Contribution to the Development of a LoRa Communications Module for the AlainSat-1 CubeSat

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Martí Fernandez Pons

In partial fulfilment

of the requirements for the degree in

TELECOMMUNICATIONS TECHNOLOGIES AND SERVICES ENGINEERING

Advisors: Sra. Lara Pilar Fernández Capón
Prof. Adriano José Camps Carmona

Barcelona, May 2022

Abstract

The Remote sensing and Interference detector with radiomeTry and vegetation Analysis (RITA) is a 1U payload that will fly onboard Alainsat-1, a 3U CubeSat. Among other experiments, it will perform a proof of concept of a LoRa custom module for space-to-Earth communications between the satellite and a terrestrial network of Internet of Things sensors.

This final degree thesis consists of the implementation of the protocols of the Media Access Control layer that will be used in the experiment mentioned above, precisely in the design, implementation, and testing of pure ALOHA, and the design of CSMA/CA with RTS/CTS.

Resum

El sensor Remot i detector d'Interferències per a radiomeTria i Anàlisi de vegetació (RITA), és una càrrega útil de 1U que volarà a bord del CubeSat de 3 Unitats Alainsat-1. Entre d'altres experiments, aquesta realitzarà un prova de concepte de la utilització d'un mòdul LoRa desenvolupat a la UPC per a comunicacions espai-terra entre el satèl·lit i una xarxa terrestre de sensors d'internet de les coses.

Aquest treball final de grau consisteix en la implementació dels protocols de la capa de control d'accés al medi que s'utilitzaran en l'experiment mencionat anteriorment; concretament amb el disseny, implementació i test de l'ALOHA pur, i el disseny del CSMA/CA amb RTS/CTS.

Resumen

El sensor Remoto y detector de Interferencias por radiomeTría i Análisis de vegetación (RITA), es una carga útil de 1U que volará a bordo del CubeSat AlainSat-1 de 3 Unidades. De entre otros experimentos, ésta realizará una prueba de concepto de la utilización de un módulo LoRa desarrollado en la UPC para las comunicaciones espacio-tierra entre el satélite i una red terrestre de sensores de internet de las cosas.

Este trabajo final de grado consiste en la implementación de los protocolos de la capa de control de acceso al medio que se utilizaran en el experimento mencionado anteriormente; concretamente en el diseño, implantación i testeo del ALOHA puro, i el diseño del CSMA/CA con RTS/CTS.

To my family and friends who have always supported me.

Acknowledgements

I would like to thank my co-tutor Lara Fernandez Capon, with whom I have been working, for her guidance during this project. I also want to thank the rest of the team at RITA for allowing me to contribute to their project.

Finally, I want to thank Adriano José Camps Carmona for making possible my collaboration in this project.

Revision history and approval record

Revision	Date	Purpose
0	26/04/2022	Document creation
1	6/04/2022	Structure and content revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Martí Fernandez Pons	marti.fernandez@estudiantat.upc.edu
Adriano José Camps Carmona	adriano.jose.camps@upc.edu
Lara Pilar Fernandez Capon	lara.fernandez.c@upc.edu

Written by:		Reviewed and approved by:	
Date	26/04/2022	Date	12/05/2022
Name	Martí Fernandez Pons	Name	Lara Fernández Capón
Position	Project Author	Position	Project Co-Supervisor
		Name	Adriano José Camps Carmona
		Position	Project Co-Supervisor

Table of contents

List of Figures

1. Introduction

1.1. Statement of purpose

With the pass of time and the onset of new technologies, the “Internet of Things” has become an increasingly popular technology. Nowadays, day-to-day objects can be connected to the Internet, allowing us to perform actions remotely or automate processes.

Even so, there are still significant challenges in the area of IoT. One of them is to accomplish global coverage, even in rural or difficult access areas where terrestrial technologies are not feasible [1]. A possible solution for this could be the use of a satellite constellation. However, due to the power limitations and the amount of these devices, not any constellation can be used. The solution best fits this scenario is a CubeSat constellation orbiting in the Earth's low orbits.

CubeSat is a technology based on integrating all the subsystems of a traditional satellite in what is known as a Unit, a cube with a side of ten centimetres and a weight equal to or less than two kilograms. In the beginning, they were used to study the behaviour of different technologies in a space environment. However, today, they are used in many services such as communications or Earth observation.

The Remote sensing and Interference detector with radiomeTry and vegetation Analysis payload, carried out by the UPC NanoSat Lab, is one of the three selected projects by the 2nd GRSS Student Grand Challenge to fly onboard Alainsat-1, a 3U CubeSat developed by United Arab Emirates’ National Space Science and Technology Centre.

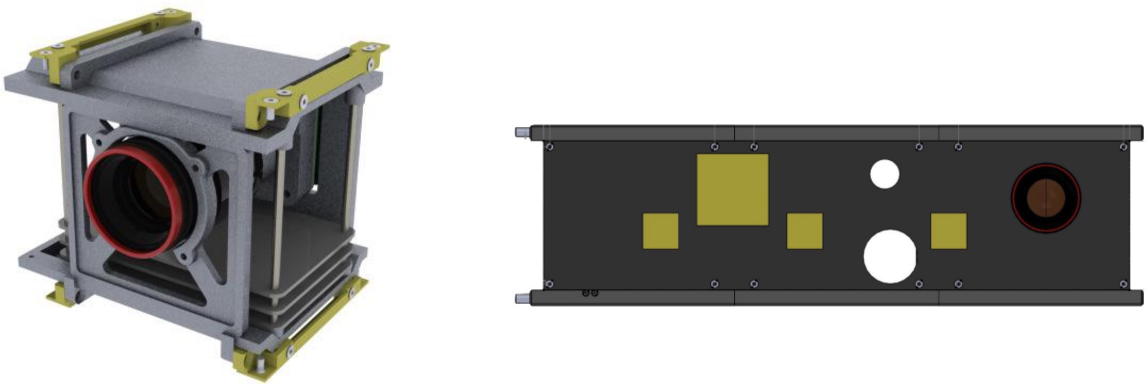


Figure 1. RITA Payload and antennas render [2]

The main objectives of RITA are to perform microwave radiometry measurements at L-band, vegetation analysis using a hyperspectral camera, and Radio-Frequency Interference (RFI) detection and classification by using a Software-Defined Radio (SDR), all with a simple and affordable CubeSat-based remote sensing payload [3].

Like some companies in the sector, such as FOSSA Systems [4] or Lacuna Space [5], that are already deploying CubeSat constellations for IoT applications, RITA will also implement a LoRa transceiver which will acquire in-situ data from a hard-to-access IoT sensors network for natural disasters monitoring as part of a multi-level remote sensing campaign. In addition, these IoT sensors will also be able to perform on-demand executions from the MicroWave Radiometer MWR and imager if necessary based on the retrieved data.

LoRa, is a wireless technology designed to connect low power consumption devices with wide coverage, but sacrificing bit rate. These conditions are accomplished using spread-spectrum modulations derived from chirp spread spectrum modulations.

At LoRa the signals are modulated using orthogonal chirps signals which are constantly and lineally jumping in frequency. Thanks to this, communications among larger distances can be achieved with the same SNR level. Furthermore, being a spread spectrum modulation makes it robust to other perturbing effects such as multipath or Doppler [6].

This makes it an interesting technology to be used in satellite communication systems, in front of others also used in IoT applications such as Sigfox or NB-IoT. Its resilience to Earth-to-space channel constraints, achieving a maximum distance of 832 km. Additionally, LoRa can be used with a variety of Media Access Control (MAC) protocols which is also an advantage in front of others modulations [7].

LoRaWAN is the MAC protocol based on a star-shaped topology designed for LoRa modulation. However, it has been demonstrated that it has certain capacity limitations in the IoT- satellite scenario. Because of the large footprint of the satellite, a large number of devices can be in range, so this scenario represents a challenge in terms of the MAC. This final degree thesis aims to design and implement MAC layer protocols that are suitable in these conditions to be used by the RITA LoRa module.

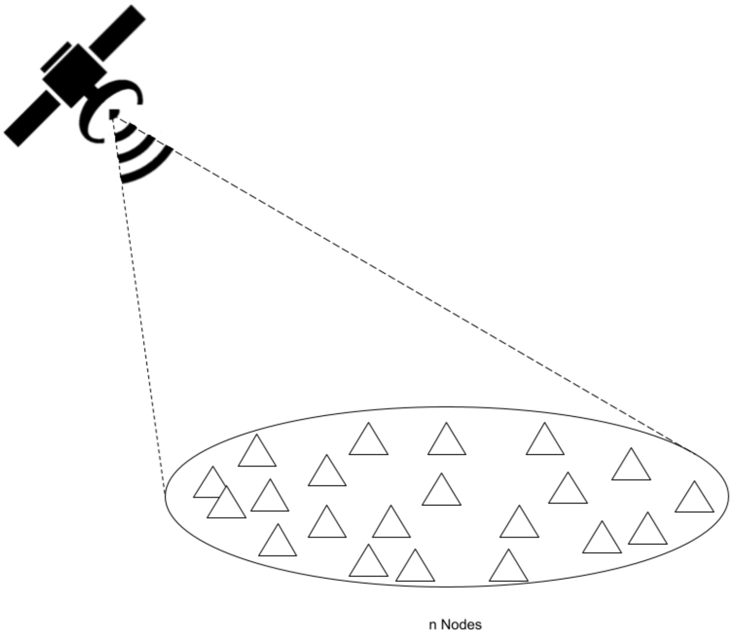


Figure 2. IoT CubeSat scenario scheme

1.2. Requirements and specifications

The main requirements and specifications of the LoRa experiment are:

- IoT sensors shall be able to transmit and receive LoRa modulation.
- IoT sensors shall be of large autonomy (a couple of years(TBD)).
- Spacecraft shall be able to transmit and send LoRa messages.
- LoRa experiment shall transmit and receive in the 868 MHz, which is the unlicensed ISM frequency band that can be used over Europe.
- LoRa module shall be able to execute on-demand other payload modules.
- Software shall be able to obtain a configuration file to set MAC protocols and modulation parameters.
- MAC protocol shall be changed depending on the configuration files of the experiment.
- Telemetry of the LoRa module shall be transmitted to the ground station using the s-band.
- UHF band shall be used to update the experiments' configuration files and do patching(TBC).
- Software shall be efficient in terms of CPU usage.
- Software shall be efficient in terms of memory usage.
- Software shall be able to identify and handle errors.
- Software shall be able to generate logs to follow its performance of it.

1.3. Work Plan

The different work packages defined and the Gantt diagram are also explained. However, due to space constraints, it was decided to move them to the beginning of the appendices.

1.3.1. Deviations from the initial plan

Regarding the division of the work into different work packages, it has to be only commented the addition of an extra package referred to the report's writing. However, the work plan has to be modified to a larger extent.

On the initial road map, it was scheduled to dedicate one month to each of the different work packages of the development. However, soon it became apparent that conducting the design, implementation, and debugging of the three protocols or even one in one month was not a realistic estimation. In addition, the first approximation to the design of the ALOHA protocol did not fit the requirements mentioned in the previous sections, and the time dedicated to it was larger than expected.

For this reason and to complete the development of the pure ALOHA protocol, I decided to extend the time of this project even though there is no time to finish the implementation and testing of CSMA/CA and the development of Slotted ALOHA protocols.

2. State of the art of the technology used or applied in this final degree project

This section presents and explains the literature consulted to understand and develop this project.

2.1. Media Access Control layer

When two or more nodes are sharing the same physical media, which is almost always the case, some of them may try to access it simultaneously, giving rise to what is known as a collision. In this case, the packets transmitted by the nodes involved may not be received at their destination.

The Media Access Control protocols and the Logical Link Control (LLC), which constitute the second layer of the Open Systems Interconnection model (OSI), are algorithms that guarantee access to the shared media in an orderly and equitably way. Even so, there is still the possibility of a collision, so MAC protocols establish a process to manage this occasion [8].

However, due to the IoT and CubeSat scenarios, not all MAC protocols are suitable. There are some additional limitations to the performance of MAC protocols which are not present in traditional satellite communications. These are related to the hardware's processing capabilities, available storage, the mobility, and the number of devices.

Unlike traditional satellites, CubeSats do not have great processing capabilities, so the proposed MAC protocols cannot involve complex processes. It has to be also taken into account channel congestion. In satellite communications, the patterns are usually one-to-one or one-to-many. In our scenario, the satellite will behave as a getaway. It will have to retrieve the packets of sensors on Earth's surface, which are unknown their location and amount (lots-to-one). Moreover, these metrics will be changing continuously due to the satellite's movement around the orbit.

Therefore, the traditionally used protocols for satellite communications based on reservation are not suitable in our scenario. This is because they are not flexible to the variations in the number of devices since they require coordination among the nodes involved. Therefore, the ALOHA-based protocols are more relevant for the MAC layer protocols in IoT scenarios due to their simplicity in terms of implementation and the low hardware requirements [9].

The existing MAC protocols usually used for IoT satellite communications can be categorised as follows:

- **Random access asynchronized protocols**: These are protocols where access to the media is performed randomly and require an ACK to confirm a correct reception. The protocols that receive this categorisation are Aloha, Enhanced Aloha (E-Aloha), Spread Spectrum Aloha (SS-Aloha), and Enhanced Spread Spectrum Aloha (E-SSA).
- **Random access synchronised protocols**: These are protocols where the channel is divided into slots of equal duration of the packet transmission time. The nodes can only transmit at the beginning of these slots. It requires synchronization

among the nodes of the network and ACK to confirm correct reception. The protocols that receive this categorisation are Slotted Aloha (S-Aloha), Contention Resolution Diversity Slotted Aloha (CRDSA), Irregular Repetition Slotted Aloha (IRSA), Coded Slotted Aloha(CSA), and Multi-slots Coded Aloha(MuSCA).

- **Medium sensing protocols**: These are protocols where the nodes sense the medium before transmitting. If it is free, it transmits; if not, it performs a random backoff and senses the medium again. Only Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) is in this category. This protocol will be explained in the following pages.
- **Reservation protocols**: These are protocols where the channel is divided into slots that nodes reserve. Only R-Aloha is in this category. Require ACK to confirm correct reception.
- **Hybrid protocols**: They are a mix of different protocols that cannot be classified in previous categories. The protocols that receive this categorisation
- are: Fixed Competitive Time Division Multiple Access (FC-TDMA) and Random Frequency Time Division Multiple Access (FTDMA).

All these above mentioned MAC protocols will accomplish the desired performance by achieving a sufficient density of sensors for each one thousand square kilometres for the case presented in this project [10]. Among them, it has been decided to implement ALOHA, Slotted ALOHA, and CSMA/CA because of their implementation simplicity in front of the others that have error correction codes or more complicated algorithms.

2.2. Protocols studied for this project

In this section, there is a more extensive explanation of the protocols to be implemented in the scope of this project:

2.2.1. Pure ALOHA

Pure ALOHA is the first and the simplest of the MAC protocols. Right when a node has a packet to send, it accesses the media and sends it without any restriction.

This approach does not address the problem of multiple access. If another node wants to transmit simultaneously, the packet will be lost. For this reason, pure ALOHA relies on an algorithm based on the repetition and the use of an extra packet called Acknowledgment (ACK) to confirm the correct reception of the useful data packet. When the transmitting node sends a packet, it starts a timer known as wait time. The duration of this time is determined by two times the propagation delay plus a certain margin and the processing time. If the ACK is received before this timer reaches zero, it would be considered a success; if not, the packet would be transmitted again.

Even so, this process does not solve all the problems. If when the wait time is over, the nodes retransmit the packet again, the situation will be the same as before. To address this, nodes will start another timer, called backoff, of an aleatory duration before retransmitting. The determination of backoff time depends on the implementation. This project selected a Binary Exponential Backoff formula which consists of taking a random number between R = [0, 2^K-1], where K is the number of attempts and multiplying by the maximum propagation delay.

There is one final scene that addresses this protocol. If every time that the attempt to send a packet fails or the ACK does not arrive, it is retried to send, this will lead to saturation of the channel at a certain moment. To avoid this casuistic, a maximum of retransmission attempts is set after which one the packet in question is descanted [8].

In our case, as the satellite behaves as the receiver, this will only perform two actions. First, it will send the Beacon to let the nodes know that they are in the range to transmit and second, the transmission of the ACK packet verifying the correct reception. In this protocol, both packets Beacon and ACK have only the common fields, which are the timestamp and the satellite identifier, and the satellite identifier, packet type, packet identifier, node identifier, and timestamp, respectively [10].

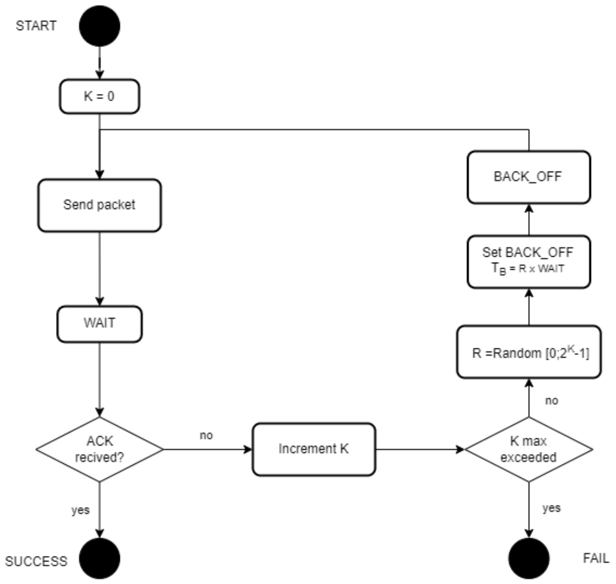


Figure 3. Pure ALOHA flow diagram.

2.2.2. Slotted ALOHA

As the name suggests, slotted ALOHA is a MAC protocol derived from the pure ALOHA to improve its performance.

Due to the lack of restriction on when to transmit, at the pure ALOHA protocol, the packet can collide with another packet previously sent or during the time that it is transmitted. This time where there is the possibility of a collision is known as vulnerable time. Considering the hypotheses explained before and considering T_p as the time to transmit a packet, the vulnerable time of pure ALOHA is two times the transmitting frame $T_v = 2{*}T_p$. The slotted ALOHA is designed to improve this vulnerable time by adding limitations to when the nodes can start to transmit.

