ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

Department of Computer Science and Engineering

Bachelor's Degree in Computer Engineering

GRADUATION THESIS

in
Foundamentals of Telecommunications

ECLSA enhancements to support the OpenFEC codec library and to take advantage of it characteristic features

CANDIDATE
Marco Raminella

Supervisor:
Prof. Carlo Caini

Cosupervisor:
Dr. Nicola Alessi

Academic Year: 2015/2016

Session: III

A mio padre, che in tutti questi anni mi è sempre stato vicino

in tutte le difficoltà che abbiamo dovuto affrontare

Summary of this excerpt

Abstract

The DTN (Delay-Tolerant Network) is a networking architecture designed to make networking communication possible also in environments challenged by long delays and disrupted or intermittent connectivity. It is based on the introduction of a new layer, the Bundle layer, between Application and Transport. The Bundle Protocol is the DTN protocol associated to this new layer, running under BP applications; it forms a store-and-forward overlay network over DTN nodes, with the main ability to store for relatively long period of times bundles (packet at bundle layer) to cope with link intermittency, network partitioning and also to enforce custody-based retransmission by intermediate nodes. The Licklider Transmission Protocol runs under BP, it is designed to provide retransmission-based reliability over links characterized by extremely long round-trip times and/or frequent interruptions in connectivity. There are many software implementations of the DTN architecture, of which the most important one is likely ION (Interplanetary Overlay Network), developed by NASA JPL. In ION the LSAs (Link Service Adapters) are an additional software layer used to let LTP communicate with inner protocols. ECLSA has been formally built as a new LSA, but it actually provides the upper protocol LTP with a new service, which we could call of “almost reliability”. In fact, ECLSA it is not just a mere interface towards lower layer protocols, as other LSA, but it protects LTP segments with complex packet-level Forward Error Correction before encapsulating them into datagrams of lower layer protocol such as UDP or other CCSDS specific protocol. ECLSA adopts Low-density parity-check codes and treats each LTP packet as a symbol to be used in the parity-check code. Each symbol is transmitted in one ECLSA packet with its own header to identify the position of the symbol in the codeword at reception, before decoding. ECLSA uses multiple threads allowing it to receive, encode/decode and pass data to other protocols at the same time, it has a modular structure so that it can be easily modified or improved.

The earlier version of ECLSA used a FEC library provided by the German Aerospace Center, DLR, called LibEC; however this library was not released as free software, by contrast to ION, where ECLSA has been implemented. In this framework, the principal aim of this thesis was to find an alternative FEC library written released as free software. The first idea was to use the same LDPC codes used by LibEC, compliant with a recent CCSDS Orange Book, but with an open source codec implementation. However, going along with the research between the codecs available, I found out that an interesting alternative was the use of another family, the Staircase LDPC codes, where the parity check matrix could be built by the decoder itself, on the spot, with a quick pseudo-random algorithm, which have the additional advantage of having been standardized by IETF. The library that uses these codes, and that I finally chose, is OpenFEC, developed by researchers of several French research institutes. I studied the API of this library and its behavior in a small enviroinment where I developed the functions required from ECLSA for managing the encoding and decoding of data using the erasure codes in order to make them work with this library. This has been possible thanks to a new version of ECLSA specifically modified by Nicola Alessi to make it ready to be linked with different codecs by means of specific interfaces, called Codec Adapters.

During the thesis I had to face, as somewhat expected, many problems. OpenFEC has its own way to manage resources, very different from LibEC, so it was necessary to build a complex abstracted interface to make these differences transparent to ECLSA. Then, after having succefully tested the encoding and decoding functions for ECLSA in a small test environment, I soon realized that there were other problems, due to a few OpenFEC bugs. With the help of Valgrind I managed to find the origin of these problems and to fix or circumwent them.

Once solved all problems, I carried out several tests to validate the work done and also to compare performance of different codes. Using the same parity check code sizes, OpenFEC has same or better processing time, very fewer memory expense and almost the same recovery ability of the DLR’s LibEC,

which is very close to ideal codes. I also tested the ability of OpenFEC to generate an LDPC parity check code of any required size, and I introduced the new Continuous Mode in ECLSA to support this ability. In brief, results confirmed the ability of the new Codec Adapter to operate correctly, the substantial equivalence of LibEC and OpenFEC codecs, in terms of performance, and last but foremost, the great performance advantage provided by ECLSA in space communications over lossy links.

The structure of this thesis is the following: we will briefly introduce the DTN architecture, Bundle Protocol and Licklider Transmission Protocol in the fist chapter. In the second chapter, we will briefly recall the Forward Error Correction techniques and the LDPC characteristics. In the third chapter we will describe in full detail the processing steps of the ECLSA protocol. Chapter 4 is focused on the implementation of ECLSA, necessary to introduce in chapter 5 all the work done to build the Codec Adapter specific to OpenFEC; this chapter is useful also to show to the reader how to create new adapters, to support other FEC codecs in ECLSA, a feature that could be of great interest to other researchers. Chapter 6 describes the tests done with the new software implementation.

1 ECLSA DESCRIPTION

This chapter has two aims: first to describe ECLSA from a functional point of view; second, to provide the reader with all ECLSA specifications. The description of the implementation has been deliberately separated and postponed to the next chapter.

1.1 ECLSA AS A LINK SERVICE ADAPTER

Let us start by reminding the role played by LTP LSAs (Link Service Adapters). In the DTN architecture with LTP as convergence layer, after LTP block segmentation, LTP segments must be transferred to the corresponding pair via a lower protocol; this lower protocol can be UDP, as usually happens in test beds, or a CCSDS protocol in operational environments. LSAs are the interfaces towards these lower protocols. ECLSA (Error Correction Link Service Adapter) is a new LSA alternative to UDPLSA (and other LSAs). It protects LTP segments with packet-level Forward Error Correction before encapsulating them into a lower layer protocol, such as UDP or another CCSDS specific protocol. In the next figure we see the network stack of two DTN nodes using ECLSA on top of UDP:

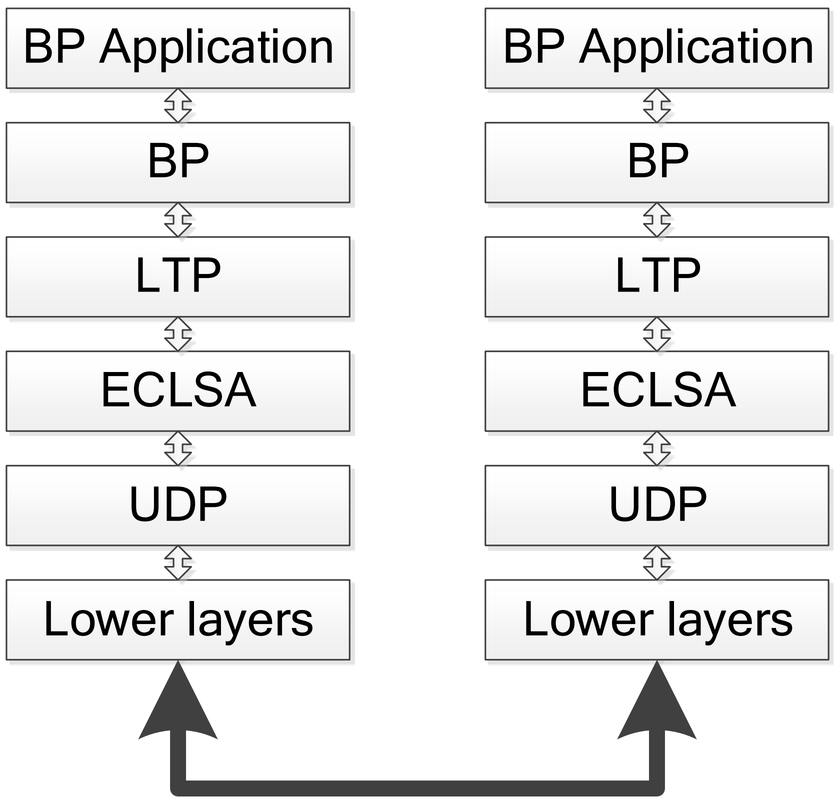


Figure 1: The protocol stack of two DTN nodes sing ECLSA on top of UDP.

Note that the same DTN node can use LTP with different LSAs to communicate with different nodes on different links, as shown in the figure below, where both UDPLSA and ECLSA are present. In the figure, each LSA is divided into Induct and Outduct channels, the former, for managing the data coming in; the latter, the data going out.

Theoretically, ECLSA is transparent to LTP. This means that both green and red parts of LTP blocks are supported in the most natural way. However, in practice LTP RTO timers for red parts must take into account the extra delay due to the use of FEC (see remarks in 1.6.2 and 1.6.3).

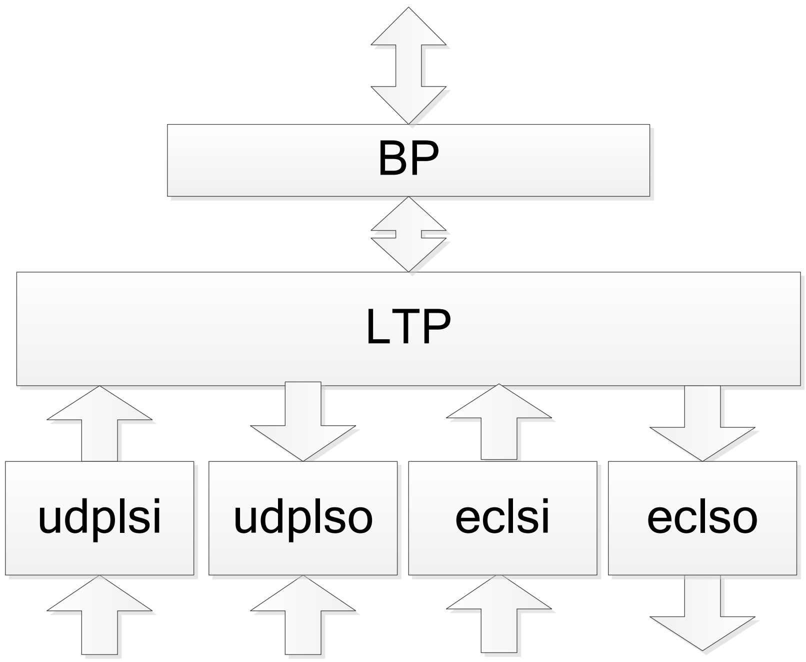


Figure 2: Part of the protocol stack of a DTN node where two alternative LSAs (UDPLSA and ECLSA in the example) are available. For each LSA the induct and outduct components are shown.

1.2 Development

The idea of introducing erasure codes within LTP is due to Tomaso de Cola of the Institute of Communications and Navigation of the German Space Agency (DLR, Deutsches Zentrum für Luft- und Raumfahrt) and was presented in the literature in [34]. As seen in the first part of this thesis, LTP allows the user to transmit bundles in a reliable way as payloads of the red part of an LTP block. However, in the presence of losses, every retransmission cycle of lost LTP segments adds at least one round-trip time (RTT) to the delivery time: the higher the propagation delay, the higher the penalization time. Thus, in space environments, where the propagation delay is very long (1s from Earth to Moon, from 3 to 20 minutes from Earth to Mars) preventing retransmissions is very desirable. This is the aim of ECLSA, which adds FEC to LTP segments to reduce retransmissions as much as possible.

The first implementation of ECLSA dates back to 2013, when Pietrofrancesco Apollonio, a student of the University of Bologna, carried out this task as his Master thesis (with the support of DLR that assigned him a grant), under the supervision of Prof. Carlo Caini and Dr. Tomaso de Cola.

The use of FEC associated with LTP, and in particular of ECLSA, was proposed in the same period in a CCSDS "Orange book" [35], a sort of initial proposal for standardization. Other thesis at the University of Bologna followed, trying to make the first version fully compliant with the Orange book extensions. However, after a while it appeared evident that the first implementation was not a good starting point for future developments, for a variety of reasons. The most important one is the fact that the original code was designed and optimized to run just one FEC code; the first version was eventually extended, by overcoming many difficulties, to allow the user to select one code between the nine proposed by the CCSDS orange book, at start-up, in a static way, while dynamic selection at run time proved impossible. In general, the original version was optimized too early, at the expenses of modularity. For this reasons, taking advantage of the experience gained and of lessons learned the hard way, the development from scratch of a new version was in order. This was implemented by Nicola Alessi, in 2016, as a post-graduate research activity financially supported by the University of Bologna, who totally redesigned and rewrote the ECLSA code. ECLSA 2.0 design was driven by the necessity of a clear logical separation of threads, where a single thread has only one task to achieve; moreover, instead of optimizing the code, modularity was the first concern. This because ECLSA is an experimental protocol, and as that the new implementation should be able to easily support future enhancement or modifications, dictated by future experience. A part from the different design, ECLSA 2.0 adds a number of new features, greatly extending the previous version. A short list is given below:

- Dynamic selection of FEC codes at run time
- Dynamic selection of FEC codes based either on the amount of data to be transmitted, or the estimated segment loss rate, or both.
- Feedbacks provided by the destination
- Possibility to connect multiple ECLSO (LTP sources) with the same ECLSI (LTP destination)
- Independency from upper and lower layer protocols; it could be used below an upper protocol different from LTP, and above a lower protocol different from UDP.

ECLSA 2.0 lacked a comprehensive documentation and one of the aim of this thesis is also to fill this gap. Therefore, ECLSA will be described in detail in this and in the following chapter. This is also instrumental to allow the reader a full comprehension of the technical work done in this thesis. In particular, the possibility of using FEC codecs alternative to that provided by DLR, which offers excellent performance, but it is not open source. Among the choices available, we selected the OpenFEC library [31], mainly developed at INRIA, and we developed an interface adapter for it. At the same time, for the sake of flexibility, we also implemented a generic interface adapter potentially able to work with any other FEC library with just minimal modifications. The OpenFEC library is compliant to the LDPC Staircase RFC [27], and since the new library and standard allow to build an LDPC erasure code with custom parameters, it has been introduced also a *continuous mode,* which tailors the dimension of the FEC code to be used on the exact amount of data to be coded, instead of selecting the best of the available sizes, as usually done (this is why is called continuous). In brief, this new version can build the most suitable erasure code on the spot, based on the needs of the particular transmission session in progress, with clear advantages.

1.3 General description

ECLSA has the function to introduce FEC (Forward Error Correction) in order to mask packet losses of the lower protocol to the upper protocol. From now on, and without loss of generality, we will consider LTP as upper protocol and UDP as lower protocol, unless otherwise specified. As said, in the DTN scenario the ECLSA aim is to minimize the retransmission cycles of LTP segments, i.e. to minimize the losses *as seen by LTP*.

Here we will limit ourselves to a preliminary description, very simplified, to provide the reader with a general framework of the process performed by ECLSA (Figure 3).

The first thing to stress is that the usual FEC concepts are applied here to packets instead of bits [35]. We will start this preliminary description from the sender side, i.e. from ECLSO. We can distinguish tree logical phases, assuming we have a systematic (N,K) code (for the sake of clarity, the description below is slightly simplified):

1. **Matrix filling (from LTP)**: K LTP segments are passed to ECLSA; each segment is an information symbol (Info packets in the figure); they are written as rows of an N-row matrix (the N-symbol codeword, or coding matrix)

2. **Matrix encoding**: M=N-K redundancy symbols (parity packets in the figure) are added in the last M row of the matrix. The code rate Rc=K/N represents the amount of information per codeword symbol. The lower the code rate, the higher the amount of redundancy introduced.
3. **Matrix passing (to UDP)**: the N rows are passed one-by-one to UDP; each row will be encapsulated into one UDP datagram.

