Alma Mater Studiorum - Università di Bologna

scuola di ingegneria e architettura

Dipartimento di Informatica - Scienza e Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA
in
ARCHITETTURE E PROTOCOLLI PER RETI SPAZIALI M

Erasure Error Correcting Codes Applied to DTN Communications

CANDIDATO:
Pietrofrancesco Apollonio

RELATORE:
Chiar.mo Prof.
Carlo Caini
CORRELATORI:
Ph. D. Tomaso de Cola
Ph. D. Gianluigi Liva

Anno Accademico 2012/2013
III Sessione

CONTENTS

ACRONYMS

DTN	Delay Tolerant Networking
ARQ	Automatic Retransmission reQuest
EC	Erasure Codes
ION	Interplanetary Overlay Network
API	Application Programming Interface
LTP	Licklider transmission protocol
LSA	Link Service Adapter
LSI	Link Service Adapter Input
LSO	Link Service Adapter Output

1

ABSTRACT IN ITALIAN

La realizzazione di protocolli e di tecnologie di comunicazione applicabili in ambienti spaziali presenta notevoli problemi implementativi, principalmente se sono destinati ad essere utilizzati a livello fisico o a livello di trasporto. In quest'ultimo caso, infatti, la mancanza di connettività continua, i ritardi molto lunghi nella trasmissione dei segnali e le frequenti perdite penalizzano molto le comunicazioni sul canale.

In ambienti che presentano i problemi sopraccitati, i normali protocolli della suite TCP/IP sono difficilmente utilizzabili[1]. Inoltre, per la criticità delle comunicazioni effettuate nello spazio, l'affidabilità è un requisito fondamentale, non essendo possibile perdere informazioni o riceverle con ritardi spropositati a causa dei problemi presenti sul canale.

In protocolli utilizzati in ambiti terresti, come ad esempio TCP, l'affidabilità è solitamente ottenuta per mezzo di metodi basati su meccanismi di *ARQ* (*Automatic Retransmission reQuest*), che però non presentano buone prestazioni in caso vi siano lunghi ritardi sul canale di trasmissione[2]. Un altro metodo per ottenere affidabilità, utilizzato però a livello fisico, è l'inserimento di *Forward Error Correction codes* (*FEC*). Questi codici si basano sull'inserimento di informazioni di ridondanza[2][3] direttamente nel flusso dati trasmesso. Nei canali binari, dove singoli bit vengono invertiti a causa del rumore presente, questa ridondanza viene utilizzata per riordinare le informazioni originali, mentre, in caso di canali binari dove i bit trasmessi non vengono modificati ma persi (*Binary Erasure Channels*), la ridondanza viene invece utilizzata per recuperare queste informazioni. I codici FEC appositamente progettati per quest'ultimo tipo di canale vengono solitamente chiamati *Erasure Codes* (*EC*). Nonostante essi siano stati studiati, in primo luogo, per canali binari, gli EC possono essere utilizzati anche a livelli superiori, dopo essere stati opportunamente modificati per lavorare su interi pacchetti anziché su singoli bit[4]. Gli EC che lavorano a livello di pacchetto offrono una interessante alternativa ai meccanismi di ARQ, principalmente quando sono presenti dei lunghi ritardi sul canale di trasmissione.

Il protocollo TCP non è particolarmente efficiente quando è utilizzato in canali con lunghi tempi di propagazione. In TCP infatti, per assicurare la corretta trasmissione dei dati, si fa largo uso di feedback inviati dal ricevitore (ACK). Come in tutti i protocolli di questo tipo, con TCP si ottengono scarse prestazioni quando i tempi di risposta si allungano. Per cercare di risolvere questi problemi, è stato appositamente progettato un nuovo protocollo, chiamato *Licklider Transmission Protocol* (*LTP*). Il suo scopo principale

è quello di sostituire TCP (ma anche a UDP) quando si lavora in ambienti spaziali dove, date le gradi distanze da coprire, i tempi di propagazione sono molto lunghi. I dati trasmessi con LTP possono essere di due tipologie: *Red* and *Green*. Se vengono utilizzati dati di tipo Red, si richiede che essi siano trasmessi affidabilmente. Per questo motivo, LTP utilizza un protocollo basato su ARQ per risolvere eventuali errori. I vantaggi di LTP sono però che questo nuovo protocollo richiede meno feedback da parte del ricevente. Infatti è solitamente richiesto un singolo ACK per tutti i dati Red di un "blocco". In pratica, si utilizzano blocchi di dati da confermare molto più grandi di un pacchetto TCP. Un ulteriore vantaggio di questo protocollo è il fatto che il tasso di trasmissione dei dati è fissato a priori e che, quindi, non viene influenzato in nessun modo da lunghi ritardi. Seppur migliore di TCP, basandosi comunque su ARQ, il bisogno di ritrasmissioni può ancora incidere negativamente sulle performance in caso di ritardo e tasso di perdita molto alti. Per ovviare a questi problemi, in LTP si può utilizzare la seconda modalità di trasmissione dei dati, che invece è chiamata Green. In questo caso, il protocollo non si preoccupa di rendere affidabile la trasmissione dei dati, comportandosi similmente a UDP. La mancanza di affidabilità è però bilanciata dal fatto che la comunicazione è resa più veloce, non venendo in questo caso aggiunti ulteriori ritardi nella trasmissione delle informazioni. Un altro importante vantaggio, nel caso che i dati siano inviati utilizzando Green, è il fatto che il protocollo può essere utilizzato anche su canali monodirezionali dove, ovviamente, non è possibile pretendere la trasmissione di dati di controllo sul canale di ritorno. Per tutte queste ragioni, LTP è il candidato ideale ad essere esteso con le funzionalità fornite dagli EC. Infatti, in entrambe le modalità di trasmissione, l'utilizzo di questi codici può portare a miglioramenti delle performance. Se si utilizza Red, il numero di ritrasmissioni è limitato al solo caso in cui gli EC non riescano a recuperare tutte le informazioni perse, diminuendo di conseguenza il tempo necessario al corretto invio dei dati. In caso di Green, invece, l'utilizzo dei codici a correzione di errore può aggiungere resistenza alle perdite, considerando comunque che, teoricamente, l'affidabilità non può essere ottenuta senza utilizzare feedback.

Lo scopo di questa tesi è l'applicazione degli EC al protocollo LTP, partendo dagli studi preliminari e dall'implementazione dei codici veri e propri compiuti da due dei miei supervisori: *Tomaso de Cola* and *Gianluigi Liva*. Il lavoro svolto per questa tesi può essere diviso in tre fasi. Prima di tutto è stato necessario studiare il protocollo LTP e progettare le modifiche da apportare a quest'ultimo per aggiungere il supporto ai codici a correzione di errore (queste fasi vengono trattate nei Capitoli 3 e 4). La seconda fase è stata l'implementazione del codice necessario e la fase di test preliminari (Capitoli 5 e 6). Infine, sono stati compiuti degli esperimenti su simulazioni di scenari reali, per valutare i miglioramenti alle performance ottenuti utilizzando l'estensione del protocollo LTP implementata (Capitolo 7). Le prime due fasi sono state svolte a Monaco

di Baviera presso il *Deutsches Zentrum für Luft- und Raumfahrt* (*DLR*), ossia l'agenzia spaziale tedesca, grazie ad una borsa di studio di 6 mesi offertami dall'istituto appena menzionato. L'ultima fase è stata invece svolta presso l'*Università di Bologna*. Sia il mio relatore che i miei correlatori mi hanno supportato per tutta la durata del lavoro di tesi, coordinando il mio lavoro.

Il codice è stato implementato per una versione del *DTN Bundle Protocol* (*BP*) sviluppata dal *NASA-JPL* (*National Aeronautics and Space Administration, Jet Propulsion Laboratory - California Institute of Technology*). Il nome di questa implementazione è *Interplanetary Overlay Network* (*ION*). La nostra speranza è che il codice sviluppato venga inserito nelle prossime release di tale software.

I risultati dei test preliminari svolti per questa tesi sono stati presentati al *Consultive Committee For Space Data Systems (CCSDS) Spring Meeting*[5] nel Giugno 2013 ed anche al *Consultive Committee For Space Data Systems (CCSDS) Fall Meeting*[6] nell'Ottobre 2013. I risultati finali, inclusi in questa tesi, verranno presentati al prossimo CCSDS Meeting.

2

INTRODUCTION

The space environment has always been one of the most challenging for communications, both at physical and network layer. Concerning the latter, the most common challenges are the lack of continuous network connectivity, very long delays and relatively frequent losses. Because of these problems, the normal TCP/IP suite protocols are hardly applicable[1]. Moreover, in space scenarios reliability is fundamental. In fact, it is usually not tolerable to lose important information or to receive it with a very large delay because of a challenging transmission channel. In terrestrial protocols, such as TCP, reliability is obtained by means of an *ARQ* (*Automatic Retransmission reQuest*) method, which, however, has not good performance when there are long delays on the transmission channel[2].

At physical layer, *Forward Error Correction Codes* (*FECs*), based on the insertion of redundant information[2][3], are an alternative way to assure reliability. On binary channels, when single bits are flipped because of channel noise, redundancy bits can be exploited to recover the original information. In the presence of binary erasure channels, where bits are not flipped but lost, redundancy can still be used to recover the original information. FECs codes, designed for this purpose, are usually called *Erasure Codes* (*ECs*). It is worth noting that ECs, primarily studied for binary channels, can also be used at upper layers, i.e. applied on packets instead of bits[4], offering a very interesting alternative to the usual ARQ methods, especially in the presence of long delays.

TCP is not a good choice on channels with large delays for a variety of reasons. Without entering into too many details, it is enough to recall that many features of TCP are based on the receiver's feedback (i.e. on ACKs). As in all feedback protocols, performance is severely impaired when the feedback loop time becomes large, which is obviously the case of space environments. As an alternative to TCP (and also to UDP), the *Licklider Transmission Protocol* (*LTP*) has been created to obtain better performance on long delay links.

Data transmitted with LTP can be divided into two parts: *Red* and *Green*. To enforce reliability on the Red part data, LTP uses ARQ methods, as TCP, to recover errors occurred during the transmission. The advantage over TCP is that LTP is less "chatty" than TCP, as it basically requires just an ACK for the entire Red part of a "block" (an LTP packet usually much larger than a TCP packet). Moreover, its transmission rate is fixed a priori, so that it is not influenced by the long delay. However, although performance in LTP are better than TCP, the need of retransmissions can still have a negatively impact

if the channel has long delays and losses. By contrast, LTP does not enforce any reliability on the Green part data, thus behaving like UDP. The lack of reliability is counterbalanced by the advantage of not adding further delays in transmitting the information. Green parts can also be used with mono-directional channels because of the absence of the acknowledgement flowing back from the destination to the source. For the reasons just mentioned, LTP appears as an ideal candidate for the application of ECs at upper layers. In both Green and Red parts, the use of ECs can lead to better performance. In fact, using this family of codes, retransmissions are limited in case of Red (except when ECs fail, i.e. rarely and only in very unfavourable conditions) while, in case of Green, ECs add robustness against losses (considering that absolute reliability cannot be obtained without feedbacks).

The aim of this thesis is the application of ECs to LTP. I started from the preliminary studies and from the design of the ECs, both carried out by two of my supervisors: *Tomaso de Cola* and *Gianluigi Liva*. I have also benefited by the great experience on *Delay-Tolerant Networking* (*DTN*) space communications of my supervisors. This work has involved three logical phases. First, the study of the protocols and the design of the modifications to be introduced into the original LTP code (that will be debated in Chapter 3 and Chapter 4), then, their actual implementation and testing (Chapter 5 and Chapter 6), finally, the evaluation of performance improvement achieved by means of the implemented LTP extension on realistic space scenarios (chapter Chapter 7).

The first two phases have been carried out at the *Deutsches Zentrum für Luft- und Raumfahrt* (*DLR*), the German Aerospace Agency, thanks to a 6 month grant, generously offered to me by this institution. This thesis has been finally completed at the *University of Bologna*. In these three steps, both my Academic supervisor and the two DLR supervisors have supported my work coordinating my efforts.

The code I developed has been included into the implementation of *DTN Bundle Protocol* (*BP*) made by *NASA-JPL* (*National Aeronautics and Space Administration, Jet Propulsion Laboratory - California Institute of Technology*), called *Interplanetary Overlay Network* (*ION*). Our hope is to have our LTP extension included into the next official ION release.

Preliminary results of this work has been presented at the *Consultive Committee For Space Data Systems (CCSDS) Spring Meeting*[5] in June 2013 and at the *Consultive Committee For Space Data Systems (CCSDS) Fall Meeting*[6] in October 2013. The final results included in this thesis will be presented at the next CCSDS Meeting.

3 SPACE COMMUNICATIONS

3.1 OVERVIEW

In 2002, the *Delay Tolerant Networking Research Group* (*DTNRG*), which is part of the *Internet Research Task Force*[7] (*IRTF*), was formed. It is a long term research group in charge of the development of a new architecture and new protocols to cope with the problems of "challenged networks", i.e. of networks where the standard TCP/IP protocols cannot operate correctly because of the presence of one or more of the following challenges: long delays, high loss, link asymmetry, disruption, lack of end-to-end connectivity. Among challenged networks, there are, in a prominent role, space communications, which are affected by many of the problems just mentioned. In 2003 a first draft was published and after some years a new RFC of a standard architecture[8] and a suite of protocols called *Delay-Tolerant Networking* (*DTN*) was finally released. After some time the *Consultive Committee For Space Data Systems*[9] (*CCSDS*), a multi-national forum composed of the major space agencies of the world (i.e. NASA, ESA, DLR), started to work on these RFC (i.e. BP, DTN, LTP) to use them as standard protocols for communications in future space missions.

3.2 STANDARDS

The *Delay- and Disruption-Tolerant Networking*[8] (DTN) architecture introduces an overlay protocol that interfaces with either the transport layer or lower layers. In this architecture, each node can store information for a long time before forwarding it and, thanks to these features, DTN is particularly suited to cope with the problems previously discussed. DTN is also essential in "data mule applications", characterized by the absence of a continuous path between the source and the destination. The essential point is that in such an overlay, delays and disruptions can be handled at each DTN "hop" in a path between a sender and a destination. Nodes on the path can then provide the storage necessary for application data before forwarding them to the next node on the path. This architecture confines the end-to-end features of the transport layer to homogeneous network segments (namely, A, B and C in Figure 1), while end-to-end data transfer, across the heterogeneous network, is provided by the *Bundle Protocol*[10] (*BP*), which is an implementation of the DTN architecture. Blocks of data transmitted by the BP are called *bundles*. A bundle is a message that carries application layer protocol data units (*APDU*),

i.e. sender and destination names, and any additional data required for end-to-end delivery. BP can be interfaced with other protocols underlying it through *Convergence Layer Adapters* (*CLAs*), as can be seen in Figure 1. Various CLAs have been defined, including the ones for TCP, UDP, and LTP[11][12][13]. The last protocol will be explained in Section 3.4.

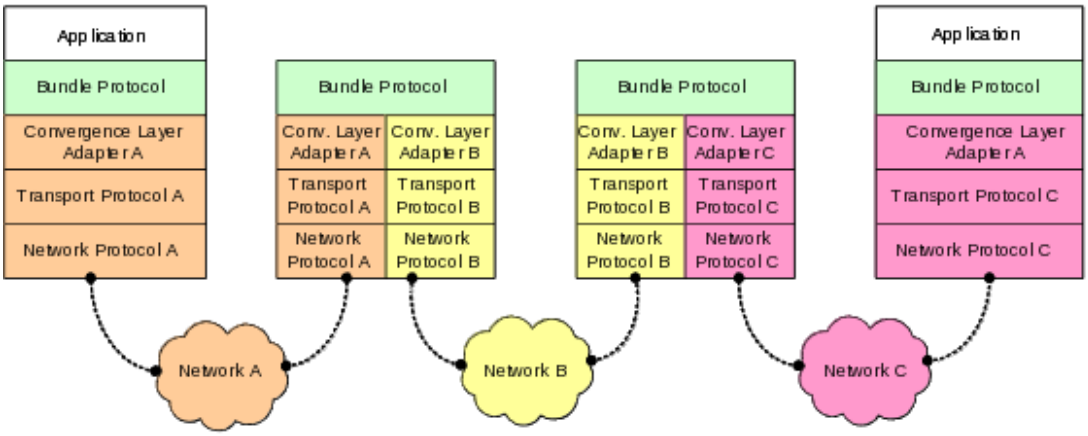

Figure 1: Stack with BP Layer

3.3 DTN IMPLEMENTATIONS

The most important implementations of the BP are:

- *DTN2*
- *Interplanetary Overlay Network* (*ION*)

3.3.1 *DTN2*

This implementation provides a framework for experimentation, extension, and real-world deployment. The code is instrumented with logging both for debugging and operational purposes. The core implementation is written in C++ using a framework called Oasys, that is designed to provide a uniform interface to the DTN2 code hiding operating system and other support package differences. More information can be found on the official page[14], and the source code, released as free software, can be downloaded on Sourceforge[15].