At slotted ALOHA, the time is divided into slots of the same duration as the transmitting frame T_p, and the nodes are limited to start the transmission at the beginning of these slots. So, when a node starts to transmit, the packet can collide with packets sent by other nodes at the same slot but not with packets transmitted at the previous or the following slots. By doing this little modification to the algorithm of pure ALOHA, a vulnerable time of T_p is accomplished, which is an improvement of 50%.

Of course, this improvement does not directly improvement of the success rate from the pure ALOHA if we find ourselves in the ideal scenario [8].

As in a pure ALOHA, the satellite will behave as the receiver. Even though this time, the Beacon will have an extra field which is synchronisation, since this field is necessary to start the slots synchronously at the different nodes in the range [10].

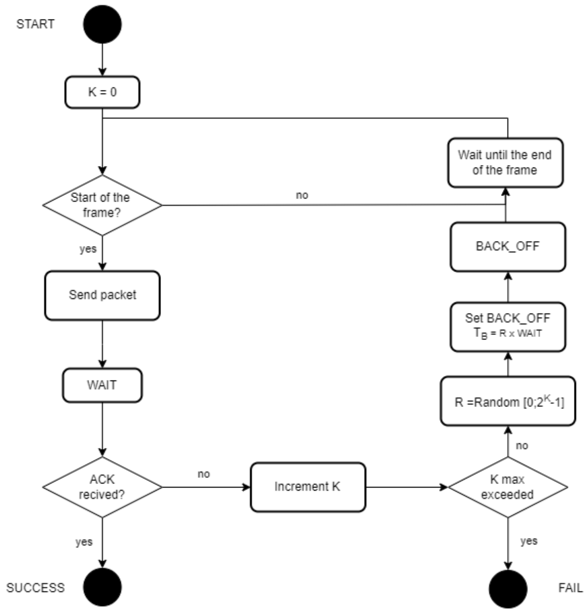


Figure 4. Slotted ALOHA flow diagram

2.2.3. Carrier Sense Multiple Access

Carrier Sense Multiple Access (CSMA) is a MAC protocol that tries to improve the performances of the ALOHA protocols by adding the limitation of sensing the channel to know if it is busy or free before transmitting.

Despite this new strategy, a collision-free channel is not accomplished. Because the transmission of a packet always has a certain delay, it may be the case that a certain node senses the media free while there is a second node that has already started to transmit a packet that has not reached the first node yet.

When a collision occurs, to handle these situations, two different procedures were defined:

- Collision Detection (CSMA/CD): in this algorithm, the media is sensed not only before it starts but also during the transmission simultaneous to the sending of the packet. It also added the transmission of a jamming signal to notify the other nodes of the network when a collision has occurred.
- Collision Avoidance (CSMA/CA): it modifies the pure ALOHA by adding two extra packets and interframe waits.

2.2.3.1. Carrier Sense Multiple Access with Collision Avoidance

As it is mentioned before, CSMA/CA relays on two extra packets, Request To Send (RTS) and Clear To Send (CTS), and interframe wait, DFC InterFrame Space (DIFS), and Short InterFrame Space (SIFS).

Before explaining the algorithm, let us explain the different components. RTS and CTS are control packets whose purpose is to inform the network that communication will start. The information contained the identifier of the nodes involved in the process and the duration while the channel will be occupied, known as Network Allocation Vector (NAV).

DIFS and SIFS are established to solve the vulnerable time. SIFS is set by the maximum delay of a transmitted packet to reach the most distant node. It is performed before the transmission of each packet once started the process. DIFS is the sum of this delay time plus an extra time defined by the binary exponential formula already explained, and it is only performed at the beginning of the process.

Once these components have been described, let us start the algorithm. At the beginning of a transmission, the node continuously senses the channel to verify if it is busy. When it is free, the node waits for a DIFS, transmits RTS to reserve the channel, and sets a timer.

When the receiving node receives an RTS, it waits for SIFS, and answers with a CTS confirming that it is ready. If the transmitting node receives a CTS before the timer reaches zero, it means the reserve of the channel is correctly achieved, and the communication process can start. If not, the transmitting node initiates a backoff process already explained at pure ALOHA.

Following the exchange of packets described above, once the CTS is received, the data packet is sent. If the ACK is received before the wait time ends, the process has been a success; if not, a backoff is started after which the data packet is retransmitted. This process is repeated until the correct reception of ACK or the number of attempts is exceeded [8].

In addition to the already mentioned packets in ALOHA, the satellite will also have to send the CTS packet in this protocol. This one is formed by satellite identifier, packet type, packet identifier, node identifier, timestamp and the NAV. Because of the broadcast nature of satellite communications, CTS will be the one which reserves the channel in our scenario [10].

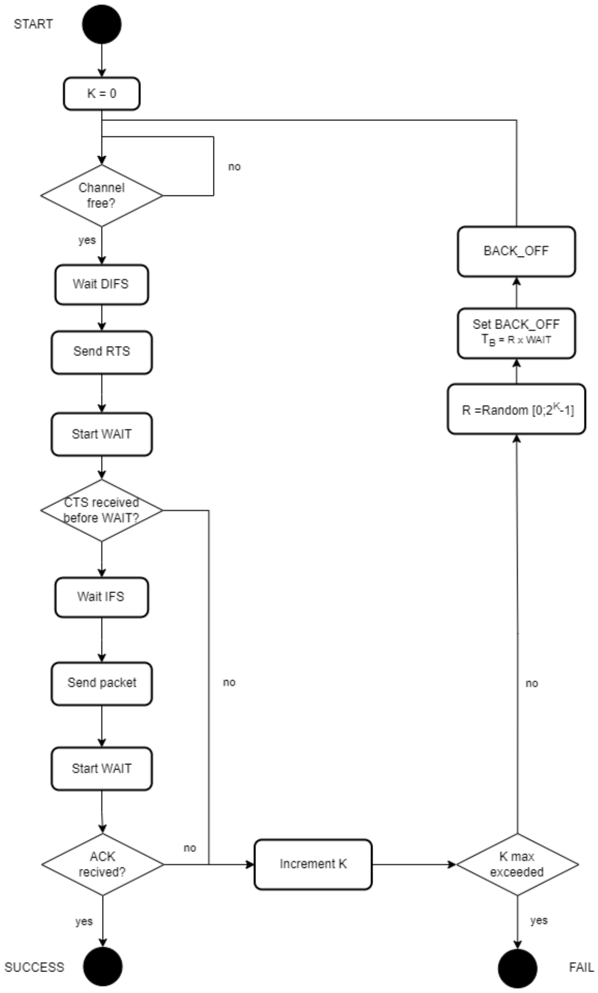


Figure 5. CSMA/CA with RTS/CTS flow diagram

3. Methodology

This section presents the methodology that has been followed and the software development process are presented.

3.1. Scope

The goal is to implement an experiment consisting of using a CubeSat as a communications relay between an IoT sensors network and a central station. These sensors will be distributed around the Earth’s surface and will take measurements of different parameters to predict, for example, natural disasters.

Due to the high quantity of sensors and constant change in the number of devices in the range, this scenario represents a challenge for the current MAC protocols. The main goal of this project is to develop a software which implements MAC protocols capable of handling the situation described above that can be used on both sides, on the satellite and the ground sensors.

3.2. General design

This software will have to work together with *LoraUtils*, which is the class that has been developed to implement the LoRa Modulation process.

The first approach that was taken to the application’s design as a joint was to implement three classes run by three independent *threads*. These will be organized in *LoraUtils*, modulation, *Protocol*, which will manage the packets, and *LoraExp*, which will be in charge of managing the useful data.

However, that is not the best solution. Due to the onboard processing capabilities of the satellite, running three *threads* for this process is a bit excessive. In addition, *LoraExp* was not needed to be running all the time, only when a new packet had to be processed.

For this reason, a new architecture was adapted. Only two threads, *LoraUtils* and *Protocol,* and *LoraExp* will be an inner class of *Protocol* called when packets have to be processed.

Referred also to the architecture, because we wanted to implement more than one MAC protocol, it was decided to use inheritance to reuse code and reduce the memory usage, another limited resource onboard the satellite. In this way, *Protocol* is an interface extended by *AlohaProtocol*, *CsmaCaProtocol*, and *SlottedAlohaProtocol*.

In addition to the protocols’ algorithms, the *Protocol* will also have to implement the periodic transmission of a beacon because of the CubeSat scenario. This packet, which components will be explained after, aims to inform the nodes if they are in the satellite’s footprint not to transmit data if there is no one to receive it.

All the data collected from the sensors will be downloaded to the ground station throw an SDR that transmits at S-band. For this reason, this protocol will record a log where all the data collected from the sensors will be written. Moreover, the actions performed by the MAC will also be recorded to do maintenance at the ground station. The *spdlog* third-party library has been used for this purpose [11].

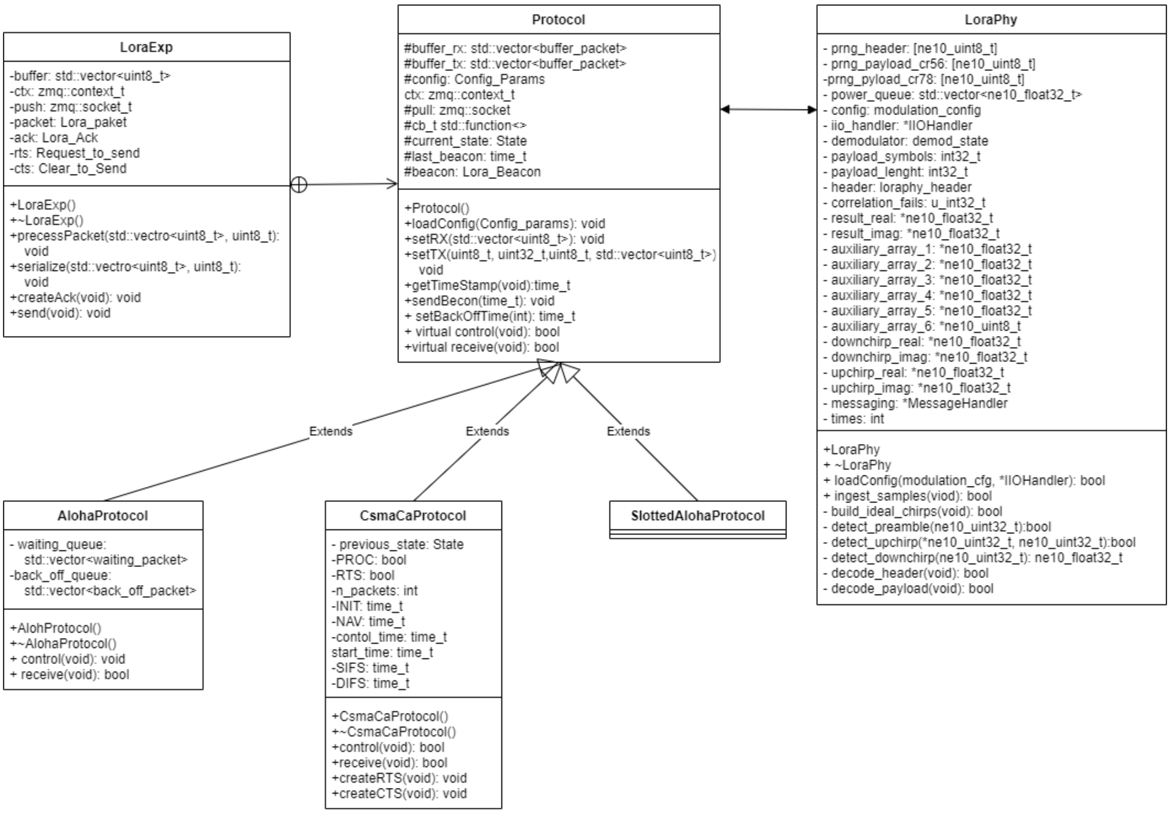

LoraExp
-buffer: std::vector<uint8_t>
-ctx: zmq::context_t
-push: zmq::socket_t
-packet: Lora_paket
-ack: Lora_Ack
-rts: Request_to_send
-cts: Clear_to_Send
+LoraExp()
+~LoraExp()
+precessPacket(std::vectro<uint8_t>, uint8_t): void
+serialize(std::vectro<uint8_t>, uint8_t): void
+createAck(void): void
+send(void): void
Protocol
#buffer_rx: std::vector<buffer_packet>
#buffer_tx: std::vector<buffer_packet>
#config: Config_Params
ctx: zmq::context_t
#pull: zmq::socket
#cb_t std::function<>
#current_state: State
#last_beacon: time_t
#beacon: Lora_Beacon
+Protocol()
+loadConfig(Config_params): void
+setRX(std::vector<uint8_t>): void
+setTX(uint8_t, uint32_t,uint8_t, std::vector<uint8_t>) void
+getTimeStamp(void):time_t
+sendBecon(time_t): void
+ setBackOffTime(int): time_t
+ virtual control(void): bool
+virtual receive(void): bool
LoraPhy
- prng_header: [ne10_uint8_t]
- prng_payload_cr56: [ne10_uint8_t]
-prng_pyload_cr78: [ne10_uint8_t]
- power_queue: std::vector<ne10_float32_t>
- config: modulation_config
- iio_handler: *IIOHandler
- demodulator: demod_state
- payload_symbols: int32_t
- payload_lenght: int32_t
- header: loraphy_header
- correlation_fails: u_int32_t
- result_real: *ne10_float32_t
- result_imag: *ne10_float32_t
- auxiliary_array_1: *ne10_float32_t
- auxiliary_array_2: *ne10_float32_t
- auxiliary_array_3: *ne10_float32_t
- auxiliary_array_4: *ne10_float32_t
- auxiliary_array_5: *ne10_float32_t
- auxiliary_array_6: *ne10_uint8_t
- downchirp_real: *ne10_float32_t
- downchirp_imag: *ne10_float32_t
- upchirp_real: *ne10_float32_t
- upchirp_imag: *ne10_float32_t
- messaging: *MessageHandler
- times: int
+LoraPhy
+ ~LoraPhy
+ loadConfig(modulation_cfg, *IIOHandler): bool
+ ingest_samples(viod): bool
- build_ideal_chirps(void): bool
- detect_preamble(ne10_uint32_t):bool
- detect_upchirp(*ne10_uint32_t, ne10_uint32_t):bool
- detect_downchirp(ne10_uint32_t): ne10_float32_t
- decode_header(void): bool
- decode_payload(void): bool
Extends
Extends
Extends
AlohaProtocol
- waiting_queue: std::vector<waiting_packet>
-back_off_queue: std::vector<back_off_packet>
+AlohProtocol()
+~AlohaProtocol()
+ control(void): void
+ receive(void): bool
CsmaCaProtocol
- previous_state: State
-PROC: bool
-RTS: bool
-n_packets: int
-INIT: time_t
-NAV: time_t
-contol_time: time_t
start_time: time_t
-SIFS: time_t
-DIFS: time_t
+CsmaCaProtocol()
+~CsmaCaProtocol()
+control(void): bool
+receive(void): bool
+createRTS(void): void
+createCTS(void): void
SlottedAlohaProtocol

Figure 6. UML diagram

Data management is an issue that has also been taken into consideration. The received packets have a structure and include a set of data that differs depending on the packet type. Also, having the necessary information groped together can be advantageous for certain processes. Because of this, different custom types have been defined to manage data in an orderly and effective way. That are listed below:

- *Lora_Beacon*: it consists of the union of an array of size ten and a struct. This struct contains the timestamp (*uint64_t*) and the satellite identifier (*uint16_t*). In the case of Slotted Aloha, it will also contain synchronization (*uint16_t*) to ensure the beginning of the slots are simultaneously for all nodes.
- *Protocol_Header*: it is a union that contains the necessary information to manage the packet. It is formed by a struct that has the satellite identifier, the type of the packet (*uint8_t*), the packet identifier (*uint8_t*), and the node identifier (*uint32_t*). This structure is used to decide how the MAC protocol should manage the packet.
- *Buffer_Packet*: it is how the packets are stored at the buffer. It is a struct that contains the header of the packet (*Protocol_Header*), the number of attempts (*uint*), and a vector to store the packet (*uint8_t).*
- *Waiting_Packet*: it is the form in which packets that are waiting for a response are stored. It consists of a struct that has the packet type, the packet identifier, the node identifier, the time at which has been sent the packet (time_t), the number of attempts, and a vector to store the packet.
- *Back_Off_Packet*: it is the form in which packets that are in backoff are stored. It consists of a struct that has the packet type, the packet identifier, the node identifier,

the time at which has started the backoff (*time_t),* the duration of this backoff (*time_t*), the number of attempts, and a vector to store the packet.

- *Config_Params*: they are the parameters to configure the protocol. It consists of a struct that has the satellite identifier, the beacon period (*time_t*), the wait time (*time_t*), the maximum backoff time (*time_t*), and the maximum number of backoff (*uint*). In this packet, there is also the pointer to the *LoraExp* object.
- *Lora_Packet*: is the definition of the data packets that will be used. It is a union that contains the satellite identifier, the type of the packet, the packet identifier, the node identifier, the timestamp, the position in x (*uint32_t*), position in y (*uint32_t*), and the position in z (*uint16_t*) of the transmitting node and the sensor data stored in an array of uint8_t (at this moment is set to 50 positions).
- *Lora_Ack*: is the definition of the ACK packets that will be used. It is a union that contains the satellite identifier, the type of the packet, the packet identifier, the node identifier, the timestamp, and free slots (*uint16_t*).
- *Request_to_Send* and *Clear_to_Send*: this is the definition of the control packets used in CSMA/CA. It is a union that contains the satellite identifier, the type of the packet, the packet identifier, the node identifier, the timestamp, and the time to reserve the channel (*uint16_t*).
- State: is an enumeration of the different states. (IDLE, RECEIVE, SEND, STOP)
- *Packet_type*: is an enumeration of the different packet types. (DATA_PACKET, ACK, RTS, CTS)

These unions and structs have been defined at Protocol and LoraExp to have the most general domain possible.

A further explanation of the previously mentioned classes' attributes, methods, and connections can be found in the sections below.

3.3. LoraExp design

LoraExp is a simple class designed to implement the actions of the Application layer. The goal is that *Protocol* can execute a constant *control*, so the time is not wasted processing the data of packets, and it enhances its performance. Therefore, this class is in charge of registering the data from the received packets in the log sanded to the ground station. It will also create the ACK packet to confirm the correct reception of data packets.