UDP datagrams are in turn passed to lower layer protocols and transmitted to the receiver node. As usual, the process is transparent and thus protocols at upper layers, such as UDP, should not care of what happens at lower layers. In practice, the only impairment that is seen by UDP and in turns by ECLSA on the receiver side are losses, either caused by congestion, or noise (flipped bits result in a failure of CRC checks at lower layers, and as a consequence a full packet is discarded). Lost packets, i.e. lost symbols are denoted by crosses on the right side of the figure; of course, both information and redundancy packets can be lost, as shown in the right side of the figure, where packets lost are crossed.

We are now ready to describe what happens on the receiver side, i.e. what are the steps performed by ECLSI. We can distinguish the following three phases, which are the dual of those just described, but in the reverse order (from UDP to LTP):

4. **Matrix filling (from UDP)**: Let L be the number of packet lost; N-L UDP datagrams arrive; their payload is read and written in an N row matrix (the N codeword at receiver side) leaving gaps (i.e. rows filled by zeros) in correspondence of missing symbols.
5. **Matrix decoding**: The aim of the decoder is to extract from the N-L received symbols the K information symbols (i.e. basically our K LTP segments). If the decoding is successful, all the first K rows are eventually filled. Put in other words, as the code is systematic, the missing info symbols (two in the figure) are recovered by exploiting the redundancy symbols. Thus, to have a success it is necessary to receive at least K of the N symbols transmitted, whatever they are (information or redundancy); this is equivalent to say that L≤M is necessary. This is however usually not sufficient, and in practice a suitable margin is requested.
6. **Matrix passing (to LTP)**: the first K rows of the matrix, i.e. the info symbols, are read and passed one-by-one to LTP; each row containing one LTP segment.

Before going on, let us stress that the lower the code rate, the higher the redundancy introduced per information symbol, thus the higher the chances of decoding success. On the other hand, the higher the redundancy, the higher the bandwidth wasted in the absence of losses or if the channel is significantly better than expected (L<<M).

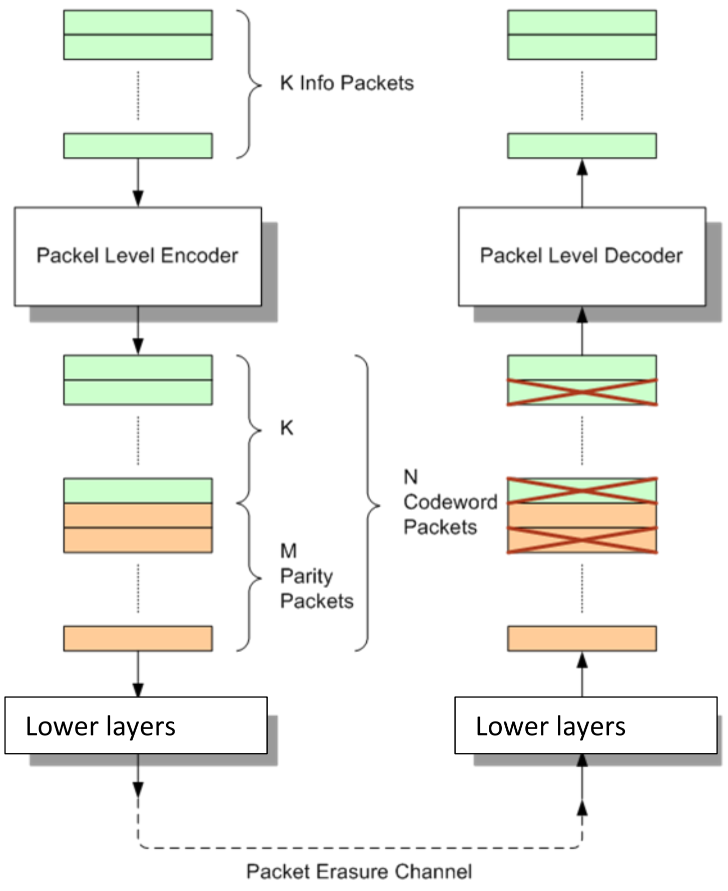


Figure 3: ECLSA logical process (simplified) (by T.De Cola).

1.4 ECLSO

1.4.1 Matrix filling (from LTP or other upper layer protocols)

Let us start by describing in full detail the organization of the “coding matrix”, representing the N-symbol codeword, which is at the core of ECLSA.

As said LTP segments are passed by LTP to ECLSO; each segment is considered as one symbol of the N-symbol codeword. As FECs in packet coding, as here, work on packets, and not on bits, symbols consist of a given number of bytes, i.e. have a size. If we imagine a symbol as a row vector of length T (in bytes), the N codeword becomes a matrix of N rows and T columns, as shown in Figure 4.

A bit more complexity in the description is however necessary, because an LTP segment could be shorter than others, as it usually happens for the last segment of an LTP block. To deal with this length diversity, the first two bytes of a symbol must be reserved (they are highlighted in yellow in the figure) for indicating the actual size of the LTP segment contained in the symbol (in blue). If the LTP segment is shorter that T-2 we have to fill the tail bytes with zeros; this is called row *padding* (in white) Another kind of padding is necessary if the number of LTP segments received by LTP, I, is lower than K. In this case the K-I empty rows are filled with info padding (row in white). After encoding, the remaining M rows are filled with redundancy symbols (in green).

As it is not possible to split an LTP segment into two lines, the maximum length of an LTP segment is T-2.

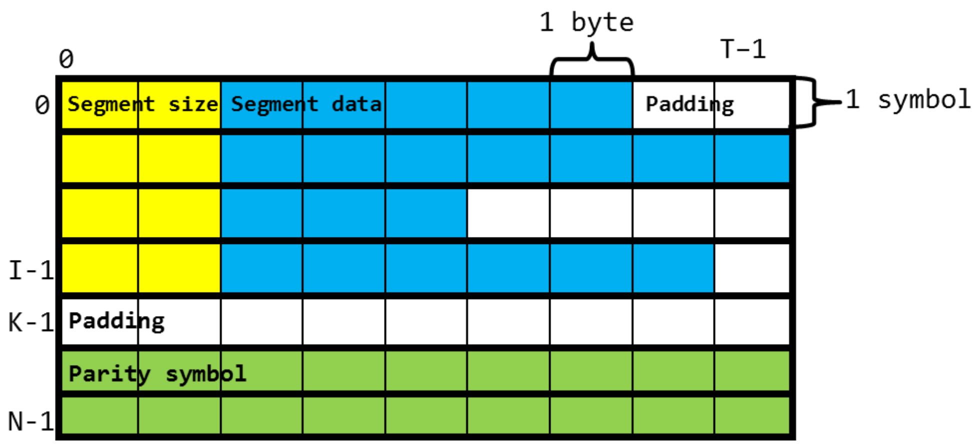


Figure 4: An example of coding matrix. Rows and columns counting start from 0. T is the size of the symbols, K the number of Information symbols, N the dimension of the codeword, i.e. the total number of symbols. In yellow we have the two bytes representing the LTP segment size, in blue the LTP segment data, in green the redundancy (or parity) symbols. White spaces are padding. I specifies the actual number of LTP segments added.

The flow chart below shows the matrix filling algorithm. When the first LTP segment is received the aggregation timer is started; then the segment is added in the first row; the process goes on until either K segments have arrived, i.e. the info part of matrix is full, or the aggregation timer expires. In both cases the matrix is passed to the encoder (Matrix encoding phase) and then to UDP (Matrix passing phase).

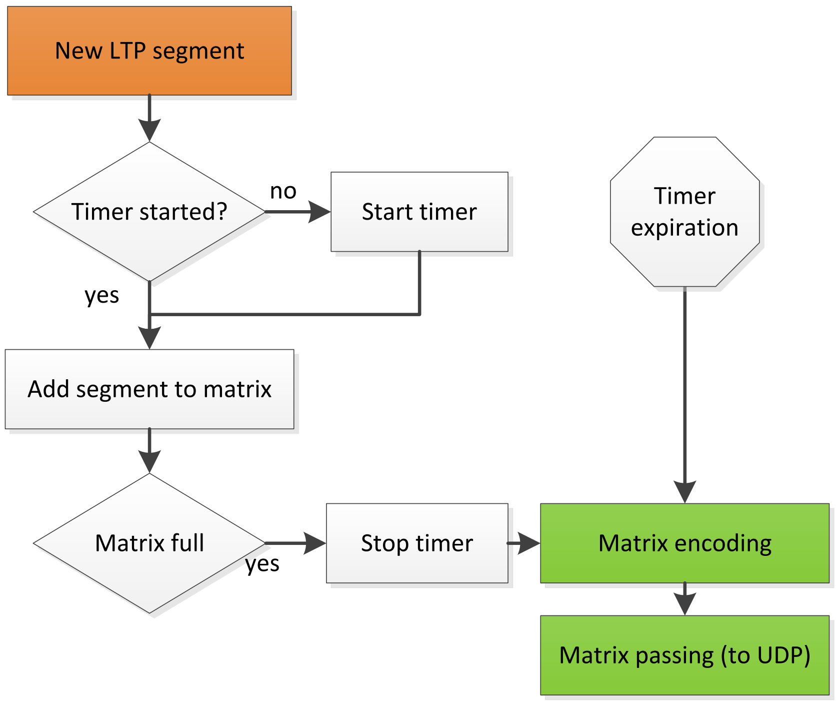


Figure 5: ECLSO process: matrix writing (expanded) followed by matrix encoding and passing.

1.4.2 Matrix Encoding

Before the actual encoding of the matrix, it is necessary to select the code. This is trivial if the code is static, much more complex if dynamic. Before going on, it is necessary to briefly examine the pros and cons of available codes. To this end, it is useful to introduce the 9 codes described in the CCSDS Orange book (see [35]). As only 3 values are considered for both K and Rc, they can be conveniently arranged in a 3x3 matrix, as shown in *Table 1***Errore. L'origine riferimento non è stata trovata.**. Note that there is a huge step between consecutive K values, while the difference is lower between Rc values (and consequently between N values in the same row).

	Rc_1=8/9	**Rc_2=4/5**	**Rc_3=2/3**	
K_1=512	N_{11}=576	N_{12}=640	N_{13}=768	
K_2=2048	N_{21}=2304	N_{22}=2560	N_{23}=3072	
K_3=16384	N_{31}=18432	N_{32}=20480	N_{33}=24576	

Table 1: FEC codes defined in the CCSDS Orange book

Let us start by considering Rc fixed (e.g. =8/9) and examine the pros and cons of different K values. The pro is that the higher the K, the closer to the ideal the performance of the code; the cons however are many and significant: the use of large codes implies the allocation of vast amount of memory; the coding delay is long; there can be a huge difference between I and K whenever the timer of Matrix filling expires. We will focus here on this last problem, as it can result in an undesirable excess of redundancy sent. In fact, although the K-I info padding rows are not sent (they are empty), all the M (=N-K) redundancy symbols are, thus, If I<<K the result is that the actual code rate (Rc_actual=I/N) can be much lower than the nominal code rate (Rc=K/N). For example, if in the ECLSO settings embedded in the span instruction is selected the code (24576, 18432), i.e. Nspan=24576 and Kspan=18432, and the timer expires when only the first 500 rows are filled, we will send on the channel redundancy in large excess (about 32 times larger than requested by the nominal rate), causing a huge waste of bandwidth. To keep the best of both large and short codes, it is necessary to introduce a dynamic selection of the code. In the ECLSO settings we can choose Kspan on the basis of the two other drawbacks, memory and delay, letting the system to reduce the actual K chosen whenever possible, if the matrix is not fully filled (i.e. I<Kspan). In the example considered, K=512 would be the best choice, as 500<512. In brief, the selection of K looks like the selection of the most suitable box for a content of variable dimension.

Moving to Rc, we start by noting that the smaller the Rc the higher the amount of redundancy introduced per information symbol, and thus the more powerful the erasure recovering performance of the code. For example, assuming an ideal code, with Rc=8/9 we can recover on the average one loss over 9 packets, with 4/5 one over five, with 2/3 one over 3. The choice of the wanted Rc should therefore dictated by an estimation of the packet loss probability of the channel. If the channel is stationary and known a priori, it is easy to make the right choice. Otherwise, if the channel is stationary but not known a priori, or variable in time, a possible choice is to select a very conservative Rc. This however, results in an excess of protection, i.e. inefficient use of the bandwidth, whenever the channel is better than expected. To cope with this second cause of potential bandwidth waste, in ECLSA it possible to dynamically select the code on the basis of a target code rate, variable in time.

Before examining the details of the algorithm used in ECLSA (see Figure 6)., we need to point out that in ECLSA the dynamic selection of the code can refer either to K (either “K continuous” or “Adaptive selection of K” options), or to the code rate (“Feedback Adaptive Rc”) or to both.

By default (no adaptive options set) the selection of the code is static and thus K=Kspan and N=Nspan. Vice versa, if at least one of the 3 adaptive options mentioned above is set, the selection is dynamic. In this case, if the matrix is full (I=Kspan) there is no need to look for a lower K, thus K=Kspan and all the blocks of dynamic K selection are skipped. By contrast if a timeout occur (I<K) and the “Adaptive selection of K” is enabled, K is chosen as the minimum among all Ki available that contains the filled rows, i.e. with I≤Ki. If the alternative “Continuous Mode” is enabled (only available with the OpenFec library), K is taken equal to I, as the code can be tailored to every K lower than the maximum (this is why this option is called K continuous).

For the sake of simplicity, we temporarily consider off the “Adpative feedback Rc option”, so that we can skip the green blocks. As said the dynamic selection of K is useful to stay as close as possible to the target Rc, which is static (=Kspan/Nspan) when the feedback option is off. To this end, the most suitable N must be chosen as last step. The best choice is the minimum Ni (among the codes with Ki=K) that results in an actual code rate (Rc_actual=I/K) lower than the target code rate Rc_trgt. Note that if the K continuous option is enabled, there will be one N which will lead to an actual Rc almost coincident with the target (as also N are continuous), thus eliminating by root any waste of bandwidth.

