unit of data that is the result of a coding operation

This document uses the following abbreviations:

E: size of an encoding symbol (i.e., source or repair symbol), assumed fixed (in bytes)

### 3. General Design Considerations

This section lists a few general considerations that govern the framework for FEC coding support in QUIC.

#### 3.1. FEC Code versus FEC Scheme, Block Codes versus Sliding Window Codes

A FEC code specifies the details of encoding and decoding operations. In addition to that, a FEC Scheme defines the additional protocol aspects required to use a particular FEC code [nc-taxonomy]. In particular the FEC Scheme defines signaling (e.g., information contained in Source and Repair Packet header or trailers) needed to synchronize encoders and decoders.

Block coding (e.g., Reed-Solomon [RFC5510]) and sliding window coding (e.g., RLC [RFC8681]) are two broad classes of FEC codes [nc-taxonomy]. In the first case, the input flow must be first segmented into a sequence of blocks, FEC encoding and decoding being performed independently on a per-block basis. In the second case rely, a sliding encoding window continuously slides over the input flow. It is envisioned that the two classes of codes could be used to bring FEC protection to QUIC, usually with an advantage for sliding window codes when it comes to low latency communications.

#### 3.2. FEC Scheme Negotiation

There are multiple FEC Scheme candidates. Therefore a negotiation step is needed 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-transport], during the QUIC handshake.

## Editor's notes:

- \* It is likely that FEC Scheme negotiation requires the use of a new dedicated Extension Frame Type. To Be Clarified and text updated.
- \* It is not clear whether negotiation is meant to select a **\*\*single\*\*** FEC Scheme or **\*\*multiple\*\*** FEC Schemes. In the second case (multiple FEC) it is required to have a complementary mechanism to indicate which FEC Scheme is used in a given REPAIR frame (which could be done through as many REPAIR frame type values as potential FEC Scheme negotiated). Is it what we want to achieve? Not sure.

### 3.3. FEC Protection Within an Encrypted Channel

FEC encoding is applied before any QUIC encryption and authentication processing. Source symbols, that constitute the data units used by the FEC codec, contain cleartext data (application and/or QUIC data).

### 3.4. About Middleboxes

The coding approach described in this document does not allow on path elements (middleboxes) to take part in FEC protection. The traffic being encrypted end-to-end, the middleboxes are not in position to perform FEC decoding, nor to add any redundant traffic.

## 4. FEC Protection Principles

The present section explains how FEC encoding can be applied to QUIC. It defines the general ideas for mapping QUIC packet frames to source symbols, as well as the associated signaling. This section does not define the FEC Scheme specific details that need to be specified in a companion document.

### 4.1. Cross Packet Frames FEC Encoding

A QUIC packet payload consists in a set of QUIC frames. These frames either carry application data (e.g., in a STREAM or DATAGRAM frame) or control information (e.g., a MAX\_DATA frame). Each packet is either entirely received or lost, and is uniquely identified by a monotonically increasing Packet Number.

Through the use of FEC encoding, application data can be protected proactively against packet losses, without requiring to go through packet retransmission. In addition to application data, QUIC transfers might benefit from protecting control frames having a potential impact on the transmission throughput, such as MAX\_DATA or

MAX\_STREAM\_DATA frames. Therefore this document introduces an FEC protection across all -- or a subset of -- the frames of a given QUIC packet. This design choice impacts the QUIC packet to source symbols mapping, as well as signaling aspects, both of them being discussed hereafter.

#### 4.2. Source Symbol Definition

The cross packet frames FEC encoding approach considers the sequence of frames (or a sub-sequence of them) transmitted within a given QUIC packet, seen as the QUIC packet payload. From this payload, it defines a mapping to source symbols (see Section 4.2.1 and Section 4.2.2). Source symbols are then used for encoding purposes, producing one or more repair symbols, the details of which depend on the FEC Scheme considered. However source symbols are never sent per se on the network. Instead the original QUIC packet, plus a dedicated signaling header, are sent and therefore implicitly carry those source symbols. The QUIC packets, containing one or more repair symbols, are sent on the network.

