Coding techniques for satellite systems
draft-irtf-nwcrg-network-coding-satellites-07

Abstract

This document is the product of the Coding for Efficient Network Communications Research Group (NWCRG). This document follows the taxonomy document [RFC8406] and considers coding as a linear combination of packets that operate in and above the network layer. In this context, this memo details a multi-gateway satellite system to identify use-cases where coding is relevant. As example, coding operating in and above the network layer can be exploited to cope with residual losses or provide reliable multicast services. The objective is to contribute to a larger deployment of such techniques in SATCOM systems. This memo also identifies open research issues related to the deployment of coding in SATCOM systems, such as the interaction between congestion controls and coding techniques.

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 1, 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. A note on satellite topology ............................ 4
3. Use-cases for improving the SATCOM system performance with coding techniques .......................... 6
   3.1. Two-way relay channel mode .......................... 6
   3.2. Reliable multicast ................................... 7
   3.3. Hybrid access ....................................... 8
   3.4. Dealing with LAN losses ............................. 8
   3.5. Dealing with varying channel conditions .......... 9
   3.6. Improving the gateway handovers .................... 10
4. Research challenges .................................... 10
   4.1. On the joint-use of coding techniques and congestion control in SATCOM systems .............. 11
   4.2. On the efficient usage of satellite resource ........ 11
   4.3. Interaction with virtualized satellite gateways and terminals .................................. 11
   4.4. Delay/Disruption Tolerant Networks .................. 12
5. Conclusion ............................................. 12
6. Glossary ................................................. 12
7. Acknowledgements ....................................... 14
8. IANA Considerations ..................................... 14
9. Security Considerations .................................. 14
10. Informative References ................................. 14
Authors’ Addresses ......................................... 16

1. Introduction

This document is the product of and represents the collaborative work and consensus of the Coding for Efficient Network Communications Research Group (NWCRG); it is not an IETF product and is not a standard. A glossary is proposed in Section 6.

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 adds to cope with link impairments, without
reducing the goodput when the channel quality is good. There is usually enough redundancy to guarantee a Quasi-Error Free transmission. The recovery time depends on the encoding block size. Considering for instance geostationary satellite system (GEO), physical or link layer erasure coding mechanisms recover transmission losses within a negligible delay compared to link delay. However, when retransmissions are triggered, this leads to a non-negligible additional delay in particular over GEO link. Further exploiting coding techniques at application or transport layers is an opportunity for releasing constraints on the physical layer and improving the performance of SATCOM systems.

The notations used in this document are based on the taxonomy document [RFC8406]:

- Channel and link codings are gathered in the PHY layer coding and are out of the scope of this document. It focuses on situations where coding is not widely deployed in current SATCOM systems.

- FEC (also called Application-Level FEC) operates in and above the network layer.

- This document considers coding (or coding techniques or coding schemes) as a linear combination and not as a content coding (e.g., to compress a video flow).

Figure 1 presents the status of the reliability schemes deployment in satellite systems.

- X1 embodies the source coding techniques that could be used at application level for instance within QUIC. This is not specific to SATCOM systems since such deployment can be relevant for broadband Internet access discussions.

- X2 embodies the physical-layer techniques exploited in SATCOM systems (note that other coding techniques can be exploited). This is out of the scope of the document.
**Figure 1: Reliability schemes in current satellite systems**

We notice an active research activity on coding techniques and SATCOM. That being said, not much has actually made it to industrial developments. In this context, this document aims at identifying opportunities for further usage of coding in these systems.

2. A note on satellite topology

This section describes a satellite system that follows the ETSI DVB standards to provide broadband Internet access. A high-level description of a multi-gateway satellite network is provided. There are 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. In this context, the increase of the available capacity that is carried out to end users and reliability requirements lead to multiple gateways for one unique satellite platform.

In this context, Figure 2 shows an example of a multi-gateway satellite system. 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 discussions and a generic SATCOM ground segment architecture for bidirectional Internet access can be found in [SAT2017].

Some functional blocks aggregate the traffic of multiple users.

<table>
<thead>
<tr>
<th>Source coding</th>
<th>Network</th>
<th>Packetization</th>
<th>PHY layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2E NC</td>
<td>X1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IntraF</td>
<td>X1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPC</td>
<td>X1</td>
<td></td>
<td>X2</td>
</tr>
<tr>
<td>MPC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 2: Data plane functions in a generic satellite multi-gateway system. More details can be found in DVB standard documents.
3. Use-cases for improving the SATCOM system performance with coding techniques

This section details use-cases where coding techniques could provide interesting features for SATCOM systems. Combination of the presented use-cases could also be relevant.