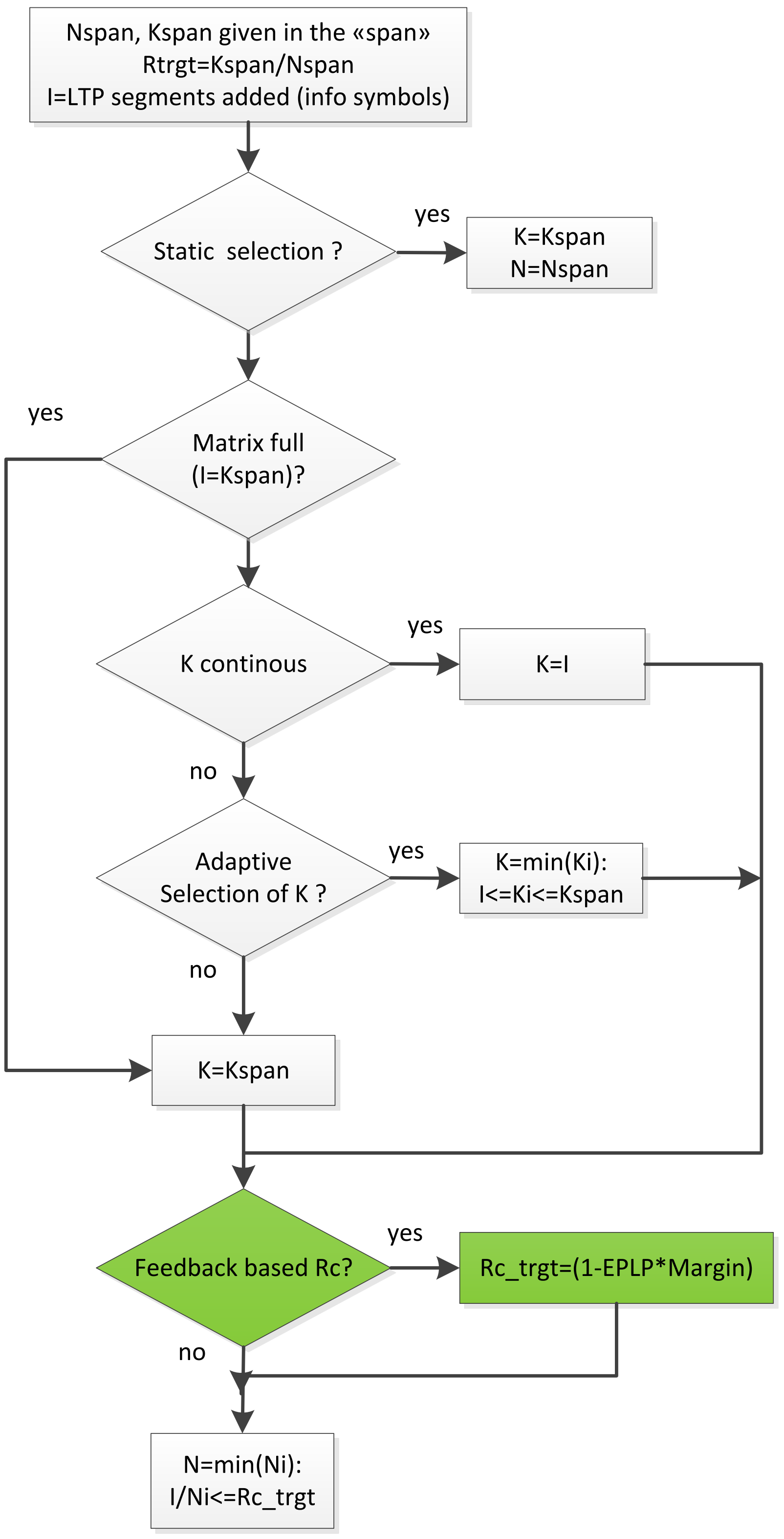


Figure 6: ECLSO Matrix encoding (second phase): FEC selection (the blocks added to consider the Feedback based RC dynamic selection are highlighted in green).

Let us now introduce the Rc dynamic selection, which aims at matching redundancy to the actual loss probability and thus requires a feedback from the receiver. It is active only when the “Feedback Adaptive Rc” option is set, which in turn requires the “Feedback Request” option enabled too. Note that it can be enabled or not independently of the options related to the dynamic choice of K. When enabled, the only difference in the algorithm is that Rc_trgt, which was static, now must be substituted by a dynamic value calculated on the basis of the information provided by feedback packets. This is given by the complementary of the Estimated Packet Loss Probability, as shown in the flow chart. The latter is multiplied by a safety margin to take into account the fact that: a) the codes are not ideal, b) the channel may be variable, c) the estimator itself is not perfect, d) k is limited, thus the number of missing rows will not be equal to PLP*K.

Of course, the Rc dynamic selection validity in operational environments essentially depends on the correlation time of the channel, being totally ineffective, and possibly harmful, if the RTT> of the correlation time. With channels almost stationary, it has the practical but significant advantage of adapting fairly well to the characteristics of the channel, thus simplifying the ECLSA settings.

The feedback packets, sent by ECLSI on the receiver node after each matrix decoding, provide ECLSO on the sender node with two kinds of information:

- the number of symbols received and of symbols expected (I+M); they allow ECLSO to update the EPLP, so that the best matching code can be chosen
- the success or failure of a decoding process; in case of a failure ECLSO immediately replaces the old EPLP value with the actual rate reported in the failure message, thus preventing further failures if the loss rate does not further impair significantly. This condition proved very effective in tests.

The details of the algorithm used for the EPLP estimation as well as the criteria for setting the margin are not reported here as they are subject to frequent modifications, being totally experimental.

1.4.3 Matrix passing (to UDP or other lower protocols)

After encoding the matrix, ECLSO performs this third phase. The N symbols of the codeword (i.e. the rows of the matrix) are read from the matrix and encapsulated one-by-one into UDP datagrams, skipping info padding rows to save bandwidth (for the same reason, if the symbol is shorter than the row, the row padding is dropped too). Before encapsulating one symbol into one UDP datagram an ECLSA header is added. The information that is contained in this header is necessary to ECLSI on the receiver to correctly fill the coding matrix at reception with the received symbols and to use in the decoding phase the same FEC code used for encoding.

The present format of the ECLSA header is temporarily the following (the protocol is in an experimental phase, thus modifications are likely):

Version (1B)	ExtCount (1B)	Flags (1B)	EngineID (2B)	MatrixID (2B)	SegmentID (2B)	I (SegAdded) (2B)	K (2B)	N (2B)	T (2B)

Figure 7: ECLSA header

The header contains this information:

- Version: this value is at present set at 0.
- ExtCount: Extension Count; reserved for future purposes.
- Flags: if b0=1 the decoding feedback is requested; if b1=1 the k continuous mode was enabled. Others bits are reserved for future purposes.
- EngineID: this value contains the node identifier, which is given by the upper protocol. For LTP, it is the corresponding Engine ID.

- MatrixID: it is a unique identifier of the coding matrix (i.e. of the codeword) computed by a certain EngineID: the pair [EngineID,MatrixID] is the unique identifier for ECLSI. The pair is requested by ECLSI to know to which coding matrix belong an incoming packet. Once the ECLSI elaboration of the coding matrix is completed, the corresponding pair [EngineID,MatrixID] is put into a *blacklist* so all future arriving packets in this list will be ignored (e.g. out of order packets that arrive after their matrix has already been decoded).
- SymbolID: it is the the symbol index, i.e. the index of the row; note that it is called ESI in OpenFEC codec library. Note that this is the only field that changes for ECLSA packets referring to the same matrix.
- I (SegAdded): the actual number of LTP segments added to the coding matrix. (see Figure 4). It must be specified to let ECLSI know the amount of matrix padding (unfilled rows).
- K: The FEC parameter K of the code actually used for encoding;
- N: The FEC parameter N of the code actually used for encoding; K and N fields are necessary to let ECLSI know the code used by ECLSO (for the specific codeword, if dynamic).
- T: The symbol size is necessary to ECLSI to know the maximum row length.

The figure below summarizes the different level of encapsulation performed by ECLSO (and vice versa by ECLSI). The first encapsulation is done when an LTP segment is received by LTP, the second when a symbol of the codeword is encapsulated in an ECLSA packet, the last one when the ECLSA packet is encapsulated in a UDP datagram. Note that the figure holds true only for the K information symbols; the M redundancy symbols are entirely generated by the FEC coding process, thus the first two layers in the figure are missing.

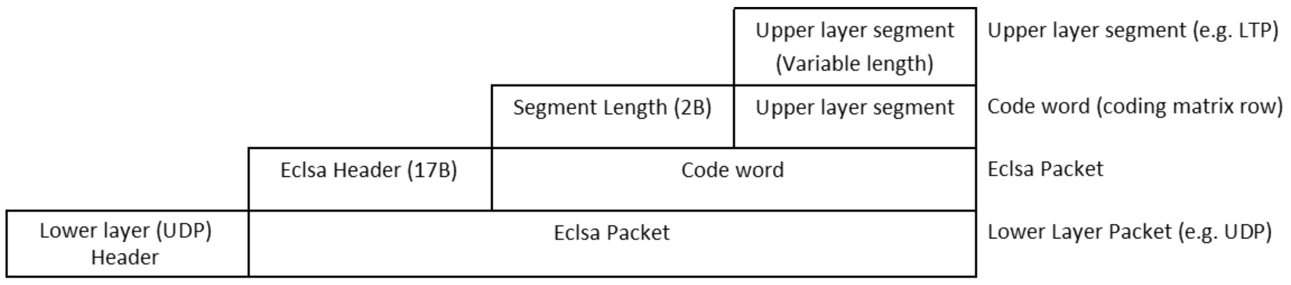

Upper layer segment (Variable length)
Upper layer segment (e.g. LTP)
Segment Length (2B)
Upper layer segment
Code word (coding matrix row)
Eclsa Header (17B)
Code word
Eclsa Packet
Lower layer (UDP) Header
Eclsa Packet
Lower Layer Packet (e.g. UDP)

Figure 8: LTP segment encapsulation in a UDP datagram via ECLSA.

1.5 ECLSI

1.5.1 Matrix filling (from UDP or other lower layer protocols)

The first phase of ECLSI is the dual of the last one of ECLSO. ECLSA packets are decapsulated from incoming UDP datagrams, then the ECLSA header of each packet is read. From the pair [EngineID,MatrixID], which uniquely identifies the coding matrix (i.e. the codeword) it belongs to, and from the SymbolID field, which specifies the row, it is always possible to find the correct location of the incoming packet. This also in the presence of out of order packets or parallel flows from different nodes (i.e. different engines). The field I tells ECLSI how many information symbols were actually present in the coding matrix (being K-I info padding rows), while N, K and T are the parameters of the FEC code (inside a family) used for the received codewords. These four parameters are associated to one coding matrix, thus are the same for all the corresponding ECLSA packets. They are deliberately replicated in all packets because in the presence of losses it is not possible to know which symbols arrive and which are lost, thus it would be unsafe to send this information associated to a specific symbol only.

At reception not all symbols sent arrive, therefore it is paramount to define when the writing process of a matrix can be considered completed. This happens when one of the following conditions occurs:

1. All info symbols have been correctly received (this is why I is necessary). In this case there is no need of waiting for (other) redundancy segments and also there is no need of decoding. This condition is usually verified when the channel does not introduce any loss, or the loss probability is very low.
2. The last redundancy symbol has arrived. This is what usually happens in the presence of moderate losses.
3. A symbol generated by the same engine and belonging to a coding matrix with sequential ID higher than the current one has arrived. This condition is met with a probability equal to the loss probability, as it requires the loss of a specific symbol, the last redundancy symbol (by neglecting the probability of its out of order arrival).
4. A timeout occurs. At reception of each new symbol of a specific codeword, a closing timer is set (first symbol) or reset (for other symbols); if no other symbols arrive the timer expires and the coding matrix is considered complete. This condition aims at preventing deadlocks, should the last redundancy symbol of the last matrix sent by the same engine be lost.

The following chart summarize the algorithm:

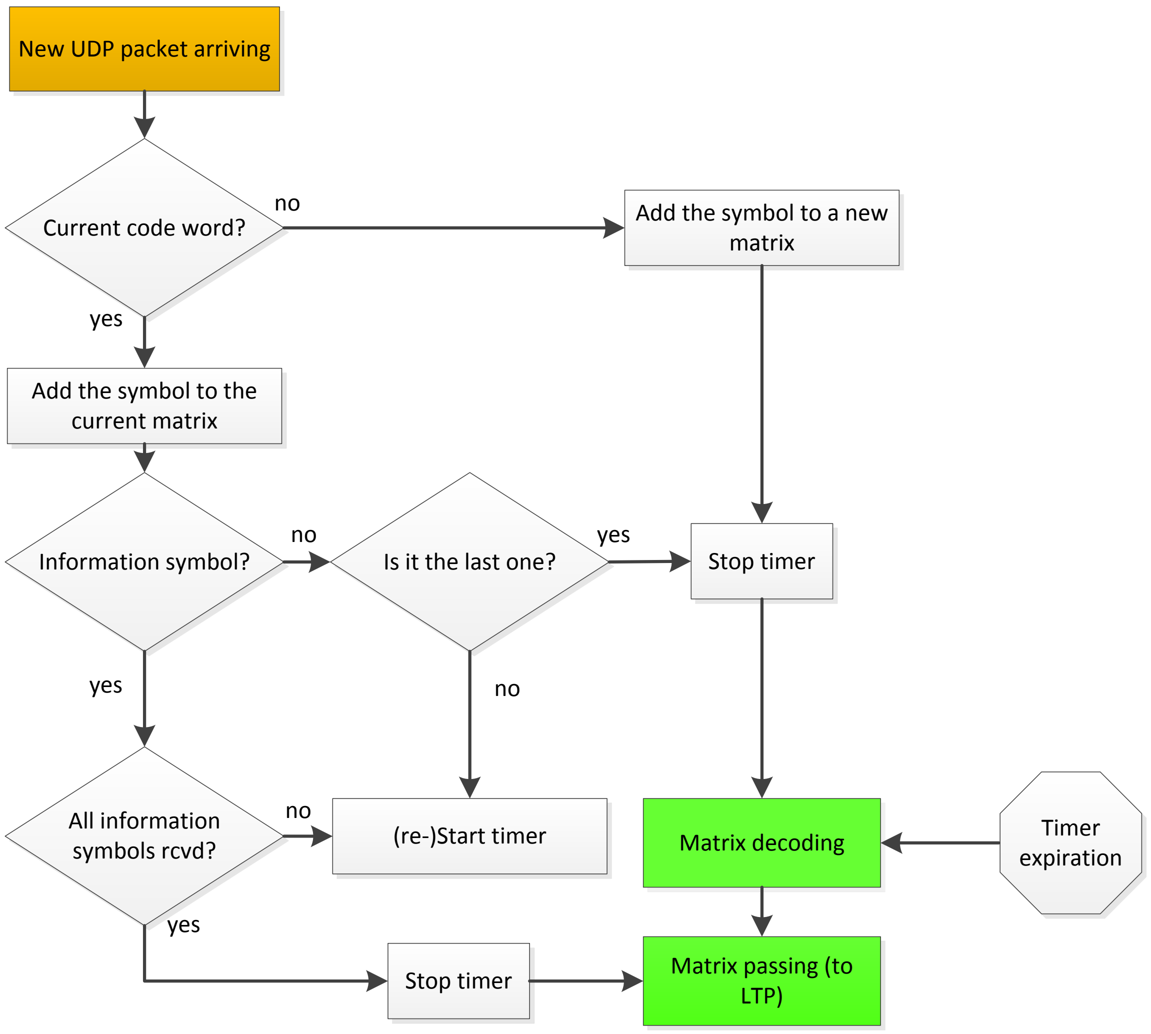


Figure 9: ECLSI algorithm: matrix filling (expanded) followed by matrix decoding and passing.

1.5.2 Matrix decoding

The decoding is skipped if either all information symbols are received or the matrix was not encoded (e.g. because the number of segment added before a timeout, I, was less than the Encoding Threshold parameter). In all other cases the matrix must be decoded. To this end ECLSI must know the code used, which is specified by the parameters N, K, and T as said. More precisely, this tuple identifies a code inside one FEC family, as two different families (e.g. those in LibecDLR and OpenFEC) have different codes with the same 3-tuples. To have the same erasure code available at both sides, its parity check matrix must either be saved to file or be generated by the same algorithm. The former solution is adopted when the LibecDLR codec library is used (9 codes specified in the CCSDS Orange book), the latter when OpenFec is preferred, as it is able to generate the wanted parity check matrix on the spot.

