



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE GRADO

Título del TFG: Development of a drone-based miniaturized payload for the LoRa communications experiment proof-of-concept

Titulación: Grado en Ingeniería Aeroespacial e Ingeniería de Telecomunicaciones

Autor: Diego Iván Tirado Hernández

Director: Hyuk Park y Lara Fernández Capon

Fecha:

Título del TFG: Development of a drone-based miniaturized payload for the LoRa communications experiment proof-of-concept

Autor: Diego Iván Tirado Hernández

Director: Hyuk Park y Lara Fernández Capon

Fecha:

Resumen

El detector de teledetección e interferencia con radiometría y análisis de vegetación (RITA), es una carga útil de 1U que volará a bordo de Alainsat-1, un CubeSat de 3U. Entre otras cargas útiles, realizará una prueba de concepto de un módulo personalizado LoRa para las comunicaciones espacio-tierra entre el satélite y una red terrestre de sensores del Internet de las Cosas.

La prueba de concepto del experimento de comunicaciones LoRa se ha realizado utilizando varios nodos de tierra IoT y una carga útil miniaturizada basada en un dron. Las comunicaciones se han realizado utilizando dos protocolos MAC compatibles con un escenario IoT: ALOHA puro y CSMA/CA con RTS/CTS. En ambos protocolos, la información útil que se envía son los datos contenidos en el paquete de datos. En este paquete se almacenan los datos obtenidos por el sensor capacitivo de humedad del suelo y el sensor de temperatura.

Para realizar la prueba de concepto del experimento de comunicaciones LoRa, se han realizado dos campañas de medidas. En la primera campaña de medidas se ha comprobado el correcto funcionamiento de los módulos y sensores LoRa. En la segunda campaña de medidas, se han realizado varios experimentos en los que se han probado los protocolos ALOHA puro y CSMA/CA con RTS/CTS. Para poder probar diferentes experimentos con distintas configuraciones de los protocolos, se ha diseñado una estructura de código general en la que tanto los nodos de tierra como la carga útil del dron son controlados por un comando enviado por el usuario. Por tanto, la elección del protocolo a utilizar, así como los parámetros configurables de cada protocolo se envían a través de un comando remoto.

Finalmente, se analizan los resultados obtenidos en ambos protocolos y se concluye cuál de los dos tiene mejores prestaciones frente a un escenario de comunicaciones IoT.

TFG title: Development of a drone-based miniaturized payload for the LoRa communications experiment proof-of-concept

Author: Diego Iván Tirado Hernández

Director: Hyuk Park and Lara Fernández Capon

Date:

Overview

The remote sensing and interference detector with Radlometry and vegeTation Analysis (RITA), is a payload of 1U that will fly onboard Alainsat-1 a 3U CubeSat. Among other tests and 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.

The LoRa communications experiment proof-of-concept has been performed using several ground IoT nodes and a miniaturized drone-based payload. The communications have been performed using two MAC protocols which are compatible with an IoT scenario: pure ALOHA and CSMA/CA with RTS/CTS. In both protocols, the useful information to be sent is the data contained in the Data Packet. In this packet, the data obtained by the capacitive soil moisture sensor and the temperature sensor are stored.

In order to perform LoRa communications experiment proof-of-concept, two measurement campaigns have been realized. In the first measurement campaign, the correct functioning of the LoRa modules and sensors has been tested. In the second measurement campaign, several experiments have been performed in which pure ALOHA and CSMA/CA protocols have been tested. In order to test different experiments with different configurations of the protocols, a general code structure has been designed where both the ground nodes and the drone payload are controlled by a command sent by the user. Therefore, the choice of the protocol to be used as well as the configurable parameters of each protocol are sent through a remote command.

Finally, the results obtained in both protocols are analyzed and it is concluded which of the two has better performance against a IoT communications scenario.

Acknowledgments

First, I would like to thank my co-tutor Lara Fernandez Capon, who has helped and guided me to make this project possible. I would also like to thank Adriano José Camps Carmona and Hyuk Park for making possible my collaboration in the RITA project at the NanoSat Lab.

Finally, a special mention to my family, friends, and couple for their support throughout this double degree.

TABLE OF CONTENTS

RESUMEN	3
OVERVIEW	4
ACKNOWLEDGMENTS	5
TABLE OF CONTENTS.....	6
CHAPTER 1: INTRODUCTION	14
1.1. Introduction.....	14
1.2. LoRa communications experiment proof-of-concept objectives	18
CHAPTER 2: STATE OF THE ART OF TECHNOLOGY USED OR APPLIED	20
2.1. Nanosatellites IoT Communication Technologies	20
2.1.1. Introduction	20
2.1.2. LEO satellite constellation	21
2.1.3. LPWAN technologies for satellite communications	22
2.2. LoRa.....	24
2.2.1. LoRa Network Architecture.....	25
2.2.2. LoRa Physical Layer Parameters.....	26
2.2.3. Physical Layer Frame Format	27
2.3. Media Access Control layer.....	29
2.4. Random-Access protocols	33
2.4.1. Pure ALOHA	34
2.4.2. Carrier Sense Multiple Access	37
2.4.2.1. Carrier Sense Multiple Access with Collision Avoidance	40
CHAPTER 3: APPLIED METHODOLOGY FOR SOFTWARE AND HARDWARE DEVELOPMENT	45
3.1. General architecture of the LoRa communications proof-of-concept experiment	45
3.2. Methodology applied in the software design	47
3.2.1. Arduino Software IDE	47
3.2.2. General design of the code developed for the experiment	51
3.2.2.1. Control commands for the experiment.....	51
3.2.2.2. Overall code design.....	53
3.2.2.3. Pure ALOHA design	65
3.2.2.4. CSMA/CA design	73
3.2.3. Calculation of the adjustable parameters of both protocols	85
3.1.1.1. ALOHA protocol parameters	85
3.1.1.2. CSMA/CA protocol parameters.....	87
3.3. Methodology applied in the hardware design	91
3.3.1. Ground node design	91
3.3.1.1. CubeCell modifications required	98

3.3.1.2.	<i>Design of the stripboard integrated into the CubeCell</i>	100
3.3.1.3.	<i>Sensor calibration</i>	103
3.3.1.4.	Connections between devices	104
3.3.2.	Drone-based miniaturized payload design for LoRa communications	108
CHAPTER 4: FIRST MEASUREMENT CAMPAIGN		111
4.1.	First Measurement Campaign	111
4.2.	Analysis of the results of the experiment	112
CHAPTER 5: SECOND MEASUREMENT CAMPAIGN		115
5.1.	Second Measurement Campaign Specifications	115
5.1.1.	Location chosen for the experiment	116
5.1.2.	Specifications of the different experiments performed	116
5.1.3.	Distribution and location of the ground nodes	117
5.1.4.	Assembly of the payload on the drone and flight path	118
5.2.	Sensor data results	120
5.3.	Results of the pure ALOHA experiment	121
5.3.1.	Analysis of packages transmitted and received	121
5.3.2.	Analysis of packages received during the waiting time	123
5.3.3.	Analysis of successful communications	124
5.4.	Results of the CSMA/CA experiment	127
5.4.1.	Analysis of packages transmitted and received	127
5.4.2.	Analysis of packages received during waiting times	131
5.4.3.	Analysis of successful communications	135
5.5.	Analysis and comparison of the performance of both experiments	138
CHAPTER 6: CONCLUSIONS AND FUTURE DEVELOPMENT		141
CHAPTER 7: BIBLIOGRAPHY		143
CHAPTER 8: APPENDICES		146
8.1.	Work plan	146
8.1.1.	Work packages	146
8.1.2.	Gantt diagram	147
8.2.	Annex A	148
8.2.1.	Analysis of packages transmitted and received	148
8.2.2.	Analysis of packages received during the waiting time	150
8.2.3.	Analysis of successful communications	151
8.3.	Annex B	154
8.3.1.	Analysis of packages transmitted and received	154
8.3.2.	Analysis of packages received during waiting times	158
8.3.3.	Analysis of successful communications	159
8.4.	Code	162

LIST OF FIGURES

Fig. 1.1: Section view of the 3U satellite, with the MWR and LoRa antennas in yellow, and the RITA payload shown in the foreground [12].....	16
Fig. 1.2: Block diagram of the payload components [12]	16
Fig. 2.1: Expected transmission ranges versus Bandwidth of LoRa and other technologies	25
Fig. 2.2: LoRa Network architecture [Source: 16]	26
Fig. 2.3: LoRa frame structure [19]	27
Fig. 2.4: Taxonomy of multiple-access protocols [21]	29
Fig. 2.5: Vulnerable time for pure ALOHA protocol [21]	35
Fig. 2.6: Procedure for pure ALOHA protocol	36
Fig. 2.7: Space/time model of a collision in CSMA [21]	37
Fig. 2.8: Vulnerable time in CSMA [21]	38
Fig. 2.9: Behavior of 1-persistent method [21]	38
Fig. 2.10: Behavior of nonpersistent method [21]	39
Fig. 2.11: Behavior of p-persistent method [21]	39
Fig. 2.12: Procedure for CSMA/CA protocol	41
Fig. 2.13: RTS/CTS Communication with NAV [22]	43
Fig. 2.14: CSMA/CA and NAV	44
Fig. 3.1: Main outline of the communication between the different devices	45
Fig. 3.2: Devices used in ground nodes.....	46
Fig. 3.4: Adjustable parameters of the LoRa physical layer	48
Fig. 3.5: Callback functions.....	49
Fig. 3.6: “void setup” configuration.....	50
Fig. 3.7: Pure ALOHA command architecture and values of the different variables and sending structure of the package	52
Fig. 3.8: CSMA/CA command structure	52
Fig. 3.9: Beacon packet structure	53
Fig. 3.10: Data Packet structure	54
Fig. 3.11: ACK packet structure	54
Fig. 3.12: RTS packet structure	55
Fig. 3.13: CTS packet structure	55
Fig. 3.14: Different categorizations of states, packets, and protocols of the ground nodes code (red) and the drone payload code (blue).....	56
Fig. 3.15: Simplified structure of the “void loop()” function of the ground nodes code.....	57
Fig. 3.16: Simplified structure of the “void OnRxDone()” function of the ground nodes code.....	58
Fig. 3.17: Structure of the “Flag Determination” on the “void OnRxDone()” function of the ground nodes code	59
Fig. 3.18: Structure of the “void OnTxDone()” function of the ground nodes code	59
Fig. 3.19: Simplified structure of the “void OnRxTimeout()” function of the ground nodes code.....	60
Fig. 3.20: Structure of the “void OnTxTimeout()” function of the ground nodes code.....	60
Fig. 3.21: Simplified structure of the “void loop()” function of the drone payload code	61

Fig. 3.22: Simplified structure of the “ <i>void OnRxDone()</i> ” function of the drone payload code.....	62
Fig. 3.23: Structure of the “ <i>Flag Determination</i> ” on the “ <i>void OnRxDone()</i> ” function of the drone payload code	63
Fig. 3.24: Structure of the “ <i>void OnTxDone()</i> ” function of the drone payload code	63
Fig. 3.25: Simplified structure of the “ <i>void OnRxTimeout()</i> ” function of the drone payload code.....	64
Fig. 3.26: Structure of the “ <i>void OnTxTimeout()</i> ” function of the drone payload code	65
Fig. 3.27: Structure of the different states of the pure ALOHA protocol in the <i>void loop()</i> function of the drone payload.	66
Fig. 3.28: Case where a beacon is received in the “ <i>void OnRxDone()</i> ” function in the ground nodes code.	67
Fig. 3.29: Structure of the different states of the pure ALOHA protocol in the “ <i>void loop()</i> ” function in the ground nodes code.	68
Fig. 3.30: Structure of the processes of the pure ALOHA protocol in the <i>void OnRxDone()</i> function of the drone payload.	69
Fig 3.31: Case where a ACK packet is received in the “ <i>void OnRxDone()</i> ” function in the ground nodes code.	70
Fig. 3.32: Case where a Data Packet is received in the “ <i>void OnRxDone()</i> ” function in the ground nodes code	71
Fig. 3.33: “ <i>void OnRxTimeout()</i> ” function of the ground nodes code	72
Fig. 3.34: Wrong packet case on the ground nodes code.....	73
Fig. 3.35: Structure of the different states of the CSMA/CA protocol in the <i>void loop()</i> function of the drone payload.	74
Fig. 3.36: Case where a beacon is received in the “ <i>void OnRxDone()</i> ” function in the ground nodes code.	75
Fig. 3.37: Structure of the <i>void loop()</i> function of the ground nodes code.	76
Fig. 3.38: Case where a RTS is received in the “ <i>void OnRxDone()</i> ” function in the ground nodes code.....	76
Fig. 3.39: Case where a CTS is received in the “ <i>void OnRxDone()</i> ” function in the ground nodes code.....	77
Fig. 3.40: Case where a ACK is received in the “ <i>void OnRxDone()</i> ” function in the ground nodes code.....	77
Fig. 3.41: Case where a Data Packet is received in the “ <i>void OnRxDone()</i> ” function in the ground nodes code.	78
Fig. 3.42: “ <i>void OnRxTimeout()</i> ” function of the ground nodes code.....	79
Fig. 3.43: Structure of the case where a RTS packet is received in the “ <i>void OnRxDone()</i> ” function of the drone payload.....	80
Fig 3.44: Structure of the case where a Data Packet is received in the <i>void OnRxDone()</i> function of the drone payload.	82
Fig. 3.45: Wrong packet case in the CSMA/CA protocol on the ground nodes code.....	84
Fig. 3.46: Further location of nodes and drone in the experiment scenario	86
Fig. 3.47: Outline of the CSMA/CA communication	90
Fig. 3.48: Capacitive Soil Moisture Sensor v1.2	92

Fig. 3.49: Missing voltage regulator in the capacitive soil moisture sensor.
93

Fig. 3.50: capacitive soil moisture sensor schematic [24] 93

Fig. 3.51: Comparison of the different locations of the via hole in the capacitive soil moisture sensor..... 94

Fig. 3.52: Solution to the unresponsive problem in the capacitive soil moisture sensor..... 94

Fig. 3.53: Temperature Accuracy vs. Temperature [25]..... 95

Fig. 3.54: HDC1080 sensor 96

Fig. 3.55: First design proposed for the ground node 97

Fig. 3.56: CubeCell HTCC-AB01 Pinout Diagram [26]..... 98

Fig. 3.57: Schematic of the AO7801 chip [26] 99

Fig. 3.58: Resistance BR01 to remove [26] 99

Fig. 3.59: Experimental VAO_{max} measurements of the capacitive soil moisture sensor..... 101

Fig. 3.60: Final model of the stripboard integrated into the CubeCell 102

Fig. 3.61: Connections of the stripboard integrated into the CubeCell... 103

Fig. 3.62: Code for reading soil moisture and temperature sensors 104

Fig. 3.63: Corrupt data and data loss of the SD reading..... 105

Fig. 3.64: Battery regulator with the 3,7 V LiPo battery..... 106

Fig. 3.65: Connections between different devices. 107

Fig. 3.66: Assembly process of the different nodes 107

Fig. 3.67: Drone 3D Robotics Iris+ with the payload design 108

Fig. 3.68: Connection diagram of the different devices in the payload design
109

Fig. 3.69: Miniaturized drone-based payload for LoRa communications and GNSS-R 110

Fig. 4.1: Receiver and transmitter of the first experiment. 111

Fig. 5.1: Map of the scenario where the measurement campaign is performed..... 116

Fig. 5.2: Different characteristics of the experiments..... 117

Fig. 5.3: Location of the different groups with the respective nodes 118

Fig. 5.4: *Drone Condor* with assembled payload. 119

Fig. 5.5: Flight path of the experiment 119

Fig. 5.7: Sensor data results..... 120

Fig. 5.8: Pure ALOHA – Average percentage of Data Packets received versus Data Packet sent by the nodes (%). 122

Fig. 5.9: Pure ALOHA – Percentage of ACK received versus ACK sent by the drone (%). 123

Fig. 5.10: Pure ALOHA – Percentage of the average packages received during the waiting time. 124

Fig. 5.11: Pure ALOHA – Average percentage of ACK received versus data packets sent. 125

Fig. 5.12: Pure ALOHA – Average percentage of successful communications versus failed communications after K_{max} attempts..... 126

Fig. 5.13: Pure ALOHA – Average percentage of attempts needed for a successful communication. 126

Fig. 5.14: CSMA/CA – Average percentage of RTS received versus RTS sent by the nodes 128

Fig. 5.15: CSMA/CA – Average percentage of CTS received versus CTS sent by the drone.....	129
Fig. 5.16: CSMA/CA – Average percentage of Data Packet received versus Data Packet sent by the nodes.....	130
Fig. 5.17: CSMA/CA– Average percentage of ACK received versus ACK sent by the drone.....	130
Fig. 5.18: CSMA/CA – Average percentage of times the CTS has been received against times the wait time has expired	132
Fig. 5.19: CSMA/CA – Average percentage of packages received during the waiting time to receive the CTS.....	133
Fig. 5.20: CSMA/CA – Average percentage of times the Data Packet has been received against times the wait time has expired	134
Fig. 5.22: CSMA/CA – Average percentage of ACK received versus RTS sent.	135
Fig. 5.23: CSMA/CA – Average percentage of successful communications versus failed communications after K_{max} attempts	136
Fig. 5.24: CSMA/CA – Average percentage of attempts needed for a successful communication	137
Fig.5.25: Comparison of communication success in experiments 1 and 6	138
Fig. 5.26: Comparison of communication success in experiments 3 and 4	139

LIST OF TABLES

Table 2.1: Main LPWAN technologies comparison [Source:16]	24
Table 3.1: Time On Air of the different packets in the CSMA/CA protocol. 88	
Table 3.2: Important characteristics of the HDC1080 sensor [25].....	95
Table 3.3: Conversion time in function of the resolution of the HDC1080 sensor [25]	96
Table 3.4: Maximum analog output voltage (V_{AOmax}) of the capacitive soil moisture sensors	100
Table 4.1: Soil moisture measurements	112
Table 4.2: Realistic soil moisture values [28]	114

LIST OF ACRONYMS

IOT	Internet Of Things
LORA	Long Range
LPWAN	Low Power Wide Area Network
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
RTS	Request To Send
CTS	Clear To Send
SIFS	Short InterFrame Space
DIFS	Distributed coordination function InterFrameSpace
ACK	ACKnowledgement
NAV	Network Allocation Vector
SF	Spreading Factor
CR	Coding Rate
BW	Bandwidth
CF	Carrier Frequency
TP	Transmission Power
FEC	Forward Error Correction
CRC	Cyclic Redundancy Check
LLC	Logical Link Control
OSI	Open Systems Interconnect
SDR	Software Defined Radio
MAC	Media Access Control
LEO	Low Earth Orbit
MEO	Medium Earth Orbit
GEO	Geosynchronous Equatorial Orbit
HEO	High Earth Orbit
PO	Polar Orbit
SSO	Sun Synchronous Orbit
RTT	Round Trip Time
COTS	Commercial Off-The-Shelf
TOA	Time On Air
RFI	Radio Frequency Interference
FMPL	Flexible Microwave Payload
GNSS-R	Global Navigation Satellite System Reflectometry
FSPL	Free Space Path Loss
3GPP	Third Generation Partnership Project
QPSK	Quadrature Phase Shift Keying
MNO	Mobile Network Operator
CSS	Chip Spread Spectrum

CHAPTER 1: INTRODUCTION

1.1. Introduction

In the early days of space exploration, satellites were large objects that cost large amounts of money and took many years to build in the 1950s, the Soviet Union initiated the Sputnik satellite project, the first artificial satellite to orbit the Earth. This event marked the beginning of a competition between two powerful countries that resulted in a technological development in aerospace sciences. The satellites created later were satellites designed for very specific missions. Each mission had its own subsystems to meet the requirements of a given project. Today, numerous satellites are in orbit to provide us with different applications such as positioning systems, Earth observation and communications. Among all applications, telecommunications have been the most widespread and used application. This is because with the use of satellites we are able to cover the entire surface of the Earth, providing connectivity to remote or isolated areas that are practically impossible to cover by other means.

Traditionally, designing and developing a satellite is difficult, complex, expensive and takes long periods of time to develop. However, decades later a new paradigm was established that significantly reduced the size of some satellites. In 1999, the CubeSat [1] standard emerged, which allowed access to space, offering satellite development opportunities to institutions that did not have access. The basic design of a CubeSat consists of a 10 cm cube, called 1U, which contains the primary subsystems for proper operation. They are used to study the behavior of different technologies in a space environment and a wide range of services such as communications or Earth observation.

Within these communication satellites, Internet Of Things (IoT) has had a lot of momentum in the past years. The IoT is a burgeoning paradigm that points out a novel direction of future internet, in which devices are provided with Internet connection and some software intelligence. These capabilities allow for IoT devices to be controlled remotely, enabling access to an ecosystem of various services. Moreover, through those easy accesses, various kinds of IoT devices such as, environmental monitoring sensors, smart household electrical appliances, actuators, vehicles, among others, are able to exchange data with IoT networks and provide unlimited services to a multitude of users: individual users, enterprise users, government users, military users, etc.

