TSVWG
Internet-Draft
Intended status: Standards Track
Expires: November 24, 2018

V. Roca
B. Teibi
INRIA

May 23, 2018

# Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME

draft-ietf-tsvwg-rlc-fec-scheme-05

## Abstract

This document describes two fully-specified Forward Erasure Correction (FEC) Schemes for Sliding Window Random Linear Codes (RLC), one for RLC over GF(2) (binary case), a second one for RLC over GF(2^^8), both of them 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 November 24, 2018.

## Copyright Notice

Table of Contents

## 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 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 when used for reliable continuous media transfers over lossy networks.

The present document only focusses on the FECFRAME protocol, used in multicast/broadcast delivery mode, with 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 a end-host/server (source) or a middlebox) and a single FEC decoding point (either a 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, 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 front 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 follow a totally different approach: the Sliding Window Random Linear Codes (RLC) over either Finite Field GF(2) or GF(2^^8). 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). 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), and which is either of fixed or variable size (elastic window). Repair packets (symbols) are generated on-the-fly, 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 of each repair symbol received) are added upon receiving new packets. Variables are removed when they are too old with respect to their validity period (real-time constraints), as well as the associated equations they are involved in (Appendix B introduces an optimization that extends the time a variable is considered in the system). Lost

source symbols are then recovered thanks to this linear system whenever its rank permits it.

With RLC codes (more generally with sliding window codes), the protection of a multicast/broadcast session also needs to be dimensioned by considering the worst communication conditions one wants to support. 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:

- 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;
- the seed used by a coding coefficients generation function (Section 3.5). This information enables 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 7). Similarly, each FEC Source Packet features a fixed 32-bit long trailer, called Explicit

Source FEC Payload ID (Figure 5), that contains the ESI of the first source symbol (see the ADUI and source symbol mapping, 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 signalling information associated to each source or repair symbol. It also defines the FEC Framework Configuration Information (FFCI) carrying signalling 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", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

This document uses the following definitions and abbreviations:

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 (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)

PRNG: pseudo-random number generator

tinymt32_rand(maxv): PRNG defined in Section 3.4 and used in this specification, that returns a new random integer in [0; maxv-1]

DT: coding coefficients density threshold, an integer between 0 and 15 (inclusive) the controls the fraction of coefficients that are non zero

## 3. Procedures

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

### 3.1. Possible Parameter Derivations

The Sliding Window RLC FEC Scheme relies on several parameters:

Maximum FEC-related latency budget, max_lat (in seconds) with real-time flows:

a source ADU flow can have real-time constraints, and therefore any FECFRAME related operation must take place within the validity period of each ADU. 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 as explained below;

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 instance, at session start, upon receiving new source ADUs, the ew_size progressively increases until it reaches its maximum value, ew_max_size. We have:

ew_size <= 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 B);

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 SHOULD fix the E parameter and 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 < cr <= 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 see (it's not required to process a repair symbol at a receiver). This code rate parameter can be fixed. 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.

The FEC Schemes can be used in various manners. They can be used to protect a source ADU flow having real-time constraints, or a non-realtime source ADU flow. The source ADU flow may be a Constant Bitrate (CBR) or Variable BitRate (VBR) flow. The features of the flow (in particular its minimum/maximum bitrate) may be known or not. The FEC Schemes can also be used over the Internet or over a CBR communication path. It follows that the FEC Scheme parameters can be derived in different ways, as described in the following sections.

### 3.1.1. 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:

```
dw_max_size = (max_lat * br_in) / (8 * E)
```

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:

```
dw_max_size = (max_lat * br_out * cr) / (8 * E)
```

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 exact value having no impact on the the FEC-related latency budget. For the FEC Schemes specified in this document, in line with [Roca17], the ew_max_size SHOULD be computed with:

```
ew_max_size = dw_max_size * 0.75
```

The ew_max_size is the main parameter at a FECFRAME sender.