The only modification to the original QUIC packet is the addition of a dedicated FEC\_SRC\_FPI frame type, meant to carry source symbol signaling (known as Source FEC Payload Information, or FPI). On the opposite, frames that carry one or more repair symbols use a dedicated REPAIR frame type. In both cases, in order to facilitate experiments and enable backward compatibility, the FEC\_SRC\_FPI and REPAIR frame types are chosen within the type range dedicated to "Extension Frames". Thereby, a legacy receiver will automatically ignore these unknown frame types. As QUIC packets can be of different lengths, a special care must be taken to ensure having a fixed Source Symbol size to ease FEC Scheme implementations.

##### 4.2.1. Packet Payload to Packet Chunk Mapping

This section defines a mechanism to segment a QUIC packet payload, composed of several frames, into fixed-size payload chunks, of size E-1 bytes or E-1-4 bytes for the first chunk when the QUIC Packet Number needs to be added ((Section 4.2.2). Depending on the relative value of E-1 (or E-1-4) and the QUIC packet payload size, a packet can potentially contain more than one chunks. This is a first step into producing source symbols. Figure 1 illustrates this process.



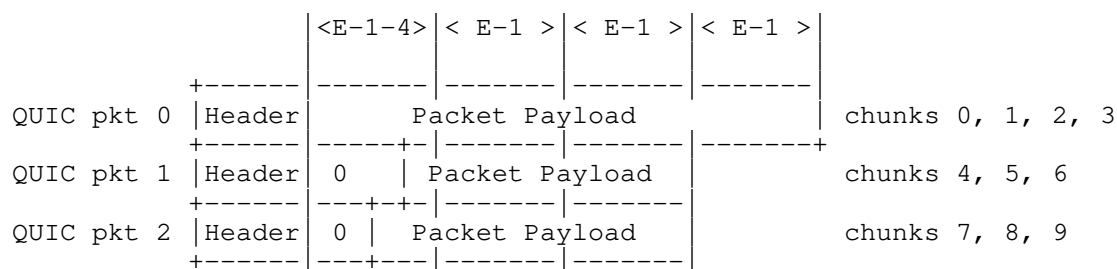


Figure 1: Example of QUIC packet to chunk mapping, when the E-1 value is relatively small, with prepended zero padding when needed (here packets 1 and 2), and assuming the first chunk contains the QUIC Packet Number in 4 bytes compressed version.

#### 4.2.2. Packet Chunk to Source Symbol Mapping

The second step consists in producing the source symbols. A source symbol is the concatenation of a single byte of metadata, potentially followed by the Packet Number of the associated source, plus a packet chunk. Figure 2 illustrates the situation where a compressed QUIC packet number is added (in general for the first chunk of a QUIC packet). Figure 3 illustrates the situation where there is no QUIC packet number (in general for the following chunk(s) of a QUIC packet). When the QUIC packet number is present, this identifier can be recovered by a receiver after successful FEC decoding. It means that a RECOVERED frame can be generated to the sender to indicate that this packet (identified by the QUIC packet number) has been recovered. Each source symbol is of fixed-size E bytes. These source symbols are only used during encoding and decoding and are not sent as-is on the network.

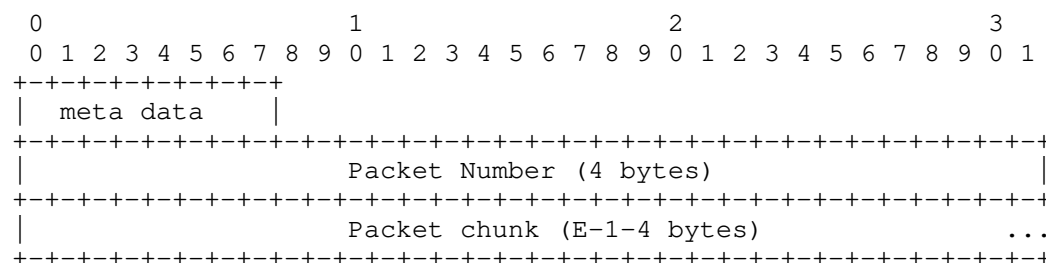


Figure 2: Source symbol format with Packet Number information (e.g., first packet chunk).

