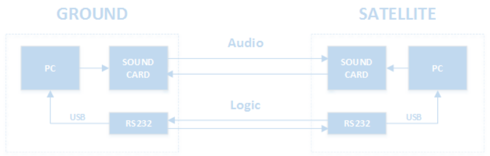
The aim of this project is the development of a data handling, telemetry telecommand protocol for onboard computer cubesat. This project has researched the last used protocol to choose between them the best option depending on it efficiency. It has also been done an improvement of the chosen protocol according to the requirements of the GranaSAT project.



Lucía Castañeda González *is a telecommunications engineer from Granada, Spain. She finished her Bachelor's Studies in 2017 at University of Granada.*

Andrés María Roldán Aranda *is the academic head of the present project, and the student's tutor. He is professor in Departament of Electronics and Computer Technology at University of Granada*

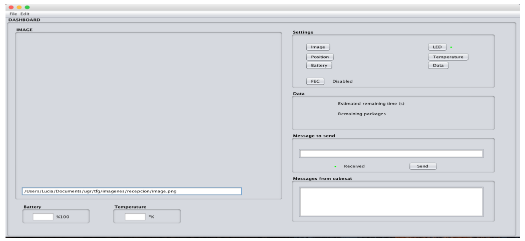
BACHELOR THESIS

Data Handling, Telemetry Telecommand protocol for Onboard Computer Cubesat

Lucía Castañeda González

UNIVERSITY OF GRANADA

Bachelor Degree in Telecommunications Technology Engineering



/Users/Lucia/Documents/ugr/tfg/imagenes/recepcion/image.png

Bachelor Thesis

Data Handling, Telemetry Telecommand protocol for Onboard Computer Cubesat

Lucía Castañeda González

Academic year 2016/2017

Tutor: Andrés María Roldán Aranda

GRADO EN INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

“Data Handling, Telemetry Telecommand protocol for Onboard Computer Cubesat”

CURSO: 2016/2017

Lucía Castañeda González

GRADO EN INGENIERÍA DE TELECOMUNICACIÓN

"Data Handling, Telemetry Telecommand protocol for Onboard Computer Cubesat"

REALIZADO POR:

Lucía Castañeda González

DIRIGIDO POR:

Andrés María Roldán Aranda

DEPARTAMENTO:

Electrónica y Tecnología de los Computadores

D. Andrés María Roldán Aranda, Profesor del departamento de Electrónica y Tecnología de los Computadores de la Universidad de Granada, como director del Trabajo Fin de Grado de Dña. Lucía Castañeda González,

Informa:

que el presente trabajo, titulado:

"Data Handling, Telemetry Telecommand protocol for Onboard Computer Cubesat"

ha sido realizado y redactado por el mencionado alumno bajo nuestra dirección, y con esta fecha autorizo a su presentación.

Granada, a 15 de julio de 2016

[illegible]

Fdo. Andrés María Roldán Aranda

Data Handling, Telemetry Telecommand protocol for Onboard Computer Cubesat

Lucía Castañeda González

PALABRAS CLAVE:

Cubesat, AX25, FX25, FEC, Eclipse, Java, Netbeans.

RESUMEN:

El propósito principal de este proyecto es el desarrollo de un protocolo para el manejo de datos y comandos. Este protocolo está implementado para el proyecto de GranaSAT. Con el protocolo el satélite de GranaSAT podrá comunicarse con el ordenador en tierra.

Este trabajo es presentado como el Trabajo Final de Grado del grado de Ingeniería de Telecomunicaciones de la Universidad de Granada.

Para este proyecto se ha investigado los últimos protocolos que se están utilizando para proyectos que trabajan con cubesats para elegir el que se adapta mejor al proyecto de GranaSAT. También se ha realizado una series de mejoras para adaptarse a los requisitos de GranaSAT. El proyecto has sido implementado en un escenario simulado con la intención de que en un futuro se desarrolle para el escenario real de GranaSAT.

Data Handling, Telemetry Telecommand protocol for Onboard Computer Cubesat

Lucía Castañeda González

KEYWORDS:

Cubesat, AX25, FX25, FEC, Eclipse, Java, Netbeans.

ABSTRACT:

The main purpose of this project is the development of a data handling, telemetry telecommand protocol for onboard computer cubesat. This protocol is implemented to the GranaSAT project. With this protocol the GranaSAT satellite will can telecommunicate with the ground computer.

The work is presented as a Degree Final Project for the degree of Telecommunications Engineering of the University of Granada.

This project has researched the last used protocol to choose between them the best option depending on it efficiency. It has also been done an improvement of the chosen protocol according to the requirements of the GranaSAT project. The project has been implemented in a simulated scenario with the intention that it will be implemented in the real scenario in the future in the GranaSAT project.

Agradecimientos:

Primero me gustaría agradecer el apoyo incondicional de mis padres, Alejandro y Lucía, de mi hermana y a Tula. Que me han acompañado en esta travesía a lo largo de mi periodo universitario.

También deseo agradecer a mis amigos por ayudarme durante esta etapa estando presentes en los momentos más felices pero también ayudándome en los más amargos.

Por último, agradezco a mi tutor, Andrés María Roldán Aranda, su dedicación, su continua preocupación por la calidad y el trabajo bien hecho y por su entusiasmo.

Acknowledgments:

First of all, I would like to thank my parents, Alejandro and Lucía, my sister Ángela and Tula their unconditional support. Who have accompanied me throughout my university period.

I thank my friends for helping me throughout this period, being present in the happiest moments but also in the bad ones helping me.

Finally, I would like to thank to my advisor, Andrés María Roldán Aranda, his dedication, his continued concern for quality and well-done work and his enthusiasm.

INDEX

Autorización Lectura v

Autorización Depósito Biblioteca vii

Resumen ix

Dedicatoria xiii

Agradecimientos xiii

Index xv

Index of figures xxi

Video Index xxiii

Table index xxv

Glossary xxvii

Acronyms **xxix**

1 Introduction **1**

1.1 State of the art: *Protocols for cubesat* 2

1.2 Motivation 2

1.3 Project goals 2

1.4 Project structure 3

2 Definition of requirements **5**

2.1 Requirements 5

2.2 Hardware requirements 6

2.2.1 Real scenario 6

2.2.1.1 Frequencies 7

2.2.1.2 Real example data 8

2.2.2 Simulated scenario 8

2.2.2.1 Comparison between sound cards 10

2.2.3 Software requirements 10

2.2.4 Test that will be done to the protocol and to the system to see if it meets the above requirements. 11

3 System analysis **13**

3.1 Solutions for sending data to cubesat 13

3.1.1 Protocols used 13

3.1.1.1 Preliminary 14

3.1.1.1.1 OSI Model 14

3.1.1.1.2 Framing 15

3.1.1.2 Original AX25 15

3.1.1.2.1 AX25 general description 15

3.1.1.2.2 Flag field 17

3.1.1.2.3 Address field 18
3.1.1.2.4 Control field 19
3.1.1.2.5 PID field 21
3.1.1.2.6 Info field 21
3.1.1.2.7 FCS field 21
3.1.1.2.8 Invalid frames 22
3.1.1.3 FX25 (AX25 with FEC) 22
3.1.1.3.1 FX25 general description 23
3.1.1.3.2 FEC algorithm 24
3.1.1.3.3 FX25 improvement 25
3.2 Analysis of possible solutions to requirements 26
3.2.1 Efficiency comparison 26
3.2.1.1 AX25 efficiency 26
3.2.1.2 FX25 efficiency 28
3.2.1.3 Efficiency comparison between AX25 and FX25 30
3.2.2 Solution choice 31

4 System design **33**
4.1 Reverse engineering 33
4.1.1 Transmission 33
4.1.2 Reception 34
4.1.3 Class packet 34
4.2 Improvement of the protocol 35
4.2.1 Class packet improvement 35
4.2.2 Packet receiver 35
4.2.3 Sending, receiving and analysis of new commands 37
4.2.3.1 Short commands 37
4.2.3.2 Long commands 38

4.2.3.3 Answer commands 39
4.2.3.4 AX25 specific commands 39
4.2.3.5 Short answer commands specifications 39
4.2.3.5.1 Possible answers from satellite 39
4.2.3.5.2 Possible answers from ground to the answers from the satellite 40
4.2.3.6 Long answer commands specifications 40
4.2.3.6.1 Possible answers from satellite 40
4.2.3.6.2 Possible answers from ground to the answers from the satellite 40
4.2.3.7 Method to send a long message 41
4.2.3.8 Commands schematics 41
4.2.4 Sending and receiving images through AX25 and FX25 45
4.2.4.1 Sending images through AX25, transmitter 45
4.2.4.2 Receiving images with AX25, receiver 48
4.2.4.3 Sending and receiving images with FX25 54
4.2.5 Forward Error Correction improvement with java 55
4.2.5.1 Generate FEC 55
4.2.5.2 Detect and correct errors with FEC 61
4.2.5.3 General overview 65
4.2.5.4 FEC packet 65
4.3 Application design end to end 69

5 Fabrication **71**
5.1 Operative systems 71
5.2 Development environments 71
5.2.1 Comparison between Netbeans and Eclipse 71
5.2.2 Netbeans 72
5.2.3 Eclipse 72

5.2.4 Development environments choice 73

6 Tests and validation **75**

6.1 Images 75

6.1.1 Completely sent and received 75

6.1.2 Images comparison 76

6.1.3 Timing 77

6.2 FEC 78

6.2.1 Working FEC 79

6.2.2 Timing with FEC 81

6.2.3 Interferences in the link with and without FEC 83

7 Conclusions and future improvements **85**

References **87**

INDEX OF FIGURES

1.1 Project Development Gantt Chart. 4

2.1 Real scenario of the proyect, when the ground sends 6

2.2 Real scenario of the proyect, when the satellite sends 7

2.3 Simulated scenario with two computers 8

2.4 Simulated scenario with one computer 9

3.1 Transmitter sends and receiver receives incorrectly and the CRC detects it. . 16

3.2 Transmitter sends and receiver receives correctly and the CRC detects it. . . 16

3.3 Transmitter sends and it does not receive any respond. The RR frame is answer by a RR frame. 17

3.4 Transmitter sends and it does not receive any respond. The RR frame is answer by a Reject frame. 17

3.5 Transmitter sends and it does not receive any respond. The RR frame is answer by a RNR frame. 18

3.6 Efficiency of AX25 with minimum overload 27

3.7 Efficiency of AX25 with maximum overload 28

3.8 Efficiency of FX25 with minimum overload 29
3.9 Efficiency of FX25 with maximum overload 29
3.10 Efficiency of AX25 and FX25 with minimum overload 30
3.11 Efficiency of AX25 and FX25 with maximum overload 30

4.1 LEDON command representation. 42
4.2 LEDOFF command representation. 42
4.3 TEMP command representation. 42
4.4 BAT command representation. 43
4.5 POS command representation. 43
4.6 IMAGE command representation. 44
4.7 DATA command representation. 44
4.8 Sending an image, in the transmitter. 45
4.9 EmitLongMessages method. 47
4.10 Receiving an image. 49
4.11 Payload complete representation. 50
4.12 Join message method representation. 52
4.13 Overview of FEC generation, in the transmitter 60
4.14 Overview of FEC solution, in the receiver 65
4.15 FEC format Packet. 68
4.16 Application design. 69

6.1 Explanatory scenario of the sending of images 77
6.2 Graphic with the compressed images 77
6.3 Image timing 79
6.4 FEC timing 82
6.5 Timing comparison 83

VIDEO INDEX

4.1 Reception video. 54

TABLE INDEX

2.1 Sound card Usb comparison 10

3.1 Layers of the OSI model 14
3.2 U and S frame construction. Adapted from Ref [5] 15
3.3 I frame construction. Adapted from Ref [5] 15
3.4 Non-repeater address fiel encoding. Adapted from Ref [5] 18
3.5 Field encoding. Adapted from Ref [11] 18
3.6 Command and respond encoding. Adapted from Ref: [5] 19
3.7 Control field with module 8 sequence number. Adapted from Ref: [11] 19
3.8 Control field with module 128 sequence number. Adapted from Ref: [11] . . 19
3.9 S frame, module 8.Adapted from Ref: [5] 20
3.10 S frame, module 128.Adapted from Ref: [5] 20
3.11 U frame encoding. Adapted from Ref: [11] 21
3.12 PID enconding. Adapted from Ref: [5] 22
3.13 FX25 frame. Adapted from Ref: [9] 23
3.14 FEC codification matrix. 24

3.15 FEC generation example 25

3.16 Example of a detection of bit flip 25

3.17 Example of the location of a bit flipped 25

3.18 FEC packet form. 26

3.19 AX25 overload 27

3.20 FX25 overload 28

GLOSSARY

AX25 AX25 is a communication protocol derived from the AX25 protocol designed for use by radio amateurs..

Eclipse Eclipse is a computer program composed by a multiplatform open source programming tools to develop applications..

FX25 FX25 is a communication protocol derived from the AX25 protocol designed for use by radio amateurs. FX25 uses Forward Error Correction..

GranaSAT GranaSAT is an academic project from the University of Granada consisting of the design and development of a picosatellite (Cubesat). Coordinated by the Professor Andrés María Roldán Aranda, GranaSAT is a multidisciplinary project with students from different degrees, where they can acquire and enlarge the necessary knowledge to face a real aerospace project. http://granasat.ugr.es/.

Netbeans Netbeans is a free integrated development environment. The main programming language for which it was developed primarily is Java..

ACRONYMS

ADC	Analog to Digital Converter.
AWT	Abstract Window Toolkit.
CRC	Cyclical Redundancy Check.
DAC	Digital to Analog Converter.
FCS	Frame Check Sequence field.
FEC	Forward Error Correction.
HDLC	High-Level Data Link Control.
IDE	Integrated Development Environments.
ISO	International Organization for Standardisation.
LSB	Least Significant Bit.
OSI	Open System Interconnection.
PID	Protocol Identifier Field.
REJ	Reject Frame.

RNR	Receiver Not Ready.
RR	Receiver Ready.
SOA	Service-Oriented Applications.
SREJ	Selective Reject.
TNC	Terminal Node Controler.
UHF	Ultra High Frequency.

CHAPTER

1 INTRODUCTION

This project is presented as a final sample of the knowledge acquired during the completion of the Degree in Telecommunication Engineering. The main objective of the project is to implement a data handling and telemetry telecommand protocol, based on the AX25 protocol.

The project starts from the necessity of a protocol for the transmission of data between a station in ground and a satellite. It must be a protocol that allows essential changes that may need to include the end user. All possible scenarios, in which two entities can communicate with each other with this protocol, will be taken into account.

This project is based on the work of Silvan Toledo that implemented a simple AX25 on Java, without the CRC correction. As it will be explained on 4.

As it will be detailed in 2.2.1 the protocol has been implemented to be useful in a real scenario but it has been implemented in a simulated scenario as it will be shown in 2.2.2.

This protocol will be useful to use it on a cubesat project. The protocol has been implemented to can be used both for ground station and for the satellite.

As discussed in section 3.1 the protocol is born from the AX25.

1.1 State of the art: *Protocols for cubesat*

AX25 is the most common protocol used in cubesats project. This protocol, described in chapter 3, is mainly used to establish connections and transfer data between nodes, detecting errors introduced by the communications channel.

The AX25 error detection is not very complex and it does not solve elaborated problems. AX25 uses CRC to search error. With this technique it can solve problems of lost or disordered frames. But AX25 protocol is mainly use for radio link connections, in this kind of links there may be interference that damage the frame, and this kind of error can not be solved with the CRC.

AX25 is a simple protocol with no many commands. This allows this protocol to be used as a base for other more complex protocols.

There are other protocols that are used for cubesats. FX25 is one of them. This protocol is based on AX25 and it implements FEC what can solve the interference problems but it increases considerably the overload on the frames.

1.2 Motivation

First of all, the most important motivation of this project is to finalize the four years invested in this graduate training.

On the other hand, to offer solutions to real problems, showing the usefulness of the knowledge acquired over the years of study. Strengthening the motivation and confidence of the student in their capacities and in her knowledge, in special in the speciality of telematics and information technologies.

1.3 Project goals

The main objectives of this project are:

- To know and analyse the needs of a protocol for a connection between a satellite and a ground base.
- To extract the main and secondary requirements of the protocol to be implemented, to assess the viability of each one of them and to propose new options.
- To carry out the analysis of possible implementations through which the needs of each of the requirements could be solved.
- To do reverse engineering the Sivan Toledo project to analyse what can be used in my project.

- Design a protocol that meets the requirements.
- To perform the necessary tests to determine that the protocol meets the requirements.
- To demonstrate the knowledge acquired by the student acquired during the speciality of telematics and information technologies in the Degree in Engineering of Telecommunication Technologies.
- Overcome the subject of Project End of Degree with success.

1.4 Project structure

The project is developed over 7 chapters. These chapters are intended to describe in a logical and chronological way the work carried out during the duration of the project.

The chapters that form the present documents are:

- This chapter, numerates as 1, is an introduction to the project.
- Chapter 2 is about the main and secondary needs of the product to be designed in the form of requirements are shown.
- Chapter 3 explain the system analysis. The possible solutions and their advantages and disadvantages.
- In chapter 4 the system design is explained. It explains the reverse engineering performed and the new implementations.
- Chapter 5 is about the manufacture of the product: an explanation of the development tools used in the code production flow.
- In Chapter 6 all the testing performed is shown.
- The last chapter, chapter 7, is a conclusion chapter as a summary of the document and some lines of future work.

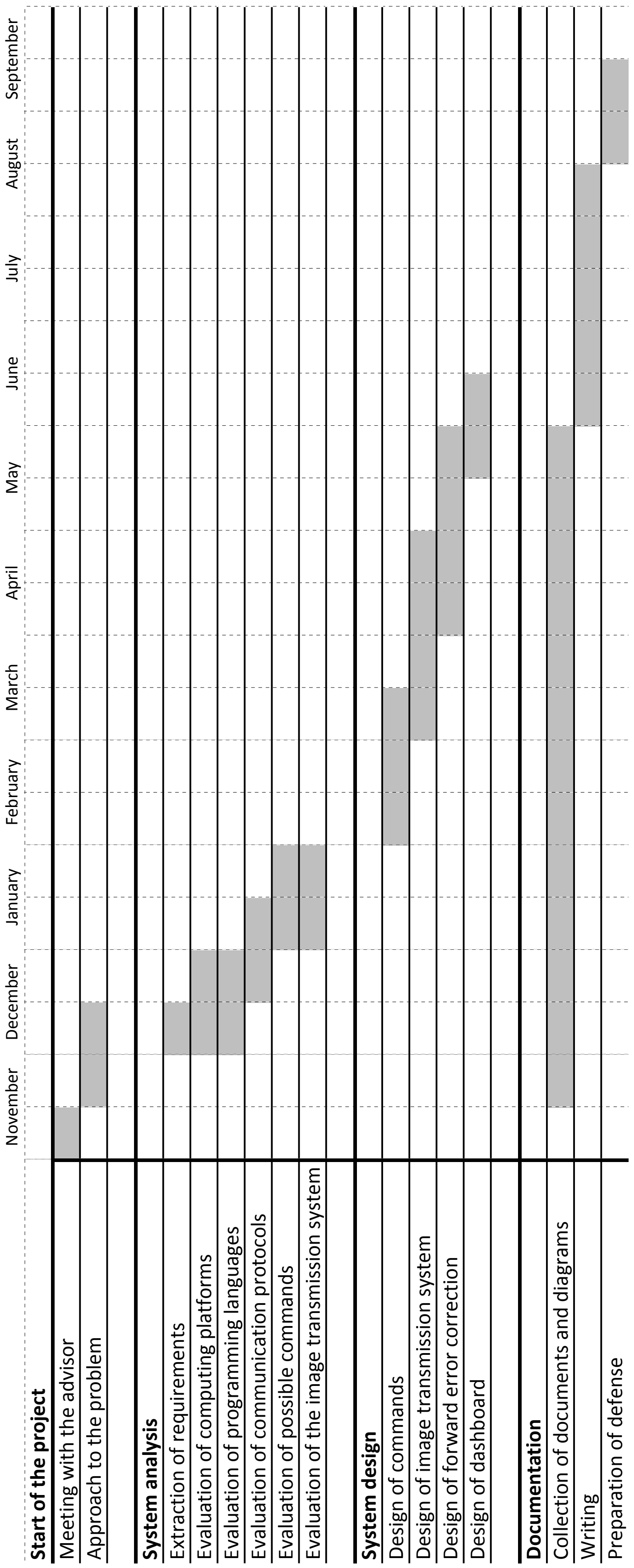


Figure 1.1 – *Project Development Gantt Chart.*

CHAPTER

2 DEFINITION OF REQUIREMENTS

After the introduction of the present project I proceed to explain the requirements of the project. In this chapter the main and secondary requirements are explained. It includes also the hardware requirements, that explain the main scenario that is going to be simulated with the Java program.

2.1 Requirements

The main requirements of this project are:

1. Program to implement the protocol for Onboard Computer Cubesat.
2. FEC generator.
3. Reconstruction of a message damaged by FEC.
4. Properly emission and reception of images from the satellite.
5. Development of telecommands with their respective responses.
6. Implementation of the developed telecommands.

The secondary requirements are:

1. That the program calculates the time remaining to complete a broadcast or reception.
2. Visual application for the control of the satellite through the implemented protocol.
3. Visual application simulation the satellite terminal.
4. To calculate satellite position to know when transmission can start.

2.2 Hardware requirements

There are two possible scenarios were we have to analyze the hardware requirements. The first one is the real scenario and the second one is the simulated scenario. For this reason the requirements for the simulated scenario are those that we will really need.

2.2.1 Real scenario

As already mentioned in the chapter 1 this proyect implements a protocol for onboard computer cubesat. This is implemented in a computer simulating the computer that is going to be on ground and the satellite, but the real scenario we are trying to implement is the one I will proceed to explain next.

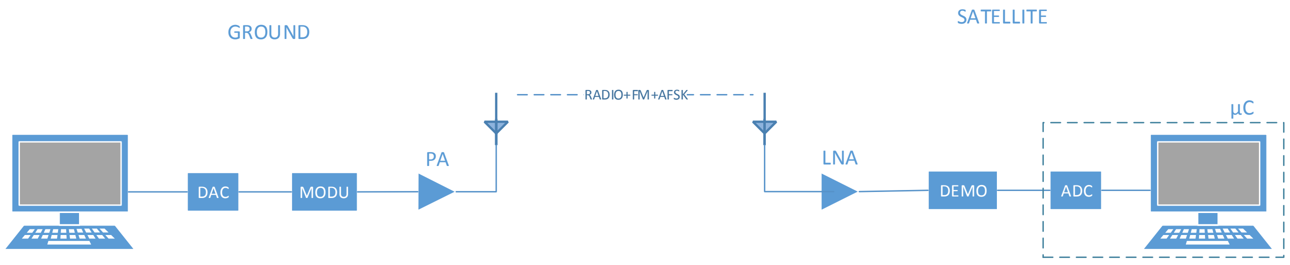


Figure 2.1 – *Real scenario of the proyect, when the ground sends*

In the figure 2.1 we can observe the real scenario that is going to be implemented. On the left side is located ground. There the main computer is situated. This computer is in charge of performing the analysis of the information that comes from the satellite and responding accordingly. When this computer send a message the signal it produces is a digital signal. This signal has to be converted to analogical with the DAC. Then the analogical signal passes through the modulator and after that through the PA. Finally the antenna transmits the signal through radio with FM plus afsk. When a message is received on ground the method is exactly the same but in the opposite direction. In this case would not work the modulator but a demodulator, and instead of a DAC would be an ADC.