3.3.2 *ION*

Interplanetary Overlay Network (*ION*) is a BP implementation by *NASA Jet Propulsion Laboratory* (*JPL*) and other contributors (i.e. *Ohio* and other Uni-

versities) explicitly focused on deep space applications[16]. One of the most important peculiarity of ION is that it contains an implementation of the LTP protocol (explained later, in Section 3.4) and *Contact Graph Routing* (*CGR*), a DTN routing for deterministic intermittent connectivity typical of space environments. CGR is built on scheduled links ("contacts"), which in turns are actually enforced only for LTP. Unlike DTN2, ION does not implement reactive bundle fragmentation, although it can cope with bundle fragments, as requested by RFCs. Similarly to DTN, the ION code is open source and available from Sourceforge[17]. The latest release is 3.2.0. I used ION as BP implementation, because my work is closely related to the LTP protocol.

3.4 LTP

LTP was designed to provide retransmission-based reliability over links with extremely long RTT and/or frequent interruptions in connectivity. LTP can be used to serve as a reliable convergence layer for BP over single-hop in all these scenarios. A peculiar characteristic of LTP is its minimum "chattiness", which makes it suitable for deep space point-to-point links. In fact, there is no connection set-up, thus saving some RTTs in the initial phase. Besides, no real congestion or flow control involving exchange of information between nodes is implemented in the protocol.

A unit of data sent by LTP is called *LTP block* (well explained after) and it identifies a communication session (*LTP session*) between two corresponding LTP nodes. For each LTP Block a new LTP thread, is initiated and the maximum number of possible parallel sessions poses a requirement of corresponding storage capacity at a node. An LTP session is always unidirectional, therefore LTP peers can only achieve a bidirectional exchange of application data by using two independent LTP sessions. The configuration of maximum number of parallel sessions is done statically, e.g. during space mission planning phase. Finally, LTP may also implement a rate-based congestion control, which avoids the saturation of network node buffers, based on transmission link configuration, which is notified to each LTP node by means of a periodically updated contact table. In the case of ION, for each link between two nodes, this table contains the propagation delay, the available rate and the contact durations, also used by LTP to schedule data transmission. In this way two LTP nodes can send and receive data as soon as (and as long as) the link between them is available, which leads to optimal utilization of the contact window.

Processing of data units from the Bundle Layer to the underlying layer is done in three steps: bundle aggregation into one block, block segmentation and segment transmission. First, the BP LTP convergence layer adapter combines the bundles from the BP layer into one *LTP block*, which is in turn forwarded

to LTP protocol entity. Block size is selected according to space mission requirements. Next, the LTP protocol entity divides any incoming block into a number of *LTP segments*, whose maximum size depends on the underlying layer protocol to which they are finally forwarded. For instance, if the LTP layer is on top of UDP, each segment will be a UDP Datagram (max 64kB, usually about 1500B).

An LTP block consists of two parts: Red and Green. The former requires a reliable transfer, like TCP, the latter, like UDP, does not. The red segments must be reliably delivered to destination, by using NAK-based ARQ mechanisms. By contrast, the green ones are just sent without any retransmission mechanism. A block contains both a Red and a Green part or only one of the two. The way reliability levels are assigned to blocks or part of blocks is implementation dependent, although appropriate mapping from the *Extended Class of Service* (*ECOS*)[18] could be used. Data transfer between two LTP peers also includes the exchange of administrative reports from destination to sender, to solicit the retransmission of the missing Red segments or to notify correct block or segment reception. *Check-Point* (*CP*) segments can be issued from the sender asking the receiver to send a *Report Segment* (*RS*) to acknowledge all received segments, or alternatively to inform the sender about the missing segments (i.e., NAK). In both cases, the sender will eventually generate a *Report Acknowledgement* (*RA*), followed by retransmitted segments in case of losses. Finally, there are some dedicated flags in the last segment of the Red section, or of the block, respectively. These flags are: the *End of Red Part* (*EORP*), which signals the completion of Red part and the *End of Block* (*EOB*) indicating the end of the entire block.

A typical LTP session is shown in Figure 2. In this example, an LTP block, made up of nine segments (six Red and three Green), is transmitted over a deep space link. The segment #3 is flagged as CP and the Receiving LTP Engine must reply to the sender with a RS that, in turn, has to acknowledge it with the RA segment. The last Red segment (#6), which is flagged as both CP and EORP, is lost during the transmission. The sending LTP engine starts a timer, when it ends (in this case, after the transmission of the Green part) it will be retransmitted. As we can see in the figure, the receiving LTP client is informed of the end of the transmission only when all the Red segments are received. Green part segments, as explained before, have a different behaviour. In fact, as we can see in the figure, the LTP client on the receiver side is informed whenever a Green segment is received. Moreover, Green segments are not retransmitted when lost, as we can see for the segment #8. When all the segments are received, the LTP session can be closed at both ends and the corresponding buffer space freed.

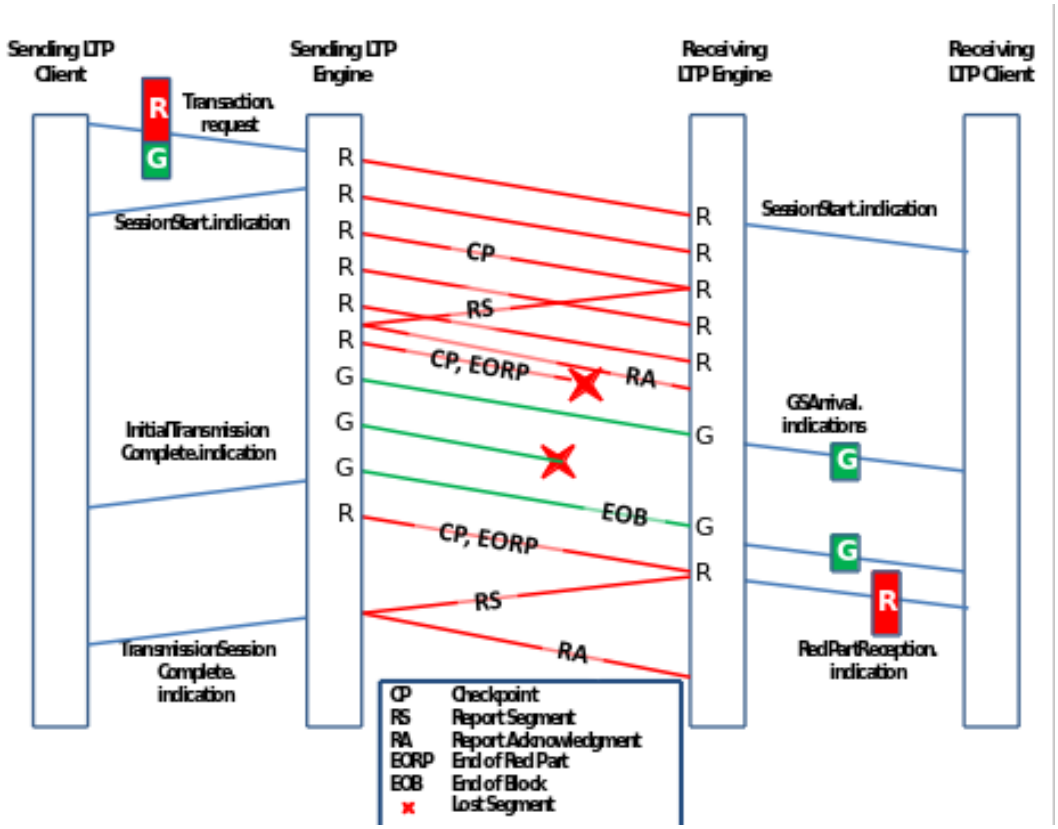

Figure 2: Example of a LTP Session with Red and Green part data

3.4.1 *Link Service Adapter*

As explained before, LTP operates as follows: the LTP "sender" node generates LTP segments from an LTP block and conducts a segment transmission "session" that ultimately enables reconstruction of the block at the LTP "receiver" node. Each segment, which is part of a block, must be sent from the sender to the receiver node and LTP can use different transport protocols to transmit these segments (as we can see in the Figure 1 where LTP is the convergence layer). The interconnection between LTP and the layer below is called *Link Service Adapter* (*LSA*).

As we can see in the Figure 3, an LSA consists of two different parts: the first one is called *LSI*, the second *LSO*. LSI is the input door of the LSA, it is the point where all the segments, sent by the other LTP peer, are received and delivered to the upper LTP protocol. At this point, it reassembles the LTP block and, once the block is completed, the original bundle (or bundles in case of aggregation) is extracted and delivered to the BP layer. The output door of LTP is *LSO*. It extracts LTP segments from the LTP block and packs them into packets of the underlying transport protocol, sending them to the correspondent LTP peer. There are some LSA included into the ION's LTP

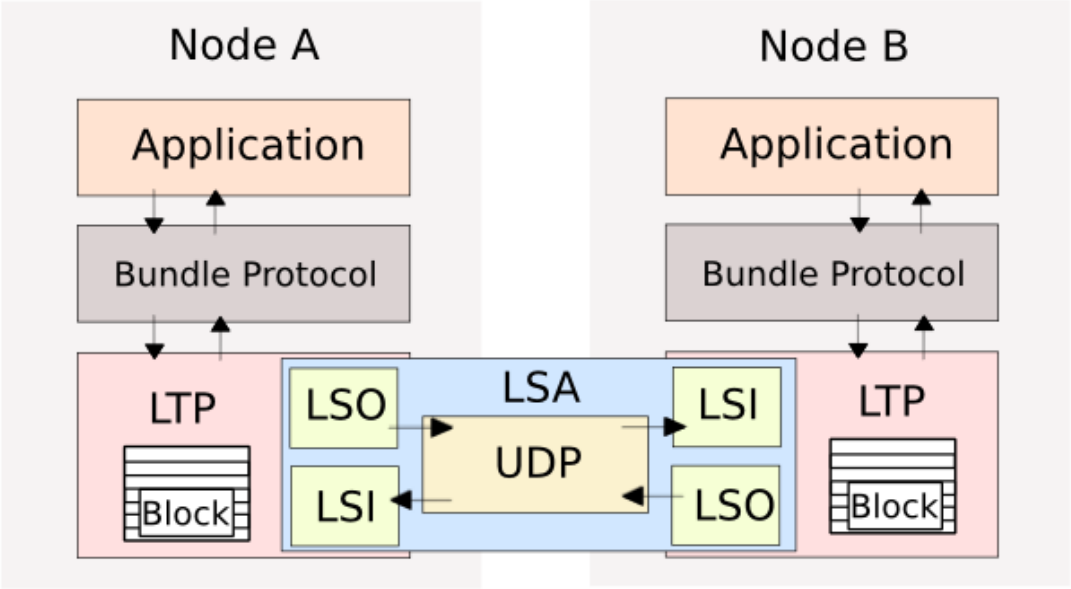

Figure 3: ION and LTP.

implementation (i.e. based on UDP, AOS, DCCP, ...). To include ECs into LTP, I have created a brand new LSA, which is called ECLSA (Chapter 6). It is based on UDPLSA, which is already implemented and included in ION. ECLSA wraps erasure codes and by using them, it offers an increased robustness against losses even if an unreliable protocol, such as UDP, is used at transport layer. As for physical layer FEC, packet erasure coding cannot offer a full protection. If the loss rate is beyond a threshold (about 50% in our case, as it will be shown later), all the losses are recovered by redundancy introduced by the code; otherwise, some residual losses will still be found. As a result, in order to guarantee full reliability, some forms of retransmissions must be enforced. ECLSA is very flexible and provides advantages to both Red and Green transmissions. When reliability is requested, bundles should be encapsulated in the red part. In this case, the advantage of ECLSA is that only residual losses (if any) must be recovered. Vice versa, if full reliability is not requested, bundles can be encapsulated in the Green part. In this other case, the advantage is the greatly reduced loss rate. Note that in the case of unidirectional links, this is the only way to offer a great, although not full, protection against segment losses. Moreover, by adding a CRC to the bundle payload (as done in our experiments carried out with DTNperf_3[19]), it is possible to check bundle integrity when Green transmission is selected, thus allowing the application to discard corrupted bundles (and also to ask their retransmissions, if possible and useful). In the chapter devoted to performance evaluation, ECLSA will be evaluated when applied to either red or block segments.

4

ERASURE CODES

As explained in Chapter 2, normally, physical layer channel coding is employed in space links along with frame validation procedures performed at the data link layer. The data units, successfully processed by this layer, are in turn delivered to the upper layers, where data units erasures can be detected in case of failure of the data link layer frame validation. To cope with these losses, it is possible to use erasure correcting codes implemented at the upper layers. In this case, a *Packet Erasure Channel* (*PEC*) is considered to carry out the code design. In this kind of channel, packets of bits are either correctly received or lost. The implementation of a packet-oriented code at some of the upper layers is not aimed at replacing the physical layer channel coding, but on the contrary, to complement it, so that the two coding schemes can coexist in the same communication system. In this chapter, we will analyse the principles which erasure codes are based on, and explain the basic functionalities to understand the building blocks of the software package I developed, and the way it works.