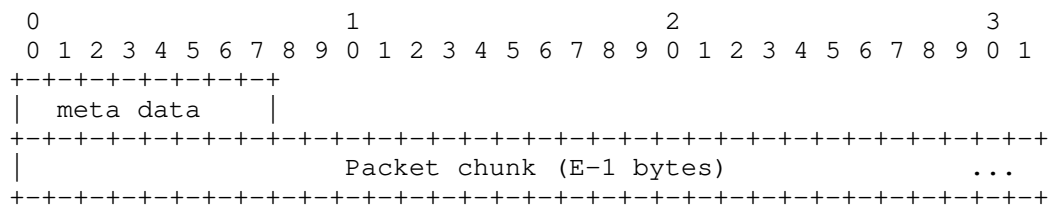


Figure 3: Source symbol format without Packet Number information (e.g., packet chunks except the first one).

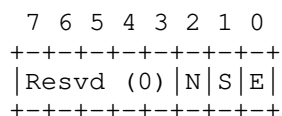


Figure 4: Source symbol metadata format.

Figure 4 shows the format of the 1 byte metadata. The fields are the following:

Reserved field (5 bits): for this specification, this field MUST be equal to zero.

Packet Number (N) field (1 bit): this field indicates that the following 4 bytes contain the Packet Number (short 32-bit representation) of the associated QUIC packet ([QUIC-transport] section 17.1., Packet Number Encoding and Decoding).

Start (S) bit (1 bit): this field, when set to 1, indicates that this source symbol contains the first chunk of the packet payload.

End bit (E) (1 bit): this field, when set to 1, indicates that this source symbol contains the last chunk of the packet payload.

Note that with a QUIC packet containing a single chunk, the associated metadata will contain S=E=1. On the opposite, a source symbol containing a intermediate chunk (i.e., neither the first nor the last chunk of the QUIC packet), the associated metadata will contain S=E=0.

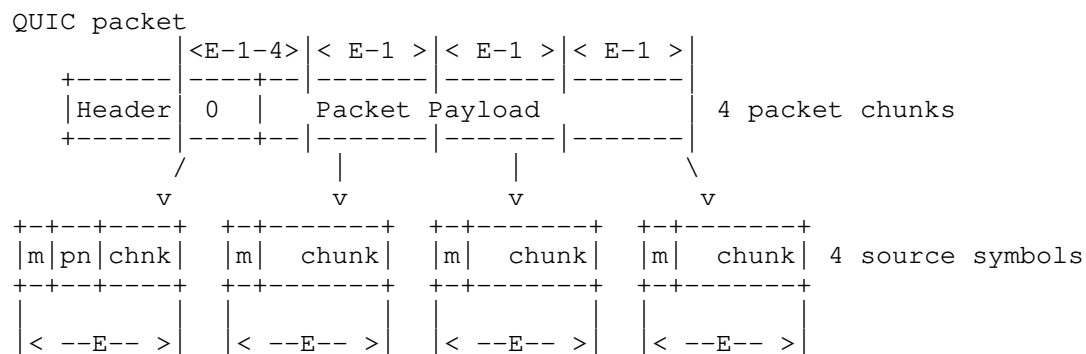


Figure 5: Example of packet chunk to source symbol mapping, when the E value is relatively small, in presence of the QUIC Packet Number for the first chunk.

Figure 5 shows an example where the 4 source symbols are created from the payload of a given QUIC packet. The first chunk may contain zero padding at the beginning in order to align the protected packet payload size to a multiple of E-1, and the first source symbol may contain the QUIC Packet Number.

Each source symbol is uniquely identified allowing to determine unambiguously its position in the source symbol flow. What information to associate to a source symbol to uniquely identify it is FEC Scheme dependent. Section 4.3 gives insight on this topic.

#### 4.2.2.1. Open questions: Content of Source Symbols Metadata? Removing certain frames from FEC protection?

NB: section to remove once fixed.

During the FEC encoding phase, additional data can be added to the source symbol. These data are only added during the encoding and MUST NOT be transmitted on the network. The encoder and decoder MUST agree on the addition of these data to the source symbol in order to avoid decoding errors. Here are some examples of data that can be added to a source symbol during encoding and that will be decoded upon a source symbol recovery:

- o The packet number: adding the packet number allows the decoder to know which packet has been recovered and potentially send a feedback of which packet has been recovered to the QUIC sender.
- o Additional QUIC frames: the FEC encoder can for example add PADDING frames to a source symbol before proceeding to encoding. Adding PADDING frames to source symbols before encoding allows

protecting packets of different sizes. The smaller packet payload will be added PADDING frames to reach a size that is a multiple of  $E-1$ .