On the right side is located the satellite. The satellite side works really similar to the ground side. The main difference is the computer capacity, for process signals and information. In the microcontroller there are the computer, which is the one in charge to process the signals and to send and receive the packages, and the ADC , which converts the analog signal incoming into a digital signal. When the signal reaches the antenna of the satellite it goes through the LNA, then passes the demodulator and arrives at the ADC that converts the analog signal into a digital signal. Finally the digital signal arrives at the computer onboard and it analyse the message and decides which actions it is going to take.

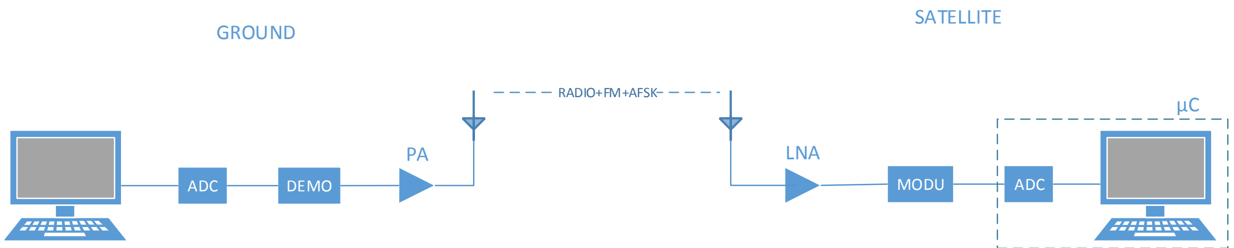


Figure 2.2 – *Real scenario of the proyect, when the satellite sends*

In the figure 2.1 the scenario represented is when ground sends and the satellite receives. It could be in the opposite way as it was described, as we can see in the figure 2.2.

Summing up, the hardware requirements to the real scenario are:

1. One computer that can run a program implemented with Java.
2. Two ADC.
3. Two demodulators.
4. Two modulators.
5. Two microprocessors.

2.2.1.1 Frequencies

To perform any radio transmission it is necessary to choose a frequency. For this kind of projects, the most used frequency is the amateur radio range of frequencies that are 145MHz-430MHz. This frequencies are low enough to transmit successfully even though the satellite is not in the ideal position. The main disadvantage about this frequencies is that they are used very frecuently and consequently the bandwidth is low. The other frequency option is 5GHz, this increases considerably the bandwidth making the transmission faster. But to use this frequency it is necessary that the ground antenna is perfectly directed to the satellite, and that the satellite is close to be just above the ground station.

2.2.1.2 Real example data

Some data about the scenario already shown are for example the distances from ground to the satellite. If the satellite is orbiting just above ground the distance between them it is the orbit height. Here there a list of different orbit height of some satellites:

- ISS: 400 km.
- MIR: 358 km.
- SkyLab: 235 km.
- Sputnik: 577 km.

There are other data about real scenarios that will depend a lot on the way the satellite is implemented. From references [12] and [7] we can find some interesting information about its satellite scenario. This article is about the cubesat implemented by the University of Oslo, which name is CubeStar. It operates in the UHF amateur satellite band (435-438MHz), operating in a 25 kHz channel. It has two different links, a high-speed link and a low-speed link. This is a regular way of work, with the high-speed link transmitting the main information or data, and with the low-speed link transmitting telecommands. In the case of CubeStar the low-speed it is used to transmit a tracking signal.

2.2.2 Simulated scenario

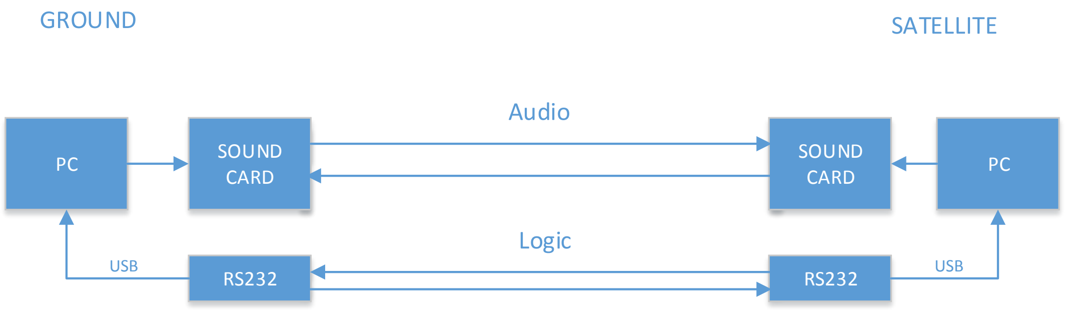


Figure 2.3 – *Simulated scenario with two computers*

The main purpose of this project was to make alive the protocol AX25 with a computer. Although the proyect needed to be based on the real scenario we do not have yet that scenario. For that reason we needed to describe a new scenario that we could implement and that would be as close to the real one as possible. This is the simulated scenario, represented in the figure 2.3.

It must be highlighted that although the figure differentiates between ground and satellite, both parts are implemented exactly the same way. The implementation of each side needs a normal computer that can run a program implemented with Java, it is also necessary two RS232 USB adapter, one RS232 cable and two jack cables 3.5 mm. If the computer has sound card that works properly, with one input and one output separately, it is not necessary an external sound card. Both side are connect by the sound cables and the RS232.

In summary, what we will need to set up our scenario will be:

1. 2 computers that can run a program implemented with Java
2. 4 RS232 usb adapters
3. 2 RS232 cables
4. 2 jack cables 3.5mm

The RS232 cables check in both directions if the channel is busy, this is done with a logic signal. Although this will always be ready, because the channel we are using is just for us, we use this to represent that we have to check if our frequency is in used to can send without interference of another emission we are not controlling.

The sound cables transmit the signal with the information by a sound signal, one cable for each direction. This represents the radio link on the real scenario. On a radio link some problems with interference or noise can appear that in the sound cables do not. For that reason it is necessary to check the program introducing ourselves this problems, this is done on chapter 6.

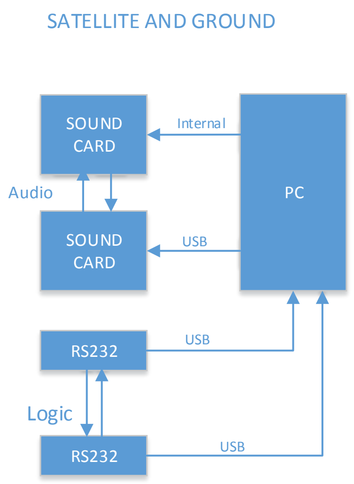


Figure 2.4 – *Simulated scenario with one computer*

In order to allow to work more comfortably, the simulated scenario can also be implemented, in almost all situation, with just one computer. This scenario can be seen in figure 2.4. In this scenario there is just a computer. It is necessary an extra sound card, that would be connected by usb. The others devices are the same as the scenario with two computers, but all of them will be connected in only one computer. The only situation where we can not use this scenario is if the computer internal sound card has only one input. On most modern computers the sound input has only one input, for headphones and microphone. If this is the case it is impossible to check the both directions of one of the side, that would be the one represented by the internal sound card, it is only possible to check the first directions initiated.

Although with two computers the scenario is more representative, for ease of use only one computer has been used in most of the work, although checking each important advance with the scenario with two computers.

2.2.2.1 Comparison between sound cards

Table 2.1 – *Sound card Usb comparison*

Mark	Price	Speed of shipment	Size (mm)	Noise	Choice
Ugreen	7.98€	High (1 day)	46x 26x 26	1/100	
Sodial (TM) USB 5.1	1.70€	Low (20 days)	64x 32x 12	2/100	X
Vodool	7.25€	High (1 day)	28x 28x 11	Unknown	

The noise is referring to how high is the volume when there is no signal input. The maximum volume the Java program can capture is 100. Even though the noise of the Sodial sound card is one point higher than Ugreen, when it is used to transfer information, the quality of the information received did not vary significantly with respect to Ugreen.

In order to implement the scenario with just one computer, or the scenario with two computers if one of them has the internal sound card with the input and the output not separately, it is necessary to acquire a sound card by usb. There many interesting options in the market, in the table 2.1 there are the analyzed ones and more interesting for our case.

All these devices can be found in Amazon.

About size Vodool, it has a little size but it width does not allow to use the near inputs, for that reason I have categorized this mark as moderate with respect to size.

The final decision was to use Sodial, but in the middle of the work I needed another sound card and I bought Ugreen and it worked really well, much better than Sodial.

2.2.3 Software requirements

To develop this project the following requirements are necessary:

1. OS that allows the installation of the necessary development programs.

2. Java Development Kit.
3. Eclipse development environment.
4. Netbeans development environments.

All this tools are going to be explained in chapter 5.

2.2.4 Test that will be done to the protocol and to the system to see if it meets the above requirements.

The tests that are going to be runned are the following:

1. Proper emission and reception of any kind of packet.
2. Analysis of RAM.
3. Time measurement.
4. Introducing cuts on the line with and without FEC working.
5. Introducing change of volume with and without FEC working.
6. Introducing noise with and without FEC working.

CHAPTER 3 SYSTEM ANALYSIS

This chapter explains the different options to fulfill the system requirements, includes a comparison between them and finally presents the chosen option.

3.1 Solutions for sending data to cubesat

Nowadays there are two different protocols that have been used to send data to cubesats. Those protocols are AX25 and FX25. In this section we will explain both of them as the possible protocols used as solution for this project.

3.1.1 Protocols used

To better understand the possible solutions it is necessary to understand the OSI model and framing that are usually used on networks that work with radio. The packet radio is the network that is going to be implemented to use our protocol. Packet radio has many structural similarities with one of the most known network, the telephone network. In a Packet Radio there are no telephone but radio transceivers. While a telephone network works with modem, on a Packet radio it is necessary TNC. And the last big difference is that the information on a Packet Radio goes by radio waves. In the reference [13] it is explained in detail how TNC works.

3.1.1.1 Preliminary

3.1.1.1.1 OSI Model

In order to understand any network or telecommunications protocol it is convenient to know the OSI model. The OSI model is a reference model for layered architecture network protocols, created by the ISO. The OSI model can be viewed in table 3.1.

Table 3.1 – *Layers of the OSI model*

	OSI MODEL
Layer 7	Application
Layer 6	Presentation
Layer 5	Sesion
Layer 4	Transport
Layer 3	Network
Layer 2	Data link
Layer 1	Physical

OSI is a regulation formed by 7 layers. Those layers defines the different phases through which the data must pass to travel from one device to another over a communication network. The first layer, the *physical layer*, defines how data is physically transmitted. It is responsible for transforming a data frame from the link level into a signal appropriate to the physical environment used in the transmission. The second layer, *data link*, is responsible for providing an error-free transmission, a reliable data traffic through a physical link. The data link layer deals with physical addressing, network topology, network access, error reporting, orderly frame distribution and flow control.

The role of the third layer, *network layer*, is to make the data arrive from the source to the destination, even if both are not directly connected. The main function of the fourth layer, *Transport Layer*, is to accept the data sent by the upper layers, split them into small parts if necessary, and pass them to the network layer. This layer also ensures that they arrive correctly on the other side of the communication. The fifth layer, the *sesion layer*, establishes of contact, manages and terminates connections between end users. This layer is responsible for the control of the session, the establishment between the sender and the receiver and the control of the concurrence. The objective of the sixth layer, the *presentation layer*, is to take care of the representation of the information. This layer also allows you to encrypt and compress data. Finally, the seventh layer, *application layer*, offers to applications the ability to access the services of other layers and defines the protocols used by applications to exchange data.

3.1.1.1.2 Framing

When we are sending data we need to send it in a way that the receiver can recover and recognise it. It is necessary to have boundaries that will delimit the different frames and the field within the frame. The way used to delimit the frames is using flags. The flags are a bit or a byte sequence that is different from anything else in the frame.

A lot of radio protocols are based on HDLC. HDLC is a bit-oriented synchronous data link layer protocol. It is used most of the time to connect two devices between them, but it can be used too to point to multipoint connections.

3.1.1.2 Original AX25

The first, and most common used protocol, is the AX25 that originally implements the CRC.

3.1.1.2.1 AX25 general description

AX25 has three general types of frames:

- Information frame (I).
- Supervisory frame (S).
- Unnumbered frame (U).

Each of this frames is formed by fields.

Table 3.2 – *U and S frame construction. Adapted from Ref [5]*

Flag	Address	Control	Info	FCS	Flag
01111110	112/224 bits	8/16 bits	N*8 bits	16 bits	01111110

Table 3.3 – *I frame construction. Adapted from Ref [5]*

Flag	Address	Control	PID	Info	FCS	Flag
01111110	112/224 bits	8/16 bits	8 bits	N*8 bits	16 bits	01111110

In the table 3.2 and table 3.3 the fields of the general types of frames are shown. PID is the Protocol Identifier Field and FCS is the Frame Check Sequence field.

The Info field is not always used, it is only used in certain frames. It is worth noting that every field have an integer number of bytes.

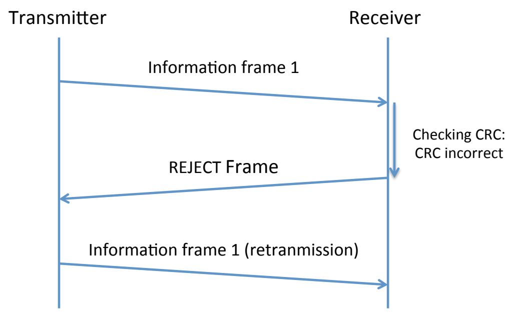


Figure 3.1 – *Transmitter sends and receiver receives incorrectly and the CRC detects it.*

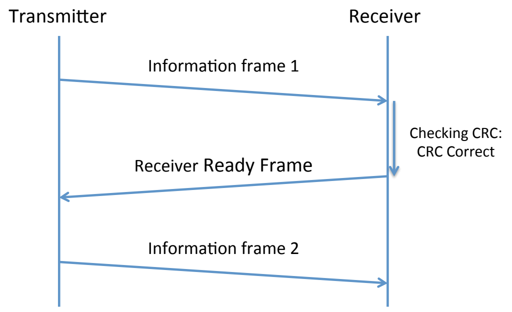


Figure 3.2 – *Transmitter sends and receiver receives correctly and the CRC detects it.*

AX25 can determine the integrity of the information on a frame with CRC. In AX25 FCS is the name given to the value of CRC. If the receiver detect a CRC error it will respond with a REJECT frame (section 4.2.3). This message exchange can be observe in the figure 3.1. The receiver checks the previous received information,then asks for the re-transmission of the damaged frame. If the receiver does not find any frame corrupted it responds with a Receiver Ready (RR)(section 4.2.3), which can be observed in the figure 3.2.

The transmitter has a timer in case it does not receive any respond from the receiver, either REJECT or RR frame, if it is time-out the transmitter sends a RR frame trying to check in which situation is the receiver. Answering to this message the receiver sends RR, REJECT or Receiver Not Ready (RNR) frame (section 4.2.3).

If the reply message is a RR frame, the transmission continues, and it does it from the last frame sent. This message exchange is described in the figure 3.3.

If the reply message is a REJECT frame, the transmitter resume transmission from the corrupted frame sequence. This message exchange is described in the figure 3.4.

If the reply message is a RNR frame, it means the receiver can not keep receiving more frames. In this last case the transmitter waits a certain time to retry the connection to the receiver, this message exchange is described in the figure 3.5.

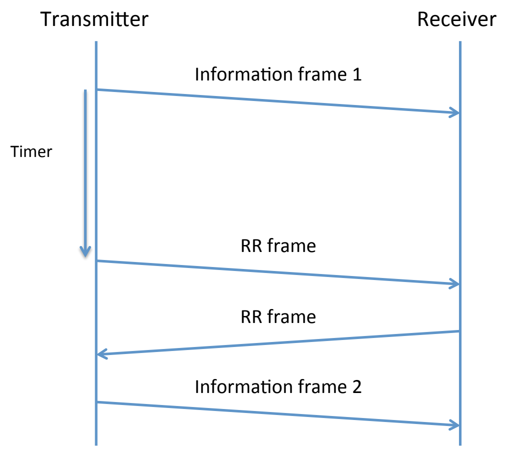


Figure 3.3 – *Transmitter sends and it does not receive any respond. The RR frame is answer by a RR frame.*

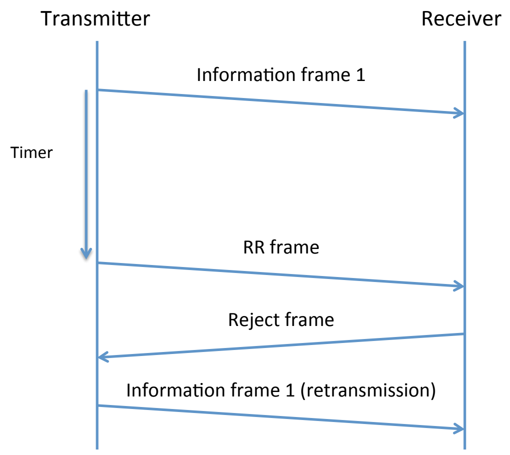


Figure 3.4 – *Transmitter sends and it does not receive any respond. The RR frame is answer by a Reject frame.*

3.1.1.2.2 Flag field

This field is used to delimit the frames. It is necessary to use one at the beginning and other at the end of each frame. This field is 8 bits long, and the pattern is 01111110. This means that this sequence is reserved and that it is not allowed to be used anywhere else inside the frame. To achieve this purpose it is necessary for the transmitter to add a 0 every five consecutive ones. If it is the receiver that detects the five consecutive ones, it will check the next bits as follow:

1. If next bit is a '0', it removes it to obtain the real sequence.

2. If the next two bits are '10', the receiver detects a flag.

3. If the next two bits are '11', the receiver detects an error.

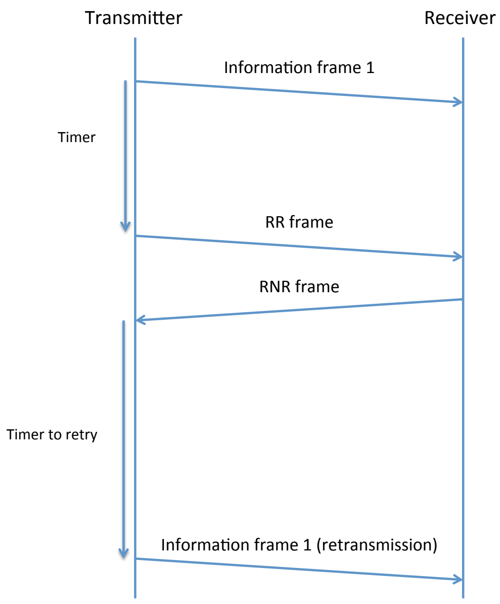


Figure 3.5 – *Transmitter sends and it does not receive any respond. The RR frame is answer by a RNR frame.*

3.1.1.2.3 Address field

This field is used to identify the source and the destination with their directions. It also has, optionally, two Layer 2 repeater sub-fields. In the following it is considered non-repeaters mode. Table 3.4 and 3.5 shows the address field and its encoding.

Table 3.4 – *Non-repeater address fiel encoding. Adapted from Ref [5]*

Address field	
Destination Address Subfield	Source Address Subfield
A1 A2 A3 A4 A5 A6 A7	A8 A9 A10 A11 A12 A13 A14

Table 3.5 – *Field encoding. Adapted from Ref [11]*

A1	call sign	A8	call sign
A2	call sign	A9	call sign
A3	call sign	A10	call sign
A4	call sign	A11	call sign
A5	call sign	A12	call sign
A6	call sign	A13	call sign
A7	CRRSSID0	A14	CRRSSID1

The 'C' bit is the response or command bit of an AX.25 frame. The 'R' bits are reserved and used in individual networks.

The address field consist of a callsign and a SSID. The callsign is formed by 6 upper-case alpha and numeric ASCII characters. SSID is a four-bit integer that identifies between station using the same callsign.

The LSB of each byte (extension bit) is set to zero to extend the address field one byte. A one indicates the end of the address field, on the other hand the zero means that the next byte contains more addressing information.

The SSID and the 'C' bit is contained in the SSID byte in each byte sub-field, at the end, it means at byte A7 and byte A14. The version protocol is identified by the 'C' bit. For the older version protocol both 'C' bits are set to zero. If it is the new version, one of the bit is set to zero and the other to one. Which is zero and which is one depends on the kind of frame (command or respond). This encoding is shown in the table 3.6.

Table 3.6 – *Command and respond encoding. Adapted from Ref: [5]*

Frame Type	SSID C-Bit	Source SSID C-bit
Previous Versions	0	0
Command (V.2)	1	0
Response (V.2)	0	1
Previous Versions	1	1

3.1.1.2.4 Control field

The control field identifies the type of frame, (I, S or U mentioned on 3.1.1.2.1). It is one or two bytes long. The control field can use sequence numbers. This way it will maintain link integrity. This sequence numbers can have two formats, 3-bits (modulo 8) or 7-bits (module 128) integer. The formats and the structure of each format is shown in the tables 3.7 and 3.8.

Table 3.7 – *Control field with module 8 sequence number. Adapted from Ref: [11]*

Frame Type	Control field bit			
	7 6 5	4	3 2 1 0	
I frame	N(R)	P	N(S)	0
S frame	N(R)	P/F	S S 0 1	
U frame	M M M	P/F	M M 1 1	

Table 3.8 – *Control field with module 128 sequence number. Adapted from Ref: [11]*

Frame Type	Control field bit			
	15 14 13 12 11 10 9	8	7 6 5 4 3 2 1 0	
I frame	N(R)	P	N(S)	0
S frame	N(R)	P/F	0 0 0 0 S S 0 1	

The I frame only contains the receiver and the transmitter sequence number. This can be seen in the tables 3.7 and 3.8.

There are different encodings for the S frame, as shown in the table 3.9 and 3.10.

Table 3.9 – *S frame, module 8.Adapted from Ref: [5]*

Frame Type	Control field bit			
	7 6 5	4	3 2	1 0
Receiver Ready	N(R)	P/F	0 0	0 1
Receiver Not Ready	N(R)	P/F	0 1	0 1
Reject	N(R)	P/F	1 0	0 1
Selective Reject	N(R)	P/F	1 1	0 1

Table 3.10 – *S frame, module 128.Adapted from Ref: [5]*

Control Field Type	Control field bit		
	15 14 13 12 11 10 9	8	7 6 5 4 3 2 1 0
Receiver Ready	N(R)	P/F	0 0 0 0 0 0 0 1
Receiver Not Ready	N(R)	P/F	0 0 0 0 0 1 0 1
Reject	N(R)	P/F	0 0 0 0 1 0 0 1
Selective Reject	N(R)	P/F	0 0 0 0 1 1 0 1

The S frame can contain any of the following telecommands:

- Receiver Ready: this command is sent if the system is ready to receive more information frames. It clear the condition sent by an RNR. It can also be used to request the status of a receiver, with the P-bit set to one.
- Receiver Not Ready: this command is sent indicating if the system cannot receive more information frames because the system is busy. The condition will be cleared if one of the following telecommands is sent: UA, RR, REJ or SABM(E) frame. It can also be used to request the status of a receiver, with the P-bit set to one.
- Reject: this command requests the re-transmission of information frames starting from the message with a sequence number indicated in the same message. It only can be sent once, in each direction, at a time. The condition will be cleared if the receiver receives the correct frames. It can also be used to request the status of a receiver, with the P-bit set to one.
- Selective Reject: this command requests just the re-transmission of an information frame by the sequence number. The condition will be cleared if the receiver receives the correct frame.