1.5.3 Matrix passing (to LTP or other upper layer protocol)

After decoding, the information symbols are orderly passed to LTP after removing the 2B header. If the "Request Feedback" option flag was set, a feedback is sent to the ECLSO engine on the source (see the figure below). It reports: the decoding status (success or failure, plus other information), the number of symbols expected (I+M, in the TotalSegments field), and that of symbols actually arrived (I+M-L, Received Segments). The feedback packet is sent directly by ECLSI to ECLSO through UDP, i.e. it is not protected by any FEC. For this reason, it can be replicated (see Feedbackburst parameter in ECLSI configuration).

MatrixID	CodecStatus	TotalSegments	ReceivedSegments
(2B)	(1B)	(2B)	(2B)

Figure 10: The feedback packet sent by ECLSI on the receiving node to ECLSO on the sending one, via UDP (or other lower layer protocol).

If the decoding is successful all information symbols are recovered and all the LTP segments that were included in the same coding matrix are passed to LTP; otherwise, there will be gaps. Should these missing segments belong to the red part of an LTP block, LTP will require retransmission, in a way completely transparent to ECLSA (i.e. ECLSA is nested in LTP and all LTP RED mechanisms are external to ECLSA).

1.6 ECLSA OPTIONS AND CONFIGURATION

1.6.1 ECLSA options

It is convenient to list all the options at present available in ECLSA before examining the way ECLSA is called in the ION .rc configuration file.

- Adaptive Encoding: This feature dynamically selects the K of the erasure code on the basis of the segment added, I, and the corresponding N to maintain the required code rate.
- Continuous mode: it enables the possibility to tailor K with I (available only with OpenFEC codec library). It requires Adaptive Encoding on.
- Feedback Requested: This feature requests that the receiver send decoding feedbacks.
- Feedback Adaptive Rc: It selects the code rate (Rc=K/N) on the basis of the estimated loss rate on the channel and decoding failures advertised by feedback packets. It requires Feedback requested on.
- Interleaving: It scrambles the sending order of the coding matrix rows, but the last. The last is never changed as its reception is one of the three conditions ECLSI uses to declare completed the reception of a coding matrix. Interleaving is used to spread losses of a bursty channel inside the same coding matrix.
- Static MID: This feature, when set, forces the first Matrix ID to be initialized to zero. The default behavior is to have the sequential MID counter starting from a random number to minimize the probability of a conflict with the ECLSI black list (see the Implementation chapter) when an ECLSO engine is restarted without restarting ECLSI on the corresponding pair. By contrast, starting MID from zero may facilitate reading logs.

1.6.2 ECLSA in ION .rc configuration file

In ION the configuration of an LSA is associated to two instructions, for the outduct and induct respectively, In the next figure we see an example, referring to a .rc configuration file:

```
#Flag configuration
#MASK_ADAPTIVE_MODE         bit 0 (adaptive selection of K)
#MASK_CONTINUOUS_MODE       bit 1 (only with OpenFEC codec library)
#MASK_FEEDBACK_REQUEST      bit 2 (feedbacks)
#MASK_FEEDBACK_ADAPTIVE_RC  bit 3 (adaptive target code rate based on feedbacks)
#MASK_INTERLEAVING          bit 4 (interleaving of symbols inside a codeword)
#MASK_STATIC_MID                 bit 5 (if set ECLSA on the receiver MUST be restarted
whenever restarted on the sender)
#Example1: Static code, no feedbaks, no interleaving, random MID -> 000000 -> 0
#Example2: Static code with feedback requested -> 0000100 -> 4
#Example3: Adaptive K, feedback request, feedback adaptive Rc -> 001101 -> 13
#Example4: Continous K, feedback request, feedback adaptive Rc -> 001101 -> 14

# Pay attention to the UDP_Tx rate that MUST include the redundancy (2000000=2Mbit/s);
# Max Tx rate is also limited by usleep value in eclsi.c (interval between consecutive
segments readings)
# At present 0.5 ms, i.e. no more than 2000 info segment per second; safe for 10Mbit/s

# ECLSA 2.0                                      N     K     AggTime  codThr  FLAGS  Tx/rate
(bit/s)
a span 2 15 15 1024 1 1 'eclso 10.0.1.2:1113  576 512   500       1    0    10000000'
#a span 2 8 8 1024 1 1 'udplso 10.0.1.2:1113 2000000'


# The max waiting time is the maximum tolerable interval between reception of consecutive
symbols of the same codeword
# If exceeded, the codec matrix is considered completed.
# FeedbackBurst = how many copies of the feedback packet must sent back by ECLSI

# ECLSA 2.0            maxWaitingTime FeedbackBurst        MaxT            MaxK (opt)
#                         (ms)        (fedback copies)  (T=LTPsegmentLenght+2)
s 'eclsi 10.0.1.1:1113     100           1                1026          2048      '
#s 'udplsi 10.0.1.1:1113'
```

Figure 11: ECLSA configuration lines inside the ION .rc configuration file.

LSA ouducts are configured inside "a span" instruction (see Figure 11). By comparing the syntax of eclso and udplso (commented line), it can be seen that the first and the last parameters are the same, (the IP address and port number, and the Tx rate in bit/s), but ECLSA has a few new parameters. They are:

- N and K of the default code (i.e. the code that is used if adaptive options are off), for the specific span;
- maxAggregationTime (ms): the maximum waiting time before a coding matrix is sent although not completely filled (I<K);
- codingThreshold: the minimum number of Information symbols (I) required for coding; 1 means that the coding is always performed. Higher values are useful to disable coding in specific circumstances (e.g. very limited traffic and very low loss probability).
- flags: this is a one-byte-integer used to set ECLSA options. The flags value is found this way: first the binary value must be found, with all options from the most to the less significant bit, where 1 means enabled and 0 disabled, then this value must be converted from binary to decimal notation. Let us consider as an example the option tuple shown in the figure below.

Reserved	Reserved	Static MID	Interleaving	Feedback Adaptive Rc	Feedback Request	K continous mode	Adaptive K mode
		0	**0**	**1**	**1**	**1**	**0**

Figure 12: Example of FLAGS settings

As $1110_2 = 14_{10}$. the flag value to be passed in the example is 14.

The options for ECLSI are:

- MaxT, the maximum size of T, the symbol size, which may be the Maximum LTP Segment Size + 2;
- closing timer (ms), the maximum time from the last received packet before decoding begins; used as a last resort when the other conditions to start the decoding are not met.
- FeedbackCopies (integer) the number of times that the feedback must be re-sent by ECLSI to cope with losses.

1.6.3 LTP and ECLSA advanced settings: extra delay for LTP red timers

LTP blocks and coding matrixes (codewords) are totally decoupled in the ECLSA design. This is to say that one coding matrix can contain multiple LTP blocks and, conversely one LTP block can be spread on two (or more) coding matrixes. The FEC mechanism is totally transparent to LTP, except the LTP red retransmissions timers that must include the extra delay due to the FEC processing on both directions, which is however much lower than RTT in space environments.

The extra delay in the forward direction (data) is given by:

- The ECLSO aggregation time. This because it coincides with the maximum interval time between the reception of the first LTP segment and the forced closing of the filling phase. It is passed as an ECLSO parameter. It can be calculated as (T-2)*Kspan/contact_Tx speed (which is in B/s) plus a margin. The higher Kspan, the higher the delay, thus it is suggested not to set a very large Kspan in the ECLSO configuration if the Tx speeds are relatively low, i.e., to generally avoid the use of the current K3 which is really huge.
- The time necessary to encode the matrix. This depends on the code used. Even in this case, the larger the matrix, the longer the delay. However, in our tests it was always marginal with respect of the other components.
- The extra transmission time necessary to UDP to send the packets of a coding matrix instead of an LTP block. As LTP block dimensions vary, depending on Bundle dimension and aggregation this value can be conservatively evaluated as Nmax*8/UDP_Tx_rate, i.e. as the transmission time of a matrix. Nmax=Nspan if the feedback based Rc is off, otherwise Nmax=Kspan*1/Rc3=Kspan*3/2.

The extra delay in the reverse direction (used by Report Segments) is given by:

- The ECLSO aggregation time on the receiver, because LTP Report segments move in the reverse direction
- The time necessary to encode the matrix on the receiver, for the same reason.
- The extra transmission time to send the coding matrix. Note that I<<Kspan when only LTP signaling is present on the reverse direction; in this case the total amount of row to send is only marginally greater than the number of redundancy symbols (M).

Note that if the channel is asymmetric, and there are no losses on the reverse path, it is possible to skip encoding in the reverse direction, which minimize the delay (the last two contributions are null).

If only green blocks are sent, the only delay is in the forward direction. However, as there are no retransmission timers, it is absolutely irrelevant concerning LTP configuration.

1.6.4 LTP and ECLSA advanced settings: LTP, ECLSA and UDP parameter constraints

There are also a few constraints internal to the settings given in the span instruction:

- The UDP payload poses a limit on T. One ECLSA packet must be contained in a UDP (or other lower layer) packet. Thus, if we want to avoid IP fragmentation and MTU=1500, we have about 1460 (20B for the IP header plus 16B for IP extensions, 4B for UDP header) available for the ECLSA packet. As its header is currently of 17B, we have 1443 Bytes available for a row, i.e. Tmax=1443.
- An LTP segment must be contained in a matrix row. As we have a 2 byte header, the maximum LTP segment length is T-2. This condition is easy to meet as the size of LTP segments is a parameter of the span instruction. Under the previous conditions LTP segment size<=1441. In our test LTP size=1024 and consequently T=1026.
- The maximum Tx speed of UDP (if set) must be enough to include also redundancy. The minimum value (in b/s) can be calculated as ContactTxSpeed(in B/s)*8/Rc, where Rc=Kspan/Nspan is the feedback adaptive Rc is off, Rc=Rc3 otherwise.

2 ECLSA IMPLEMENTATION

In this chapter we are going to briefly describe the ECLSA implementation. Its design was driven by the "one task -one thread" principle, having modularity as first concern. The rationale of this choice is that until ECLSA is experimental, we would like to be able to introduce modifications and enhancements without too much effort.

ECLSA is written in C and has been developed to be inserted in ION [25]. However, it does not use the memory management (SDR) used by ION components, but standard C "malloc" and "frees". This design choice is to make this experimental implementation usable in ION tests, while at the same time allowing its possible use on other DTN protocol implementations, with minimal modifications. Additional reasons were code readability for people (e.g. future students) not familiar with ION SDR.

First we will examine a couple of topics that are in common to ECLSO and ECLSI, and then their threads. Note that ECLSA thread synchronization logic is based on Dijkstra's semaphores; the logic and implementation details of semaphores are out of the scope of the present document.

2.1.1 Initialization

When ECLSO and ECLSI are started a few initialization functions are called. Among them, the most important is `fecManagerInit` which makes the FEC codec library in use to identify the erasure codes available.

The codec library is chosen at compilation time; at present the choice is between either LibecDLR or OpenFEC, but other codecs can be added in future (the third option, DummyDEC, is just a template to this end). LibecDLR makes use of the 9 LDPC codes described in the Orange book [35], whose parity check matrixes and other parameters are written in 9 files, to be read at start-up and put in suitable structures. To have them in RAM is a design choice made to avoid time-consuming I/O operations at execution time. By contrast, OpenFEC use triangular LDPC codes, described in [27]. An interesting feature of these codes is that they can be built on the spot, thus they do not need any files. For the sake of commonality, for each code present in the Orange book we have considered its triangular equivalent, i.e. the code with the same pair (N,K). Thus, for both the codec considered we have the same 9 pairs (N,K) potentially available. They are reported in [35]. By the way, our tests have shown that corresponding LibecDLR and OpenFEC codes provide roughly the same performance.

2.1.2 The matrix buffer

Both ECLSO and ECLSI have a "matrix buffer", consisting of a pre-defined number of ECLSA_matrix elements. The buffer length is set with a define instruction and could be longer in ECLSI than in ECLSO to cope with multiple ECLSA packets flows, coming from different nodes. The current default length is 5 for both. The matrix buffers are allocated at start up, which allows saving memory if large matrixes (i.e. codes with long N and K) are not required by the ECLSO and ECLSI configuration settings. Each ECLSA_matrix consists of

- a "container" for the code-word symbols, called codec_matrix. More specifically this container consists of the codewordBox matrix (Nmax x Tmax), where the codeword symbols are stored, and the auxiliary vector symbolStatus (Nmax). Nmax is the max N among the FEC codes to be considered available in a specific configuration, and can be calculated as Nmax=Kspan/Rc (if the feedback RC adaptive is off) or as Nmax=Kspan/Rc3 (feedback RC adaptive on). In brief, the total amount of RAM will be directly proportional to Nmax and thus to Kspan. This is a further reason not to exceed with large values, especially if there are RAM constraints. Let us present some figures. Assuming the feedback adaptive Rc on, on the codes reported in the CCSDS Orange Book, one codewordBox would require about 25 MB with Kspan=K3, 3 MB with K2 and 700kB with K1 and in the buffer by default we need five of them for ECLSI and five for ECLSO.

- a number of ancillary fields related to the whole codeword inserted in the container, and necessary to its management, such as the following
    - Engine ID (EID)
    - Matrix Index (MID, unique identifier)
    - General Index (GID, used by ECLSI as a unique identifier)
    - (N, K) i.e. code used, and T, i.e. dimension of the symbols in byte
    - I (number of information symbols added)
    - SEQ (Sequence used by the interleaver)
    - Markers “available”, “ready to encode”, “ready to pass”, etc...

This buffer is instrumental to allow the threads to work in parallel. We will see how this buffer is used in detail in next sections. Before going on, let us point out that the term matrix is per se ambiguous at it can refer to both the content (the codeword) and the container. We used the term “coding matrix” as a synonym of codeword, and we are going to use “buffer matrix” as a synonym of the structure that has to contain it (either an ECLSA_matrix, a codec_matrix or a codewordBox depending on the context). Of course, coding matrixes, i.e. codewords, are potentially infinite, while the number of buffer matrixes is finite and coincides with the length of the buffer. When the context prevents any ambiguity the term “matrix” was used and will be still used not to make the text too awkward.

2.2 ECLSO

The ECLSO core has 3 main threads, corresponding to the 3 logical phases seen in the previous chapter. It is paramount to note that the 3 phases above, although logically concatenate, can be executed in parallel by ECLSO, if there is pressure. This is why we have 3 threads instead of just one. They are:

1) T1, Fill Matrix (from LTP)
2) T2, Encode Matrix
3) T3, Pass Matrix (to UDP)

We have also an ancillary stand-alone 4th thread

4) T4, Feedback Handling, which waits for decoding feedbacks coming back from the receiver.

The general process is the following (see Figure 13, where arrows in blue represent data, arrows in green signals for thread synchronization):

1) Segments come from the upper protocol and T1 puts them in the first available buffer matrix (1 in the figure); when the part of the buffer matrix devoted to hosting information symbols is full (I=K), or a timeout expires, T1 will mark it as ready to be encoded and will signal this to T2, should T2 be sleeping.
2) While T1 is busy with the filling of buffer matrix 1, T2 can be either sleeping or, as in the case shown in the figure, encoding or busy with the encoding of a previously filled matrix (2 in the figure). Once encoding of matrix 2 is completed, T2 marks it as ready to be sent, and informs T3, should T3 be sleeping.
3) While T1 and T2 are busy, T3 can be either sleeping, or, as in the figure, busy with the sending the data of another matrix to the lower protocol (3 in the figure). It may also happen that another matrix (4 in the figure) has already been encoded (after matrix 3 of course) and is waiting for transmission; Note than when there are multiple matrixes ready to be processed, the FIFO order is followed.