Note: Maybe the decision of adding data such as padding in the source symbols should be left to the underlying FEC Scheme.

Besides adding data to source symbols before encoding, some frames can be removed from the source symbol if their protection is not crucial for the transmission in order to reduce the size of the source symbol. For example, ACK frames can be systematically stripped out of the source symbols. Stream frames of non-delay-sensitive streams could also be removed from the source symbol. The encoder and decoder MUST agree on which frames must be stripped out of packet payloads. This information might for example be encoded in the Source Symbol ID by the FEC encoder.

Note: We might want to propose standard ways/algorithms to add/remove data before the encoding ?

TODO: Add a mechanism to add QUIC packet identifier to the metadata. It's useful.

#### 4.2.3. Source Symbol Size ( $E$ ) Considerations

The source symbol size,  $E$ , MUST be strictly greater than zero bytes and strictly smaller than the minimum PMTU value allowed by QUIC. The packet header is not part of the FEC-protected data. When the packet payload size is not a multiple of  $E-1$ , zero-padding MUST be added at the beginning of the first chunk of the packet payload. This is equivalent to inserting PADDING frames at the beginning of the payload. This zero-padding, only used for FEC encoding, SHOULD NOT be sent on the wire.

The choice of an appropriate value for  $E$  may depends on the use case (in particular on the nature of application data). A reasonably small value reduces the expected value of the added padding needed to align the payload size with a multiple of  $E-1$ , which can be a good approach when dealing with QUIC packets whose size significantly vary. However an overly small value also increases processing complexity (FEC encoding and decoding are performed over a larger linear system since there are more source symbols), so there is an incentive to use a larger value. An appropriate balance should be found, and this choice is considered as out of scope for this document. Since a repair symbol will transit through a frame, the  $E$  value must take this into account to avoid having REPAIR frames that do not fit into a single QUIC packet.

#### 4.3. Source Symbol Signaling

An explicit signaling is needed by a decoder to identify the source symbols and their position in the block (i.e., for block codes) or coding window (i.e., for sliding window codes). While the QUIC packet number increases monotonically, it cannot be used to identify the position of a packet in the coding window as the packet number is not needed to increase by 1 for each new packet. There is thus an ambiguity on the decoder-side between lost packets and packets that do not exist. Similarly to FECFRAME, we propose to assign a identifier to source symbols to avoid this ambiguity. This identifier is opaque to the protocol and will be defined by the underlying FEC schemes. This is out of the scope of this document. An example of identifier could be an integer increasing by 1 for each new source symbol

In order to announce the source symbol identifier to the FEC decoder, we propose to add a new frame, the FEC\_SRC\_FPI frame to packets whose payload will contain one or more source symbols from the FEC decoder point of view. The FEC\_SRC\_FPI frame is part of the packet payload itself. Any packet containing a FEC\_SRC\_FPI frame MUST see its payload considered as one or more source symbol(s).

The FEC\_SRC\_FPI frame format is FEC Scheme specific and MUST be specified in the associated document.

#### 4.4. Repair Symbol Signaling

An explicit signaling is needed by a decoder for each repair symbol received through a REPAIR frame. The goals are manifold: identifying the repair symbols and their position in the block (i.e., for block codes) or coding window (i.e., for sliding window codes); carrying information on the way this repair symbol has been produced (e.g., with sliding window codes, it can indicate the encoding window composition).

One or more repair symbols can be present in a given QUIC packet. When there are multiple symbols, they SHOULD be concatenated in the same REPAIR frame. How to achieve this goal is FEC Scheme specific and therefore must be defined in the document describing this FEC Scheme.