There are many different kinds of encoding on the U frame control. All of them are

Table 3.11 – *U frame encoding. Adapted from Ref: [11]*

Frame Type	Type	Control field bit			
		7 6 5	4	3 2	1 0
Set Async Balanced Mode Extended	Cmd	0 1 1	P	1 1	1 1
Set Async Balanced Mode	Cmd	0 0 1	P	1 1	1 1
Disconnect	Cmd	0 1 0	P	0 0	1 1
Disconnect Mode	Res	0 0 0	F	1 1	1 1
Unnumbered Acknowledge	Res	0 1 1	F	0 0	1 1
Frame Reject	Res	1 0 0	F	0 1	1 1
Unnumbered Information	Either	0 0 0	P/F	0 0	1 1
Exchange Identification	Either	1 0 1	P/F	1 1	1 1
Test	Either	1 1 1	P/F	0 0	1 1

extensively explained in the Reference [5] from page 22 to page 30. In the table 3.11 are shown the frame types.

3.1.1.2.5 PID field

The PID field is used to identify the protocol running. In the table 3.12 is exposed the PID encoding.

3.1.1.2.6 Info field

The info field carries the main information, the data that the user want to send. This field has a maximum length of 256 bytes. This field is only used in the following frames:

- I frame.
- UI frame.
- XID frame.
- TEST frame.
- FRMR frame.

3.1.1.2.7 FCS field

The FCS field carries the 8 bytes number calculated by both the sender and the receiver. This field make possible to check if the frame was corrupted in the transmission. The FCS is calculated as the HDLC reference document, ISO 3309, recommends. The sequence calculated is a CRC calculation, which use a generator polynomial of $G(x) = x^{16}+x^{12}+x^5+1$. The calculation is done over all the field except the flags.

Table 3.12 – *PID enconding. Adapted from Ref: [5]*

HEX	Bin Data	Translation
**	yy01yyyy	AX.25 layer 3 implemented.
**	yy10yyyy	AX.25 layer 3 implemented.
0x01	00000001	ISO 8208/CCITT X.25 PLP
0x06	00000110	Compressed TCP/IP packet. Van Jacobson (RFC 1144)
0x06	00000111	Uncompressed TCP/IP packet. Van Jacobson (RFC 1144)
0x08	00001000	Segmentation fragment
0xC3	11000011	TEXNET datagram protocol
0xC4	11000100	Link Quality Protocol
0xCA	11001010	Appletalk
0xCB	11001011	Appletalk ARP
0xCC	11001100	ARPA Internet Protocol
0xCD	11001101	ARPA Address resolution
0xCE	11001110	FlexNet
0xCF	11001111	NET/ROM
0xF0	11110000	No layer 3 protocol implemented.
0xFF	11111111	Escape character. Next octet contains more Level 3 protocol information.
Escape character. Next octet contains more Level 3 protocol information.	00001000	

3.1.1.2.8 Invalid frames

There are three scenarios where the AX25 protocol is considered an invalid frame. Those are the following:

1. The frame does not have an integer number of bytes.
2. The frame has less than 136 bits (7 bytes) length, including the flags.
3. The frame is not delimit by opening and closing flags.

3.1.1.3 FX25 (AX25 with FEC)

FX25 arises as an advance of AX25. In short, is the same protocol but adding to every frame the FEC. FX25 was created to avoid the main problem of AX25. In AX25 a single bit error can cause the loss of the whole frame, and it can also cause the loss of two frames. The FX25 frame was invented by Stensat group in 2005, to solve the problem adding the FEC to the AX25 frame. One of the most interesting characteristics is that the FX25 is compatible with AX25. This means that if a station is not prepare to decode FX25 frames is still able to decode the AX25 frame inside the FX25 frame, and the station will work just like if the FX25 frame was a AX25 frame. This is thanks to the order of the FX25 fields.

3.1.1.3.1 FX25 general description

Table 3.13 – *FX25 frame. Adapted from Ref: [9]*

Preamble	Correlation Tag	AX25 Packet Start	AX25 Packet Body	AX25 Packet FCS	AX25 Packet End	Padding	FEC Check Symbols	Postamble

Table 3.13 shows the FX25 disposition. The FEC code block goes from the AX25 Packet Start to the FEC check Symbols, both included.

FEC code block

This block represent the data are going to be included in the FEC algorithm. The size of this block will depend on the implementation of the FEC.

Preamble field

The first field, the preamble field, is a field that delimits the FX25 frames. It always has the byte 0x7E, which in bits is 1111110. This field makes possible to differentiate between two different FX25 frames. In fact, the preamble field works as the flag field in AX25.

In some occasions it will be necessary to have additional bytes, this will depend on the receiver and its characteristics and also it will depend on the transmitted line rate. For that reason it is recommended to transmit a minimum of 4 bytes with the sequence 0x7E7E7E7E. The original protocol does not explain more about this topic.

If the preamble field is used with the correlation tag, it can be useful to determine byte alignment.

Correlation tag field

The correlation tag field has 64 bits. It identifies the start of the frame and the FEC algorithm applied. The position of this field it is just before the from the FEC code block. This is done to ensure a correct correlation in every occasion, even though there is a channel error. This allows to identify the start of a frame even if there are errors on the correlation tag.

Gold Codes are used to generate the tags, those gives high autocorrelation and low cross correlation. This codes are used because they are simple to generate. For more detail regarding the code generation in reference [9].

Padding field

To be allowed to implement the FEC is necessary the padding field, this is because, as it will be explained at section 3.1.1.3.2, it is necessary to have two bytes of payload to implement one byte of FEC. This is because the FEC of one byte is half a byte (a nibble). So if we want to represent the bytes we need complete bytes, not nibbles. For that reason it is necessary an even number of bytes of the payload. If there are not the padding will have a byte. For all those reason the padding field does not carry any data and it is discarded.

The data inserted on the padding field will depend on the implementation but it is recommended to use the same sequence applied on the preamble field, 0x7E. This is because it is identified as null frames by AX25.

FEC check symbol field

The position of the field is very important. It is necessary to be correctly located for the FEC to function correctly. If the bit stuffing algorithm adds or removes one bit and a bit error occurs, the FEC correction will not succeed.

Postamble field

This field is a sequences of 0X7E bytes. This field is used to separate the end of the FX25 frame and the transmitter un-key event.

Similar to the preamble, it is recommended a minimum of 2 bytes to transmit, 0x7E7E. Also, like in the preamble, it can be needed additional bytes; this will depend on the receiver characteristics and the transmitted line rate.

3.1.1.3.2 FEC algorithm

The FEC is an error correction mechanism.

It introduces 4 bits for every 8 bits of data, 4 bits for every byte. This allows correcting any single bit error in each block of 12 received bits (8 bits of data and 4 bits of FEC). To carry out the FEC it is necessary the FEC matrix, as shown in table 3.14.

Table 3.14 – *FEC codification matrix.*

FEC code	**Parity bits**
11101100	1000
11010011	0100
10111010	0010
01110101	0001

Then when a byte is going to be sent it is logically ANDed with the four FEC codes bits. From the result it is calculated the parity bit: as there are 4 FEC codes there will be 4 parity bits. Those parity bits are added to the 8 data bits, which make a payload of 12 bits. And this is the sequence that will be send. When this sequence is received it is logically ANDed with the 12 bit word from table 3.14. That account forms four new words with 12 bits each one. The parity of those words have to be analyzed. If all the parities bits are 0, there are not any flip bit. If one of more of the parity bits are 1 there have been some flip. To locate the error it is necessary to make the last calculation. It is necessary to AND all the FEC codes that form the table 3.14. If there was a parity 0 the FEC code associated to the word with that parity 0 is inverted, the other are ANDed without invertion. In tables 3.15, 3.16 and 3.17 is shown the complete FEC process.

Table 3.15 – *FEC generation example*

FEC matrix	11101100	11010011	10111010	01110101
Original bits	11010110	11010110	11010110	11010110
AND result	11000100	11010010	10010010	01010100
Parity	1	0	1	1

Table 3.16 – *Example of a detection of bit flip*

FEC matrix	11101100 1000	11010011 0100	10111010 0010	01110101 0001
Received	11010111 1011	11010111 1011	11010111 1011	11010111 1011
AND result	11000100 1000	11010011 0000	10010010 0010	01010101 0001
Parity	0	1	0	1

Table 3.17 – *Example of the location of a bit flipped*

Row number	**Bit sequence**
1 inverted	00010011 0111
2	11010011 0100
3 inverted	01000101 1101
4	01110101 0001
AND	00000001 0000

The example 8 bits message is 11010110. In the table 3.15 is calculated the FEC that will be sent. The complete sequence that it is sent is 11010110 1011. In the example 1 bit has been flipped, and the receiver receives 11010111 1011. In the table 3.16 it is detected that there was an error. And in the table 3.17 the bit flipped is located. As we can see the FEC located the bit flip in its position 00000001 0000.

3.1.1.3.3 FX25 improvement

In section 3.2.2 it will be explain why it is used this mode as a solution. But this FX25 has been changed slightly.

FX25 packet is basically formed by a AX25 packet, with its flags, and then it is added the FEC and the FX25 flags. In this way a receptor that is only able to read AX25 will be able to extract the AX25 packet of the FX25 packet, this receptor can work with a FX25 transmitter. But the default of this mode is that we are using the double bytes of flags, and that the CRC of the AX25 is not useful in the FX25 packet. In the case of this project we are going to use both receptor and transmitter that are able to work with the protocol that is build in this project. For that reason it is not necessary this kind of compatibility. For that reason I remake the FX25 packet to a new one that has the form expressed in the table 3.18.

Table 3.18 – *FEC packet form.*

Flag	Destination	Source	Path	Control	PID	Payload	FEC	CRC	Flag

In this way we are only using two flags, not 4, and the CRC is done to the whole packet, not only to the packet without the FEC.

This new packet is simpler than the FX25 packet, this packet is like a AX25 packet with the FEC at the end of the payload. In this new packet the Correlation Tag field, that could be included in futures improvements, and the padding field have been removed . But the padding field is included in every FEC field as it will be explained in chapter 4: it would be as if the padding where located after the FEC field instead of before it but in fact it is included in the FEC field and analyzed with it.

3.2 Analysis of possible solutions to requirements

In this section three possible solutions commented in section 3.1 are going to be analyzed. Whit this purpose in mind, first, we are going to do an efficiency comparison.

3.2.1 Efficiency comparison

When we talk about a protocol is important to be aware of it efficiency. It will depend on the overload that it is needed to send the user data. This means that for the same data different protocols will have a throughput or another.

3.2.1.1 AX25 efficiency

We first analice the AX25 with CRC efficiency. As we have seen in this section the whole packet has a variable length that does not depend on the information field.

In the table 3.19 there are two of the main cases that can happen. In the minimum length case there is a situation in which the length of all fields is the minimum, and the total is 20 bytes length, that will be added to the payload (information field) length. In the other hand

Table 3.19 – *AX25 overload*

Field	**Minimum length**	**Maximum length**
First flag	1 Byte	1 Byte
Address	14 Bytes	70 Bytes
Control	1 Byte	1 Byte
PID	1 Byte	1 Byte
FCS	2 Bytes	2 Bytes
Second flag	1 Byte	1 Byte

we have the opposite situation, where all the fields has it maximum length, that makes 76 bytes of overload. In the maximum length case the address field is including the maximum digipeaters directions, see [10].

Knowing the efficiency as the division of the real data between the throughput it is sent to transmit the real data:

$Efficiency = \frac{Payload}{Throughput}\%$

The maximum efficiency would be 100%.

For the minimum length we would have the efficiency like this:

$Efficiency = \frac{Payload}{Payload+20}x100\%$

And for the maximum length the efficiency is:

$Efficiency = \frac{Payload}{Payload+76}x100\%$

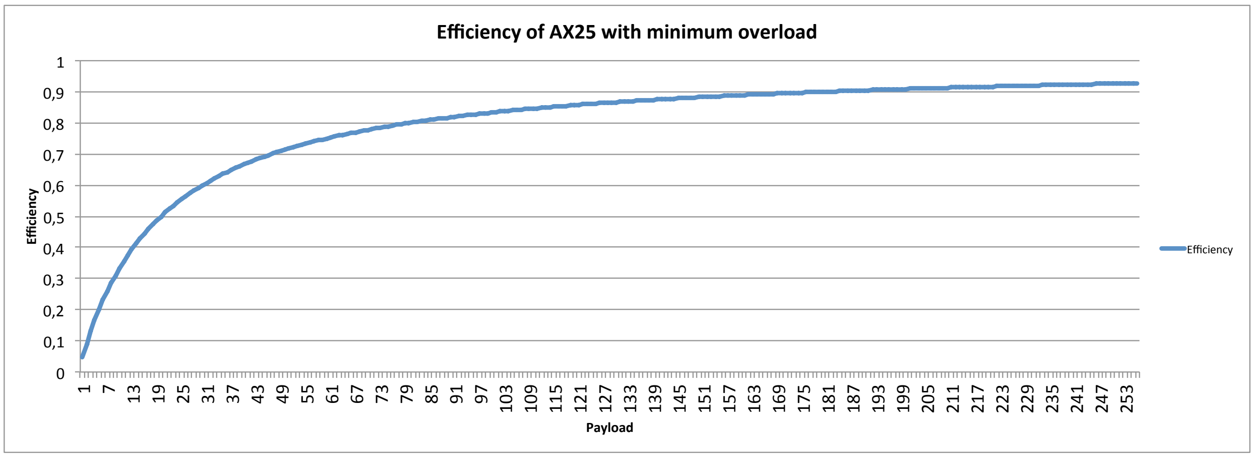


Figure 3.6 – *Efficiency of AX25 with minimum overload*

In the graphics 3.6 and 3.7 the situation is shown in a range from 1 to 256 bytes of payload (from the minimum to the maximum payload). It is notable de difference between them, where minimum overload is more efficient, because we need less bytes to send the same message. In the comparison section 3.2.1.3 we will compare the three options in the

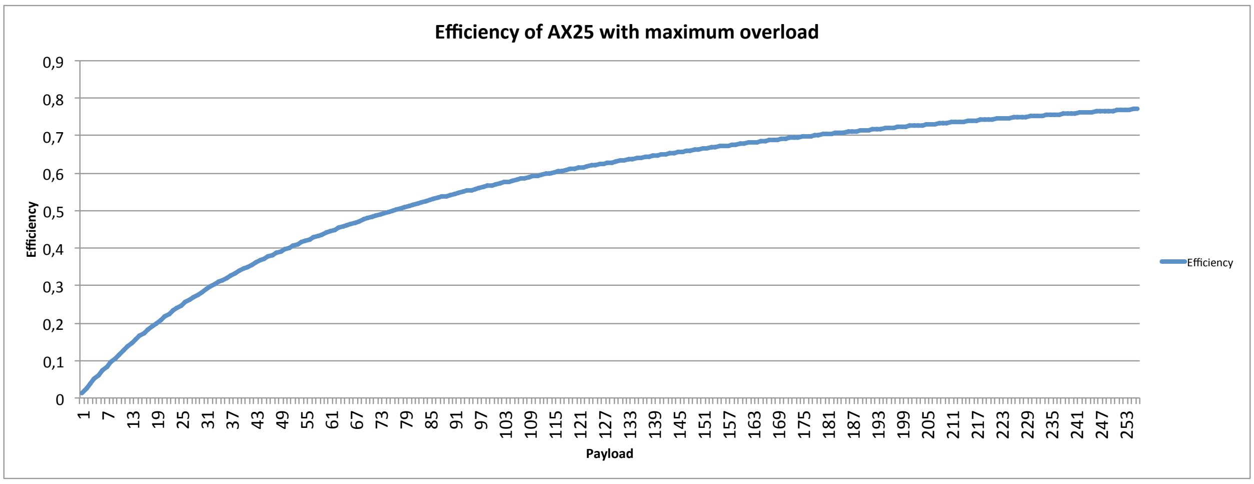


Figure 3.7 – *Efficiency of AX25 with maximum overload*

maximum efficiency mode.

3.2.1.2 FX25 efficiency

Now the FX25 is going to be analyzed. FX25 is going to be analyzed with the changes already explained in section 3.1.

As it was done with FX25 it is calculated the overload of FX25:

Table 3.20 – *FX25 overload*

Field	**Minimum length**	**Maximum length**
First flag	1 Byte	1 Byte
Address	14 Bytes	70 Bytes
Control	1 Byte	1 Byte
PID	1 Byte	1 Byte
CRC	2 Bytes	2 Bytes
Second flag	1 Byte	1 Byte
FEC	payload/2 Byte	payload/2 Byte

In the table 3.20 there are two of the main cases that can happen, where the FEC field length is a variable depending on the payload. In the minimum length case the total is 20 plus payload/2 bytes, that will be added to the payload (information field) length. In the other situation the total overload of the packet it is 76 plus payload/2 bytes of overload. In the maximum length case the address field is including the maximum digipeaters directions, see [10].

Knowing the efficiency like the division of the real data between the throughput it is sent to transmit the real data:

$Efficiency = \frac{Payload}{Throughput}\%$

Where the maximum efficiency would be 100%. For the minimum length we would have the efficiency like this:

$Efficiency = \frac{Payload}{Payload+20+\frac{Payload}{2}}x100\%$

And for the maximum length the efficiency is:

$Efficiency = \frac{Payload}{Payload+76+\frac{Payload}{2}}x100\%$

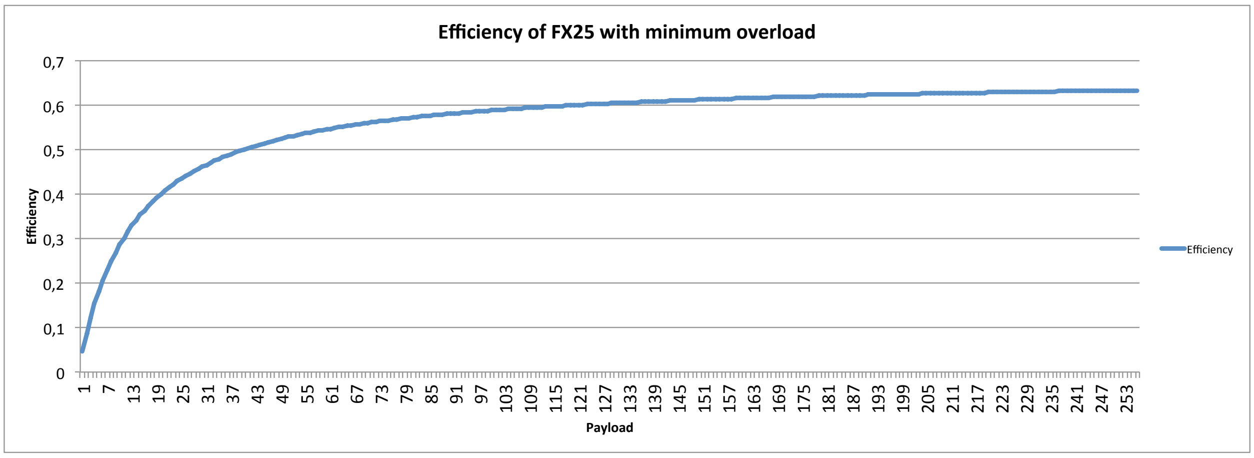


Figure 3.8 – *Efficiency of FX25 with minimum overload*

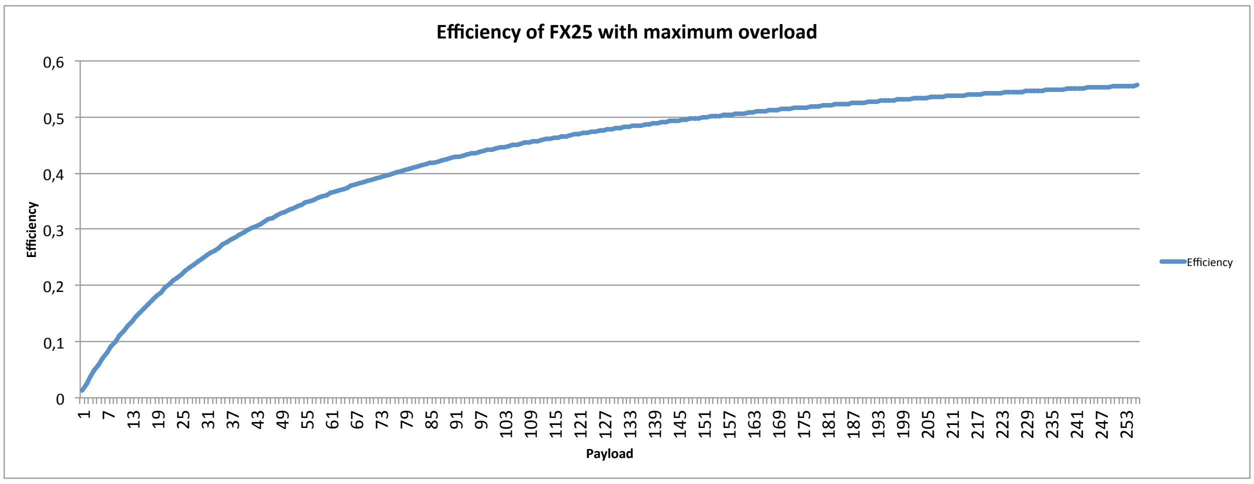


Figure 3.9 – *Efficiency of FX25 with maximum overload*

In the graphics 3.8 and 3.9 the situation is shown in a range from 1 to 256 bytes of payload (from the minimum to the maximum payload). Like with the AX25, it is notable de difference between efficiencies, where minimum overload is more efficient, because we need less bytes to send the same message.

3.2.1.3 Efficiency comparison between AX25 and FX25

In this section the efficiency between AX25 and FX25 is going to be compare. For this purpose it is going to be used the efficiency calculated in sections 3.2.1.1 and 3.2.1.2. Joining both graphics in only one in order to campare easily.