4) Once T3 has finished sending the matrix data to the lower protocol, the buffer matrix is flushed and marked as available; T1 is notified that a new buffer matrix is available, should it be locked waiting for a new matrix (this is clearly not the case in the figure, but T3 cannot know this).

T4 is always waiting for feedback coming from a receiver.

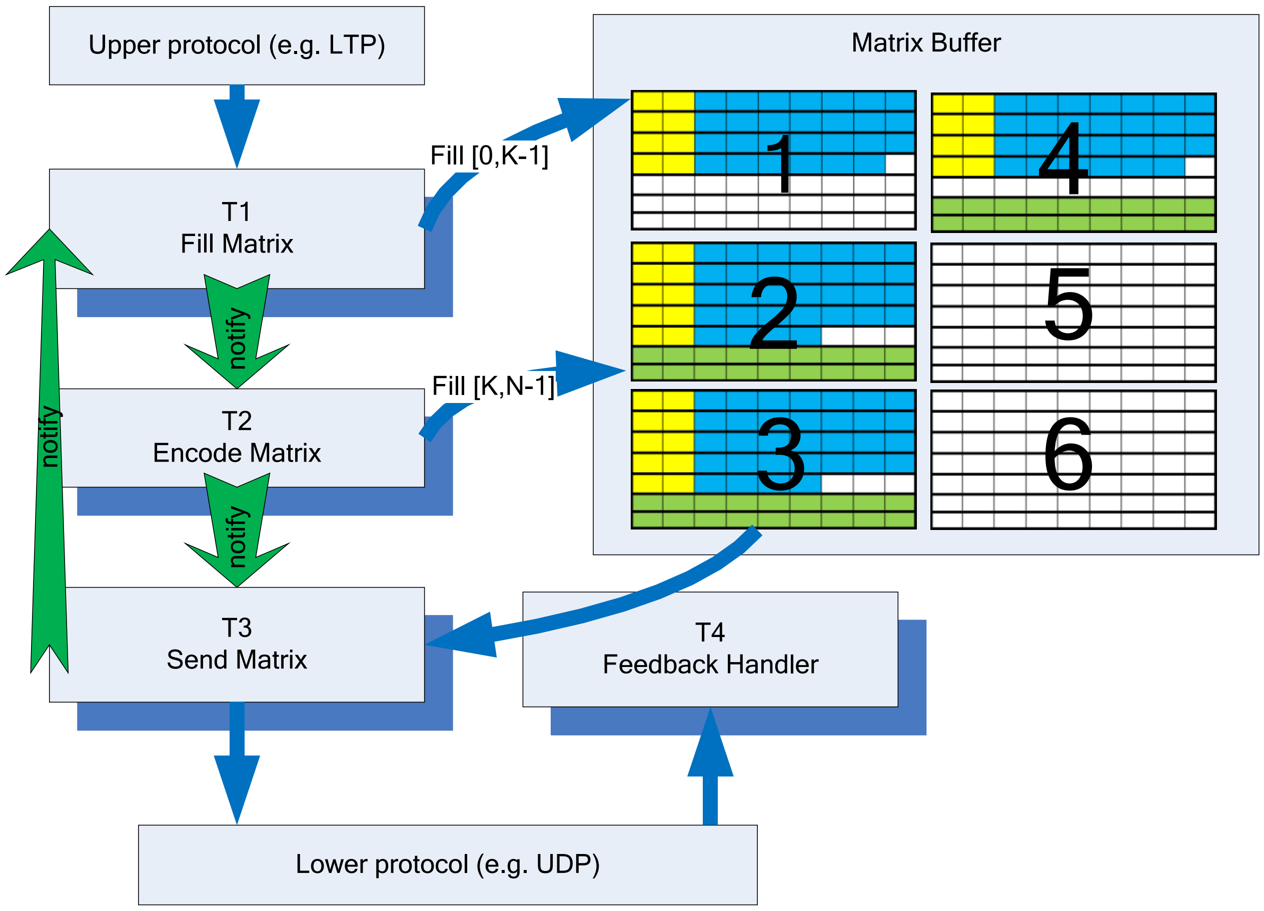


Figure 13: The workflow of ECLSO threads. Blue arrows represent data, green signaling for thread synchronization. In this a bit extreme example all threads work concurrently: matrix 1 is filled, 2 encoded and 3 transmitted in parallel by T1, T2 and T3. Matrix 4 is waiting to be sent while Matrixes 5 and 6 are empty and ready to be filled. No ancillary data are represented in the figure.

2.2.1 T1: fill_matrix_thread

As said, the role of this thread is to write data in the buffer matrix. The first operation is performed by a call to

```
void receiveSegmentFromUpperProtocol(segment,&segmentLength);
```

which receives one new LTP segment and puts it (and its size) in a temporary buffer. This procedure has a blocking read, waiting for new LTP segments; when a new LTP segment becomes available, a call to `getMatrixToFill(…)` is performed to find in which ECLSA_matrix the segment must be inserted; there are two possibilities, either there is already a matrix active (i.e. an ECLSA_matrix where the writing is ongoing), or, a new available matrix (i.e. empty) must be found. If there is not any buffer matrix either active or available, because the buffer is full, the call to `getMatrixToFill(…)` fails and T1 puts itself back to sleep and waits until T3, after flushing an ECLSA_matrix, notifies T1 that a new matrix is available. Otherwise, the function returns the matrix to be filled. If the matrix returned is not a new one, the segment is just inserted in the active matrix with a call to `addSegmentToEclsaMatrix`. Vice versa, when a new matrix is returned, the associated ancillary fields must be initialized first. The most relevant fields are:

- `EngineID` i.e. the sender ID;
- `maxAggregationTime`, the maximum amount of time that can elapse from the writing of the first row and the forced closing of the matrix, although not completely filled yet;
- `ID`, the sequence identifier of the coding matrix (codeword) that is going to be written in the ECLSA_matrix.

A timer, set to `maxAggregationTime,` is also started. After completing this initialization, the new buffer matrix becomes the active matrix and the segment arrived is inserted as before.

The filling cycle of the active matrix goes on until either the info part of the buffer matrix is full (I=K), or the timer expires (I<K). In the former case, the timer is stopped, the matrix is marked as ready to encode and T2 is notified to wake up in order to begin the encoding. If the timer expires, an asynchronous handler marks the partially filled buffer matrix as ready to encode and notifies T2 as before. Note that T2 could be already asleep, working on a previously filled matrix (as seen in the figure above, where T2 works on buffer matrix 2 while T1 fills buffer matrix 1).

The flow chart of the filling process is shown in Figure 14. Note that hexagons denote asynchronous events and red and green boxes locking calls and their unlocking. Where semaphores are involved in these calls, a semaphore symbol is also shown.

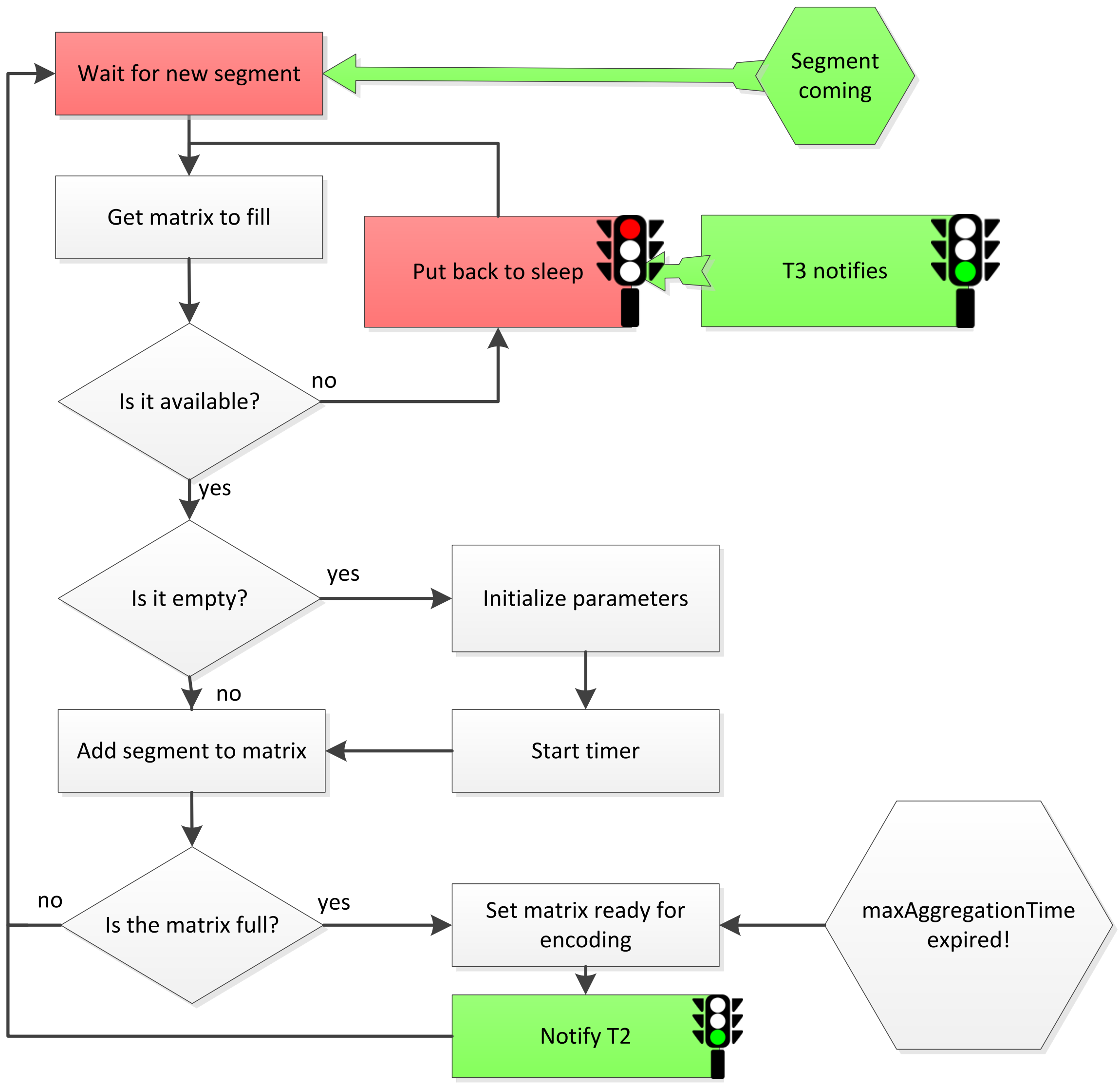


Figure 14: ECLSO T1. Dark orange boxes denote locking calls, green boxes unlocking calls. Semaphore symbols denote calls involving Dijkstra's semaphores. Hexagons represent asynchronous events.

2.2.2 T2: encode_matrix_thread

This thread has the task to encode the matrix; however, before doing this, it must select the most suitable erasure code implementing the logic described in the previous chapter (see Figure 6 and related description).

The selection of the code is done by the `getBestFec` function:

```
FecElement* getBestFec(unsigned int infoSegmentAddedCount, float currentRate)
```

The number of codes actually available (i.e. that can be used) in ECLSO depends on the ECLSO settings embedded in the span instruction. If Adaptive Encoding is disabled only the code specified in the span instruction is available. Otherwise, all codes with K≤Kspan are considered available, to a maximum of 9. We have an important exception when continuous mode is enabled, at present possible only with OpenFEC. In fact, in this case the code is tailored on the amount of information symbols written in the matrix before timeout expiring, by setting K=I. In practice, in this case all the codes with K in the range [Kmin,Kspan] are available, with continuity (Kmin is a parameter specified in the OpenFEC library, typically about 10).

The thread continues by calling `encodeCodecMatrix` with the parameters of the chosen code, to have the encoding performed. There are multiple implementation of this function, being specific to one codec library. We have one file for each implementation supported (LibecDLR, OpenFEC and DummyFEC). The right one is chosen at compilation, by means of parameter that specifies the codec library to use (e.g. `--OpenFEC`). In the next chapter the OpenFEC implementation will be described, while at the end of this chapter the sample "dummy" implementation will be introduced as a template for future extensions.

After the encoding, the matrix is marked as ready to send, and T3 is notified.

2.2.3 T3: pass_matrix_thread

This thread has to pass matrix rows (i.e. code-word symbols) to UDP (or to another lower layer). Once started, it waits for the first buffer matrix marked as ready-to-send from T2. When it is notified, it wakes up, builds the ECLSA header (see Figure 7) for each symbol and it starts passing them to UDP. For every packet, the SymbolID information in header is set, then the payload is filled with the symbol contained in the buffer matrix row and the packet is sent to UDP. Once all symbols are passed, the matrix buffer is flushed, and T1 is notified.

2.2.4 T4: feedback_handler_thread

This is an ancillary thread whose task is to wait for feedback packets sent back by the ECLSI in the receiver, once a matrix is decoded and passed to LTP.

As T1, T4 makes a blocking call to read feedback packets arriving from the listening port. Feedback packets are sent by means of the same lower layer protocol used by T3, i.e. usually by UDP.

The structure of the feedback packet was presented in the previous chapter and is not repeated here (see Figure 8). The process is the following: first, the MatrixID, which is a sequential index of codewords, is checked to verify if it is between the first matrix that has not received feedback yet and the last matrix that has been passed to UDP; if the checks is passed the feedback is processed otherwise discarded. This is a safety check also to prevent possible attacks.

On the basis of the data reported in the feedback datagram, the "Estimated Success Rate" is updated. It tries to estimate the complement of the packet loss probability, i.e. the probability that a UDP datagram sent by the present node is delivered successfully, by filtering data provided by consecutive feedbacks. This estimate is then used in the selection of the code, with the obvious aim of selecting a code whose redundancy is enough to ensure the correct decoding at the receiver side, but not in excess, i.e. the right Rc, as shown in the previous chapter.

Let us stress once again that the adaptive selection works well if the channel is stationary or if the coherence time of the channel is much longer than the RTT. Otherwise, there is a clear risk of instability. The great advantage of the adaptive selection lies in the fact that in the presence of a stationary or slowly varying channel, the user is get rid of the task of selecting the right code rate, as the best selection is automatically provided by the algorithm itself.

2.3 ECLSI

As ECLSO, ECLSI has 3 main threads corresponding to the 3 logical phases seen in the previous chapter:

1) T1, Fill Matrix (from UDP)
2) T2, Decode Matrix
3) T3, Pass Matrix (to LTP)

The ECLSI general process is the following (see Figure 15, where arrows in blue represent data, arrows in green signals for thread synchronization):

1) The payload of incoming ECLSA packets passed by UDP (or other lower layer protocol) are put by T1 in the right buffer matrix (2 in the figure); when one of the three completion conditions is met (see previous chapter and below), T1 will mark the matrix as ready to decode and will notifies this to T2, should T2 be sleeping.
2) While T1 is busy with the filling of buffer matrix 2, T2 can be either sleeping or, as in the case shown in the figure, busy with the decoding of the codeword contained in a previously filled matrix (3). Once decoding of 3 is completed, T2 will mark the buffer matrix 3 as ready to send, and will inform T3, should T3 be sleeping.
3) While T1 and T2 are busy, T3 may be either sleeping, or, as in the figure, busy, passing rows of another matrix (4) to LTP. It may also happen that another matrix has already been decoded (after matrix 4 of course) and is waiting for being processed by T3; Note than when there are multiple buffer matrixes ready to be processed, the FIFO order is followed.
4) Once T3 has finished passing the matrix to LTP (or other upper protocol), the buffer matrix is flushed and marked as available; a feedback is sent via UDP (or other lower protocol), and finally T1 is notified that a new matrix buffer is available, should it be locked waiting for a new matrix (this is clearly not the case in the figure, but T3 cannot know this).