IoT devices can be classified according to communication ranges: short-range and Low Power Wide Area Network (LPWAN) [2]. Compared to short-range connections based on Zigbee or Bluetooth, LPWANs have a longer communications range still with low power consumptions, and are more suitable for rural or industrial scenarios, such as smart grid and environmental monitoring. The main LPWAN technologies are Sigfox [3], NarrowBand-IoT (NB-IoT) [4], and LongRange (LoRa) [5]. Each of these devices communicates independently with

a gateway or base station, which in turn connects to the network, to make data available. This required infrastructure is feasible to deploy in rural areas. However, in remote areas, where the placement of gateways requires the deployment of considerable infrastructure, satellites are used to communicate with these devices [2]. In addition, the demand for connectivity is increasing worldwide. It is estimated that the IoT communications market will have an impact on the economy of close to three to eleven trillion dollars per year by 2025 [6]. In this scenario, satellite technology seems to offer a critical solution to the global connectivity problem. However, traditional satellites are expensive, so cheaper satellite solutions have become the focus of growing interest. By bringing together the need for greater coverage of IoT networks and new technologies that offer smaller and cheaper satellites, a constellation of CubeSat satellites orbiting in the Earth's low orbits can be the best answer to the global connectivity that IoT demands.

Space-to-Earth communications are a challenge due to long distances, attenuations and satellite movement. For that reason, it is necessary to study the different LPWAN technologies and determine which one is the most appropriate for use in this type of communication. In addition, these wireless communications must be controlled with medium access layer mechanisms, to ensure the proper coordination of frame transmissions, together with the logic for retransmissions and the recovery of data in case of collisions. Among LPWANs, LoRa is a novel technology that has gained great interest in recent years for satellite communications. Several studies have evaluated the limitations of the technology in space-to-Earth communications [2], in addition to studying the different MAC protocols for LoRa modulation. The most frequently used one is LoRaWAN, proposed by the LoRa Alliance, and it uses an extensive network of gateways denominated the Things Network. However, it has been demonstrated that this MAC protocol has certain capacity limitations [7]. Aside from LoRaWAN, other different protocols have been studied to enhance the capacity and the range of LoRa networks [8][9]. However, all these papers considered an architecture in which nodes are always in range of the gateway. In a scenario where the satellite is the gateway, the different IoT devices are always not within reach of the gateway, which is why it is necessary to consider MAC protocols suited to a scenario where the gateway is not accessible at every moment and where multiple devices try to access the medium simultaneously. In general, the most suitable protocols for IoT satellite communication scenarios were identified in the article [10], where a state-of-the-art study of the proposed protocols is presented providing different types of metrics. Two of the protocols proposed in [10] have been selected to perform the proof of concept in this thesis: pure ALOHA and CSMA/CA with RTS/CTS.

The Remote sensing and Interference detector with RadlomeTry and vegetation Analysis (RITA) payload, carried out by the NanoSat Lab an organization of the "*Universitat Politècnica de Catalunya*" (UPC), is one of the Remote Sensing payloads selected by the 2nd GRSS Student Grand Challenge in 2019 to fly on board of a 3U satellite that is being developed at the National Space Science and Technology Center (NSSTC), United Arab Emirates University [11]. The main objectives of RITA are to perform microwave radiometry measurements at L-

band, vegetation analysis using a hyper-spectral camera, Radio-Frequency Interference (RFI) detection and classification, and a technology demonstration of sensor networks using a custom LoRa transceiver. Radiometry measurements, RFI detection, and the LoRa experiment will be performed using a Software Defined Radio (SDR) with a frontend designed as a modified version of the Flexible Microwave Payload 1 (FMPL-1) used in the 3Cat-4 mission.



Fig. 1.1: Section view of the 3U satellite, with the MWR and LoRa antennas in yellow, and the RITA payload shown in the foreground [12]

The LoRa IoT module [12] embarked as part of the RITA payload will be a proof-of-concept payload to verify the communications space-to-Earth with an SDR-based LoRa modulation and a radiofrequency Front-End for signal conditioning. In the figure 1.2 it can be observed the block diagram of the RITA payload components.

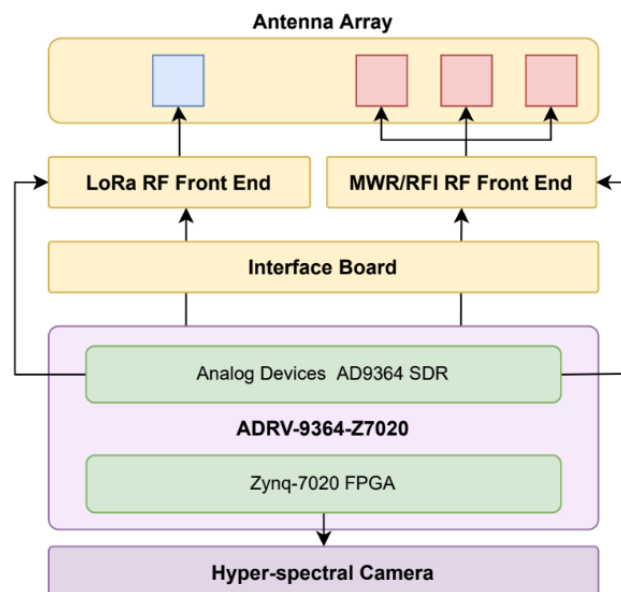


Fig. 1.2: Block diagram of the payload components [12]

To test LoRa communications using different MAC protocols and compare which one is the most suitable, multiple LoRa modules with sensors will be placed in a target area, and these sensors will communicate with the LoRa module of the RITA payload using the HelTec CubeCell Dev-Board HTCC-AB01 transceiver.

In the scenario, the RITA satellite orbits the Earth in a LEO orbit and communicates with the different ground nodes located around the Earth. These ground nodes are located with different spatial densities depending on the environmental disaster to be measured [13]. Given the low altitude of satellites in LEO orbit, they are not seen statically from the Earth. In fact, from a fixed point on Earth, a satellite in LEO is only seen for 8 to 10 minutes, depending on the latitude and longitude, where the node is located. This creates a disruption from the nodes' point of view, as they do not know when a satellite is available to transmit data packets. Therefore, the satellite transmits a periodic beacon every 8 minutes, which reaches all nodes within range. This beacon ensures that all nodes are aware that a satellite is ready to receive or transmit data. In addition to the beacon, acknowledgment packets (ACK) and control packets (CTS) are also sent.

The structure of this project is divided into 6 chapters, this first chapter is an introduction to the work. The second chapter is the State of the Art of technology used or applied. This chapter explains the technologies used in this project, including IoT technology, LoRa technology and MAC protocols. The third chapter details the applied methodology for software and hardware development. This chapter is separated into two main parts. Firstly, it explains how the software has been implemented using flowcharts. Secondly, it is described how the hardware has been developed for both the ground nodes and the miniaturized drone-based payload. The fourth chapter explains how the first measurement campaign was conducted and what results and conclusions were obtained. In this campaign LoRa communications were tested using LoRa modules and one of the sensors used was also tested. The fifth chapter is structured in five parts. The first part describes how the second measurement campaign was performed. Secondly, the results of the soil moisture and temperature sensors are analyzed. Thirdly and fourthly, the results obtained in the pure ALOHA and CSMA/CA experiments are analyzed, respectively. Fifth, both protocols are analyzed and compared. Finally, the sixth chapter contains the conclusions obtained in this project and discusses possible improvements and future developments.

1.2. LoRa communications experiment proof-of-concept objectives

The main idea of this thesis is to perform a proof of concept of the different media access protocols for LoRa communications using a miniaturized drone-based payload and several IoT ground nodes including two types of sensors (capacitive soil moisture sensor and temperature sensor) and the HelTec CubeCell Dev-Board HTCC-AB01 transceiver, which can communicate using the LoRa physical layer. In this experiment, two types of MAC protocols suitable to IoT environments will be tested to communicate with the miniaturized drone-based payload and the obtained results will be analyzed. The main objectives are the followings.

The first objective of the thesis is to design and implement pure ALOHA and CSMA/CA protocols in a single code on the HelTec CubeCell Dev-Board HTCC-AB01 transceiver. In the experiment, both protocols will be tested separately, so it is necessary to design a code which can switch between protocols when desired. This will be done through a command which will be sent by the user. This command contains the necessary information of the protocol to be used and reaches both the IoT devices and the gateway, which will be implemented in a drone that will perform the function of the satellite. Once the command reaches the drone's payload, it sends a beacon to initiate communications with the chosen protocol.

The second objective of the thesis is to design the IoT ground nodes and the drone-based miniaturized payload to be assembled on a drone. To perform the proof-of-concept, it is necessary to simulate the IoT ground nodes with various sensors and the payload concerning the LoRa communications of the RITA satellite. A soil moisture sensor and a temperature sensor, will be considered in the design of the IoT ground nodes. The data collected by these sensors shall be sent as data packets in the different protocols. In the design of the drone-based miniaturized payload, a lightweight and space-efficient design should be considered. Since the RITA payload is equipped with a software-defined radio for microwave radiometry and RFI detection, a hyperspectral camera and a LoRa transceiver (which will work together to produce more accurate vegetation-related measurements), the drone payload should also consider space for both experiments and test them simultaneously as in a realistic case. For that reason, the design of the drone payload considers two systems. The system for the LoRa communications experiment and the system for the GNSS-R experiment were performed by a member of the NanoSat Lab.

The third objective is to test the implemented protocols in a measurement campaign in which the miniaturized drone-based payload will be assembled on a drone and fly around a given area where the different ground nodes will be located. The ground nodes will be clustered with different densities and different experiments will be performed with the implemented protocols where the efficiency will be tested as a function of channel saturation. The ground nodes will capture the data and process it to determine which protocol has the best performance.

Initially, one of the parts of this thesis only proposed the implementation of the two MAC layer protocols in the transceivers of the ground nodes to communicate with the RITA LoRa module. However, due to certain setbacks, a CubeCell LoRa transceiver has finally been used as a payload of the drone to be able to make communications with the IoT nodes. Therefore, the code of the drone-based payload has also had to be designed and implemented to be able to carry out the campaign of measures and not delay the delivery of this final degree project.

CHAPTER 2: State of the art of technology used or applied

2.1. Nanosatellites IoT Communication Technologies

2.1.1. Introduction

The Internet of Things (IoT) is a revolutionary technology that aims to connect devices (or "things") placed all over the world for different applications such as environmental monitoring, security, among others. To perform all these applications, it is necessary to have devices that store this information and are able to transmit it. Therefore, these devices require the ability to transmit and receive information. In addition, they also require connectivity to a network such as the Internet or other private networks. In some cases where these devices are used, they are in rural or remote areas which are difficult to access. For this reason, these devices have been designed to be low-powered and thus reduce the interaction with maintenance. IoT devices are often powered by batteries or solar panels if the application of use allows it. Some more modern devices are capable of generating power from the environment, achieving up to several years of autonomy. However, this low-power profile limits the power transmitted to communicate. This causes the communication range and data rate to be affected. Considering these limitations, different IoT technologies have emerged over the years to meet these requirements. Different standards have appeared. Some examples are IEEE 802.15.4, Bluetooth Low Energy (BLE), and ZigBee. However, if the application of using the devices must be placed in rural or remote areas, it is necessary to deploy an expensive infrastructure for the devices to be connected to it in order to be connected to a network.

To minimize infrastructure costs in rural areas and maximize the reach of different devices, different IoT technologies have emerged. Some of them are classified as low-power wide area networks (LPWANs), which are optimal for IoT applications, as they only require sending small amounts of information over long distances. These new LPWAN technologies have emerged in both licensed and unlicensed frequency bands. Among them, Sigfox, LoRa, and NB-IoT are the current leaders.

LPWANs can cover between 1 km and 10 km in urban areas and between 10 km and 20 km in rural areas [14]. The way in which the different devices communicate is as follows. Each device communicates independently with a gateway or base station that is connected to the network. This communication is bidirectional, so both devices (devices and gateway) receive and transmit information. However, the uplink is defined as the messages sent from the devices to the gateway. On the other hand, a downlink is defined as messages from the gateway to the devices. Since the gateway is the central node between all devices and they are able to communicate over long distances, infrastructure costs are reduced in rural areas. However, the deployment of this infrastructure in remote areas such as the poles or oceans remains costly and very complex.

Therefore, the design of an alternative infrastructure capable of covering such remote areas remains necessary. A feasible solution for such scenarios is a constellation of low earth orbit (LEO) satellites [6]. By using a LEO satellite constellation, it is possible to achieve global coverage, thus managing to cover the most remote areas. In the next section, technical aspects of the LEO satellite constellation are discussed. Then, LPWAN technologies to be embarked in CubeSat platforms for satellite communications are discussed. In this section, different LPWAN technologies are presented, and it is demonstrated why LoRa technology is the selected technology to be applied in this project.

2.1.2. LEO satellite constellation

This section categorizes the different types of orbits and presents the characteristics that favor satellites in low Earth orbits (LEO) over other types of orbits such as geostationary orbits (GEO).

Orbit types can be classified in several ways, by central body, inclination, eccentricity, direction or synchronicity. However, orbit types are usually classified by altitude. Within this categorization we find the low Earth orbit (LEO), medium Earth orbit (MEO), geosynchronous orbit (GEO) and high Earth orbit (HEO). All these orbits fall into the geocentric orbit family, as all these satellites orbit the Earth.

LEO orbits are circular or elliptical orbits at an altitude between 200 km and 2000 km. The orbital period depends on the latitude and varies between 88 minutes and 127 minutes. On the other hand, the velocity reached by these satellites is 27000 km/h, completing a total of 16 orbits around the Earth. For this reason, the maximum time that a satellite is above the local horizon for a terrestrial observer is up to 20 min. This time is used to transfer data to ground stations. This type of satellites, being in a low orbit, have a minimum atmospheric resistance that causes the gradual deterioration of the equipment and its permanence in space is limited. Satellites can have orbits inclined between 0 and 90 degrees with respect to the equatorial plane. There are two types of orbits derived from LEO orbits. Polar orbits (PO) and sun synchronous orbits (SSO). A polar orbit is a type of low Earth orbit in which the satellites pass approximately over the poles of the planet. The approximate inclination is 90 degrees, although a deviation of 20 to 30 degrees is also accepted as a polar orbit. Sun synchronous orbits (SSO) is a type of polar orbit. Objects in this orbit are synchronized with the Sun, so they pass over a region of the Earth at the same local time every day.

Satellites in MEO orbit are located between 2000 km and 35786 km. Satellites orbiting in this zone are mostly used for geographic positioning, such as GPS, Galileo and GLONASS. The most commonly used altitude is 20200 km, with an orbital period of 12 hours.

Satellites in GEO orbit are located at an altitude of 35786 km. The ideal satellites for this zone are those destined for telecommunications, since the orbital period of the satellite is the same as that of the Earth's rotation and it is more difficult for

them to lose the signal. The geosynchronous equatorial orbit is a type of GEO orbit whose inclination is 0° , that is, the satellite's position is always maintained on the equator of the celestial plane.

Finally, the satellites in HEO orbit are at a level above 35786 km altitude. The orbital periods in this zone are longer than 24 hours. For this reason, satellites located in this zone have an apparent backward movement. Since their velocity is lower than that of the Earth's rotation, visually the satellite moves in the opposite direction to the common objects in the sky.

Depending on the type of orbit a satellite follows, there are certain advantages and disadvantages. Some of the advantages of LEO orbits over GEO orbits are mentioned in the following paragraph.

LEO satellite constellation technology has unique advantages compared to GEO systems. The first advantage is that a LEO satellite constellation has a shorter propagation delay because it has a lower altitude orbit compared to GEO satellites. The propagation delay is quantified by the round-trip time (RTT). A satellite in LEO orbit has an RTT less than 100 ms, meanwhile, a satellite in GEO orbit has an RTT greater than 600 ms [15]. Therefore, a constellation of LEO satellites has less latency than a constellation of GEO satellites. The second advantage is related to lower propagation losses due to a shorter distance between the ground devices and the satellite. However, satellites in LEO orbit have some disadvantages. Satellites in this type of orbit suffer from communication disruptions, as they are not always visible to a ground-based device. This is why a constellation of satellites orbiting the earth is necessary to ensure global and uninterrupted coverage between satellite handovers.

Therefore, satellite constellations in LEO orbit offer greater advantages than constellations in GEO orbit. In order to cover certain remote areas of the Earth and have global coverage, it would be necessary to design a LEO satellite constellation with a certain configuration. Employing a LEO constellation of CubeSats using LPWAN gateways would provide coverage to all those IoT devices located in remote areas at a reasonable cost. To enable different IoT devices to communicate with satellites, certain limitations must be considered that are not taken into account in the vast majority of terrestrial communications. In space-to-ground communications, channel losses and Doppler frequency shifts in the signal carrier must be considered. These losses are modeled with the Free Space Path Loss (FSPL) model, as line-of-sight is achieved between the satellite and the terrestrial IoT device. In addition, signals can also be attenuated by effects such as atmospheric absorption and other weather conditions such as rain and clouds. Therefore, it is necessary to evaluate the feasibility of onboard LPWAN technologies on satellites.

2.1.3. LPWAN technologies for satellite communications

The most widely used LPWAN technologies are Sigfox, NarrowBand-IoT (NB-IoT), and LoRa. In order to evaluate the different technologies presented above,

a comparison of the characteristics of the Physical Access Control/Media (PHY/MAC) layer is considered. Among these characteristics used in LEO space-to-Earth communications are modulation-coding techniques, frequency band, maximum data rate, etc.

Sigfox is an LPWAN network operator offering an end-to-end IoT connectivity solution based on its patented technologies. Sigfox deploys its own base stations equipped with software-defined cognitive radios and connects them to back-end servers via an IP-based network. End devices connect to these base stations using binary phase shift keying (BPSK) modulation on an ultra-narrow (100 Hz) sub-GHz ISM band carrier. Sigfox uses unlicensed ISM bands, for example, in Europe an 868 MHz band is used, in North America 915 MHz, and in Asia 433 MHz. By employing the ultra-narrow band, Sigfox uses the frequency bandwidth in an efficient manner and suffers from very low noise levels, leading to very low power consumption and high receiver sensitivity with a maximum throughput ranging from 100 to 600 bps. On the other hand, in some of these bands the transmitted power can be up to 22 dBm, and due to the modulation used the received power sensitivity is -126 dBm. In addition, Sigfox technology is able to compensate for frequency drift of up to ± 30 Hz [16]. In addition, the MAC layer protocol is tolerant to the delay experienced when communicating through LEO satellites. However, the deployment of the base stations is the exclusive responsibility of Sigfox, so it is not possible for other companies to embark gateways on their satellites.

NB-IoT, referred to as cellular LPWAN, has been developed by the Third Generation Partnership Project (3GPP) and is being integrated as part of 4G and 5G networks. This technology uses narrowband quadrature phase shift keying (QPSK) modulation in a licensed band, with a maximum transmit power of 23 dBm and a sensitivity of -125 dBm. The data rate of NB-IoT technology is 26 kbps from the base station to the devices and 66 kbps from the devices to the base stations with eventually peaks at up to 250 kbps. The PHY/MAC layer protocol of the NB-IoT technology is affected by both delay and Doppler. Therefore, this protocol cannot be used without being adapted for space and Earth communications. In addition, base stations are deployed by mobile network operators (MNOs), so it is yet another limitation of the use of this LPWAN technology.

LoRa is a long-range wireless communications system which uses a patented Chirp Spread Spectrum (CSS) modulation, which is more resilient than others to interference and jamming. LoRa uses unlicensed ISM bands as Sigfox and has several parameters that must be configured in the transceivers. These configurable parameters are transmitted power, bandwidth (BW), spreading factor (SF), and coding rate (CR). The transmitted power can be up to 22 dBm, and the sensitivity can be up to -125 dBm, offering a data rate of up to 27 kbps. LoRa technology can be used with several different MAC layer protocols, being LoRaWAN the most established. There are several manufacturers offering both LoRa modules and gateways as COTS components. Therefore, it is entirely feasible to propose a satellite gateway solution based on LoRa technology. In fact, multiple studies and experiments have been carried out demonstrating the

great features of LoRa technology in space communications. Below is a summary table of the properties of each of the above LPWAN technologies.

Table 2.1: Main LPWAN technologies comparison [Source:16]

	Sigfox	NB-IoT	LoRa
Bandwidth (kHz)	0.1	200	125, 250, 500
Modulation	BPSK	QPSK	CSS
Frequency Band (MHz)	ISM	LTE	ISM
P_r (dBm)	22	23	22
P_s (dBm)	-126	-125	-125
Maximum data rate (kbps)	0,1 - 0,6	250	27
Delay Tolerant	Yes	No	Yes
Network Interconnection Device	Sigfox Base Station	LTE Base Station	LoRa Gateway

Looking at the above table and considering the above mentioned for each of the LPWAN technologies, the technology with the highest compatibility for satellite communications is LoRa modulation. In addition, there are several manufacturers which offer LoRa modules, making it much easier to deploy gateways compared to NB-IoT and Sigfox. For these reasons, LoRa is the technology studied and used in this project. In the following section, we will go into more detail about LoRa technology.