The communication between both classes is implemented by performing a callback to the *processPacket* function from the Protocol side and using ZMQ sockets from *the LoraExp* side.

Depending on the context where sockets are used or how the packets are distributed among them, ZMQ offers us six types of patterns or couples of sockets: Request-Reply, Dealer-Router, Publish-Subscribe, XPublish-XSubscribe, Push-Pull, Exclusive pair.

At the Request-Replay pattern, the communication between sockets is bidirectional, meaning packets go in both directions. In this pattern, the Request socket always starts the communication, sends a packet to the Reply and then responds synchronously. The Dealer-Router pattern has the same functionality as the pattern explained before. However, it gives the possibility to implement the communication asynchronously. Moreover, because both patterns have the same communication algorithm, Request/Dealer and Replay/Router can be connected, but always like this.

Publish-Subscribe and XPublish-XSubscribe are one-to-many sockets. At this pattern, a Subscribe socket has to be subscribed to a Publish socket; if not, the first one cannot receive the message sent by the second one. The difference between both is that in the XPublish-XSubscrib pattern, the subscription to a Publish socket can be performed by sending a message to it, which cannot be done in Publish-Subscribe. However, they can be used together.

The Push-Pull pattern is compared with a pipeline; it directly connects both sockets. So, the push socket can only fill the pipeline with packets, while the pull socket can only retrieve packets. The Exclusive pair is also a one-to-one socket pattern like this last one. However, the difference between both lies in that the Exclusive pair has been specifically designed to be used different throw threads safely.

For our case, because the communication from Protocol to LoraExp is done performing a callback, Request-Reply and Dealer-Router are discarded since the communication by socket is not bidirectional. Also, Publish-Subscribe and XPublish-XSubscrib are discarded because they are designed for one-to-many, and this is one-to-one communication. Finally, the Exclusive pair pattern is discarded because there is no need to be thread-safe. So, the communication from LoraExp to Protocol, among the different options that ZMQ offers, will be performed by the Push-Pull pattern, which is the one that better fits the functionality that it is searched for [12].

The exchange of information between the different classes, *LoraUtils*, *Protocol*, and *LoraExp*, is performed using vectors of *uint8_t*. This is because *uint8_t* is the minimum data type storable and is also used by ZMQ [13]. As for the vectors, they are used because they have some very useful properties already implemented at the standard, such as the adding and removal of elements.

The different types of packets are defined in this class as a *union* between an array of *uint8_t* and a *struct* that contains an attribute for each one of the fields of the packets. These are organized in such a form to avoid alignment problems and separate between useful information and the MAC information used to handle packets.

Despite the good properties of vectors, they cannot be used to define a *union*. It is needed to have the same size in terms of storage between the fields of the *union* to establish it. For this reason, the vectors received from *Protocol* are converted into an array. Then, the use of *union* grants us the possibility to convert a succession of ordered bytes that cannot be understood directly to a container with the same information correctly separated by fields and ready to be logged.

3.4. <u>Protocol</u>

This class is designed as an interface where all the common attributes and methods, as well as a group of custom types of variables, are defined and which the different protocols will extend and inherit.

It has two vectors of Buffer_Pakets to store the received packets, buffer_rx, and the packets to send, buffer_tx. It also has a Config_Params attribute where the parameters of the protocol set by the ground operator are stored, the definition of the ZMQ socket to retrieve the packets from LoraExp, and the call back to process packets. And finally, the beacon packet and a time_t to store when the last one has been sent.

Of course, since the control method is implemented as a state machine, a variable to manage the state of this one is needed. This is defined as a State variable and is called *current_state*.

The functions implemented in this class are:

- *loadConfig*: this function has a *Config_Params* as an argument and returns void. In this method, the configuration parameters and the initial state are set. It is also created the call back to processPacket from LoraExp.
- *setRX*: this function has a vector as an argument and returns void. This function creates a *Buffer_Packet* and adds it to the *buffer_rx*.
- *setTX*: this method returns void, and its arguments are: an *uint8 _t* for packet type, an *uint32_t*d for the node identifier, an *uint8_t* for the packet identifier, and a vector for the data. This function creates a *Buffer_Packet* and adds it to the *buffer_tx*.
- *getTimestamp*: this function does not have arguments and returns a *time_t* which is the Unix timestamp in milliseconds of the system.
- *sendBeacon*: this method returns void and takes the time stamp as an argument. This function adds the timestamp to the beacon and sends it to *LoraUtils*.
- *setBackOffTime*: this function takes the number of attempts K realized of this packet, obtains a random number between R = [0, 2^{K}-1], and multiplies it by the waiting time. This is the value that returns the duration of the backoff.
- *printBufferPacket, printWaitingPacket, and PrintBackOffPacket*: these functions print the different values of the packets at the log. They are purely for debugging purposes.

In addition, there is also the declaration as virtual of *control* and *receive,* which will be implemented at the heir class.

3.5. Pure Aloha design

As is commented before, the *AlohaProtocol* class inherits from *Protocol*; therefore, all the components and methods defined before are also the propriety of this one. So, in this class, the main work consists of the redefinition of the virtual methods *of control* and *receive*.

Before going into the details on how these methods have been implemented, the addition of two extra components concerning the interface has to be mentioned. Waiting and Backoff queue are two vectors where the packets waiting for an ACK or at the backoff process are stored to manage the time. The variables stored are of type *Waiting_Packet* and *Back_Off_Packet*, respectively, which are structs with the necessary information for the process. *Waiting_Packet* is formed by *packet_type*, *packet_id*, *node_id*, *send_time*, *n_backoffs*, and *packet*, a vector storing the useful data. Back_Off_Packet the same information listed above, changing the *send_time* for *start_back_off* and adding *back_off_time*, which is the duration of the backoff.

Control methods are implemented as a machine of states of three different possibilities: *IDLE*, *RECEIVE*, and *SEND*.

Before entering any state, *getTimestamp* is called to obtain the Unix time and stored in *unix_timestamp_ms*. It is also checked if the *beacon_period* is exceeded, in which case it is called *sendBeacon*. Finally, if there are any packets from LoraExp, they are retrieved by calling to *receive* until there are no more packets at the socket.

At IDLE state, first, the *waiting_queue* is revised. In the case of a *Waiting_Packet* has finished the *waiting_time*, which means that the actual time, *unix_timestamp_ms*, minus the initial time, *send_time*, is bigger than *waiting_time*, it is checked if the maximum number of attempts has been exceeded. If so, sending the packet is aborted, which means that the packet is removed from the waiting queue and is not attempted to be sent anymore. If not, a *Back_off_Packet* is created and added to the *back_off_queue*, setting the *start_back_off* as the actual time, the *back_off_time* by calling *setBackOffTime* and adding 1 to *n_backoffs*. Then, the *Waiting_Packet* is eliminated from the *waiting_queue*. Because the *waiting_time* is equal for all the packets, and these are added to the queue according to the order in which they have been sent, from the first packet that has not finished, the waiting time forward is not necessary to continue checking so the program moves to next step which is revising the *back_off_queue*.

Due to the backoff time being random for each packet, unlike the waiting queue, all packets stored at the *back_off_queue* have to be checked. If a *Back_Off_Packet* has exceeded the *back_off_time*, it is created a new *Buffer_Packed*, added to the *buffer_tx*, and removed from *back_off_queue*.

Finally, it is checked if there are packets to process in the buffers. If *buffer_rx* is not empty, the *current_state* is changed to *RECEIVE*; if it is and *buffer_tx* is not empty, it is changed to *SEND*. If both are empty, the *current_state* remains *IDLE*.

At RECEIVE state, it is taken the first *Buffer_Packet* of *buffer_rx* and its *header* is obtained. Then is verified if the packet is addressed to us by comparing the *satellite_id.* If the *satellite_id* coincides, which means is it addressed to us, the packet processed may be an ACK or a data packet. If the packet is not addressed to us, it is discarded and removed from the *buffer_rx* without processing it.

If it is a data packet, it is made a callback *LoraExp* to process it. If it is an ACK, it is verified if the *node_id* and *packet_id* coincide with a packet from *waiting_queue*. If the data packet that is corroborating its correct reception is not founded because the *wait_time* has already finished, the packet is discarded. If it is found, the packet is eliminated from the *waiting_queue*, and it is performed a callback to *LoraExp* to process the packet from *buffer_rx*. In all cases, in the end, the packet is removed from the *buffer_rx*.

Finally, if there are more packets at *buffer_rx*, the state remains to *RECEIVE;* if not, it changes to *IDLE*.

The last and the simplest state is *SEND*. The *header* and *unix_timestamp_ms* are converted to a vector of *uint8_t* and added to the useful data stored at the *packet* of the first *Buffer_Packet* of *buffer_tx,* and then it is sent to *LoraUtils*. If it is a data packet, a *Waiting_Packet* is created and added to the *waiting_queue*. Finally, as in *RECEIVE*, if there are more packets at *buffer_tx,* the state remains *to SEND*; if not, it changes to *IDLE*.

The other method implemented is *receive*. At this method, all the packets at the *pull* socket of the class are retrieved. It is made using the properties that ZMQ offers by using the flag *ZMQ_NONBLOKING*, which makes the socket not have a blocking behaviour. So, when calling *zmq::receive*, if there is a packet to receive, it returns true, a *Buffer_Packet* is created and added to *buffer_tx* at the end, and this process is repeated. If not, it returns false and gets out of the method.

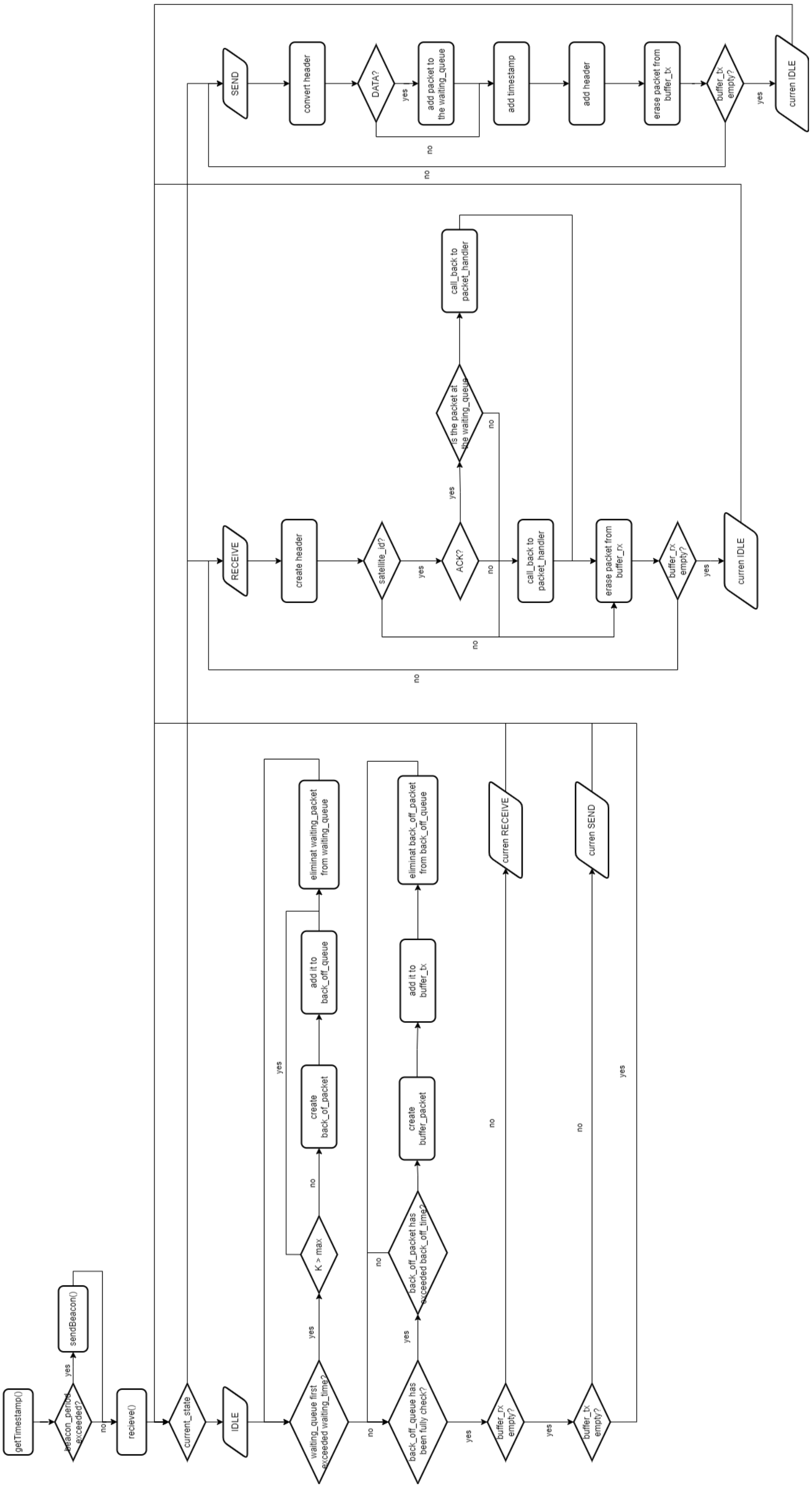


Figure 7. Control flow method diagram of AlohaProtocol

3.6. Carrier Sense Multiple Access with Collision Avoidance design

CSMA/CA has been designed following the same scheme as pure ALOHA, which means that the managing of packets is only performed at *RECEIVE* and *SEND*, and it is in the *IDLE* state where the different waiting and processes are performed.

Due to the complexity of this protocol, several flags and times have to be added. Because the node has just sent or received a packet, it has to be done one thing or another; it is important to know which is the estate of the previous execution, so a new *State* variable called *previous_state* has been added to the list of attributes. Two more flags, PROC and RTS, have also been added. The first is a Boolean that is true if the node is involved in the process which has the channel reserved. The second one indicates that the node is trying to start a new process.

It has also been defined as one variable of type *time_t* to store each of the different times that are used in the algorithm and are only calculated once at the beginning or are used simultaneously, that are: *DIFS*, *SIFS, INIT*, which is the initial time of the process and *NAV*. For both remaining times, wait and back off, it has been defined *as control_time*. And finally, to update the time, three references are needed: initial, duration, and current time, *start_time* has been defined, and it is used to store the initial time of any one of the time processes, except *NAV* which one has *INIT* for this purpose.

As at *AlohaProtocol, control* is the main method also implemented as a state machine. Before entering any state, the Unix timestamp of the system is obtained, the beacon is sent if the *beacon_period* has been exceeded, and packets coming from *LoraExp* are retrieved if there are any.

Once entered in the *IDLE* state, the different times are updated if necessary. Then NAV is checked. If NAV has not been exceeded, but PROC is false, which means that there is an ongoing process in which the node does not participate, it gets out of the IDLE state, performing a loop until NAV is finished.

If the PROC flag is true, depending on the *previous_state* different actions are performed. If *previous_state* is *RECEIVE*, first it is checked if *SIFS* has finished; if not, it starts a loop that consists of getting in *IDLE*, checking *SIFS*, and getting out until *SIFS* has been exceeded. Once *SIFS* is finished, the *current_state* is changed to *SEND* and the *previous_state* to *IDLE*.

If *the previous_state* is *SEND*, it is checked if the wait time has finished. If it has not finished, it is verified if a new packet has been received. If it is assertive, the current state is changed to *RECEIVE*; if not, it does nothing. This process is repeated until a packet is received or wait time is finished, in which cases it is set a backoff and the *previous_state* is changed to *IDLE*.

If the *previous_state* is *IDLE,* it is checked if backoff has finished, and when it is, the *current_state* is changed to *SEND* and *previous_state* to *IDLE*.

If *NAV* has not been exceeded, but *PROC* is true, that means that the communication has not finished yet, but the channel is not reserved more. So, the communication is aborted, which means that *PROC* is set to false, all times are set to zero, and the packet from *buffer_tx* is removed.

If any of the previous cases is not the actual situation, it is checked by the *RTS* flag. If it is true, a loop is performed where it is checked if, while a backoff time, new packets have been received. This is how is modelled the active sense in this software. If a new packet is received before the backoff time, the number of packets at *buffer_rx* is reset, and a new backoff is started. If not, it waits for *DIFS*, and the *current_state* is changed to *SEND* and *the previous state* to *IDLE*.

Finally, two more casuistic are contemplated. If *buffer_rx* is not empty, the *current_state* is changed to *RECEIVE* and *the previous state* to *IDLE*. Alternatively, if *buffer_tx* is not empty, in which case, if it is needed, *createRTS* is called, the number of packets at *buffer_rx* is stored, a backoff time is set, and the *RTS* flag is set to *true*, and *DIFS* and *SIFS* are established.

At *RECEIVE,* it is taken the first *Buffer_Packet* of *buffer_rx;* its *header* is obtained and is checked if it is of *packet_type* CTS.

If it is a CTS and addressed to us, *PROC* is set to true. If it is not addressed to us, *PROC* is set to false. In both cases, *NAV* time is established to the time obtained from the packet, *INIT* is set at the current time, and the RTS at the *buffer_tx* is removed. Usually, at CSMA/CA, the channel is reserved by the RTS, but in this software, it is reserved by the CTS. This is because, in our scenario, the satellite will be the one to send the CTS, thus acquiring a broadcast nature which is not accomplished by land nodes which will send RTS. In addition, land nodes are further apart than the usual case and at difficult accesses, which increases the probability of the hidden node effect, so, because of the more extensive coverage of the satellite on behalf of land nodes, CTS is better to reserve the media.

If the packet is not a CTS but is addressed to us, depending on the *packet_type* different actions are performed. If it is a data packet, it is sent to *LoraExp* to be processed. If it is an ACK, it is verified if the *node_id* and *packet_id* match with the data packet stored at *buffer_tx* and in that case, it is sent to *LoraExp* to be processed, and *PROC* is set to *false*. If it is an RTS, the response packet CTS is created, *PROC* is set to *true,* and *NAV* and INIT are established to the correspondent time. In the three before-mentioned cases, *SIFS* time is reset, and the *current_state* is changed to *IDLE* and *previous_state* to *SEND*.

If the packet is not addressed to us or is of ACK type, but the *node_id* and *packet_id* do not match with the packet to verify, the *current_state* is changed to *IDLE*, but *the previous_state* is not modified. This is to avoid getting out of the wait time if it is waiting for a packet. At last, before getting out from *SEND* state, the packet processed is always deleted from the *buffer_rx*.