It is worth noting that these use-cases mostly focus on the middleware and packetization UDP/IP of Figure 1. There are already lots of recovery mechanisms at the physical-layer in currently deployed systems while E2E source coding is done at the application level. In a multi-gateway SATCOM Internet access, the deployment opportunities are more relevant in specific SATCOM components such as the "network function" block or the "access gateway" of Figure 2.

3.1. Two-way relay channel mode

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

Satellite terminal A sends a flow A and satellite terminal B sends a flow B to a coding server. The coding server sends a combination of both terminal flows. This results in non-negligible capacity savings and has been demonstrated [ASMS2010]. In the proposed example, a dedicated coding server is introduced. Its location could be changed depending on the deployment use-case. With On-Board Processing satellite payloads, the coding operations could be done at the satellite level; although this would require lots of computational resource on-board and may not be relevant with today’s payloads.

\[-X\] - : traffic from satellite terminal X to the server
\[=\{X+Y=\] : traffic from X and Y combined sent from the server to terminals X and Y

\[\begin{align*}
\text{Sat term A} & \quad \longrightarrow \longrightarrow \\
\text{Sat term B} & \quad \longrightarrow \longrightarrow \\
\text{Coding} & \quad \longrightarrow \longrightarrow \\
\text{Gateway} & \quad \longrightarrow \longrightarrow \\
\text{Server} & \quad \longrightarrow \longrightarrow \\
\end{align*}\]

\[\begin{align*}
\text{Sat term A} & \quad \longrightarrow \longrightarrow \\
\text{Sat term B} & \quad \longrightarrow \longrightarrow \\
\text{Coding} & \quad \longrightarrow \longrightarrow \\
\text{Gateway} & \quad \longrightarrow \longrightarrow \\
\text{Server} & \quad \longrightarrow \longrightarrow \\
\end{align*}\]

Figure 3: Network architecture for two way relay channel with NC
3.2. Reliable multicast

Using multicast servers is a way to better exploit the satellite broadcast capabilities. This approach is proposed in the SHINE ESA project [I-D.vazquez-nfvrg-netcod-function-virtualization] [SHINE]. This use-case considers adding redundancy to a multicast flow depending on what has been received by different end-users, resulting in non-negligible scarce resource saving. We propose an illustration for this scenario in Figure 4.

-Li}~ : packet indicating the loss of packet i of a multicast flow M

={M=} : multicast flow including the missing packets

Figure 4: Network architecture for a reliable multicast with NC

A multicast flow (M) is forwarded to both satellite terminals A and B. However packet Ni (resp. Nj) gets lost at terminal A (resp. B), and terminal A (resp. B) returns a negative acknowledgment Li (resp. Lj), indicating that the packet is missing. Then either the access gateway or the multicast server includes a repair packet (rather than the individual Ni and Nj packets) in the multicast flow to let both terminals recover from losses.

This could be achieved by using other multicast or broadcast systems, such as NACK-Oriented Reliable Multicast (NORM) [RFC5740] or File Delivery over Unidirectional Transport (FLUTE) [RFC6726]. Note that both NORM and FLUTE are limited to block coding, none of them supporting sliding window encoding schemes [RFC8406]. Note that although FLUTE is defined as an unidirectional protocol, the RFC proposes a bidirectional communication method to enable full reliability transfer and for security purposes.
3.3. Hybrid access

This use-case considers the use of multiple path management with coding at the transport layer to increase the reliability and/or the total capacity (using multiple paths does not guarantee an improvement of both the reliability and the total capacity). 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 [ETSI TR2017]. This kind of architecture is also discussed in the TCPM working group [I-D.ietf-tcpm-converters].

To cope with packet loss (due to either end-user mobility or physical-layer impairments), coding techniques could be introduced both at the CPE and at the concentrator. Apart from packet losses, other gains could be envisioned, such as a better tolerance to out-of-order packets which occur when exploited links exhibit high asymmetry in terms of RTT. Depending on the ground architecture [I-D.chin-nfvrg-cloud-5g-core-structure-yang] [SAT2017], some equipments might be hosting both SATCOM and cellular functions.

Figure 5: Network architecture for an hybrid access using coding

3.4. Dealing with LAN losses

This use-case considers the usage of coding techniques to cope with cases where the end user connects to the satellite terminal with a Wi-Fi link that exhibits losses. In the case of encrypted end-to-end applications based on UDP, PEP cannot operate. The Wi-Fi losses result in an end-to-end retransmission that would harm the quality of experience of the end user.

The architecture is recalled in Figure 6.
In this use-case, adding coding techniques could prevent the end-to-end retransmission from occurring.