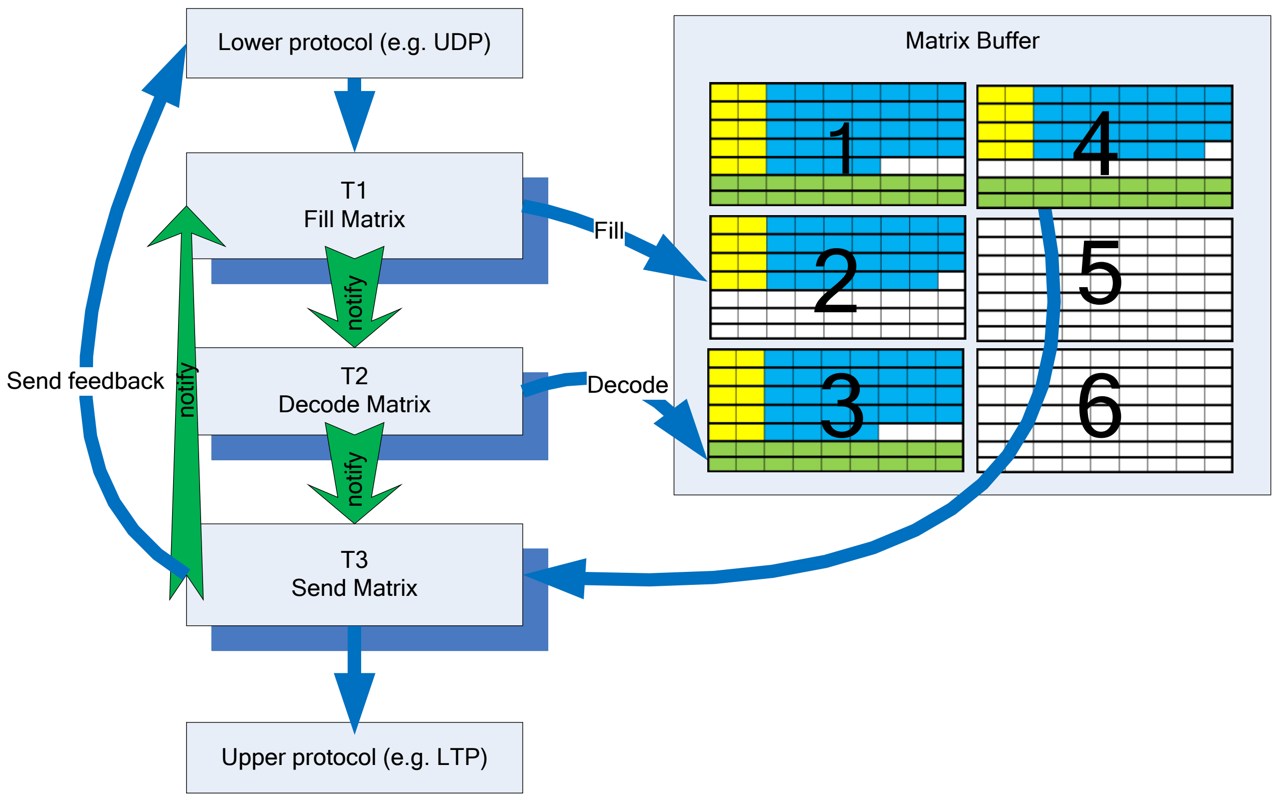


Figure 15: The workflow of ECLSI threads. Blue arrows represent data, green signaling for thread synchronization. In this extreme example all threads work concurrently: matrix 2 is filled with segments given by the lower layer protocol (UDP), matrix 3 decoded and 4 passed to the upper protocol (LTP) in parallel by T1, T2 and T3. Matrix 1 is waiting to be decoded while Matrixes 5 and 6 are empty and ready to be filled.

2.3.1 T1: fill_matrix_thread

This thread has to write data coming from UDP in the buffer matrix. It first performs a call to `receivePacketFromLowerProtocol` which receives one new packet from the lower protocol and put it (and its size) in a temporary buffer. This procedure performs a blocking reading.

After receiving the packet, a call to `parseEclsaIncomingPacket` is made to extract form the the ECLSA header the codeword it belongs to (the sequential Matrix Index and the EngineID). If the codeword is in a blacklist, the packet has arrived too late and is discarded. Otherwise, the symbol index (corresponding to the matrix row) and other parameters of the codeword are read too. A few checks are performed on these data and if they are not compliant with the current standard the packet is discarded.

The next operation is to find the buffer matrix where the symbol contained in the packet that has just arrived must be inserted. This operation is done by `matrix=getMatrixToFill(…)` and the filling process is the same as described for the ECLSO thread T1, except the exit conditions, which are made more complex by the fact that at reception some symbols could be missing. They have been explained in the previous chapter (see 1.5.1) and will not be repeated here.

2.3.2 T2 decode_matrix thread

This thread has the task of decoding the code-word contained in a buffer matrix to retrieve all K information symbols.

Once the code has been selected, the `decodeCodecMatrix` function is called, with the parameters of the chosen code. The implementation of this function depends on the codec library in use, as for its pair in ECLSO. The `decodeCodecMatrix` function will receive from the codec functions the “decoding state”, i.e. the result, “success” (all information symbols recovered) or “failure” (some gaps) plus additional information depending on the codec library used, such as the kind of algorithm used (iterative or Maximum Likelihood).

Each row containing an information symbol that has been received or recovered by the decoding process, is marked as valid by T2. At the end of this process the matrix buffer is marked as ready to pass, and T3 is notified.

2.3.3 Thread 3

This thread reads the content of buffer matrix rows marked as valid, i.e. the information symbol received or recovered, and pass them to LTP (or to another upper layer protocol). Before it is necessary to remove the 2B header and the row padding to obtain the original LTP segments.

The process is the following. At start up, T3 goes sleeping to wait for a buffer matrix marked as ready to pass to the upper protocol. When a matrix is marked as ready to pass by T2, T3 is notified and wakes up. At this point the corresponding MatrixID is added to a blacklist, to immediately discard possible late packets, i.e. ECLSA packets with the same MID arriving late, which otherwise would cause the writing of a new matrix. Then, T3 starts passing all the segments contained in the buffer matrix rows marked as valid. To do that, for each valid row T3 reads the segment length in the row header, to know the actual size of the original LTP segment, then reads the corresponding bytes and passes the reconstructed LTP segment to LTP. This way LTP will receive all available segments.

If the “Feedback Requested” option was set in the header of packets belonging to the received matrix, a feedback packet is built (see Figure 10).

Once all operations have been completed, the buffer matrix is flushed and marked as available, and T1 is notified.

2.4 CODEC ADAPTERS

Since the encoding and decoding functions in T2 threads of ECLSO and ECLSI are strictly bound to the codec library used, we choose to define them in a header file, and to implement a different version specific to each codec (at present LibecDLR in and OpenFEC, plus a dummy version to be used as a template for future expansions). At compilation time a parameter tells the compiler which version to choose among the following:

- `eclsaCodecAdapter_LibecDLR.c`
- `eclsaCodecAdapter_OpenFEC.c`
- `eclsaCodecAdapter_DummyDEC.c)`

The functions to be implemented in the codec interfaces are:

- `void FECArrayLoad(int envT,bool continuousModeEnabled);` this function "loads" the erasure codes available. Loads means that makes them available to the program. The exact function performed to this end is dependent on the Codec Adapter selected. If the codec library does not support the *on the spot* generation of erasure codes, the Continuous Mode cannot be enabled.
- `void destroyCodecVars(FecElement *fec);` this function frees the CodecVars allocated with FECArrayLoad.
- `int encodeCodecMatrix(CodecMatrix *codecMatrix,FecElement *encodingCode);` this function has to encode the codeword contained in the codecMatrix, the "container" of an ECLSA_matrix.
- `int decodeCodecMatrix(CodecMatrix *codecMatrix,FecElement *encodingCode,int paddingFrom,int paddingTo);` this function is the dual, it has to decode the codeword contained in the codecMatrix, the "container" of an ECLSA_matrix.
- `char convertToUniversalCodecStatus(char codecStatus);` this function has to convert the codec status provided by the codec in use, which assumes specific values, into an universal status identifier, as defined in `eclsaCodecMatrix.h:`
  typedef enum
- {
    - STATUS_CODEC_NOT_DECODED = 127,
    - STATUS_CODEC_SUCCESS = 1,
    - STATUS_CODEC_FAILED = 0
- } UniversalCodecStatus;
- `char *getCodecStatusString(int codecStatus);` this function has to return a string that describes in human language the meaning of the current codecStatus.
- `bool isContinuousModeAvailable();` this function has to return `true` if the codec supports the generation on the spot of erasure codes, `false` if it does not.

As said, there is a dummy implementation of these functions in a file called `eclsaCodecAdapter_DummyDEC.c`. This implementation obviously does not work if there are losses, since it does not call any real codec, but it has been included to encourage other researchers to extend the present support to new adapters.

3 Using an open-source library: OpenFEC

3.1 OpenFEC: description and API

OpenFEC is an open source FEC codec library whose main contributors, as stated in the website [36] are researchers of INRIA (Institut National de Recherche en Informatique et en Automatique) [37], and ISAE, Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-SUPAERO) [38].

This library is developed in C, in order to have high performance, and its code is optimized for the architecture it is compiled for (it works differently in 32 or 64 bit systems). It consist of many codec, supporting different types of erasure codes:

- Reed-Solomon stable codec over Galois Field GF(2^8)
- Reed-Solomon stable codec over GF(2^m)
- “1D-2D parity check matrix” stable codec
- “LDPC-staircase” stable codec
- “LDPC from file” advanced codec

The “LDPC from file” is an *advanced* codec as it can load arbitrary erasure codes defined in text files (see the API description inside the OpenFEC source files). This version, however, support only regular codes, this means that every column of the parity check matrix must contain the same number of symbols.

The LDPC-staircase codec implements a subset of the codes described in RFC 5170 [27], already described in Chapter **Errore. L'origine riferimento non è stata trovata.Errore. L'origine riferimento non è stata trovata.**.

All the supported erasure codes are used via a unique and simple API that is described in a small document included in the source code (howto_use_openfec_lib.pdf), which makes it easier to interface with the desired software. Unfortunately, the behavior of the library is not fully described, and so the programmer has to make some work in order to understand how this FEC library processes data in encoding and decoding phase.

3.2 Creating a Codec Adapter for OpenFEC

The LibecDLR Codec Adapter, developed by Nicola Alessi and already present in ECLSA when this thesis was started, was designed to work with the LibEC codec developed by DLR. To support the LDPC codes defined in the Orange Book [35], this codec must be initialized by reading the parity check matrixes contained in a series of files, and by saving them in suitable structures, called LibecVars, whose memory is allocated dynamically.

For the OpenFEC Codec Adapter, I chose to use the LDPC-staircase codec, which does not support the LDPC codes defined in the Orange Book, but a subset of RFC 5170 erasure codes. Although this choice was somewhat compulsory, as the Orange Book codes are irregular and OpenFEC lacks the support to irregular codes, it has the great advantage of allowing the user of creating on the spot the parity matrix, by means of a pseudo-random algorithm, thus getting rid of any files.

In OpenFEC every encoding and decoding of one coding matrix (i.e. one codeword) has to be carried out in one *codec session* since the OpenFEC library allocates dynamically only the resources required for a specific instance, particularly for decoding, and these resources have to be released after the decoding is completed and thus they cannot be used for processing other coding matrixes, even if the erasure code to use is the same.

In the following we will briefly describe the functions implemented by the OpenFEC Codec Adapter, defined in `eclsaCodecAdapter.h` (as said common to all adapters) and contained in `eclsaCodecAdapter_OpenFEC.c` (specific to OpenFEC).

3.2.1 FecArrayLoad function

This function "loads" the available FEC codes. Since in OpenFEC parity check matrixes are created on the spot, there are no files to be read and the only the only action performed by FecArrayLoad is the definition of a couple of arrays containing the erasure code N and K values of the files defined in the CCSDS Orange Book. These are the codes potentially available in OpenFEC. Those actually available will be a subset depending on the settings passed in the configuration file.

The `codecVars` inside the `FecElement` are not initialized here, but later they will be pointed to a structure that contains the variables related to each OpenFEC encoder / decoder session, i.e. the encoding or decoding of a specific codeword:

```
typedef struct
{
	of_status_t of_status[2];
	of_session_t * of_session[2];
	of_ldpc_parameters_t * of_parameters;
} OpenFECVars;

// this is an internal type used to define two distinct codecs inside OpenFECVars
enum CODEC { ENCODER, DECODER };
```

This structure can contain two sessions: the former for the encoder and the latter for the decoder session (to keep them independent), as defined in the enum CODEC. The two arrays inside `OpenFECVars` are `*of_status[2]`, which contains the codec status for the related session, `*of_session[2]`, which contains the structure that the OpenFEC library uses to identify a unique codec session.

The `of_status_t` is an enum that can have these values:

```
typedef enum
{
	OF_STATUS_OK = 0,
	OF_STATUS_FAILURE,
	OF_STATUS_ERROR,
	OF_STATUS_FATAL_ERROR
} of_status_t;
```

The first two values, OF_STATUS_OK and OF_STATUS_FAILURE, are the normal status values after every decoding; if the decoding has succeeded, all source symbols have been recovered, otherwise only a subset, or none. The other two values, OF_STATUS_ERROR and OF_STATUS_FATAL_ERROR, are anomalous conditions: the session either could have been initialized with bad parameters, or there could be no memory available on the machine where it is running, or the OpenFEC API functions may have been used improperly.

Each encoding/decoding session is initialized and destroyed when required, in the functions `encodeCodecMatrix` or `decodeCodecMatrix`.

3.2.2 initSession function

```
static void initSession(FecElement * fecElement, enum CODEC type)
```
This is an auxiliary function to initialize an ENCODER or a DECODER session as required.

As said before, the codecVars pointer inside fecElement contains the OpenFECVars structure. If a specific erasure code is used for the first time, all the parameters required for initializing the session are set in of_parameters. Since we are using the LDPC-staircase codec, the `of_ldpc_parameters_t` structure has to be used.

3.2.2.1 *The "of_ldpc_parameters_t" structure*

This is defined in `of_ldpc_staircase_api.h`, from OpenFEC source code:

```
typedef struct of_ldpc_parameters
{
	UINT32		nb_source_symbols;	/* must be 1st item */
	UINT32		nb_repair_symbols;	/* must be 2nd item */
	UINT32		encoding_symbol_length; /* must be 3d item */
	/*
	 * FEC codec id specific attributes follow...
	 */
	INT32		prng_seed;
	UINT8		N1;
} of_ldpc_parameters_t;
```

The parameters that have to be set are:

- `nb_source_symbols`: this equals to K;
- `nb_repair_symbols`: this equals to N-K;
- `encoding_symbol_length`: this equals to T * sizeof(uint8_t), since the symbol length is specified in bits;
- `prng_seed`: this parameter is defined in RFC 5170 [27]; it serves for the pseudo-random parity check matrix generator that defines the erasure code;
- `N1:` also this parameter is defined in RFC 5170 [27], and defines the amount of symbols contained in each row of the parity check matrix.