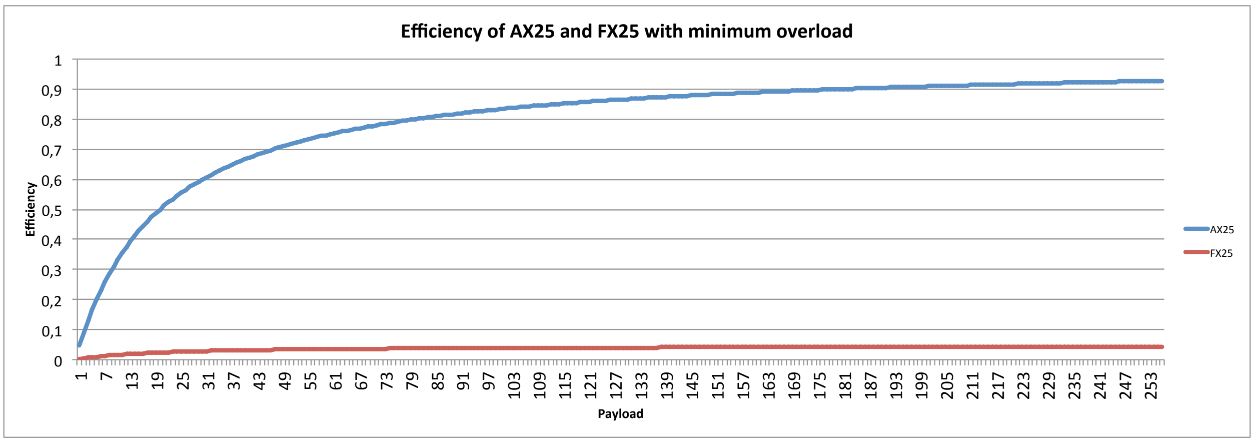


Figure 3.10 – *Efficiency of AX25 and FX25 with minimum overload*

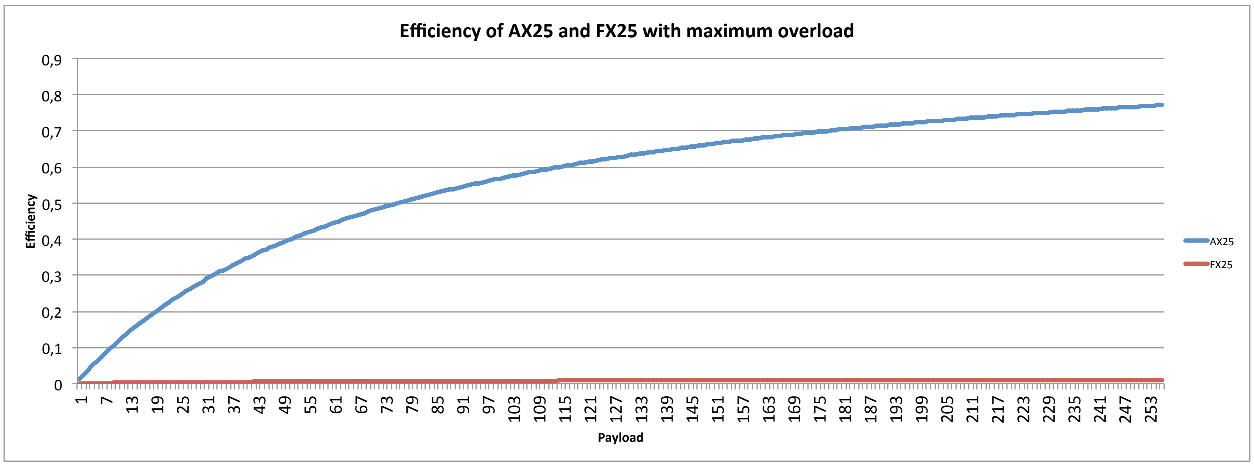


Figure 3.11 – *Efficiency of AX25 and FX25 with maximum overload*

In both graphics it can be observed clearly that the efficiency of the AX25 is better than the one of FX25. This has sense, because the throughput with FX25 is much bigger than the one of AX25. To compare both throughput an specific case is going to be analyzed, one that is the most usual. When the overload is minimum and the payload is maximum, because when transmitter is sending image the most of the packages has the maximum length (256 bytes). The following functions show the throughput of them in this case:

$$AX25\ throughput = Payload + overload = 256 + 20 = 276bytes$$

$$FX25\ throughput = Payload + overload + \frac{Payload}{2} = 256 + 20 + \frac{256}{2} = 404\ bytes$$

The throughput necessary to send the same data with FX25 than with AX25 it is 1.57 bigger than the AX25 throughput.

But here it has only been analyzed the throughput in an ideal scenario, with no noise. The AX25 only works properly without noise, because it can not recover errors. This situation only happens when the satellite it is situated in the closest point to the ground computer. This would happen only a few second in every round. For that reason FX25 would be necessary. With FX25 can be recovered 1 bit of every 8 bits. The FX25 throughput is very big for a little data, but in cases where the AX25 can not transmit FX25 is working, even though the efficiency is low.

3.2.2 Solution choice

The main benefit of the original AX25 is that it has already been used in a lot of cubesat projects that have worked or are working properly. Is the standard solution so we have the certainty that it can work. Some examples of other cubesat projects using AX25 protocol are the project BLUEsat of University of New South Wales (Sydney, Australia), AAU-Cubesat of Aalborg University, CubeSTAR of University of Oslo in Norway or AMSAT-UK (reference [1]).

Having into account the last section 3.2.1.3, I came to the conclusion of using AX25 and FX25, depending on the scenario. This means that the program that has been developed for this project can easily change between FX25 and AX25. This kind of hybrid protocol was also proposed by Jan-Hielke Le Roux, see [11].

CHAPTER

4 SYSTEM DESIGN

In this chapter I will explain the system design, how the program is implemented. As this program is based in other one, this chapter starts explaining how the first program work. This is done in section 4.1. Then, in section 4.2, all the improvements I made to the protocol to achieve the objectives set for this project are explained.

4.1 Reverse engineering

To manage the emissions reception of audio via the soundcard there is a class called Soundcard. This has a constructor with which the name of the input and output soundcard it is saved, for later used in the same class. Then there is a method to send audio (*transmit)*, a method to receiv audio (*receive*) via the soundcard, and also a method that finds the name of the different soundcards the computer can have.

4.1.1 Transmission

For transmitting sound the library *javax.sound.sampled.SourceDataLine* is used. With this library it is created a *SourceDataLine*. And using the method of this kind of variable the transmission mode is started. The methods used are *flush*, *start*, *write*, *drain* and finally *stop*. The method *flush* it is used to delete the queued data of the line. The method *start* allows a line to engage in data I/O. The method *write* it is used to transmit the samples. The method *drain* waits until all data has been sent. And the method *stop* stops the line.

The samples that are going to be sent are created with the class *Afsk1200Modulator*. With the method *prepareToTransmit* it is copied to a variable the whole packet, that it is in byte array. Then it is used the method *getSample*, this method it is in the class *Afsk1200Modulator* too. This method copied every byte of the byte array of the whole packet in an integer. To do this it is used other method called *byteToSymbols*. The method *byteToSymbols* it is used for every byte of the byte array. This method used the method *generateSymbolSamples*. In this method while the variable that has the symbol phase it is less than $2\text{x}\pi$ the counter adds in one. Besides the increment of the counter it happens more things in this while condition. In every interaction it is added to the symbol phase the variable *phase_inc_symbol* that is equal to 2πx1200xsample rate. The *getSample* method returns the counter calculated in *generateSymbolSamples*.

4.1.2 Reception

For receiving audio it is used the library *javax.sound.sampled.TargetDataLine*, to make a *TargetDataLine* variable. First of all it is used the method *flush*, used to delete the queued data of the line. Then with *while(true)* the method start listening using the method *read* of the *TargetDataLine* variable.

Then it is necessary that it listens the AX25 packet. This is done with a method called *addSamplesPrivate* in the class *Afsk1200MultiDemodulator*. This method sums the samples collected with *TargetDataLine*.

4.1.3 Class packet

Sivan Toledo created the class Packet to make AX25 packets. In this class there are two constructors to make a packet, in the first one the input is a byte array with the whole packet except the CRC. The second and most used constructor has as input every field of an AX25 packet except the CRC that it is included in the constructor. What this constructor does it is to join all the different fields in a new byte array, calculating the final size of the packet, and with this size it calculates the CRC and adds it to the byte array, that now is a complete AX25 packet.

The most interesting constructor is the second one, that makes the whole packet. In this constructor it is firstly calculated the length of the packet, without the CRC. This is the sum of the length of the destination, the length of the source, the path length, the two bytes of the control plus the PID and the payload length: $Packet\ length = 7 + 7 + 7 * path\ length + 2 + payload\ length$

For the first three fields (destination, source and path) it is used a method called *addCall* that adds these fields, that are in an string, to the byte array. For the control field and the PID, that are input as integer, it is used a casting to byte and they are added directly to the final byte array. The last field, the payload field, is input as a byte array and it is needed a for loop to pass every single byte of the payload to the packet array.

In this class packet there is also a method used to show the packet to the user. This method received a byte array and it finds the different fields of the packet and shows them separately. To find the different fields it used a pointer, called offset, that goes through the byte array. The destination and source always have the same length, 7 bytes, so they are easy to locate in the first 14 bytes. Then it is located the path, to find the end of it it is used the while condition, if the offset (the pointer) is less than the size of the packet, and more important the byte in the packet array in the position offset - 1 anded to the byte 0x01 it is equal to 0 it means that there is an other path element in the packet array. After finding the last path element the offset is incremented in 2, to skip the control field and the PID field. Then the remaining bytes except two (CRC) are the payload.

4.2 Improvement of the protocol

In this section it is explained how the protocol was improved to fulfill the requirements of this project. This was done with the programming language Java, using the references [6] and [8].

4.2.1 Class packet improvement

In the class packet I implemented different methods in order to help to work with the packets. These methods extract the different fields of a packet. These methods are based in the method implemented by Sivan Toledo called *format*. With this method it is possible to find only the field that it is needed to be analyzed. There is one method for every field of an AX25 packet.

The methods *destination*, *source* and *path* returns an string with its corresponding fields. And the method *payload* returns a byte array with the payload of the packet. As input those methods has a byte array with the complete packet.

4.2.2 Packet receiver

The method *packetReceiver* is a method that analyzes and then acts accordingly with every packet arrived. This method is called every time a packet has arrived.

The method is the following:

```
public void packetReceiver(byte[] bytes) {
    byte[] withoutCRC= Arrays.copyOf(bytes, bytes.length-FCS_LENGTH); //FCS_LENGTH=2

    packetComplete[r]=bytes;

    byte[] payload=null;


    if(!fecMode){
      payload= extractPayload(bytes);
    }
    if(fecMode){
      payload= extractFecPayload(bytes);
      byte[] fec=extractFec(bytes);    //fec recibido
```

```
        payload=Fec.payloadFecSolved(payload, fec);        }

       payloadComplete[r]=payload;

      if(ready){
        analyzePayload(payload);
      }

      //If an image is arriving it is calculated the number
      // of packets and it will save it when all them arrive
      if (imageArriving){

        int rest=0;

        if((MESSAGE_LENGTH%256)!=0){
          rest=1;
        }
        int packetNumber=rest+MESSAGE_LENGTH/256;
        int aux1=packetNumber+startOfImage−1−r;
        String packetsRemained= ""+aux1;
        dashboard.remainingPackages.setText(packetsRemained); //we inform about the number of
          packages remaining
        int aux2=(int)((double)aux1*3.4);
        String secondRemained=""+aux2;
        dashboard.remainingTime.setText(secondRemained+"s"); //we inform about the seconds
          remaining to finish the image issue
        saveIncompleteImage(); //this method save the bytes on a new image to go showing the
          image arrived
        if(r==(packetNumber+startOfImage−1)){
          saveImage(packetNumber);
          ready=true;
          sendACK("IMAGE");// If the image is sent and saved the ACK is sent
        }
        counter++;
      }

      if(longArriving){

        int rest=0;
        if((MESSAGE_LENGTH%256)!=0){
          rest=1;
        }
        int packetNumber=rest+MESSAGE_LENGTH/256;

        if(r==(packetNumber+startOfLong)){
          saveLong(packetNumber);
          ready=true;
          sendACK("LONG");  // If the long message is sent and saved the ACK is sent
        }
      }

      r++;

      return;
  }
```

This method has as input the complete packet.

It firstly takes out the CRC, this is at the last two bytes of the packet, and it is saved in a matrix were every row is a complete packet.

Then the FEC mode is checked. If the FEC mode is disactivated, this means fecMode=false, a simple method is used that calls the class packet where there is a method that extract the payload, see 4.2.1. If the FEC mode is activated, this means fecMode=true, the program extracts the payload and the FEC and the method *payloadFecSolved* used to solve all the possible errors the payload can contain and the FEC can solved. The method *payloadFecSolved()* is explained in section 4.2.5.2.

After this, whatever the FEC mode was, we have a byte array called payload with the payload arrived. If FEC mode is activated this payload is solved. This payload is saved in a matrix where every row in it is the payload of a packet.

Then if ready is true it is analyzed the payload, to know what kind of packet has arrived.

The ready variable is true if we are not receiving a long message or a image message. This is done because is not necessary to analyzed those packets we already know form part of a bigger message, if it is an image or a long message until it finishes arriving it does not make sense to continue analyzing the packages, because what is arriving are not commands that we have to analyze but data that we have to process. The analyzing of the payload is done with the method *analyzePayload* that is explained in section 4.2.3. In this method it is checked what to do with the information on this payload.

If in the last packet it was detected that is coming an image, the image arriving boolean has been turned on and ready has been turned off. In this case every packet arrived until the image has been completely arrived will be saved to rebuilt the image. And every five packets the parts of the image already arrived is rebuilt. This is explained in section 4.2.4.2.

If in the last packet it was detected that is coming a long message, this means that the message was bigger than 256 bytes and it has been splitted to send it, the long arriving boolean has been turned on and ready has been turned off. In this case every packet arrived until the long message has been completely arrived will be saved to rebuilt the whole message. This works exactly the same as the image, but without saving the image but saving a text file with the method *saveLong*, that will save the whole message in a file with the directory that the user identified before it started the receiver.

Finally the variable r is added one, because it represents the number of packets arrived. This is a class variable. And then the method returns to the method that called it.

4.2.3 Sending, receiving and analysis of new commands

In this section we are going to explain the kind of commands that have been developed for this project. This commands solve the requirements related only to this project. Those requirements are:

- To send and receive images.
- To turn on and off a led that will be situated on the satellite.
- To request data about the satellite and to receive a response

Every command has a length of 7 bytes. For the commands that has less than 7 letters (each letter is a byte) they are filled with zeros.

In order to explain all the commands it has been done four groups of commands: short commands, long commands , answer commands and AX25 specific commands.

4.2.3.1 Short commands

LEDON

Ground sends this message to the satellite to turn on the led.

LEDOFF

Ground sends this message to the satellite to turn off the led.

TEMP

Ground sends this message to the satellite asking for the temperature. The temperature goes in kelvin degrees.

BAT

Ground sends this message to the satellite: asking for the battery. The battery is express with a percentage of the battery it has.

POS

Ground sends this message to the satellite asking for the position of the satellite.

The position is expressed as follows: latitude [deg], altitude [km], length [deg], azimuth [deg].

4.2.3.2 Long commands

IMAGE

If ground sends this message to the satellite this means that ground is asking for all the picture the satellite has made since the last time the satellite sent a picture.

When this command it is used by satellite it indicates the size of the next picture that will be sent. Because of in this way it is an answer to the ground petition this will have an ack command before the image command. Both commands and the length of the picture will go in the same packet.

LONG

This message is sent to warn the next message has been divided because it is too long (more than 256 Bytes). The payload of the packet with this command would contain the command and the length of the message.

DATA

If it is ground which uses this command it is asking for all the data the satellite has saved.

If it is satellite which sends this message to the ground computer it is used to send all the data it has saved. It is usually a long message; the command would be DATA000LONG000 and then the length. After that the packets with all the information will came.

4.2.3.3 Answer commands

ACK

This message is sent, for any of the two extremes, to confirm every packet received

ERROR This command is used for any kind of error that would be specified just before this command, in the same message.

4.2.3.4 AX25 specific commands

RR Receiver Ready: this command is sent if the system is ready to receive more information frames

RNR Receiver Not Ready: this command is sent indicating if the system cannot receive more information frames because the system is busy.

REJ Reject: this commands request the re-transmission of information frames starting from the message with a sequence number indicated in the message with the REJ.

SREJ Selective Reject: this command requests just the re-transmission of on information frame by the sequence number.

4.2.3.5 Short answer commands specifications

4.2.3.5.1 Possible answers from satellite

ACK

The message has arrived and the action or petition was carried out. In the case of LED$_ONandLED_OFF, justarriveACKfollowedbythecommandreceived.ForTEMP, BATandPOSitc$
ACK0000$TEMP$000273).

ERROR

The message has arrived but the action or petition could not be carried out.

No answer

When the timer finishes ground supposes the message did not arrived. The message could have arrived but the answer to ground did not arrive, for practical purposes this would be the same as if the message never arrived. Because if ground repeats the message and the led was already turned on nothing changes.

4.2.3.5.2 Possible answers from ground to the answers from the satellite

ACK

Ground does not do anything else.

ERROR

Ground send the command DATA, to find out what could have caused the error.

No answer

If this is the first, second or third time this message has been sent, ground repeats the message. Between every new attempt ground waits a fixed time before sending again. If there are more than three tries the packet is discard for ground, not attempting it again.

4.2.3.6 Long answer commands specifications

4.2.3.6.1 Possible answers from satellite

ACK

The message has arrived and the petition is caring out. This command is followed by the bytes of the information is coming. For the three cases first arrived a message with the length of the information that is coming and then all the information will come.

There is an exception with DATA: if the DATA is long, in the ACK it will indicates (ACK0000DATA000LONG000). In the particular case of the pictures every time a new picture is going to be sent, first a message ACK0000IMAGE00 plus the length it is sent, and then the image is sent.

ERROR

The message has arrived but the action or petition could not be carried out.

No answer

When the timer finishes ground supposes the message did not arrived. The message could have arrived but the answer to ground did not arrive, for practical purposes this would be the same as if the message never arrived. Because if ground repeats the message and the led was already turned on nothing changes.

4.2.3.6.2 Possible answers from ground to the answers from the satellite

ACK

Ground does not do anything else.

ERROR

Ground sends the command DATA, to find out what could have caused the error.

No answer

If this is the first, second or third time this message has been sent, ground repeats the message. Between every new attempt ground waits a fixed time before sending again. If there are more than three tries the packet is discard for ground, not attempting it again. After this the dashboard shows a message warning about the problem. Then the users can ask for the data or image again, if this petition was manually.

4.2.3.7 Method to send a long message

First of all the sender has to analyse the length of the message. Then the message will be sent split. In the program the packet is sent on bytes, as a byte array. The message is saved in another byte array, and every packet sent the sender adds 256 bytes of this byte array with the message, and a pointer is moved 256 positions on the message array. So the next packet sent will have the next 256 bytes of the message starting on the pointer position. Usually the message is not multiple of 256, so the program needs to know the rest of the message, that will be the last packet. What the program does to divide the length by 256 and adds plus 1 if the module (length % 256 in Java) is different to 0, what means that the length is not multiple of 256 so we need one more packet to send the whole message. Then the program will send all this packages adding in the payload all the bytes of the message that can fix until the message end.

The receiver first calculates the number of packages that are going to arrive, the same way the sender does. The receiver saves all the payloads, with a counter it manage the number of packages arrived and when this counter reaches the number of packages calculated stop saving the payloads in the same variable. If this message is an image the receiver saves the whole message in the location as a JPG image.

For this commands no answer is needed, except for the cases already described.

In the section 4.2.4, it will explained how this process is used in the program.

The way the commands are implemented in Java is very intuitive. It has been used a control structure (switch-case) where the payload arrived is analyzed to see if it contains any commands acting accordingly to it. There are a lot of factors that have to take into account.

4.2.3.8 Commands schematics

In this section all the theory already explained about the implementation of the commands is going to be illustrated with some schematics, to a better understanding.

The next two figures represent the commands *LEDON* (4.1) and *LEDOFF* (4.1).

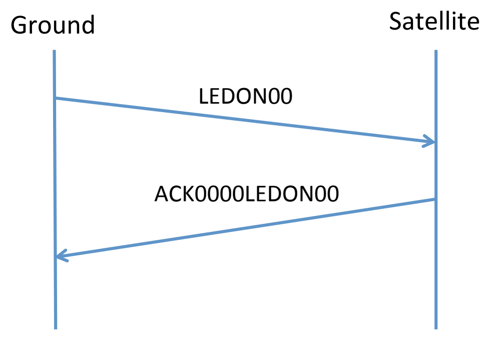


Figure 4.1 – *LEDON command representation.*

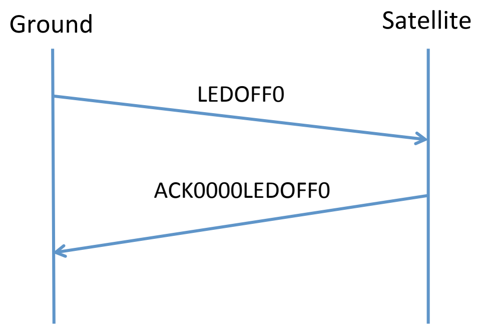


Figure 4.2 – *LEDOFF command representation.*

The next three figures represent the commands *TEMP* (4.3), *BAT* (4.4) and *POS* (4.5). In the *TEMP* schematic the KKK represents the three bytes intended to report the temperature, in the *BAT* schematic the KK represents the two bytes intended to report the percentage of battery and in the *POS* schematic the KKKKKKKK represent the eight bytes intended to report the position in latitude, altitude, longitude and azimuth.

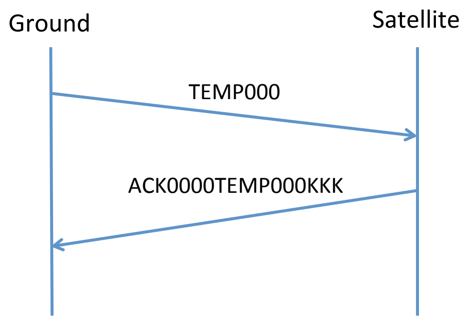


Figure 4.3 – *TEMP command representation.*

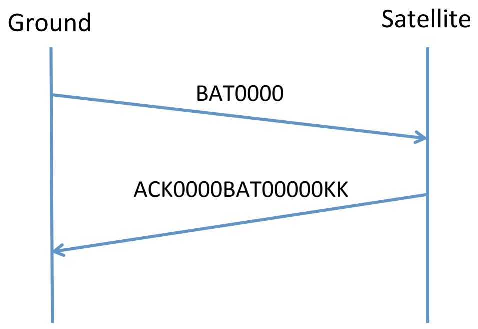


Figure 4.4 – *BAT command representation.*

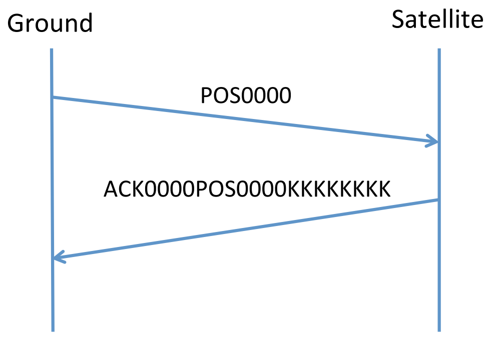


Figure 4.5 – *POS command representation.*

The next figure represents the *IMAGE* and the *DATA* commands. In the *DATA* case it is when the *DATA* has a bigger size than 256 bytes. In the *IMAGE* command case the BBBBB represents the five bytes intended to report the length of the image that it is going to be sent and in the *DATA* command case the BBBBB represents the five bytes intended to report the length of the long message that it is going to be sent.

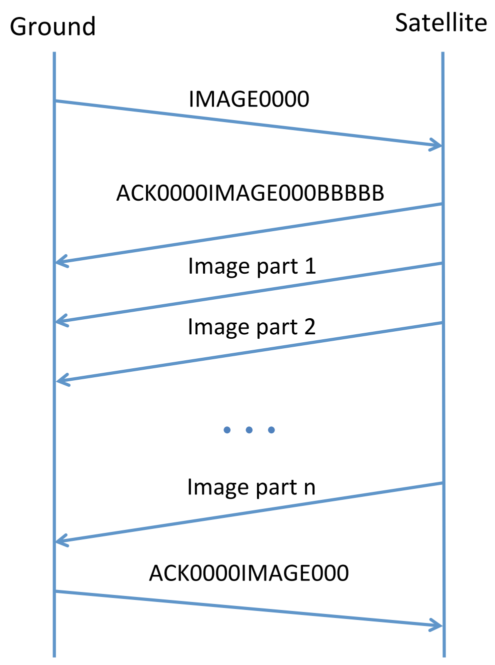


Figure 4.6 – *IMAGE command representation.*

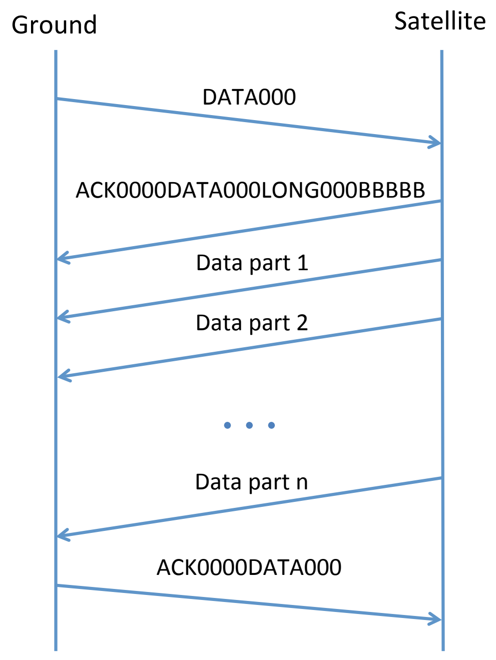


Figure 4.7 – *DATA command representation.*

4.2.4 Sending and receiving images through AX25 and FX25

The main purpose of this protocol is to be able to send and receive images. The main problem is that the images are too big to AX25 packets. Like it has been explained in chapter 3, the AX25 packet, like the FX25 packet, has a payload of 256 bytes length as maximum. For this reason it is necessary to split the images in parts of 256 bytes to send them. And then in the receiver it is necessary to be able to rebuild the image. To explain the whole process I am going to divide it then in two main parts, the code to receive an image in the receiver and the code to sendit in the transmitter.