-{}- : bidirectional link
-''- : Wi-Fi link
C : where coding techniques could be introduced

Figure 6: Network architecture for dealing with LAN losses

3.5. Dealing with varying channel conditions

This use-case considers the usage of coding techniques to cope with cases where channel condition can change in less than a second and where the physical-layer codes could not efficiently guarantee a Quasi-Error-Free (QEF) transmission.

The architecture is recalled in Figure 7. In these cases, the mechanisms that are exploited to adapt the physical-layer codes (Adaptive Coding and Modulation (ACM)) may adapt the modulation and coding in time: remaining errors could be recovered with higher layer redundancy packets. Coding may be applied on IP packets or on layer-2 proprietary format packets.

This use-case is mostly relevant when mobile users are considered or when the chosen band induces 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 coding techniques is different.

Figure 7: Network architecture for dealing with varying link characteristics
3.6. 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 or if the algorithm that is exploited to trigger gateway handovers shows a non negligible probability of missed detection, this may result in packet losses. During these critical phases, coding can be added to improve the reliability of the transmission and allow a seamless gateway handover. Coding could be applied at either the access gateway or the network function block. A potential control plane is in charge of taking the decision to change the communication gateway and the consequent routes.

Figure 8 illustrates this use-case.

Figure 8: Network architecture for dealing with gateway handover schemes

4. Research challenges

This section proposes a few potential approaches to introduce and use coding techniques in SATCOM systems.
4.1. On the joint-use of coding techniques and congestion control in SATCOM systems

SATCOM systems typically feature Performance Enhancing Proxy (PEP) RFC 3135 [RFC3135]. PEPs usually split TCP end-to-end connections and forward TCP packets to the satellite baseband gateway that deals with layer-2 and layer-1 encapsulations. PEP contributes to mitigate congestion in a SATCOM systems. PEP could host coding mechanisms and thus support use-cases that have been discussed in this document.

Deploying coding schemes at the TCP level in these equipment could be relevant and independent from the specific characteristics of a SATCOM link. This leads to research questions on the interaction between coding schemes and TCP congestion controls.

4.2. On the efficient usage of satellite resource

The recurrent trade-off in SATCOM systems remains: how much overhead from redundant reliability packets can be introduced to guarantee a better end-user QoE while optimizing capacity usage? At which layer this supplementary coding could be added?

This problem has been tackled in the past for physical-layer code, but there remains questions on how to adapt the overhead for, e.g., the quickly varying channel conditions use-case where ACM may not be reacting quickly enough.

4.3. Interaction with virtualized satellite gateways and terminals

Related to the foreseen virtualized network infrastructure, coding techniques could be easily deployed as VNF. Next generation of SATCOM ground segments could rely on a virtualized environment. This trend can also be seen in cellular networks, making these discussions extendable to other deployment scenarios [I-D.chin-nfvrg-cloud-5g-core-structure-yang]. As one example, the coding VNF deployment in a virtualized environment is presented in [I-D.vazquez-nfvrg-netcod-function-virtualization].

A research challenge would be the optimization of the NFV service function chaining, considering a virtualized infrastructure and other SATCOM specific functions, to guarantee efficient radio usage and easy-to-deploy SATCOM services. Moreover, another challenge related to a virtualized SATCOM equipment is the management of limited buffered capacities.
4.4. Delay/Disruption Tolerant Networks

Communications among deep-space platforms and terrestrial gateways can be a challenge. Reliable end-to-end (E2E) communications over such paths must cope with long delay and frequent link disruptions; indeed, contemporaneous E2E connectivity may be available only intermittently or never. 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. Moreover, DTN can also be seen as an alternative solution to transfer the data between a central PEP and a remote PEP.

Coding enables E2E reliable communication over DTN with adaptive re-encoding, as proposed in [THAI15]. In this case, the use-cases proposed in Section 3.5 would legitimize the usage of coding within the DTN stack to improve the channel utilization and the E2E transmission delay. In this context, the use of erasure coding techniques inside a Consultative Committee for Space Data Systems (CCSDS) architecture has been specified in [CCSDS-131.5-O-1]. A research challenge would be on how such coding can be integrated in the IETF DTN stack.

5. Conclusion

This document discusses some opportunities to introduce coding techniques at a wider scale in satellite telecommunications systems. Even though this document focuses on satellite systems, it is worth pointing out that some scenarios proposed may be relevant to other wireless telecommunication systems. As one example, the generic architecture proposed in Figure 2 may be mapped to cellular networks as follows: the ‘network function’ block gathers 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.