Note that two codecs initialized with the same prng_seed and N1 generate the same parity check matrixes for the same N and K, so that one node can decode a codeword generated by the other node, by means of a parity check matrix generated on the spot. This eliminates the need of saving parity check matrixes in advance. The newer RFC 6816 [28] defines also a specific header for the software using the LDPC Staircase FEC that contains also the prng_seed and N1, but at present ECLSA is not compliant to the RFC 6816 and these two parameters are simply hard-coded.

3.2.2.2 *Codec session initialization*

After the initialization of the structure, the codec session is initialized with

```
openFECVars->of_status[type] = of_create_codec_instance(&openFECVars->of_session[type],
OF_CODEC_LDPC_STAIRCASE_STABLE,codec_type,VERBOSITY);
```

where codec_type can be either `OF_ENCODER` or `OF_DECODER` and the `VERBOSITY` is set to either 0 or 1 (only for debug purposes). The other parameters are set with

```
openFECVars->of_status[type]=of_set_fec_parameters(
openFECVars->of_session[type],(of_parameters_t *)openFECVars->of_parameters);
```

using the structure initialized before. Finally, the OpenFEC library generates the required erasure code parity check equations (i.e. the parity check matrix).

3.2.3 destroySession function

This function is the opposite of initSession, as it releases the codec instance with

```
((OpenFECVars *) fecElement->codecVars)->of_status[type] =
of_release_codec_instance(((OpenFECVars *) fecElement->codecVars)->of_session[type]);
```

and points the `of_session[type]` to NULL. Note that the resources allocated by `initSession` are not completely freed, since another session could be initialized with the same erasure code, unless in Continuous Mode, where the alternative `destroyCodecVars` function is called instead (see below). In fact, in Continuous mode the code used must be dynamically changed block by block, to match the actual size of the current LTP block. The same call to `destroyCodecVars` is done when ECLSA is closed.

3.2.4 destroyCodecVars function

```
void destroyCodecVars(FecElement *fec)
```

This function frees in the `OpenFECVars` structure the `of_parameters` field, which contains the erasure code parameters dynamically allocated and pointed to `codecVars` inside `FecElement`. This function is called when the erasure code to be deleted was built in the Continuous Mode or when ECLSA is closed.

3.2.5 encodeCodecMatrix function

```
int  encodeCodecMatrix(CodecMatrix *codecMatrix,FecElement *encodingCode)
```

This function performs the encoding task of the FEC library: generating the M=K-N parity symbols from the K source symbols, given an (N,K) erasure code. To this end, this function first initializes a new OpenFEC session with `initSession(encodingCode,ENCODER);` after this, all the redundancy (parity) symbols (i.e. with indexes between K and N) are built with

```
openFECVars->of_status[ENCODER] = of_build_repair_symbol(

openFECVars->of_session[ENCODER],(void **) codecMatrix->codewordBOX,esi);
```

After all parity symbols have been inserted in the codewordBox, the codec session is closed with `destroySession(encodingCode,ENCODER)`. Actually, the initialization of a new OpenFEC session for every new codeword would not be strictly necessary for the OpenFEC encoder, but since this is mandatory for the OpenFEC decoder the same is done here.

3.2.6 decodeCodecMatrix function

```
int  decodeCodecMatrix(CodecMatrix *codecMatrix,FecElement *encodingCode,

int paddingFrom,int paddingTo)
```

This function is the converse of the previous one. However, it requires two additional parameters that indicate the *padding* range inside the codec matrix. More important, we need to stress that decoding is much more complex than encoding. It requires more CPU time, more memory, and last, but not least, is an operation that can fail. I encountered many problems in this phase to make OpenFEC work correctly with ECLSA, for two reasons: the first is that OpenFEC library uses a different format for missing symbols; the second is that the original code had a couple of memory management bugs that I had to fix. In this paragraph I am going to describe how I managed to use the `codecMatrix->codewordBox` with the OpenFEC library, while the problems about memory management will be described later in 3.2.8.

3.2.6.1 Format conversion of missing symbols: from ECLSA to OpenFEC.

As said before, in ECLSA all the matrix is allocated and zeroed before its use, so it can be seen as a matrix of N rows by T columns, where before decoding the rows of the received symbols are filled, while the ones corresponding to both padding rows and lost symbols remain filled by zeros. To distinguish padding rows form lost symbols, padding rows are marked as valid in the rowStatus. In the next image we can see an example of a buffer matrix corresponding `codecMatrix` where two symbols are missing:

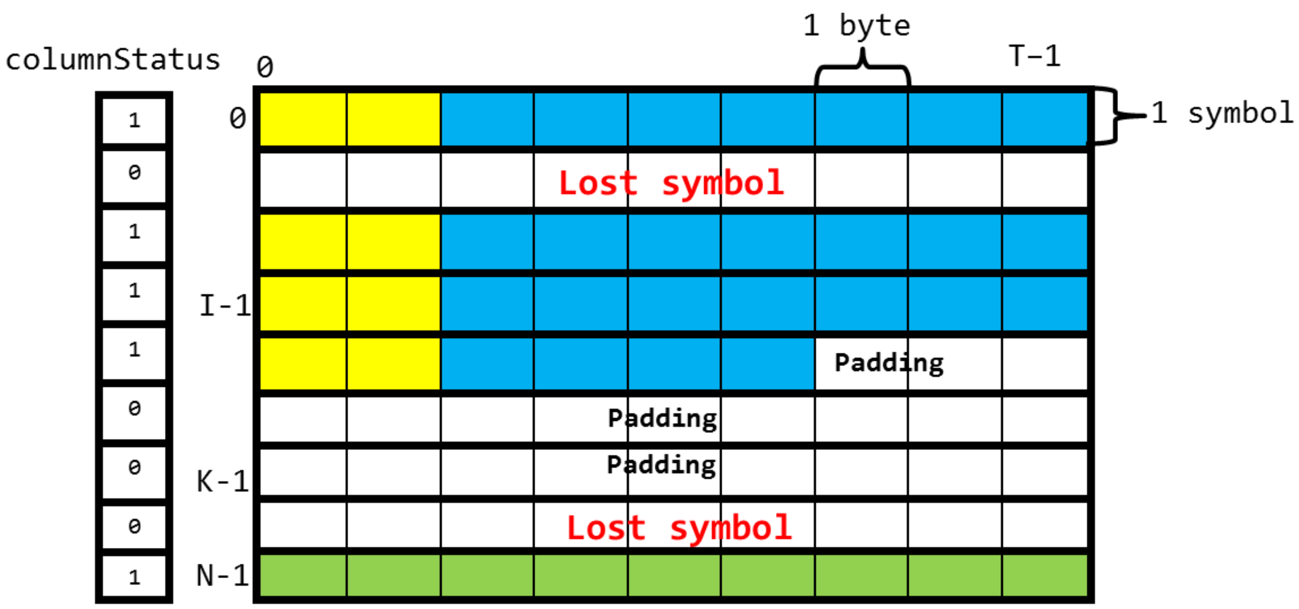


Figure 16: A codecMatrix where two symbols have been lost. In the left we have the rowStatus vector, in the right the codewordBox matrix. Please note the blank fields, as the padding: they are all set to zero, and the columnStatus is set to zero for the missing symbols.

This format worked well with the LibecDLR codec, for which ECLSA was first developed. Unfortunately, this is incompatible with OpenFEC, which requires the missing symbols be pointers to NULL, not zero rows. To make ECLSA compatible with OIpenFEC it is therefore necessary to convert the format of missing symbols, as seen in the next figure.

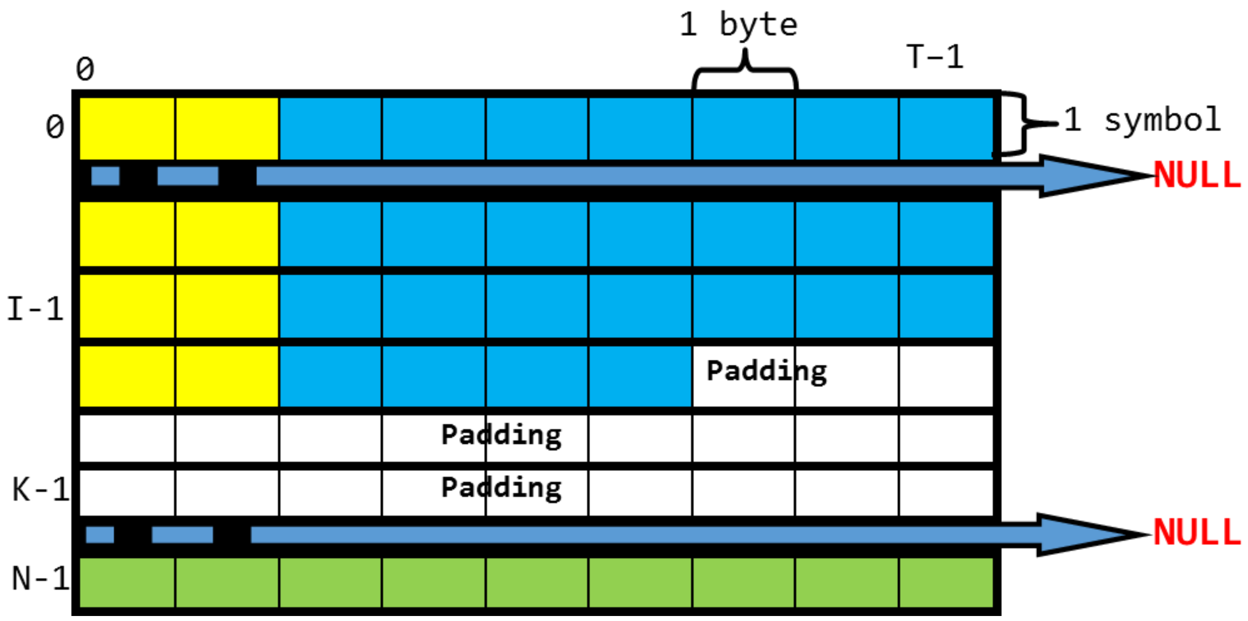


Figure 17: The buffer matrix of the previous figure example, after conversion to make them compatible with OpenFEC. The missing symbol rows are now pointers to NULL.

For the conversion an internal buffer is used, defined as `uint8_t ** in_data_vector`, where at the beginning only the rows are allocated:

```
in_data_vector =   (uint8_t **)calloc(encodingCode->N,sizeof(uint8_t *));
```

After this the OpenFEC session is initialized with initSession(encodingCode,*DECODER*) and the rows between the paddingFrom and paddingTo are marked as valid in the rowStatus of the codecMatrix. After this, for each row of valid data, a corresponding row is allocated in the in_data_vector** buffer, and the data from codecMatrix->codewordBox** is copied, otherwise that row points to NULL.

3.2.6.2 Decoding

At this point we are ready to send the symbols to the library with of_set_available_symbols:

```
openFECVars->of_status[DECODER] = of_set_available_symbols(

openFECVars->of_session[DECODER],(void **) in_data_vector);
```

After this primitive a first *trivial* decoding is done with the available symbols, as described in Chapter **Errore. L'origine riferimento non è stata trovata.Errore. L'origine riferimento non è stata trovata.**. If only few symbols were lost, at this point they could have been recovered. To know this, there is another primitive of the OpenFEC API: of_is_decoding_complete(openFECVars->of_session[*DECODER*]). If this is the case, the buffer is ready to be prepared for ECLSI, otherwise another primitive has to be called, which involves the resolution of the linear system of the parity check equation using Gaussian elimination:

```
openFECVars->of_status[DECODER] = of_finish_decoding(openFECVars->of_session[DECODER]);
```

Now, of_status[*DECODER*] should be either *OF_STATUS_OK* or *OF_STATUS_FAILURE* (unless an internal error has occurred). In both cases, all the available information symbols (all or in part, respectively) need to be passed to LTP.

3.2.6.3 Format conversion of missing symbols: from OpenFEC to ECLSA

To convert the format back to ECLSA, the OpenFEC API function that updates the buffer is called:

```
openFECVars->of_status[DECODER] = of_get_source_symbols_tab(

openFECVars->of_session[DECODER],(void **) in_data_vector);
```

All rows of the in_data_vector matrix not pointing to NULL are copied to the original codecMatrix->codewordBox** and the corresponding segmentStatus are set to 1. The missing rows are set to zero and their corresponding segmentStatus to 0. Every row of the in_data_vector** that was allocated before is freed and after that every column has been iterated, the in_data_vector ** itself is freed.

Finally, the OpenFEC decoder session is closed with destroySession(encodingCode,*DECODER*) to free all the memory allocated by the decoder.

3.2.7 Additional Codec Statuses

There are many codec statuses associated to the OpenFEC library, as we have seen before in the of_status_t structure, but I added a couple of them in order to provide the user with a more accurate information about the decoding process. These additional statuses are *OF_STATUS_SUCCESS_IT*, corresponding to the trivial (i.e. iterative) decoding success, and *OF_STATUS_SUCCESS_ML*, corresponding to the Gaussian elimination success. A part from these, other possible statuses are: *STATUS_CODEC_NOT_DECODED*, which is issued by ECLSA when no decoding has been performed, for example when skipped since all information symbols were received; finally, we have the already mentioned *OF_STATUS_FAILURE*, *OF_STATUS_ERROR*, *OF_STATUS_FATAL_ERROR*.

3.2.8 Bugs encountered in OpenFEC library

In addition to the problem related to the different format of missing symbols, I encountered also a couple of bugs related to memory management, which in C is a very delicate issue because with the malloc, memset, calloc, memcpy and free primitives the programmer has the freedom to do anything with memory, but he has the responsibility to free the allocated resources when they are no more needed, since there is no garbage collector in C. The two bugs are examined below.