4.1 MAIN PRINCIPLES

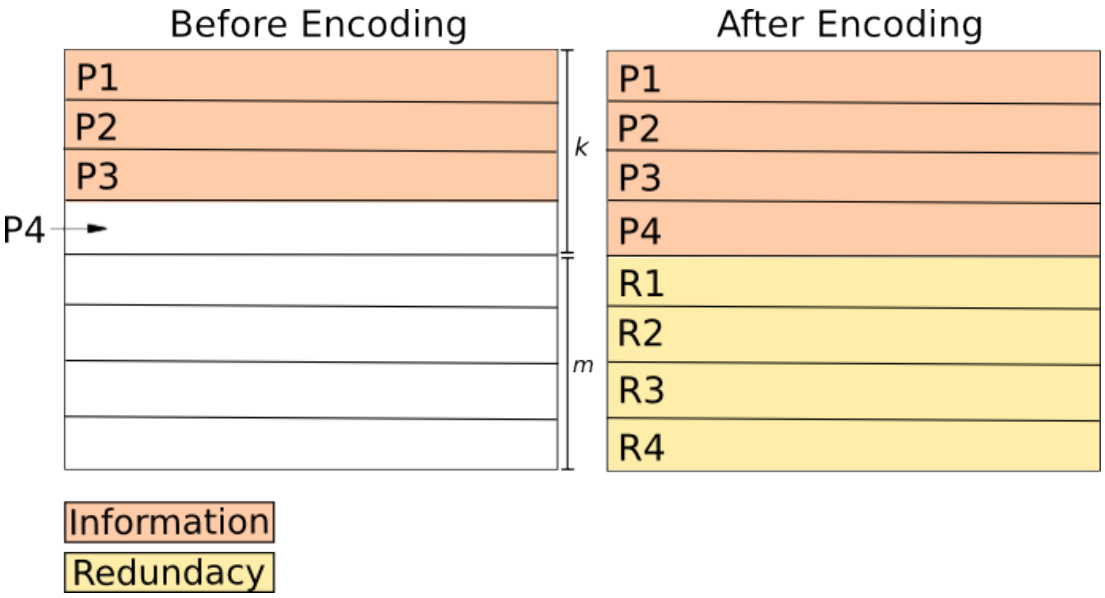

Figure 4: Example of encoding procedure

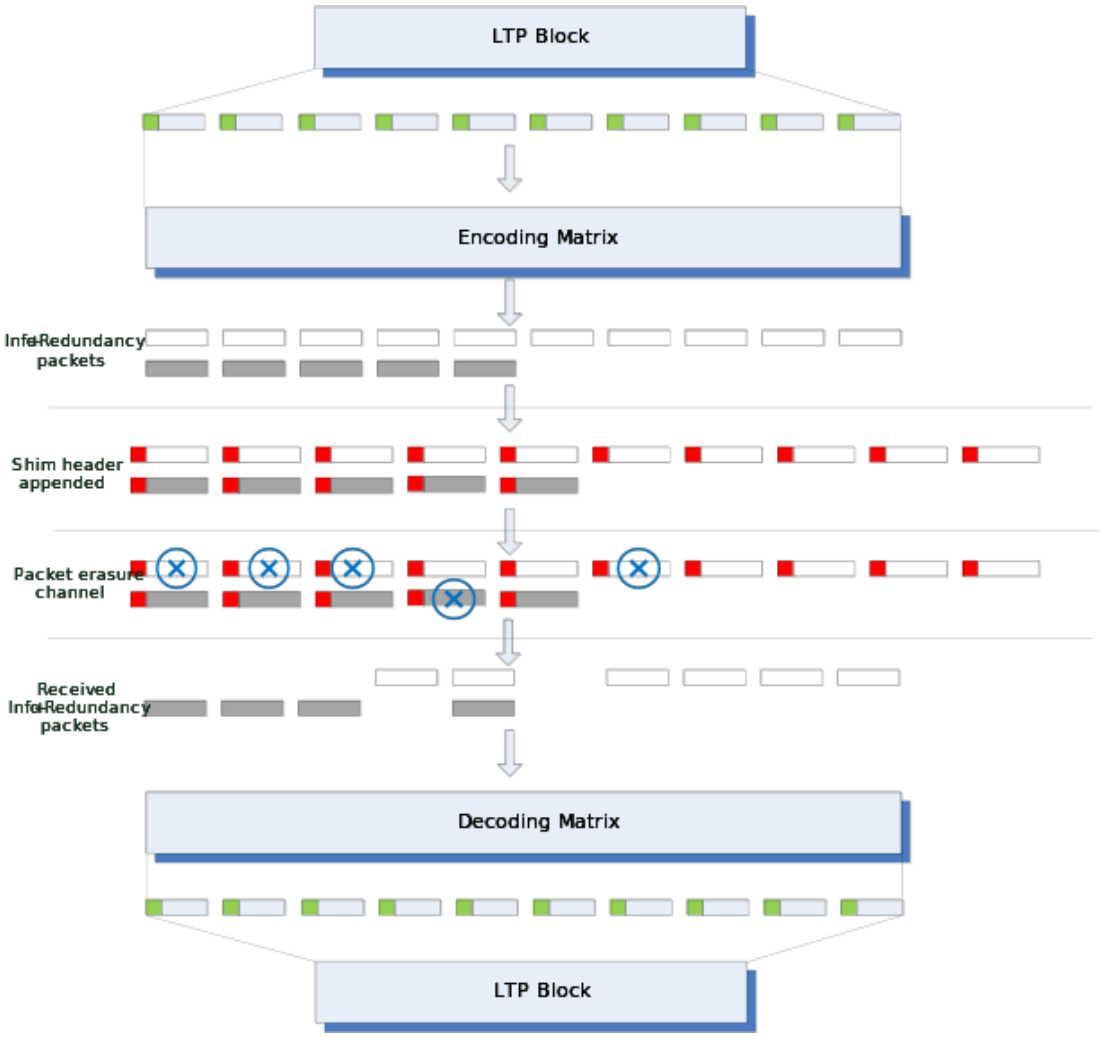

Figure 5: Example of encoding procedure

Assume that we have a set of k fixed-length input segments, each of T bytes length, we want to send to another peer. As we can see in Figure 4 and Figure 5, these k segments are inserted into an "*encoding matrix*", consisting of $n = k + m$ rows each of T bytes. After that, they are encoded to obtain n total segments, each of T bytes length. The encoded segments are composed of the k input segments followed by $m = n - k$ checksum segments. In the same way, on the receiver side, the n encoded segments, are inserted into a "*decoding matrix*" and after the decode procedure, all the original k segments are recovered. The aforementioned k input segments are obtained from a certain number of PDUs belonging to the protocol stack layer in which the erasure code is implemented. These PDUs are known as source packets and they may be of either constant or variable length. Usually, only the payload of a source packet is introduced into the source block, together with a few additional data necessary on the decoder side (i.e. the length of the payload of the source packet). Each source

packet occupies a certain number of rows of the source block, where the last row, associated with a source packet, is completed by padding bits if needed.

However, if erasure codes are used it has to be considered that part of the bandwidth is used for the transmission of the redundancy information, decreasing in this way the bandwidth usable by application data.

4.2 CODES USED

In this study, I have not implemented any erasure codes algorithm, but I have used codes implemented by DLR, previously presented in another work[20]. As explained in the just mentioned paper, the EC used is based on *Low-Density Parity-Check* (*LDPC*) and *Low-Complexity Iterative* (*IT*) decoding. This family of codes behave exactly as explained in Section 4.1 except for the minimum amount of segments that must be received to have a correctly decoded matrix. In fact, theoretically ECs require at least k segments (either information or checksum segments) to successfully decode a matrix, while, in practise, the minimum value of packets required is slightly larger than k. This means that, if in the receiver side k packets are successfully received, the decoding procedure might fail and the lost segments might not be recovered. This is important to well understand experimental results presented in Chapter 7.

5 ERASURE CODE LIBRARIES

5.1 OVERVIEW

The main objective of this work is to allow ION to take advantage of erasure codes, in order to limit, in case of lossy channel, the number of either segment losses in the case of the Green part of the LTP block, or retransmissions in the Red part.

Now that the protocols stack has been explained, it should be clear that LTP is the best position to insert erasure codes. In this protocol there are already segmented data that can be used to fill an encoding matrix. Moreover, LTP can be easily extended, creating a brand new LSA (Section 3.4) that is in charge to create redundancy segments or to recover lost segments.

First of all, I have written some utility libraries to simplify the inclusion of ECs into the ION code. The first library created aims to encapsulate an erasure code needed to encode and decode the various data segments, it is called *Libec*. This library is important not to bind the ECLSA code to only one erasure code implementation. In fact, everything in the ECLSA has been designed to allow the user to easily change the code. To this end, an API-compatible erasure code must be created and eventually the ECLSA must be configured properly.

Considering that the segments sent to the receiver peer require information for the decoding procedure, the original LTP segment is packed into another packet containing additional information. For this reason, I have written another library capable to pack or unpack the original LTP segment with the additional information into an UDP packet. The name of this library is *Libecpackets*. Similarly to the previous library, ELCSA can be used with every transport protocol without modifying the source code of the link service adapter. Libecpackets can be expanded and modified to use not only UDP (like ECLSA does in my implementation) but, for example, TCP or other protocols. In the next chapter (Chapter 6) the new LSA added to the ION implementation will be presented. It uses the two libraries mentioned in this chapter to dispatch the packets to the destination and to recover lost segments. This chapter starts with a little introduction for each library, which explains how the library itself works; than an API list with the most important information for a developer is presented, helping him to use the two libraries.

5.2.1 *General Description*

Libec has been written to encapsulate all the functions and the data structures of the erasure codes used, in order to decouple the ECLSA from a specific erasure code implementation. Its most important functions are the encoder and the decoder. The former generates redundant information (segments) from the LTP segments. The latter regenerates LTP segments lost, using segments received (both information and redundancy segments). All the other functions are essential to operate with the data structures required by the encoder and the decoder.

The most important data structures are *"ADT"* and *"columnStatus"*. The former has two different functionalities. It is seen by the sender as a data structure used to store the segments read from the LTP protocol. It is also seen as the place where the encoder function puts the redundancy information. The receiver, on the other hand, sees ADT as the place where it can put the data received by the UDP socket and, after the decoding procedure, where it can find the original information segments to be passed to the LTP protocol above.

The array *"columnStatus"* is very important during the decoding procedure. In fact, it contains the status of each column of the ADT matrix. Differently from what explained in Chapter 4, from Libec point of view, each segment passed by the LTP protocol is inserted into a new column. By using this array, the decoder knows which information segments are present and which are missing and must be recovered. The array is essentially an indicator for the decoder, which enables it to understand where it has to work.

5.2.2 *Data Structures*

ADT is a multidimensional array (Figure 6), it can be used to store LTP segments for a single encoding/decoding procedure. As explained in Section 4.1, from the erasure code point of view, which works on a matrix of n rows, the encoding procedure can start only when k rows have been filled with the information data (Section 4.1). Otherwise, the decoding procedure requires (at least) more than k rows because of its non-ideal behaviour. When the matrix is ready to be encoded or decoded, it has to be locked as long as the encoding or decoding procedure works on it. As explained in Chapter 6, the ECLSA is a multithreaded architecture and it can concurrency do different operations. For instance, on the sender side, there is one thread, which extracts segments from the LTP client, and one that elaborates the ADT's content to create redundancy information. During this period, the matrix, currently elaborated by the encoder, cannot be used to store other segments, and the first thread would

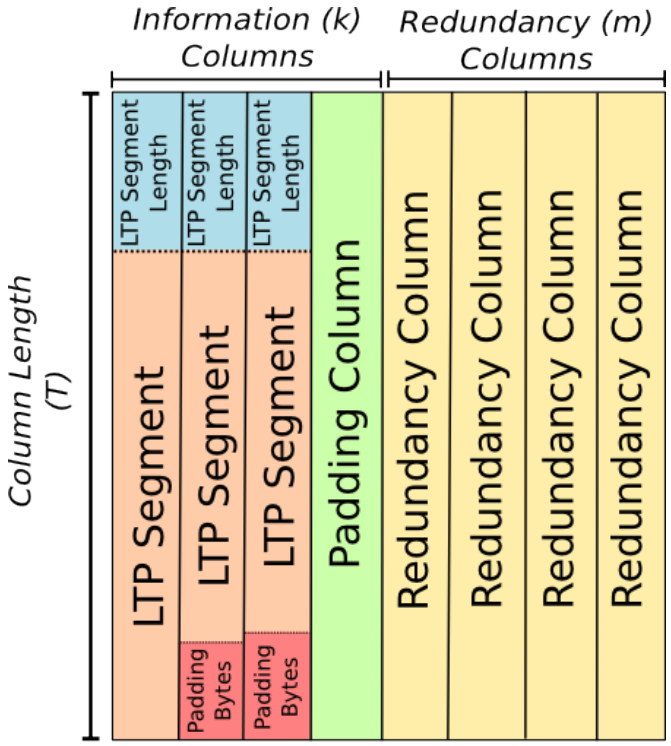

Figure 6: The ADT Matrix

wait until the second one finishes its work. This is a bottleneck for the system performance. The erasure code implementation I have used operates only on one matrix at a time. For the reasons I have explained before in this paragraph, I have projected the ECLSA to work with more than one ADT, implementing a matrix-switching method, based on a new data structure, which aggregates all the ADT-related data:

```
typedef struct
{
 uint32 **ADT;
 uint8 *columnStatus;
 uint8 *rowStatus;
} ec_data;
```

Listing 5.1: ec_data Structure

Two fields of the Listing 5.1 structure have been already explained before, the only one that needs a explanation is *rowStatus*. It is used, during the iteration of the decoding procedure, to mark an equation as resolved.

The variables N, K, M are also important. They indicate as follows:

N: the total length (the number of columns) of the ADT matrix ($N = K + M$),

K: the number of columns containing LTP segments,

Name	Type
ec_data	struct
T	int
N	int
K	int
M	int
ALPHAMAX	int

Table 1: List of libec variables

M: the number of columns containing the redundancy information,

T: indicates the length (in bytes) of a single matrix column. It has to be coherent with other configuration parameters like the LTP segment size. For more information read the Section 5.3.4.

5.2.3 *Column Content and Padding*

A column of the ADT matrix contains not only an LTP segment but also the information to properly interpret the bytes of the column (Figure 6). In fact, a column has a fixed dimension T and during the initialization of the matrix all the T bytes of each column are set to zero. When an LTP segment of X bytes length ($X < T$) is copied into a column, only the X bytes are overwritten, leaving the other ones set to zero. In this way, a padding to the column is automatically inserted if the length of the LTP segment is lower than T. Obviously, the receiver has to know the length of the original LTP segment (X) and, if padding is used, where is the first byte of padding. The original length of the LTP segment has to be recovered in case of a UDP packet is lost during the transmission. For this purpose, the first two bytes of an information column are filled with the actual length of the LTP segment. In this way, not only the LTP segment but also its length is recovered in case of data losses.

5.2.4 *API Function Descriptions*

In Listing 5.2 the most important functions for a developer are listed. In addition to them, Libec contains a large number of procedures used only by the functions listed below, these other functions will not be explained.

```
// Library management
int init_libec(ec_data *ecdata, char *MATRIXNAME1, int Ti,
               int Ni, int Ki, int Mi, int ALPHAMAXi);
```

```
void close_libec(ec_data *ecdata);

// ec_data management
void init_ecdata(ec_data *ec_data);
void free_ecdata(ec_data *ec_data);

// ADT management
void reset_ADT(uint32 **ADTi, int Ni, int Ti);
void add_to_ADT(uint32 **ADTi, uint32 *toAdd, int Kp);
void get_from_ADT(uint32 **ADTi, uint32 *buff, int Kp);
```

Listing 5.2: Libec Functions

The functions presented in Listing 5.2 can be divided into three different groups: *library management*, *ec_data management* and *ADT management*. The first one contains *init_libec* and *close_libec* both related to the startup and the termination of the library. The *init_libec* function is the first function that has to be executed to initialize all the data structures required by the encoder and the decoder. It requires a pointer *ecdata* that, after the execution of the function, will contain references of the first ADT matrix. *MATRIXNAME1* is the name of the file where the function will find a parity check matrix, which is required by the implementation of the erasure codes I have used. *Ti*, *Ni*, *Ki*, *Mi* and *ALPHAMAXi* have been already explained in Section 5.2.2. The *close_libec* function is required to close the library and free the memory used by its data structures. For this reason, it only takes one parameter which is the same *ecdata* pointer used with the previous function *init_libec*.

The second group of functions contains *init_ecdata* and *free_ecdata*. These are related to the data structures explained in Section 5.2.2. Note that the multi-matrices behaviour described in Section 5.2.2 is implemented only in ECLSA, and that all the memory management is done outside the two function described in the library management section. For this reason I implemented this two functions which can be used to allocate or free the memory for a new ec_data structure.

The last group of functions contains *reset_ADT*, *add_to_ADT* and *get_from_ADT* related to the management of a single ADT table. The first function (*reset_ADT*) can be used by the application to reset an ADT matrix when its content is invalid. It takes the pointer to the matrix (*ADTi*) and its dimension, that is to say the number of columns *Ni* and the number of bytes of each column *Ti*. The function *add_to_ADT* inserts an array of length T, called *toAdd*, into the matrix in the position *Kp*. The function *get_from_ADT*, on the contrary, extracts from the matrix the column *Kp* and inserts it into an array *buff*.

The last functions of the library are the encoder (Listing 5.3) and the decoder (Listing 5.4).

```
int universal_encoder(int K, int N, int T, uint8 *
    columnStatus, uint8 *rowStatus, uint32 **ADT, int *
    CNDegree, int maxCNDegree, int **Hc, node *CN, node *VN,
    uint32 **C, int ALPHAMAX);
```

Listing 5.3: The Encoder

Most of the parameters that the encoder requires have been already explained before. The others are automatically loaded by the init_libec function and the programmer has only to write the name of the variables calling the function. The names of the variables are the same reported in the prototype. The return value of the function is very important. In fact, if the encoder fails, it returns a value lower or equal to zero.

```
int decoder_ML(int K, int N, int T, uint8 *columnStatus,
    uint8 *rowStatus, uint32 **ADT, uint32 **ADTRec, int *
    CNDegree, int maxCNDegree, int **Hc, node *CN, node *VN,
    uint32 **C, int ALPHAMAX);
```

Listing 5.4: The Decoder

The parameters and the return value of the decoder are the same as the encoder.

5.3 LIBECPACKETS

5.3.1 *Description*

Libecpackets encapsulates LTP segments into UDP packets. It is useful to not overload the code with the packet assembling and de-assembling functions, but it is also very important to not bind the implementation of the ECLSA to the UDP protocol, used for the transmission of the packets in my implementation. For the same reason, as we will see in the next section, each *ECPacket* (a packet created by using Libecpackets) contains a field with 32 bit of CRC. This field is redundant if the channel uses UDP for the transmission. This protocol has already an optional field with CRC, which can be used on the system. But it is not always true, because not all the transport protocols are able to filter packets with incorrect bit in their PDUs. This is the main reason why we decided to insert a CRC into the ECPacket.

A basic mechanism of extension has been provided into the header to permit future protocol extensions. For instance, it is possible with this mechanism to extend the protocol to change, at runtime, the information about the code used for the transmission (i.e. the erasure code used, the matrix dimension, the parity check matrix used, ...). In my implementation this extension method

is only used for the padding of the ADT matrix, as will be explained in Section 6.2.2.

Another feature of this library is the *circular buffer management*. Some functions and data structures have been inserted to help ECLSI with the management of the input queue of ECPackets.

5.3.2 *Packet description*

An ECPacket transports not only a column of the ADT matrix over the communication channel, but also the information necessary to insert this column in the right position on the receiver's side. The packet is composed by two different parts: the *ECHeader*, which contains the information about the position in the matrix and the *ECPayload*, which is the content of a column of the matrix.