#### 4.5. Signaling a Symbol Recovery

When all the source symbols of a given QUIC packet have been lost but are recovered during FEC decoding, a QUIC receiver SHOULD advertise it to the sender in order to avoid the retransmission of already available data. However, the QUIC receiver MUST NOT acknowledge this

recovered packet through a regular acknowledgement, as it would interfere with the behaviour of loss-based congestion controls such as [Cubic]. Therefore this document introduces a dedicated RECOVERED frame, that enables a receiver to indicate that a specific QUIC packet has been recovered through FEC decoding.

The RECOVERED frame works at the packet level. It is therefore required to be able to identify to which packet the recovered source symbols belong to. This is made possible by the QUIC packet identifier field added to the meta data prior to FEC encoding (Section 4.2.2).

#### 4.6. About Gaps in the Set of Source Symbols Considered During Encoding

A given FEC Scheme MAY support or not the presence of gaps in the set of source symbols that constitute a block (for Block codes) or an encoding window (for Sliding Window codes). A potential cause for non contiguous sets of source symbols is the acknowledgment of one of them. When this happens, the QUIC sending endpoint may want to remove this source symbol from further FEC encodings. This is particularly true with Sliding Window codes because of their flexibility during FEC encoding (i.e., the encoding window can change between two consecutive FEC encodings).

Supporting gaps can be motivated by the desire to reduce encoding and decoding complexity since there are fewer variables. However this choice has major consequences in terms of signaling. Indeed each repair symbol transmitted MUST be accompanied by enough information for the QUIC decoding endpoint to unambiguously identify the exact composition of the block or encoding window. Without any gap, the identity of the first source symbol plus the number of symbols in the block or encoding window is sufficient. With gaps, a more complex encoding needs to be used, perhaps similar to the encoding used for selective acknowledgments.

Whether gaps are supported MUST be clarified in each FEC Scheme.

#### 5. FEC Scheme Negotiation in QUIC

FEC Scheme negotiation has two goals:

- o Selecting a FEC Scheme (or FEC Schemes) that can be used by the QUIC transmission and reception endpoints. This process requires an exchange between them;
- o Communicating a certain number of parameters, the "Configuration Information", that are not expected to change over the session lifetime. For instance, this is the case of the symbol size

parameter, E (in bytes), that needs either to be agreed between the endpoints, or chosen by the sender and communicated to the receiver(s);

Editor's notes:

- \* It is likely that FEC Scheme negotiation requires the use of a new dedicated Extension Frame Type. The details remain TBD.
- \* The Negotiation Frame Type format remains TBD.
- \* How to communicate the parameters remains TBD.
- \* The present document only provides high level principles, the details are of course the responsibility of the FEC Scheme.
- \* In case negotiation is different when protecting a single versus several streams, this section may be moved to the respective sections.
- \* How does it work in case of a multicast session?
- \* Do we negotiate here a FEC Scheme on a per-Stream basis (or group of Streams to be protected jointly)? Or do we negotiate a FEC Scheme on a QUIC session basis, therefore to be used for all the Streams that need FEC protection?

## 5.1. FEC Scheme Negotiation

Before defining the transport parameters, we define two structures, `encoder_fec_scheme_t` and `decoder_fec_scheme_t`, in Figure 6. The `config` field is an opaque field allowing the decoder to define supported configuration information for the associated FEC Scheme. A FEC Scheme specification MUST define the set of valid configurations for the FEC Scheme.

```
struct {  
    varint fec_scheme_id;  
} encoder_fec_scheme_t  
  
struct {  
    varint fec_scheme_id;  
    uint16_t config_length;  
    uint8_t config[config_length];  
} decoder_fec_scheme_t
```

Figure 6: encoder\_fec\_scheme\_t and decoder\_fec\_scheme\_t structures.

The following three transport parameters are used by the QUIC endpoints to negotiate the FEC Scheme used during the connection.

- o supported\_encoder\_fec\_schemes: list of supported FEC schemes for the encoding part listed from the most to the least preferred. The value of this parameter consists in a list of encoder\_fec\_scheme\_t. When announcing a FEC Scheme, the encoder MUST be able handle every FEC Scheme configuration considered valid.
- o supported\_decoder\_fec\_schemes: list of supported FEC schemes for the decoding part listed from the most to the least preferred. The value of this parameter consists in a list of decoder\_fec\_scheme\_t, each one representing the ID of a supported FEC Scheme.
- o receiving\_symbol\_size: the size in bytes of the symbols the peer is willing to receive and recover. The value is a 16-bits integer.