When there is a packet to transmit, the state is set to *SEND*. At this state, for the first *Buffer_Packet* of *buffer_tx*, it is verified if the maximum number of attempts has been exceeded. If it is the case, the process is aborted, and current and previous states are set to *IDLE*. If this is not the case, the header and timestamp are added to the data, the packet is sent to *LoraUtils*, the wait time is established, the number of attempts is increased by one, and *current_state* is set to *IDLE* and *previous_state* to *SEND*.

As in *AlohaProtocol,* the *receive* method is in charge retrieve the packets from LoraExp, so its implementation is the same. However, this time, the packet received is added at the beginning of *buffer_tx*. This is because, in contrast to the ALOHA, at CSMA/CA, a sequence of packets has to be followed.

In this protocol, there are two extra packets which are in charge of the channel reservation process. Due to this control packets necessity, other two methods have been added to *CsmaCaProtocol* concerning *Protocol*: *createRTS* and *createCTS*. The first one is called at the beginning of the process and creates an RTS type packet using the information of the first packet of *buffer_tx,* which is the packet desired to send, and add it to the beginning. In the case of *createCTS*, it is called while processing an RTS at *RECEIVE* state. The procedure is the same as *createCTS,* but this time using the information of the RTS that has been processed to which is answering.

Because this class has not been implemented yet, it has to be mentioned that both attributes and method or design can be subjected to changes in the future.

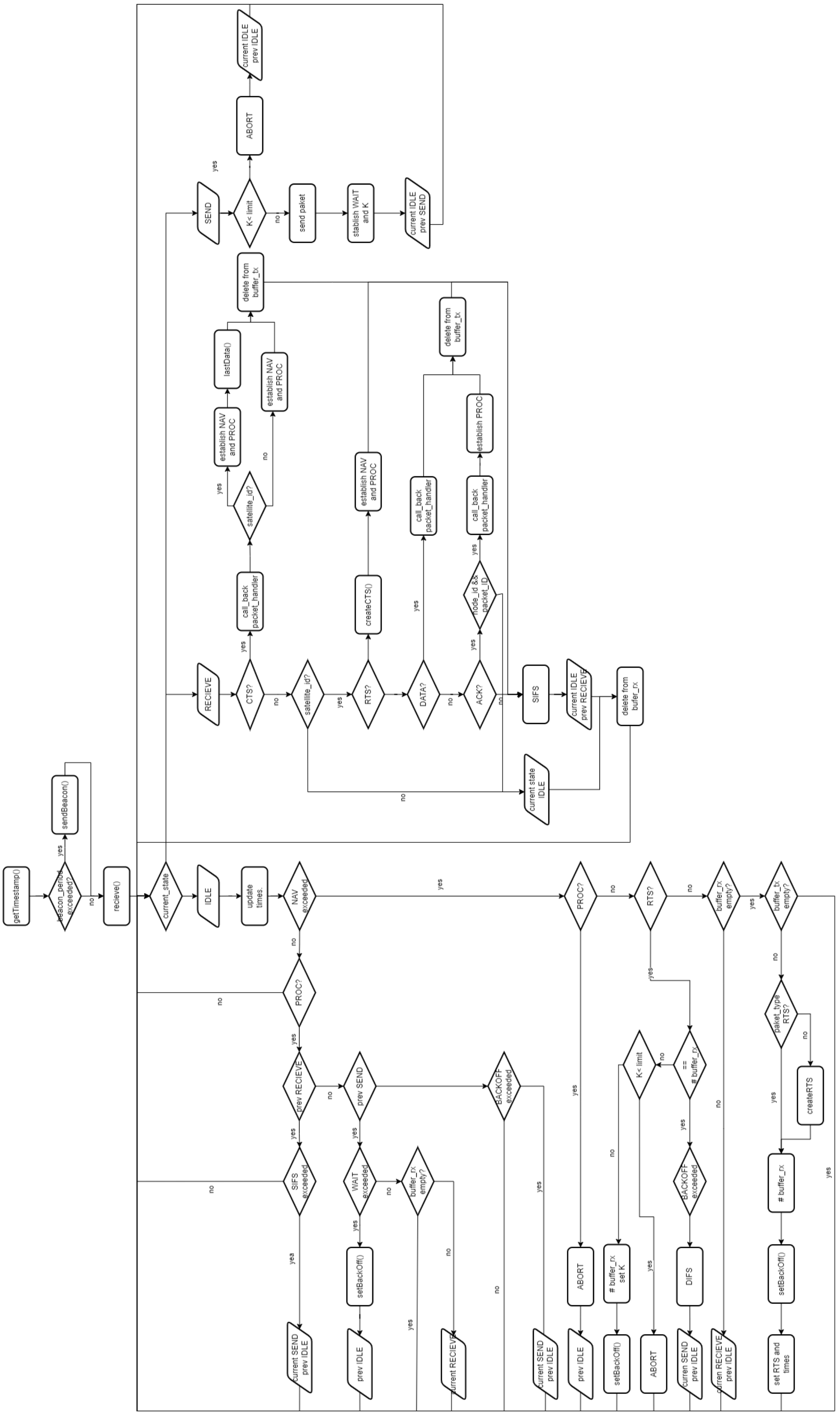


Figure 8. Control method flow diagram of CsmaCaProtocol

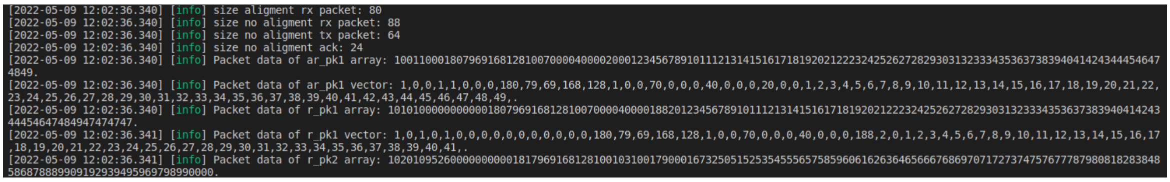
4. Results

This section presents the results obtained about the correct definition and implementation after performing a series of tests on a computer and the Pluto-SDR.

4.1. Unit Testing

4.1.1. Basic functionalities

Before proving if the software is capable of correctly managing the situations that it will have to face during its performance, it was verified if the custom types were correctly defined. The order of the packets' header components should be adjusted from the packets proposed in [3] to avoid alignment problems. As seen in the following capture, the packet with the wrong ordering of the header (the second one) adds 8 bytes between the useful data, which provokes that the packet was not correctly processed by *AlohaProtocol* or *LoraExp*.

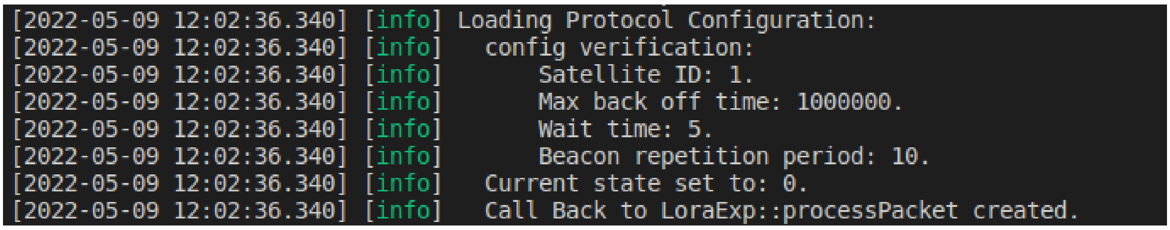


```
[2022-05-09 12:02:36.340] [info] size aligment rx packet: 80
[2022-05-09 12:02:36.340] [info] size no aligment rx packet: 88
[2022-05-09 12:02:36.340] [info] size no aligment tx packet: 64
[2022-05-09 12:02:36.340] [info] size no aligment ack: 24
[2022-05-09 12:02:36.340] [info] Packet data of ar_pk1 array: 100110001807969168128100700004000020001234567891011121314151617181920212223242526272829303132333435363738394041424344454647
4849.
[2022-05-09 12:02:36.340] [info] Packet data of ar_pk1 vector: 1,0,0,1,1,0,0,0,180,79,69,168,128,1,0,0,70,0,0,0,40,0,0,0,20,0,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,.
[2022-05-09 12:02:36.340] [info] Packet data of r_pk1 array: 10101000000000000180796916812810070000400001882012345678910111213141516171819202122232425262728293031323334353637383940414243
44454647484947474747.
[2022-05-09 12:02:36.341] [info] Packet data of r_pk1 vector: 1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,180,79,69,168,128,1,0,0,70,0,0,0,40,0,0,0,188,2,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17
,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,.
[2022-05-09 12:02:36.341] [info] Packet data of r_pk2 array: 1020109526000000000018179691681281001031001790001673250515253545556575859606162636465666768697071727374757677787980818283848
586878889909192939495969798990000.
```

Figure 9. Size of the custom types.

In the previous picture, it can also be seen that the size of the packets is a little bit bigger than the sum of the bytes of its fields. The extra bytes are known as padding, and they are added to conform the struct to its biggest component. In our custom types, the timestamp is the most significant field and is 8 bytes, so the total size of the struct must be divisible by eight. In contrast with the alignment bytes, these bytes will not create a malfunction of the software since they are located at the end.

Once it is proven that the types defined are correct, the different methods implemented are tested. First, the *LoraExp* and *AlohaProtocol* objects are created and configured by calling the *loadConfig* function. The values used in this case are not the ones of the actual case, they have been selected only for test proposes.

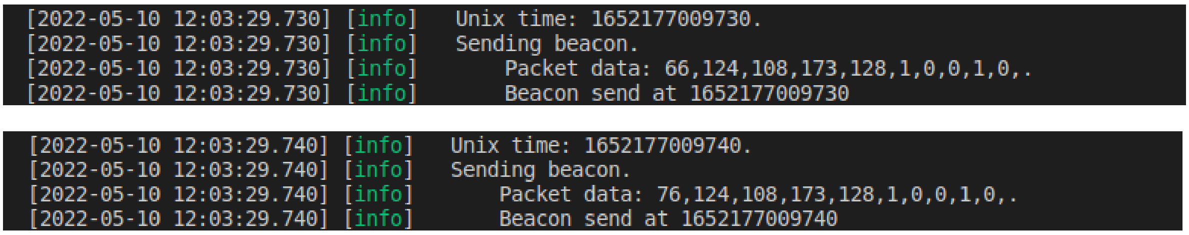


```
[2022-05-09 12:02:36.340] [info] Loading Protocol Configuration:
[2022-05-09 12:02:36.340] [info]   config verification:
[2022-05-09 12:02:36.340] [info]       Satellite ID: 1.
[2022-05-09 12:02:36.340] [info]       Max back off time: 1000000.
[2022-05-09 12:02:36.340] [info]       Wait time: 5.
[2022-05-09 12:02:36.340] [info]       Beacon repetition period: 10.
[2022-05-09 12:02:36.340] [info]   Current state set to: 0.
[2022-05-09 12:02:36.340] [info]   Call Back to LoraExp::processPacket created.
```

Figure 10. Configuration of the protocol parameters.

Then, the different states are tested. To do this, a simulation is conducted of the different cases the software will have to face in each of these states introducing different types of packets to the buffers manually. This simulation is performed by realizing several lops of the control method execution.

First, it is checked the *IDLE* state. However, no actions are performed when there are no packets to process at any of the buffers or queues. So, this initial test is used to prove the most basic task that this software has, which is to send the beacon periodically. This task is performed not only at the *IDLE* state but also at *RECEIVE* and *SEND*.



```
 [2022-05-10 12:03:29.730] [info]   Unix time: 1652177009730.
 [2022-05-10 12:03:29.730] [info]   Sending beacon.
 [2022-05-10 12:03:29.730] [info]       Packet data: 66,124,108,173,128,1,0,0,1,0,.
 [2022-05-10 12:03:29.730] [info]       Beacon send at 1652177009730
```

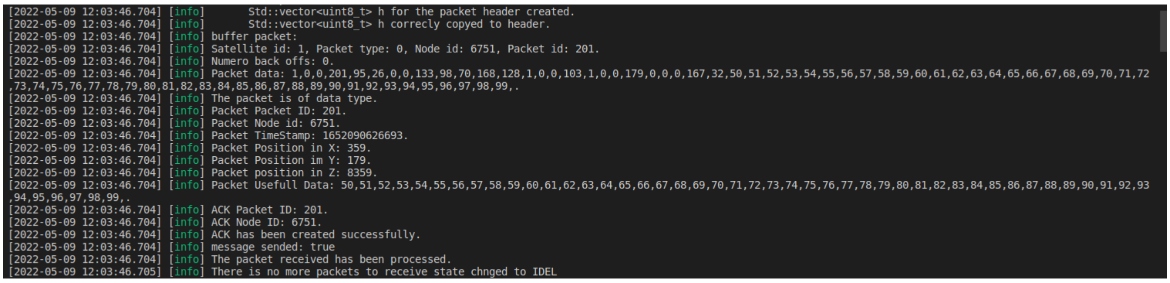
```
 [2022-05-10 12:03:29.740] [info]   Unix time: 1652177009740.
 [2022-05-10 12:03:29.740] [info]   Sending beacon.
 [2022-05-10 12:03:29.740] [info]       Packet data: 76,124,108,173,128,1,0,0,1,0,.
 [2022-05-10 12:03:29.740] [info]       Beacon send at 1652177009740
```

Figure 11. Periodic transmission of the beacon

The previous figures show that the beacon is correctly created with the timestamp and the satellite identifier, and it is sent 10 ms after the previous one has been sent.

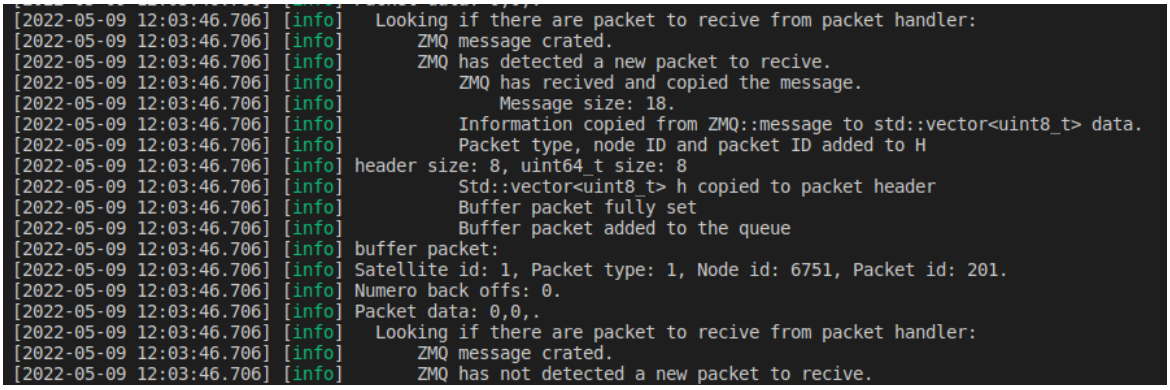
Then, the functionality in *RECEIVE* state is tested. To simulate this situation, two packets are added to the *buffer_rx*. Because these are simulating packets received from *LoraUtils*, which only send a vector of *uint8_t*, of the different fields that *Buffer_Packet* contains is only set the packet. Their header is still unknown, so this is randomly created.

When these packets are detected, the state is changed. Once in the receiving state, the packet's header is obtained and stored at the *Buffer_Packet*. If the packet received is of *DATA* type, by calling the callback, this packet is sent to *LoraExp* to be processed, and the ACK to confirm the correct reception is created with the correspondent information.



```
[2022-05-09 12:03:46.704] [info]      Std::vector<uint8_t> h for the packet header created.
[2022-05-09 12:03:46.704] [info]      Std::vector<uint8_t> h correcly copyed to header.
[2022-05-09 12:03:46.704] [info] buffer packet:
[2022-05-09 12:03:46.704] [info] Satellite id: 1, Packet type: 0, Node id: 6751, Packet id: 201.
[2022-05-09 12:03:46.704] [info] Numero back offs: 0.
[2022-05-09 12:03:46.704] [info] Packet data: 1,0,0,201,95,26,0,0,133,98,70,168,128,1,0,0,103,1,0,0,179,0,0,0,167,32,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72
,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,.
[2022-05-09 12:03:46.704] [info] The packet is of data type.
[2022-05-09 12:03:46.704] [info] Packet Packet ID: 201.
[2022-05-09 12:03:46.704] [info] Packet Node id: 6751.
[2022-05-09 12:03:46.704] [info] Packet TimeStamp: 1652090626693.
[2022-05-09 12:03:46.704] [info] Packet Position in X: 359.
[2022-05-09 12:03:46.704] [info] Packet Position im Y: 179.
[2022-05-09 12:03:46.704] [info] Packet position in Z: 8359.
[2022-05-09 12:03:46.704] [info] Packet Usefull Data: 50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93
,94,95,96,97,98,99,.
[2022-05-09 12:03:46.704] [info] ACK Packet ID: 201.
[2022-05-09 12:03:46.704] [info] ACK Node ID: 6751.
[2022-05-09 12:03:46.704] [info] ACK has been created successfully.
[2022-05-09 12:03:46.704] [info] message sended: true
[2022-05-09 12:03:46.704] [info] The packet received has been processed.
[2022-05-09 12:03:46.705] [info] There is no more packets to receive state chnged to IDEL
```

Figure 12. Received packet processing



```
[2022-05-09 12:03:46.706] [info]   Looking if there are packet to recive from packet handler:
[2022-05-09 12:03:46.706] [info]       ZMQ message crated.
[2022-05-09 12:03:46.706] [info]       ZMQ has detected a new packet to recive.
[2022-05-09 12:03:46.706] [info]           ZMQ has recived and copied the message.
[2022-05-09 12:03:46.706] [info]               Message size: 18.
[2022-05-09 12:03:46.706] [info]           Information copied from ZMQ::message to std::vector<uint8_t> data.
[2022-05-09 12:03:46.706] [info]           Packet type, node ID and packet ID added to H
[2022-05-09 12:03:46.706] [info] header size: 8, uint64_t size: 8
[2022-05-09 12:03:46.706] [info]           Std::vector<uint8_t> h copied to packet header
[2022-05-09 12:03:46.706] [info]           Buffer packet fully set
[2022-05-09 12:03:46.706] [info]           Buffer packet added to the queue
[2022-05-09 12:03:46.706] [info] buffer packet:
[2022-05-09 12:03:46.706] [info] Satellite id: 1, Packet type: 1, Node id: 6751, Packet id: 201.
[2022-05-09 12:03:46.706] [info] Numero back offs: 0.
[2022-05-09 12:03:46.706] [info] Packet data: 0,0,.
[2022-05-09 12:03:46.706] [info]   Looking if there are packet to recive from packet handler:
[2022-05-09 12:03:46.706] [info]       ZMQ message crated.
[2022-05-09 12:03:46.706] [info]       ZMQ has not detected a new packet to recive.
```

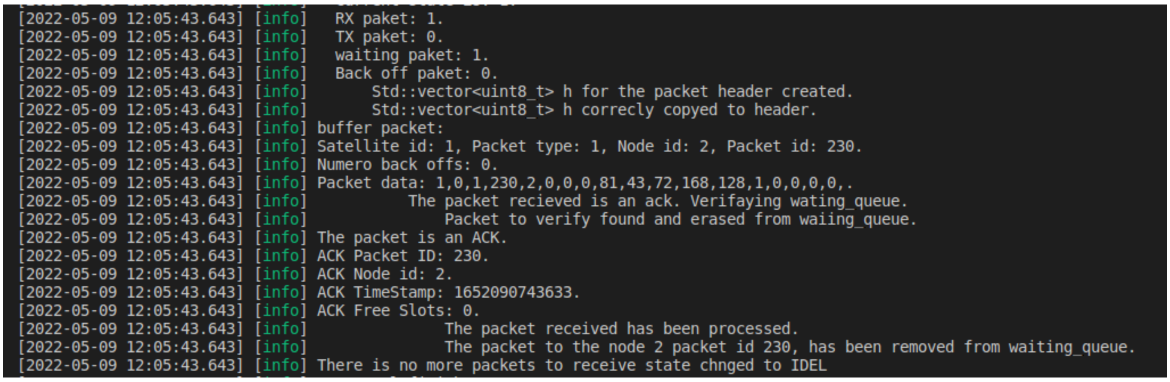
Figure 13. Retrieval of ACK from LoraExp by ZMQ sockets

Then, the ACK is retrieved at the subsequent executions using the ZMQ sockets. The array of bytes received from this one is processed to obtain the packet's header, and it is created a buffer packet which is added at the end of the *buffer_tx*.