3.2.8.1 of_decode_with_new_symbol and Valgrind

The OpenFEC API suggests, at the decoding phase, to load one symbol a time, so instead of using the of_set_available_symbols that we have seen in 3.2.6.2, the programmer may use the of_set_decode_with_new_symbol this way:

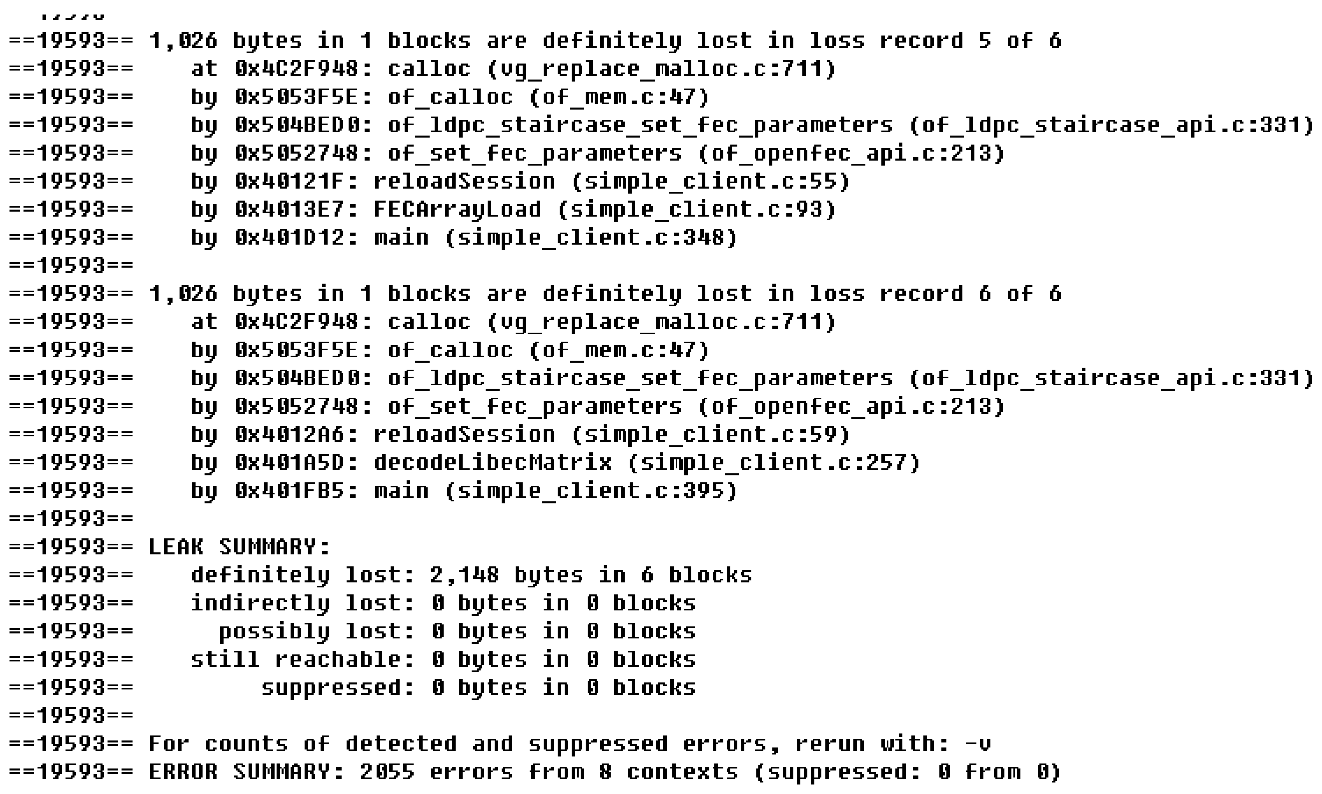
```
of_status_t  of_decode_with_new_symbol (of_session_t*     ses,
                                        void* const new_symbol_buf,
                                        UINT32            new_symbol_esi)
```

So, instead of sending the entire buffer matrix containing all the symbols, each symbol is passed via the **void*** **const** new_symbol_buf, and its ESI (the number of the symbol) passed via UINT32 new_symbol_esi. This approach works, and initially I was using this function to pass the symbols to OpenFEC. By the way after testing with a simple main program I made that was simulating the encoding and decoding of random data, after some iterations there was a very big memory consumption.

In order to identify what was the cause of this memory leakage, I used an open source tool called Valgrind [39], made exactly to identify the memory left over by the functions of the program when they return and see the call back trace to the malloc, memset or calloc that caused the leaking.

Valgrind in order to work needs the C program compiled with debug options enabled. Once Valgrind has been downloaded, compiled and installed, it is possible to run the program with memory leakage checking simply using the command valgrind –leak-check=yes myProgram args.

As you can see in the next image, there is a significant amount of memory left by a calloc when the OpenFEC session is released, after the decoding is done (at the time, the initSession and destroySession were in an unique reloadSession function):



```
==19593== 1,026 bytes in 1 blocks are definitely lost in loss record 5 of 6
==19593==    at 0x4C2F948: calloc (vg_replace_malloc.c:711)
==19593==    by 0x5053F5E: of_calloc (of_mem.c:47)
==19593==    by 0x504BED0: of_ldpc_staircase_set_fec_parameters (of_ldpc_staircase_api.c:331)
==19593==    by 0x5052748: of_set_fec_parameters (of_openfec_api.c:213)
==19593==    by 0x40121F: reloadSession (simple_client.c:55)
==19593==    by 0x4013E7: FECArrayLoad (simple_client.c:93)
==19593==    by 0x401D12: main (simple_client.c:348)
==19593==
==19593== 1,026 bytes in 1 blocks are definitely lost in loss record 6 of 6
==19593==    at 0x4C2F948: calloc (vg_replace_malloc.c:711)
==19593==    by 0x5053F5E: of_calloc (of_mem.c:47)
==19593==    by 0x504BED0: of_ldpc_staircase_set_fec_parameters (of_ldpc_staircase_api.c:331)
==19593==    by 0x5052748: of_set_fec_parameters (of_openfec_api.c:213)
==19593==    by 0x4012A6: reloadSession (simple_client.c:59)
==19593==    by 0x401A5D: decodeLibecMatrix (simple_client.c:257)
==19593==    by 0x401FB5: main (simple_client.c:395)
==19593==
==19593== LEAK SUMMARY:
==19593==    definitely lost: 2,148 bytes in 6 blocks
==19593==    indirectly lost: 0 bytes in 0 blocks
==19593==      possibly lost: 0 bytes in 0 blocks
==19593==    still reachable: 0 bytes in 0 blocks
==19593==         suppressed: 0 bytes in 0 blocks
==19593==
==19593== For counts of detected and suppressed errors, rerun with: -v
==19593== ERROR SUMMARY: 2055 errors from 8 contexts (suppressed: 0 from 0)
```

Figure 18: Valgrind showing the memory leakage in the test program.

At this point, I tried to use `of_set_available_symbols` to pass the entire symbol buffer instead of iterating with `of_set_decode_with_new_symbol`, and all the leaking disappeared. Since I don't know exactly how OpenFEC code works, I suppose there is something related to passing pointers stated as constant to the symbols, instead of passing the entire matrix directly, since the OpenFEC API specifies that OpenFEC works directly on the buffer matrix.

3.2.8.2 The 32-bit XOR bug

After completing the stand-alone developing and test, the library adapter was inserted in ION and tested, by using the DTNperf evaluation tool. After a while, we noticed that there was a problem on decoding data only on virtual machines running a 32-bit version of Linux. After some research on the web, I found a patch made by a Russian developer, Viktor Gaydov (nickname gavv), available on his GitHub site [40]. This patch corrects a bug in the file `of_symbol.c` of OpenFEC, which was discovered after the release of the last version, in December 2014. This problem is related to the casting of the symbols while making the XOR operations on them, which for performance optimization are made on UINT32 for 32 bit machines and on UINT64 for 64 bit machines. After applying the patch, the library worked perfectly, so the problem was resolved. It is a bit disappointing that this patch has not been applied yet to OpenFEC by its maintainers.

3.2.9 Trying to Use Orange Book LDPC codes with OpenFEC

After solving the previous bugs, I tried to use the *OF_CODEC_LDPC_FROM_FILE_ADVANCED* codec of the OpenFEC library, which allow to read the parity matrix of a FEC from a file, as in LibecDLR, with the aim of including the possibility of using the original LDPC files specified in the CCSDS Orange Book. Since the OpenFEC library has a unique API for all the erasure code families it supports, the changes to try this codec were minimal: instead of the `of_ldpc_parameters_t` to configure the parameters, I used `of_ldpc_ff_parameters_t`:

```
typedef struct of_ldpc_ff_parameters
{
      UINT32       nb_source_symbols;  /* must be 1st item */
```

```
	UINT32		nb_repair_symbols; /* must be 2nd item */
	UINT32		encoding_symbol_length; /* must be 3rd item */
	/*
	* FEC codec id specific attributes follow...
	*/
	/** Input file containing the binary parity check matrix, of size n-k x n */
	char	*pchk_file;
} of_ldpc_ff_parameters_t;
```

As you can see, the path to the text file containing the parity check matrix has to be specified in `*pchk_file`. This file, as described in the API documentation [41], must have the following structure:

```
number of rows
number of columns
number of source symbols
number of repair symbols
line_x1 col_y1 col_y2 col_y3 ...
line_x2 col_y1 col_y2 col_y6 ...
```

Example for a k=10, r=10 matrix :

```
10
20
10
10
0 0 1 3 4 5 10
2
1 2 3 6 7 8 10 11
2 2 5 6 7 9 11 12
3 3 4 6 8 9 12 13
4 4 5 6 8 9 13 14
5 0 1 4 5 8 14 15
6 0 2 4 7 8 15 16
7 0 1 3 5 7 16 17
8 1 2 3 6 9 17 18
9 0 1 2 7 9 18 19
```

Since the number of columns is fixed, the code must be regular. This is a problem, as the LDPC codes of the CCCSDS Orange book are vice versa irregular. I tried anyways to load the smallest LDPC parity check matrix provided by DLR, by adapting the file of that code and inserting a "-1" where there was an empty entry. After that I made a small fix to the function that loads the parity check matrix in OpenFEC, to ignore the entries with -1 and skip the insertion of the value on the internal structures representing the parity check matrix. After loading successfully the parity check matrix, however the encoding failed, making ECLSO crash with a SEGFAULT caused by an OpenFEC primitive. I suppose this is because the library's code is optimized to work with regular codes, since it uses the same code adopted for the LDPC Staircase tables, the only difference is that the parity check matrix instead of being built in compliance with RFC 5170 [27] is loaded from a file. After this unsuccessful attempt, I stopped trying, since the modification of a FEC library is an extremely tricky and time spending task, and it was not crucial to the ECSLA development, as the Orange book is by definition experimental.

4 BIBLIOGRAPHY

[1] A. McHahon and S. Farrell, "Delay- and Disruption-Tolerant Networking," *IEEE Internet Computing,* vol. 13, no. 6, pp. 82-87, Dec 2009.

[2] DTNRG, Internet Research Task Force DTN Reseach Group, 18 Feb 2010. [Online]. Available: https://web.archive.org/web/20100218141610/http://www.dtnrg.org/wiki.

[3] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall and H. Weiss, "Delay-Tolerant Networking Architecture, RFC 4838," IETF, 2007.

[4] K. Scott and S. Burleigh, "Bundle Protocol Specification, RFC 5050," IETF, 2007.

[5] S. Burleigh, M. Ramadas and S. Farrell, "Licklider Transmission Protocol - Motivation, RFC 5325," IETF, 2008.

[6] M. Ramadas, S. Burleigh and S. Farrell, "Licklider Transmission Protocol - Specification, RFC 5326," IETF, 2008.

[7] S. Farrell, M. Ramadas and S. Burleigh, "Licklider Transmission Protocol - Security Extensions, RFC 5327," IETF, 2008.

[8] Internet Engineering Task Force, "DTN Working Group (DTNWG)," 2016. [Online]. Available: https://datatracker.ietf.org/wg/dtnwg/charter/.

[9] C. Caini, H. Cruickshank, S. Farrell and M. Marchese, "Delay- and Disruption-Tolerant Networking (DTN): An Alternative Solution for Future Satellite Networking Applications," *Proceedings of the IEEE,* vol. 99, pp. 1980-1997, nov 2011.

[10] J. Postel, "Transmission Control Protocol – Protocol Specification, RFC 793," 1981.

[11] M. Allman, V. Paxson and E. Blanton, "TCP Congestion Control, RFC 5681," IETF, 2009.

[12] M. Duke, R. Braden, W. Eddy, E. Blanton and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents, RFC 7414," IETF, 2015.

[13] A. S. Tanenbaum and D. J. Wetherall, Computer Networks ed. 5, vol. 13, Prentice Hall, 2011.

[14] V. Paxson, M. Allman, J. Chu and M. Sargent, "Computing TCP Retransmission Timer, RFC 6928," RFC Editor, 2011.

[15] J. Padhye, V. Firoiu, D. Towsley and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," Proc. ACM SIGCOMM 1998.

[16] S. Burleigh, "Delay Tolerant Networking LTP Convergence Layer (LTPCL) Adapter," 2013.

[17] CCSDS, "734.0-G-1 "Rationale, Scenarios, and Requirements for DTN in Space"," 2010.

[18] CCSDS, "734.1-B-1 "Licklider Transmission Protocol (LTP) for CCSDS"," 2015.

[19] M. Demmer, J. Ott and S. Perreault, "Delay-Tolerant Networking TCP Convergence-Layer Protocol," RFC Editor, 2014.

[20] N. Alessi, S. Burleigh, C. Caini and T. D. Cola, "LTP robustness enhancements to cope with high losses on space channels," Bologna, 2016.

[21] Sourceforge, "DTN2," Dec 2016. [Online]. Available: https://sourceforge.net/projects/dtn/.

[22] Networking for Communications Challenged Communities, "N4C," 2012. [Online]. Available: http://www.n4c.eu/N4C-open-source-code.php. [Accessed December 2016].

[23] C. Caini, A. Amico and M. Rodolfi, "DTNperf3: A further enhanced tool for Delay-/Disruption- Tolerant

Networking Performance evaluation," in *2013 (IEEE) Global Communications Conference (GLOBECOM),* 2013.

[24] NASA, "Disruption Tolerant Networking," 17 Aug 2016. [Online]. Available: https://www.nasa.gov/content/dtn.

[25] Sourceforge, "ION," Dec 2016. [Online]. Available: https://sourceforge.net/projects/ion-dtn/.

[26] J. K. Wolf, "An introduction to error correcting codes," 2010. [Online]. Available: http://circuit.ucsd.edu/~yhk/ece154c-spr15/ErrorCorrectionIII.pdf.

[27] V. Roca, INRIA, C. Neumann, D. F. Thomson and STMicroelectronics, "Low Density Parity Check (LDPC) Staircase and Triangle Forward Error Correction (FEC) Schemes, RFC 5170," 2008.

[28] V. Roca, M. Cunche and J. Lacan, "Simple Low-Density Parity Check (LDPC) Staircase Forward Error Correction (FEC) Scheme for FECFRAME, RFC 6816," IETF, 2012.

[29] Watson, M.; Netflix, Inc.; Begen, A.; Cisco; Roca, V.; INRIA, "Forward Error Correction (FEC) Framework, RFC 6363," IETF, 2011.

[30] INRIA, "Tools for Large Scale Content Distribution," Jul 2008. [Online]. Available: http://planete-bcast.inrialpes.fr/. [Accessed October 2016].

[31] INRIA, ISAE, "Welcome to the OpenFEC.org project!," Dec 2014. [Online]. Available: http://openfec.org/. [Accessed November 2016].

[32] V. Zyablov and M. Pinsker, "Decoding Complexity of Low-Density Codes for Transmission in a Channel with Erasures," *Problemy Peredachi Informatsii,* pp. 15-28, Mar 1974.

[33] M. Cunche and V. Roca, "Improving the Decoding of LDPC Codes for the Packet Erasure Channel with a Hybrid Zybalov Iterative Decoding/Gaussian Elimination Scheme," *INRIA Research Report RR-6473,* Mar 2008.

[34] T. d. Cola, E. Paolini, G. Liva and G. P. Calzolari, "Reliability Options for Data Communications in the Future Deep-Space Missions," *Proceedings of the IEEE,* vol. 99, no. 11, p. 2069, 2011.

[35] CCSDS, "131.5-O-1 Erasure Correcting Codes for Near Earth and Deep Space communications," 2014.

[36] OpenFEC.org, "OpenFEC.org - about us," Mar 2016. [Online]. Available: http://openfec.org/aboutus.html.

[37] INRIA, "INRIA IN BRIEF," [Online]. Available: https://www.inria.fr/en/institute/inria-in-brief/history-of-inria.

[38] ISAE-SUPAERO, "ISAE-SUPAERO," [Online]. Available: https://www.isae-supaero.fr/en/about-isae-supaero/isae-supaero-78/isae-supaero/.

[39] Valgrind™ Developers, "Valgrind Home," 2016. [Online]. Available: http://valgrind.org/.

[40] V. Gaydov, "openfec-1.4.2-32bit.patch," Oct 2016. [Online]. Available: https://gist.github.com/gavv/4325c090fc21a3033988ad745c03bdff#file-openfec-1-4-2-32bit-patch.

[41] V. Roca, J. Detchart, M. Cunche, V. Savin and J. Lacan, "Using the OpenFEC.org Library: a Simple User Guide," 2014.