Field Name	**Size (bits)**
bid	32
pid	32
crc	32
ext	8
exts	variable

Table 2: List of the fields of an ECPacket

ECHeader, as we can see in Figure 7, has some standard and fixed fields: *Block ID*, *Packet ID*, *CRC*. *Block ID* (bid in Table 2) is the identifier to select on the receiver's side the right matrix where the segment has to be put. In fact, there are a lot of different matrices of encoding; some of them are active and some of them, instead, have been already decoded. The column index of the matrix where the ECLSO has to insert the segment is noted into the *Packet ID* field (pid in Table 2). The last field is CRC, which, as explained in the previous section, contains the CRC (crc in Table 2) of the payload computed, using the functions explained in Section 5.3.4.

Ext Type Value	**Length (bits)**	**Description**
0x1	32	Padding
0x2	16 *3	K, N, T

Table 3: List of the extensions of the header

ECHeader has also a viable extension method. There are two different fields required to use this feature: *extension type* and *extension value*, in Table 2 the

names are *ext* for the first field and *exts* for the latter. Figure 8 shows the packet with extensions but, as explained later in this paragraph, its extension value has not a fixed dimension. The extension type identifies which extension is used in the current ECPacket, to correctly operate on the packet. This field is also used to know the length of the extension value part of the ECHeader. In fact, for each extension type, there is a fixed buffer dimension thought to contain all the data required to work on the particular packet. There are two extension types already used, these are reported in Table 3.

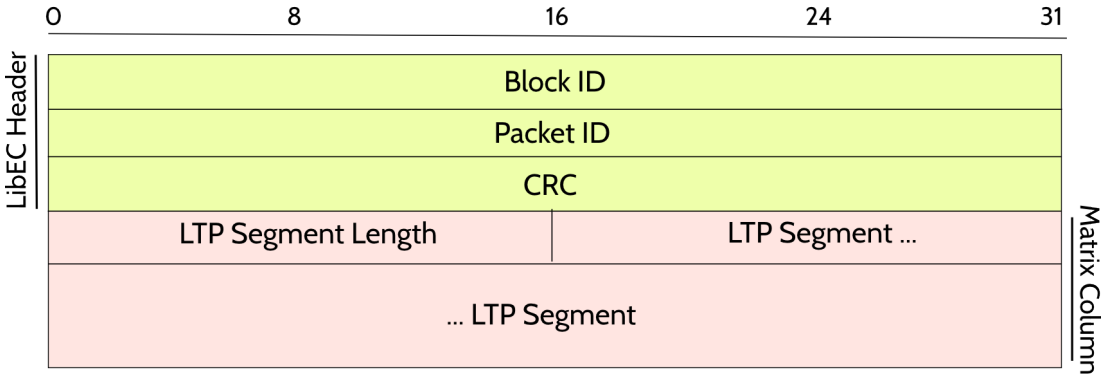

Figure 7: Libec Packet Header.

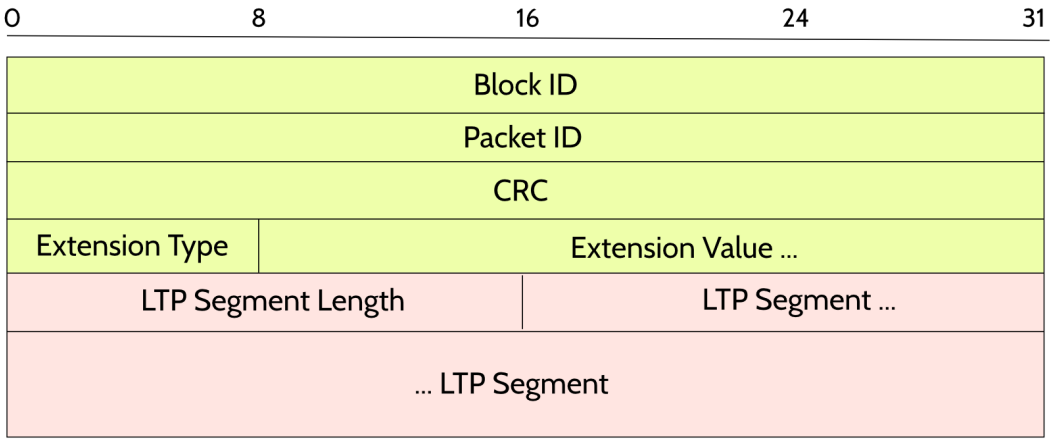

Figure 8: Libec Packet Header with Extensions.

The *ECPayload* part of an ECPacket contains a single column of the encoding/decoding matrix. As explained in Section 5.2.3 and how we can see in Figure 7 and Figure 8, the content of the column and of the ECPayload is not only the LTP segment but also its length.

5.3.3 *Data Structures*

There are two different structures to model the header: *pack_header*, which contains standard and fixed information about the ECPacket and *pack_ext_header*, which is used only for extensions. The two structures are reported in Listing 5.5. The *pack_header* adds to a packet 12 bytes of informations. If an extension is used the header dimension is incremented of 1 byte for the extension type field plus a variable number of bytes for the value of the extension. These values are indicated in Table 3.

```
typedef struct {    // HEADER SIZE 96b=12B
 uint32_t bid;      // THE BLOCK NUMBER
 uint32_t pid;      // COLUMN ID IN THE BLOCK
 uint32_t crc;      // CRC

} pack_header;

typedef struct {
 uint8_t  *exts;
 uint8_t ext;        // EXTENSIONs:
                     // 0x1 -> padding (32b)
                     // 0x2 -> K, N, T (16b*3)

 uint32_t length;    // used locally, not included into the
    header
} pack_ext_header;
```

Listing 5.5: ECHeader structures

ECPayload contains a column of the ADT matrix (both information columns and redundancy columns). The dimension of a single LTP segment can be configured in ION's configuration files. For instance, in my implementation and in my tests, I have fixed a maximum dimension of 1024 bytes for the LTP segment and, for this reason, I have added two other bytes to a column of the ADT matrix to store the actual length of the segment. Accordingly to this dimension, the total length of the ECPayload in my implementation is 1026 bytes. As we can see in Listing 5.6, the column dimension can be easily extended, changing the *DATA_NO* define. To extend the payload length, it is important to change not only this define but also the T parameter discussed above in Section 5.2.

```
#define DATA_NO 1026
typedef struct { // DATA SIZE 1024B + 2B = 1026B
 uint32 data[DATA_NO];
} pack_data;
```

Listing 5.6: ECPayload structure

The last data structure contained in Libecpackets is *circ_buff*, as described in Listing 5.7. As explained before, this structure is used to store received ECPackets that are waiting to be processed. The fields of this structure are the following:

SIZE: the maximum number of elements of the buffer,

START: the position of the first unextracted element,

END: the position of the last queued element,

CNT: the number of elements already enqueued,

ELEMS: an array of pointers with length *size*,

MUTEX: a mutex to synchronize the access to the queue.

```
typedef struct {
 int size;
 int start;
 int end;
 int cnt;
 char **elems;
 pthread_mutex_t mutex;
} circ_buff;
```

Listing 5.7: Circular Buffer Descriptor

5.3.4 *API Function Descriptions*

The first functions, described in Listing 5.8, are used to send or to receive a ECPacket, using a transport protocol (in our implementation, as said before, we used UDP). The most important functions used are the last two in the list. They require a socket (*linkSocket*), a message to send (*msg*) with its length (*length*) and, finally, the last parameter (*parameters*) which can be used to exchange other information with the two functions.

```
void *initChannel(int mode, void *addr, void *parms);
void closeChannel(void *toClose);
int sendSegment(int linkSocket, void *msg, int length, void
   *parameters);
int recvSegment(int linkSocket, void *msg, int length, void
   *parameters);
```

Listing 5.8: Send And Receive Functions

The functions described in Listing 5.9 can be used to create an ECPacket. The most important is the last one (*htonHeaderData*), which can be used to merge all the information given to the function (a pack_header *h*, a pack_ext_header *e* and a pack_data *d*) into an array of bytes (*buffer*). This array can be directly given to the function to send the ECPackets described before.

```
void htonHeader(pack_header h, char *buffer, int bufsize);
void htonExtensions(pack_ext_header *e, char *buffer, int
   bufsize);
void htonData(uint32 *data, char *buffer, int bufsize);
void htonHeaderData(pack_header h, pack_ext_header *e,
   pack_data d, char *buffer, int bufsize);
```

Listing 5.9: Packet Creation Functions

On the other peer of the communication some functions to extracts LTP segments from ECPackets are required. These functions are listed in Listing 5.10. The most important function here is *ntohHeaderData*, which, given a *buffer* received using the function described before, extracts and fills all the data structures required to process the data (a pack_header *h*, a pack_ext_header *e* and a pack_data *d*).

```
void ntohHeader(pack_header *h, char *buffer, int bufsize);
void ntohExtensions(pack_ext_header *e, char *buffer, int
   bufsize);
void ntohData(uint32 *data, char *buffer, int bufsize);
void ntohHeaderData(pack_header *h, pack_ext_header *e,
   pack_data *d, char *buffer, int bufsize);
```

Listing 5.10: LTP Segment Extraction Functions

As we can see in Listing 5.11, the function *ec_hext_add* makes it easier to fill the extension value fields of an ECHeader. It copies *dataLength* bytes of the buffer *data* into the exts field of the pack_ext_header *e*. The extension type contained in the pack_ext_header must have the extension type field already set. Another useful function, if the extension method is used, is *ec_hext_length*, which, using a given extension type, returns the required length in bytes of the extension value field. This information is useful to read the correct number of bytes when extension value is accessed by the application.

```
int ec_hext_add(pack_ext_header *e, char *data, int
   dataLength);
uint32_t ec_hext_length(uint8_t ext_byte);
```

Listing 5.11: Header Extension Function

The functions necessary to use CRC are *calc_crc32_d8* and *calc_crc32_d32*. The prototypes are described in Listing 5.12. The final CRC of the *len* bytes

of the *data* parameter is returned by the function. The initialization value for the CRC is the *crc* parameter. The difference between the two functions concerns simply the type of the input data: in the first function the length of each element of the array *data* is 8 bits, in the last one it is 32 bits.

```
uint32_t calc_crc32_d8(uint32_t crc, uint8_t *data, int len)
    ;
uint32_t calc_crc32_d32(uint32_t crc, uint32_t *data, int
    len);
```

Listing 5.12: CRC Functions

The last functions, presented in Listing 5.13, are necessary to operate with a circular buffer. The function *circ_buff_init* can be used to allocate the memory for the buffer of *circ_buf_len* elements, while the *circ_buff_free* function is used to deallocate it. There are other two functions used to check the status of the circular buffer: one is used to check if it is full (*circ_buff_is_full*) and another to check if it is empty (*circ_buff_is_empty*). Moreover, the function *circ_buff_write* is used to insert an element *buf* of length *buflen* (in bytes) into the circular buffer. Finally, the function called *circ_buff_read* is used to extract elements from the circular buffer. The element extracted by this last function is returned into the *buf* parameter.

```
void circ_buff_init(circ_buff *cb, int circ_buf_len);
void circ_buff_free(circ_buff *cb);
int circ_buff_is_full(circ_buff *cb);
int circ_buff_is_empty(circ_buff *cb);
void circ_buff_write(circ_buff *cb, char *buf, int buflen);
void circ_buff_read(circ_buff *cb, char *buf, int *buflen);
```

Listing 5.13: Circular Buffer Functions

6

ECLSA

6.1 GENERAL DESCRIPTION

As explained in the previous chapters, the best solution, to easily add ECs support in ION, is to create a brand new LSA for the LTP implementation included in ION. As explained in Section 3.4, LTP is interfaced with the underlying layers through various LSA, which are a kind of sub-layers of the LTP protocol. Each LSA uses a different protocol to actually transport the segments through the communication channel. An LSA works on LTP segments. On the sender's side, an LSA extracts LTP segments from a queue in the LTP engine and sends them using the underneath level. On the receiver's side, LSA extracts the original LTP segments from the PDUs received and passes these LTP segments to the LTP engine. These two functions of the LSA are performed by two different entities: one on the sender peer, which is called LSO, and one on the receiver peer, which is called LSI.

The link service adapter that I have created is called "*Erasure Codes Link Service Adapter*" or ECLSA and it uses UDP as transport layer. The dedicated LSO (*ECLSO*) extracts LTP segments and, after the encoding procedure, it creates ECPackets to be sent to the receiver LTP peer via the UDP protocol. The complementary LSI (*ECLSI*) receives ECPackets, inserts them into a decoding matrix and, after the decoding procedure, it passes the LTP segments to the receiver's LTP engine. Both ECLSO and ECLSI may operate on different encoding or decoding matrices for performance purpose (Section 5.2.2). This feature has been implemented after some preliminary experiments carried out with a version of ECLSA, which operated on the matrices one by one. In these experiments, the time spent by the ECLSO to encode the ADT matrix caused a bandwidth decrease. In fact, during this time, ECLSO has to wait until the end of the encoding procedure, without sending anything. For the same reason, the last version of ECLSA has a fixed number (a pool) of ADT matrices, which can be simultaneously used.

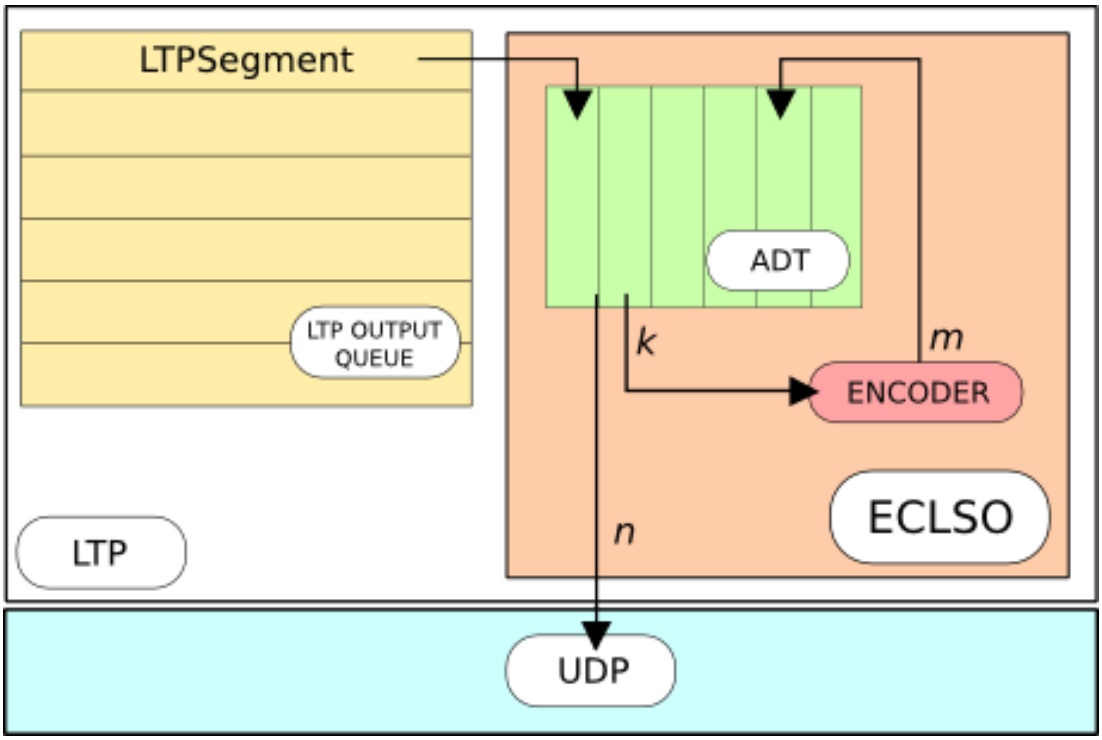

Figure 9: ECLSO General Working Method.

6.2 ECLSO

6.2.1 *General Description*

Before describing the internal code structure, we will discuss the general working method of the ECLSO, which has been partially discussed in other paragraphs. The content of this paragraph is briefly explained in Figure 9.

The operations performed by ECLSO are the following:

1. extracting LTP segments and inserting them into an ADT matrix;
2. starting the encoder and generating redundancy columns of the ADT matrix;
3. creating ECPackets and sending them using a UDP socket.

At the beginning (point 1), ECLSO extracts LTP segments from the LTP queue and inserts them into the ADT matrix, one for each column, filling up to k columns. A timeout is (re)started each time that a new LTP segment is inserted in the matrix. If this timer expires, the ADT is filled with padding columns, as explained in Section 6.2.2. In any case, at this point (point 2), ECLSO starts the encoding procedure when there are k LTP segments into the ADT matrix. The encoder takes these segments and generates m redundancy

columns, which are, in turn, inserted into the ADT matrix. Finally, after that the encoding procedure has finished its work, n (k + m) total columns of the ADT matrix contain data. During the last operation (point 3), each column of the ADT matrix is extracted and is encapsulated into an ECPacket, which is sent using the UDP socket to the destination LTP peer.