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:

```
dw_max_size = max_NSS_observed / 0.75
```

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

```
ls_max_size >= dw_max_size
```

It is good practice to use a larger value for ls_max_size as explained in Appendix B, which does not impact maximum latency nor interoperability. However the linear system size should not be too large for practical reasons (e.g., in order to limit computation complexity).

The particular case of session start needs to be managed appropriately. Here ew_size increases each time a new source ADU is received by the FECFRAME sender, until it reaches the ew_max_size value. 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.

### 3.1.2. Other Types of Real-Time Flow

In other configurations, a real-time source ADU flow, with a max_lat latency budget, features a variable bitrate (VBR). 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 Section 3.1.1. Considering the smallest bitrate means that the encoding window and decoding window maximum sizes estimation are pessimistic: these windows have the smallest size required to enable a decoding on-time 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:

```
max_lat_for_encoding = max_lat * 0.75
```

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 (Section 3.1.3).

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 Section 3.1.1.

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. However it is usually preferable to use a ls_max_size "too large" (which can increase computation complexity and memory requirements) than the opposite (which can reduce recovery performance).

Beyond these general guidelines, the details of how to manage these situations at a FECFRAME sender and receiver can depend on additional considerations that are out of scope of this document.

### 3.1.3. 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.

Beyond these general guidelines, the details of how to manage these situations at a FECFRAME sender and receiver can depend on additional considerations that are out of scope of this document.

## 3.2. ADU, ADUI and Source Symbols Mappings

At a sender, an ADU coming from the application cannot directly be 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). At a sender, this identifier is prepended to each ADU before FEC encoding. This way, FEC decoding at a receiver also recovers this Flow ID and a recovered ADU can be assigned to the right source flow (note that transport port numbers and IP addresses cannot be used to that purpose as they are not recovered during FEC decoding).

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 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.

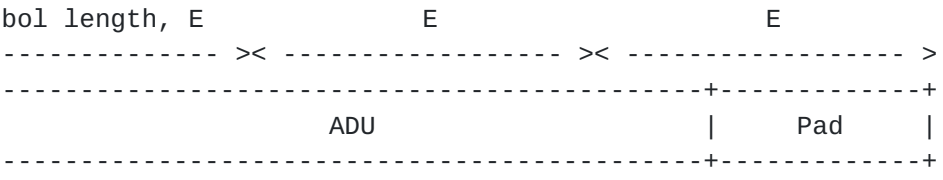


Figure 1: ADUI Creation example (here 3 source symbols are created for this ADUI).

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:

- 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;
- 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. Pseudo-Random Number Generator (PRNG)

The RLC FEC Schemes defined in this document rely on the TinyMT32 PRNG, a small-sized variant of the Mersenne Twister PRNG, as defined in the reference implementation version 1.1 (2015/04/24) by Mutsuo Saito (Hiroshima University) and Makoto Matsumoto (The University of Tokyo).

- Official web site: <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/>
- Official github site and reference implementation: <https://github.com/MersenneTwister-Lab/TinyMT>

For the RLC FEC Schemes defined in this document, the tinymt32 32-bit version (rather than the 64-bit version) MUST be used. This PRNG requires a parameter set that needs to be pre-calculated. For the RLC FEC Schemes defined in this document, the following parameter set MUST be used:

- mat1 = 0x8f7011ee = 2406486510;

- mat2 = 0xfc78ff1f = 4235788063;
- tmat = 0x3793fdff = 932445695.

This parameter set is the first entry of the precalculated parameter sets in file tinymt32dc.0.1048576.txt, by Kenji Rikitake, and available at:

- <https://github.com/jj1bdx/tinymtdc-longbatch/blob/master/tinymt32dc/tinymt32dc.0.1048576.txt>.

This is also the parameter set used in [KR12].

The PRNG reference implementation is distributed under a BSD license and excerpts of it are reproduced in Appendix A. In order to validate an implementation of this PRNG, using seed 1, the 10,000th value returned by: tinymt32_rand(s, 0xffff) MUST be equal to 0x7c37.