4.2.4.1 Sending images through AX25, transmitter

Sending the image involves splitting the image and converting it, whatever the format it has, to bytes that can be sent. Knowing the directory the program find the image, then the image is passed to bytes, making a byte array with the image. With the byte array now we can have the length of it. With this length the first packet with the ack that confirms the image is going to be sent (to send an image the receiver has asked it before, how the commands work is explain in section 4.2.3) and the length of the image. Then it is sent as many packages as necessary, this means that if the length of the image out of 256 is not an integer we add one.

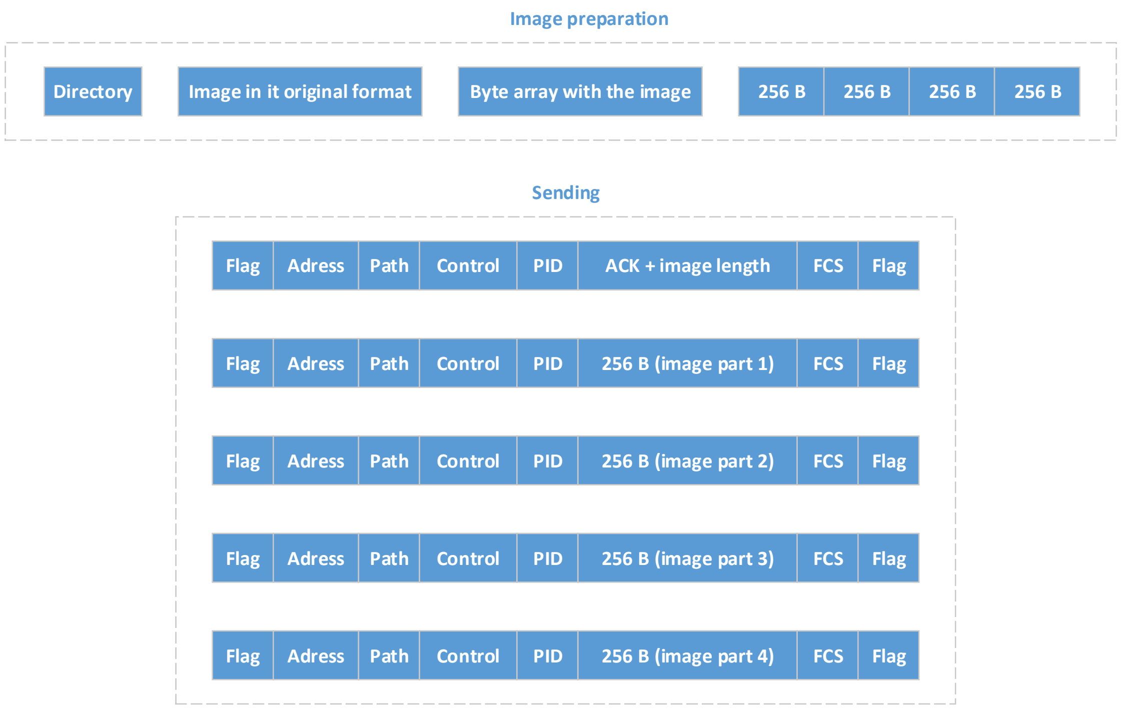


Figure 4.8 – *Sending an image, in the transmitter.*

An example of the whole process is shown in Figure 4.8. In this example it is going to be sent an image of 1024 bytes in 4 parts. In the image the length of the packets is not proportional to it real size. The first packet, when it is sent the ACK, is smallest with a big difference. This packet has a payload of approximately 20 bytes, it depends on how many bytes we need to express the length of the image.

Once analyzed how sending images works it is going to be analyzed the code to make this possible. The main method to send an image is *SendImage*:

```
1  private static void SendImage(String directory) {
2      byte[] array= ImageToByte (directory);
3      int length;
4      int counter=0;
5      int size=0; //size of the next part to be sent
6      int arrayLength=array.length;
7
8      String firstPayload="ACK0000IMAGE00"+arrayLength;
9
10     Packet firstPacket = new Packet("APRS",
11            callsign,
12            new String[] {"WIDE1-1", "WIDE2-2"},
13            Packet.AX25_CONTROL_APRS,
14            Packet.AX25_PROTOCOL_NO_LAYER_3,
15            firstPayload.getBytes());
16        sound(firstPacket, mod, ptt, multi, rate, t, p);
17
18     for(length= array.length; length>=0;length=length-256 ){
19       size=length;
20       if(length>=256){
21         size=256;
22       }
23       Packet packet;
24         packet=EmitLongMessages(array, counter, size, callsign );
25       counter=counter+256;     //bytes counter
26       sound(packet, mod, ptt, multi, rate, t, p);
27     }
28 }
```

This method has as input an String that will have the directory of the image. First of all the image is introduce , already translated to bytes, into a byte array. This is done with the method *ImageToByte* that will be explained later. Then, in line 8, it is created the payload of the first packet that is going to indicate the confirmation of the image sending and the length of it, this is a command defined in section 4.2.3. Then the packet is created , line 10 to 15, and it is emitted in line 16 with the method *sound*. Then we start to send the parts of the image.

With the *for* loop that start in line 18 we go through the byte array where the image is saved . The integer called *size* it is used to indicate the next packet size, and the integer *length* indicates the rest of the image that has not been sent. For that reason *size* equals length, and then if *size* is more than 256 it will be equals to 256. This means that if the rest of the image is now less than 256 the size of the payload is the size of the rest of the image. If not it will be needed more splits. Then the next packet is created , knowing the last byte sent (*counter*) and the length of the next part of the image that need to be sent. It is used the method *EmitLongMessages*. Now it is necessary to increased in 256 bytes the pointer that indicates the last byte sent (*counter*) Then it is used the method *sound()* to send this packet. All this is going to be repeated as many times as necessary to complete the whole emission of the image. The number of packets needed to send an image is the following:

$$Packets = \frac{Image.length}{256} + 1$$

To translate the image to a byte array the method *ImageToByte* is used :

```
public static byte[] Image_To_Byte (String route){
    try {
      File file = new File(route);
      FileInputStream fis = new FileInputStream(file);
      ByteArrayOutputStream bos = new ByteArrayOutputStream();
      byte[] buf = new byte[1024];
      try {
        for (int readNum; (readNum = fis.read(buf)) != -1;) {
          bos.write(buf, 0, readNum);
        }
        byte[] bytes = bos.toByteArray();
        return bytes;
      } catch (IOException ex) {
        Logger.getLogger(Principal.class.getName()).log(Level.SEVERE, null, ex);
      }
      byte[] bytes = bos.toByteArray();
      return bytes;
    }catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
    byte[] bytes2= null;
    return bytes2;
}
```

This method has as input the directory where the image is saved.

It is necessary to import previously the following: java.awt.image.BufferedImage, java.io.ByteArrayInputStream, java.io.ByteArrayOutputStream, java.io.File, java.io.FileInputStream, java.io.FileNotFoundException, java.io.IOException, java.util.logging.Level, java.util.logging.Logger. This libraries contains the necessary functions used to pass the image to bytes. This method is based in a method taken from [2].

To create the packet of a part of the image the method *EmitLongMessages* is used:

```
public static Packet EmitLongMessages(byte[] bytes, int pointer, int size, String callsign){
    Packet packet = new Packet("APRS",
        callsign,
        new String[] {"WIDE1-1", "WIDE2-2"},
        Packet.AX25_CONTROL_APRS,
        Packet.AX25_PROTOCOL_NO_LAYER_3,
        Split(bytes, pointer, size));
    return packet;
}
```

This method has as input a complete byte array that can not be sent in only one packet, an integer pointer that indicates the last byte sent, the size of the payload of this packet and then an String with the callsign. The output is a packet that can be sent with the method *sound*.

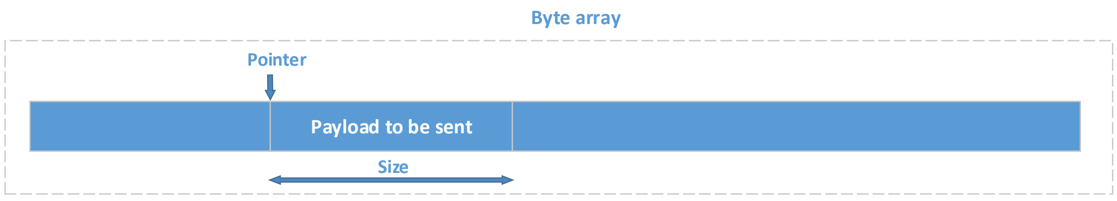


Figure 4.9 – *EmitLongMessages method.*

In the figure 4.9 is shown an example to better understand how the method works. The most important thing in the method it is to know well the inputs highlighted in the figure 4.9. This inputs are going to be used with other method called *Split*.

The method *EmitLongMessages* creates a packet just like any other packet with the difference of the payload. The payload is entered in the line 7, and it is created with the method *Split*. This method takes the byte array, the pointer and the size and returns a byte array with the payload indicated by the inputs.

The method *Split* is the following:

```java
public static byte[] Split (byte[] bytes,int pointer, int size){
    byte[] splited=new byte[size];
    for(int i=0; i<size; i++ , pointer ++){
      splited[i]=bytes[pointer];
    }
    return splited;
}
```

The inputs of this method are a byte array, with the whole data to be sent, and two integers, the first integer being the pointer, from where the data is going to be sent and the second integer being the size of the part of data that is going to be sent. As output it has a byte array that will contain the payload that has to be sent in the next packet.

As for the method, it creates, first of all, the byte array that is going to be the output of the method with the size indicated by the integer input *size*. Then we go through the output array introducing the bytes of the data array. The output array *splited* start at 0 and the data array *bytes*start at the pointer. In every interaction the pointer is incremented. For this reason it is necessary to use two pointers, the original input and an auxiliary integer. Both of them incremented at one in every iteration. When the output array is empty the method has completed its task and it returns the array.

4.2.4.2 Receiving images with AX25, receiver

Receiving the image involves receive the different parts of the image (in bytes), putting them together and finally converting those bytes to an image.

First of all the message is received with the ack and the image length. With this information the program knows how many parts are going to arrive. Then the image parts arrive and finally those are put together in a directory that the user has input.

An example of all the packets that are received is shown in Figure 4.10. In this example it is going to be received an image of 1024 bytes in 4 parts. In the image the length of the packets is not proportional to its real size. The first packet, with the ACK, is smallest with a big difference. This packet has a payload of approximately 20 bytes, it depends on how many bytes we need to express the length of the image. The length is used to know how the emission has finished and then the reverse process done in the transmitter is carried out. All the parts are joined and passed to bytes.

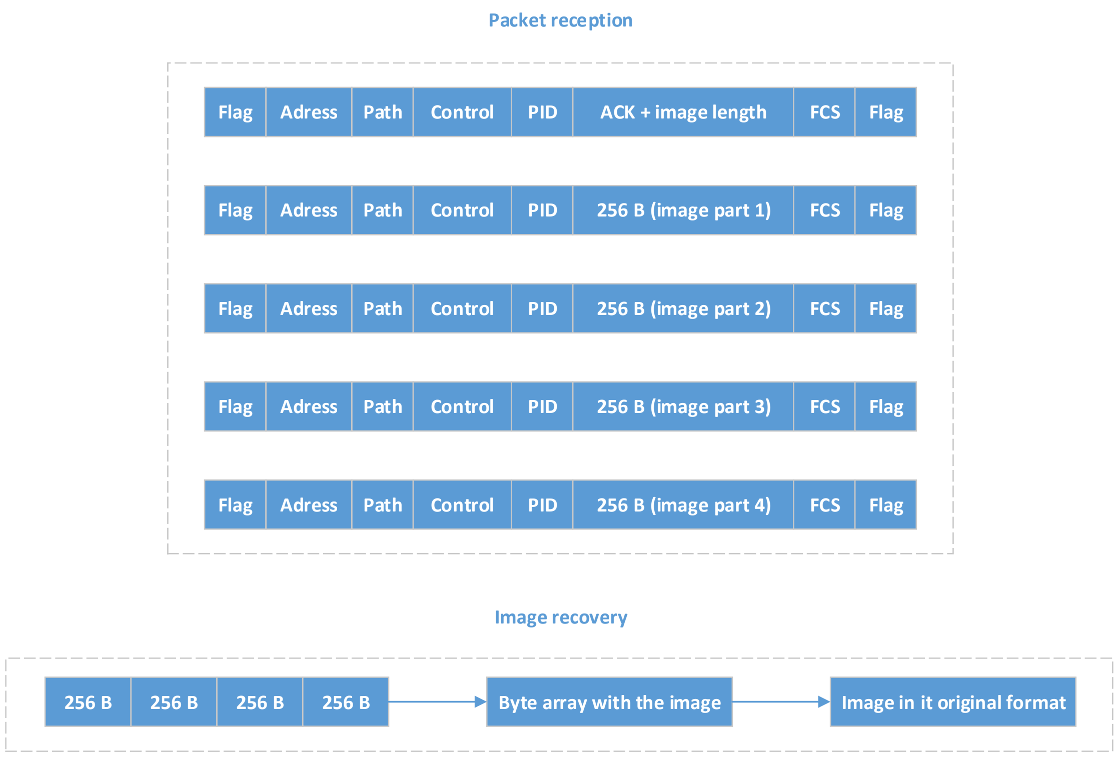


Figure 4.10 – *Receiving an image.*

Once analyzed how receiving images works the code that makes this possible is going to be analyzed. When any packet arrives the payload is analyzed. If the receiver has asked for the image it is waiting the ack from the transmitter. When it is received the ack is checked by the method *analyzePayload()* and then with the method *whichACK()*, this method is described in section 4.2.3. With this last method it is extracted the length from the packet and this length is saved in the integer MESSAGE_LENGTH. And the boolean imageArriving is activated, this means that it turns from false to true indicating to the program that the following message are parts of an image. The boolean ready is turned to false, this means that the receiver stops analyzing the received packets until the whole image is arrived, because it will not be useful to analyzed those packets if the program already knows those packets represent part of an image. This is going to be used in the method *packetReceiver* as follow:

```
if (imageArriving){
        int rest=0;
        if((MESSAGE_LENGTH%256)!=0){
          rest=1;
        }
        int packetNumber=rest+MESSAGE_LENGTH/256;
        int aux1=packetNumber+startOfImage-1-r;
        String packetsRemained= ""+aux1;
        dashboard.remainingPackages.setText(packetsRemained); //number of packages remaining
        int aux2=(int)((double)aux1*3.4);
        String secondRemained=""+aux2;
        dashboard.remainingTime.setText(secondRemained+"s"); //seconds remaining to finish the
         image issue
        saveIncompleteImage(); //this method save the bytes on a new image to go showing the
         image arrived
        if(r==(packetNumber+startOfImage-1)){
          saveImage(packetNumber);
          ready=true;
17            sendACK("IMAGE");// If the image is sent and saved the ACK is sent
18        }
19        counter++;
20 }
```

First of all it is calculated the number of packets that are going to arrived. This is calculated dividing the number of bytes of the image between 256, because this is the maximum of bytes that can fill in a packet. But is is also necessary to calculate if there is any rest on this division, if there is rest it is added 1 to the final number of packets. This is done because if the last image part does not fill the whole packet this packet is still sent but in the division result it is not shown. For example, if an image is 258 bytes two packets will be sent, the division of 258 between 256 is 1 plus a rest. This means 1+1, two packets are going to be received. This calculation is done from line 2 to line 6.

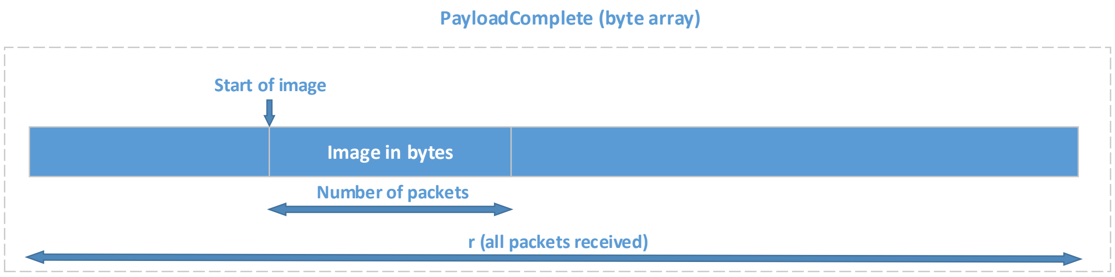


Figure 4.11 – *Payload complete representation.*

Then from line 7 to line 12 the dashboard shows the remaining packets and the estimated remaining time. The remaining packets are calculated in the integer aux1. To make this calculation it is necessary to add to the packet number the start of image (the packet arrived when the image starts, this is necessary if there were more packets before) less r (number of packets received since the receiver was started), less 1 because the first packet related to the image (that makes imageArrivin true) is the ack packet. This variables are represented in figure 4.11

To calculate the estimated remaining time different tests are applied to prove that if the number of packets is multiplied by 3.4 we have the remaining time. This is only useful for the scenario we are using.

After that the method *saveIncompleteImage*, that will be analyzed in detail later, is used. But basically what it does is to save the image bytes that have arrived at an image in JPG, in order to be able to visualize the image that arrives.

Then in line 14 it is analyzed if all the packets have arrived , if this happen the systme will save the complete image and the image arriving mode is turned to false (in the method *saveImage*), finishing the reception of the image and ready is turned to true, making the packet receiver free again to analyzed the following packets. The image is saved with the method *saveImage* that is very similar to the method *saveIncompleteImage*.

Saving images

In order to save an image I implemented two methods, *saveImage* and *saveIncompleteImage*. The first one saves the complete image already arrived in a directory given by the user, the second one saves the incomplete image every 5 packets arrived.

The method *saveImage* is the following:

```
private void saveImage(int packetNumber) {
        byte[] image= joinMessage(payloadComplete,packetNumber,startOfImage);
        String imageDir = p.getProperty("directory", "/Users/Lucia/Documents/ugr/tfg/imagenes/
          recepcion/image");
        Byte_To_Image (image, imageDir);
        imageArriving=false;
        imageDir=imageDir+".jpg"; //this is the directory where the image is saved, we add the
          name of the image
        dashboard.reloadPictureDirectory(imageDir); //every time the image is saved it is shown
           on the application

        imageCounter=0; //It is restarted the image counter used to save the incomplete images.
}
```

This method has as input the number of packets needed to contain the whole image.

First of all all the parts of the image are joined, this is done with the method *joinMessage*. This method need the payload complete (a byte array where all the payload arrived, since the receiver was turned on, has been saved), the number of packets and the point in this byte array where the image starts (startOfImage), see figure 4.11.

Then it is necessary to save the directory that the users want to use in an string. This is done with the variable Properties p, if there is no directory it is saved in the directory by default (/Users/Lucia/Documents/ugr/tfg/imagenes/recepcion/image). This default directory is only useful in my computer, in case the user want to have a directory by default to save the final image he only have to change the directory in the method. But that is not necessary if the user always says which directory he wants in the properties.

Knowing the directory the image is saved with the method *Byte_To_Image*. This method needs the byte array with the whole image and the directory where it is going to be saved. This method will be analyzed in detail later. But basically what it does is the inverse function of the method *Image_To_Byte* anlayzed in section 4.2.4.1.

Then *imageArriving* is turned to false, as it was explained before, to indicate that the reception of the image has been completed.

Then the dashboard shows the image, every time the image is saved it is shown on the application. To make this possible it is necessary to add to the directory string (*imageDir*) the kind of file the image is. Then this completed directory is used as an input in the method *reloadPictureDirectory()* that is situated in the class dashboard, what it does is showing the image.

The last line is to reload the class variable that it is used in the method *saveIncompleteImage*. We restart the image counter used to save the incomplete images for the case there were not a number of packets divisible between 5.

After analyzing the complete method to save images we are going to analyzed the different methods involved. The first one is *joinMessage*:

```
public byte[] joinMessage (byte[][] bytes, int packets, int start){
    int totalLength=0;
    for(int i=start; i<=packets+start -1; i++){
      totalLength=totalLength+bytes[i].length;
    }

    byte[] message=new byte[totalLength];
    int length=0;

    for (int i=start; i<=(packets+start -1); i++){
      System.arraycopy(bytes[i],0, message,length,bytes[i].length);
      length= length+bytes[i].length;
    }
    return message;
}
```

This method has as input byte matrix, where in every row there is the payload of a packet. In each row each column represent a byte. This byte matrix saves all the payloads arrived since the receiver turned on. This method also has as input the number of packets and the start of the message that want to be join. The method output is a byte array with the whole message joined.

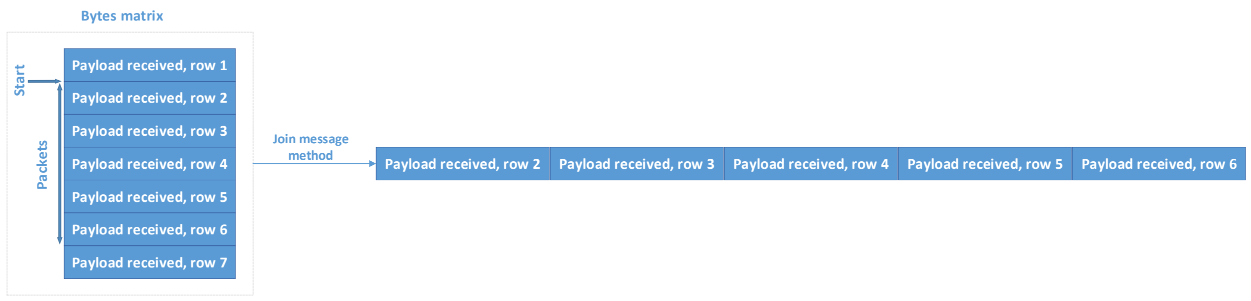


Figure 4.12 – *Join message method representation.*

In the figure 4.12 it is shown how this method works. This figure is intended to facilitate the correct understanding of the method.

The method, first of all, calculates the total length that will have the message already joined. Then it creates an empty byte array. And with the method arraycopy, java own, every line of the matrix that has to be copied is copied in the byte array. It is also necessary to have the variable length that has the length of the message in each moment.

The next method involved in the method *saveImage()* it is the method *Byte_To_Image*:

```
public  void Byte_To_Image (byte[] bytes, String route){
    try {
      File newFile= new File(route+".jpg");
      BufferedImage imag=ImageIO.read(new ByteArrayInputStream(bytes));
      ImageIO.write(imag, "jpg", newFile);
    }catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
}
```

This method has as input the byte array with the image and the directory where the user wants to save the image. It is necessary to import previously the following: java.awt.image.BufferedImage, java.io.ByteArrayInputStream, java.io.ByteArrayOutputStream, java.io.File, java.io.FileInputStream, java.io.FileNotFoundException, java.io.IOException, java.util.logging.Level, java.util.logging.Logger. This libraries contains the necessary functions used to pass the image to bytes. This method is based in a method taken from [2].