6.2.2 *Matrix Padding*

An ADT matrix needs k information columns to generate the redundancy information. Each time a new LTP segment is extracted from the LTP block and inserted into the ADT matrix, a timer starts. If it expires, when for instance, there are f filled columns in the matrix (with f less than k), the remaining r columns (r = k - f) are set to zero and the encoding procedure is started (an example of filled ADT matrix is shown in Figure 6). In this case, where padding is inserted, an extension of the ECPacket header is used to signal to the destination the using of the padding. This is required, on the receiver's side, because the r padding columns are not sent to the destination since they are all set to zero. If the receiver finds the padding extension in the ECHeader, it knows that padding has been used and it fills the padding columns. The extension values of the redundancy ECPackets (between k and (n - 1)) contain the index of the first element not filled (i.e. f because the counter starts from zero). It is possible to not use the padding by setting a *padding threshold* (Section 6.2.7). This threshold is the number of columns, contained into the ADT matrix, that must be passed to force the encoding procedure to start. This means that, if the padding threshold is X, and the number of elements contained in the ADT matrix, when the timer expires, is Y (X>Y), the padding extension is not used in this case and only these Y elements are sent to the destination, without starting any encoding procedure.

6.2.3 *Interaction with LTP*

Normally, the ION's function to extract elements from the LTP block is a locking function. Once called, it returns only when new data are ready to be extracted. To use the padding method explained in the previous section, I added two other functions, based on the standard ones, but with a timer, which, if expired, forces the function to return the control to the caller. The prototypes of this functions are listed in Listing 6.1.

```
int     ltpTimedDequeueOutboundSegment(LtpVspan *vspan, char
   **buf, const struct timespec *timeout);
int     sm_TimedSemTake(sm_SemId i, const struct timespec *
   timeout);
```

Listing 6.1: Functions added to ION

The first function is used in ECLSO, the second one is used in *ltpTimedDequeueOutboundSegment* to avoid a deadlock waiting on the semaphore, used by LTP to notify that there are new data to be read.

6.2.4 *Threads Interaction Diagram*

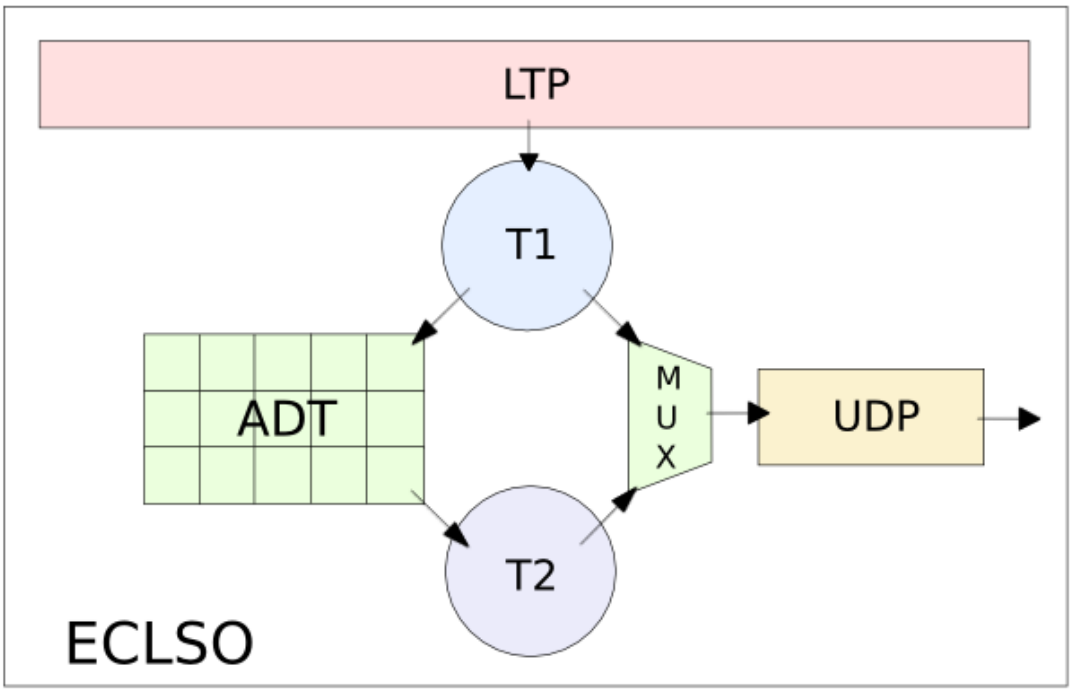

Figure 10: ECLSO Threads.

ECLSO uses a multi-threaded architecture, to improve the performance. In this way, as already explained, more than an ADT matrix can be processed concurrently. There are two different threads in the implementation of ECLSO: T1 and T2. The first one (*T1* in Figure 10) extracts LTP segments from the LTP output queue, and inserts them into one ADT matrix. After, using the LTP columns just inserted, T1 creates new ECPackets, which are immediately sent to the destination LTP peer, to not cause problems with the upper layer (LPT/BP) timeouts. If the number of columns already filled is *k - 1*, T1 notifies the other thread (*T2*) that it must start the encoding procedure.

T2 contains the encoding part of ECLSO. As soon as T1 notifies that T2 must start with the encoding, the second thread executes the encoding function, if it not already working on another ADT. Once finished its task, T2 sends first of all the not already sent information columns of the matrix and, after, the redundancy columns just generated, encapsulating each column sent into an ECPacket. While T2 is encoding the matrix, T1 continues to extract LTP

segments, inserting them into another ADT matrix of the pool and sending, if the socket is not used by T2, ECPackets to the destination. T2 has a queue of jobs to elaborate. This queue is supplied by T1, which generates a new job and inserts it into the queue when k columns of an ADT matrix are filled.

Obviously, only one thread at a time can send ECPackets to the destination node. For this reason, a semaphore has been added to access sequentially to the UDP socket. This alternation leads to a basic interleaving behaviour, which improves the performance of error correcting codes. In fact, in this way, information columns of different ADT matrices are sent to the destination mixed together with redundancy columns, which can be of another ADT matrix. In addition, another interleaver, which sends all the columns of an ADT matrix in a random order, has been implemented and inserted in ECLSO. The use of this additional interleaver leads to poor performance because T1 has to wait that T2 finishes its work before starting to send the content of the ADT matrix. A way to solve this problem has been projected but not implemented. A new output buffer can be inserted in ECLSO, which collects all the UDP segments sent. These segments have to be extracted in a random order from this buffer and not from each ADT matrix, improving, in this way, the performance of the interleaver, but adding a new data structure to the system.

6.2.5 *Data Structures*

The first data structure explained is *eclso_vars* (Listing 6.2). It is used to store the status of a single ADT matrix. The first field is *working_block*, which is the identifier of the matrix itself (*bid* in the ECPacket header). The variable *sent_segments* is a counter to trace the number of ECPackets already sent to the destination. In case of padding (variable *padding* equals to 1), the extension value of ECPackets (only if them contains redundancy informations) header is filled with the index of the last information column, which is the first one before the padded columns. This value is stored into the *last_segment_sent* variable.

```
typedef struct
{
 int              working_block;
 int              sent_segments;
 uint8_t          padding;
 int              last_segment_sent;
} eclso_vars;
```

Listing 6.2: ECLSO status vars

The second structure is *EncoderThreadParms* (Listing 6.3). It contains all the parameters (*linkSocket* and *peerInetName*) required by T2 to successfully send ECPackets to the correspondent LTP peer. If the ECLSO must be closed, T1, using the *running* variable, forces T2 to quit. Finally, UDP congestion can be controlled by setting the rate of UDP segment transmission in ECLSO. This feature uses *sleepSecPerBit* to know how long it has to sleep after each transmission. This variable must be set, as explained in Section 6.2.7, to the value supported by the underlying network.

```
typedef struct {
 int                 linkSocket;
 struct sockaddr_in *peerInetName;
 int                 sleepSecPerBit;
 int                 running;
} EncoderThreadParms;
```

Listing 6.3: Receiver thread parameters

Finally, *enc_job* (Listing 6.4) is one of the element composing the jobs list of T2. This structure contains the BID of the ADT matrix (*wb*) as already explained in the previous chapter. The other variable *i* is the index identifying the specific ADT matrix of the pool of matrices. The value returned by the encoding function is stored in *exe* and the semaphore *enc* is used to lock the modification on the job descriptor. Finally, there is a pointer (*next*) to the next element of the list.

```
typedef struct job
{
 int         wb;
 int         i;
 int         exe;
 sem_t       enc;
 struct job *next;
} enc_job;
```

Listing 6.4: T2 job structure

6.2.6 *Function Descriptions*

The functions used in ECLSO can be divided into four groups:

MISC FUNCTIONS: this group contains functions operating on the variables used inside ECLSO;

POOL MANAGEMENT FUNCTIONS: these are functions for managing the ADT matrices pool;

ECPACKET FUNCTIONS: functions for creating or sending ECPackets, using the Libecpackets library;

THREADS FUNCTIONS: functions executed by the two threads.

Listing 6.5 contains the functions of the first category. The most valuable function in this group is *add_to_list*. It can be used to add to the T2's jobs list a new enc_job for the ADT marked with BID *wb* and identified by the id *i* in the pool.

```
// ** MISC FUNCTIONS **
void reset_eclso_vars(eclso_vars *eclsov, int blockno);
static void shutDownLso();
// ADD A NEW JOB TO THE LIST
enc_job* add_to_list(enc_job *head, int wb, int i);
```

Listing 6.5: List of the misc functions of ECLSO

The second group of functions, listed in Listing 6.6, can be used to manage the pool of ADT matrices. For each function in the group there is a brief description in the comment, which describes its main functionality.

```
// ** ADT MATRICES POOL MANAGEMENT **
// GET THE NUMBER OF FREE MATRIX OF THE POOL
int get_free_matrix_no();
// GET ONE FREE MATRIX FROM THE POOL
int get_free_matrix();
// SEARCH THE MATRIX WHICH CONTAINS DATA FOR A BID
int search_for_bid(int bid);
// GET THE MATRIX WITH THE MINIMUM BID
int get_min(int *arr, int length);
```

Listing 6.6: List of the pool management functions of ECLSO

The code in Listing 6.7 contains functions, which help the creation of ECPackets and send them to the destination. The first function is *createECPacket*, it creates the ECPayload and the ECHeader for the ECPacket. Both these information are packed together into a UDP packet using the functions of the Libecpackets. Finally, the ECPayload is inserted into the ADT matrix indicated by the *toADT* parameter. The last notable function is *redundancy_to_send*, it is used by T2 to send all the redundancy segments, created by the encoding procedure.

```
// ** ECPACKET CREATION AND DELIVERY **
// CREATE A NEW ECPACKET
int createECPacket(char *outbuf, char *segment, int
    segmentLength, char *toADT, int bid, int pid, uint8_t
    ext_byte, char *data, int dataLength);
```

```
// SEND AN ECPACKET TO THE DESTINATION LTP PEER
int sendECPacket(char *buf, int segmentLength, int socket,
   struct sockaddr_in *peerInetName, float sleepSecPerBit);
// SEND REDUNDANCY
int redundancy_to_send(enc_job *next_job,
   ReceiverThreadParms *rtp);
```

Listing 6.7: List of the ECPackets management functions of ECLSO

The last list of functions is Listing 6.8. It contains the *main* function executed by T1. It works extracting an LTP segment from the LTP output buffer and generating an ECPacket, sent through the UDP socket. In addition, the related ECPayload of the ECPacket is inserted into an ADT matrix. When the matrix, currently used by T1, is full, the thread notifies to T2 that it can start with the encoding procedure.

T2 main function is *encode_matrix*. This function takes the ADT matrix with lower BID and encodes it. When the encoding procedure finishes, the redundancy segments are sent to the destination LTP peer.

```
// ** THREADS FUNCTIONS **
// MAIN THREAD (T1)
int main(int argc, char *argv[]);
void close_mainthread(ReceiverThreadParms *rtp, pthread_t *
   receiverThread);
// ENCODER THREAD (T2)
static void *encode_matrix(void *parm);
```

Listing 6.8: List of the threads functions of ECLSO

6.2.7 *Invocation Method*

ELCSO must be configured, like any others LSO, in the *ltpadmin* part of ION's configuration files, using the command "*a span*". The command parameters are:

```
eclso IP:PORT
      N K M T ALPHAMAX
      PADDING\_THRESHOLD
      TXBPS
      INTERLEAVER
```

Listing 6.9: ECLSO invocation

IP:PORT is the IP address and the port of the correspondent LTP peer, where an ECLSI must be previously started, as explained in Section 6.3.6. *N*, *K*, *M*, *T* and *ALPHAMAX* are the EC parameters, as explained in Chapter 5.

PADDING_THRESHOLD has been explained in Section 6.2.2. *TXBPS* is used to calculate the *sleepSecPerBit*, which have been explained in Section 6.2.5. Finally, the *INTERLEAVER* parameter is the seed used by the interleaver (as explained in Section 6.2.4) to randomize the extracting order of the ECPackets from an ADT matrix. If it is set to "-1", the interleaver will be disabled.

6.3 ECLSI

6.3.1 *General description*

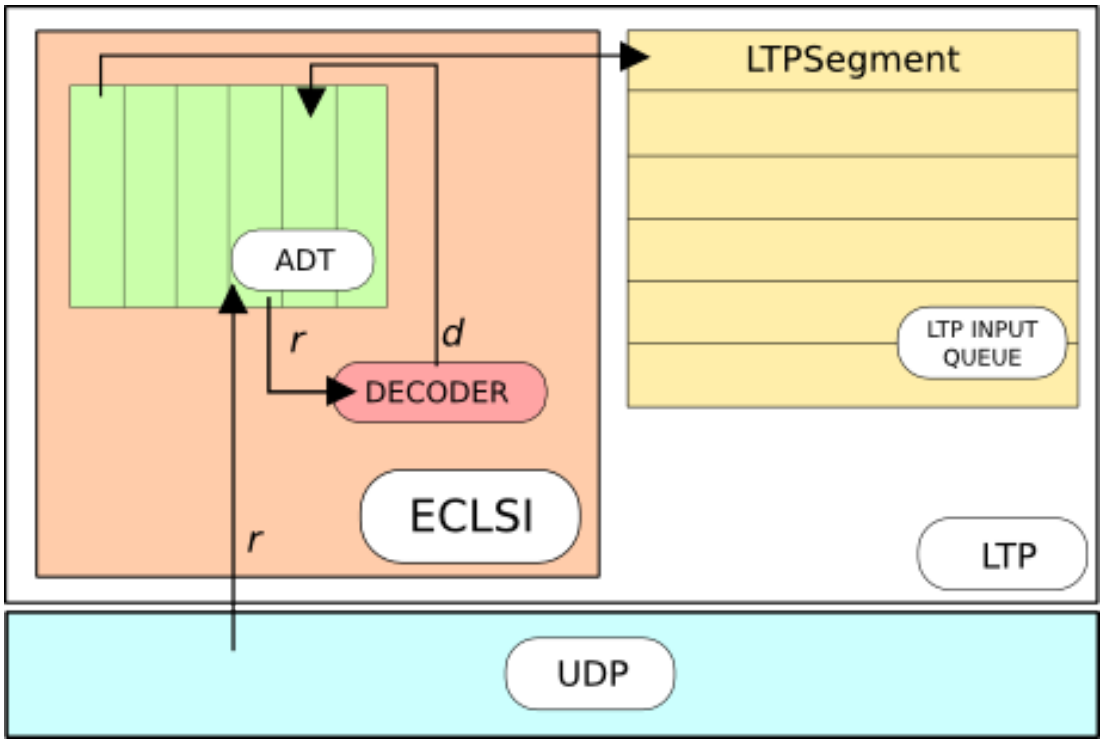

Figure 11: ECLSI General Working Method.

The working method of the ECLSI, partially discussed above, is explained in Figure 11. The basic operations are the following:

1. receiving ECPackets from the UDP socket;
2. calculating the CRC of a received ECPacket and comparing it with the one contained in the ECHeader:
   - if CRC is correct, it inserts the ECPayload into the ADT matrix and continues from 3;
   - else it discards the whole ECPacket and restarts from 1;
3. starting the decoder to recover lost columns;

4. forwarding all the valid LTP segments (both received and recovered) to the LTP engine.

As shown in Figure 11, r ADT's columns are extracted from r received ECPackets. When there are no empty columns in the ADT matrix (or if a timeout occurs as explained in Section 6.3.2), ECLSI starts the decoder, which may recover d lost columns of the matrix and the LTP segments contained in them. If a redundancy ECPacket with the padding extension is received, padding columns are inserted in the matrix. For instance, if the extension value of the ECPacket received with padding extension type is *X*, the first padding column is the *(X+1)*-th, the last one is the *(k-1)*-th column. If the ECPackets are received in order and there are no packets lost, the corresponding LTP segments are immediately passed to the LTP layer. If ECPackets are received out of order, they are reordered while they are inserted in the ADT matrix and, as soon as a sequence of segments has been reordered, ECLSI passes the segments to the LTP layer. If the decoding procedure fails, only the LTP segments contained in the received information columns are passed to the LTP protocol.