This PRNG MUST first be initialized with a 32-bit unsigned integer, used as a seed. The following function is used to this purpose:

```
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 equal to 0), since this is the Repair_Key 16-bit field value of the Repair FEC Payload ID (Section 4.1.3). 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 maxv-1 inclusive is needed, the following function is used:

```
uint32_t tinymt32_rand (tinymt32_t * s, uint32_t maxv);
```

This function takes as parameter both a pointer to the same tinymt32_t structure (that needs to be left unchanged between successive calls to the function) and the maxv value.

## 3.5. 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. When DT equals 15, the maximum value, the function guaranties that all coefficients are non zero (i.e., maximum density). When DT is between 0 (minimum

value) and strictly inferior to 15, the average probability of having a non zero coefficient equals (DT +1) / 16.

These considerations apply 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 MUST be equal to 1. With RLC over GF(2^^8) FEC Scheme (Section 4), m MUST be equal to 8.

<CODE BEGINS>

```
/*
 * 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=2 and dt=15
 * (in) 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 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 an error code
 */
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 SOMETHING_WENT_WRONG; /* 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++) {
                if (tinymt32_rand(&s, 16) <= dt) {
                    cc_tab[i] = (uint8_t) 1;
                } else {
                    cc_tab[i] = (uint8_t) 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_rand(&s, 256);
                } while (cc_tab[i] == 0);
            }
        } else {
            /* here a certain fraction of coefficients should be 0 */
            for (i = 0 ; i < cc_nb ; i++) {
                if (tinymt32_rand(&s, 16) <= dt) {
                    do {
                        cc_tab[i] = (uint8_t) tinymt32_rand(&s, 256);
                    } while (cc_tab[i] == 0);
                } else {
                    cc_tab[i] = 0;
                }
            }
        }
        break;

    default:
        /* bad parameter m */
        return SOMETHING_WENT_WRONG;
    }
    return EVERYTHING_IS_OKAY;
}
<CODE ENDS>
```

Figure 2: Coding Coefficients Generation Function pseudo-code

## 3.6. Finite Fields Operations

### 3.6.1. 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 MUST be used for GF(2^^8):

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

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

### 3.6.2. 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:

```
repair[i] = cc_tab[0] * src_0[i] + cc_tab[1] * src_1[i] + ... +
cc_tab[ew_size - 1] * src_ew_size_1[i]
```

where * is the multiplication over GF(2^^8) and + is an XOR operation. 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 ADU 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 FCCI 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

- 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;

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

If another mechanism requires the FSSI to be carried as an opaque octet string (for instance, after a Base64 encoding), the encoding format consists of the following 2 octets:

Encoding symbol length (E): 16-bit field.

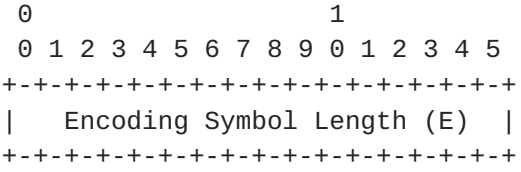


Figure 3: FSSI Encoding Format

### 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 4.

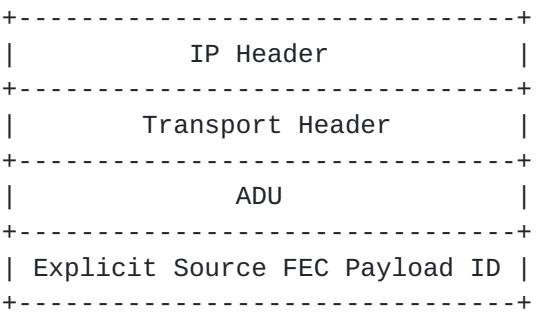


Figure 4: Structure of an FEC Source Packet with the Explicit Source FEC Payload ID

More precisely, the Explicit Source FEC Payload ID is composed of the following field (Figure 5):

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.

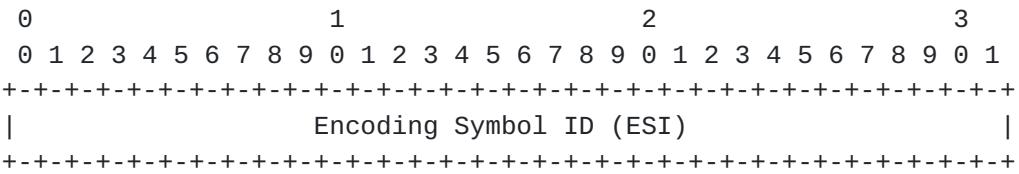


Figure 5: Source FEC Payload ID Encoding Format

### 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 6.

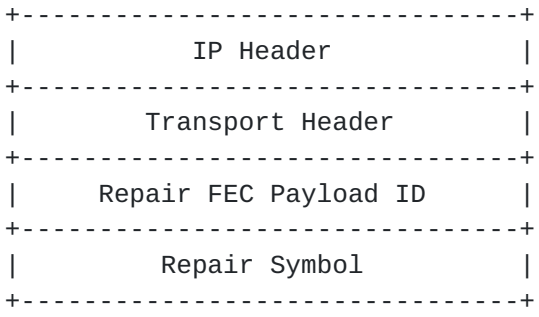


Figure 6: Structure of an FEC Repair Packet with the Repair FEC Payload ID

More precisely, the Repair FEC Payload ID is composed of the following fields (Figure 7):

Repair_Key (16-bit field): this unsigned integer is used as a seed by the coefficient generation function (Section 3.5) 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.

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 3.5. 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;

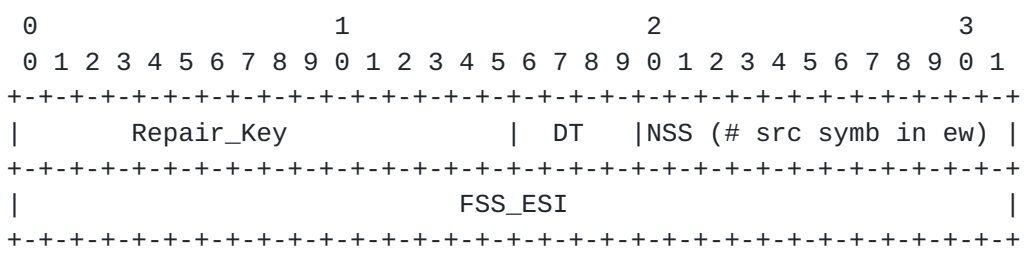


Figure 7: Repair FEC Payload ID Encoding Format

### 4.1.4. Additional Procedures

The following procedure applies:

- 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 32-bit value.

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

- 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 4.1.1.2 apply here.

### 5.1.3. Repair FEC Payload ID

All the considerations of Section 4.1.1.2 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 it is RECOMMENDED that the sender use value 0 for the Repair_Key field, but a receiver SHALL ignore this field.

### 5.1.4. Additional Procedures

All the considerations of Section 4.1.1.2 apply here.

## 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 3.5) 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 3.6). 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.

### 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 Section 3.5), 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. 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 B 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:

- Organisation: Inria
- 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]).
- Maturity: prototype.
- Coverage: this software complies with the Sliding Window RLC FEC Scheme.
- Licensing: proprietary.
- Contact: vincent.roca@inria.fr

## 8. Security Considerations

The FEC Framework document [RFC6363] provides a 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):

- FEC Encoding ID: changing this parameter leads the receivers to consider a different FEC Scheme, which enables an attacker to create a Denial of Service (DoS);
- Encoding symbol length (E): setting this E parameter to a different value will confuse the receivers and create a DoS. More precisely, the FEC Repair Packets received will probably no longer be multiple of E, leading receivers to reject them;

It is therefore RECOMMENDED that security measures are 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: by modifying the Encoding Symbol ID (ESI), or the repair key, NSS or FSS_ESI. 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.

## 9. Operations and Management Considerations

The FEC Framework document [RFC6363] provides a 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^^8)

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^^8) 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]). Of course the CPU overhead will increase with the encoding window size, because more operations in the GF(2^^8) 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^^8), 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 compute the XOR sum of all the source symbols in the encoding window. In that case: (1) 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:

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

## 11. Acknowledgments

The authors would like to thank Jonathan Detchart, Gorry Fairhurst, and Marie-Jose Montpetit for their valuable feedbacks on this document.

## 12. References

### 12.1. Normative References

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

[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>.

### 12.2. Informative References

[KR12] Rikitake, K., "TinyMT Pseudo Random Number Generator for Erlang", ACM 11th SIGPLAN Erlang Workshop (Erlang'12), September 14, 2012, Copenhagen, Denmark, DOI: http://dx.doi.org/10.1145/2364489.2364504, September 2012.

[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>.

[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>.

[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>.

[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 Pseudo-Random Number Generator

The TinyMT32 PRNG reference implementation is distributed under a BSD license by the authors and excerpts of it are reproduced in Figure 8. Differences with respect to the original source code are the following:

- unused parts of the original source code have been removed;
- the appropriate parameter set has been added to the initialisation function;
- function tinymt32_rand() has been added;
- function order has been changed;
- certain internal variables have been renamed for compactness purposes.