6. Glossary

The glossary of this memo extends the glossary of the taxonomy document [RFC8406] as follows:

- ACM: Adaptive Coding and Modulation;
- 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;

- CPE: Customer Premises Equipment;
- COM: COMmunication;
- DSL: Digital Subscriber Line;
- DTN: Delay/Disruption Tolerant Network;
- DVB: Digital Video Broadcasting;
- E2E: End-to-end;
- ETSI: European Telecommunications Standards Institute;
- FEC: Forward Error Correction;
- FLUTE: File Delivery over Unidirectional Transport;
- IntraF: Intra-Flow Coding;
- InterF: Inter-Flow Coding;
- IoT: Internet of Things;
- LTE: Long Term Evolution;
- MPC: Multi-Path Coding;
- NC: Network Coding;
- NFV: Network Function Virtualization;
- NORM: NACK-Oriented Reliable Multicast;
- PEP: Performance Enhancing Proxy [RFC3135] - a typical PEP for satellite communications include compression, caching and TCP acceleration;
- PLFRAME: Physical Layer FRAME - modulated version of a BBFRAME with additional information (e.g., related to synchronization);
- QEF: Quasi-Error-Free;
7. Acknowledgements

Many thanks to John Border, Stuart Card, Tomaso de Cola, Vincent Roca, Lloyd Wood and Marie-Jose Montpetit for their help in writing this document.

8. IANA Considerations

This memo includes no request to IANA.

9. Security Considerations

Security considerations are inherent to any access network, and in particular SATCOM systems. The use of FEC or Network Coding in SATCOM also comes with risks (e.g., a single corrupted redundant packet may propagate to several flows when they are protected together in an Inter-Flow coding approach, see section Section 3). However this is not specific to the SATCOM use-case and this document does not further elaborate on it.

10. Informative References


"Satellite Earth Stations and Systems (SES); Multi-link routing scheme in hybrid access network with heterogeneous links", ETSI TR 103 351, 2017.

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.


Internet-Draft    Coding techniques for satellite systems    October 2019


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
Generic Application Programming Interface (API) for Sliding Window FEC Codes
draft-roca-nwcrg-generic-fec-api-05

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 September 28, 2019.
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:

- sliding window codes, that manage an encoding window (of fixed or variable size) that slides over the set of source symbols at the sender. On the opposite, block codes (e.g., Reed-Solomon, LDPC, Raptor(Q)) are out of scope;
- codes that are restricted to use-cases that involve a single encoding point and a single decoding point (i.e., FEC operations are carried out either within the end-hosts or middle-boxes), as
well as codes that can be used with use-cases that involve in-
network re-coding operations;

- use-cases that are limited to an intra-flow coding (simple case),
as well as use-cases that involve inter-flow coding. This second
case is more complex to address (e.g., with questions such as how
to identify a packet of a flow?) however this is the
responsibility of the application or protocol using this codec and
not the codec itself. This aspect is therefore transparent to the
API;

- use-cases that are limited to single-path communications and use-
cases that consider multi-path communications. Here also this is
a usage consideration that is transparent to the API;

- use-cases that involve a dynamic adaptation of the codec
parameters (e.g., its code rate because the communication path
losses is known thanks to feedbacks and an appropriate strategy
can be defined);

- fixed code rate or not FEC codes, including rateless codes where
the number of repair symbols that can be generated is huge (in
theory unlimited);

- ideal (MDS) or non-ideal (non-MDS) codes. However most of the
time, sliding window codes are non-ideal codes, meaning that
slightly more than l repair symbols may be required to recover all
the l lost source symbols;

A key question is to determine what mechanisms are included in the
codec and what mechanisms are left to the responsibility of the
caller (i.e., an application or a protocol making use of this codec)
(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.
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:

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

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

- session management (sender and receiver);
- encoding window management (sender and receiver);
- set/get/generate coding coefficients (sender and receiver);
- build coded symbol (sender only);
- 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.

```c
/**
 * 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,
```
```c
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 */
    SWIF_CODEPOINT_AAA_CODEC,
    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;
```
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;

typedef struct swif_decoder {
    swif_codepoint_t        codepoint;

    /* 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. */
} swif_decoder_t;
/* 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;

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

swif_encoder_t* swif_encoder_create (swif_codepoint_t codepoint,
                      uint32_t verbosity,
                      uint32_t symbol_size,
                      uint32_t max_coding_window_size);

swif_status_t   swif_encoder_release (swif_encoder_t*        enc);

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! */

Encoder API proposal

<CODE BEGINS>

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

```c
/**
 * 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 resources.
 * @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 (
Decoder API proposal

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 keeps 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.