6.3.2 *Forcing the Decoding Procedure*

The decoding procedure of an ADT matrix starts when all its columns are filled. If this were the only way to start the decoding function, in case of missing ECPackets, the ECLSI would wait forever for new data. To overcome this problem, a timer can be set up. When this timer expires, the empty columns are considered as missing ECPackets and the decoding procedure is started. The value of the timer can be configured in the ION configuration file as explained in Section 6.3.6.

There is also a security mechanism to avoid packet losses, caused by both the limited number of ADT matrix of the pool, and the limited dimension of the circular buffer. If the number of free matrices passes a threshold *FORCE_DECODE_MATRIX_NO*, the decoding of the ADT matrix with the lower BID is forced, as it will be explained in Section 6.3.4.

6.3.3 *Threads Interaction Diagram*

ECLSI has three different threads cooperating to model the behaviour presented. *T1* is the receiving thread that receives the ECPackets from the UDP socket, and stores them into the circular buffer described in Section 5.3. This intermediate step is required to operate as quick as possible to not loose any UDP packet on the input channel.

T2 extracts ECPackets from the circular buffer and dispatch the corresponding ECPayloads to the appropriate ADT matrix of the pool. T2 is in charge of

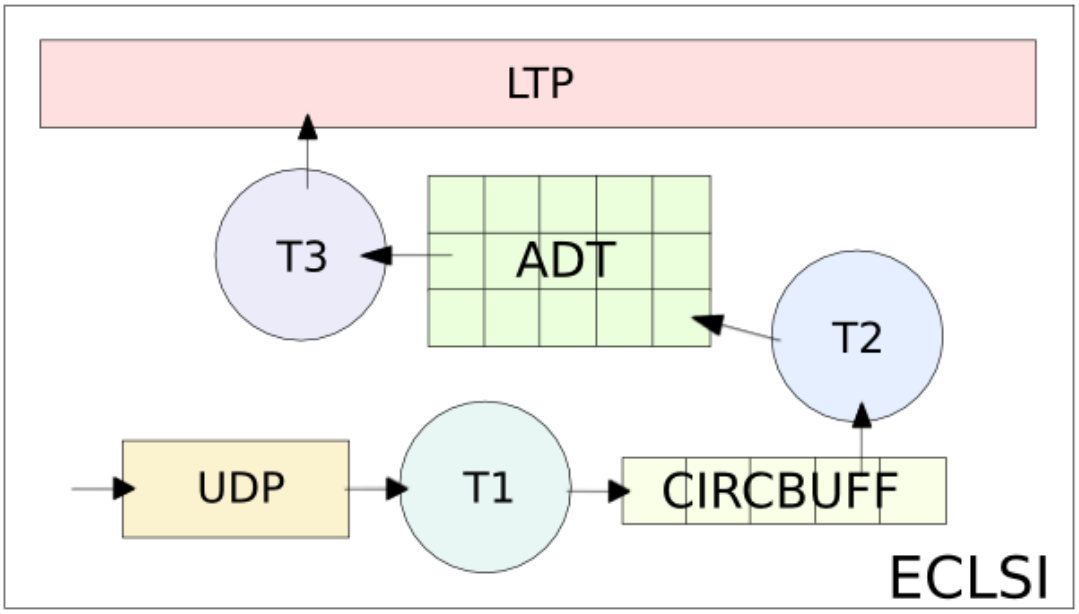

Figure 12: ECLSI Working Method.

the correct switching of the packets, finding for each packet the corresponding ADT matrix. If ECPackets are received without "gaps", the extracted LTP segments are immediately passed to the LTP engine, to not cause problems with the upper layer timeouts. In case of out of order or missing ECPackets, as soon as the original sequence of the packets is reconstructed, they are passed to the upper layer for the same reasons explained before.

When the ADT matrix is full, or when it is forced to be closed, *T3* starts to decode the matrix, forwarding the remaining LTP segments to the LTP engine, after finishing the decoding procedure. At the end the ADT matrix of the pool is freed and it can be used for packets with different BIDs.

6.3.4 *Data Structures*

The data structures used in ECLSI are *eclsi_vars* and *ReceiverThreadParms*. The first structure contains all the parameters of a single ADT matrix of the pool. The second one is used to pass information to T2 and T3.

The structure *eclsi_vars*, which is described in Listing 6.10, is a descriptor of a single ADT matrix of the pool. A structure of this type contains all the variables used to store the status of a single ADT matrix of the pool. When an ECPacket with a new BID is received, a free ADT matrix is taken from the pool with the corresponding descriptor of type eclsi_vars. The variables contained in the structure can be divided into four groups.

The first group (called ADT status) contains *exp_bid* and *exp_pid*, which are the BID of the matrix and the expected PID (column index) of the next packet. The counter *wrong_columns* contains the number of missing columns of the matrix. The variable *missing_k* is the number of missing columns with index lower than *k*. On the contrary, *added_segments* counts the elements already added to the matrix. In the same group there is also the variable *last_k_received*, containing the index of the last received column with PID lower than *k*. The next segment that has to be passed to the LTP engine is *next_segment_to_send*. If the padding extension is used, not all the *k* columns of the ADT matrix has to be forwarded to the LTP engine, for this reason, the variable *last_segment_to_send* contains the last column before the padding columns.

The second group of variables stores the status of the decoding thread. The variable *flush* is used to force the decoding of the matrix when required (Section 6.3.2). The return value of the decoding function is stored into the *decoded* variable.

The third group contains the timeout method, which is used while inserting ECPayloads into the ADT matrix (as explained in Section 6.3.2), using the variable *ts_to*.

Finally, the last group contains a semaphore and a mutex. The semaphore *received* is used to signal to the decoding thread if a new ECPayload is inserted into the ADT matrix with lower BID. In this way, the thread restarts the timeout previously discussed. The mutex (*mutex*) is used to serialize the access to the data structure.

```
typedef struct
{

 // ADT STATUS
 int              exp_bid;
 int              exp_pid;
 int              wrong_columns;
 int              added_segments;
 int              missing_k;
 uint32_t         last_k_received;
 int              next_segment_to_send;
 uint32_t         last_segment_to_send;

 // DECODER STATUS
 uint32_t         flush;
 int              decoded;

 // TIMEOUT
 struct timespec ts_to;
```

```
 // SYNCH MUTEX
 sem_t            received;
 pthread_mutex_t  mutex;

} eclsi_vars;
```

Listing 6.10: eclsi_vars Structure

The second data structure is *ReceiverThreadParms* and it contains four different fields. The first, which is called *linkSocket*, is the socket used to receive the UDP segments. If the main thread wants to force the other threads to quit, the variable *running* is set to zero. Finally, in *to_sec* and in *to_usec*, the timeout value of a matrix, which is read as ECLSI parameter, is stored.

```
typedef struct
{

 int     linkSocket;
 int     running;
 int     to_sec;
 int     to_usec;

} ReceiverThreadParms;
```

Listing 6.11: ReceiverThreadParms Structure

6.3.5 *Function Descriptions*

The functions of ECLSI are divided into four groups, as we can see in Listing 6.12.

The first group contains only *reset_eclsi_vars*, which can be used to reset an *eclsi_vars* data structure.

The "*ADT Functions*" group contains all the functions required to operate with the ADT matrix pool. Their prototypes clearly indicate their functions. The return value of each function is an index of the ADT matrix in the pool.

The third group contains the function *sendSegmentsToLTP*, which forwards the LTP segments from the *start*-th column to the *limit*-th column of the matrix with index *matrix_index*. The function *add_padding* inserts padding columns into the *wmi*-th ADT matrix, starting from the column with index *start*. The Libec decoder is wrapped by the *check_and_decode* function, which checks, before starting the decoder, if there are any missing columns among the first k columns of the ADT matrix. Finally, the *main* function is used to initialize all the data structures and to start the three threads of the application.

The last group contains three functions, one for each thread. T1 executes the *addSegmentsToADT* function, which extracts LTP segments from ECPackets

and inserts them into the correct ADT matrix. The function *handleDatagram* receives UDP packets from the socket, and inserts them into the circular buffer. The third thread executes the *dispatchSegments*, which waits for the reception of the last column of the ADT table or for the expiring of the timer. At this point, the function decodes the matrix (if there are any missing columns with index lower than *k*) and sends the not already sent LTP segments to the LTP engine.

```
// ** MISC FUNCTIONS **
void reset_eclsi_vars(eclsi_vars *eclsi_v, int exp_bid, int
   already_locked);

// ** ADT FUNCTIONS **
int search_for_bid(int bid);
int get_free_matrix_no();
int get_min_bid();
int get_free_matrix();

// ** UTILITY FUNCTIONS **
void sendSegmentsToLTP(int start, int limit, int
   matrix_index);
void add_padding(int start, int wmi);
int check_and_decode(int timedout);
int main(int argc, char *argv[]);

// ** THREAD FUNCTIONS **
void addSegmentsToADT(int to_sec, int to_usec); // T1
static void *handleDatagrams(void *parm); // T2
static void *dispatchSegments(void *parm); // T3
```

Listing 6.12: ECLSI Functions

6.3.6 *Invocation Method*

ELCSI must be configured, like any others LSI, in the *ltpadmin* part of ION's configuration files, using the command "*s*". The parameters of the command are the following:

```
eclsi IP:PORT
      N K M T ALPHAMAX
      ADT_TIMEOUT
```

Listing 6.13: ECLSI Invocation

The only parameter which differs from the previously discussed ECLSO (Section 6.2.7) is *ADT_TIMEOUT*. It is the parameter used to set up the timeout explained in Section 6.3.2.

7 PERFORMANCE ANALYSIS

7.1 GENERAL DESCRIPTION

As explained in Section 3.4, LTP is able to overcome some of the typical problems of the space environment. Despite these improvements, the protocol may still have problems in both its two communication methods: Green and Red.

In the former case, erasure codes add robustness against losses, which, as well as being often useful, it is the only possible countermeasure against losses in mono-directional channels, where *ARQ* methods are not usable.

In the latter case, although the ARQ mechanism associated with Red data provides the user with full reliability, this is not for free, especially in the case of very high error rates and long RTTs typical of space links; in fact, in such conditions multiple retransmission can lead to unaceptable long delivery times. Erasure codes can greatly improve the performance, by limiting the number of retransmissions cycles to a minimum (often zero), thus decreasing the delivery time.

To quantitatively assess these benefits, we have planned a series of experiments, with both Green and Red data. However, while we manage to complete the tests of Green data, during the subsequent tests with Red data, we have found an unexpected behaviour in LTP retransmission in the presence of high PER, which causes the delivery of a number of copies of the same bundle or that prevents a bundle to be correctly delivered. The nature of this problem is independent of the ECLSA. In fact, we found it with ECLSA disabled, when we wanted to evaluate performance of LTP alone using UDPLSO, as a benchmark for subsequent ECLSA tests. The problem is related to the LTP implementation included in ION which does not work well with high PER and big bundles. The cause of the problem will be explained in Section 7.7. This problem, although independent, has an impact on performance and on the accuracy of the tests also when ECLSA is enabled, in particular when residual losses (i.e. losses not recovered by ECLSA) trigger LTP retransmissions. This is a not frequent case, but, in order not to present results whose reliability may be somewhat questionable, we have preferred to skip results with Red data, although they were very promising. The results presented here will therefore refer only to the case of Green data; even with this limitations, they will be enough to prove the important benefits introduced by ECLSA. It is our commitment to integrate this thesis with an Appendix presenting results for Red data as soon as the problem of LTP retransmission is fixed.

7.2 SOFTWARE USED

The entire testbed used in our experiments has been created and managed by means of *VirtualBricks*[21] (*VB*). VB is a GUI software released under the GNU GPLv2 and included in the official Debian distribution (actually, I am on of the main developer of the tool). VB aims is to create and manage not just single virtual machines, but network testbeds consisting of multiple virtual machines interconnected by means of virtual networks components (i.e. switches, cables, channel emulators, ...).

Virtual Machines (*VMs*) can be based either on *QEMU*[22] or on *Kernel based Virtual Machines*[23] (*KVM*), the virtual network devices are part of the *Virtual Distributed Network*[24] (*VDE*) toolset.

In out testbed we will also use, as channel emulator, *vde-netemu*, a variant of the *vde-wirefilter* specifically developed for our tests. It can be used to add delays, bandwidth restriction, loss rate (PER). It is also able to create a two states (good, bad) Markov chain to model correlated channels. The two parameters used to set the Markov chain are *loss* and *lostburst* (Table 4). The probability to exit from the bad state is described by the Equation 1, the probability to enter the faulty state is described by Equation 2.

$$P_o = \frac{1}{lostburst} \tag{1}$$

$$P_i = \frac{loss}{lostburst - (1 - loss)} \tag{2}$$

On the VMs, we installed the *Debian 7 GNU/Linux* distribution, with *Linux Kernel 3.2.0*. The BP implementations used are *ION 3.1.2* and, later, *ION 3.2.0*. The application used to send bundles between the two nodes is *DTNPerf_3*[19]. This is a software tool for DTN performance evaluation developed at the *University of Bologna* under the supervision of Prof. *Carlo Caini*. I contributed to it by adding a CRC capability to check if bundles are correctly received a feature that was essential in our experiments. In fact, the LTP CL fills with zeros the gaps corresponding to the LTP segments lost. Thus, bundles extracted from an LTP block may differ from the original ones in case of loss. This behaviour is called "*zeros-filling*".

7.3 SCENARIOS AND TESTBED CONFIGURATION

The scenarios used for my experiments is characterized by two nodes, one on the Earth and one on the Moon, communicating over an optical channel (numeric parameters will be listed later). Two types of channel models have

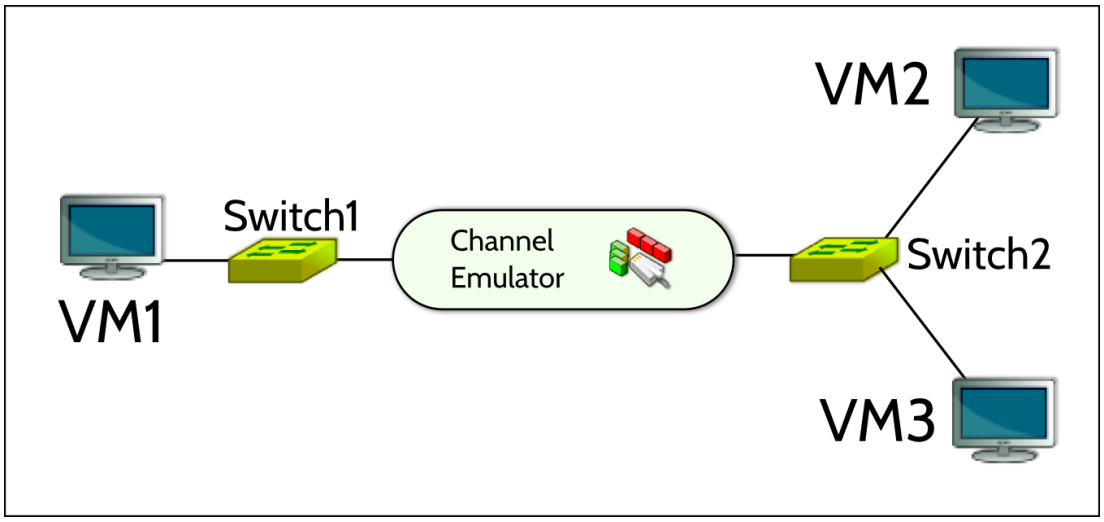

Figure 13: The Virtual Testbed Used.

been used for the experiments: an uncorrelated channel and a correlated channel. As explained in Section 7.2, the used channel emulator is able to correctly receive UDP segments or to lost them, in practice, a UDP segment cannot be received with incorrect bits in it. For this reason, *Packet Error Rate* (*PER*) will be used to model the channel instead of the usual Bit Error Rate.

As we can see in Figure 13, the testbed used to simulate the scenario uses three virtual machines. VM1 is the sender, VM2 the receiver and VM3 the monitor, i.e. an additional node with monitoring task only. It receives some informative bundles, called *Status Reports* (*SRs*) sent by the BP of the nodes. They are used to report the time when an application bundle (a data bundle sent by the user) is processed (i.e. forwarded by the sender to the receiver, received by the receiver, delivered by BP to the final application). These SRs, which are a characteristic features of the BP, allow the user to carry out a micro-analysis (i.e. bundle by bundle) of the traffic, which is essential to discover possible anomalies.

The two switches *Switch1* and *Switch2* are required by VDE because, if two components of the toolset have to be interconnected, they uses data sent through the switch, which is then fundamental.