2.2. LoRa

LoRa defines a physical layer technology developed by Cycleo in 2010, a company that two years later was acquired by Semtech. This technology is suitable for applications that transmit little data at low bit rates. One of the properties of LoRa is that data can be transmitted over longer distances compared to technologies such as Wi-Fi, Bluetooth, or ZigBee. The figure below shows some access technologies that can be used for wireless data transmission and their transmission ranges versus bandwidth.

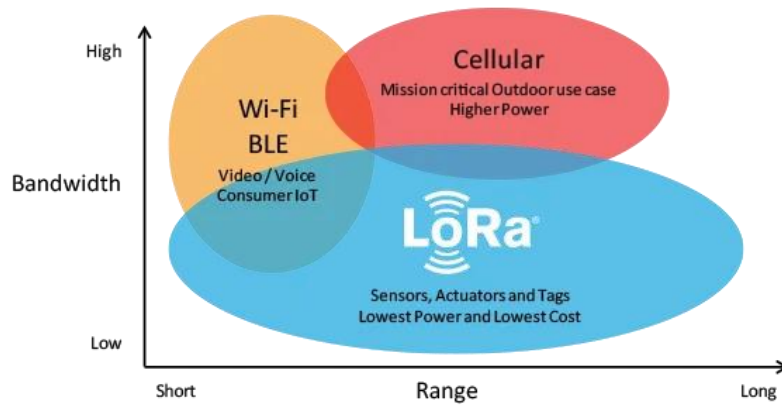


Fig. 2.1: Expected transmission ranges versus Bandwidth of LoRa and other technologies

LoRa, is a modulation technique based on spread spectrum techniques and a variation of chirp spread spectrum (CSS). The LoRa chirp spread spectrum (CSS) modulation uses frequency chirps with a linear variation of frequency over time in order to encode information. Because it uses spread spectrum modulation techniques, it uses the entire channel bandwidth to transmit a signal. This makes the signal robust to channel noise.

Other key features that make LoRa stand out from other IoT technologies [18] are the following. First, LoRa modulation is bandwidth and frequency scalable. Frequency hopping can be used in narrow band and wideband direct sequence applications. Second, it has a low power consumption. Third, has low noise levels, making it highly resistant to interference, and difficult to detect or jam. Fourth, LoRa is doppler resistant. Frequency offsets between the transmitter and the receiver are equal to the timing offsets due to the linearity of the chirp. Fifth, LoRa enhanced network capacity. LoRa allows multiple spread signals to be transmitted at the same time and on the same channel without any degradation. This is due to the use of orthogonal spreading factors. Finally, LoRa can be used for ranging and localization. LoRa has the ability to linearly discriminate frequency and time errors. It is an ideal modulation for radar applications and is therefore suitable for ranging and location applications such as real-time location services.

2.2.1. LoRa Network Architecture

A typical LoRa network includes three types of devices: End-devices (IoT Devices), Gateway/Base Station and Network Server, as shown in the figure 2.2.

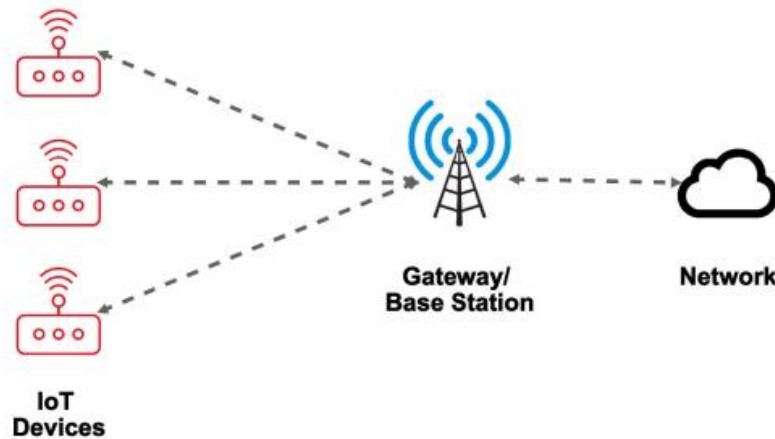


Fig. 2.2: LoRa Network architecture [Source: 16]

Communication is bidirectional, defining the uplink and the downlink as follows. The uplink messages are messages sent by the end devices (IoT devices) to the Network Server relayed by one or more gateways. On the other hand, the downlink messages are messages sent by the Network Server to a single end device, which are relayed by a single gateway.

2.2.2. LoRa Physical Layer Parameters

In order to achieve the best performance in a given scenario, different parameters can be configured: Bandwidth (BW), Spreading Factor (SF), Code Rate (CR), Transmission Power (TP), and Carrier Frequency (CF).

With the aim of improve the spectral efficiency and the network capacity, the LoRa modulation presents six orthogonal spreading factors (SF7, SF8, ..., SF12), that result in six different data rates. For an available bandwidth, a higher spreading factor reduces the bit rate and reduces the battery life by increasing the transmission time. A given propagation factor (SF) and bandwidth (BW) gives a bit rate defined by (2.1):

$$BitRate = SF \cdot \frac{BW}{2^{SF}} \quad (2.1)$$

In LoRa modulation the BW is configurable, and it can be set to 125, 250, or 500 kHz. Higher BW gives a higher data rate, but a lower sensitivity because of integration of additional noise. In the other hand, a lower BW gives a higher sensitivity, but a lower data rate.

CR is the Forward Error Correction (FEC) rate used by the LoRa modem that offers protection against bursts of interference, and can be set to either 4/5, 4/6, 4/7 or 4/8. A higher CR offers more protection but increases time on air. This CR provides a code gain that for the LoRa modulation is not specified. In fact, the sensitivity depends only on the SF and the BW. Therefore, since CR does not influence sensitivity, CR will not be considered in the link budget discussed in the following sections.

These three modulation parameters determine the capacity (C) in bps of the channel, which is computed as shown in (2.2):

$$C = SF \cdot \frac{CR}{\left[\frac{2^{SF}}{BW} \right]} \quad (2.2)$$

As can be seen in (2.2), the capacity increases with an increase in BW, a decrease in SF and a decrease in redundancy. The transmitted power can be up to 22 dBm, however, depending on the implemented hardware it can be significantly improved. Finally, the carrier frequency (CF) is the center frequency that can be programmed in 61 Hz steps between 137 MHz and 1020 MHz [17]. Depending on the LoRa chip, this range may be limited to 860 MHz and 1020 MHz.

2.2.3. Physical Layer Frame Format

Although arbitrary frames can be transmitted in LoRa modulation, Semtech has specified a physical frame format in which the bandwidth and spreading factor are constant for a frame [19] as can be seen in the figure 2.3:

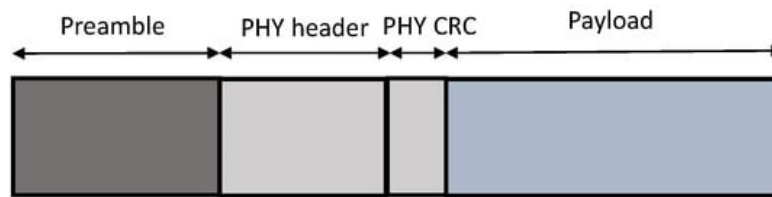


Fig. 2.3: LoRa frame structure [19]

The LoRa frame starts with a preamble. The preamble begins with a sequence of upchirps (signal at which the frequency increases) covering the entire frequency band. The last two upchirps encode the sync word, which is a value used to differentiate LoRa networks using the same frequency bands. After the preamble comes an optional header indicating the size of the payload, the code rate used for the end of the transmission and if there is a cyclic redundancy check (CRC) at the end of the payload. It also contains a CRC so that the receiver can

discard packages with non-valid headers. After the header the payload is sent. Finally, in the end of the frame payload, an optional CRC is sent.

The transmission time of a signal varies depending on the different parameters of the LoRa modulation. This time is called time on air (ToA). The ToA is the time it takes for a signal to be transmitted, so it is the time during which the channel is busy. The formula used to calculate the ToA of the packet is as follows.

$$T_{packet} = T_{preamble} + T_{payload} \quad (2.3)$$

$$T_{preamble} = (n_{preamble} + 4,25) \cdot T_{sym} \quad (2.4)$$

$$T_{sym} = \frac{2^{SF}}{BW} \quad (2.5)$$

$$T_{payload} = n_{payload} \cdot T_{sym} \quad (2.6)$$

$$n_{payload} = 8 + \max \left(\text{ceil} \left(\frac{8 \cdot PL - 4SF + 28 + 16 \cdot CRC - 20IH}{4 \cdot (SF - 2DE)} \right) \cdot (CR + 4), 0 \right) \quad (2.7)$$

Where T_{sym} indicates symbol duration in ms; PL indicates Payload size in bytes; SF indicates spreading factor; BW indicates bandwidth; CRC indicates Cyclic Redundancy Check used for error detection of LoRaWAN packet, it can be either enabled (1 - default) or disabled (0); Header ('H') can be implicit or explicit; Low Data Rate Optimize (DE) can be enabled (1) or disabled (0); CR indicates Coding Rate; $T_{preamble}$ is the preamble duration; $n_{preamble}$ is the number of symbols in the preamble; $T_{payload}$ is the payload and header duration; and $n_{payload}$ is the number of symbols in the payload period.

In this work, ToA is calculated using a tool provided by The Things Network [20], which measures ToA as a function of payload bytes, spreading factor (SF), region and bandwidth.

2.3. Media Access Control layer

In a scenario where multiple nodes try to access the physical medium simultaneously, it may cause several packets to collide, losing the information they contain. To avoid corruption or destruction of information transmitted through IoT nodes, access to the shared media in an orderly and equitably way will be managed through the Media Access Control protocols and the Logical Link Control (LLC), which constitute the 2nd layer of the Open Systems Interconnection model (OSI).

Many MAC protocols have been devised to handle access to a shared link, there are categorized into three groups, as shown in Figure 2.4:

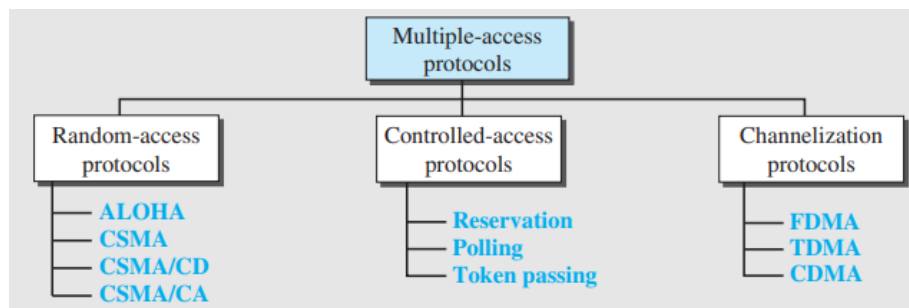


Fig. 2.4: Taxonomy of multiple-access protocols [21]

Based on the strategy adopted to distribute the channel among the nodes, they can be divided into three groups: Random Access, Controlled Access, and Channelization.

❖ Random Access

In random access or contention methods, no node is superior to others and has no control over others. Access to the medium in these types of protocols is completely random, so any node can access the media without any preference. In this group, we can find the protocols ALOHA, CSMA, CSMA/CD and CSMA/CA. The last protocols will be explained the section 2.4, and specifically the first and the last will be explained in more detail because they have been selected to perform the experiment.

❖ Controlled Access

In these protocols, a node can only transmit when it has been permitted by the other nodes of the network. The devices rotate a testimony that indicates who has permission to transmit. In this group we can find the following protocols:

- **Reservation:** in this type of method a station needs to make a reservation before sending any data. The time is divided into intervals and in each interval a reservation frame precedes the data frames sent in that interval.
- **Polling:** this type of method works with topologies in which one device is designated as a primary station and the other devices are secondary stations. All data exchanges must be made through the primary device which controls the link while the secondary devices follow its instructions.
- **Token passing:** in this type of method the nodes of a network are organized in a logical ring where there is a predecessor and a successor. The predecessor is the node that is logically before the node in the ring while the successor is the node that is after the node in the ring. The transmission permission is passed between the different nodes logically between the predecessor node, the current node, and the successor node when the current node has no more data to send.

❖ Channelization

Channelization or channel partition is a multiple-access method in which the available bandwidth of a link is shared in different ways (frequency, time, and code) among different nodes. Depending on the channel partition we can find three different protocols:

- **FDMA:** the protocol that divides the available bandwidth into frequencies is called frequency-division multiple access (FDMA), where each band is reserved for a specific node, and it belongs to the node all the time. In FDMA, the available bandwidth of the common channel is divided into bands that are separated by guard bands.
- **TDMA:** the protocol that divides the channel into time is called time-division multiple access (TDMA), where each node is allocated a time slot during which it can send data. In TDMA, the bandwidth is just one channel that is time-shared between different stations.
- **CDMA:** the protocol that divides the channel using the properties of orthogonal codes is code-division multiple access (CDMA), where each node is assigned a code and communicates with other nodes without timesharing through a unique channel that carries all transmissions simultaneously and occupies the entire bandwidth of the link.

Previously, three groups on how to share access to the physical medium along with the characteristics and protocols of each have been explained. However, when we take into account the IoT scenario and a scenario where the satellites are CubeSat, we must consider certain limitations of the performance of MAC protocols that are not present in traditional satellite communications. Some CubeSat limitations are related to the processing capabilities of the hardware and available storage. Other limitations due to the IoT scenario are a large number of devices (IoT nodes) trying to communicate with the nanosatellite or constellation of nanosatellites in motion.

First, given the hardware limitations of CubeSats, it is not possible to consider using MAC protocols that perform complex processes and saturate the hardware. The optimal and ideal for the IoT scenario is the use of a MAC protocol that performs simple processes. One of the protocols that fit these characteristics is the ALOHA-based protocols due to its simplicity in terms of implementation and the low hardware requirements [29].

Second, the IoT scenario is different from a traditional satellite communication scenario due to channel congestion. In a satellite communications scenario, the patterns are usually one-to-one or one-to-lots. In the IoT scenario, the CubeSat will behave as a gateway, so it will receive, process, and send different packets to the different IoT nodes around the Earth's surface (lots-to-one). In addition, the CubeSat does not always know the location of these nodes and the moment in which they want to communicate with it. As an added complexity, these metrics will be changing continuously due to the movement of the satellite around an orbit.

Therefore, due to the different limitations presented in the IoT scenarios, the protocols traditionally used for satellite communications based on fixed assignments cannot be considered for our scenario. That is why we must choose among the protocols that best fit an IoT scenario. The existing MAC protocols usually used for IoT satellite communications can be categorized as follows [30]:

❖ **Random access asynchronous protocols**

Random access asynchronous protocols are protocols where access to media is performed randomly and require an acknowledgment (ACK) to confirm the correct reception of the transmitted data. The four protocols that receive this categorization are the followings:

- **Aloha:** this protocol is the base of the following ones and the simplest. The network devices can always send the packets without any additional complexity added to them. If the transmitted data packet has been correctly received, the satellite responds with an ACK. In case the node does not receive any ACK, it re-transmits the packet after a random time-out.
- **Enhanced Aloha (E-Aloha):** this protocol proposes a solution to packets that are transmitted with the same periodicity with a contention window larger than the transmission time of the packets. Nodes can select randomly the time at which they transmit within that time window. With the help of this time window, nodes that have the same periodicity to send packets, vary the instant at which they transmit.

- **Spread Spectrum Aloha (SS-Aloha):** the SS-Aloha protocol uses spread-spectrum techniques to separate the channels in which each of the packets are sent and increase the amount of information sent.
- **Enhanced Spread Spectrum Aloha (E-SSA):** this protocol uses the same technique as SS-Aloha, but it also uses a Recursive Successive Interference Cancellation algorithm, thanks to it there is no need for ACK.

❖ Random access synchronized protocols

Random access synchronized protocols 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 one of these slots. One of the keys of this protocol is the synchronization among the nodes of the network and ACK to confirm the correct reception. The five protocols that receive this categorization are the followings:

- **Slotted Aloha (S-Aloha):** this protocol is like Aloha and is the base of the following ones. The medium of S-Aloha is slotted, so the devices that want to transmit must wait until one slot begins and then start the transmission.
- **Contention Resolution Diversity Slotted Aloha (CRDSA):** this protocol uses the same technique as S-Aloha and adds a Successive Interference Cancellation (SIC) mechanism in the receiver, so it can cancel interferences cancellation with the packets.
- **Irregular Repetition Slotted Aloha (IRSA):** this protocol has many similarities with protocol CRDSA; however, protocol IRSA has multiple transmissions of the packet.
- **Coded Slotted Aloha (CSA):** in this protocol, the packets are divided into sub-packets of the same length which include error correction codes that allow an ACK to not be needed. Then, the receiver applies a maximum-a-posteriori (MAP) decoder, to be able to recover subpackets that are lost. Additionally, the receiver also implements an interference cancellation scheme to receive from multiple senders.
- **Multi-slots Coded Aloha (MuSCA):** this protocol implements a 1/4 Turbo code as an error correction code that does not need ACK.

❖ Medium sensing protocols

Medium sensing protocols are protocols where the nodes sense the medium before transmitting. In the case where the medium is busy, it performs a random back-off and senses the medium again. Otherwise, if the medium is available, the packet is transmitted. Only Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) is in this category. This protocol will be explained in the following sections.

❖ Reservation protocols

Reservation protocols divide the medium into different slots and reserve certain slots of the medium for certain nodes. In these protocols, nodes must be aware of which slots are reserved and which ones are free. Also, this protocol requires precise time synchronization. Only R-Aloha is in this category:

- **R-Aloha:** this protocol defines frames, which are further divided into several slots. Nodes can transmit randomly in any of these slots within a frame. If an ACK is received, the slot is reserved for the node due to the success of the communication.

❖ Hybrid protocols

Hybrid protocols are a mix of different protocols that cannot be classified in previous categories. The two protocols that receive this categorization are the followings:

- **Fixed Competitive Time Division Multiple Access (FC-TDMA):** in this protocol, the channel is divided into frames and each of these frames contains a configurable number of slots. The satellite predicts how many slots are necessary for the next communication based on the collisions that occurred in the previous one.
- **Random Frequency Time Division Multiple Access (FTDMA):** in this protocol packets are transmitted with a random carrier frequency within a range. They require ACK to confirm the correct reception of the packet.

Previously, the five categories in which MAC protocols oriented to satellite communications in IoT scenarios can be divided have been explained. All these protocols consider a high density of nodes around the Earth's surface as is the case presented for this project [21]. Among all the possible protocols presented, only two have been chosen to be implemented and tested in the LoRa communications experiment proof-of-concept. Pure Aloha and CSMA/CA have been the protocols selected because they trade-off between complexity and performance. Both protocols have been explained in detail in the following section.

2.4. Random-Access protocols

In random-access or contention methods, no station is superior to another station, and none is assigned control over another. In each instance, a station that has data to send uses a procedure defined by the protocol to decide on whether to

send or not. This decision depends on the state of the medium after sensing it. The medium can be busy (if there is another communication in the process) or idle (if the medium is free).

In random-access protocols, there is no scheduled time for a station to transmit and the transmissions are random among the stations. Another feature of this protocol is that no rules specify which station should send next, so each station has the right to the medium without being controlled by any other station. However, if more than one station tries to send a packet there will be an access conflict, which will cause a collision destroying or modifying the information. To avoid an access conflict each station follows a defined protocol which dictates the steps to follow to avoid this collision or what to do if it has it.

The random-access methods below are the protocols used for the LoRa communications experiment proof-of-concept. The first to be explained in detail will be the oldest of all, pure Aloha. This method uses a procedure called multiple access (MA). This method was improved by adding a procedure that forced the station to sense the medium before transmitting. This is how the Carrier Sense Multiple Access (CSMA) method came about. This method later evolved into two parallel methods: *carrier sense multiple access with collision detection* (CSMA/CD), which tells the station what to do when a collision is detected, and *carrier sense multiple access with collision avoidance* (CSMA/CA), which tries to avoid the collision. This second variant will be explained in detail in the following sections.

2.4.1. Pure ALOHA

Pure ALOHA is the earliest random-access method, was developed at the University of Hawaii in early 1970. The original ALOHA protocol is called pure ALOHA and is the simplest of the MAC protocols. In pure ALOHA when a node has a packet to send access the media and sends the information without any restriction. However, since there is only one channel to share, there is the possibility of collision between frames from different stations.