The next method involved in the method *saveImage()* it is the method *reloadPictureDirectory* in the class dashboard:

```java
public static void reloadPictureDirectory(String directory) {
    try {
        BufferedImage bufImg = ImageIO.read(new File(directory));
        pictureLabel.setIcon(new ImageIcon(bufImg));

    } catch (IOException ex) {
        System.out.println("Unable to read image file");
    }
}
```

It can be noticed that it is necessary to create a buffered image. Then it is used the method ImageIO.read Java own, which introduces the directory as an input. Then it is used the method setIcon creating an ImageIcon. The image will appear in the label pictureLabel, this label is shown in section 4.3. The try-catch block is used to catch the exception related to reading the image file.

Now we are going to analize the method *saveIncompleteImage*. This method is based in the method *saveImage*:

```java
private void saveIncompleteImage() {
    imageCounter++;
    if(imageCounter==5){

      String imageDir = p.getProperty("directoryIncomplete", "/Users/Lucia/Documents/ugr/tfg/
        imagenes/recepcion/Incomplete/image");

      byte[] image= joinMessage(payloadComplete,(r-startOfImage),(startOfImage));

      //Threads to save image and to avoid significant delays
      Runnable incompleteSave = new saveImageThread(imageDir, image);
      Thread thread1 = new Thread(incompleteSave);
      thread1.start();

      imageDir=imageDir+".jpg"; //this is the directory where the image is saved, we add the
        name of the image
      dashboard.reloadPictureDirectory(imageDir);  //every time the image is saved it is
        shown in the application

      imageCounter=0; //The image counter is restarted
    }
}
```

This method needs a class variable that indicates every how many packages have to save the bytes. This counter increases every packet received and restarts every time an image is saved.

When the counter is five it saves the image. To save the image this method uses the method *joinMessage* already explained. In this case the number of packets is variable and is the whole number of packets already received since the receiver turned on (r) less the point the image started to arrived (startOfImage). This means it is going to join all the payloads arrived since it was indicated that the image arrived.

Then the image is saved , but in this case it is used a thread to use the method Byte_To_Image. This is because every time the program uses this method it stops to listen new packages. For that reason it is used a thread that saves the image at the same time that the program is still listening for new packages. This is necessary in this method because it is known that the image is still arriving while this method is saving the already arrived bytes.

Then the image is shown in the dashboard in the same way it does with a complete image.

Video 4.1 – *Reception video.*

This method make possible that the user can see the image as it arrives without have to wait that the image arrives completely. In the video 4.1 can be observed this effect. In the scenery where this picture was sent and arrived the speed was 10 times slower, this video has been accelerated to make the reproduction of the video more gently.

4.2.4.3 Sending and receiving images with FX25

The only difference between sending images through AX25 and FX25 is that the payload is longer in the FX25, so it is just necessary to use the method that will be explained in section 4.2.5 to create the FEC and join it to the payload, then the image will be sent in the same way.

For that reason the only difference at code is when we create the packet. The way the image is splitted is exactly the same. FX25 firstly splits the message, then at every split the code inserts the FEC. To make this it is used the method *SendFecImage* that differs from

SendImage in the first packet, that in FEC mode uses the constructor in the class packet made to this situation, that adds the FEC. And it also differs in the fact that the method *EmitFecLongMessages*that sends the packet with FEC.

When it is received, it is the same for every FX25 packet, image or not. As it was explained at 4.2.2, if the FEC mode is activated when the packet is arrived it is used the method that used the FEC and solved the possible errors in the payload. Then the payload already checked is saved and joined with the others payloads of the same image. The way the image is firstly splitted and then joined is the same, the same methods are used.

4.2.5 Forward Error Correction improvement with java

In this section it is explained how I implemented the FEC algorithm that is described at 3.1.1.3.2.

It is necessary to firstly generate the FEC of a byte array on the transmitter and then detect and correct mistakes of the byte array with the FEC in the receiver. To generate it I implemented the method *calculateFec*, which uses other smaller methods to calculate the FEC of the complete byte array.

It is important to highlight that it is necessary to work with bytes and bits. To work with bytes I use the byte variable of java. But to work with bits I use integers, that will take the form of one or zero to represent the bits. This was because the variable that java provides to work with bits was more difficult to work with arrays than integers.

To detect and correct all the possible errors that the FEC can solve I implemented the method *payloadFecSolved*, that returns the payload already corrected when we introduce to it the payload arrived and the FEC. This method is used by the receiver.

4.2.5.1 Generate FEC

Description

The method *calculateFec* generates the whole FEC of the byte array that is going to be sent. The method *calculateFec* uses the method*generateFec* for every byte of the byte array. This means that the method *generateFec* is called depending on the length of the byte array.

```java
public static byte[] calculateFec(byte[] bytes) {

    int[] bits=convertBytesToBits(bytes);

    int [] auxBits= new int[8];
    int [] auxFec= new int[4];
    int resto=0;
    if((bytes.length)%2!=0){
      resto=4;                     //4 bits, half byte
    }

    int [] fec= new int [bits.length/2+resto];  //This is the final array with all the bits

    int u=0;
    int s=0;

17      for (int i=0; i<=bytes.length-1;i++){
18
19        for (int j=0; j<=7; j++){
20          auxBits[j]=bits[u];
21          u++;
22        }
23
24        auxFec=generateFEC(auxBits);
25
26        for (int r=0; r<=3; r++){
27          fec[s]=auxFec[r];
28          s++;
29        }
30      }
31
32      if(resto==4){
33        for(int i=fec.length-4; i<=fec.length-1;i++)  {
34            fec[i]=0;
35        }
36      }
37
38      byte [] fecByte= convertBitsToBytes(fec);     //The bits with the FEC is converted to
         bytes.
39
40      return fecByte;
41  }
```

The input and output of *calculateFec* are bytes, but the method works with bits. For this reason *calculateFec* firstly uses *convertBytestoBit* then generates the whole FEC and then uses *convertBitstoBytes* to transform the FEC on bits to the FEC on bytes to be sent.

Firstly this method initialises the bits arrays that are going to be used in the method. The first one (bits) is going to contain the payload already translated to bits. Then there are two auxiliary bits arrays, auxBits (in this array it will be enter the original bits 8 by 8 bits) and auxFEC (in this array will be enter the result of the FEC 8 by 8 bits).

From the line 17 until the line 30 the code makes the following: for every byte of the byte array it saves 8 by 8 bits of the array with all the bits to make it possible to use the method generateFEC. It is generated the FEC of 8 bits. It is saved the FEC of every 8 bits that at the end will have the complete FEC of the byte array.

The last part of the code (lines 32-36) are needed because of FX25 need and integer number of bytes, but the FEC is the half of the length of the byte array. For that reason if there is an odd number of bytes the number of bits of the FEC will not complete the last byte, this last byte will be filled with the 4 zeros that are needed.

The method *generateFec* generates the FEC of only one byte.

```
public static  int[] generateFEC(int [] bits){
    int [][] AND_result=new int[4][8];
    for(int i=0; i<=3;i++){
      for(int j=0; j<=7;j++){
        AND_result[i][j]=FECCode[i][j]&bits[j];
      }
    }

    int [] paridad = parity(AND_result);
    return paridad;
}
```

The input and outputs are integer arrays, representing a bits array. Like it was explain on 3.1.1.3.2 it is necessary to ANDed the FEC code matrix with the bits array. We need two for to go through the matrix and only one to go through the array. To AND both it is used the operator . Finally it is necessary to find the parity array of the ANDed matrix (in other method) that will be the FEC of the byte input.

The method parity find the parity of any matrix and enter it in an array with a length four positions. This method still working with the representation of bits in integer arrays.

```
  public static  int [] parity (int [][] matrix){
    int aux=0;
    int [] output={0,0,0,0};
    for(int i=0; i<=matrix.length-1;i++){
      for(int j=0; j<=matrix[i].length-1;j++){
        if(matrix[i][j]==1)
          aux++;
      }
      if ((aux%2)!=0)
        output[i]=1;
      aux=0;
    }
    return output;
}
```

For every column it is counted the number of ones that it has. To make this it is necessary two for statements, the first one is the one is the columns and the second the positions on the columns. Then it is checked if this number of ones is even or odd analyzing the rest of the division between the counter of ones and two. If the rest is not zero means that the parity of that column is 1. This process is repeated for every column, that are four. The parity is saved in order on an integer array that will be the output.

Conversions

It is also important the used of the methods to convert from bytes to bits and vice-versa.

Bytes to bits

In order to change from bytes to bits I developed the method *convertBytestoBits*.

```
    public static int[] convertBytesToBits(byte[] bytes){
    int [] bits=new int [bytes.length*8];
    int [] auxBits=new int [8];
    byte[] auxByte=new byte[1];
    int s=0;
    for(int i=0; i<=bytes.length-1; i++){
      auxByte[0]=bytes[i];

      auxBits=byteToBits(auxByte);

      for(int j=0; j<=7; j++){
        bits[s]=auxBits[j];
        s++;
      }
    }

    return bits;
}
```

The input of this method is an array of bytes and the output an array of integers, that represent bits. To convert the byte array I use two auxiliary arrays to save the actual byte that is being translating to bit (this is an array of only one position) and then an array of integers with length 8 to save the byte already translated to 8 bits. Then the bits are added to the final bit array. To translate each byte I implemented other method that converts only one byte to 8 bits, *byteToBits()*:

```
public static int[] byteToBits(byte[] bytes){
    int[] bits=new int[8];
    String valor=byteToString(bytes);

    if (valor.length()==4){
      valor=""+valor.charAt(2)+valor.charAt(3);
      int dec=Integer.parseInt(valor,16);
      if(dec<=127){
        bits=ascii(dec);
      }else{
        valor=Integer.toBinaryString(dec);

        for(int i=0; i<=valor.length()-1; i++){
          if(valor.charAt(i)=='1'){
            bits[i]=1;
          }
        }
       }
    }else{
      int numero=bytes[0];
      bits=ascii(numero);
    }

    return bits;
}
```

This method firstly obtains the value of the byte in a string. If this value has a length of 4, means that it is not a letter, for example \x13. In this case we rewrote the value in a suitable way, removing the \ and the x, the example at this point would be 13 that now can be easily analyzed, this number is in hexadecimal. Then this number is translated to decimal. If the decimal number is less than 127 this means that it is included in the ascii code, so I implemented other method () to translate those bits. If it is not in the ascii code it is used the method, already included in java, *Integer.toBinaryString*. This method was not able to translate those numbers that was in the ascii code. For those numbers it is necessary to do it manual. This ascii method is also used to the case in which the byte represents a letter. Those situation were separated because of the way it is introduced the input of the ascii. In the case of the letter it is only necessary to equal the integer input to the byte, for the other case it is necessary to extract the value of the byte. The ascii method is the following:

```
public static int[] ascii(int numero){
    int temp = numero;
    int[] bits=new int[8];

    String resultado = "";
    while (temp != 0){
      if(temp % 2 == 0){
        resultado = "0" + resultado;
      }else{
        resultado = "1" + resultado;
      }
      temp = temp / 2;
    }
    int length=resultado.length();
    for(int i=0; i<=8-length-1; i++){
      resultado="0"+resultado;
    }
    for(int i=0; i<=resultado.length()-1; i++){
          if(resultado.charAt(i)=='1'){
            bits[i]=1;
          }
        }
    return bits;
}
```

In the *ascii()* method it is implemented the manual way to calculate a binary number from a decimal number, that in this case is the bit array. The input of this method is the decimal number in an integer and the output is the array of bits. To do this calculation it is necessary to divide between two the number all necessary times until it is less than 1. It can be observe that this is implemented with a while, in the line 6, where it will be dividing between two while the number (saved in the auxiliary temp) is different from 0. The number is an integer so if it the division is not exactly the result that would be between one an zero the number saved in the auxiliary would be 0, so if the number is less than 1 it has to go through 0 finalizing the while. The result of the divisions are saved on the auxiliary, that is going to decrease at every turn. If the division on a turn has no rest the next bit would be 0. If there is any rest it is 1. Those numbers are added to an String that we would translate later to a bit array. In the *for* initialized in the line 15 we add all the zeros necessary to complete the 8 bits, this is necessary for the numbers that are not long enough. In the second *for* (from line 18 to line 22) it is passes the string to an array of integer representing the array of bits.

Bits to bytes

In order to change from bits to bytes I developed the method *convertBitsToBytes*.

```java
public static byte[] convertBitsToBytes(int[] bits){
    byte[] bytes=new byte[(bits.length/8)];
    int [] auxBits= new int [8];
    byte[] auxByte=new byte[1];
    int s=0;
    int u=0;

    for (int i=0; i<=bytes.length-1; i++){
      for (int j=0; j<=7; j++){
        auxBits[j]=bits[u];
        u++;
      }
      auxByte=bitsToByte(auxBits);
      bytes[s]=auxByte[0];
      s++;
    }
    return bytes;
}
```

The input of this method is an array of bits and the output an array of bytes. To convert the bit array I use two auxiliary arrays to save the actual 8 bits that are being translating to a byte and then an array of bytes to save the bits already translated to a byte. Then the bytes are added to the final byte array. To translate each 8 bits I implemented other method that converts only 8 bits to one byte, *bitsToByte()*:

```java
public static byte[] bitsToByte(int[] bits){
    String exit="";

    if (bits[7]==1){
      exit="1";
    }else{
      exit="0"+exit;
    }
    if(bits[6]==1){
      exit="1"+exit;
    }else{
      exit="0"+exit;
    }
    if (bits[5]==1){
      exit="1"+exit;
    }else{
      exit="0"+exit;
    }
    if (bits[4]==1){
      exit="1"+exit;
    }else{
      exit="0"+exit;
    }
    if (bits[3]==1){
      exit="1"+exit;
    }else{
        exit="0"+exit;
      }
      if(bits[2]==1){
        exit="1"+exit;
      }else{
        exit="0"+exit;
      }
      if (bits[1]==1){
        exit="1"+exit;
      }else{
        exit="0"+exit;
      }
      if (bits[0]==1){
        exit="1"+exit;
      }else{
        exit="0"+exit;
      }

      byte[] bytes=new byte[1];
      bytes[0]=(byte)Integer.parseInt(exit, 2);

      return bytes;
  }
```

This method firstly passes the 8 bits on the array of integers to a String (exit). Then it uses the java native method Integer.parseInt to pass the String exit to an integer and with a casting we pass that integer to a byte. This byte is an array with only one position. This is done to the case we need to add to this byte array more bytes easily.

General overview

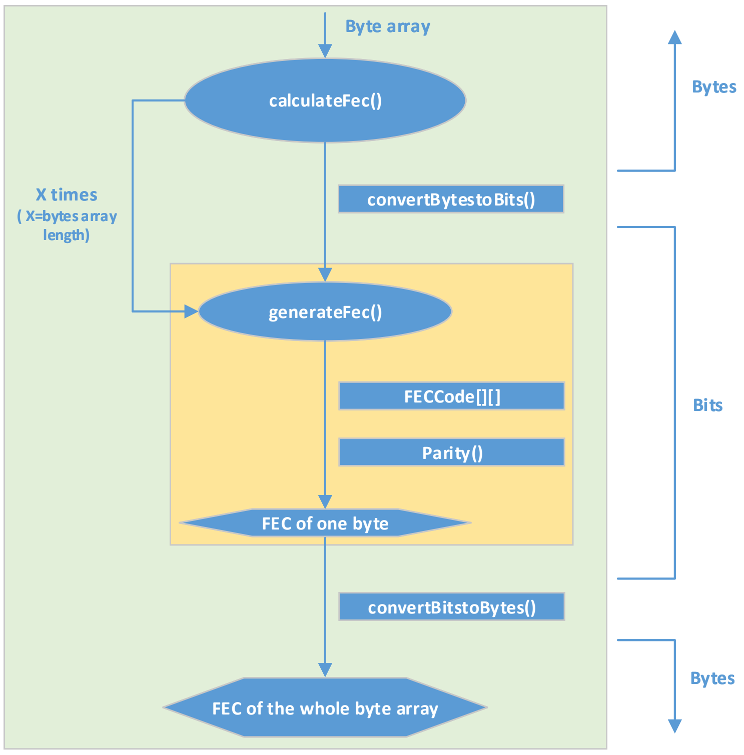


Figure 4.13 – *Overview of FEC generation, in the transmitter*

The figure 4.13 represents an overview of the generation of the FEC, already explained. In this representation the ellipses represents the main method, the rectangle represents the methods or matrix used by the actual state to pass to the following state and the hexagon is used to represent the outputs. In the orange background we situate the method *generateFEC* and in the green background the *calculatedFEC*. In this representation it is also illustrated how the methods works with bytes and bits.

Mode of use

The way the method *calculateFec* is use introducing the whole payload. The FEC is used in different parts of the main class, one example is to send a normal message. A normal message without FEC is send as follow:

```
private static void SendNormal(String payload) {
    Packet packet = new Packet("APRS",
          callsign,
          new String[] {"WIDE1-1", "WIDE2-2"},
          Packet.AX25_CONTROL_APRS,
          Packet.AX25_PROTOCOL_NO_LAYER_3,
          payload.getBytes());
      Transmission transmission=new Transmission();
      transmission.sound(packet, mod, ptt, multi, rate, t, p);
}
```

A normal message with FEC is send as follow:

```
private static void sendFecPacket(String payload) {
    Packet packet = new Packet("APRS",
          callsign,
          new String[] {"WIDE1-1", "WIDE2-2"},
          Packet.AX25_CONTROL_APRS,
          Packet.AX25_PROTOCOL_NO_LAYER_3,
          payload.getBytes(), Fec.calculateFec(payload.getBytes()));
        Transmission.sound(packet, mod, ptt, multi, rate, t, p);
}
```

In the section 4.2.5.4 we will explain the class Packet. But here it can be observed than to calculate the FEC the method *calculateFec* is used with the input payload translated to bytes. In the example without FEC it has been used other constructor of Packet that does not have the input of FEC.

4.2.5.2 Detect and correct errors with FEC

Description

The method *payloadFecSolved* returns the corrected payload using the FEC in the same packet. This method uses the method *solveFEC* to fulfill its tasks. Similar to the generation of the FEC the method *payloadFecSolved* calls the method *solverFEC()* as many bytes there are on the byte array (the payload array, not the payload plus the FEC):

```
public static byte[] payloadFecSolved(byte[] payload, byte[] fec) {
    byte[] payloadSolved= new byte[payload.length];
    int [] bitsPayload= convertBytesToBits(payload);
    int [] bitsFec= convertBytesToBits(fec);
    int [] bitPayloadSolved = new int[bitsPayload.length];
    int [] wordArrived= new int [12];//8 bits of payload + 4 bits of their fec
    int payloadCounter=0;
    int fecCounter=0;
    int finalCounter=0;

    for (int i=0; i<=payload.length-1; i++){
      for (int r=0; r<=11; r++){
```

```
13            if(r<=7){              //wordArrived=payload+fec of 8 bits
14               wordArrived[r]=bitsPayload[payloadCounter];
15               payloadCounter++;
16            }
17            if (r>7){
18               wordArrived[r]=bitsFec[fecCounter];
19               fecCounter++;
20            }
21         }
22
23         int[] aux=solveFEC(wordArrived);
24
25         for (int s=0; s<= 7; s++){
26            bitPayloadSolved[finalCounter]=aux[s];
27            finalCounter++;
28         }
29      }
30
31      payloadSolved=convertBitsToBytes(bitPayloadSolved);
32
33      return payloadSolved;
34 }
```

The input of the method *payloadFecSolved()* are two bytes array, the first with the payload and the second with the FEC. The output is a byte array with the payload already solved.

This method has in common to the generation method that, despite its inputs and output are byte array, inside the method it works with bits. To do this it has to use the method already explained at section 4.2.5.1 that converts from bytes to bit, at the beginning, and from bits to bytes at the end.

First of all it initialises all the arrays necessaries to implement the method. Those are the byte array where the payload already solved (output) will be saved, with the length of the payload not solved (input). Then there are created the two bits arrays where the input bytes arrays already translated to bits will be inserted . Then it is initialised an empty bits array where the payload solved will be saved in bits, in bits. The last bit array is an auxiliary that will be used to solve the payload for every 8 bits, those bits solved then are saved in the bit array called bitPayloadSolved. The last three integers are counters.

As it was explained before, the main method is *solveFEC*, this method used as input the 8 bits of payload and the 4 bits of FEC already joined. For that reason, in the method *payloadFecSolved* it is necessary to join every 8 bits of payload with it corresponding FEC. This is done in the *for* loop that starts at line 12 and ends at line 21. This *for* loop goes through the auxiliary bit array (wordArrived) saving for the first 8 bits the payload input from the pointer payloadCounter and for the rest 4 bits the FEC corresponding. For the FEC another pointer (fecCounter) is also used .

In the line 23 the *solveFEC* method is used , with the 12 bits corresponding to that round. Then in the last for (from line 25 to line 28) the payload already solved of 8 bits of that round is saved in the global bit array. This is repeated the length of the payload in bytes.

The method *solveFEC* solved the payload of only one byte plus the nibble (half byte) of the FEC.

```
public static  int[] solveFEC(int [] wordArrived){
    int[] paridad=detectBitFlipFEC(wordArrived);
    int position=locateFECError(paridad);
    int[] wordSolved=new int[wordArrived.length];

```

```
 6     for (int i=0; i<=wordArrived.length-1;i++){
 7       wordSolved=wordArrived;
 8     }
 9     if (position!=13){
10       wordSolved[position]=inverted(wordArrived[position]);
11     }
12
13     return wordSolved;
14 }
```

This method has as input the word arrived into the receiver with the combination of, firstly, one byte of payload and then the FEC of that byte of payload. This word is in an array of bits. The output is the word arrived already solved. This means that the method analises if there is any error and if it detects an error in a bit, thanks to FEC algorithm, it will be corrected. As is was explained in section 3.1.1.3, of every 8 bits, thanks to FEC algorithm, it can be solved 1 bit. This bit can be in any position, including the 4 bits of FEC. This means that the algorithm can state that the error is located in any of the 12 bits.

The method works on the following way. It firstly finds the parity array with the method *detectBitFlipFEC()*, then it finds the position of the error with the parity array as an input in the method *locateFECError()*. If there was no error, the number return by the method*locateFECError()* is 13. This two method will be analyzed later.

Then it is initialized a bit array with the length of the bit array input. In this array is going to be saved the word arrived (lines 6 to 8). Then, if the number position is not 13, meaning that there is an error, the bit corresponding to this position is inverted with the method *inverted*. What this method does is to change the number it was introduced as an integer to the opposite, if a 0 is input the method outputs a 1 and vice-versa.

The method *detectBitFlipFEC()* performs the calculation needed to find the parity array, check section 3.1.1.3.2, example 3.16.

```
public static int [] detectBitFlipFEC(int [] wordArrived){
    int [][] AND_result=new int[FECWithParity.length][FECWithParity[0].length];
    for(int i=0; i<=FECWithParity.length-1;i++){
      for(int j=0; j<=FECWithParity[i].length-1;j++){
        AND_result[i][j]=FECWithParity[i][j]&wordArrived[j];
      }
    }
    int[] paridad=parity(AND_result);

    return paridad;
}
```

This method has as input the bit array with the word arrived, this means the payload plus the FEC of 8 bits. And it has as output an array of bits, with the parity of the word input.