For experiments in Section 7.4, the channel emulator has been used with the parameters described in Table 4. The parameters for the configuration of the real scenarios experiments are explained in Section 7.5.

The test campaign for Green data was planned and carried out in the following logical order:

- Preliminary tests:
  1. uncorrelated channel with UDPLSA (as benchmark data and to validate the channel emulator)

Parameter	Value	Description
delay	1500ms	Propagation time of the channel
bandwidth	20Mbps	Bandwidth restriction
loss	variable	Packet Error Rate
lostburst	variable	Losses burst length

Table 4: VDE-Netemu parameters

2. uncorrelated channel with ECLSA (as benchmark data and to validate the erasure coding/decoding)
3. correlated channel with UDPLSA (basically the same as 1)
4. correlated channel with ECLSA (to assess the impact of correlation on erasure code performance, which is an essential aspect, given the expected high burstiness of real channel)

- Application scenarios (4 scenarios, with same RTT but different correlation and PER, UDPLSA vs. ECLSA)

7.4 PRELIMINARY TESTS

7.4.1 *Uncorrelated Channel and UDPLSA*

Some preliminary tests have been carried out to prove the validity of the channel emulator and to show that the bundle dimension has an important impact on the relation between the packet error rate and the bundle error rate in the case of Green data with ECLSA disabled. The benchmark parameter considered in these tests is the *Bundle Error Rate* (*BndER*). We uses the values described mathematically by Equation 3, comparing them with the experimental values calculated using Equation 4.

$$BndER_p = 1 - (1 - p)^{Bundle_{length}} \tag{3}$$

$$BndER = \frac{Bundles_s - (Bundles_l + Bundles_w)}{Bundles_s} \tag{4}$$

In Equation 3, the parameter $Bundle_{length}$ is the number of LTP segments composing the bundle. The other values, calculated using Equation 4, uses the $Bundle_s$ that is the number of bundles sent, $Bundle_l$ that is the number of bundles lost and $Bundle_w$ that is the number of bundles wrong. A bundle is

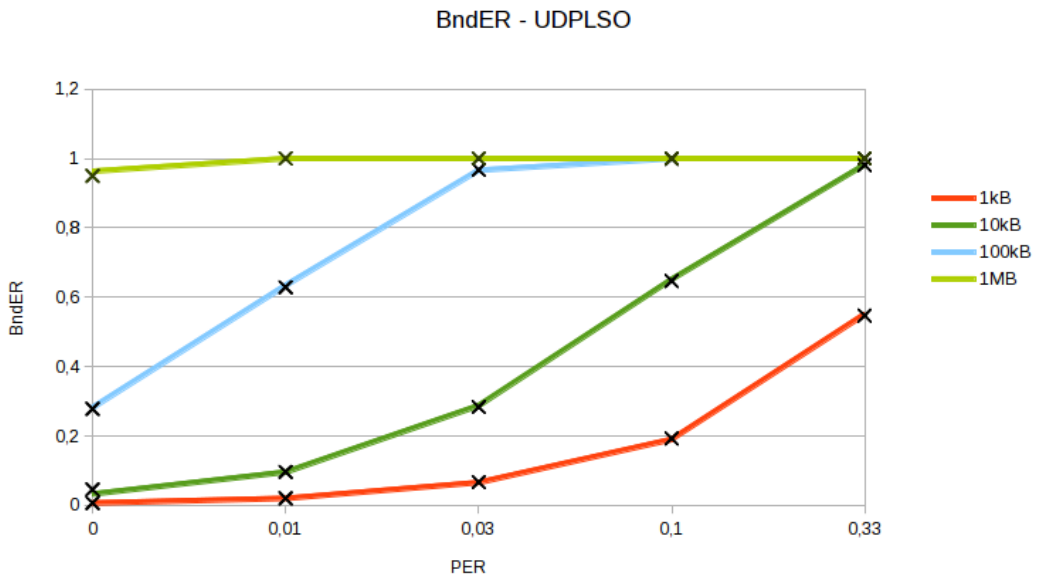

Figure 14: UDPLSA and Green: Bundle Error Rates

wrong when there is at least one missing segment, because of the zeros-filling explained before. The reason why a bundle is *lost* is because the LTP engine, to extract the bundles, waits for the last segment of the block but, if the last segment is lost, no bundles are extracted from it.

The results of these tests, obtained with an uncorrelated channel, are shown in Figure 14, where the Bundle Error Rate (BndER) vs. the PER for various bundle sizes considered (1kB, 10kB, 100kB, 1MB) are plotted. In the figure theoretical results achieved by means of Equation 3 describe the curves. Crosses on the curves identified the values obtained with experimental results. As we can see the two results (theoretical and experimental) are very similar. Besides the obvious result that the BndER increases with PER, we observe that for all the bundle sizes considered, except the first one (1kB), the BndER is very close to 1, if the PER is at the highest value considered, i.e. 0.33. The lower tolerance to loss of large bundles is due to the fact that it is enough to have one loss in a bundle to have a bundle received different from the original one, and this probability increases with the bundle length.

For a better insight, in Figure 15, the different contributes to the BndER are given, namely "wrong" and "lost" bundles, with reference to 1MB bundles. We can observe that wrong bundles are higher than lost for all PER considered but the highest (0.33).

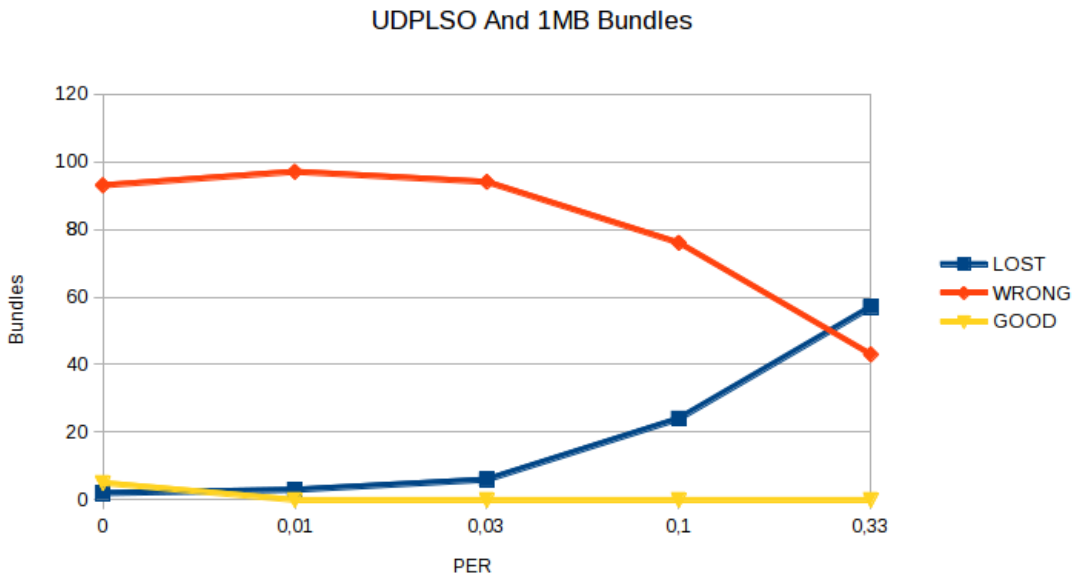

Figure 15: Bundles Errors Using UDPLSO and Bundles of 1MB

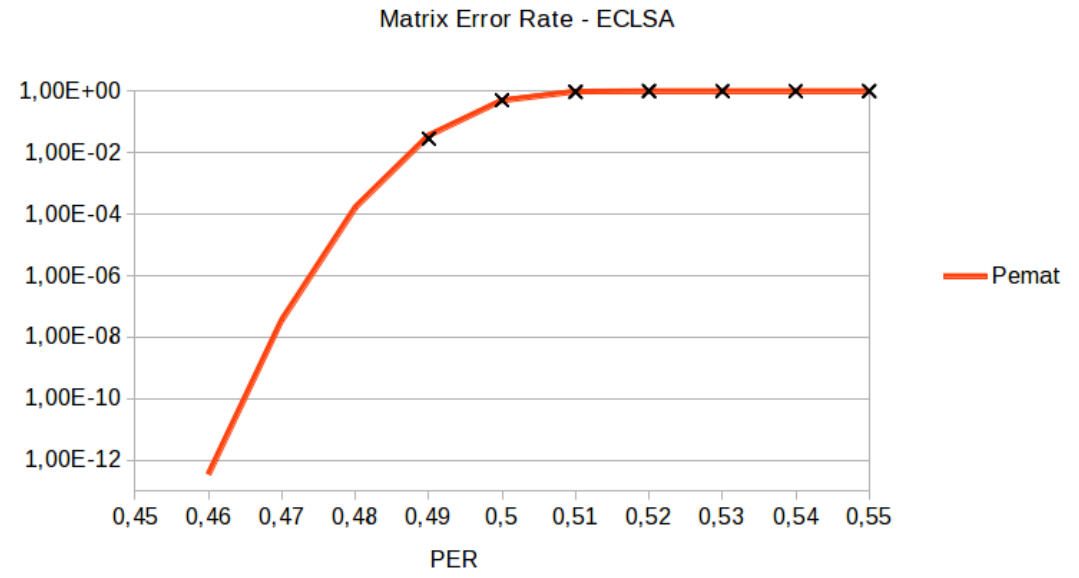

Figure 16: MER Chart

7.4.2 *Uncorrelated Channel and ECLSA*

A second series of tests have been carried out to explore the differences between the implementation of the erasure code included in ECLSA and an ideal case. Erasure codes, characterized by the parameters n, k and m, as explained in Chapter 4, require at least k (in case of ideal behaviour) columns to correctly decode a matrix. The probability of having a fail, during the decoding procedure of the matrix, can be mathematically described by the binomial distribution:

$$b(k, n, p) = 1 - \binom{n}{k} p^k (1-p)^{n-k} \tag{5}$$

In Equation 5, k is the number of success required in a sequence of n total experiments and p the probability of success of a single experiment. The experiments in this section check if actually the erasure codes that I have used have the behaviour described by the Equation 5. Moreover, these experiments have been very useful to understand where the EC implementation starts not to recover lost information, giving some reference points for the experiments with the correlated channel. The value calculated is the *Matrix Error Rate* (*MER*), described by Equation 6.

$$MER = \frac{Matrices_e}{Matrices_s} \tag{6}$$

In Equation 6, $Matrices_e$ is the number of EC decoding failures; $Matrices_s$ is the total number of matrices encoded.

These experimental results aim to validate the correctness of the EC implementation used in ECLSA. Experimental and theoretical results are compared to this end.

The results of these experiments are shown in Table 5 and Figure 16. In the data reported, the values obtained with the Equation 5 (the theoretical results) are called *Pemat*. The crosses on the curves represent values obtained with experimental results and computed by using Equation 6. It is clear reading the numeric result of the Table 5 that the values obtained in the two different ways are very similar, demonstrating that the used EC implementation works as expected. The results in Table 5 demonstrates that all the matrices, thanks to the used EC implementation, are correctly recovered if on the channel a PER up to 0.46 is present.

PER	Pemat	Pecalc
0,45	0	0
0,46	3,5283E-013	0
0,47	3,6677E-008	0
0,48	0,0001	0
0,49	0,0359	0,0283
0,5	0,4955	0,4964
0,51	0,9623	0,9362
0,52	0,9998	1
0,53	0,9999	1
0,54	1	1
0,55	1	1

Table 5: Matrix Error Rate Values

7.4.3 *Correlated Channel and UDPLSA*

Experiments with UDPLSA on a correlated channel have a similar behaviour of the previous tests, presented in Section 7.4.1. For this reason, the only additional experiments carried out are presented in Section 7.5.

7.4.4 *Correlated Channel and ECLSA*

In this series of tests, a correlated channel has been used with our implementation of ECLSA. With a correlated channel, the success of the decoding procedure depends on the average length of a burst of lost packets. These experiments try to explore the relation between the burst length and the *Bundle Error Rate* (*BndER*). BndER is calculated using the aforementioned Equation 4. Different PERs and different losses burst lengths have been used as explained later.

PER values, used in these tests, are near the limit, found in the previous set of experiments, where the erasure code implementation included ECLSA is not able to recover bundles lost. For each experiment there is a fixed PER (0.4, 0.43, 0.46) and a variable burst length (increasing percentage of the number n of the matrix columns). Graphical results can be seen in Figure 17, Figure 18, Figure 19. The parameters showed in each graphic are the error rates based on: the number of *wrong*, *lost* and *lost plus wrong* bundles and the *Pecalc* (BndER obtained with Equation 4). All these parameters are similar to the ones used for the other experiments of the previous sections.

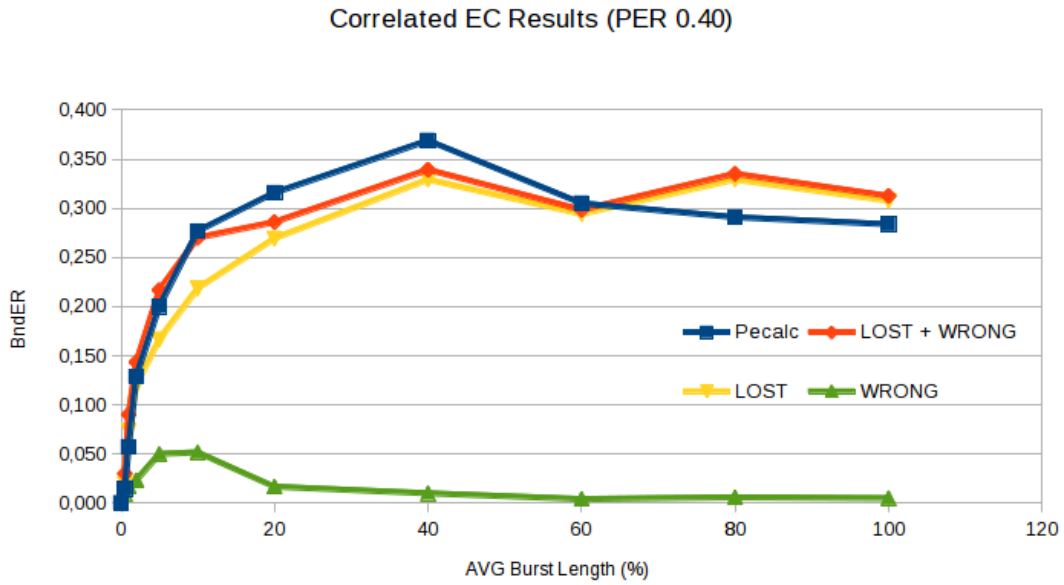

Figure 17: Correlated BndER (PER 0.40)

In Figure 17, the curves start from zero if *lostburst* is zero, which means that the channel is considered as uncorrelated. As soon as the percentage is incremented, the error rates is incremented, reaching, after that the average burst length is the 20% of n, a value close to the one configured as the required *loss* rate (in Figure 17 the value is 40). At this point the curves settle on this loss value.

If the lostburst value reaches values close to 100% the *wrong* curve is decreased to zero and the *lost* curve is increased. This happens because all the lost segments are concentrated in a single matrix, causing the decoding procedure of this matrix to fail. It is worth noting that, for the same reason, with the highest burst length, the *Pecalc* is lower than before. In fact, considering that the total loss rate is fixed, if a large number of losses are "consumed" in a single burst, the sequence of not lost segments will be increased too, causing for a while a lower number of fails during the decoding procedures.

The same observations done for the Figure 17 are valid for the other images (Figure 18 and Figure 19). The only different behaviour is that the curves are sharper in the first part, reaching sooner the loss value.