To minimize the probability of collision, pure ALOHA proposes an algorithm based on the retransmission and use of an extra confirmation packet, this packet is the Acknowledgment (ACK). This packet confirms the correct reception of the packet information sent by the node that must transmit information to the receiving station. In the case of our study, the satellite. When the transmitting node sends a packet, it then starts a counter known as "*Wait Time*". Wait time is a timer that ends after a predetermined time. During this time, the node that has transmitted the information packet is waiting to receive the ACK, thus confirming that the transmitted packet has been received correctly. If the ACK is received before this time reaches zero it would be considered a success. If the ACK does not arrive after a time-out period, the station assumes that the frame (or the acknowledgment) has been destroyed and resends the packet. The waiting time is calculated as the maximum possible round-trip propagation delay, which is

twice the amount of time required to send a frame between the two most widely separated stations. Is calculates as follows (2.8):

$$T_{wait} = 2 \cdot T_p \quad (2.8)$$

The wait time calculation is directly related to the vulnerable time calculation, which is the length of time in which there is a possibility of collision. For this calculation, we assume that nodes send fixed-length frames with each frame taking T_{fr} seconds to send. In the figure 2.5, we can find a graphic representation of vulnerable time.

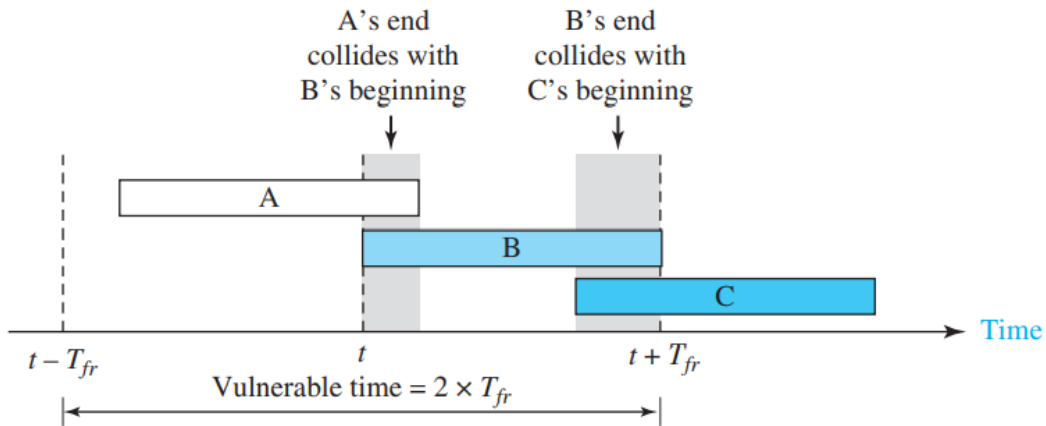


Fig. 2.5: Vulnerable time for pure ALOHA protocol [21]

As we can see in Figure 2.5, there are three fictitious nodes (A, B and C) that transmit information during a given and equal frame time for all (T_{fr}). If node A transmits after $t - T_{fr}$ there will be a collision with the transmitted packet of node B and both packets will be modified or destroyed. The same happens if node C transmits before $t + T_{fr}$. Therefore, the vulnerable time during which a collision may occur in pure ALOHA is two times the frame transmission time. Since in our scenario each package has a different frame time due to its size, we will calculate the wait time as twice the maximum propagation time.

Since a collision is due to two or more stations trying to transmit at the same time, it would not make sense for them to re-transmit at once. This would again cause the same collision uninterruptedly. To solve this problem, pure ALOHA sets a random time where the node must wait before re-transmitting its frame. This random time is known as backoff time (T_B) and this randomness will help avoid more collisions. The backoff time (T_B) is a random value that normally depends on K (the number of attempted unsuccessful transmissions). The determination of backoff time depends on the implementation, for this project it is 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 retransmission attempts) multiplied by the maximum propagation time (T_p).

Pure ALOHA has a second method to prevent congesting the channel with retransmitted frames. If after the packet has been retransmitted a certain number of times (K_{max}) the ACK has not been received, the communication is given as failed and the node stops trying to transmit the packet to try later.

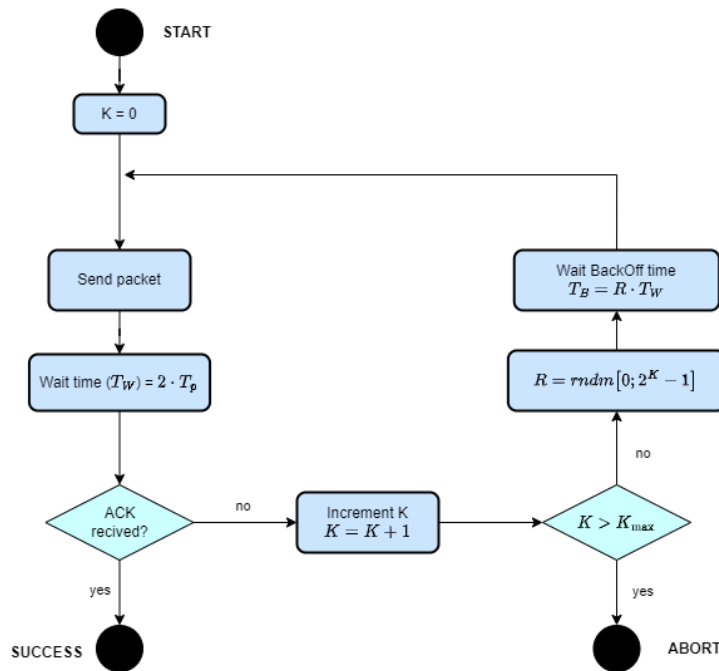


Fig. 2.6: Procedure for pure ALOHA protocol

In the Figure 2.6 we can find the procedure for pure ALOHA protocol, where:

- K : the number of attempted unsuccessful transmissions
- K_{max} : maximum number of retransmission attempts
- T_p : Maximum propagation time
- $T_W(\text{wait time}) = 2 \cdot T_p$
- $T_B(\text{Backoff time}) = R \cdot T_p$
- $R(\text{Random number}) = [0, 2^K - 1]$

2.4.2. Carrier Sense Multiple Access

Carrier Sense Multiple Access (CSMA) is a MAC protocol that improves the performance of ALOHA protocol by adding the limitation of sensing the channel to know if it is busy or idle before transmitting. The chance of collision can be reduced if a station senses the medium before trying to use it. That's why CSMA is based on the "listen before talk" principle.

Despite the addition of this new strategy, a collision-free channel is not possible. This is due to propagation delay. When a node sends a frame there is a certain time until the first bit of the packet reaches another node and senses it. In other words, there may be the casuistry that a node senses the medium as idle when there is actually another node that has already transmitted the frame, nevertheless that frame has not yet reached the node. In the figure 2.7 we can see a representation of this casuistry:

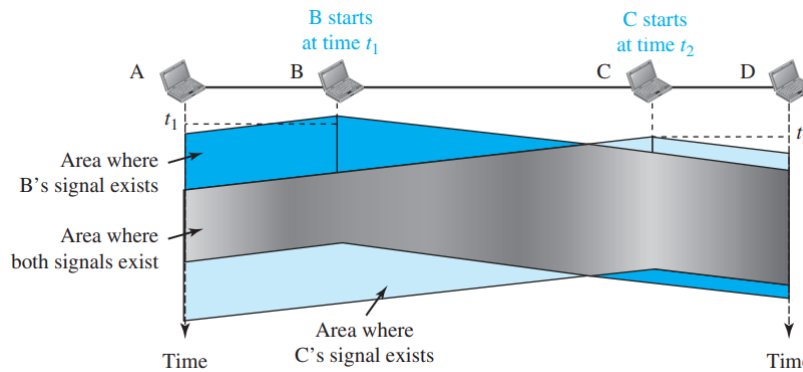


Fig. 2.7: Space/time model of a collision in CSMA [21]

As we can see in the figure 2.7, both the frame of node B and the frame of node C collide and both packets are destroyed. This has happened given that node B has transmitted in the instant t_1 when it has sensed the medium as free. However, node C in instant t_2 has also sensed the free medium when it is not because the frame of node B has not yet reached node C due to the propagation delay.

Vulnerable time

Therefore, the vulnerable time in CSMA is conditioned by the propagation time (T_p). This is the time needed for a signal to propagate from one end of the medium to the other. When a station sends a frame and any other station tries to send a frame during this time, a collision will occur. The figure 2.8 show this scenario:

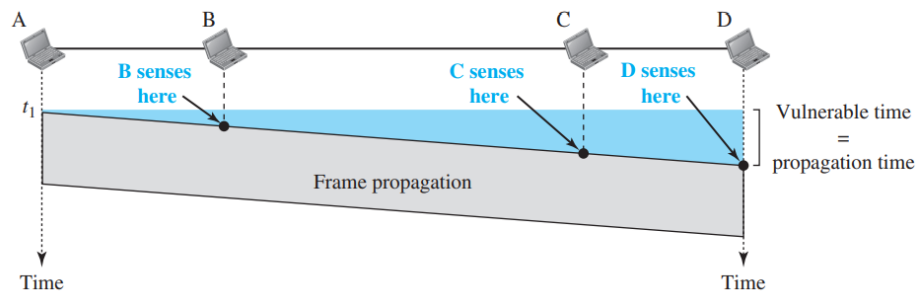


Fig. 2.8: Vulnerable time in CSMA [21]

In the above image the worst case is proposed, where a node receives the frame of the farthest node. The leftmost node (A) sends a frame at time t_1 , which reaches the rightmost node (D) at time $t_1 + T_p$. The gray area shows the vulnerable area in time and space.

Persistence Methods

The implementation of a new sensing technique in CSMA protocols is an elegant technique that prevents collisions. In this subsection, the different methods used to sense the channel and the method used in our case will be explained.

In total there are three different methods to sense the channel. Each uses a different technique and defines what to do in case the channel is idle or busy. These three methods are the *1-persistent* method, the *nonpersistent* method, and the *p-persistent* method.

- 1-Persistent:** in this method, after the station finds the medium idle, it sends its frame immediately. This method has the highest chance of collision because two or more stations may find the line idle and send their frames at the same time. In the figure 2.9 we can find a visual representation of the method and its flow diagram.

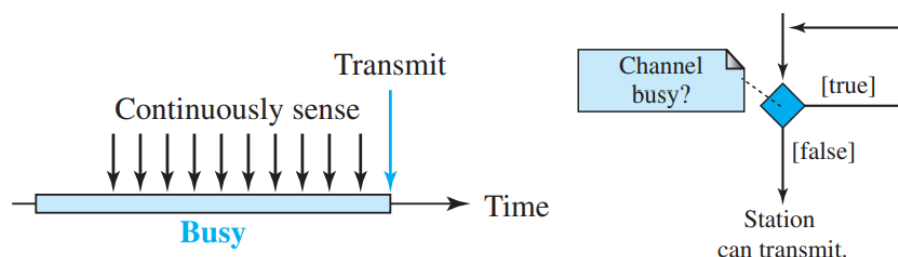


Fig. 2.9: Behavior of 1-persistent method [21]

- Nonpersistent:** in the nonpersistent method, a node that has a frame to send senses the line. If by sensing the medium it finds the free channel, then it will send the package immediately. However, if the channel is not free it will wait a random time to re-sense the channel and determine its

status. A positive point of this method is the reduction of collision possibilities. This is because it is unlikely that two nodes will wait the same amount of time after the channel is busy and retransmit at the same time. However, a negative point has the reduction of the efficiency of the network. This is because at certain times the channel remains idle when there are nodes that have packets to send.

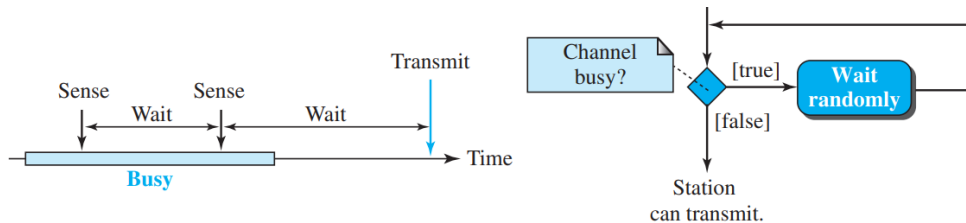


Fig. 2.10: Behavior of nonpersistent method [21]

- p-Persistent:** this method is used if the channel has time slots with a slot duration equal to or greater than the maximum propagation time. The p-persistent approach combines the advantages of the other two methods. It reduces the chance of collision and improves efficiency. After the station finds the line idle, with probability p , the station sends its frame. With probability $q = 1 - p$, the station waits for the beginning of the next time slot and checks the state of the medium again. In case the line is idle, it goes to step 1. However, if the line is busy, it acts as though a collision has occurred and uses the backoff procedure.

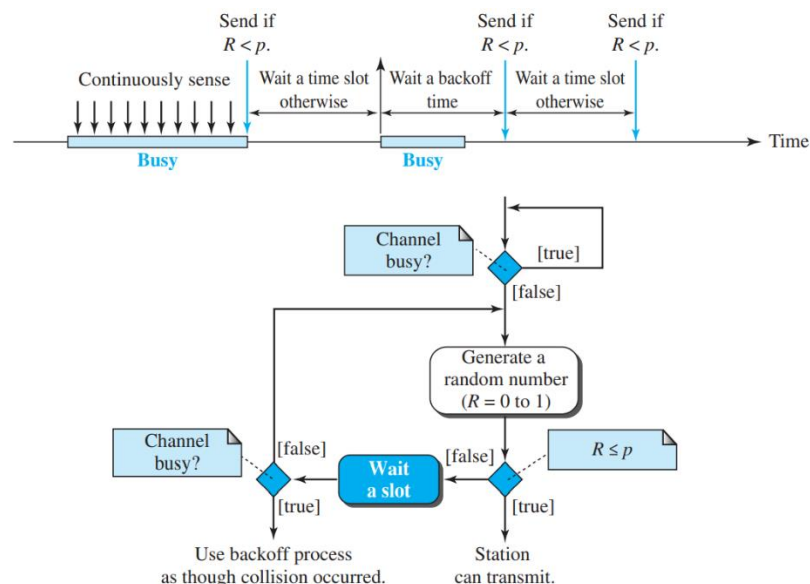


Fig. 2.11: Behavior of p-persistent method [21]

2.4.2.1. Carrier Sense Multiple Access with Collision Avoidance

Carrier sense multiple access with collision avoidance (CSMA/CA) relies on two extra packets: Request To Send (RTS) and Clear To Send (CTS). Moreover, two interframe wait are established to solve the vulnerable time: DFC InterFrame Space (DIFS) and Short InterFrame Space (SIFS).

Before explaining in detail the algorithm of the CSMA/CA protocol, its different and new components will be explained: RTS, CTS, SIFS and DIFS.

- **Request To Send (RTS):** it is a control packet which requests access to the medium in order to start transmission. Once this packet is sent, the node waits for the CTS. The information contained in the RTS packet is the identifier of the node and the duration the channel will be occupied, known as Network Allocation Vector (NAV).
- **Clear To Send (CTS):** is a control packet which is sent by the receiving station (the satellite in our case) after having received an RTS from a node. When the node receives the CTS, the channel is reserved, and the transmission of useful information begins. The information contained in the CTS packet is the identifier of the node who sent the RTS and the duration the channel will be occupied (NAV).
- **Short InterFrame Space (SIFS):** is set by the maximum delay of a transmitted packet to reach the most distant node and it is performed before the transmission of each packet once started the process.
- **DFC InterFrame Space (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.

The flow chart describing the CSMA/CA algorithm can be seen in Figure 2.12:

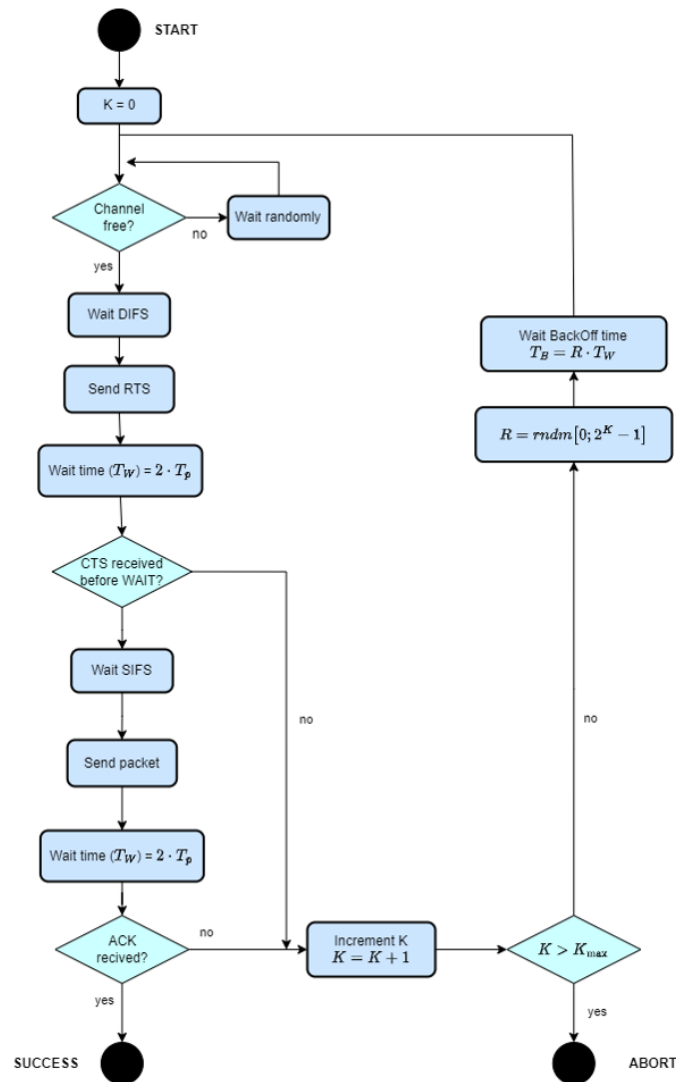


Fig. 2.12: Procedure for CSMA/CA protocol

At the beginning of the transmission, the node senses the channel to know its status. If the channel is busy, it will sense the medium again once a random time has passed. If while sensing the channel the node receives an RTS or CTS, it must wait a certain time specified in the NAV of each packet. If instead, the channel is idle, then wait a DIFS time to avoid possible collisions within the vulnerability time. Once the DIFS time has passed, the node transmits an RTS, and the timeout begins.

The RTS packet sent by the node is received at the receiving station (the satellite in our case) if there has been no collision. The satellite waits for a SIFS time and then transmits a CTS which contains the node identifier. If the CTS packet is received by the node before the wait time ends, it means the reservation of the channel is correctly achieved and the communication process can start. If it is not, the transmitting node initiates a backoff process already explained at pure

ALOHA. Once the backoff process is done, the node returns to sense the medium again.

Therefore, if the CTS has been received correctly, the data packet is transmitted after waiting for a SIFS time and a waiting time starts again. When the satellite receives the data packet, it waits for a SIFS and sends the ACK which contains the identifier of the node with which it is communicating. If the node receives the ACK before the wait time is finished, the communication is given as successful. However, if the ACK is not received after the wait time, a reception attempt must be added and the backoff process initiated. If the number of receiving attempts is greater than the allowed number (K_{max}), the communication is considered a failure and the node stops trying.

Network Allocation Vector

The network allocation vector (NAV) is an essential time of this protocol to avoid collisions. The NAV is a time that determines how much time the channel will be occupied by the node that is communicating with the reception station. The NAV time is within the content of the RTS and CTS packets, being the largest NAV time for the RTS.

During the communication process in CSMA/CA, nodes exchange different type of packets. The RTS is the first of them, which is sent once the channel has been sensed and detected as free. The other nodes that are sensing the channel detect this RTS packet and make the reading of its NAV. This information allows them to identify how long the channel will be occupied, so they are disabled until the NAV counter has expired. The same happens when a CTS is received while the nodes sense the channel. These will detect the NAV containing the package and wait a stipulated time until the channel is sensed again, where it should supposedly be idle given that the previous communication process has finished.