At the previous captures, referent to the efficiency of the software, it can be seen that there is a certain delay between the transmission and retrieval of the ACK packets. This delay reaches a maximum of 3 milliseconds which could be relevant depending on the wait time that has been selected.

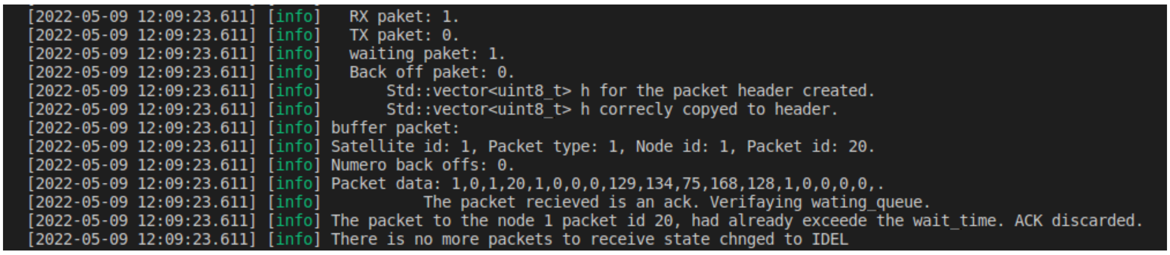
Another result that can be deduced is that, in addition to the correct processing of the packet, it can also be confirmed that the exchange of information between the different objects of *AlohaProtocol* and *LoraExp* is performed with no losses or alterations of the information stored.

If the packet added to the *buffer_rx* is an ACK, the waiting queue is verified. Suppose the packet with the node and packet identifier is found. In that case, the packet is sent to de callback and processed, and both packets are eliminated. If not, the ACK packet is discarded.



```
[2022-05-09 12:05:43.643] [info]   RX paket: 1.
[2022-05-09 12:05:43.643] [info]   TX paket: 0.
[2022-05-09 12:05:43.643] [info]   waiting paket: 1.
[2022-05-09 12:05:43.643] [info]   Back off paket: 0.
[2022-05-09 12:05:43.643] [info]       Std::vector<uint8_t> h for the packet header created.
[2022-05-09 12:05:43.643] [info]       Std::vector<uint8_t> h correcly copyed to header.
[2022-05-09 12:05:43.643] [info] buffer packet:
[2022-05-09 12:05:43.643] [info] Satellite id: 1, Packet type: 1, Node id: 2, Packet id: 230.
[2022-05-09 12:05:43.643] [info] Numero back offs: 0.
[2022-05-09 12:05:43.643] [info] Packet data: 1,0,1,230,2,0,0,0,81,43,72,168,128,1,0,0,0,0,.
[2022-05-09 12:05:43.643] [info]           The packet recieved is an ack. Verifaying wating_queue.
[2022-05-09 12:05:43.643] [info]               Packet to verify found and erased from waiing_queue.
[2022-05-09 12:05:43.643] [info] The packet is an ACK.
[2022-05-09 12:05:43.643] [info] ACK Packet ID: 230.
[2022-05-09 12:05:43.643] [info] ACK Node id: 2.
[2022-05-09 12:05:43.643] [info] ACK TimeStamp: 1652090743633.
[2022-05-09 12:05:43.643] [info] ACK Free Slots: 0.
[2022-05-09 12:05:43.643] [info]               The packet received has been processed.
[2022-05-09 12:05:43.643] [info]               The packet to the node 2 packet id 230, has been removed from waiting_queue.
[2022-05-09 12:05:43.643] [info] There is no more packets to receive state chnged to IDEL
```

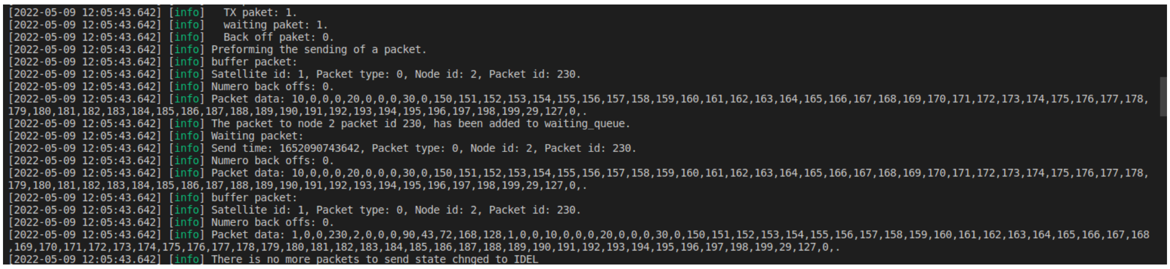
Figure 14. Confirmation of the waiting packet



```
[2022-05-09 12:09:23.611] [info]   RX paket: 1.
[2022-05-09 12:09:23.611] [info]   TX paket: 0.
[2022-05-09 12:09:23.611] [info]   waiting paket: 1.
[2022-05-09 12:09:23.611] [info]   Back off paket: 0.
[2022-05-09 12:09:23.611] [info]       Std::vector<uint8_t> h for the packet header created.
[2022-05-09 12:09:23.611] [info]       Std::vector<uint8_t> h correcly copyed to header.
[2022-05-09 12:09:23.611] [info] buffer packet:
[2022-05-09 12:09:23.611] [info] Satellite id: 1, Packet type: 1, Node id: 1, Packet id: 20.
[2022-05-09 12:09:23.611] [info] Numero back offs: 0.
[2022-05-09 12:09:23.611] [info] Packet data: 1,0,1,20,1,0,0,0,129,134,75,168,128,1,0,0,0,0,.
[2022-05-09 12:09:23.611] [info]           The packet recieved is an ack. Verifaying wating_queue.
[2022-05-09 12:09:23.611] [info] The packet to the node 1 packet id 20, had already exceede the wait_time. ACK discarded.
[2022-05-09 12:09:23.611] [info] There is no more packets to receive state chnged to IDEL
```

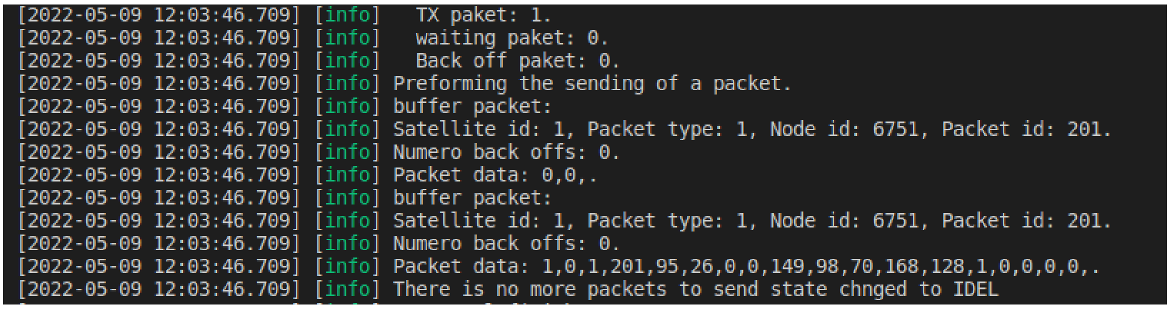
Figure 15. Discarded ACK

Then, the *SEND* state is tested. Some packets are added to the transmission buffer to check if they are correct. In contrast to the *RECEIVE* state, these buffer packets are added to the *buffer_tx* with the set header. Suppose the packet that has to be sent is of *DATA* type. In that case, before sharing it with *LoraUtils*, a waiting packet is created with the necessary information for retransmission if the communication fails and is added to the waiting queue. If it is an ACK, the packet is only sent to *LoraUtils* because there is no need to be added to the waiting queue since this type of packet is not retransmitted.



```
[2022-05-09 12:05:43.642] [info]   TX paket: 1.
[2022-05-09 12:05:43.642] [info]   waiting paket: 1.
[2022-05-09 12:05:43.642] [info]   Back off paket: 0.
[2022-05-09 12:05:43.642] [info] Preforming the sending of a packet.
[2022-05-09 12:05:43.642] [info] buffer packet:
[2022-05-09 12:05:43.642] [info] Satellite id: 1, Packet type: 0, Node id: 2, Packet id: 230.
[2022-05-09 12:05:43.642] [info] Numero back offs: 0.
[2022-05-09 12:05:43.642] [info] Packet data: 10,0,0,0,20,0,0,0,30,0,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,
179,180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,29,127,0,.
[2022-05-09 12:05:43.642] [info] The packet to node 2 packet id 230, has been added to waiting_queue.
[2022-05-09 12:05:43.642] [info] Waiting packet:
[2022-05-09 12:05:43.642] [info] Send time: 1652090743642, Packet type: 0, Node id: 2, Packet id: 230.
[2022-05-09 12:05:43.642] [info] Numero back offs: 0.
[2022-05-09 12:05:43.642] [info] Packet data: 10,0,0,0,20,0,0,0,30,0,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,
179,180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,29,127,0,.
[2022-05-09 12:05:43.642] [info] buffer packet:
[2022-05-09 12:05:43.642] [info] Satellite id: 1, Packet type: 0, Node id: 2, Packet id: 230.
[2022-05-09 12:05:43.642] [info] Numero back offs: 0.
[2022-05-09 12:05:43.642] [info] Packet data: 1,0,0,230,2,0,0,0,90,43,72,168,128,1,0,0,10,0,0,0,20,0,0,0,30,0,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,168
,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,29,127,0,.
[2022-05-09 12:05:43.642] [info] There is no more packets to send state chnged to IDEL
```

Figure 16. Data packet transmission

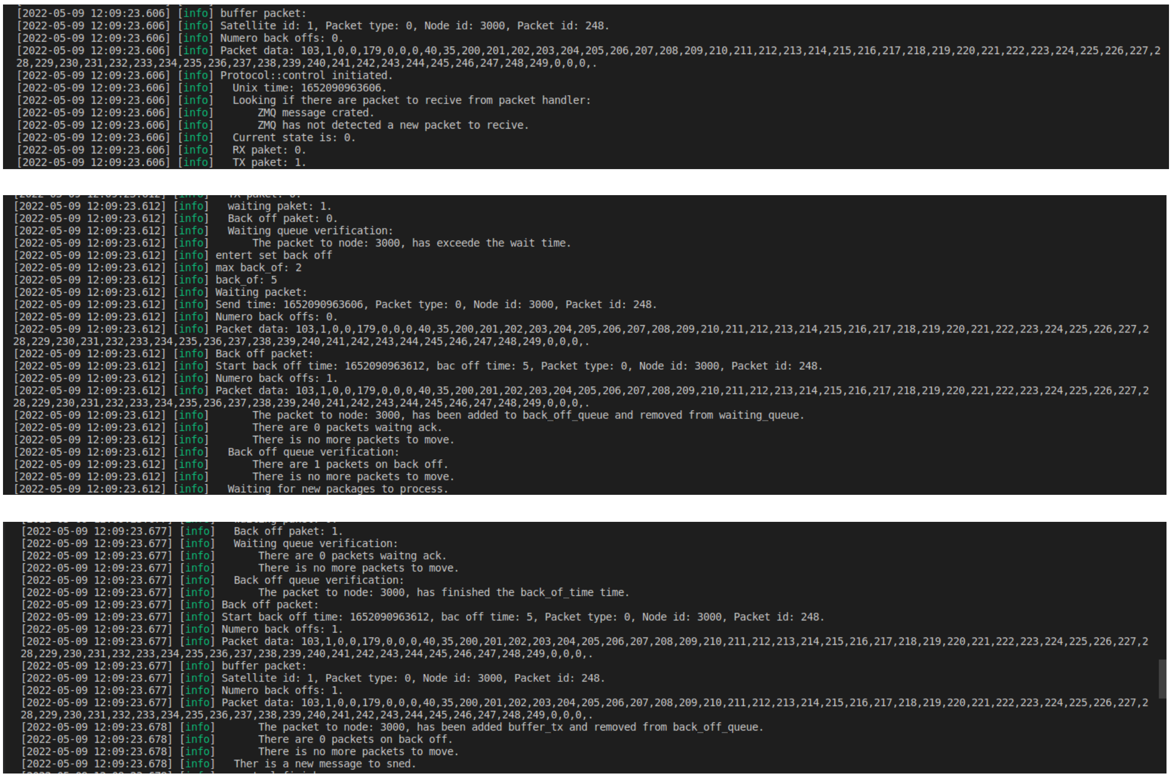


```
[2022-05-09 12:03:46.709] [info]   TX paket: 1.
[2022-05-09 12:03:46.709] [info]   waiting paket: 0.
[2022-05-09 12:03:46.709] [info]   Back off paket: 0.
[2022-05-09 12:03:46.709] [info] Preforming the sending of a packet.
[2022-05-09 12:03:46.709] [info] buffer packet:
[2022-05-09 12:03:46.709] [info] Satellite id: 1, Packet type: 1, Node id: 6751, Packet id: 201.
[2022-05-09 12:03:46.709] [info] Numero back offs: 0.
[2022-05-09 12:03:46.709] [info] Packet data: 0,0,.
[2022-05-09 12:03:46.709] [info] buffer packet:
[2022-05-09 12:03:46.709] [info] Satellite id: 1, Packet type: 1, Node id: 6751, Packet id: 201.
[2022-05-09 12:03:46.709] [info] Numero back offs: 0.
[2022-05-09 12:03:46.709] [info] Packet data: 1,0,1,201,95,26,0,0,149,98,70,168,128,1,0,0,0,0,.
[2022-05-09 12:03:46.709] [info] There is no more packets to send state chnged to IDEL
```

Figure 17. ACK transmission

In addition to the transmission process, at the previous captures of the log, it can also be seen that the addition of the header and the timestamp to the packet is done correctly.