Since communications can be bidirectional, each QUIC endpoint can provide the three parameters. Conversely, providing an empty list indicates this endpoint does not support FEC for the associated communication path (e.g., an empty supported\_decoder\_fec\_schemes list indicates this endpoint cannot perform FEC decoding).

The decoding FEC Scheme of a QUIC endpoint is set to the first FEC Scheme listed in its own supported\_decoder\_fec\_schemes that also appears in the peer's supported\_encoder\_fec\_schemes. The encoding FEC Scheme of a QUIC endpoint is set to the first FEC Scheme listed in the peer's supported\_decoder\_fec\_schemes that also appears in its own supported\_encoder\_fec\_schemes. The encoder-side symbol size (E) of a QUIC endpoint is set to the value announced by the peer's receiving\_symbol\_size transport parameter. The decoder-side symbol



size of a QUIC endpoint is set to the value announced in its own receiving\_symbol\_size transport parameter.

```

Host 1                                     Host 2
< -----
    supported_encoder_fec_schemes{RLC_GF256, REED_SOLOMON, XOR}
    supported_decoder_fec_schemes{REED_SOLOMON, XOR}
    receiving_symbol_size{500}

----- >
supported_encoder_fec_schemes{RLC_GF256, REED_SOLOMON, XOR}
supported_decoder_fec_schemes{RLC_GF256, REED_SOLOMON}
receiving_symbol_size{200}

ENCODER_FEC_SCHEME = REED_SOLOMON
DECODER_FEC_SCHEME = RLC_GF256
ENCODER_SYMBOL_SIZE = 500
DECODER_SYMBOL_SIZE = 200

                                ENCODER_FEC_SCHEME = RLC_GF256
                                DECODER_FEC_SCHEME = REED_SOLOMON
                                ENCODER_SYMBOL_SIZE = 200
                                DECODER_SYMBOL_SIZE = 500

```

Figure 7: Example FEC Schemes negotiation during the QUIC handshake.

It is possible that the QUIC endpoint that receives one or more FEC Scheme proposals from the initiator cannot select any of them. In that case the negotiation process fails and no FEC protection is used.

## 6. Security Considerations

TBD

## 7. IANA Considerations

TBD

## 8. Acknowledgments

TBD

## 9. References

## 9.1. Normative References

- [Cubic] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks", RFC 8312, DOI 10.17487/RFC8312, February 2018, <<https://www.rfc-editor.org/info/rfc8312>>.
- [QUIC-transport] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport (Work in Progress) (work in progress), January 2019, <<https://datatracker.ietf.org/doc/draft-ietf-quic-transport/>>.
- [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>>.

## 9.2. Informative References

- [nc-taxonomy] Roca (Ed.) et al., V., "Taxonomy of Coding Techniques for Efficient Network Communications", Request For Comments RFC 8406, June 2018, <<https://datatracker.ietf.org/doc/draft-irtf-nwcrg-network-coding-taxonomy/>>.
- [RFC5510] Lacan, J., Roca, V., Peltotalo, J., and S. Peltotalo, "Reed-Solomon Forward Error Correction (FEC) Schemes", RFC 5510, DOI 10.17487/RFC5510, April 2009, <<https://www.rfc-editor.org/info/rfc5510>>.
- [RFC8681] Roca, V. and B. Teibi, "Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME", RFC 8681, DOI 10.17487/RFC8681, January 2020, <<https://www.rfc-editor.org/info/rfc8681>>.

## Authors' Addresses

Ian Swett  
Google  
Cambridge, MA  
US

Email: [ianswett@google.com](mailto:ianswett@google.com)

Marie-Jose Montpetit  
Triangle Video  
Boston, MA  
US

Email: [marie@mjmontpetit.com](mailto:marie@mjmontpetit.com)

Vincent Roca  
INRIA  
Univ. Grenoble Alpes  
France

Email: [vincent.roca@inria.fr](mailto:vincent.roca@inria.fr)

Francois Michel  
UCLouvain  
Louvain-la-Neuve  
Belgium

Email: [francois.michel@uclouvain.be](mailto:francois.michel@uclouvain.be)