To find the parity array it is necessary to and the FEC matrix with its parity (this matrix is in table 3.14) with the word arrived. It is necessary two *for* loops to go through the matrix and only one to go through the array. To AND both the operator & is used . Then this matrix is input in the method *parity()* already explained at section 4.2.5.1. The result of the method is the output of the method *detectBitFlipFEC()*.

The method *locateFECError()* finds the location where there was an error, check section

3.1.1.3.2, example 3.17.

```
 1 private static  int locateFECError(int[] paridad) {
 2     int [][] fecToAND=new int[FECWithParity.length][FECWithParity[0].length];
 3     int [] AND_result=new int[FECWithParity[0].length];
 4     int position=13;
 5     //This for invert the necessary rows
 6     for(int i=0; i<=FECWithParity.length-1;i++){
 7       for(int j=0; j<=FECWithParity[i].length-1;j++){
 8         if(paridad[i]==1){
 9           fecToAND[i][j]=FECWithParity[i][j];
10         }else{
11           fecToAND[i][j]=inverted(FECWithParity[i][j]);
12         }
13       }
14     }
15 
16     for(int j=0; j<=FECWithParity[0].length-1;j++){
17       AND_result[j]=fecToAND[0][j]&fecToAND[1][j]&fecToAND[2][j]&fecToAND[3][j];
18     }
19 
20     for(int j=0; j<=FECWithParity[0].length-1;j++){
21       if(AND_result[j]==1){
22         position=j;
23       }
24     }
25 
26     return position;
27 }
```

The method has as input the parity array and as output an integer with the position of the error. If there was no error, the integer output will be the number 13.

The location of the bit can be divided in two parts. In the first one the method invert the row of the FEC matrix (the FEC matrix with its parity) whose parity is 0. To make this possible the method uses two *for* loops (line 6 to line 14) to go through the FEC matrix, and it is used the first for to go through the parity array. If the parity is 1, the FEC matrix row is not changed; if the parity is 0, the FEC matrix row is inverted. The new FEC matrix is saved in an auxiliary matrix.

The second part of the location is to and all the rows of the new FEC matrix, this is done from line 16 to line 18. The result of this is saved in an array that will be checked if it has any 1 on it. If there is a 1 its position is saved, because this position indicates the position of an error. If there is no error, the position number still 13. That indicates there is no error. This position number is the output of the method.

Mode of use

Every time a packet is arrived, if the mode of FEC is activated, (fecMode=true) the packet is analyzed with the method *payloadFecSolved()*. This is done on the next way:

```
      byte[] payload=null;
      payload= extractFecPayload(bytes); //received payload, it can contain errors
      byte[] fec=extractFec(bytes);   //received fec
      payload=Fec.payloadFecSolved(payload, fec);   //recovered payload
```

Firstly the payload and the FEC of the packet, with a method that calls the class packet, is extract. How the class packet extracts those will be explain in the section 4.2.5.4. Then the method *payloadFecSolved* is used inserting as inputs the payload and the FEC already extracted.

4.2.5.3 General overview

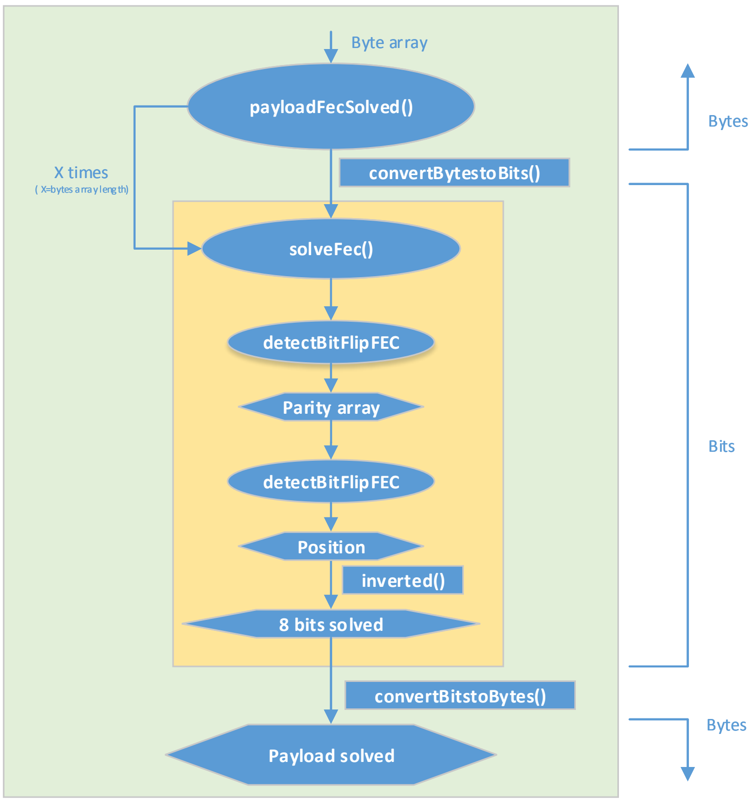


Figure 4.14 – *Overview of FEC solution, in the receiver*

The figure 4.14 represents an overview of the detection and correction of errors with FEC, already explained to simplify the comprehension. In this representation the ellipses represent the main method, the rectangle represent the methods used by the actual state to pass to the following state and the hexagon is used to represent the outputs. In the orange background the method *solveFEC* is situated and in the green background the *payloadFecSolve*. In this representation it is also illustrated how the methods work with bytes and bits. Apart from this examples this procedure is also used for every packet sent if the FEC mode is activated, including the images packet.

4.2.5.4 FEC packet

In this section it is going to be explained how FEC is added to the packet and the different methods that were implemented to access to the different part of the packet. The procedure uses different constructors in the same class packet. The first constructor is the one explained in section 4.1.3, implemented by Sivan Toledo, and the second constructor is the packet with the FEC implemented by me.

FEC packet constructor

The constructor that makes packet with its FEC is the following:

```
 1 public Packet(String destination,
 2              String source,
 3              String[] path,
 4              int      control,
 5              int      protocol,
 6              byte[]   payload,
 7              byte[] fec) {
 8
 9     int n = 7 + 7 + 7*path.length + 2 + payload.length+fec.length;
10
11     byte[] bytes = new byte[n];
12
13     int offset = 0;
14
15     addCall(bytes, offset, destination, false);
16
17     offset += 7;
18
19     addCall(bytes, offset, source, path==null || path.length==0);
20
21     offset += 7;
22
23     for (int i=0; i<path.length; i++) {
24       addCall(bytes, offset, path[i], i==path.length-1);
25       offset += 7;
26     }
27
28     bytes[offset++] = (byte) control;
29
30     bytes[offset++] = (byte) protocol;
31
32     for (int j=0; j<payload.length; j++) {
33       bytes[offset++] = payload[j];
34     }
35
36     for (int j=0; j<fec.length; j++) {
37       bytes[offset++] = fec[j];
38     }
39
40     assert(offset == n);
41     assert(size == 0);
42     assert (crc == CRC_CCITT_INIT_VAL);
43     assert (bytes.length+2 <= MAX_FRAME_SIZE);
44
45     for (int i=0; i<bytes.length; i++) {
46       packet[size] = bytes[i];
47       crc_ccitt_update(packet[size]);
48       size++;
49     }
50
51     int crcl = (crc & 0xff) ^ 0xff;
52     int crch = (crc >> 8) ^ 0xff;
53
54     packet[size] = (byte) crcl;
55     crc_ccitt_update(packet[size]);
56     size++;
57
58     packet[size] = (byte) crch;
59     crc_ccitt_update(packet[size]);
60     size++;
61
62     assert (crc == AX25_CRC_CORRECT);
63 }
```

As it can be noticed, the packet is formed in the same way a normal packet does.

It is important to highlight that the packet is formed in the way that it is explained in chapter 3, section 3.1.1.3.3, as it can be seen in table 3.18.

The length of this packet is the same as the normal packet but adding the FEC length. This can be seen in line 9.

The FEC is included in the packet the same way the payload does. This is done from line 36 to line 38.

The size increases in the same way it was explained in section 4.1.3, once all the packet without the CRC is inserted in the byte array the size will increase the byte array length.

Then, as the normal packet without FEC, the CRC is added and the size increases in one.

Methods to work with the FEC packet

It was necessary to implement in the same class packet three different methods. This methods are used to show the packet to the users (method *fecFormat()*) and to extract the payload of one packet (method *fecPayload()* and the FEC of one packet (method *fec*).

The method *fecFormat()* is very similar to the method *format()*.This method prints the whole packet in a way the user can read. This means that this method translates the bytes into an string. The method *fecFormat()* is the next:

```java
public static String fecFormat(byte[] packet) {

    String   source,destination;
    String[] path = null;
    byte[]   payload;
    byte[]   fec;
    int offset= 0;
    int repeaters = 0;
    int size = packet.length;

    destination = parseCall(packet,offset);
    offset += 7;
    source = parseCall(packet,offset);
    offset += 7;
    int aux=offset;
    while (offset+7 <=aux+14 && (packet[offset -1] & 0x01) == 0) {
      repeaters++;
      if (repeaters > 8) break;
      String path_element = parseCall(packet,offset);
      offset += 7;

      if (path == null) {
        path = new String[1];
        path[0] = path_element;
      } else {
        path = Arrays.copyOf(path,path.length+1);
        path[path.length -1] = path_element;
      }
    }

    offset += 2; // skip PID, control

    double dto=(2.0*((packet.length-1-offset))/3.0)+offset;
    int to=(int)dto;

    payload = Arrays.copyOfRange(packet, offset, to);
    offset+=payload.length; //2 es crc y 1 es el flag

    fec = Arrays.copyOfRange(packet, offset, size);

    StringBuilder builder = new StringBuilder();
    builder.append("[");

    builder.append(source);
    builder.append('>');
    builder.append(destination);
    if (path!=null) for (String via: path) {
      builder.append(',');
      builder.append(via);
    }
    builder.append(':');

    for (int i=0; i<payload.length; i++) {
      char c = (char) payload[i];
      if (c >= 0x20 && c <= 0x7E) builder.append(c);
      else builder.append(String.format("\\x%02x",payload[i]));
    }

    builder.append(':');
    for (int i=0; i<fec.length; i++) {
      char c = (char) fec[i];
      if (c >= 0x20 && c <= 0x7E) builder.append(c);
      else builder.append(String.format("\\x%02x",fec[i]));
    }
    builder.append("]");
```

```
        return builder.toString();
}
```

This method is very similar to the method *format* except in the way it is calculated from where to where the payload is situated and because in this method it is also found the FEC.

Until the payload is found, everything is found in the same way the method *format()* does, see 4.1.3. When the program arrives in the line 33 the offset already indicates the first bytes of the payload. Then the method calculates the last byte of the payload. This is done with the next function:

$Last\ payload\ byte = 2 * (\frac{Packet\ length - Offset}{3}) + Offset$

To understand how this function is formed it is necessary to well understand the different parts of it. Those are represented in the figure 4.15.

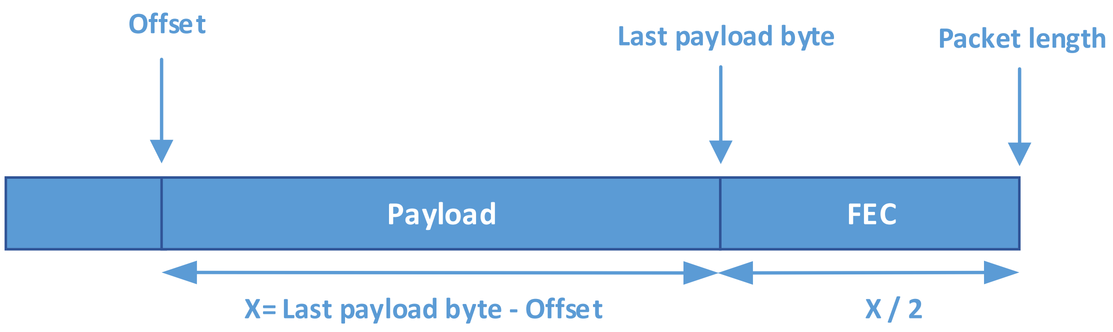


Figure 4.15 – *FEC format Packet.*

The final function comes from the simplification of the following function:

$Packet\ length - Offset = X + \frac{X}{2} = \frac{3}{2} * (Last\ Payload\ Byte - Offset)$

But it can be noticed a difference between the last function and the ones written in the method. In the method, in the line 33 can be seen the same function but also that the packet length has been subtracted one byte. This is because it is necessary the number of the last byte of the packet, and this would be packet.length - 1.

It is used firstly a double variable to work with tenths, and then it is done a casting to delete those tenths, because an integer number is needed to indicate the position on a byte array.

Then, using the method *Arrays.copyOfRange* (line 36), the payload is found and copied in a byte array. The offset increases the number of bytes the payload has. The method *Arrays.copyOfRange* has as input firstly the whole packet (in a byte array), then the offset (that works as a pointer) and finally the last byte of the packet that has to be copied.

Finding the FEC is simpler, it will be copied the last part of the packet, except the CRC.

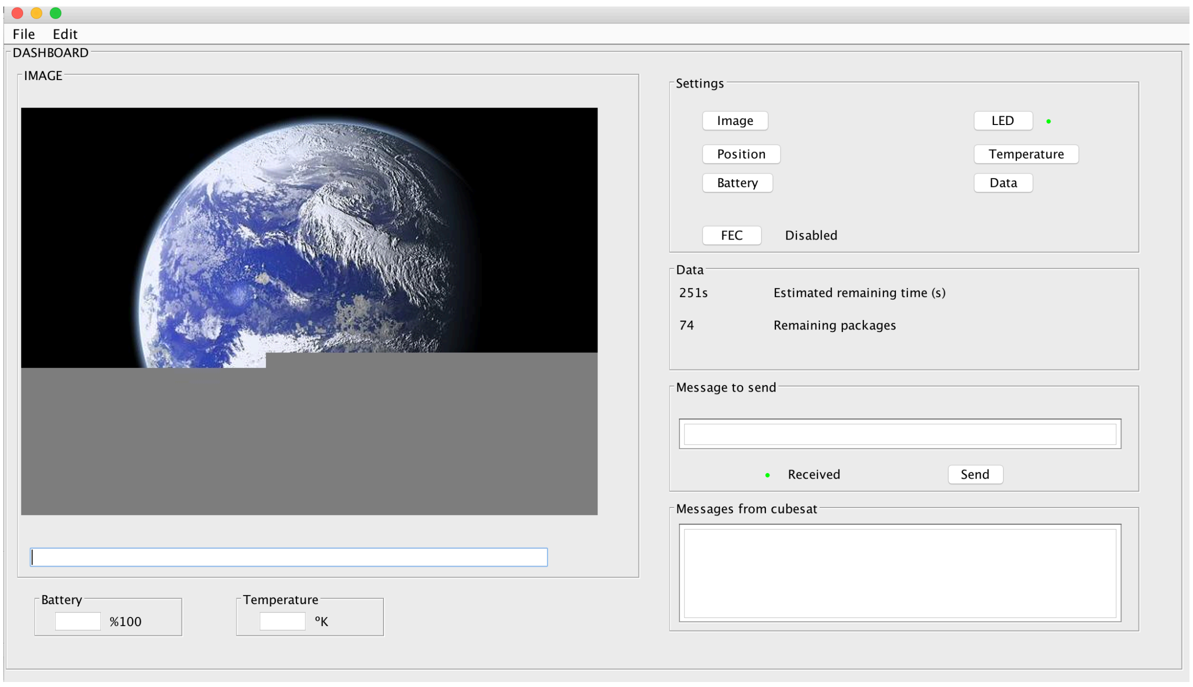
This is done with the method *Arrays.copyOfRange*. As input it has the whole packet (in a byte array), then the offset (that works as a pointer) and finally the last position of the FEC in the packet. The last position it is the size variable, this variable indicates the size of the packet without the CRC.

Now, once it has been found all the parts of the packet, the method *fecFormat* shows it by printing it with different signals that delimit the different fields of the packet.

Once explained the method *fecFormat*, the other two methods are simpler. The method *fecPayload* uses the same way to find the payload as the method *fecFormat* does, and then it returns the payload as a byte array. On the other hand, the method *fec* uses the same way to find the FEC as the method *fecFormat* does, and then it returns, and then it return the FEC as a byte array. The input used in both methods is the whole packet a a byte array.

4.3 Application design end to end

This section describes how the application design works. For this purpose it has been done a picture of the application in the moment it was arriving an image. The application is the same for ground and satellite, but in satellite when it is sending the image it is not shown any image while in the ground application it is shown the image as it arrived. In the ground the application shows the estimated remaining time and the remaining packages. The figure 4.16 shows the dashboard that is receiving the image.



251s Estimated remaining time (s)
74 Remaining packages

Figure 4.16 – *Application design.*

In the left it is situated the image, in this place the screen shows the image as it comes and when the reception has finalized it shows the whole image. Under the image there is a finder where you can search any image on your computer and it will be shown in the dashboard.

Under the image there are situated the data that can come from the satellite when it is requested. This data can be the battery, the temperature and the position. But the position is not shown in that place, it would be shown in the message form cubesat, for its length.

In the right place there are situated the settings to send different messages and to change the FEC mode. The buttons *Image*, *Position*, *Battery*, *LED*, *Temperature* and *Data* send the corresponding command asking for that information, see section 4.2.3. And the button *FEC* turns the mode FEC on, if it is off, or turns the mode FEC off if it is on. The mode is signalled with the words *disable* or *able*. In the picture 4.16 the FEC mode it is off. The led situated next to the *LED* button turns on or off depending in which way the *LED* in the satellite is.

The next section it is the Data. This place shows the information of the image that is arriving. It shows the estimated remaining time (calculated with the functions explain in chapter 6. And it also shows the remaining packages. The number of them is calculated in the receiver as the number calculated knowing the total length of the image out of 256. Then every packet arrived it is decreased the number of packet remaining. With this information the user is informed of the process of reception of the image.

The section *Message to Send* it is used to send any message from the user (in the ground computer) to the satellite). The user writes the message and then uses the button *Send to sent*. When it is received by the satellite this sends an ACK message that, when is received by the ground computer, makes the green led turned on. In the example the led is turned on because the last packet transmitted by the user was arrived at the computer representing the satellite an it informed the ground computer about it.

The last section in the dashboard it is the *Message form cubesat*. This place shows every message arrived from cubesat, already translated the bytes to an string that the user can read.

CHAPTER

5 FABRICATION

5.1 Operative systems

The way in which the simulated scenario has been implemented was with two computers. Each of the computer has a different operative system. The first is a MacOS Sierra and the second one is a Windows 7. There were some differences when working such as the way each computer detected the link, the names shown by the java program was different, but once the link is found both operative systems work exactly the same. But in general terms the differences were not important because in both computers I worked with the same development environments.

5.2 Development environments

Java is a programming language that can be implemented in many ways. In my case, I chose two development environment, Eclipse and Netbeans. Eclipse was used to develop the main program and to develop the dashboard I used Netbeans using jframes. The choose of them is explained in the following section, section 5.2.1.

5.2.1 Comparison between Netbeans and Eclipse

This section contains the analysis of both developments environments (Netbeans and Eclipse):

5.2.2 Netbeans

Netbeans (reference [4]), created by Sun Microsystems, is a free integrated development environment. The main programming language for which it was developed primarily is Java. It is an open source project with a constantly growing community, and with nearly 100 partners around the world.

Its platform allows the applications to be developed from a set of modules, which can be developed independently.

Applications can install modules dynamically and some can add an update module to allow users to download digital signature updates and new features directly in the running application; reinstalling an update or a new version, what makes possible that the users do not need to download the entire application again.

Among its characteristics, the next ones stand out:

- Management of the user interface (menus and toolbars),
- User configuration management.
- Storage management.
- Window management.
- Assistant framework.
- Netbeans visual library.
- Integrated development tools.

Netbeans environment provides support for the creation of SOA, including XML schema tools, a WSDL editor, and a BPEL editor for web services and makes possible the creation of Web applications with PHP.

5.2.3 Eclipse

Eclipse (reference [3]), which was created by IBM, is a computer program composed by a multiplatform open source programming tools to develop applications. This platform has mainly been used to develop IDE, such as Java. However, it can also be used for other types of client applications, examples of this are BitTorrent or Azureus.

Eclipse widgets are implemented by a Java tool called the Standard Widget Toolkit, unlike most Java applications, which use the standard options AWT or Swing. Eclipse's integrated development environment employs modules to provide all of its functionality at the forefront of the rich client platform, unlike other monolithic environments where features are all included, whether the user needs them or not. This module mechanism is a lightweight platform for software components. In addition, Eclipse is able to be extended using other programming languages such as C / C ++ and Python, and working with languages for text processing such as LaTeX, networked applications such as Telnet and Database Management System.

5.2.4 Development environments choice

In summary, Netbeans and Eclipse are two options that are very complete and robust, and that make possible to develop a big range of applications. So the choice of an IDE or another based on the possibilities offered is not simple, since both allow you practically the same. That is why my choice has been made taking into account the comfort and simplicity with which I work with one IDE and another. That is a particular choice because it is due to my own previous experiences with both IDE.

I chose Netbeans to develop the application because I have already worked with it using Jframe. The experience with it makes me possible to work easily and faster. Because of I have already worked with it I had all the plug-ins necessaries already installed and I could start to work immediately.

I chose Eclipse to develop the main program because of the program I based my own program was thought to be run on Eclipse, so it was easier to use Eclipse. But Eclipse has other benefits. The run configuration is much easier in Eclipse, and in my project I needed to be changing it in many occasions.

CHAPTER

6 TESTS AND VALIDATION

6.1 Images

As it was explained at chapter 4 I implemented a method to split the images in parts of 256 bytes that can fix in AX25 and FX25 packets, to be sent and then rebuilt in the receiver. This method needed to be tested. I did different tests for this purpose.

6.1.1 Completely sent and received

The first one was to be sure that the image has been completely sent and received. To ensure this I introduce two byte counters both in the receiver and the transmitter. In the transmitter the first counter (transmitterCounter1) indicates the length of the payload of the packet sent, the second counter (transmitterCounter2) indicates the total length of all the payloads already sent, that is the addition of all the transmitterCounter1. The counters were situated in the method that build the packet (public Packet) before it is sent, for every packet the counter was added the length of the payload, transmitterCounter1+=payload.length and transmitterCounter2+=transmitterCounter1 and both were printed on screen.

In the receiver, as the transmitter, the first counter (receiverCounter1) indicates the length of the payload of the packet received, and the second counter (receiverCounter2) indicates the total length of all the payloads already received, that is the addition of all the receiverCounter1. The counters were situated in the method that extracts the payload (public static byte[] payload). It is implemented in the same way as in the transmitter, receiverCounter1=+payload.length and receiverCounter2+= receiverCounter1.

With this both counters I make sure that we have received the complete number of bytes. I can compare the packets separately and the whole image. To every packet sent and received it was checked that transmitterCounter1 and receiverCounter1 was the same, the most of the cases it has to be 256 (256 bytes) except the last packet that can has less than 256, as it is explained at chapter 4. To check in which packet we are situated I made other counter that increases for every packet sent (in the transmitter) or received (in the receiver). If everything went right, as it does in the last version, the transmitterCounter2 had the same number (length on bytes) that the receiverCounter2.

This test helped me in the following way. At first both final counters were the same but the image with at a naked eye was wrong. It was because the receiver was analysing twice the same packet, so when the transmitter has sent the half of the image, the packet counter on the receiver said the image was completed. So it stopped to receive, even when the transmitter follow emitting. To find the error I had to check the first counter on both points. In the receiver the last packet was 256 bytes length, because the packet achieved as the last was actually an intermediate packet. Meanwhile in the transmitter were 4 bytes.