Finally, the backoff process is tested. A new data packet is added to the transmission buffer to do this. Once it has been transmitted and correctly added to the waiting queue, it is waited until the end of the wait time to see if this is correctly detected, and the packet is added to the backoff queue.



```
[2022-05-09 12:09:23.606] [info] buffer packet:
[2022-05-09 12:09:23.606] [info] Satellite id: 1, Packet type: 0, Node id: 3000, Packet id: 248.
[2022-05-09 12:09:23.606] [info] Numero back offs: 0.
[2022-05-09 12:09:23.606] [info] Packet data: 103,1,0,0,179,0,0,0,40,35,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,2
28,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,0,0,0,.
[2022-05-09 12:09:23.606] [info] Protocol::control initiated.
[2022-05-09 12:09:23.606] [info]   Unix time: 1652090963606.
[2022-05-09 12:09:23.606] [info]   Looking if there are packet to recive from packet handler:
[2022-05-09 12:09:23.606] [info]       ZMQ message crated.
[2022-05-09 12:09:23.606] [info]       ZMQ has not detected a new packet to recive.
[2022-05-09 12:09:23.606] [info]   Current state is: 0.
[2022-05-09 12:09:23.606] [info]   RX paket: 0.
[2022-05-09 12:09:23.606] [info]   TX paket: 1.
```

```
[2022-05-09 12:09:23.612] [info]   waiting paket: 1.
[2022-05-09 12:09:23.612] [info]   Back off paket: 0.
[2022-05-09 12:09:23.612] [info]   Waiting queue verification:
[2022-05-09 12:09:23.612] [info]       The packet to node: 3000, has exceede the wait time.
[2022-05-09 12:09:23.612] [info] entert set back off
[2022-05-09 12:09:23.612] [info] max back_of: 2
[2022-05-09 12:09:23.612] [info] back_of: 5
[2022-05-09 12:09:23.612] [info] Waiting packet:
[2022-05-09 12:09:23.612] [info] Send time: 1652090963606, Packet type: 0, Node id: 3000, Packet id: 248.
[2022-05-09 12:09:23.612] [info] Numero back offs: 0.
[2022-05-09 12:09:23.612] [info] Packet data: 103,1,0,0,179,0,0,0,40,35,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,2
28,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,0,0,0,.
[2022-05-09 12:09:23.612] [info] Back off packet:
[2022-05-09 12:09:23.612] [info] Start back off time: 1652090963612, bac off time: 5, Packet type: 0, Node id: 3000, Packet id: 248.
[2022-05-09 12:09:23.612] [info] Numero back offs: 1.
[2022-05-09 12:09:23.612] [info] Packet data: 103,1,0,0,179,0,0,0,40,35,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,2
28,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,0,0,0,.
[2022-05-09 12:09:23.612] [info]       The packet to node: 3000, has been added to back_off_queue and removed from waiting_queue.
[2022-05-09 12:09:23.612] [info]       There are 0 packets waitng ack.
[2022-05-09 12:09:23.612] [info]       There is no more packets to move.
[2022-05-09 12:09:23.612] [info]   Back off queue verification:
[2022-05-09 12:09:23.612] [info]       There are 1 packets on back off.
[2022-05-09 12:09:23.612] [info]       There is no more packets to move.
[2022-05-09 12:09:23.612] [info]   Waiting for new packages to process.
```

```
[2022-05-09 12:09:23.677] [info]   Back off paket: 1.
[2022-05-09 12:09:23.677] [info]   Waiting queue verification:
[2022-05-09 12:09:23.677] [info]       There are 0 packets waitng ack.
[2022-05-09 12:09:23.677] [info]       There is no more packets to move.
[2022-05-09 12:09:23.677] [info]   Back off queue verification:
[2022-05-09 12:09:23.677] [info]       The packet to node: 3000, has finished the back_of_time time.
[2022-05-09 12:09:23.677] [info] Back off packet:
[2022-05-09 12:09:23.677] [info] Start back off time: 1652090963612, bac off time: 5, Packet type: 0, Node id: 3000, Packet id: 248.
[2022-05-09 12:09:23.677] [info] Numero back offs: 1.
[2022-05-09 12:09:23.677] [info] Packet data: 103,1,0,0,179,0,0,0,40,35,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,2
28,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,0,0,0,.
[2022-05-09 12:09:23.677] [info] buffer packet:
[2022-05-09 12:09:23.677] [info] Satellite id: 1, Packet type: 0, Node id: 3000, Packet id: 248.
[2022-05-09 12:09:23.677] [info] Numero back offs: 1.
[2022-05-09 12:09:23.677] [info] Packet data: 103,1,0,0,179,0,0,0,40,35,200,201,202,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226,227,2
28,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246,247,248,249,0,0,0,.
[2022-05-09 12:09:23.678] [info]       The packet to node: 3000, has been added buffer_tx and removed from back_off_queue.
[2022-05-09 12:09:23.678] [info]       There are 0 packets on back off.
[2022-05-09 12:09:23.678] [info]       There is no more packets to move.
[2022-05-09 12:09:23.678] [info]   Ther is a new message to sned.
```

Figure 18. Addition and removal of a packet from the backoff queue

In the previous figures, it can be seen that the exchange of packets between the waiting queue, the backoff queue and the transmission buffer is achieved with no problems. It can also be seen that the *setBackOffTime* correctly models the binary exponential formula that implements.

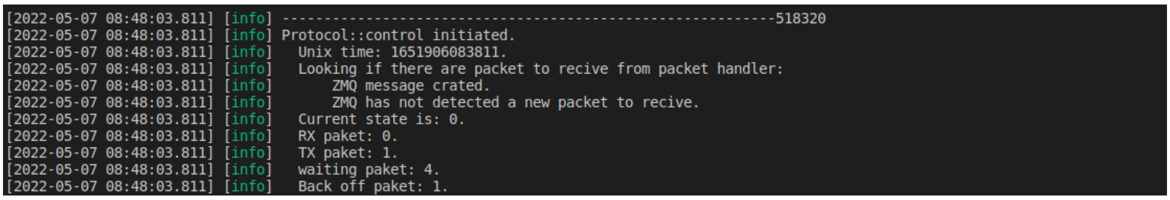
In what refers to efficiency, on average, the control method is executed three times in a millisecond, which means that at least one packet can be processed every millisecond if a change of state is performed.

4.1.2. Congestion test

Doe to the success rate of ALOHA, which is around 18%, the packets of failed communication could accumulate in the waiting or backoff queues provoking the memory fills quickly. This casuistic could be a problem considering the limited storage available on the CubeSat.

To prove that if the maximum backoff algorithm is correctly implemented and the case explained before is avoided, is performed a test to see how the program behaves along time. This test consists of a loop of thirty executions, of the *control* method where the software has to manage two received packets and send their correspondents ACK, two successful transmissions and one failed transmission.

This loop is performed for ten minutes with a waiting time of 5 milliseconds and a maximum of 5 backoff. After more than half a million executions, only four packets are in the waiting queue and one in the backoff queue.



```
[2022-05-07 08:48:03.811] [info] ----------------------------------------------------------518320
[2022-05-07 08:48:03.811] [info] Protocol::control initiated.
[2022-05-07 08:48:03.811] [info]   Unix time: 1651906083811.
[2022-05-07 08:48:03.811] [info]   Looking if there are packet to recive from packet handler:
[2022-05-07 08:48:03.811] [info]       ZMQ message crated.
[2022-05-07 08:48:03.811] [info]       ZMQ has not detected a new packet to recive.
[2022-05-07 08:48:03.811] [info]   Current state is: 0.
[2022-05-07 08:48:03.811] [info]   RX paket: 0.
[2022-05-07 08:48:03.811] [info]   TX paket: 1.
[2022-05-07 08:48:03.811] [info]   waiting paket: 4.
[2022-05-07 08:48:03.811] [info]   Back off paket: 1.
```

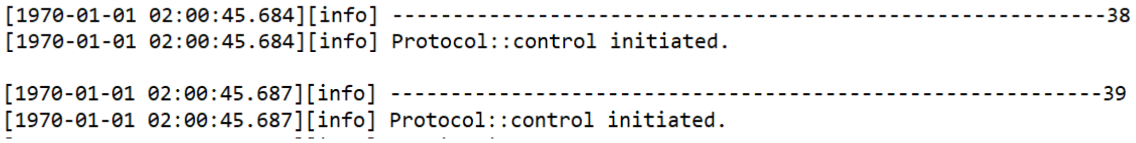
Figure 19. Packets congestion after 10 minutes of execution

4.2. Testing in Flight Hardware

The ADALM-Pluto is a module based on the AD9363 transceiver of Analog Devices, which can perform communications by a Software Defined Radio, radiofrequency and wireless. Its processor is a Single ARM cortex-A9 that works at 667 MHZ and 256 MB RAM. Therefore, the processing capacities of this hardware are very restrictive in front of an actual computer. For this reason, the ADALM-Pluto allows us to test the software in the hardware conditions that we will have on the satellite.

Because of the different architectures between the computer and the Pluto-SDR, it was needed to cross-compiled to the ARM architecture to test the software at the Pluto-SDR and perform hardware in the loop testing.

By seeing the output log, it is proven that the algorithm is correctly working, also the Pluto-SDR. In terms of efficiency, at the Pluto-SDR, the test executes control once in an average time of three milliseconds. That means that one packet needs six milliseconds to be processed.

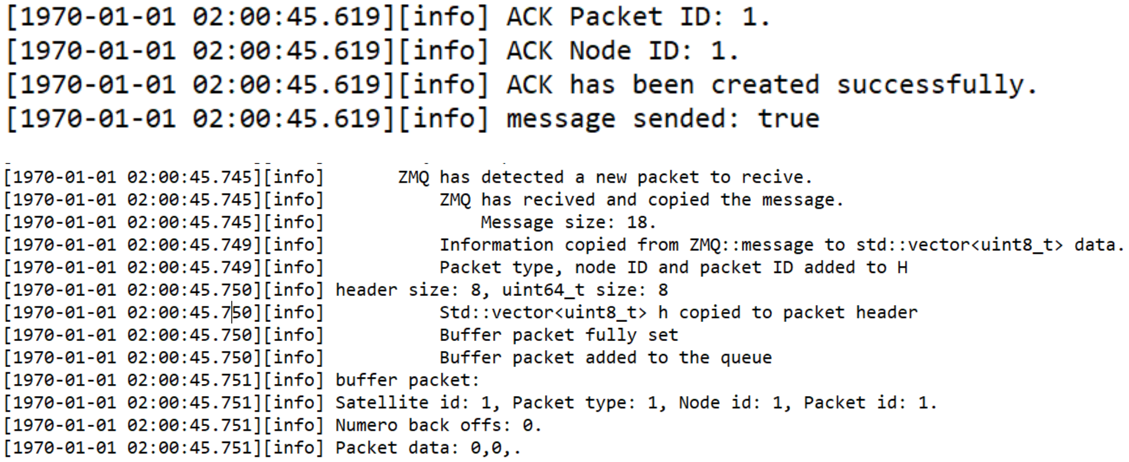


```
[1970-01-01 02:00:45.684][info] -----------------------------------------------------------38
[1970-01-01 02:00:45.684][info] Protocol::control initiated.

[1970-01-01 02:00:45.687][info] -----------------------------------------------------------39
[1970-01-01 02:00:45.687][info] Protocol::control initiated.
```

Figure 20. Duration of a control execution

In addition, it can be observed that the time it takes for ZMQ sockets to exchange the packets between LoraExp and Aloha protocol remains and, in some cases, gets worse. At my computer, the maximum delay is around 5ms; meanwhile, at the Pluto-SDR, it is 130 milliseconds which will provoke the wait time to be a bit longer.



```
[1970-01-01 02:00:45.619][info] ACK Packet ID: 1.
[1970-01-01 02:00:45.619][info] ACK Node ID: 1.
[1970-01-01 02:00:45.619][info] ACK has been created successfully.
[1970-01-01 02:00:45.619][info] message sended: true
```

```
[1970-01-01 02:00:45.745][info]       ZMQ has detected a new packet to recive.
[1970-01-01 02:00:45.745][info]           ZMQ has recived and copied the message.
[1970-01-01 02:00:45.745][info]               Message size: 18.
[1970-01-01 02:00:45.749][info]           Information copied from ZMQ::message to std::vector<uint8_t> data.
[1970-01-01 02:00:45.749][info]           Packet type, node ID and packet ID added to H
[1970-01-01 02:00:45.750][info] header size: 8, uint64_t size: 8
[1970-01-01 02:00:45.750][info]           Std::vector<uint8_t> h copied to packet header
[1970-01-01 02:00:45.750][info]           Buffer packet fully set
[1970-01-01 02:00:45.750][info]           Buffer packet added to the queue
[1970-01-01 02:00:45.751][info] buffer packet:
[1970-01-01 02:00:45.751][info] Satellite id: 1, Packet type: 1, Node id: 1, Packet id: 1.
[1970-01-01 02:00:45.751][info] Numero back offs: 0.
[1970-01-01 02:00:45.751][info] Packet data: 0,0,.
```

Figure 21. ZMQ sockets performance on Pluto-SDR

5. Budget

This project consists of software development, so the cost has been divided into hardware, software and developers' salary costs.

The hardware used in the development of this project is presented in the following table. To calculate the depreciation of these components have been selected a useful life of 5 years and a residual value of 10%.

Element	Number	Price	Depreciation
Computer	1	900 €	162 €
ADALM-PLUTO Active Learning Module	1	163,19 €	29.38 €
	TOTAL	**1063,19 €**	**191.38 €**

From the software side, the text editor used is open-source, so it has no cost.

So, the main cost of this project is deducted from the salary of the engineers who have contributed to it. The following table it is calculated the cost of this project taking into account dedication, on average, of 30 hours for the junior engineer, me, and 1 hour for the senior engineer each week for 34 weeks; so the total amount spent in euros is:

Position	Number	Dedication	Duration	Wage/hour	Subtotal
Junior Engineer	1	30 h/week	34 weeks	15.00 €/h	15,300.00 €
Senior Engineer	1	1 h/week	34weeks	30.00 €/h	1,020.00 €
				TOTAL	**16,320.00 €**

Therefore, the total cost to carry out the development of this project is **16,511.38** €.

6. Conclusions and future development:

The objective of this final degree thesis has been to develop software capable of emulating the algorithm of some selected MAC protocols that meets specific requirements due to the limitations of the hardware and conditions that will be used.

As commented in the results, functional implementation of the Pure ALOHA protocol has been accomplished. It has also completed the design CSMA/CA; however, it has been started to implement but could not be finished due to time constraints.

This project will continue in the future to reach the need of the RITA mission.

The first consideration for the future will be to continue implementing CSMA/CA. Once the implementation and testing are finished, and a functional result has been achieved, it will be preceded by the developing of the Slotted ALOHA protocol.

Having the three protocols finished, an alternative to the ZMQ sockets can be studied to reduce the delay in exchanging information between Protocol and LoraExp. It can also be studied if the performance of the software can be enhanced.

It will also be interesting to create a test to simulate the communication between some node and a satellite in which the channel was modelled as an idle channel that only introduces a delay in the communication since the issue of power is not essential at the link layer.

Finally, an end-to-end test, by putting together the entire software and uploading it to the hardware, should be performed to ensure the correct function in an actual situation. Of course, there is always the possibility of adding more Mac protocols.

Bibliography:

[1] *K. Mekki, E. Bajic, F. Chaxel and F. Meyer, "A comparative study of LPWAN technologies for large-scale IoT deployment", ICT Express, vol. 5, no. 1, pp. 1-7, Mar. 2019, [online] Available: http://www.sciencedirect.com/science/article/pii/S2405959517302953.*

[2] *L. Fernandez* et al., *"SDR-Based Lora Enabled On-Demand Remote Acquisition Experiment On-Board the Alainsat-1," 2021 IEEE International Geoscience and Remote Sensing Symposium IGARSS, 2021, pp. 8111-8114, doi: 10.1109/IGARSS47720.2021.9553020*.

[3] *A. Pérez* et al., *"RITA: Requirements and Preliminary Design of an L-Band Microwave Radiometer, Optical Imager, and RFI Detection Payload for a 3U CubeSat," IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*, *2020, pp. 5986-5989, doi: 10.1109/IGARSS39084.2020.9324458.*

[4] *FOSSA Systems. Available: https://fossa.systems/.*

[5] *Lucana Sapce. Available: https://lacuna.space/.*

[6] *Semtech: AN1200.22 LoRa™ Modulation Basics. Accessed: September 2021. [Online]. Available: https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R0000001OJu/xvKUc5w9yjG1q5Pb2IIkpolW54YYqGb.frOZ7HQBcRc. (Accessed:05/2022)*

[7] *L. Fernandez, J. A. Ruiz-De-Azua, A. Calveras and A. Camps, "Assessing LoRa for Satellite-to-Earth Communications Considering the Impact of Ionospheric Scintillation," in IEEE Access, vol. 8, pp. 165570-165582, 2020, doi: 10.1109/ACCESS.2020.3022433.*

[8] *Forouzan, B.A. Data communications and networking. 5th ed. New York: McGraw-Hill, 2013. ISBN 0071254420.*

[9] *Ferrer, T.; Céspedes, S.; Becerra, A. Review and Evaluation of MAC Protocols for Satellite IoT Systems Using Nanosatellites. Sensors 2019,* 19, *1947. https://doi.org/10.3390/s19081947.*

[10] *Fernandez, L.; Ruiz-de-Azua, J.A.; Calveras, A.; Camps, A. On-Demand Satellite Payload Execution Strategy for Natural Disasters Monitoring Using LoRa: Observation Requirements and Optimum Medium Access Layer Mechanisms.* Remote Sens. *2021,* 13, *4014. https://doi.org/10.3390/rs13194014.*

[11] *Spdlog: Fast C++ logging library. Available: https://github.com/gabime/spdlog.*

[12] *"ZMQ guide". Accessed: November 2021. [Online]. Available: https://zguide.zeromq.org/.*

[13] *"ZMQ API". Accessed: November 2021. [Online]. Available: http://api.zeromq.org/.*

7. Appendices:

7.1. Work Plan

7.1.1. Work packages

This project is organized into five subgroups of work.

- WP1: Research. It consists of searching and understanding all the necessary basic knowledge to carry out this project.
- WP2: Implementation. A previous work of design followed by the writing of the code in C++.
- WP3: Unit testing. Debug and perform a test of the different casuistic that the code will have to approach until a functional result is achieved.
- WP4: hardware on de loop testing. Compile the software to upload to the Pluto-SDR, check its correct operation and optimize.
- WP5: Documentation. Writing a series of reports to document the progress of this project and the results derived from it.