7.5 REAL SCENARIOS

After all the preliminary experiments, four real scenarios are simulated to evaluate the real benefits, obtained using ECLSA. For these experiments 5GB of data, divided into 2500 bundles each one of 2MB, are sent over an optical channel from the Earth to the Moon. The parameters of the network emulator,

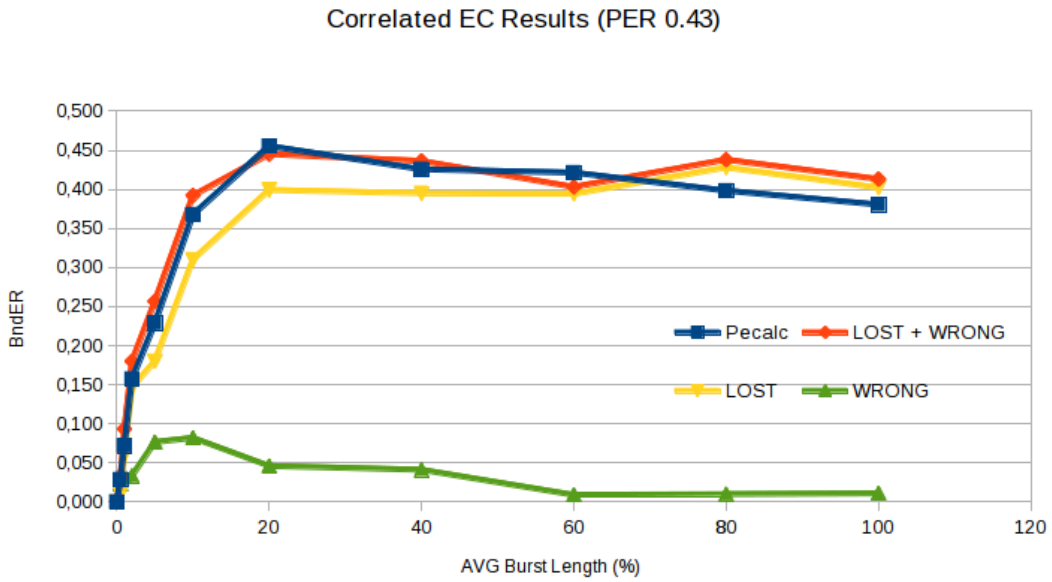

Figure 18: Correlated BndER (PER 0.43)

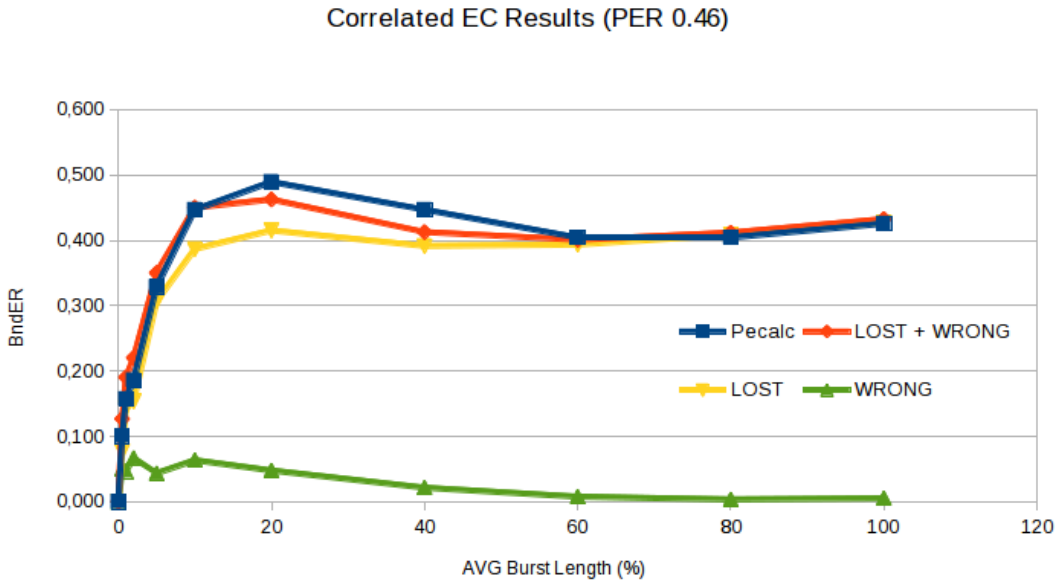

Figure 19: Correlated BndER (PER 0.46)

given by DLR, for the four scenarios are listed in Table 6. The data are sent using both UDPLSA and ECLSA.

Name	Bitrate (Mbps)	RTT (s)	Loss (Pe)	Lostburst (Segments)
S1	20	2.5	0.33	221
S2	20	2.5	0.13	338
S3	20	2.5	0.8	124
S4	20	2.5	0.03	186

Table 6: Real Scenarios Parameters

The results can be seen in Figure 20 and read in Table 7. In all the experiments BndER is considered as benchmark for performance of the transfers. Except for the first scenario, which is the most unfavourable, ECLSA is able to recover all the losses occurred, decreasing the BndER to zero. This result can be achieved because PER in the last three scenarios is not too high and then the erasure code implementation is able to recover all the lost segments. On the contrary, the BndERs, when UDPLSA is used, is high because the bundles size is very huge (2MB) and with these bundles, it is enough that a single segment is lost, to mark the bundle as wrong, increasing in this way the BndER. The result of the test using UDPLSO in S2, where the average burst length is higher than the one used in S3, indicates that the BndER is lower than the one in the third scenario. As explained in Section 7.4.4, this is an effect of the average good burst length (series of segments not lost) corresponding to the lost burst length analysed (series of segments lost). In fact, if a large number of losses are concentrated in a single burst, the PER value obtained will be incremented and, until this value do not return near the value set as limit in the channel emulator, a long series of segments will not be lost, causing during this time a lower number of fails during the decoding procedures.

In S1, ECLSA is not able to recover all the segments lost and this produces a residual BndER. This behaviour is caused by the *average* losses burst that can be worse than the one set on the channel emulator, causing the PER to be higher for brief periods.

LSA	SC1	SC2	SC3	SC4
UDP	0.991	0.640	0.765	0.032
EC	0.034	0.000	0.000	0.000

Table 7: Real Scenarios Bundle Error Rate Results (considering 2500 bundles transmitted)

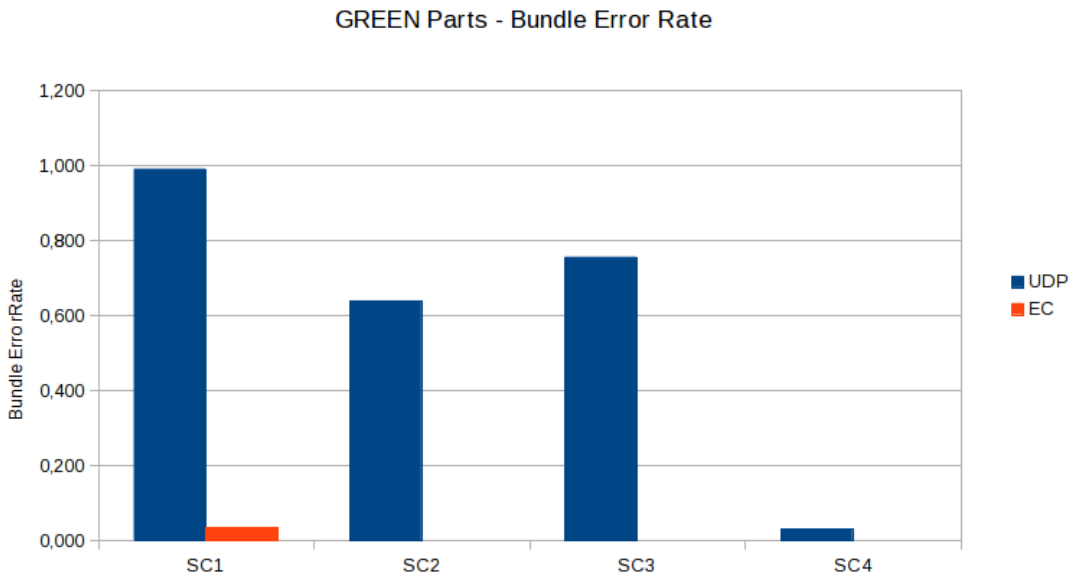

Figure 20: Real Scenarios Results (considering 2500 bundles transmitted)

7.6 GREEN EXPERIMENTS RESULTS ANALYSIS

The final results, presented in this chapter, shows the bad performance of UDPLSO if it is used with a channel with high PER. The experiments show also how the new implemented ECLSA is able, using ECs, to limit the number of LTP segments lost. Obviously, the bandwidth spent to send the redundancy informations has to be considered while designing a system which uses erasure codes. But, in some situation, where no other solutions are viable, the constraint on the bandwidth may be ignored, adding on the receiver node the capability to recover almost all the data lost.

7.7 PLANNED TEST WITH RED PART DATA

As explained above, we have found a problem in the LTP implementation used, which reduces performance of UDPLSO. In fact, trying to carry out our experiments with an uncorrelated channel with a PER of 0.10, and big bundles (1MB), as we have done in the previous experiments with Green, we discovered that more than one copies of a single bundles are delivered on the receiver. This is caused by a report segments limit which limits the number of report segments (RS) related to the same LTP session. In fact mixing the high PER used in our test (in preliminary tests is 0.10) and the dimension of our bundles (1MB) we had on the receiver a larger number of bundles than the number of bundles sent. Analysing the data transmitted on the channel,

we discover that, in the implementation used, the maximum number of RS transmission limit is ten. For each RS a maximum of 20 missing LTP segments can be reported. For instance, in order to well understand the problem, we carried out a one test with a bundle of 2MB (PER is always 0.10). This bundle is composed by about 1954 LTP segments, each of them of 1024 bytes. In this case, with the PER used, there are about 195 lost segments. The sender LTP peer sends one CP after the transmission of all the initial LTP segments (1954). Each RS can contain 20 maximum *Reception Claim* which are an assertion of reception of some number of contiguous bytes of application data (a subset of a block). The number of lost segments generates exactly 10 RS, which flows back from the receiver to the sender. The sender that receives 10 RS related to the same session, close the export session; the BP sends the same bundle again, creating a loop. In this particular example, no one bundle is received because of the session is always closed after each transmission. This behaviour in some circumstances (i.e. bundles of 1MB of length) leads to duplicate bundles or to have long delays during the transmission of the bundles. In the same scenario but using ECLSA, the problem does not occur, because ECLSA, recovering all the segments lost (with PER up to 0.46) "*solves*" the problem. For this behaviour of LTP, which must be discussed with the ION developers team, and not to present results whose reliability may be somewhat questionable, as said before, we have preferred to skip results with Red data.

However, We want to explain the planned tests with Red data. The aim of these tests is to prove the benefits offered by ECLSA when Red data are used. It is worth pointing out that here it would be no point in evaluating the BndER as done in dealing with the Green part, as with Red a full reliability, i.e. BndER=0, is almost always assured by the ARQ method implemented in LTP (exceptions require a number of consecutive retransmissions higher than a large maximum value). Vice versa, retransmissions results in a longer delivery time; ECLSA trades bandwidth, which is halved, with a reduction of retransmissions (necessary only in the unlikely case of a decoding failure). The idea therefore, to consider both the delivery time (pro) and the bandwidth halving (con) was to use goodput as figure of merit.

Similarly to the tests carried out with Green data, the experiments are divided into several different groups:

UNCORRELATED WITH UDPLSA: tests to prove the bad performance of Red data used with long RTTs;

UNCORRELATED WITH ECLSA: test the improvements in term of delivery time and goodput, obtained by using ECLSA;

REAL SCENARIOS: experiments in some real scenarios with long RTTs, i.e. further than Earth-Moon.

8

CONCLUSIONS

The objective of this thesis was twofold: to include erasure codes in the LTP protocol, and to assess the advantages offered by this solution in a variety of possible conditions with both green and red data. The rationale of adding erasure codes to LTP is given by the possibility of recovering LTP segments lost without resorting to retransmissions, as retransmissions may be too expensive in the presence of very long RTTs typical of space links (e.g. a few seconds to Moon, minutes to Mars, ...), or even impossible in the presence of unidirectional links. To this end a new link service adapter, the ECLSA, and some other auxiliary software components have been developed. Once achieved this first aim, we focused on performance evaluation. By comparing the basic UDP link service adapter, UDPLSA, with the new ECLSA, we demonstrated that in a variety of conditions ECLSA can offer better performance.

All the different phases of the work were carried out with the collaboration of both DLR and University of Bologna. During the six months spent in the Munich site of DLR, supported by a DLR grant, I carried out preliminary study, the design, and the implementation of ECLSA. Testbed tuning, experiments and results analysis, were then carried out at the University of Bologna.

All the experiments performed, especially the ones on real scenarios, have demonstrated the great potentiality of ECLSA for space communications. The interest on this subject has been perceived also in the last two CCSDS meeting where preliminary results have been presented. We hope that the new link service adapter may be included by NASA in the future releases of ION and that joint use of erasure codes with LTP may be eventually standardized by the CCSDS itself. The final wish is that our efforts, one day, will be useful in some real space deployment.

8.1 FUTURE WORKS

The tests presented in this thesis have demonstrated that the implementation of ECLSA works as expected and that can provides real advantages whit green data. However, during the first tests with Red data, we have found a problem related to LTP retransmissions in the presence of very high PER and huge bundles, which prevent us from carrying on the planned experiments. Although the problem is only related to the particular implementation of the LTP protocol used and then independent of ECLSA, it has a negative impact on performance of UDPLSO, used as a benchmark, and then on the accuracy

of the tests. For this reason, before continuing with the experiments, we want to fix the problem in a proper way, reporting the problem to the NASA's developers and finding with them a solution. As soon as the problem is fixed, it is our commitment to integrate this thesis with an Appendix presenting these new Red data results.

ACKNOWLEDGEMENTS

First of all, I would like to thank my two DLR's supervisors *Tomaso de Cola* and *Gianluigi Liva* who helped me in all the phases of my work, especially during the time spent in Germany. I would like also to thank *Balazs Matuz* who helped me with the first phases of this thesis. This work was supported in part by a grant from DLR, which helped me in the initial six months of work carried out in the *DLR Research Centre* in *Oberpfaffenhofen* (*Munich*). Finally, I would like to thank all the people all over the world who shares their knowledges, releasing the results achieved and the information discovered.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

BIBLIOGRAPHY

[1] S. Farrel, V. Cahill, D. Geraghty, and I. Humphreys. When TCP Breaks: Delay- and Disruption- Tolerant Networking. *Internet Computing, IEEE*, 10(4):72–78, July-August 2006.

[2] G. Liva, B. Matuz, M. Chiani, and E. Paolini. Maximum Likelihood Erasure Decoding of LDPC Codes: Pivoting Algorithms and Code Design. *Transactions on Communications, IEEE*, 60(11):3209–3220, November 2012.

[3] L. Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, April 1997.

[4] G.P. Calzolari, T. de Cola, G. Liva, and E. Paolini. Reliability Options for Data Communications in the Future Deep-Space Missions. *Proceedings of the IEEE*, 99(11):2056–2074, November 2011.

[5] P. Apollonio, C. Caini, T. de Cola, G. Liva, and B. Mutuz. Implementation of Erasure Codes as LTP Sublayer in ION. http://cwe.ccsds.org/sis/docs/SIS-DTN/Meeting%20Materials/2013/Spring/LTP_erasure%20codes_final.pdf, April 2013.

[6] T. de Cola. Erasure Codes Meet LTP in ION: Update and Results. http://cwe.ccsds.org/sis/docs/SIS-DTN/Meeting%20Materials/2013/Fall%20--%20San%20Antonio/LTP_w_EC.pdf, October 2013.

[7] Internet Research Task Force. http://www.ietf.org.

[8] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss. Delay Tolerant Networking Architecture. http://tools.ietf.org/html/rfc4838. IETF - RFC 4838.

[9] Consultive Committee For Space Data Systems. http://www.ccsds.org.

[10] K. Scott and S. Burleigh. Bundle Protocol Specification. http://tools.ietf.org/html/rfc5050. IETF - RFC 5050.

[11] S. Burleigh, M. Ramadas, and S. Farrell. Licklider Transmission Protocol Motivation. http://tools.ietf.org/html/rfc5325, September 2008. IETF - RFC 5325.

[12] M. Ramadas, S. Burleigh, and S. Farrell. Licklider Transmission Protocol. http://tools.ietf.org/html/rfc5326, September 2008. IETF - RFC 5326.

[13] S. Farrell, M. Ramadas, and S. Burleigh. Licklider Transmission Protocol Security Extensions. http://tools.ietf.org/html/rfc5327, September 2008. IETF - RFC 5327.

[14] DTN2 Documentation. https://sites.google.com/site/dtnresgroup/home/code/dtn2documentation.

[15] DTN2 Sources. http://sourceforge.net/projects/dtn/.

[16] S. Burleigh. Interplanetary Overlay Network: An Implementation of the DTN Bundle Protocol. In *Proc. of 4th IEEE Consumer Communications and Networking Conference*, pages 222–226, January 2007.

[17] ION Sources. http://sourceforge.net/projects/ion-dtn/.

[18] CCSDS Bundle Protocol Specification. http://public.ccsds.org/sites/cwe/rids/Lists/CCSDS7342R2/Attachments/734x2r2.pdf, October 2013. CCSDS - Red Book - 734.2-R-2.

[19] DTNPerf3 Web Page. http://cnrl.dei.unibo.it/new/software.php.

[20] E. Paolini, G. Liva, B. Matuz, and M. Chiani. Maximum Likelihood Erasure Decoding of LDPC Codes: Pivoting Algorithms and Code Design. *IEEE Transactions on Communications*, 60(11):3209–3220, November 2012.

[21] Virtualbricks on Launchpad. https://launchpad.net/virtualbrick.

[22] QEMU. http://www.qemu.org.

[23] Kernel based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[24] Virtual Distributed Ethernet. http://vde.sourceforge.net/.