The way to calculate NAV times are defined by the following equations (2.9 and 2.10):

$$NAV(RTS) = 3 \cdot SIFS + T_{p_{CTS}} + T_{p_{DP}} + T_{p_{ACK}} \quad (2.9)$$

$$NAV(CTS) = 2 \cdot SIFS + T_{p_{DP}} + T_{p_{ACK}} \quad (2.10)$$

Where:

- $T_{p_{CTS}}$: Clear To Send propagation time
- $T_{p_{DP}}$: Data Packet propagation time
- $T_{p_{ACK}}$: ACK propagation time

The justification for the above formulations is drawn from the figure 2.13, where the necessary time-out of the NAV RTS and the NAV CTS can be observed [22]:

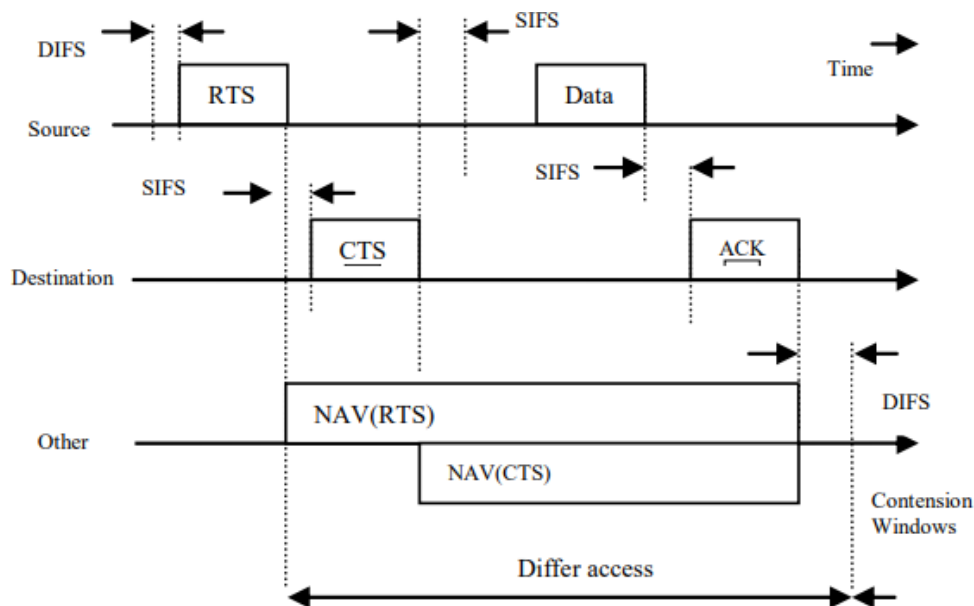


Fig. 2.13: RTS/CTS Communication with NAV [22]

Collision During Handshaking

This phenomenon called "collision during handshaking" occurs when there is a collision during the sending of RTS or CTS. Two nodes may detect the medium as idle and transmit at the same time. If this happens, these two packets will likely collide. However, since no mechanism in CSMA/CA detects collisions, the transmitter will assume that there has been a collision if after a wait time it has not received the CTS from the receiving station. In this case, the backoff process starts to retransmit the packet again if it detects the channel free.

When the receiving station sends a CTS, it sends the confirmation that the node can start to transmit since the channel has been reserved for this communication. If after a certain time the receiving station does not receive the data packet of the node it listens again new RTS of other nodes to send new CTS and reserve the channel to new communications.

Hidden-Station Problem

One of the reasons why NAV is found in both the RTS and CTS packages is because of the Hidden-Station Problem. This casuistry can be seen in Figure 2.14, where the frame exchange timeline of the communication between different nodes and the receiving station is shown.

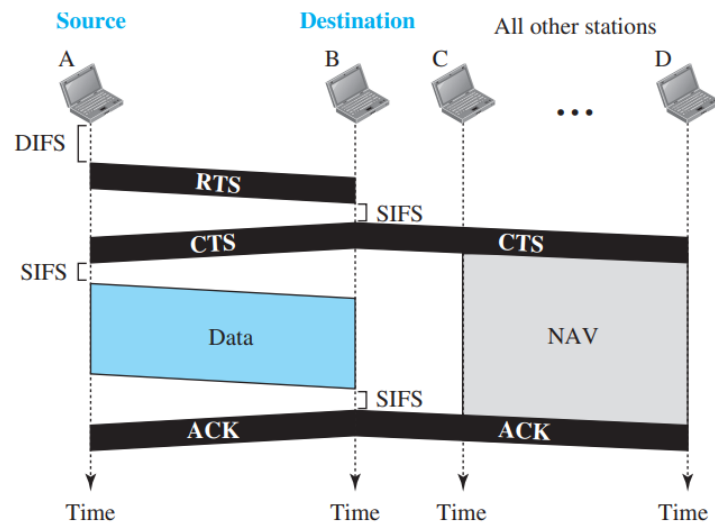


Fig. 2.14: CSMA/CA and NAV

As can be seen, the RTS sent by node A is received by the receiving station (B) but not by the stations farther away from A (C, D, etc.). However, the receiving station (B) is within reach of the other nodes, and therefore the CTS packet sent to node A is received by the other nodes. In this way, nodes that are out of reach of this node are aware that communication is in process and starts a timeout defined by the NAV of the CTS.

In IoT satellite communications often encounter this problem. This is because the nodes are located around the Earth's surface and not all are within reach of each other. But the satellite is within range of all nodes. Then, when the satellite sends the CTS the rest of nodes are aware that a communication is in process.

CHAPTER 3: Applied methodology for software and hardware development

In this chapter, the methodology applied to perform the experiment will be explained. This chapter is divided into three parts where the general architecture of the experiment is first explained, followed by the software-related part, and finally the hardware-related part.

Section 3.1 provides an overview of the experiment where the equipment used is introduced and how they work simultaneously using the implemented software.

Section 3.2 explains the implementation of the pure ALOHA and CSMA/CA protocols. The control method using commands to manage the different experiments will be explained. Finally, the different adjustable parameters of the protocols will be calculated.

Finally, the section 3.3 explains the hardware used as well as the methodology followed to make modifications, calibrations, and connections that the hardware requires to perform the proof-of-concept.

3.1. General architecture of the LoRa communications proof-of-concept experiment

To perform the LoRa communications proof-of-concept experiment has required the use of several devices working simultaneously, as well as a correct implementation of the protocols to be tested in them. The main outline of the communication between the different devices can be seen in Figure 3.1.

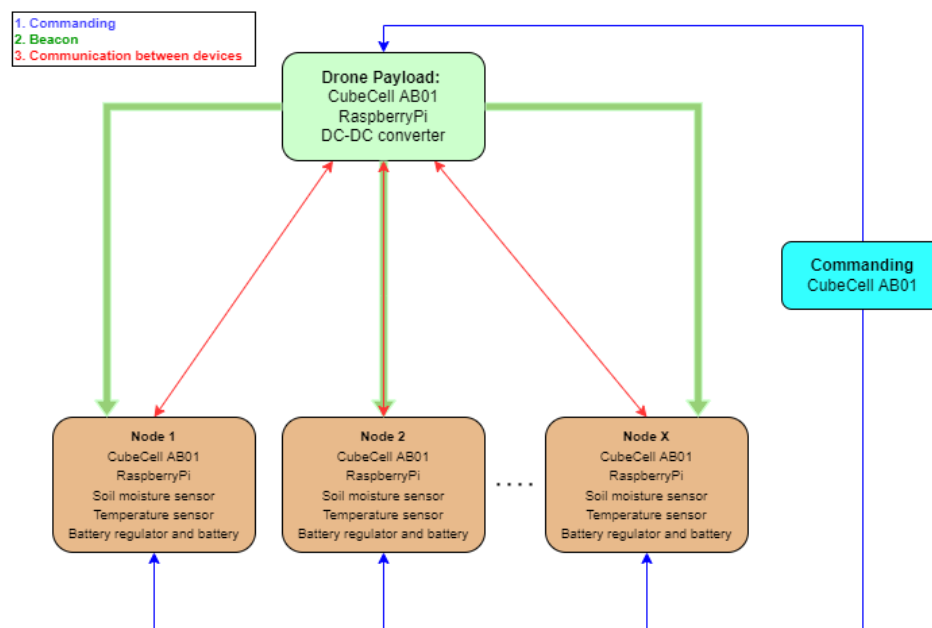


Fig. 3.1: Main outline of the communication between the different devices

First, performing the proof of concept required various IoT ground nodes and a drone-based payload that functioned as a receiving station.

The IoT ground nodes are formed by the connection between various devices, being the heart of all of them the HelTec CubeCell Dev-Board HTCC-AB01 transceiver. The other devices forming the ground nodes are the moisture sensor (Soil Moisture sensor v1.2); the temperature sensor (HDC1080); a Raspberry Pi which is used to read the data received from the CubeCell through the UART; and finally, a regulator next to a lithium battery to power the different equipment. In addition, it was necessary to add an integrated stripboard to the CubeCell for ADC dynamic range adjustment. To take advantage of the space and the direct connection to the CubeCell, the HDC1080 has been inserted into the stripboard. A total of 20 nodes were planned for the measurement campaign, however due to hardware limitations 13 IoT nodes were finally used. The equipment used is shown below in the Figure 3.2.

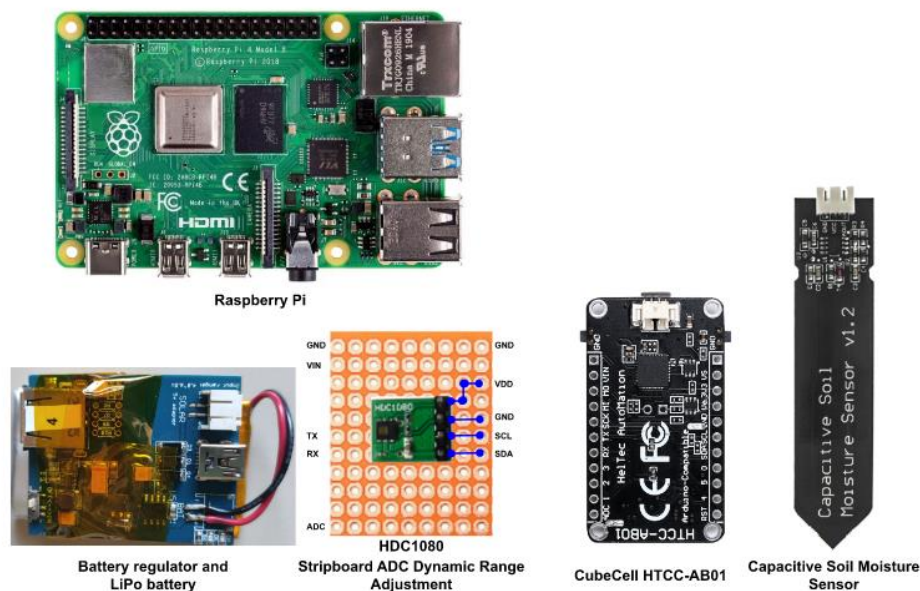


Fig. 3.2: Devices used in ground nodes.

The payload that will be assembled in the drone is formed by various devices, the CubeCell being again the main equipment. In the payload of the drone, there is also a Raspberry Pi which is responsible for saving the data captured through the UART, and a DC-DC converter which feeds the Raspberry Pi through the voltage provided by the drone batteries using a XT60 cable. It should be noted that the design of the drone-based payload for the LoRa communications proof-of-concept experiment has been made to be used simultaneously with the GNSS-R experiment. Therefore, the payload that we will see in the following sections brings together different equipment for both experiments. Figure 3.3 shows the equipment used for the LoRa communications proof-of-concept experiment.

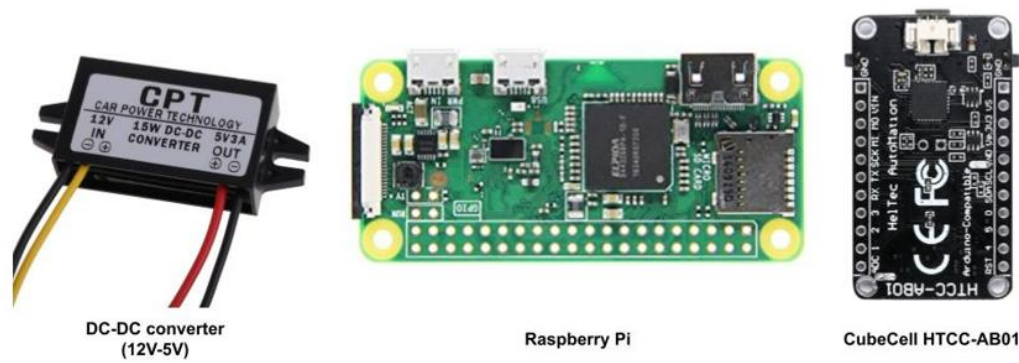


Fig. 3.3: Devices used in miniaturized drone-based payload

Secondly, both the software implemented at IoT ground nodes and the implemented in the drone has been carried out through CubeCells. These transceivers can communicate using LoRa and are compatible with Arduino, so all the code for both protocols has been programmed using C++. For the experiment, it was necessary to program the algorithm followed by the pure ALOHA and CSMA/CA protocols at the ground nodes and at the receiving station. For both cases, a single code has been flashed on the CubeCell, so both protocols are in the same code. The choice of which protocol to use along with its different features is controlled through a third CubeCell which sends a command to the ground nodes and the drone payload. This command marks the start of a new experiment using the currently chosen protocol along with different protocol attributes. Both the ground nodes and the payload of the drone are initiated by listening to the medium, once they receive the command of which protocol to execute the communications begin. Given the nature of the IoT satellite communications scenario, the principle of all communications begins with the transmission of a beacon from the drone to the rest of the ground nodes, which will be listening the medium until it is received.

3.2. Methodology applied in the software design

One of the key points of the experiment has been the correct implementation of MAC protocols. In the first section, the working environment will be explained in addition to the facilities it offers. In the second section, we will explain in detail the steps taken to implement the code in both the ground nodes and the CubeCell of the drone payload. Finally, the third section will explain how the different adjustable parameters of both protocols have been calculated.

3.2.1. Arduino Software IDE

As explained in previous sections, the CubeCell HTCC-AB01 transceiver is a suitable module for LoRa/LoRaWAN node applications. This module is also perfectly compatible with Arduino, so it can be programmed from the Arduino IDE in a language similar to C++.

In addition to downloading the Arduino IDE, it is also necessary to download the SiLabs CP2104 Driver to establish a serial connection between the computer and the CubeCell board [23]. Once you have installed the Arduino IDE, you need to finish configuring certain additional preferences. In the section "Additional Boards Manager URLs" it is necessary to enter the three JSON files for the CubeCell and the Arduino IDE to operate correctly [23].

Once the working environment has been adjusted, we see below the facilities provided by the downloaded libraries on which we base the software of the experiment. Since the Arduino IDE has been compatible with CubeCell, several examples have been downloaded which can be loaded on the HTCC-AB01 board. Among these examples, we find a section related to LoRa communications. These basic LoRa examples include: *LoRaReceiver*, *LoRaSender*, *LoRaSender_ReadBattery*, *pingpong* and *TxPowerTest*.

Considering the scenario of the experiment, where the nodes and the satellite perform both sender and receiver actions, the communication architecture that best fits our case is a "*pingpong*" architecture. Within this architecture are various functions to perform reception and transmission functions using LoRa. Within this example, the software of the experiment has been developed.

In the architecture of the example "*pingpong*" the first thing we find is the adjustable parameters of the LoRa physical layer. See Figure 3.4:

```
#define RF_FREQUENCY                868000000 // Hz
#define TX_OUTPUT_POWER            0          // dBm
#define LORA_BANDWIDTH              0          // [0: 125 kHz,
                                              // 1: 250 kHz,
                                              // 2: 500 kHz,
                                              // 3: Reserved]
#define LORA_SPREADING_FACTOR      8          // [SF7..SF12]
#define LORA_CODINGRATE             1          // [1: 4/5,
                                              // 2: 4/6,
                                              // 3: 4/7,
                                              // 4: 4/8]
```

Fig. 3.4: Adjustable parameters of the LoRa physical layer

Among the 5 most important adjustable parameters we have:

- ***RF_FREQUENCY*:** The frequency is determined by the region and by the working frequency of the module to be used. In the case of this work, the module used is the Semtech SX1261. It operates between the ISM bands allowed in Europe, which are 433 MHz and 868 MHz. Particularly, in this work, it operates at 868 MHz.
- ***TX_OUTPUT_POWER*:** The maximum transmission power can be set to 14 dBm at 868 MHz and 22 dBm at 915 MHz, these values are the maximum according to the data sheet of the Semtech SX1261 modules,

which is used in the HelTec CubeCell Dev-Board HTCC-AB01 transceiver of this project. Nevertheless, the chosen transmission power is determined by the working environment in which we are. In our case, the experiment is carried out in an environment where the distances between nodes do not exceed 800 meters. So, it has been decided to adjust the transmission power to 0 dBm.

- **LORA_BANDWIDTH:** The bandwidth in LoRa modulation is configurable between different values, the most typical being 125 kHz, 250 kHz and 500 kHz. According to the Semtech SX1261 module datasheet, this parameter determines the maximum center frequency offset that the modules are capable of compensating. The modules can compensate up to 25% of the BW, so the larger the BW the greater the Doppler frequency shift compensation. However, the larger the BW the higher the noise power. In [16] it is determined that the best BW that compensates the Doppler effect in LEO orbits, reduces the noise power and reduces the transmission time is 125 KHz.
- **LORA_SPREADING_FACTOR:** To improve bit rate and capacity, it is preferable to use a small SF. In [16] a link budget analysis is performed using different attributes of the LoRa. In this study it is determined that with the use of the radiofrequency Front End of the RITA payload, the lowest SF to have communications at practically any LEO orbit elevation is a SF of 8.
- **LORA_CODINGRATE:** The CR is also configurable, and can be set to 4/5, 4/6, 4/7 or 4/8, having 1, 2, 3 or 4 bytes of redundancy respectively. In [16] it is determined that the best CR in terms of capacity corresponds to a CR of 4/5.

After adjusting the different LoRa parameters, the four callback functions belonging to the *RadioEvents* constructor are presented. These functions are shown in the Figure 3.5:

```
static RadioEvents_t RadioEvents;
void OnTxDone( void );
void OnTxTimeout( void );
void OnRxDone( uint8_t *payload, uint16_t size, int16_t rssi, int8_t snr );
void OnRxTimeout(void);
```

Fig. 3.5: Callback functions

In the first instance the creation of the Driver for the SX1272 RF Transceiver is shown. Below, there are the callback functions (defined in the “*radio.h*” file) which have the following functionalities.

- **Void OnTxDone (void):** is the function in which the program is directed after any packet sent.
- **Void OnTxTimeout (void):** is the function to which the program is directed if there is no connection between the micro-controller and the radio.
- **Void OnRxDone (...):** is the function in which the program is directed when a packet is received. As we can see in figure 3.4, the function initializes various variables which will be used later. First there is the *uint8_t *payload*, which saves the information of the captured packet at the reception. Second, the *uint16_t size*, which determines the number of bytes of the package received. Third, the *int16_t rssi*, which shows us the received signal strength. Finally, the *int8_t snr*, which indicates the signal to noise ratio.
- **Void OnRxTimeout (void):** It is the function to which the program is directed if it has not detected any packets after a certain time of reception.

Apart from these four functions, two other basic and mandatory functions of any Arduino program are used. These are the "void setup()" and "void loop()" functions. With the help of the *void loop()* and *void setup()* functions in our sketch, we give the instructions to the Arduino microcontroller. Everything inside the configuration "void setup()" will run once. The contents of the "void loop()" will run in the loop while the Arduino controller remains on.

- **Void setup():** the first function is the first to be executed and initializes the program. First, there is the baud with which a serial connection is established between the PC and the Arduino (*Serial.begin(115200)*). In addition, various functions of *RadioEvents* are also declared and the Radio is configured with the various variables of LoRa.

```
void setup() {
    Serial.begin(115200);
    Rssi=0;
    RadioEvents.TxDone = OnTxDone;
    RadioEvents.TxTimeout = OnTxTimeout;
    RadioEvents.RxDone = OnRxDone;
    RadioEvents.RxTimeout = OnRxTimeout;

    Radio.Init( &RadioEvents );
    Radio.SetChannel( RF_FREQUENCY );
    Radio.SetTxConfig( MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                      LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                      LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                      true, 0, 0, LORA_IQ_INVERSION_ON, 3000 );

    Radio.SetRxConfig( MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                      LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                      LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                      0, true, 0, 0, LORA_IQ_INVERSION_ON, true );
}
```

Fig. 3.6: "void setup" configuration

- ***Void loop()***: this function loops the program. In the next section the details of the algorithm programmed to run in a loop will be explained.

During all communications, having a time reference is important, either to compare different timestamps or to control the time elapsed since an event. To do this, Arduino has its own function (*millis()*) which returns the number of milliseconds passed since the Arduino board began running the program.

3.2.2. General design of the code developed for the experiment

In this section, it is first explained how the experiment is controlled through commands. Subsequently, the general design of the code is analyzed where it is seen how it is structured in the Arduino IDE. Subsequently, the design and implementation of the pure ALOHA algorithm is detailed along with the different types of packets that are part of the protocol. Finally, the design and implementation of the CSMA/CA algorithm with its different packets and characteristics are explained.

3.2.2.1. Control commands for the experiment

In the scenario of the experiment, we encountered a situation in which we want to test the performance of the pure ALOHA and CSMA/CA protocols with different IoT ground nodes and a receiving station which is in the drone payload. To control both sides, both the ground nodes and the drone, it has been necessary to program a third code that sends a command to decide the type of experiment to be performed. Figure 3.1. shows the communication scheme that is carried out to start any experiment.

When all the devices are powered, both the drone payload and all the IoT ground nodes remain in listening mode, waiting to receive the command that tells them which protocol to execute and with which parameters. The different types of packets are defined as a union between an array of *uint8_t*, and a struct containing an attribute for each of the packet fields. These are organized in such a way that alignment problems are avoided, and useful information is separated. The structures of the pure ALOHA and CSMA/CA command packets will be explained below.