```c
/**<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);

/**<CODE BEGINS>
/**
 * 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
 */
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_esi);
</CODE BEGINS>
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);

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:

- the application provides the list of coding coefficients to use  
  for the next build_repair_symbol();  
- 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();  
- 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;

```
+ 1. 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.
+ 1. Encoder: calling one of them MUST be done before calling build_repair_symbol().
+ 1. Decoder: calling one of them MUST be done before calling decode_with_new_repair_symbol().

+ 1. Encoder: this function specifies the coding coefficients chosen by the application if this is the way the codec works.
+ 1. Decoder: communicate with this function the coding coefficients associated to a repair symbol and carried in the packet header.

+ 1. @param coding_coefs_tab
+ 1. (IN) table of coding coefficients associated to each of the source symbols currently in the encoding window.
+ 1. 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.
+ 1. @param nb_coefs_in_tab
+ 1. (IN) number of entries (i.e., coefficients) in the table.

+ 1. @return

```

```c
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);

```

+ 1. 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
* \texttt{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
*/

\texttt{swif\_status\_t swif\_encoder\_generate\_coding\_coefs (}
\texttt{swif\_encoder\_t* enc,}
\texttt{uint32\_t key,}
\texttt{uint32\_t add\_param);}

\texttt{swif\_status\_t swif\_decoder\_generate\_coding\_coefs (}
\texttt{swif\_decoder\_t* dec,}
\texttt{uint32\_t key,}
\texttt{uint32\_t add\_param);}

/**
* This function enables the application to retrieve the set of coding
* coefficients generated and used by \texttt{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 \texttt{coeffs\_tab[]}.}
* @return
/**
 * 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);
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


8.2. Informative References


Authors’ Addresses

Vincent Roca
INRIA
Univ. Grenoble Alpes
France
EMail: vincent.roca@inria.fr

Jonathan Detchart
ISAE - Supaero
France
EMail: jonathan.detchart@isae-supaero.fr
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 January 9, 2020.
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 ................................ 5
   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 Selection Process .......................... 13
   5.2. FEC Scheme Configuration Information ................... 14
6. Security Considerations ....................................... 14
7. IANA Considerations ........................................ 14
8. Acknowledgments ............................................. 14
9. References .................................................. 14
   9.1. Normative References .................................. 14
   9.2. Informative References ................................. 15
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:

- 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;
- 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;
- 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;
- 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 [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.

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.

```
< E-1 > | < E-1 > | < E-1 > | < E-1 > |
```

| QUIC pkt 0 | Header | Packet Payload | chunks 0, 1, 2, 3 |
| QUIC pkt 1 | Header | 0 | Packet Payload | chunks 4, 5, 6 |
| QUIC pkt 2 | Header | 0 | Packet Payload | chunks 7, 8, 9 |

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 symbols 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 indicated 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.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

```
<table>
<thead>
<tr>
<th>meta data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet Number (4 bytes)</td>
</tr>
</tbody>
</table>
| Packet chunk (E-1-4 bytes) | ...
```

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:

- 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.

- 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 manyfold: 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 acknowledgment, 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 metadata 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 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.

5. FEC Scheme Negotiation in QUIC

FEC Scheme negotiation has two goals:

- 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;

- 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 Selection Process

Let us consider the FEC Scheme selection process between the QUIC endpoints. Figure 6 illustrates the principle when a QUIC reception endpoint initiates the exchange.

<table>
<thead>
<tr>
<th>QUIC sender</th>
<th>QUIC receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; ------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>supported_fec_scheme_32b{FEC_Encoding_ID1</td>
<td>other}</td>
</tr>
<tr>
<td>supported_fec_scheme_64b{FEC_Encoding_ID2</td>
<td>other}</td>
</tr>
<tr>
<td>-------------------------------------------------&gt;</td>
<td></td>
</tr>
<tr>
<td>supported_fec_scheme_32b(FEC_Encoding_ID1</td>
<td>other)</td>
</tr>
</tbody>
</table>

Figure 6: 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 5.2).

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...

Editor's notes:

* So what? How does it finishes? Consequences?

* Can the second QUIC endpoint change the proposed static parameters? In that case can the initiator refuse?

5.2. FEC Scheme Configuration Information

Let us now focus on the communication of configuration information specific to the selected FEC Scheme. In Figure 6, 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.

6. Security Considerations

TBD

7. IANA Considerations

TBD

8. Acknowledgments

TBD

9. References

9.1. Normative References


9.2. Informative References

[nc-taxonomy]

[RFC5510]

[RLC]

Authors’ Addresses

Ian Swett
Google
Cambridge, MA
US

Email: ianswett@google.com