```
<CODE BEGINS>
/**
 * Tiny Mersenne Twister only 127 bit internal state
 *
 * Authors : Mutsuo Saito (Hiroshima University)
 *           Makoto Matsumoto (University of Tokyo)
 *
 * Copyright (c) 2011, 2013 Mutsuo Saito, Makoto Matsumoto,
 * Hiroshima University and The University of Tokyo.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 *   - Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *   - Redistributions in binary form must reproduce the above
 *     copyright notice, this list of conditions and the following
 *     disclaimer in the documentation and/or other materials
 *     provided with the distribution.
 *   - Neither the name of the Hiroshima University nor the names of
 *     its contributors may be used to endorse or promote products
 *     derived from this software without specific prior written
 *     permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
 * CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
```

```
 * TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON
 * ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
 * TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
 * THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */

/**
 * tinymt32 internal state vector and parameters
 */
typedef struct {
    uint32_t status[4];
    uint32_t mat1;
    uint32_t mat2;
    uint32_t tmat;
} tinymt32_t;

static void tinymt32_next_state (tinymt32_t * s);
static uint32_t tinymt32_temper (tinymt32_t * s);
static double tinymt32_generate_32double (tinymt32_t * s);

/**
 * Parameter set to use for RLC FEC Schemes. Do not change.
 */
#define TINYMT32_MAT1_PARAM     0x8f7011ee
#define TINYMT32_MAT2_PARAM     0xfc78ff1f
#define TINYMT32_TMAT_PARAM     0x3793fdff


/**
 * This function initializes the internal state array with a 32-bit
 * unsigned integer seed.
 * @param s     tinymt state vector.
 * @param seed  a 32-bit unsigned integer used as a seed.
 */
void tinymt32_init (tinymt32_t * s, uint32_t seed)
{
#define MIN_LOOP 8
#define PRE_LOOP 8
    s->status[0] = seed;
    s->status[1] = s->mat1 = TINYMT32_MAT1_PARAM;
    s->status[2] = s->mat2 = TINYMT32_MAT2_PARAM;
    s->status[3] = s->tmat = TINYMT32_TMAT_PARAM;
    for (int i = 1; i < MIN_LOOP; i++) {
        s->status[i & 3] ^= i + UINT32_C(1812433253)
            * (s->status[(i - 1) & 3]
               ^ (s->status[(i - 1) & 3] >> 30));
```

```
        }
        for (int i = 0; i < PRE_LOOP; i++) {
            tinymt32_next_state(s);
        }
    }

    /**
     * This function outputs an integer in the [0 .. maxv-1] range.
     * @param s     tinymt internal status
     * @return      floating point number r (0.0 <= r < 1.0)
     */
    uint32_t tinymt32_rand (tinymt32_t * s, uint32_t maxv)
    {
        return (uint32_t)(tinymt32_generate_32double(s) * (double)maxv);
    }