7.1.2. Gantt diagram

April 2022				May 2022			Task	Work Package
6	13	20	27	4	11	18		
							Read the documentation about RITA, the experiment and LoRa modulation	WP1
							Understand the code of RITA-LoRa already done	WP1
							Search and learn about pure ALOHA	WP1
							Protocol Interface design and implementation	WP2
							Pure ALOHA design	WP2
							Pure ALOHA implementation	WP2
							Pure ALOHA debugging	WP2
							Pure ALOHA unit testing	WP3
							Pure Aloha testing on Pluto-SDR	WP4
							Search and learn about CSMA/CA with RTS/CTS	WP1
🟩							CSMA/CA design	WP2
	🟩	🟩					CSMA/CA implementation	WP2
							Project Proposal and Work Plan	WP5
							Project Critical Review	WP5
			🟩	🟩	🟩	🟩	Final report writing	WP5

September 2021					October 2021				November 20221				December 2021					January 2022				February 2022				March 2022				
1	8	15	22	29	6	13	20	27	3	10	17	24	1	8	15	22	29	5	12	19	26	2	9	16	23	2	9	16	23	30
🟩	🟩																													
		🟩																												
			🟩	🟩																										
					🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩																		
							🟩	🟩	🟩	🟩	🟩	🟩																		
									🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩														
																	🟩	🟩	🟩	🟩	🟩									
																						🟩	🟩	🟩						
																									🟩	🟩				
																											🟩	🟩		
																													🟩	🟩
				🟩	🟩																									
											🟩	🟩																		

7.2. Code

7.2.1. Protocol.hpp

This is the header file of the interface *Protocol*. Here there are defined the different custom types of information used in the MAC layer and common methods are defined.

```
#ifndef PROTOCOL_HPP
#define PROTOCOL_HPP

#include "spdlog/spdlog.h"
#include <bits/stdint-uintn.h>
#include <bits/types/time_t.h>
#include <chrono>
#include <functional>
#include <stdio.h>
#include <zmq.h>
#include <zmq.hpp>

#include "LoraExp.hpp"

class Protocol
{
 public:
   typedef union {
       uint8_t raw[10];
       struct
       {
           uint64_t timestamp;
           uint16_t satellite_id;
           // uint16_t sync_slots;
           // uint16_t num_slots;
           // uint16_t free_slots;
           // uint16_t time_window;
           // uint8_t packet_size;
       };

   } Lora_Beacon;

   typedef union {
       uint8_t raw[8];
       struct
       {
           uint16_t satellite_id;
           uint8_t packet_type;
           uint8_t packet_id;
           uint32_t node_id;
       };

   } Protocol_Header;

   typedef struct
   {
       Protocol_Header header;
       uint n_backoffs;
       std::vector<uint8_t> packet;
   } Buffer_Packet;

   typedef struct
   {
       uint8_t packet_type;
       uint8_t packet_id;
       uint32_t node_id;
       time_t send_time;
       uint n_backoffs;
       std::vector<uint8_t> packet;

   } Waiting_Packet;

   typedef struct
   {
       uint8_t packet_type;
       uint8_t packet_id;
       uint32_t node_id;
       time_t start_back_off;
       time_t back_off_time;
       uint n_backoffs;
       std::vector<uint8_t> packet;

   } Back_Off_Packet;

   typedef struct
   {
       uint16_t satellite_id;
       time_t period;
       time_t wait_time;
       time_t max_back_off_time;
       int max_back_off;
       LoraExp *packet_handler;

   } Config_Params;

   enum State
   {
       IDLE,
       RECEIVE,
       SEND,
       STOP
   };

   enum Packet_Type
   {
       DATA_PACKET,
       ACK,
       RTS,
       CTS,
       INFO
   };

   Protocol();
```

```
   ~Protocol();
   void loadConfig(Config_Params params);
   void setRX(std::vector<uint8_t> p);
   void setTX(uint8_t p_t, uint32_t n_id, uint8_t p_id,
std::vector<uint8_t> p);
   time_t getTimeStamp();
   void sendBeacon(time_t time);
   time_t setBackOffTime(int n);
   void printBufferPacket(Buffer_Packet p);
   void printWaitingPacket(Waiting_Packet p);
   void printBackOffPacket(Back_Off_Packet p);
   virtual bool control();
   virtual bool receive();

 protected:
   std::vector<Buffer_Packet> buffer_rx;
   std::vector<Buffer_Packet> buffer_tx;
   Config_Params config;
   zmq::context_t ctx;
   zmq::socket_t pull;
   std::function<void(LoraExp *, std::vector<uint8_t>, uint8_t)>
cb_t;
   State current_state;
   time_t last_beacon;
   Lora_Beacon beacon;
};

#endif
```

7.2.2. Protocol.cpp

This is the code file of the interface where the different functions previously defined are implemented.

```
#include "Protocol.hpp"
#include <bits/stdint-uintn.h>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <spdlog/spdlog.h>
#include <zmq.h>

Protocol::Protocol() : ctx(1), pull(ctx, ZMQ_PULL)
{
   spdlog::info("Creating Protocol:");
   pull.bind("tcp://*:5557");
   spdlog::info("  Binded to: tcp://*:5557");
}

Protocol::~Protocol()
{
}

void Protocol::loadConfig(Config_Params params)
{
   spdlog::info("Loading Protocol Configuration:");
   beacon.satellite_id = params.satellite_id;
   config = params;
   spdlog::info("  config verification:");
   spdlog::info("      Satellite ID: {}.", config.satellite_id);
   spdlog::info("      Max back off time: {}.",
config.max_back_off_time);
   spdlog::info("      Wait time: {}.", config.wait_time);
   spdlog::info("      Beacon repetition period: {}.",
config.period);
   last_beacon = 0;
   current_state = Protocol::State::IDLE;
   spdlog::info("  Current state set to: {}.", current_state);
   cb_t = &LoraExp::processPacket;
   spdlog::info("  Call Back to LoraExp::processPacket created.");
}

void Protocol::setRX(std::vector<uint8_t> p)
{
   Buffer_Packet temp;

   temp.n_backoffs = 0;
   temp.packet = p;
   buffer_rx.push_back(temp);
   printBufferPacket(buffer_rx.back());
}

void Protocol::setTX(uint8_t pt, uint32_t n_id, uint8_t p_id,
std::vector<uint8_t> p)
{
   Buffer_Packet temp;
   temp.header.satellite_id = config.satellite_id;
   temp.header.packet_type = pt;
   temp.header.node_id = n_id;
   temp.header.packet_id = p_id;
   temp.n_backoffs = 0;
   temp.packet = p;
   buffer_tx.push_back(temp);
   printBufferPacket(buffer_tx.back());
}

time_t Protocol::getTimeStamp()
{
   std::chrono::system_clock::time_point tp =
std::chrono::system_clock::now();
   time_t unix_timestamp_ms =
std::chrono::duration_cast<std::chrono::milliseconds>(tp.time_since_e
poch()).count();
   return unix_timestamp_ms;
}

void Protocol::sendBeacon(time_t time)
{
   beacon.timestamp = time;
   std::stringstream data;
   std::copy(std::begin(beacon.raw), std::end(beacon.raw),
std::ostream_iterator<int>(data, ","));
   spdlog::info("      Packet data: {}.", data.str().c_str());
   last_beacon = time;
   spdlog::info("      Beacon send at {}", time);
}

time_t Protocol::setBackOffTime(int n)
{
   spdlog::info("entert set back off");
   int a = std::pow(2, n);
   spdlog::info("max back_of: {}", a);
   int temp = (rand() % a) * config.wait_time;
   spdlog::info("back_of: {}", temp);
   return temp;
}

void Protocol::printBufferPacket(Buffer_Packet p)
{
   spdlog::info("buffer packet:");
   spdlog::info("Satellite id: {}, Packet type: {}, Node id: {},
Packet id: {}.", p.header.satellite_id,
                p.header.packet_type, p.header.node_id,
p.header.packet_id);
   spdlog::info("Numero back offs: {}.", p.n_backoffs);
   std::stringstream data;
   std::copy(p.packet.begin(), p.packet.end(),
std::ostream_iterator<int>(data, ","));
   spdlog::info("Packet data: {}.", data.str().c_str());
}

void Protocol::printWaitingPacket(Waiting_Packet p)
{
   spdlog::info("Waiting packet:");
   spdlog::info("Send time: {}, Packet type: {}, Node id: {}, Packet
id: {}.", p.send_time, p.packet_type, p.node_id,
                p.packet_id);
   spdlog::info("Numero back offs: {}.", p.n_backoffs);
   std::stringstream data;
   std::copy(p.packet.begin(), p.packet.end(),
std::ostream_iterator<int>(data, ","));
   spdlog::info("Packet data: {}.", data.str().c_str());
}

void Protocol::printBackOffPacket(Back_Off_Packet p)
{
   spdlog::info("Back off packet:");
   spdlog::info("Start back off time: {}, bac off time: {}, Packet
type: {}, Node id: {}, Packet id: {}.",
                p.start_back_off, p.back_off_time, p.packet_type,
p.node_id, p.packet_id);
   spdlog::info("Numero back offs: {}.", p.n_backoffs);
   std::stringstream data;
   std::copy(p.packet.begin(), p.packet.end(),
std::ostream_iterator<int>(data, ","));
   spdlog::info("Packet data: {}.", data.str().c_str());
```

```
}

bool Protocol::control()
{
   spdlog::info("  enter switch");
   return true;
}

bool Protocol::receive()
{
   return true;
}
```

7.2.3. AlohaProtocol.hpp

This is the header file of the class that extends the interface to implement the ALOHA protocol.

```
#ifndef ALOHAPROTOCOL_HPP
#define ALOHAPROTOCOL_HPP

#include "LoraExp.hpp"
#include "Protocol.hpp"
#include <bits/types/time_t.h>

class AlohaProtocol : public Protocol
{
 public:
   bool control() override;
   bool receive() override;
   AlohaProtocol() : Protocol(){};
   ~AlohaProtocol(){};

 private:
   std::vector<Waiting_Packet> waiting_queue;
   std::vector<Back_Off_Packet> back_off_queue;
};

#endif
```

7.2.4. AlohaProtocol.cpp

This is the code file of the class that extends the interface to implement the ALOHA protocol. Here there is the redefinition of the methods of *control* and *receive*.

```
#include "AlohaProtocol.hpp"
#include <zmq.h>

bool AlohaProtocol::control()
{
   spdlog::info("Protocol::control initiated.");
   time_t unix_timestamp_ms = getTimeStamp();
   spdlog::info("  Unix time: {}.", unix_timestamp_ms);
   if (config.period =< unix_timestamp_ms - last_beacon)
```

```
    {
        spdlog::info("  Sending beacon.");
        sendBeacon(unix_timestamp_ms);
    }

    bool more = true;
    while (more)
    {
        more = receive();
    }
    spdlog::info("  Current state is: {}.", current_state);
    spdlog::info("  RX paket: {}.", buffer_rx.size());
    spdlog::info("  TX paket: {}.", buffer_tx.size());
    spdlog::info("  waiting paket: {}.", waiting_queue.size());
    spdlog::info("  Back off paket: {}.", back_off_queue.size());
    switch (current_state)
    {
    case Protocol::State::IDLE: {

        bool a = true;
        spdlog::info("  Waiting queue verification:");
        while (!waiting_queue.empty() && a)
        {
            if (unix_timestamp_ms - waiting_queue[0].send_time >
config.wait_time)
            {
                spdlog::info("      The packet to node: {}, has
exceede the wait time.", waiting_queue[0].node_id);
                if (waiting_queue[0].n_backoffs < config.max_back_off)
                {
                    Back_Off_Packet temp;
                    temp.packet_type = waiting_queue[0].packet_type;
                    temp.node_id = waiting_queue[0].node_id;
                    temp.packet_id = waiting_queue[0].packet_id;
                    temp.start_back_off = unix_timestamp_ms;
                    temp.n_backoffs = waiting_queue[0].n_backoffs + 1;
                    temp.back_off_time =
setBackOffTime(temp.n_backoffs);
                    temp.packet = waiting_queue[0].packet;
                    back_off_queue.push_back(temp);
                    printWaitingPacket(waiting_queue.front());
                    printBackOffPacket(back_off_queue.back());
                    spdlog::info("      The packet to node: {}, has
been added to back_off_queue and removed from "
                                 "waiting_queue.",
                                 temp.node_id);
                }
                else
                {
                    spdlog::info("      The packet to node: {}, has
exceede the max attemps.",
                                 waiting_queue[0].node_id);
                }
                waiting_queue.erase(waiting_queue.begin());
            }
            else
            {
                a = false;
            }
        }
        spdlog::info("      There are {} packets waitng ack.",
waiting_queue.size());
        spdlog::info("      There is no more packets to move.");
        int idx = 0;
        spdlog::info("  Back off queue verification:");
        while (idx < back_off_queue.size() && !back_off_queue.empty())
        {
            if (unix_timestamp_ms - back_off_queue[idx].start_back_off
> back_off_queue[idx].back_off_time)
            {
                spdlog::info("      The packet to node: {}, has
finished the back_of_time time.",
                             back_off_queue[idx].node_id);
                Buffer_Packet temp;
                temp.header.satellite_id = config.satellite_id;
                temp.header.packet_type =
back_off_queue[idx].packet_type;
                temp.header.node_id = back_off_queue[idx].node_id;
                temp.header.packet_id = back_off_queue[idx].packet_id;
                temp.n_backoffs = back_off_queue[idx].n_backoffs;
                temp.packet = back_off_queue[idx].packet;
                buffer_tx.push_back(temp);
                printBackOffPacket(back_off_queue[idx]);
                printBufferPacket(buffer_tx.back());
                back_off_queue.erase(back_off_queue.begin() + idx);
                spdlog::info("      The packet to node: {}, has been
added buffer_tx and removed from back_off_queue.",
                             temp.header.node_id);
                // break;
            }
            else
            {
                idx++;
            }
        }
        spdlog::info("      There are {} packets on back off.",
back_off_queue.size());
        spdlog::info("      There is no more packets to move.");
        if (!buffer_rx.empty())
        {
            current_state = Protocol::State::RECEIVE;
            spdlog::info("  New packet has been recived.");
            break;
        }
        else if (!buffer_tx.empty())
        {
            current_state = Protocol::State::SEND;
            spdlog::info("  Ther is a new message to sned.");
            break;
        }
        else
        {
            spdlog::info("  Waiting for new packages to process.");
```

```
       }

       break;
   }

   case Protocol::State::RECEIVE: {
       std::vector<uint8_t> h(buffer_rx[0].packet.begin(),
buffer_rx[0].packet.begin() + sizeof(Protocol_Header));
       spdlog::info("      Std::vector<uint8_t> h for the packet
header created.");
       std::copy(h.begin(), h.end(), buffer_rx[0].header.raw);
       spdlog::info("      Std::vector<uint8_t> h correcly copyed to
header.");
       printBufferPacket(buffer_rx.front());
       if (buffer_rx[0].header.satellite_id == config.satellite_id)
       {
           if (buffer_rx[0].header.packet_type ==
Protocol::Packet_Type::ACK)
           {
               spdlog::info("          The packet recieved is an ack.
Verifaying wating_queue.");
               bool a = true;
               for (int i = 0; i < waiting_queue.size(); i++)
               {
                   if (waiting_queue[i].node_id ==
buffer_rx[0].header.node_id &&
                       waiting_queue[i].packet_id ==
buffer_rx[0].header.packet_id)
                   {

                       waiting_queue.erase(waiting_queue.begin() +
i);
                       spdlog::info("              Packet to verify
found and erased from waiing_queue.");
                       cb_t(config.packet_handler,
buffer_rx[0].packet, buffer_rx[0].header.packet_type);
                       spdlog::info("              The packet
received has been processed.");
                       spdlog::info("              The packet to the
node {} packet id {}, has been removed from "
                                    "waiting_queue.",
                                    buffer_rx[0].header.node_id,
buffer_rx[0].header.packet_id);
                       a = false;
                       break;
                   }
               }
               if (a)
               {
                   spdlog::info(
                       "The packet to the node {} packet id {}, had
already exceede the wait_time. ACK discarded.",
                       buffer_rx[0].header.node_id,
buffer_rx[0].header.packet_id);
               }
           }
           else
```

```
            {
                cb_t(config.packet_handler, buffer_rx[0].packet,
buffer_rx[0].header.packet_type);
                spdlog::info("The packet received has been
processed.");
            }
        }

        buffer_rx.erase(buffer_rx.begin());
        if (buffer_rx.empty())
        {
            current_state = Protocol::State::IDLE;
            spdlog::info("There is no more packets to receive state
chnged to IDEL");
        }
        else
        {
            current_state = Protocol::State::RECEIVE;
            spdlog::info("There is {} packets to receive. State remain
RECEIVE", buffer_rx.size());
        }
        // current_state = Protocol::State::IDLE;
        break;
    }

    case Protocol::State::SEND: {
        spdlog::info("Preforming the sending of a packet.");
        printBufferPacket(buffer_tx.front());
        std::vector<uint8_t> h(std::begin(buffer_tx[0].header.raw),
std::end(buffer_tx[0].header.raw));
        if (buffer_tx[0].header.packet_type ==
Protocol::Packet_Type::DATA_PACKET)
        {
            Waiting_Packet temp;
            temp.packet_type = buffer_tx[0].header.packet_type;
            temp.node_id = buffer_tx[0].header.node_id;
            temp.packet_id = buffer_tx[0].header.packet_id;
            temp.send_time = unix_timestamp_ms;
            temp.n_backoffs = buffer_tx[0].n_backoffs;
            temp.packet = buffer_tx[0].packet;
            waiting_queue.push_back(temp);
            spdlog::info("The packet to node {} packet id {}, has been
added to waiting_queue.", temp.node_id,
                        temp.packet_id);
            printWaitingPacket(waiting_queue.back());
        }
        uint8_t temp[8];
        std::memcpy(temp, &unix_timestamp_ms,
sizeof(unix_timestamp_ms));
        buffer_tx[0].packet.insert(buffer_tx[0].packet.begin(),
std::begin(temp), std::end(temp));
        buffer_tx[0].packet.insert(buffer_tx[0].packet.begin(),
h.begin(), h.end());
        printBufferPacket(buffer_tx.front());
        buffer_tx.erase(buffer_tx.begin());
        if (buffer_tx.empty())
        {
            current_state = Protocol::State::IDLE;
            spdlog::info("There is no more packets to send state
chnged to IDEL");
        }
        else
        {
            current_state = Protocol::State::SEND;
            spdlog::info("There is {} packets to send. State remain
SEND", buffer_tx.size());
        }
        // current_state = Protocol::State::IDLE;
        break;
    }

    case Protocol::State::STOP: {

        break;
    }
    }
    spdlog::info("  control finish");
    return true;
}

bool AlohaProtocol::receive()
{
    spdlog::info("  Looking if there are packet to recive from packet
handler:");
    zmq::message_t message;
    spdlog::info("      ZMQ message crated.");
    if (pull.recv(&message, ZMQ_NOBLOCK))
    {
        spdlog::info("      ZMQ has detected a new packet to
recive.");
        spdlog::info("          ZMQ has recived and copied the
message.");
        std::vector<uint8_t> temp(message.size());
        spdlog::info("              Message size: {}.",
message.size());
        memcpy(temp.data(), message.data(), message.size());
        spdlog::info("          Information copied from ZMQ::message
to std::vector<uint8_t> data.");
        Buffer_Packet p;
        std::vector<uint8_t> h(temp.begin(), temp.begin() +
sizeof(Protocol_Header));
        spdlog::info("          Packet type, node ID and packet ID
added to H");
        spdlog::info("header size: {}, uint64_t size: {}",
sizeof(Protocol_Header), sizeof(uint64_t));
        temp.erase(temp.begin(), temp.begin() +
sizeof(Protocol_Header) + sizeof(uint64_t));
        std::copy(h.begin(), h.end(), p.header.raw);
        spdlog::info("          Std::vector<uint8_t> h copied to
packet header");
        p.n_backoffs = 0;
        p.packet = temp;
        spdlog::info("          Buffer packet fully set");
        buffer_tx.push_back(p);
```

```
        spdlog::info("          Buffer packet added to the queue");
        printBufferPacket(buffer_tx.back());
        return true;
    }
    else
    {
        spdlog::info("      ZMQ has not detected a new packet to 
recive.");
        return false;
    }
}
```