In the following image, we can see the structure of the packet that is transmitted to activate the pure ALOHA protocol. The first component is the *flag (uint16_t)* of the package so that the devices know how to identify the type of package they are receiving. *TimeNextPacket (uint16_t)* is used to determine the rate of packet submission. *ExperimentTime (uint32_t)* is used to determine the duration of the experiment. The *WAIT_TIME_ALOHA (uint16_t)* and *TRY_ACK_MAX (uint16_t)* are adjustable parameters of the pure ALOHA protocol itself. The

WAIT_TIME_ALOHA determines the waiting time to receive an ACK after sending the Data Packet. The *TRY_ACK_MAX* is used to determine the maximum number of retries to receive the ACK. Finally, *T_BEACON* (*uint32_t*) is used to determine the sending rate of the beacon. This information reaches both the ground nodes and the drone, which processes the packet and readjusts its variables. Below is how the structure is filled with the different attributes. It is then sent using the "*Radio.Send*" function, which sends the full *byteArray* of the package. The numerical value of the other paragraphs will be explained later in section 3.2.3.

```
//LORA COMMAND ALOHA
typedef struct Lora_CommandA_Struct{
    uint16_t flag;           //5
    uint16_t TimeNextPacket;
    uint32_t ExperimentTime;
    uint16_t WAIT_TIME_ALOHA;
    uint16_t TRY_ACK_MAX;
    uint32_t T_BEACON;
};
typedef union Lora_CommandA{
    Lora_CommandA_Struct Lora_ComA;
    uint8_t byteArray[16];
};
Lora_CommandA ComA;

ComA.Lora_ComA.flag = 5;
ComA.Lora_ComA.ExperimentTime = 300000;
ComA.Lora_ComA.TimeNextPacket = 5000;
ComA.Lora_ComA.WAIT_TIME_ALOHA = 352;
ComA.Lora_ComA.TRY_ACK_MAX = 5;
ComA.Lora_ComA.T_BEACON = 80000;
//Command Send
Serial.println("CC1;Comand ALOHA sended;");
turnOnRGB(COLOR_SEND,0);
Radio.Send((uint8_t *)ComA.byteArray, sizeof(ComA.byteArray));
turnOnRGB(0,0);
```

Fig. 3.7: Pure ALOHA command architecture and values of the different variables and sending structure of the package

The CSMA/CA command maintains the same structure. However, it occupies a total of 24 bytes since it includes 6 adjustable parameters of the CSMA/CA protocol in addition to the *flag*, *TimeNextPacket*, *ExperimentTime* and *T_BEACON*, which are basic in both command packages.

```
//LORA COMMAND CSMA_CA
typedef struct Lora_CommandC_Struct{
    uint16_t flag;           //6
    uint16_t TimeNextPacket;
    uint32_t ExperimentTime;
    uint16_t TRY_ACK_MAX;
    uint16_t SENS_TIME;
    uint16_t WAIT_TIME;
    uint16_t T_SIFS;
    uint16_t T_NAV_RTS;
    uint16_t T_NAV_CTS;
    uint32_t T_BEACON;
};
typedef union Lora_CommandC{
    Lora_CommandC_Struct Lora_ComC;
    uint8_t byteArray[24];
};
Lora_CommandC ComC;
```

Fig. 3.8: CSMA/CA command structure

Once the command is received by the different devices, the execution of the protocol begins and the exchange of packets between devices does not stop until the time of the given experiment elapses.

3.2.2.2. Overall code design

In this section, the types of packets used in both protocols will be detailed. Subsequently, an overview of the code implemented in both the CubeCell of the ground nodes and the payload of the drone will be observed.

In the previous section, the type of structure that packages follow has been detailed. These are formed by the union between an array of *uint8_t* and a *struct* containing the attributes of each of the packets. These are organized in a structured way to avoid alignment problems. Below are introduced the different types of packages used with the attributes of each.

- **Beacon:** this packet type is used in both protocols. It has a dimension of 8 bytes and consists of the following attributes. The *flag*, which determines the type of packet. The *satellite_id*, which determines the identifier of the satellite. The *timestamp*, which indicates the relative time.

```
typedef struct Lora_Beacon_Struct{
    uint16_t flag;           //2
    uint16_t satellite_id;
    uint32_t timestamp;
};
typedef union Lora_Beacon{
    Lora_Beacon_Struct Lora_BS;
    uint8_t byteArray[8];
};
Lora_Beacon LoraB;
```

Fig. 3.9: Beacon packet structure

- **Data Packet:** this type of packet is used in both protocols. It has a size of 30 bytes and consists of the following attributes. The *flag*, which determines the type of packet it is. The *satellite_id*, which is obtained once the beacon is received from the satellite. The *packet_type*, which is 0 if it is a pure ALOHA protocol packet, or 1 if it is from the CSMA/CA protocol. The *node_id*, which determines the identification of the node. The *packet_id*, which determines the number of packets sent. The *timestamp*, which indicates the relative time of the node. The *pos_x*, *pos_y*, and *pos_z*, which determine the position of the node. The *temperature* and *soilmoisture* attributes are the values obtained from the measurements of the sensors.

```

typedef struct Lora_Packet_Struct{
    uint8_t flag;                //0
    uint8_t satellite_id;
    uint8_t packet_type;        //0 or 1
    uint8_t node_id;
    uint32_t packet_id;
    uint64_t timestamp;
    uint32_t pos_x;
    uint32_t pos_y;
    uint16_t pos_z;
    uint16_t temperature;
    uint16_t soilmoisture;
};
typedef union Lora_Packet{
    Lora_Packet_Struct Lora_Packet_S;
    uint8_t byteArray[30];
};
Lora_Packet LoraPacket;

```

Fig. 3.10: Data Packet structure

- **ACK:** this type of packet is used in both protocols. It has a size of 18 bytes and consists of the following attributes. The *flag*, which determines the type of packet it is. The *satellite_id*, which determines the identifier of the satellite. The *packet_type*, which is 0 if it is a pure ALOHA protocol packet, or 1 if it is from the CSMA/CA protocol. The *node_id*, which determines which node the ACK packet is being sent to. The *packet_id*, which determines the number of packets sent. The *timestamp*, which indicates the relative time of the drone. Finally, the *free_slots*, which are not really used in this protocol but are added for later integrations.

```

typedef struct Lora_ACK_Struct{
    uint8_t flag;                //1
    uint8_t satellite_id;
    uint8_t packet_type;
    uint8_t node_id;
    uint32_t packet_id;
    uint64_t timestamp;
    uint16_t free_slots;
};
typedef union Lora_ACK{
    Lora_ACK_Struct Lora_ACK_S;
    uint8_t byteArray[18];
};
Lora_ACK LoraACK;

```

Fig. 3.11: ACK packet structure

- **RTS:** this type of packet is only used in the CSMA/CA protocol. It has a size of 18 bytes and consists of the following attributes. The *flag*, which determines

the type of packet it is. The *satellite_id*, which is obtained once the beacon is received from the satellite. The *packet_type*, which is 0 if it is a pure ALOHA protocol packet, or 1 if it is from the CSMA/CA protocol. The *node_id* which determines the identification of the node. The *packet_id*, which determines the number of packets sent. The *timestamp*, which indicates the relative time of the node. Finally, the *NAV_RTS*, which determines the waiting time necessary to re-sense the medium in the CSMA/CA protocol.

```
//LoraRTS
typedef struct Lora_RTS_Struct{
    uint8_t flag;                //4
    uint8_t satellite_id;
    uint8_t packet_type;
    uint8_t node_id;
    uint32_t packet_id;
    uint64_t timestamp;
    uint16_t NAV_RTS;
};
typedef union Lora_RTS{
    Lora_RTS_Struct Lora_RTS_S;
    uint8_t byteArray[18];
};
Lora_RTS LoraRTS;
```

Fig. 3.12: RTS packet structure

- **CTS:** this type of packet is only used in the CSMA/CA protocol. It has a size of 18 bytes and consists of the following attributes. The *flag*, which determines the type of packet it is. The *satellite_id*, which determines the identifier of the satellite. The *packet_type*, which is 0 if it is a pure ALOHA protocol packet, or 1 if it is from the CSMA/CA protocol. The *node_id*, which determines which node the ACK packet is being sent to. The *packet_id*, which determines the number of packets sent. The *timestamp*, which indicates the relative time of the drone. Finally, the *NAV_CTS*, which determines the waiting time necessary to re-sense the medium in the CSMA/CA protocol.

```
//LoraCTS
typedef struct Lora_CTS_Struct{
    uint8_t flag;                //3
    uint8_t satellite_id;
    uint8_t packet_type;
    uint8_t node_id;
    uint32_t packet_id;
    uint64_t timestamp;
    uint16_t NAV_CTS;
};
typedef union Lora_CTS{
    Lora_CTS_Struct Lora_CTS_S;
    uint8_t byteArray[18];
};
Lora_CTS LoraCTS;
```

Fig. 3.13: CTS packet structure

One of the essential attributes of all packages is the flag. Once a packet is received and addressed to the void OnRxDone() the packet type is determined from the flag. To categorize these packages a "typedef enum" list has been created with the different types. Categorizing them in this way serves to then use a state machine. In the same way, the diverse types of transmission or reception status have also been categorized, as well as the distinct types of protocols. The Figure 3.14 shows the different categorizations of states, packets, and protocols in the code of the ground nodes (red) and the drone payload (blue). As can be seen, different states are used and in the case of the drone code only a list is created with two types of packets (RTS and Data Packet), which are expected to be received during communications using the protocols. The different states of the protocols are detailed below.

<pre>typedef enum { RX, RX_ALOHA, RX_CSMA_CA, TX_ALOHA, TX_CSMA_CA, LOWPOWER }States_t; States_t state;</pre>	<pre>typedef enum { P_BEACON, P_RTS, P_CTS, P_ACK, P_DATAPACKET }Packets_t; Packets_t Packet;</pre>	<pre>typedef enum { ALOHA, CSMA_CA }Protocol_t; Protocol_t protocol;</pre>
<pre>typedef enum { RX_START, RX, TX_ALOHA, TX_CSMA_CA, LOWPOWER }States_t; States_t state;</pre>	<pre>typedef enum { P_RTS, P_DATAPACKET }Packets_t; Packets_t Packet;</pre>	<pre>typedef enum { ALOHA, CSMA_CA }Protocol_t; Protocol_t protocol;</pre>

Fig. 3.14: Different categorizations of states, packets, and protocols of the ground nodes code (red) and the drone payload code (blue).

As detailed in previous sections, each code works with 6 different functions: *void setup()*, *void loop()*, *void OnRxDone()*, *void OnTxDone()*, *void OnRxTimeout()* and *void OnTxTimeout()*. The following lines and figures detail the architecture of each of these functions.

Overall code design – Ground nodes: In this section, the structure that follows each of these functions in the ground nodes will be explained.

- ***void setup()*:** In addition to the configuration of the adjustable parameters of the LoRa physical layer, certain variables are also configured that are specific to each node, such as the identifier and the position. Finally, the state with which the loop will start is configured. Within the different states mentioned above in Figure 3.14, the starting state is the "RX" state.
- ***void loop()*:** this function varies between states with a switch. The first state to which switches is the "RX". This is because the setup has been configured to always be the first to run. Thus, the node remains in listening mode

whenever it is switched on for the first time. This allows a correct reception of the protocol command. The figure 3.15 shows a self-explanatory diagram. The code for each case will be explained below.

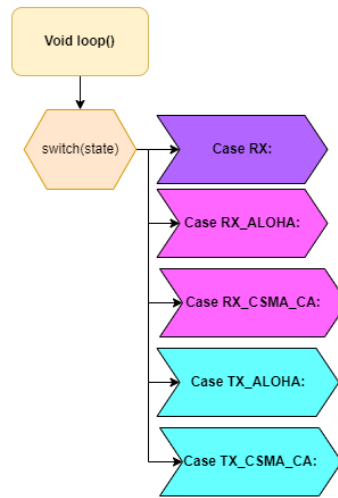


Fig. 3.15: Simplified structure of the “*void loop()*” function of the ground nodes code

- ***void OnRxDone(...)*:** the following function is executed when some kind of package has been received. The first step is to determine the type of package with the “*Flag Determination*”. This previous step is necessary as it categorizes the package type and decides the protocol to run. The first packet that all devices receive is the command packet, which determines the protocol that the user specifies along with other attributes of the protocol itself. The type of protocol to run is saved in the “*protocol*” variable. It is then determined whether the beacon sending the drone payload has been received. Since a command packet has been received and “*min_one_beacon_received*” is set as false at first, it goes to the “*RX*” state again to wait for the beacon that will send the drone payload.

After receiving the next package, which should be the beacon, the program executes again the “*void OnRxDone()*” function and the “*Flag Determination*” determines what type of package it is. If it is a Beacon, the boolean “*min_one_beacon_received*” becomes true and the *Packet* variable equals *P_BEACON*. This variable will then be used to switch between the distinct types of packages on the state machine.

Next, it is determined which type of protocol must be executed based on the “*protocol*” variable, which has been determined earlier with the command packet. Once either protocol is initiated, the next step is the same for both. In this step, it is determined whether the time of the experiment configured in the command packet has elapsed. This is achieved by comparing the current time of the node (*millis()*), with the time in which the command was received (*t_envio_com*) plus the time set for the duration of the experiment. If the elapsed time is less than the sum of these variables, the switch(*Packet*) is

executed, which varies between the different packets depending on which has been received. After the switch, it is checked if the "*wrong_packet*" boolean has been activated in any of the above cases. This happens in pure ALOHA when for example an ACK is expected and a DataPacket is received. In CSMA/CA it can occur when an RTS is received when a CTS is expected. How this process works will be detailed later along with the explanation of the protocols.

On the other hand, if the elapsed time exceeds the sum of the two variables, the experiment ends. At the end of the experiment, two Booleans become false. "*ProtocolAloha*" is a boolean used later, and "*min_one_beacon_received*" is set to false to return to the initial state. All nodes wait 10 seconds before re-listening the channel to receive the next command. This is done to avoid receiving any packets sent from another unsynchronized node after the experiment time has finished.

Figure 3.16 below shows the simplified structure of the "*void OnRxDone()*" function. In addition, Figure 3.17 has been added, which shows in more detail the functions performed by the "*Flag Determination*".



Fig. 3.16: Simplified structure of the "*void OnRxDone()*" function of the ground nodes code

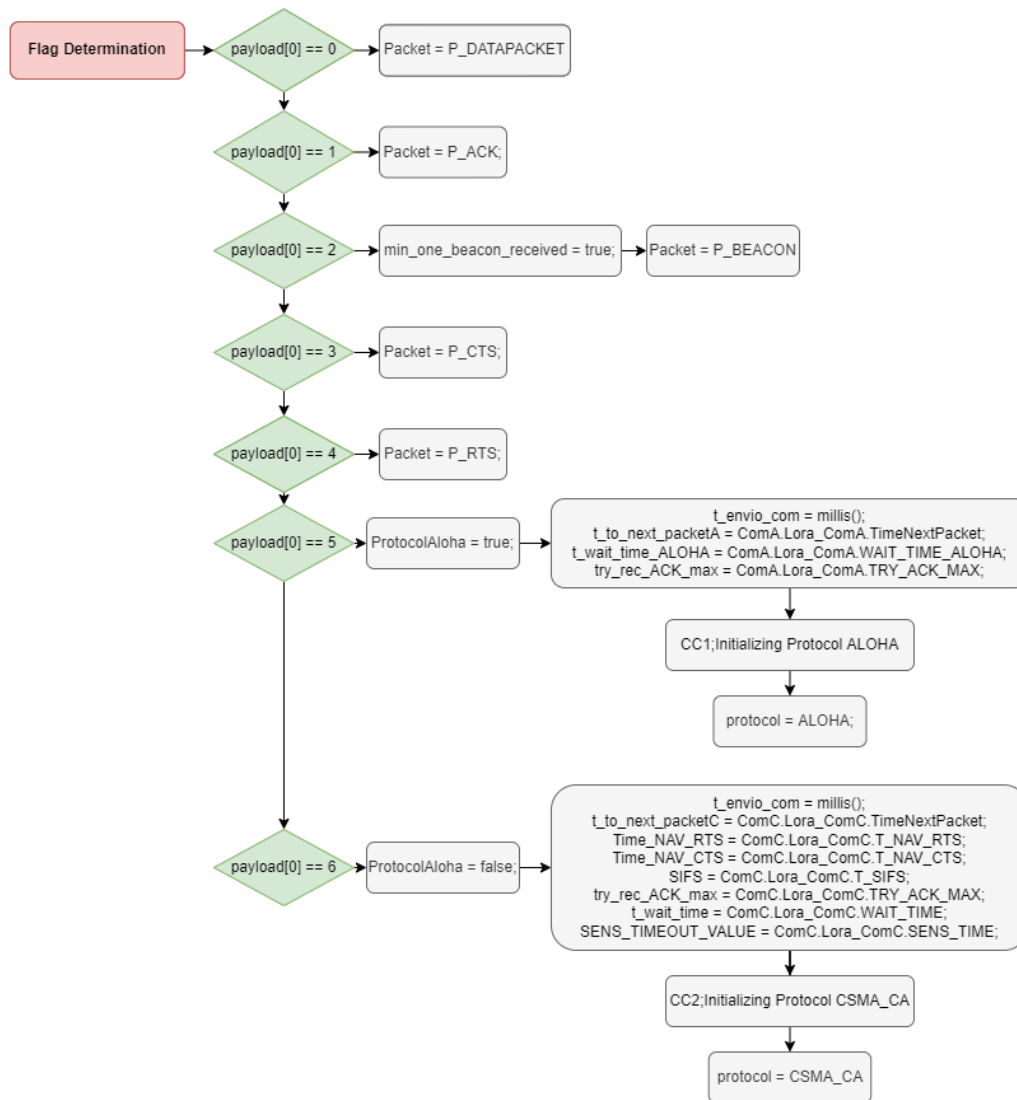


Fig. 3.17: Structure of the “*Flag Determination*” on the “*void OnRxDone()*” function of the ground nodes code

- ***void OnTxDone()***: the following function is executed once a package has been sent. As can be seen in the figure 3.18, it is determined which type of protocol is running based on the boolean “*ProtocolAloha*”. When the command is received with the protocol type to execute, this boolean becomes true if it is pure ALOHA and false if it is CSMA/CA.

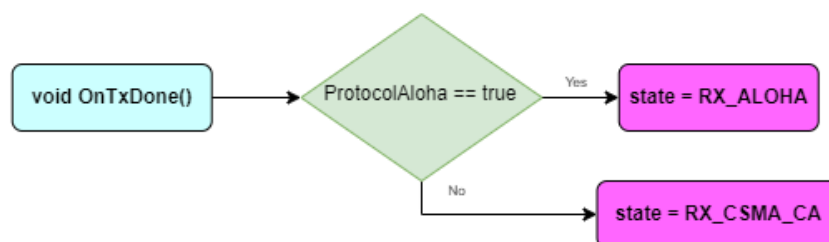


Fig. 3.18: Structure of the “*void OnTxDone()*” function of the ground nodes code

- ***void OnRxTimeout()***: the following function is executed when a certain time has passed in reception and no packets have been received. In these cases, this function is directed towards the case of the protocol being executed. Each case will be detailed later when the protocols are explained.

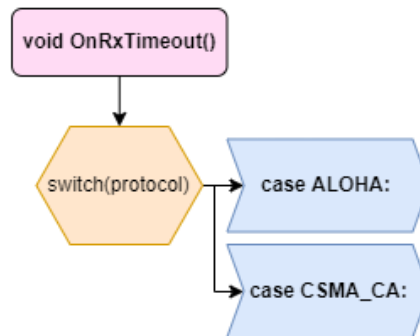


Fig. 3.19: Simplified structure of the “*void OnRxTimeout()*” function of the ground nodes code

- ***OnTxTimeout()***: the following function is executed if there is no connection between the micro-controller and the radio. This function should not be executed at any time if there is no problem.

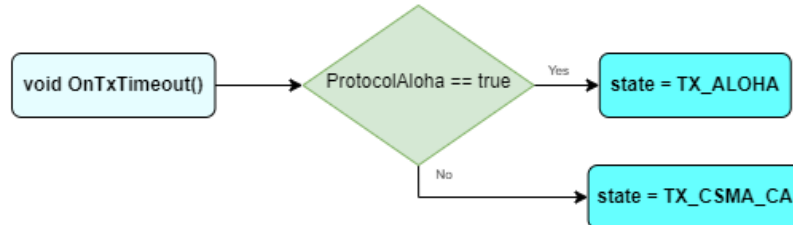


Fig. 3.20: Structure of the “*void OnTxTimeout()*” function of the ground nodes code

Overall code design – Drone payload: In this section, the structure that follows each of these functions in the drone payload will be explained.

- ***void setup()***: in addition to the configuration of the adjustable parameters of the LoRa physical layer, the identifier of the satellite is also configured. Finally, the state with which the loop will start is configured. Within the different states mentioned above in Figure 3.14, the starting state is the “*RX_START*” state.
- ***void loop()***: this function varies between states with a switch. The first state to which switches is the “*RX_START*”. This is because the setup has been configured to always be the first to run. Thus, the node remains in listening mode whenever it is switched on for the first time. This allows a correct reception of the protocol command. The figure 3.21 shows a self-explanatory diagram. The code for each case will be explained below.