    /**
     * Internal tinymt32 constants and functions.
     * Users should not call these functions directly.
     */
    #define TINYMT32_MEXP 127
    #define TINYMT32_SH0 1
    #define TINYMT32_SH1 10
    #define TINYMT32_SH8 8
    #define TINYMT32_MASK UINT32_C(0x7fffffff)
    #define TINYMT32_MUL (1.0f / 16777216.0f)

    /**
     * This function changes internal state of tinymt32.
     * @param s     tinymt internal status
     */
    static void tinymt32_next_state (tinymt32_t * s)
    {
        uint32_t x;
        uint32_t y;

        y = s->status[3];
        x = (s->status[0] & TINYMT32_MASK)
            ^ s->status[1]
            ^ s->status[2];
        x ^= (x << TINYMT32_SH0);
        y ^= (y >> TINYMT32_SH0) ^ x;
        s->status[0] = s->status[1];
        s->status[1] = s->status[2];
        s->status[2] = x ^ (y << TINYMT32_SH1);
        s->status[3] = y;
        s->status[1] ^= -((int32_t)(y & 1)) & s->mat1;
```

```
    s->status[2] ^= -((int32_t)(y & 1)) & s->mat2;
}

/**
 * This function outputs 32-bit unsigned integer from internal state.
 * @param s     tinymt internal status
 * @return      32-bit unsigned pseudos number
 */
static uint32_t tinymt32_temper (tinymt32_t * s)
{
    uint32_t t0, t1;
    t0 = s->status[3];
    t1 = s->status[0] + (s->status[2] >> TINYMT32_SH8);
    t0 ^= t1;
    t0 ^= -((int32_t)(t1 & 1)) & s->tmat;
    return t0;
}

/**
 * This function outputs double precision floating point number from
 * internal state. The returned value has 32-bit precision.
 * In other words, this function makes one double precision floating
 * point number from one 32-bit unsigned integer.
 * @param s     tinymt internal status
 * @return      floating point number r (0.0 <= r < 1.0)
 */
static double tinymt32_generate_32double (tinymt32_t * s)
{
    tinymt32_next_state(s);
    return (double)tinymt32_temper(s) * (1.0 / 4294967296.0);
}
<CODE ENDS>
```

Figure 8: TinyMT32 pseudo-code

## Appendix B. Decoding Beyond Maximum Latency Optimization

This annex introduces non normative considerations. They are 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 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 good receiver, i.e., a receiver that benefits from a protection that is significantly sufficient to recover from the packet losses, can choose to reduce its ls_max_size significantly. In that case lost ADUs will be recovered rapidly, without relying on this optimization.

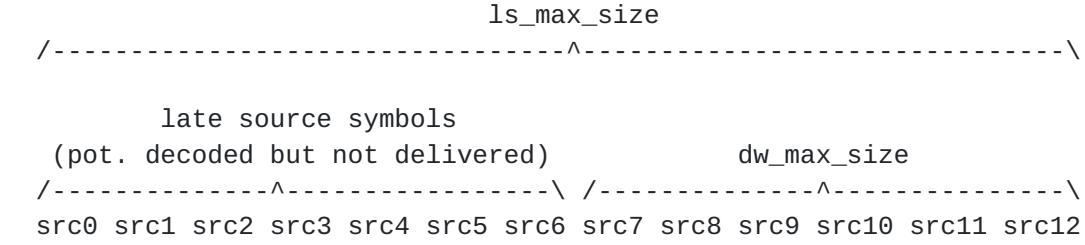


Figure 9: Relationship between parameters to decode beyond maximum latency.

It means that source symbols, and therefore ADUs, may be decoded even if the added latency exceeds the maximum value permitted by the application. 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 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

Belkacem Teibi
INRIA
Univ. Grenoble Alpes
France

EMail: belkacem.teibi@inria.fr