7.2.5. LoraExp.hpp

This is the header file of the class that implements the actions of the application layer. Here there are defined the different packet types used in the three MAC protocols, the method to write the log, and the ZMQ socket.

```
#ifndef LORAEXP_HPP
#define LORAEXP_HPP

#include "spdlog/spdlog.h"
#include <bits/stdint-uintn.h>
#include <bits/types/time_t.h>
#include <chrono>
#include <zmq.h>
#include <zmq.hpp>

#define MAX_SENSOR_DATA = 50;

class LoraExp
{
 public:
    typedef union {
        uint8_t raw[76];
        struct
        {
            uint16_t satellite_id;
            uint8_t packet_type;
            uint8_t packet_id;
            uint32_t node_id;
            uint64_t timestamp;
            uint32_t pos_x;
            uint32_t pos_y;
            uint16_t pos_z;
            uint8_t sensor_data[50];
        };
    } Lora_Packet;

    typedef union {
        uint8_t raw[18];
        struct
        {
            uint16_t satellite_id;
            uint8_t packet_type;
            uint8_t packet_id;
            uint32_t node_id;
```

```
           uint64_t timestamp;
           uint16_t free_slots;
       };
   } Lora_Ack;

   typedef union {
       uint8_t raw[15];
       struct
       {
           uint16_t satellite_id;
           uint8_t packet_type;
           uint8_t packet_id;
           uint32_t node_id;
           uint64_t timestamp;
           uint16_t duration;
       };
   } Request_to_Send;

   typedef union {
       uint8_t raw[15];
       struct
       {
           uint16_t satellite_id;
           uint8_t packet_type;
           uint8_t packet_id;
           uint32_t node_id;
           uint64_t timestamp;
           uint16_t duration;
       };
   } Clear_to_Send;

   enum Packet_Type
   {
       DATA_PACKET,
       ACK,
       RTS,
       CTS,
   };

   LoraExp();
   ~LoraExp();
   void processPacket(std::vector<uint8_t> packet, uint8_t
packet_type);

 private:
   std::vector<uint8_t> buffer;
   zmq::context_t ctx;
   zmq::socket_t push;
   Lora_Packet packet;
   Lora_Ack ack;
   Request_to_Send rts;
   Clear_to_Send cts;

   void serialize(uint8_t packet_type, std::vector<uint8_t> packet);
   void createAck();
   void send();
```

```
};

#endif
```

7.2.6. LoraExp.cpp

This is the code file where it is implemented the methods defined at LoraExp.hpp.

```
#include "LoraExp.hpp"
#include <bits/stdint-uintn.h>
#include <spdlog/spdlog.h>
#include <zmq.hpp>

LoraExp::LoraExp() : ctx(1), push(ctx, ZMQ_PUSH)
{
   spdlog::info("Creating packet Handler:");
   push.connect("tcp://localhost:5557");
   spdlog::info("  Connected to: tcp://localhost:5557");
}

LoraExp::~LoraExp()
{
}

void LoraExp::processPacket(std::vector<uint8_t> packet, uint8_t
packet_type)
{

   serialize(packet_type, packet);
   if (packet_type == LoraExp::Packet_Type::DATA_PACKET)
   {
       createAck();
   }
}

void LoraExp::serialize(uint8_t packet_type, std::vector<uint8_t> p)
{
   switch (packet_type)
   {
   case LoraExp::Packet_Type::DATA_PACKET: {
       spdlog::info("The packet is of data type.");
       std::copy(p.begin(), p.end(), packet.raw);
       spdlog::info("Packet Packet ID: {}.", packet.packet_id);
       spdlog::info("Packet Node id: {}.", packet.node_id);
       spdlog::info("Packet TimeStamp: {}.", packet.timestamp);
       spdlog::info("Packet Position in X: {}.", packet.pos_x);
       spdlog::info("Packet Position im Y: {}.", packet.pos_y);
       spdlog::info("Packet position in Z: {}.", packet.pos_z);
       std::stringstream data;
       std::copy(std::begin(packet.sensor_data),
std::end(packet.sensor_data), std::ostream_iterator<int>(data, ","));
       spdlog::info("Packet Usefull Data: {}.", data.str().c_str());
       break;
   }
   case LoraExp::Packet_Type::ACK: {
       spdlog::info("The packet is an ACK.");
       std::copy(p.begin(), p.end(), ack.raw);
       spdlog::info("ACK Packet ID: {}.", ack.packet_id);
       spdlog::info("ACK Node id: {}.", ack.node_id);
       spdlog::info("ACK TimeStamp: {}.", ack.timestamp);
       spdlog::info("ACK Free Slots: {}.", ack.free_slots);
       break;
   }
   case LoraExp::Packet_Type::RTS: {
       spdlog::info("The packet is a RTS.");
       std::copy(p.begin(), p.end(), rts.raw);
       spdlog::info("RTS Node id: {}.", rts.node_id);
       spdlog::info("RTS TimeStamp: {}.", rts.timestamp);
       spdlog::info("RTS Packet Duradtion: {}.", rts.duration);
       break;
   }
   case LoraExp::Packet_Type::CTS: {
       spdlog::info("The packet is an CTS.");
       std::copy(p.begin(), p.end(), cts.raw);
       spdlog::info("CTS Node id: {}.", cts.node_id);
       spdlog::info("CTS TimeStamp: {}.", cts.timestamp);
       spdlog::info("CTS Packet Duradtion: {}.", cts.duration);
       break;
   }
   }
}

void LoraExp::createAck()
{
   ack.satellite_id = packet.satellite_id;
   ack.packet_type = LoraExp::Packet_Type::ACK;
   ack.packet_id = packet.packet_id;
   spdlog::info("ACK Packet ID: {}.", ack.packet_id);
   ack.node_id = packet.node_id;
   spdlog::info("ACK Node ID: {}.", ack.node_id);
   ack.timestamp = 0;
   ack.free_slots = 0;
   spdlog::info("ACK has been created successfully.");
   std::vector<uint8_t> temp(std::begin(ack.raw), std::end(ack.raw));
   buffer = temp;
   send();
}

void LoraExp::send()
{
   zmq::message_t message(buffer.size());
   memcpy(message.data(), buffer.data(), buffer.size());
   bool a = push.send(message);
   spdlog::info("message sended: {}", a);
}
```

7.2.7. Main

This is a code file where the different test has been implemented. It is verified the correct creation of the data structs and methods defined in the above files. It has been also implemented the simulation of some cases the protocol will ha to face.

```
#include <bits/stdint-uintn.h>
#include <chrono>
#include <iostream>
#include <iterator>
#include <new>
#include <stdio.h>

#include "spdlog/logger.h"
#include "spdlog/sinks/basic_file_sink.h"
#include "spdlog/sinks/stdout_color_sinks.h"
#include "spdlog/spdlog.h"

#include "AlohaProtocol.hpp"
#include "LoraExp.hpp"
#include "Protocol.hpp"

void setupLogging();
time_t getTimeStamp();
void printVector(std::vector<uint8_t> a, std::string name);
void printArray(uint8_t a[], uint8_t size, std::string name);

int main(int argc, char *argv[])
{
   setupLogging();
   spdlog::info("Lora prove.");

   spdlog::info("Welcome to the LoRa transceiver!");

   spdlog::info("Initialising Packet Handler");
   LoraExp p_h;

   spdlog::info("Parsing configuration");
   Protocol::Config_Params config;
   config.satellite_id = 1;
   config.period = 10;
   config.wait_time = 5;
   config.max_back_off_time = 1000000;
   config.max_back_off = 5;
   config.packet_handler = &p_h;

   spdlog::info("Initialising Protocol");
   AlohaProtocol aloha;
   aloha.loadConfig(config);
   typedef union {
       uint8_t raw[76]; // 80
       struct
       {

           uint16_t satellite_id;
           uint8_t packet_type;
```

```
        uint8_t packet_id;
        uint32_t node_id;
        uint64_t timestamp;
        uint32_t pos_x;
        uint32_t pos_y;
        uint16_t pos_z;
        uint8_t sensor_data[50];
    };
} L_P_A;
spdlog::info("size aligment rx packet: {}", sizeof(L_P_A));

typedef union {
    uint8_t raw[76]; // 80
    struct
    {
        uint16_t satellite_id;
        uint8_t packet_type;
        uint8_t packet_id;
        uint32_t node_id;
        uint64_t timestamp;
        uint32_t pos_x;
        uint32_t pos_y;
        uint16_t pos_z;
        uint8_t sensor_data[50];
    };
} L_P;
spdlog::info("size no aligment rx packet: {}", sizeof(L_P));

typedef union {
    uint8_t raw[63]; // 64
    struct
    {
        uint32_t pos_x;
        uint32_t pos_y;
        uint16_t pos_z;
        uint8_t sensor_data[50];
    };
} L_P2;
spdlog::info("size no aligment tx packet: {}", sizeof(L_P2));

typedef union {
    uint8_t raw[18]; // 24
    struct
    {
        uint16_t satellite_id;
        uint8_t packet_type;
        uint8_t packet_id;
        uint32_t node_id;
        uint64_t timestamp;
        uint16_t free_slots;
    };
} L_A;
spdlog::info("size no aligment ack: {}", sizeof(L_A));

L_P_A ar_pk1;
ar_pk1.satellite_id = 1;
```

```
    ar_pk1.packet_type = 0;
    ar_pk1.node_id = 1;
    ar_pk1.timestamp = getTimeStamp();
    ar_pk1.packet_id = 1;
    ar_pk1.pos_x = 70;
    ar_pk1.pos_y = 40;
    ar_pk1.pos_z = 20;
    int e = 0;
    for (int i = 0; i < 50; i++)
    {
        ar_pk1.sensor_data[i] = e;
        e++;
    }
    std::vector<uint8_t> z(std::begin(ar_pk1.raw),
std::end(ar_pk1.raw));
    printArray(ar_pk1.raw, 76, "ar_pk1 array");
    printVector(z, "ar_pk1 vector");

    L_P r_pk1;
    r_pk1.satellite_id = 1;
    r_pk1.packet_type = 0;
    r_pk1.node_id = 1;
    r_pk1.timestamp = getTimeStamp();
    r_pk1.packet_id = 1;
    r_pk1.pos_x = 70;
    r_pk1.pos_y = 40;
    r_pk1.pos_z = 700;
    e = 0;
    for (int i = 0; i < 50; i++)
    {
        r_pk1.sensor_data[i] = e;
        e++;
    }
    std::vector<uint8_t> a(std::begin(r_pk1.raw),
std::end(r_pk1.raw));
    printArray(r_pk1.raw, sizeof(r_pk1), "r_pk1 array");
    printVector(a, "r_pk1 vector");

    L_P r_pk2;
    r_pk2.satellite_id = 1;
    r_pk2.packet_type = 0;
    r_pk2.node_id = 6751;
    r_pk2.timestamp = getTimeStamp();
    r_pk2.packet_id = 201;
    r_pk2.pos_x = 359;
    r_pk2.pos_y = 179;
    r_pk2.pos_z = 8359;
    for (int i = 0; i < 10; i++)
    {
        r_pk2.sensor_data[i] = e;
        e++;
    }
    std::vector<uint8_t> b(std::begin(r_pk2.raw),
std::end(r_pk2.raw));
    printArray(r_pk2.raw, sizeof(r_pk2), "r_pk2 array");
    printVector(b, "r_pk2 vector");
```

```
   L_P2 s_pk1;
   s_pk1.pos_x = 10;
   s_pk1.pos_y = 20;
   s_pk1.pos_z = 30;
   for (int i = 0; i < 10; i++)
   {
       s_pk1.sensor_data[i] = e;
       e++;
   }
   std::vector<uint8_t> c(std::begin(s_pk1.raw),
std::end(s_pk1.raw));
   printArray(s_pk1.raw, 63, "s_pk1 array");
   printVector(c, "s_pk1 vector");

   L_P2 s_pk2;
   s_pk2.pos_x = 10;
   s_pk2.pos_y = 20;
   s_pk2.pos_z = 30;
   for (int i = 0; i < 10; i++)
   {
       s_pk2.sensor_data[i] = e;
       e++;
   }
   std::vector<uint8_t> d(std::begin(s_pk2.raw),
std::end(s_pk2.raw));
   printArray(s_pk2.raw, 63, "s_pk2 array");
   printVector(d, "s_pk2 vector");

   L_P2 s_pk3;
   s_pk3.pos_x = 359;
   s_pk3.pos_y = 179;
   s_pk3.pos_z = 9000;
   for (int i = 0; i < 10; i++)
   {
       s_pk3.sensor_data[i] = e;
       e++;
   }
   std::vector<uint8_t> h(std::begin(s_pk3.raw),
std::end(s_pk3.raw));
   printArray(s_pk3.raw, 63, "s_pk3 array");
   printVector(h, "s_pk2 vector");

   L_A ack1;
   ack1.satellite_id = 1;
   ack1.packet_type = 1;
   ack1.node_id = 1;
   ack1.timestamp = getTimeStamp();
   ack1.packet_id = 20;
   ack1.free_slots = 0;
   std::vector<uint8_t> g(std::begin(ack1.raw), std::end(ack1.raw));
   printArray(ack1.raw, sizeof(L_A), "ack1 array");
   printVector(g, "ack1 vector");

   L_A ack2;
   ack2.satellite_id = 1;
   ack2.packet_type = 1;
   ack2.node_id = 2;
   ack2.timestamp = getTimeStamp();
   ack2.packet_id = 230;
   ack2.free_slots = 0;
   std::vector<uint8_t> f(std::begin(ack2.raw), std::end(ack2.raw));
   printArray(ack2.raw, sizeof(L_A), "ack2 array");
   printVector(f, "ack2 vector");

   L_A ack3;
   ack3.satellite_id = 1;
   ack3.packet_type = 1;
   ack3.node_id = 3000;
   ack3.timestamp = getTimeStamp();
   ack3.packet_id = 248;
   ack3.free_slots = 0;
   std::vector<uint8_t> j(std::begin(ack3.raw), std::end(ack3.raw));
   printArray(ack3.raw, sizeof(L_A), "ack3 array");
   printVector(j, "ack3 vector");
   int i = 0;
   e = 1;
   spdlog::info("provando funcionamiento estado IDLE sin paquetes");
   for (i = 0; i < 10; i++)
   {
       aloha.control();
       spdlog::info("------------------------------------------------
-----------{}", e);
       e++;
   }

   spdlog::info("provando funcionamiento estado RECEIVE");
   for (i = 0; i < 20; i++)
   {
       if (i == 2)
       {
           aloha.setRX(a);
           aloha.setRX(b);
       }
       aloha.control();
       spdlog::info("------------------------------------------------
-----------{}", e);
       e++;
   }

   spdlog::info("provando funcionamiento estado SEND i rebra ACK");
   for (i = 0; i < 20; i++)
   {
       if (i == 2)
       {
           aloha.setTX(0, 1, 20, c);
           aloha.setTX(0, 2, 230, d);
       }
       if (i == 6)
       {
           aloha.setRX(g);
           aloha.setRX(f);
       }
       aloha.control();
```

```
        spdlog::info("------------------------------------------------
-----------{}", e);
        e++;
    }

    spdlog::info("provando funcionamiento estado SEND i rebra ACK con
back off");
    for (i = 0; i < 50; i++)
    {
        if (i == 1)
        {
            aloha.setTX(0, 3000, 248, h);
        }
        if (i == 4)
        {
            aloha.setRX(g);
        }
        if (i == 35 || i == 45)
        {
            aloha.setRX(j);
        }
        aloha.control();
        spdlog::info("------------------------------------------------
-----------{}", e);
        e++;
    }
    spdlog::info("Lora prove finished.");
    spdlog::info("infinite Loop inition.");
    e = 0;
    int loop = 30;
    while (1)
    {
        if (e % loop == 0)
        {
            aloha.setRX(a);
        }
        if (e % loop == 2)
        {
            aloha.setRX(b);
        }
        if (e % loop == 4)
        {
            aloha.setTX(0, 3000, 248, h);
            aloha.setTX(0, 2, 230, d);
        }
        if (e % loop == 8)
        {
            aloha.setRX(f);
        }
        if (e % loop == 9)
        {
            aloha.setTX(0, 1, 20, c);
        }
        if (e % loop == 13)
        {
            aloha.setRX(g);
        }
```

```
       if (e % loop == 12)
       {
           aloha.setRX(j);
       }
       if (e % loop == 19)
       {
           aloha.setRX(j);
       }
       aloha.control();
       spdlog::info("------------------------------------------------
-----------{}", e);
       e++;
   }
}

void setupLogging()
{
   auto console_sink =
std::make_shared<spdlog::sinks::stdout_color_sink_mt>();
   console_sink->set_level(spdlog::level::trace);
   console_sink->set_pattern("[%Y-%m-%d %H:%M:%S.%e] [%^%l%$] %v");

   auto file_sink =
std::make_shared<spdlog::sinks::basic_file_sink_mt>("rita-lora.log",
true);
   file_sink->set_level(spdlog::level::info); // info.
   file_sink->set_pattern("[%Y-%m-%d %H:%M:%S.%e][%l] %v");

   auto rita_logger = spdlog::logger("rita_logger", {console_sink,
file_sink});
   rita_logger.set_level(spdlog::level::trace);

   spdlog::set_default_logger(std::make_shared<spdlog::logger>(rita_l
ogger));
}

time_t getTimeStamp()
{
   std::chrono::system_clock::time_point tp =
std::chrono::system_clock::now();
   time_t unix_timestamp_ms =
std::chrono::duration_cast<std::chrono::milliseconds>(tp.time_since_e
poch()).count();
   return unix_timestamp_ms;
}

void printVector(std::vector<uint8_t> a, std::string name)
{
   std::stringstream data;
   std::copy(a.begin(), a.end(), std::ostream_iterator<int>(data,
","));
   spdlog::info("Packet data of {}: {}.", name, data.str().c_str());
}

void printArray(uint8_t a[], uint8_t size, std::string name)
{
   std::stringstream data;
   // std::copy(std::begin(a), std::end(a),
std::ostream_iterator<uint8_t>(data, ","));
   for (int i = 0; i < size; i++)
   {
       data << std::to_string(a[i]);
   }
   spdlog::info("Packet data of {}: {}.", name, data.str().c_str());
}
```