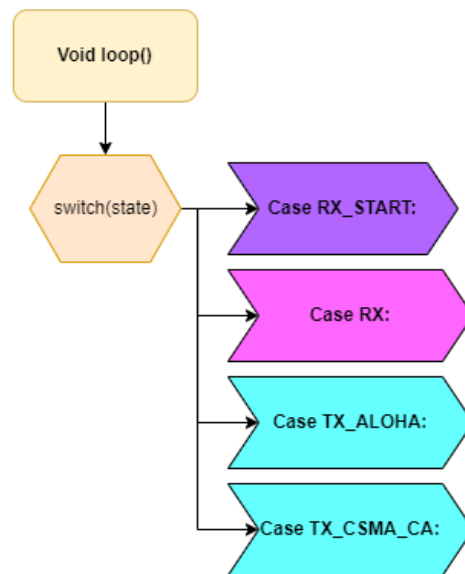


Fig. 3.21: Simplified structure of the “*void loop()*” function of the drone payload code

- ***void OnRxDone()*:** the following function is executed when some kind of package has been received. The first step is to determine the type of package with the “*Flag Determination*”. This previous step is necessary as it categorizes the package type and decides the protocol to run. The first packet that all devices receive is the command packet, which determines the protocol that the user specifies along with other attributes of the protocol itself. The type of protocol to run is saved in the “*protocol*” variable. After selecting the protocol with the switch, it is determined whether the beacon has been sent to the ground nodes to initiate communications. Since a command packet has been received and “*min_one_beacon_sended*” is set as false at first, it goes to the “*TX_ALOHA*” or “*TX_CSMA_CA*” state to send the beacon. Figure 3.22 also shows how before executing the different transmission states, the “*beacon_sended*” boolean is configured as false. Later, it is detailed how this boolean intervenes in the different states.

After sending the beacon, the program listens to the channel until it receives the next packet. Once it receives the next packet, the program executes again the “*void OnRxDone()*” function and the “*Flag Determination*” determines what type of package it is (saved in *Packet*). Remaining in the same protocol, now the “*min_one_beacon_sended*” boolean is true, so the next step is taken.

In this step, it is determined whether the time of the experiment configured in the command packet has elapsed. This is achieved by comparing the current time of the node (*millis()*), with the time in which the command was received (*t_envio_com*) plus the time set for the duration of the experiment. If the elapsed time is less than the sum of these variables, the switch (*Packet*) is executed, which varies between the different packets depending on which has been received. After the switch, it is checked if the “*wrong_packet*” boolean

has been activated in any of the above cases. This happens in pure ALOHA when for example an ACK is expected and a Data Packet is received. In CSMA/CA it can occur when an RTS is received when a CTS is expected. How this process works will be detailed later along with the explanation of the protocols.

On the other hand, if the elapsed time exceeds the sum of the two variables, the experiment ends. At the end of the experiment, the different Booleans of each protocol become false to set everything up as in a startup. Then, the drone payload waits 10 seconds before re-listening the channel to receive the next command. This is done to avoid receiving any packets sent from another unsynchronized node after the experiment time has finished.

Figure 3.22 below shows the simplified structure of the “void OnRxDone()” function of the drone payload code. In addition, Figure 3.23 has been added, which shows in more detail the functions performed by the “Flag Determination”.

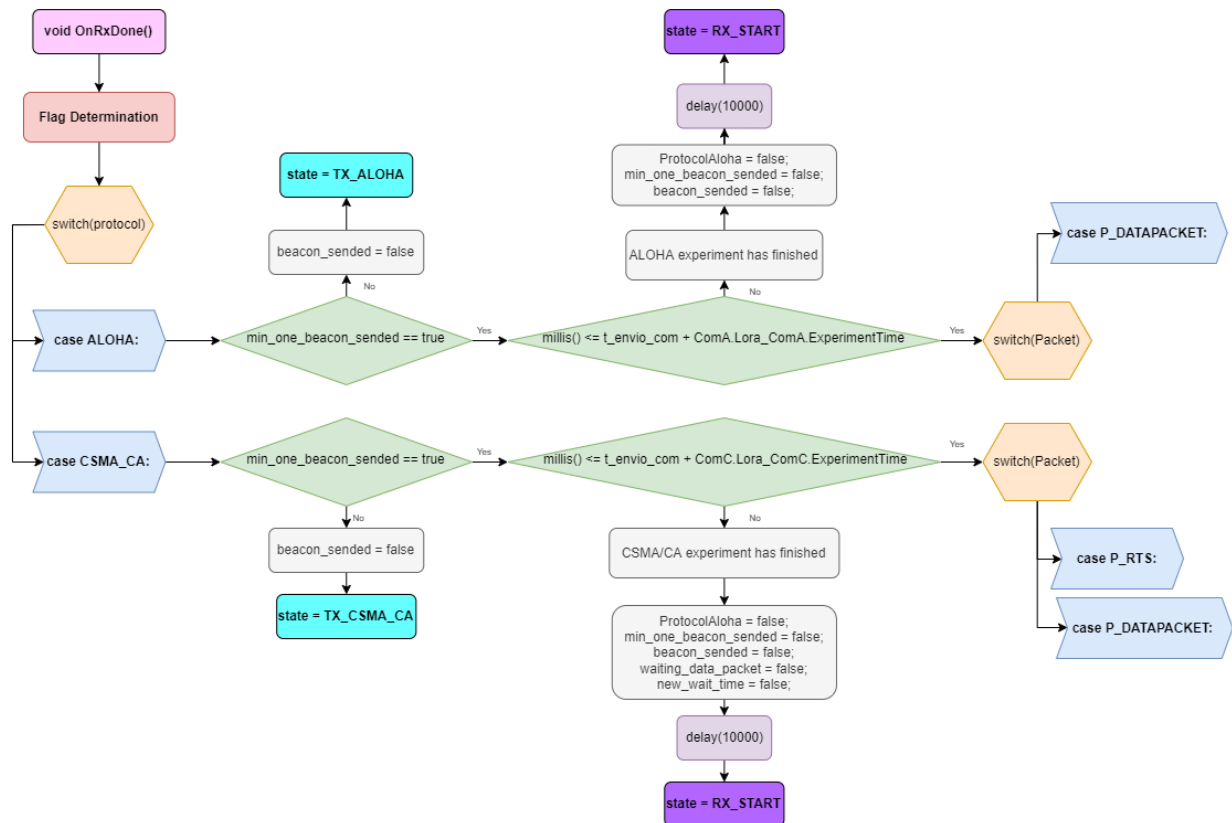


Fig. 3.22: Simplified structure of the “void OnRxDone()” function of the drone payload code

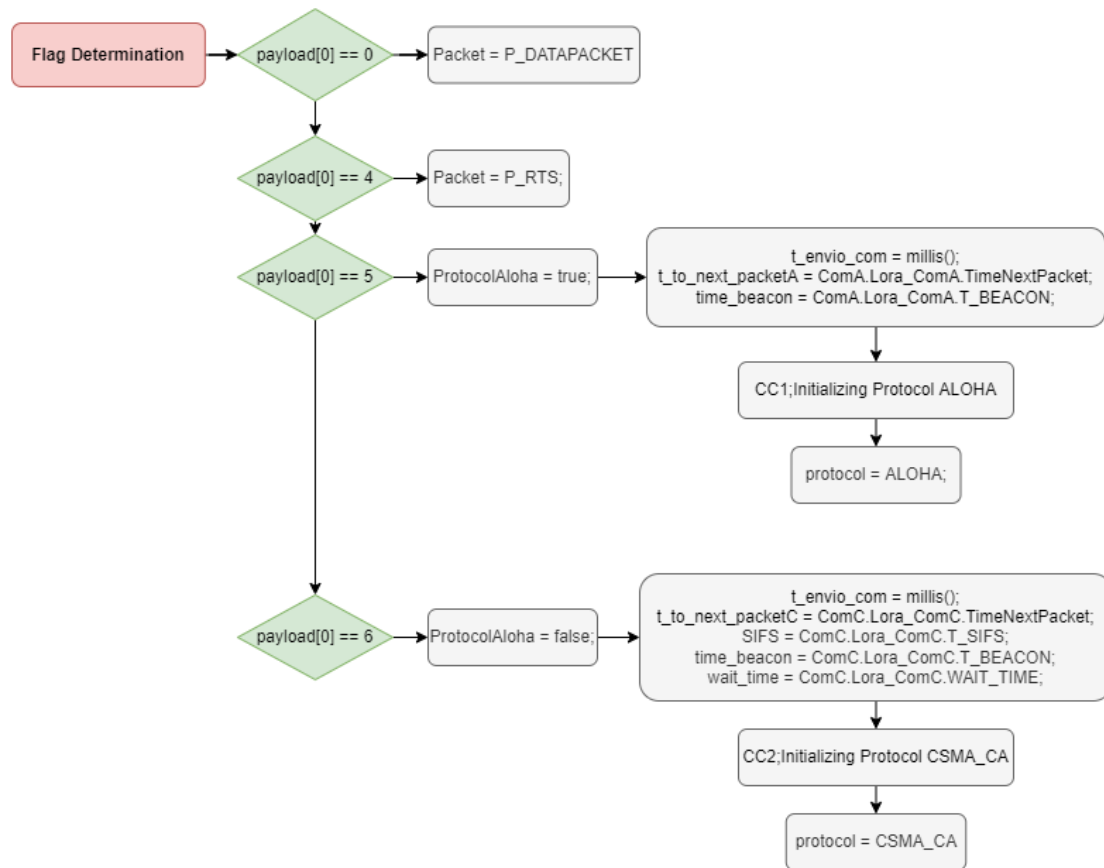


Fig. 3.23: Structure of the “*Flag Determination*” on the “*void OnRxDone()*” function of the drone payload code

- ***void OnTxDone()***: the following function is executed once a package has been sent. As can be seen in the figure 3.24, the protocol is changed through the switch. In case it is the pure ALOHA protocol, after transmitting any packet it always listens to the channel permanently. However, if the protocol is CSMA/CA, it listens to the channel permanently only if the Boolean CTS is false. If it is true, it listens to the channel for a certain time (wait time).

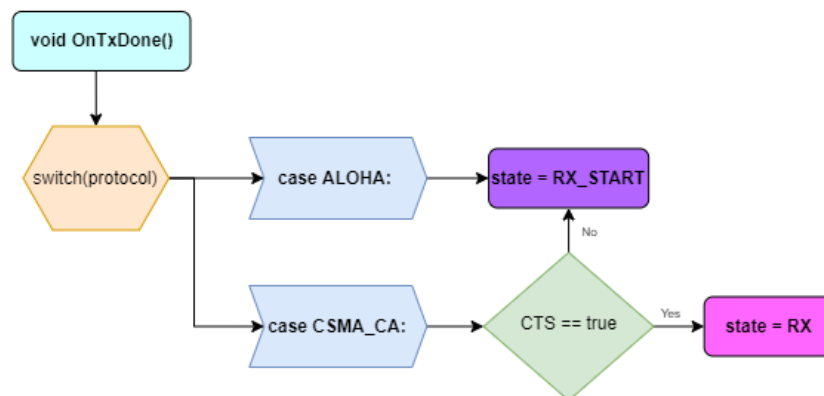


Fig. 3.24: Structure of the “*void OnTxDone()*” function of the drone payload code

- ***void OnRxTimeout()***: the following function is executed when a certain time has passed in reception and no packets have been received. In this case, this function is only used for responses in the CSMA/CA protocol. When a CTS is sent, the receiving station waits for a certain time to receive the Data Packet from the node to which the CTS has been sent. This mechanism is implemented to prevent the receiving station from waiting for an infinite amount of time for a package that never arrives. During this waiting time, RTS packets may be received from other nodes. These are recorded and continued with the listening until the waiting time ends or the Data Packet is received.

As can be seen in the figure 3.25, the first step is to determine whether the elapsed time is longer than the duration proposed for the experiment. If the entire time of the experiment has not yet passed, a message is printed which says that the Data Packet has not been received after the waiting time. Subsequently, three Booleans are configured as false. The first of these, *waiting_data_packet* is set as false since the waiting time to receive the data packet has elapsed. Secondly, the CTS is configured as false to configure the program as in the initial state. Finally, *new_waiting_time* is set to false. This term will be explained in the following sections where the processes followed by the CSMA/CA protocol are detailed.

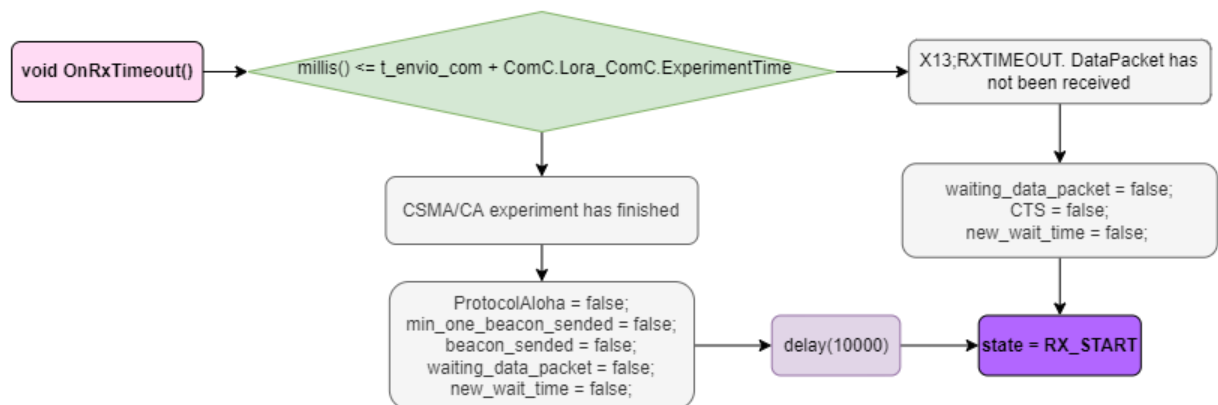


Fig. 3.25: Simplified structure of the “*void OnRxTimeout()*” function of the drone payload code

- ***OnTxTimeout()***: the following function is executed if there is no connection between the micro-controller and the radio. This function should not be executed at any time if there is no problem.

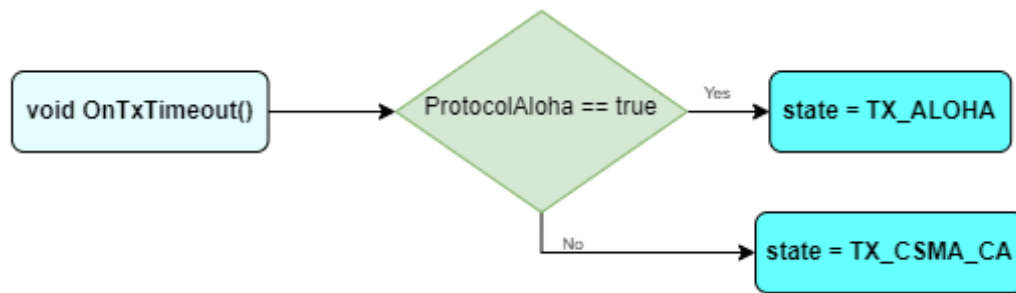


Fig. 3.26: Structure of the “*void OnTxTimeout()*” function of the drone payload code

Before explaining the design of each protocol, it will be explained how each packet that is sent or received has been recorded. Also, it is explained how it has been done to synchronize the times of all nodes with the relative time of the drone payload.

First, all messages that are printed are recorded with a particular label as the first component. Subsequently, the data are sorted by separating them with a “,” in an orderly manner. This makes processing easier and permits to separate data into columns and apply filters.

Secondly, to synchronize all nodes with the relative time of the drone payload the variable “*Dif_t*” has been used. This variable indicates the time difference between the ground node and the drone payload. It is calculated on the ground nodes when the beacon is received using the following formula (3.1):

$$Dif_t = tiempo_exp - (LoraB.Lora_BS.timestamp + t_prop_bea) \quad (3.1)$$

In the experiment, all ground nodes are switched on before the drone, so they all have a larger relative time. These times are calculated at the moment the beacon is received using the *millis()* function, which returns in milliseconds the time the program loaded on the computer has been running. The received beacon contains the relative time in which it was sent from the drone. To this is added the propagation time since it does not propagate immediately. With the subtraction indicated in the formula, the temporal difference between the node and the drone is obtained. Having the value of this variable, for each message that is registered as sent or received, the timestamp calculated as the relative time of the node (*millis()*) minus *Dif_t* will be printed. In this way, a temporary control of the flow of packets between all nodes is achieved.

3.2.2.3. Pure ALOHA design

The following section will explain in detail how the pure ALOHA protocol has been implemented. Throughout this section you will see two different codes, the code of the algorithm implemented on the ground nodes, and the code of the receiving station, in our case the drone. To explain the implementation of the algorithm of

the pure ALOHA protocol, an ideal case will be presented where the event line is fulfilled in an orderly manner. During the explanation, the different casuistics will be explained for the different cases.

Communications are initiated when the beacon is received. Figure 3.27 shows the procedure used to send the beacon from the drone. The *TX_ALOHA* state is configured after receiving the pure ALOHA protocol command, at this moment the nodes will be listening to the channel waiting to receive the beacon. As can be seen in the figure 3.27, the first step is to check that the experiment time is not finished, then a variable called "*time_lapse*" is calculated. The result of this variable is determined by the subtraction of the current time ($t_{actual} = \text{millis}()$) minus the time in which the last beacon was sent (t_{send_beacon}). This is done to then compare if the elapsed time (*time_lapse*) is greater than the beacon sending periodicity time (t_{beacon}). If this happens (which is not the case for the first send), the boolean *beacon_sended* and *ACK* are set to false. Then, the boolean *beacon_sended* is detected to be false and the beacon is sent. With this sending, two booleans are activated (*min_one beacon_sended* and *beacon_sended*) and the relative time in which it was sent is saved in the t_{send_beacon} variable. Once the beacon is sent, the program is directed to the "*void OnTxDone()*" function shown in Figure 3.24. This function sets the next state of the loop to *RX_START*. In this state, the drone remains listening to the channel until it receives the next package.

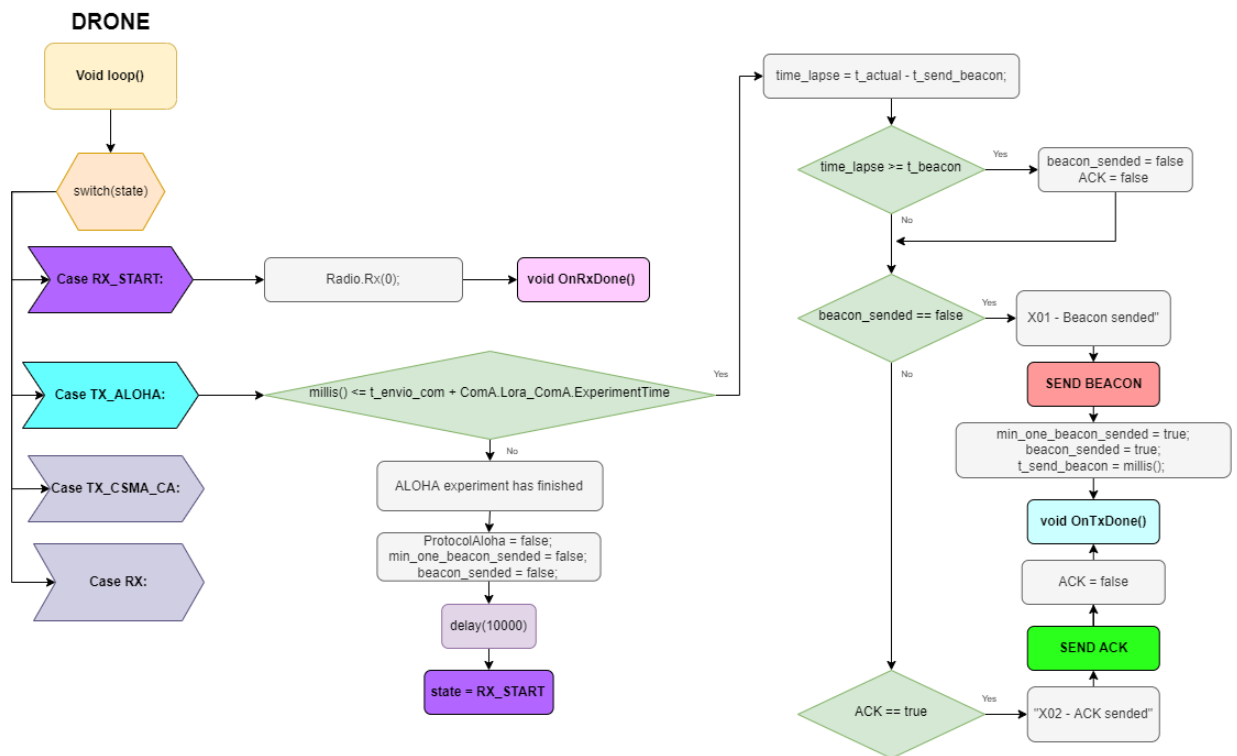


Fig. 3.27: Structure of the different states of the pure ALOHA protocol in the *void loop()* function of the drone payload.