6.1.2 Images comparison

But this test only certifies that the length sent it the same as the received. So we need a test to verify if the images are the same. Even though at a first look the sent image and the received image seem the same. But we need to check every byte. To do this we first send an image, that will be tested, and then we compare every byte of the image sent and the image received. To do this I implemented a different program that uses the method to transform the image to a byte array (Image To Byte). The program compares every byte of both images with a for and if it finds a different byte it prints the position of those byte and the actual byte of each one.

```java
byte[] imagenEnviada=Image_To_Byte("/Users/Lucia/Documents/ugr/tfg/imagenes/
  guardada_imagenes1496779439184.jpg");
byte[] imagenRecibida=Image_To_Byte("/Users/Lucia/Documents/ugr/tfg/imagenes/recepcion/
  image1496899324800.jpg");

//If the length between both images are the same we check every byte of both images. If there
   is any difference between both it will be show, the byte and the position of it.
if (imagenEnviada.length==imagenRecibida.length){
  for(int i=0; i<=imagenEnviada.length-1; i++){
    if (imagenEnviada[i]!=imagenRecibida[i]){
      byte[] e = new byte[1];
      e[0]=imagenEnviada[i];
      byte[] r = new byte[1];
      r[0]=imagenRecibida[i];
      System.out.println("difference found in position "+i+" that in the image sent it is the
         byte "
        +byteToString(e)+" and in the received image is the byte"+byteToString(r));
    }
  }
}
```

The program that saves the image converts the sequence of bytes into an image with the Java method ImageIO.write, this method compresses the image. So we need to test not just the image arrived (that has already been compressed) but also the bytes arrived before it is compressed. This can also be tested with the same program, just introducing in the for the bytes array of the original image and the bytes array of the image arrived.

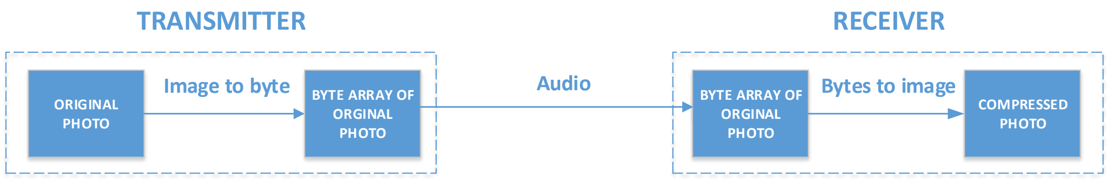


Figure 6.1 – *Explanatory scenario of the sending of images*

This situation is shown in the image 6.1.

This scenario shows how the original bytes arrived to the receiver, this means that in a future improvement it could be developed a different method to save the image without compression. Before showing the results of both tests, the compression of the image with ImageIO.write is shown. This is shown in the graphic 6.2.

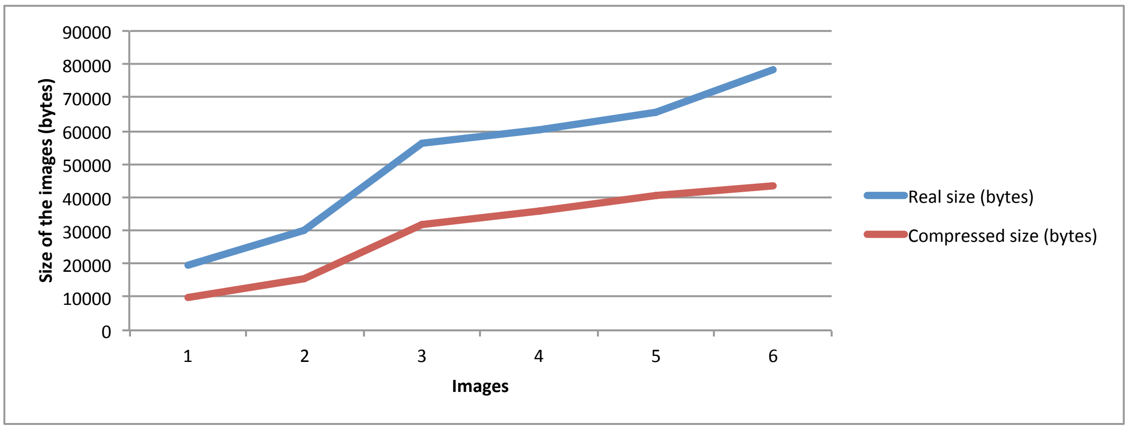


Figure 6.2 – *Graphic with the compressed images*

The results of the comparison between images sent and received test are the followings.

I have tested the image with normally sent, without any noise introduced. I have tested, as I said before, the bytes compressed and non-compressed. The results for both situations are the same; they show that there is no difference between the bytes of the image sent and the bytes received, and also that there is no difference between the image sent compressed and the image received. To compare the image received compressed it is necessary to compressed the image sent, to see if there is any difference. I have tested the 6 different images and none of them had any difference between the sent one and the received. This means that the image sender works properly.

6.1.3 Timing

About the timing we have to take into account the following considerations:

- To send an image it is necessary one short packet with the length of the image and then as many packet as necessary to complete the image.

- The number of packets to send the whole image is calculated knowing that the maximum size of the payload is 256.
- To calculate the number of packets we need the length of the image divided between 256 more 1 if the division is not an exact division.
- The packets with 256 bytes takes 680ms since it is sent by the transmitter until it is analysed by the receiver (in a scenario where transmitter and receiver are connected directly as it is explained in chapter 2).
- The small packets, like the packet with the length of the image, takes 15ms.
- Considering the worst case, where the last packet, which does not complete the 256 bytes, is 255 bytes, the timing calculation would be the next, which would take the same time as a 256 payloads packet:

  $ImageTiming = (\frac{length}{256} + 1) * 680ms + 15ms$

  $length$ is the length of the image. And to calculate the number of packet we divided between 256 and add 1 because we are considering the rest packet as the bigger one. The real size of the last packet is

  $LastPacketTiming = length - 256 * \frac{length}{256}$
- In this theory time is not been considered the time the transmitter takes to split the images and decide if it is necessary to send another message.
- The computational time required is approximately 1560ms per packet.
- This means that the final theory time would be:

  $ImageTiming = (\frac{length}{256} + 1) * (680ms + 1560ms) + 15ms$

The graphic 6.3 shows the theory time and the real time to send 6 images from the moment the users give the instruction to send until the image is actually arrived. We can observe they are really similar. This means that there are not odd characteristics that we are not taking into account.

6.2 FEC

The test of the FEC have different characteristics. First of all it is necessary to make sure that the FEC calculation is working, then the time it takes is compared to the AX25 and finally it checks how well the FEC react with interference on the link.

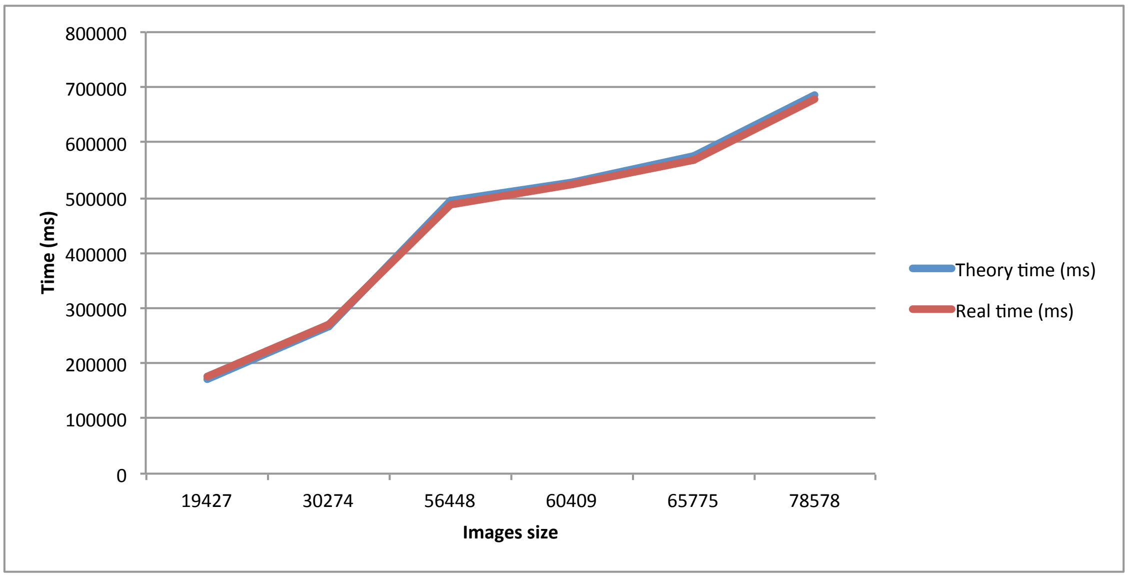


Figure 6.3 – *Image timing*

6.2.1 Working FEC

To be sure that the FEC is working we need to test different things.

1. The first of all is testing if the program is able to calculate the FEC of a byte.
2. Then we have to check if with the FEC calculated by the program can resolve a bit error in a byte.
3. The third test is checking if the program can calculate the FEC of a sequence of bytes and put it in the right order.
4. Then we have to test if the program is capable to solve bits error on a sequence of bytes.

In the first test there was not any problem. Everything worked properly. The same happened when it worked on the second test. The problem came when I tested the sequence, when the sequence was an odd number of bytes. The problem was due to the length of the FEC. For every byte the FEC calculated is a nibble (half of a byte) and if we want to transmitted via AX25 we need complete bytes. The solution to this was to add a padding with zeros in the FEC until finish the byte (4 bits of 0), this is explained on chapter 4.

For this test I used the followings codes:

TEST 1

```
int [] fec= Fec.generateFEC(bitsc); //I use the function I implemented on the class FEC to
  generate the FEC
System.out.println("fec: ");
for (int i=0;i<=fec.length-1; i++){
    System.out.print(fec[i]);       //This will print on screen the FEC of the letter h, that
      is 1100
}
```

TEST 2

```
int [] fec= Fec.generateFEC(bitsc); //I use the function I implemented on the class FEC to
  generate the FEC
int [] total = new int[bitsh.length+fec.length]; //I build an array with the whole FEC and
  payload, simulating what it is going to be sent.
System.arraycopy(bitsh, 0, total, 0, bitsh.length)
System.arraycopy(fec, 0, total, bitsh.length, fec.length);
int[] payloadSolved=solveFEC(total);
for (int i=0;i<=fec.length-1; i++){
    System.out.print(payloadSolved[i]);       //This will print on screen the payload+FEC,
      that is 011010001100
}
```

With regard to the test 2 we can change one bit of the *bitsh* array, and the payloadSolved will still showing the same, because the method solveFEC will change the bit wrong to the right thanks to the FEC. For example, if we change the third bit of bitsh, (0,1,**0**,0,1,0,0,0) the result on screen will still the same. But if we change more than one bit, the result will be inconsistent. The method cannot recover the byte because of the FEC only recovers one bit per byte.

TEST 3

```
      {
        0,1,1,0,1,0,0,0        //bits representing h on ascii
       ,0,1,1,0,1,1,1,1        //bits representing o on ascii
       ,0,1,1,0,1,1,0,0        //bits representing l on ascii
       ,0,1,1,0,0,0,0,1        //bits representing a on ascii
      };
byte[] payloadArriving=Fec.convertBitsToBytes(bitsReceived);
byte[] fec=Fec.calculateFec(payloadArriving);
System.out.println(byteToString(fec)); //We print the payloadSolved converting the byte to
  string. It will print \xc6S, that are the bytes of 11000110 01010011. (S on ascii is
  01010011).
```

TEST 4

```
      {
        0,1,1,0,1,0,0,0        //bits representing ascii h
       ,0,1,1,0,1,1,1,1        //bits representing ascii o
       ,0,1,1,0,1,1,0,0        //bits representing ascii l
       ,0,1,1,0,0,0,0,1        //bits representing ascii a
      };

      int [] fec=      //hola's fec
         {1,1,0,0,     //fec of h
          0,1,1,0,     //fec of o
          0,1,0,1,     //fec of l
          0,0,1,1};    //fec of a
    byte[] payloadArriving=Fec.convertBitsToBytes(bitsReceived); //we transform the bits to
      byte, because what \glsname{AX25} will send are bytes
    byte[] fecArriving=Fec.convertBitsToBytes(fec);//we transform the bits to byte of the FEC
    byte [] payloadSolved=Fec.payloadFecSolved(payloadArriving,fecArriving); //this funtion
      check is there was any mistake, with the \acrshort{FEC}, in the payload. If it find a
      wrong bit it tranform it to wright.
    System.out.println(byteToString(payloadSolved)); //We print the payloadSolved converting
      the byte to string. I print "hola".
```

In the same way as in the test 3 we can change any bit of any of the 4 bytes tested and we will have on screen the byte correct. In this case we can change one bit of every byte and the result will still correct.

The test 3 and the test 4 have been also implemented with sequences much more longer, until the 256 bytes of maximum. And it works perfectly. This means that the FEC's recovery works, because I introduced all the possible noise, manually, and the FEC recovers all the messages perfectly.

6.2.2 Timing with FEC

About the timing with the FEC mode we have to take into account the following considerations:

- Every packet sent will have the FEC of the payload, including the first packet.
- The FEC takes the half of the payload, if the payload is 256 bytes the FEC will be 128. This means a packet of 384 bytes apart from the other fields.
- To send an image it is necessary one short packet with the length of the image and then as many packet as necessary to complete the image.
- The number of packets to send the whole image is calculated knowing that the maximum size of the payload is 256.
- To calculate the number of packets we need the length of the image divided between 256 more 1 if the division is not an exact division.
- The packets with 384 bytes of payload takes 1020ms since it is sent by the transmitter until it is analysed by the receiver (in a scenario where transmitter and receiver are connected directly as it is explained in chapter 2).
- The small packets, like the packet with the length of the image, take 15ms.
- Considering the worst case, where the last packet, which does not complete the 256 bytes, is 255 bytes, which would take the same time as a 256 payloads packet:

  $ImageTiming = (\frac{length}{256} + 1) * 1020ms + 15ms$

  length is the length of the image. And to calculate the number of packet we divided between 256 and add 1 because we are considering the rest packet as the bigger one. The real size of the last packet is

  $LastPacketTiming = length - 256 * \frac{length}{256}$.
- In this theory time is not been considered the time the transmitter takes to split the images and decide if it is necessary to send another message.
- We also need to have into account the time necessary to generate the FEC, this is 300ms if the payload field is 256 bytes.
- The computational time required is 1560ms per packet.

- This means that the final theory time would be:

  $Image\ time = (\frac{length}{256} + 1) * (120ms + 1560ms + 300ms) + 15ms$

With all these data now we do the same timing test that we do to the images, but this time with the FEC.

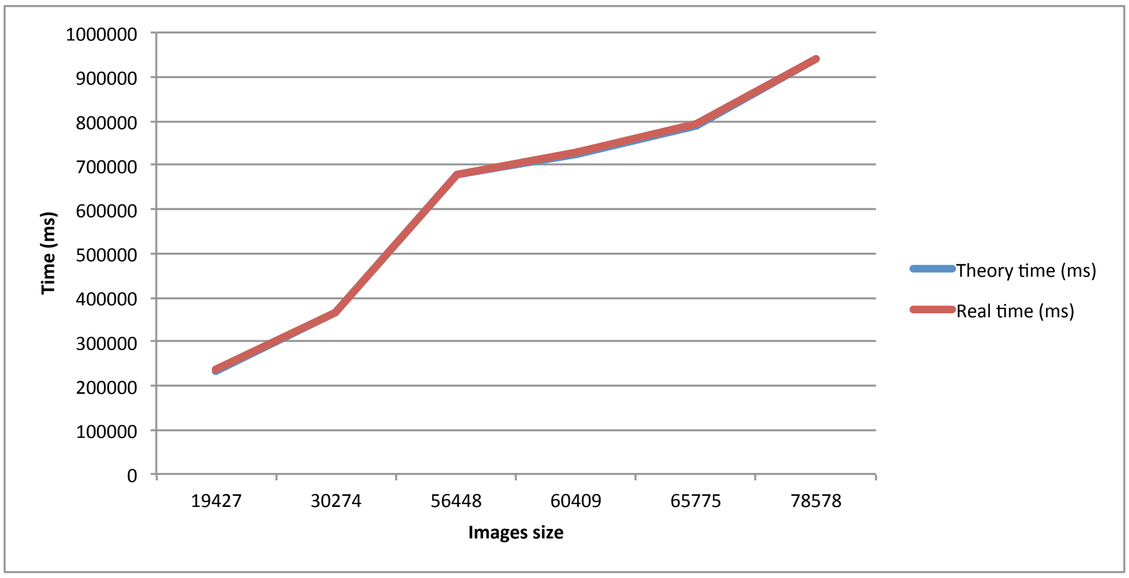


Figure 6.4 – *FEC timing*

In the graphic 6.4 we can observe that there is not much differences between the real timing and the theory timing. This means, like without the FEC, that there no much data that we could not been taking into account.

To see how different are both modes (with or without FEC) it is necessary to compare the real times between both of them.

In the graphic 6.5 we observe that there are not much difference between them. In special the average difference is 17000ms. It can be observed how as the size of the images increases, on the x-axis, the difference is increasing. This is understandable because the relationship between both modes grows progressively with respect to the length of the message.

If we compare the theory results, FEC timing less No FEC timing, we have the next expression:

$$Time\ with\ FEC - Time\ without\ FEC = \frac{length}{256} * 820ms$$

What this expression means is the longer is the message the bigger is the time difference between FEC and without FEC. This is important to be aware of because if we have to send a very big message we would have to be careful with the FEC mode. The length of the FEC

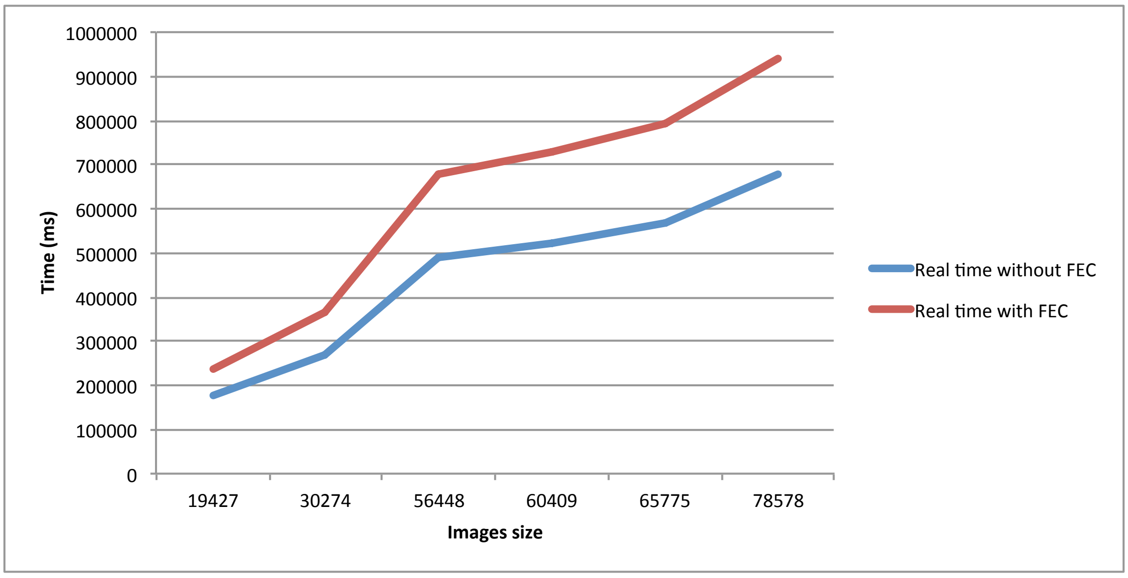


Figure 6.5 – *Timing comparison*

is directly proportional to the length of the payload, which in turn is directly proportional to the time needed to send the complete message.

6.2.3 Interferences in the link with and without FEC

To make this test is necessary to introduce some noise or interference into the link. This is a difficult task due to the link. We have a link that goes directly from the transmitter to the receiver, so we would have to introduce a lot of noise to lose our signal, but the loud of that noise makes impossible for the receiver to even notify that it is coming some information.

I first tested AX25 without FEC. I introduce, with Matlab, with the same computer I am sending the information, white noise. This was done with y = awgn (x, snr), matlab's own function But this did not affect at any way the signal received. The message was still the same at the receiver. So I try to introduce different kind of noise. But this only affected to the receiver when the noise was so loud that the receiver could not even notice the signal.

In the same way I tried with the FEC activated, but it was the same.

The purpose of the test was to receive the signal but wrongly, but the signal passes from been perfect to not been notified. For that reason this test could not verify how efficient is the FEC mode. Because the FEC can recover if it finds 1 bit of 8 wrong, but not if there is no signal, or if the signal is totally correct.

CHAPTER

7

CONCLUSIONS AND FUTURE IMPROVEMENTS

In this document has been displayed the development of a data handling, telemetry telecommand protocol for onboard computer cubesat in the context of the GranaSAT academic project.

From the point of view of the student it has been a challenge to overcome the *End of Degree Project* subject of the degree of *Telecommunication Technology Engineering*. It has been possible thanks to the skills that have been acquired in the *Escuela Tecnica Superior de Ingenieria Informatica y de Telecomunicacion*.

The protocol implemented in this project can work in a simulated scenario, but it needs to be improved to work in the real scenario. This work will be done by my colleagues in the following years. Apart from the need to implement it for a real scenario it could be done some other improvement that have been named throughout this report and are here summarized:

- Implementing a method of recovering packets. An identification number could be added to the package, that the receiver would check. If the number is not sequential to the last number the receiver would ask to the transmitter to resend the packet. This new field in the packet would take between 1 byte.
- It could be implemented a mobile application that would notify the user in which situation is the satellite. This application would be connected to the ground computer

that would advice when the satellite is sending message.

- Implement the full-duplex mode. Nowadays the GranaSAT project do not expect to send the satellite to work with full-duplex. For that reason this mode was not developed. But for that case it could be implemented just using threads when sending and receiving. In this mode both ends (the ground computer and the satellite) could send and receive at the same time.

As a final conclusion, it is worth noting the great amount of knowledge that the student considers to have acquired throughout the project work, and above all the personal satisfaction that always involves facing the challenges associated with the profession that she has been decided to exercise.

REFERENCES

[1] Amsat-uk. AMSAT Website. https://amsat-uk.org/new-members/.

[2] Convertir imagen a array de bytes, convertir array de bytes a imagen. http://nullbrainexception.blogspot.com.es/2010/12/java-convertir-imagen-array-de-bytes.html.

[3] Eclipse foundation. Eclipse Website. https://eclipse.org.

[4] Netbeans. Netbeans Website. https://netbeans.org/.

[5] Beech, W. A., Nielsen, D. E., and Taylor, J. Ax.25 link access protocol for amateur packet radio, July 1998.

[6] Deitel, P. J., and Deitel, H. M. *Java How to Program*. 2008.

[7] Gronstad, M. A. Implementation of a communication protocol for cubestar, July 2010.

[8] Liang, Y. D. *Introduction to java programming*. 2011.

[9] McGuire, J., Galysh, I., Doherty, K., Heidt, H., and Neimi, D. Forward error correction extension to ax.25 link protocol for amateur packet radio, September 2016.

[10] Menna, B. Aprs-tramas ax.25. https://es.slideshare.net/bmenna/aprs-y-tramas-ax25.

[11] Roux, J.-H. L. Development of a satellite network simulator tool and simulation of ax.25, fx.25 and a hybrid protocol for nano-satellite communications, December 2014.

[12] Tresvig, J. L., Bekkeng, T. A., and Lindem, T. Cubestar – a nanosatellite for space weather monitoring.

[13] ZIELINNSKI, B. An analytical model of tnc controller, July 2009.