The nodes are listening to the channel until they receive the beacon. Once it is received, the “*void OnRxDone()*” function is executed. Since a Beacon has been received, the *Packet* equals *P_BEACON* in the *Flag Determination* and the switch selects this case. Figure 3.28 explains in detail what happens when a beacon is received at the ground nodes. First, it is recorded that a beacon has been received, then the relative time of the node is saved in the time variable *tiempo_exp*, which is then used to calculate *Dif_t*. Finally, the Boolean *min_one_beacon_received* is set to true as the first beacon has been received. The status is then changed to *TX_ALOHA* to send the Data Packet.

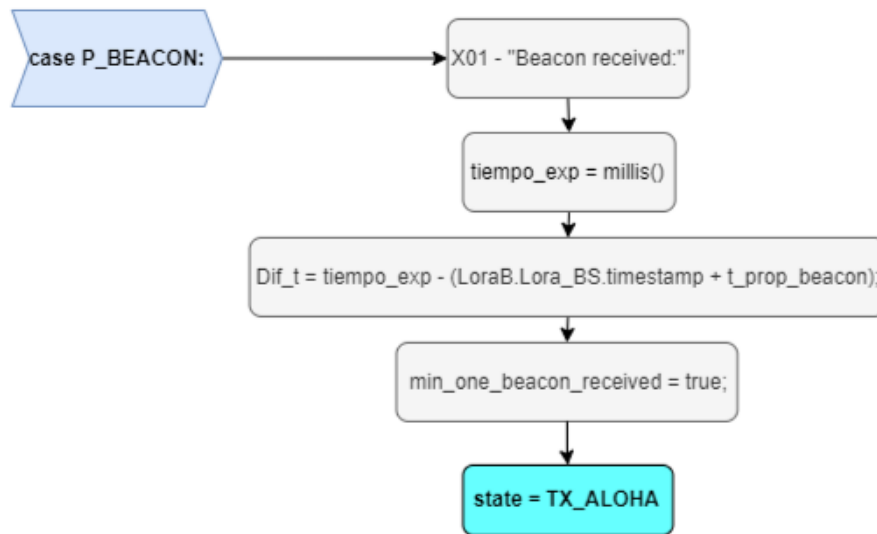


Fig. 3.28: Case where a beacon is received in the “*void OnRxDone()*” function in the ground nodes code.

Once the nodes have received the beacon, the next step is to send the Data Packet. This action is carried out in the “*void loop()*”, specifically in the *TX_ALOHA* state. The Figure 3.29 shows the procedure used to send the data package in the *TX_ALOHA* case. After sending it, the send is recorded. Finally, the relative time of the node is saved using “*tiempo_envio*”. This variable indicates the moment in which the data packet was sent. Once the data packet has been sent, the program execute the “*void OnTxDone()*” function (Figure 3.18), the state changes to *RX_ALOHA* and the loop is executed again.

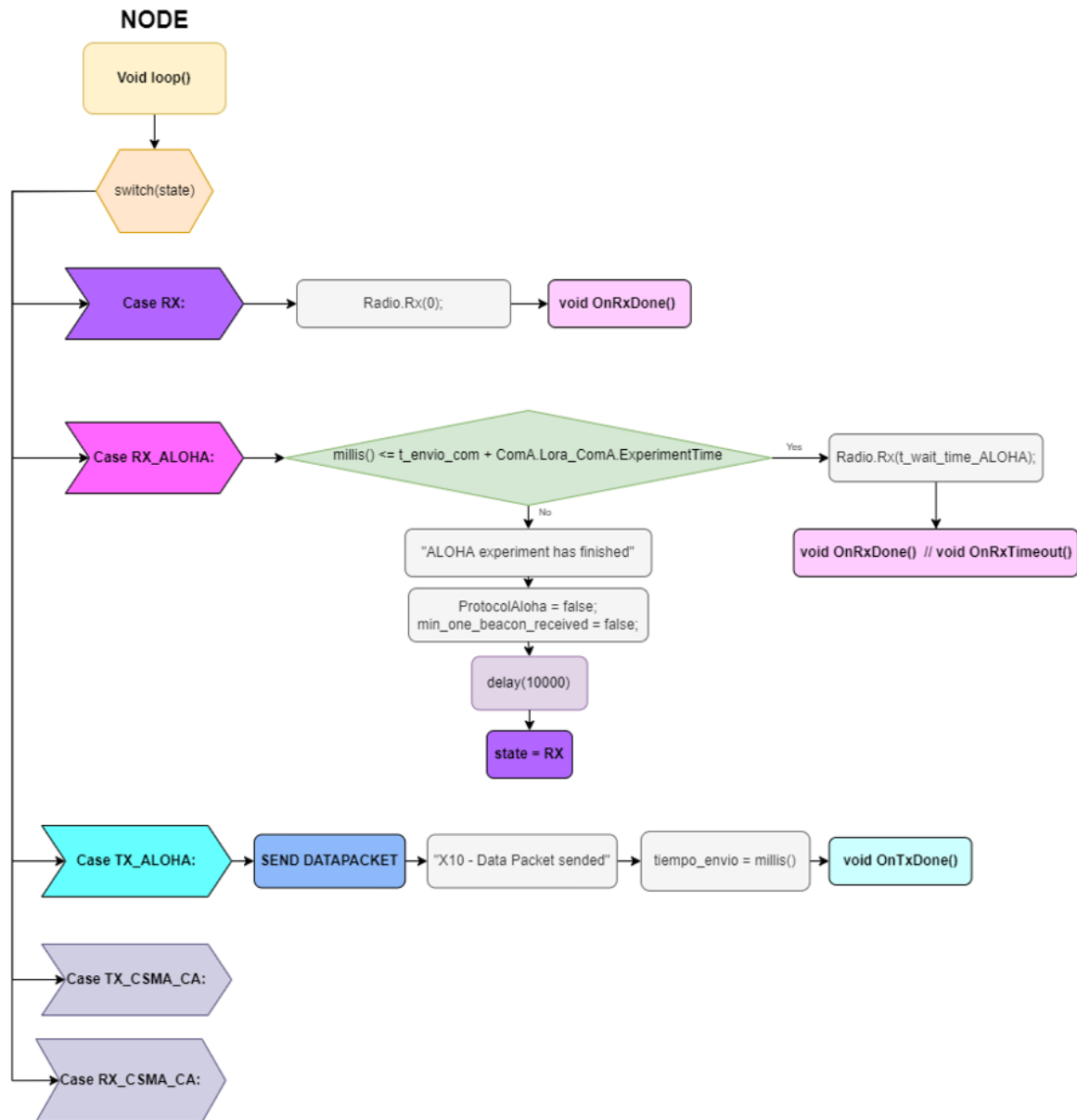


Fig. 3.29: Structure of the different states of the pure ALOHA protocol in the “void loop()” function in the ground nodes code.

The data packet sent by the node reaches the drone if there has been no collision. In that case, the “void OnRxDone()” function of the drone payload code is executed as shown in Figure 3.30. First, the received packet type is detected using the *Flag Determination*. Subsequently, the ALOHA protocol is switched. Then, since at least one beacon has already been sent, *min_one_beacon_sended* is true. In the next step, it is checked that the experiment time is not finished. Finally, it is switched to the case where a packet of data has been received. First, it is checked that the data package corresponds to the satellite/drone by comparing the *satellite_id* contained in the data package with the *SAT_ID*. If they match, the data packet is logged and the ACK boolean is set to true.

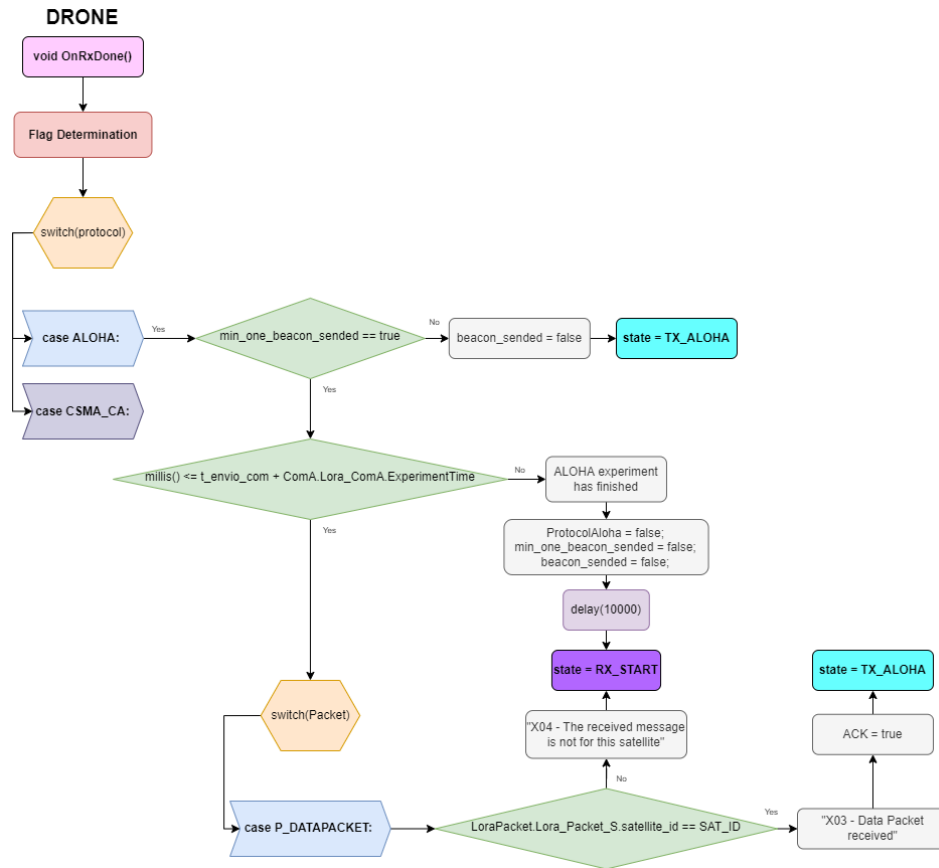


Fig. 3.30: Structure of the processes of the pure ALOHA protocol in the *void OnRxDone()* function of the drone payload.

State changes to TX_ALOHA and “*void loop()*” function of the drone payload is executed again (Figure 3.27). In the TX_ALOHA state, the ACK boolean is detected to be true and the ACK is sent. The packet is logged and the boolean is set as false again.

After sending the data packet, the ground nodes begin to listen to the channel waiting to receive the ACK sent by the drone. The medium is heard for a time determined by the variable $t_{wait_time_ALOHA}$. The value of this variable is sent through the command, and its numeric value will be calculated in section 3.2.3. When the medium is heard for a certain time, four things can happen. The first is to receive the expected ACK. The second, receive an ACK directed to another node. The third, receive a data packet from another node. The fourth is that the waiting time elapses, and no package is received. In the first three cases, the program goes to the “*void OnRxDone()*” function as a packet has been received. In the latter case, the program executes the “*void OnRxTimeout()*” function. Figures 3.16 and 3.19, previously seen, gave an overview of the architecture of the functions “*void OnRxDone()*” and “*void OnRxTimeout()*” of the code of the ground nodes. Each of the individual cases that may occur depending on the package received is explained below.

The first case corresponds to the process of receiving the correct ACK. As can be seen in Figure 3.31, when an ACK is received, the first thing that is checked is if there is a bug related to the reception of the beacon, since it could be that the ACK is received without the first beacon having arrived. Next, check whether the *node_id* contained in the ACK package corresponds to the node. If so, the ACK package has been received and the communication is considered as successful. After recording the ACK, two variables used in the *BackOff* process are configured (*try_rec_ACK* and *T_b*). Both are set to zero, configuring them as well as the initial state for the next communication attempt. Finally, the node has a waiting time equal to the *t_wait_time_ALOHA* before transmitting again. This is done to prevent the same node from occupying the channel just after receiving the ACK. Also, it is added the time determined by the user that indicates the periodicity of sending Data Packets.

The second case corresponds to the process of receiving the wrong ACK. This occurs when an ACK packet is received where the node identifier does not correspond to the node. In these cases, the erroneous ACK packet is recorded and the *wrong_packet* Boolean is configured as true.

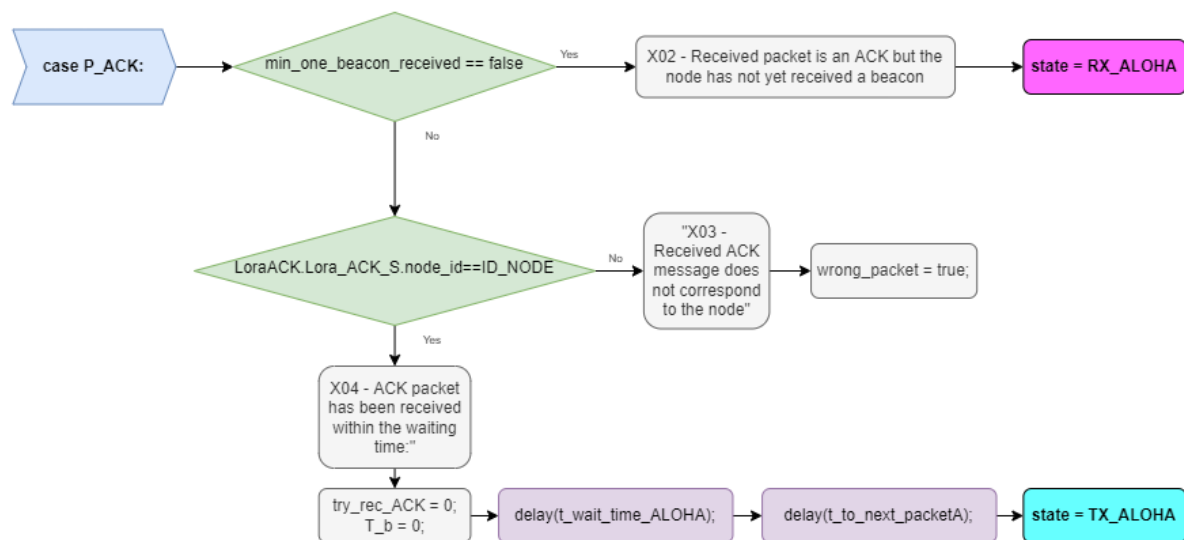


Fig 3.31: Case where a ACK packet is received in the “*void OnRxDone()*” function in the ground nodes code.

The third case corresponds to receiving a data packet from another node during the waiting time. In these cases, the data packet is recorded and the *wrong_packet* Boolean is set as true. Figure 3.32 shows the outline of the procedure to be followed in these cases.

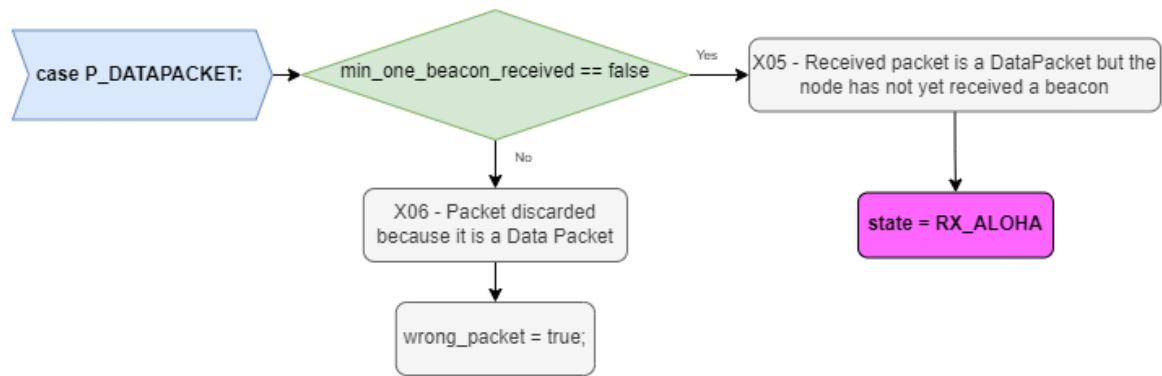


Fig. 3.32: Case where a Data Packet is received in the “*void OnRxDone()*” function in the ground nodes code

The fourth case occurs when the waiting time is exceeded, and no package is received. In such cases, the program executes the function “*void OnRxTimeout()*”. As can be seen in Figure 3.33, the first step is to check whether the experiment time has not finished. The *X11* message is then recorded, indicating that the ACK packet has not been received after the waiting time. The *back-off* process of the pure ALOHA protocol is then performed. First a reception attempt is added, then it is verified that the ACK reception attempts have not been exceeded. If the reception attempts have been exceeded, the message *X09* is recorded, and the variables *try_rec_ACK* and *T_b* are configured. Both are set to zero, configuring them as well as the initial state for the next communication attempt. If the reception attempts have not been exceeded, the *back-off* time is calculated as the multiplication of the waiting time by a random number *R*. The program then waits for a time *T_b* until the state is changed to *TX_ALOHA*. In this way, the communication process is restarted by sending the Data Packet and then trying to receive the ACK.

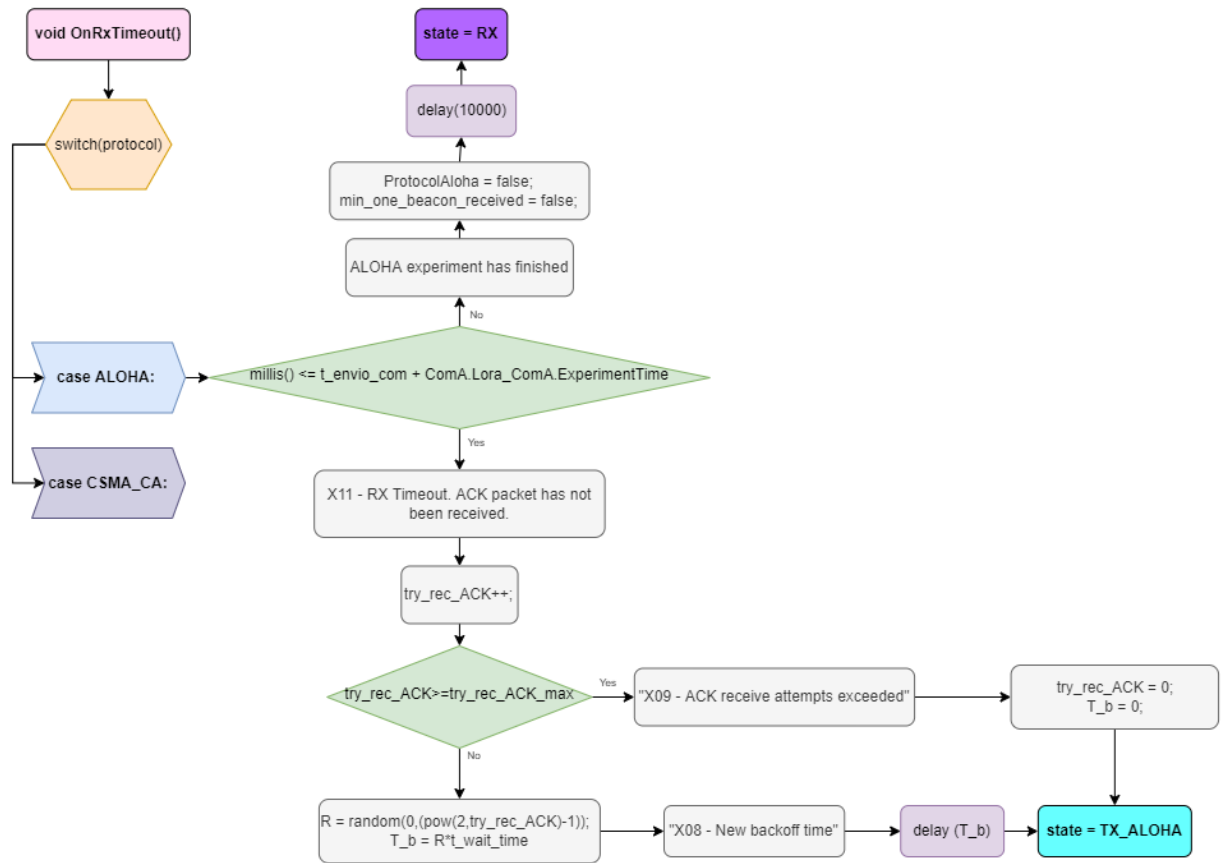


Fig. 3.33: “void OnRxTimeout()” function of the ground nodes code

Finally, it remains to clarify what happens when a wrong packet is received and the Boolean *wrong_packet* is activated after passing through the Switch. This process is controlled by an “if”, so it is only executed if the *wrong_packet* Boolean is true. First, the Boolean *wrong_packet* is set as false again. Next, the timer variable is calculated. This variable is calculated as the subtraction between the node's current time minus the time the data packet was sent. If this result is not equal to or greater than the waiting time, calculate it again until it is. Once the waiting time has been exceeded, and the ACK has not been received the communication cannot be given as successful. In this case, the Back-off process explained above is performed again. Finally, the *TX_ALOHA* state is reconfigured. In this way, the communication process is restarted by sending the Data Packet and then trying to receive the ACK.

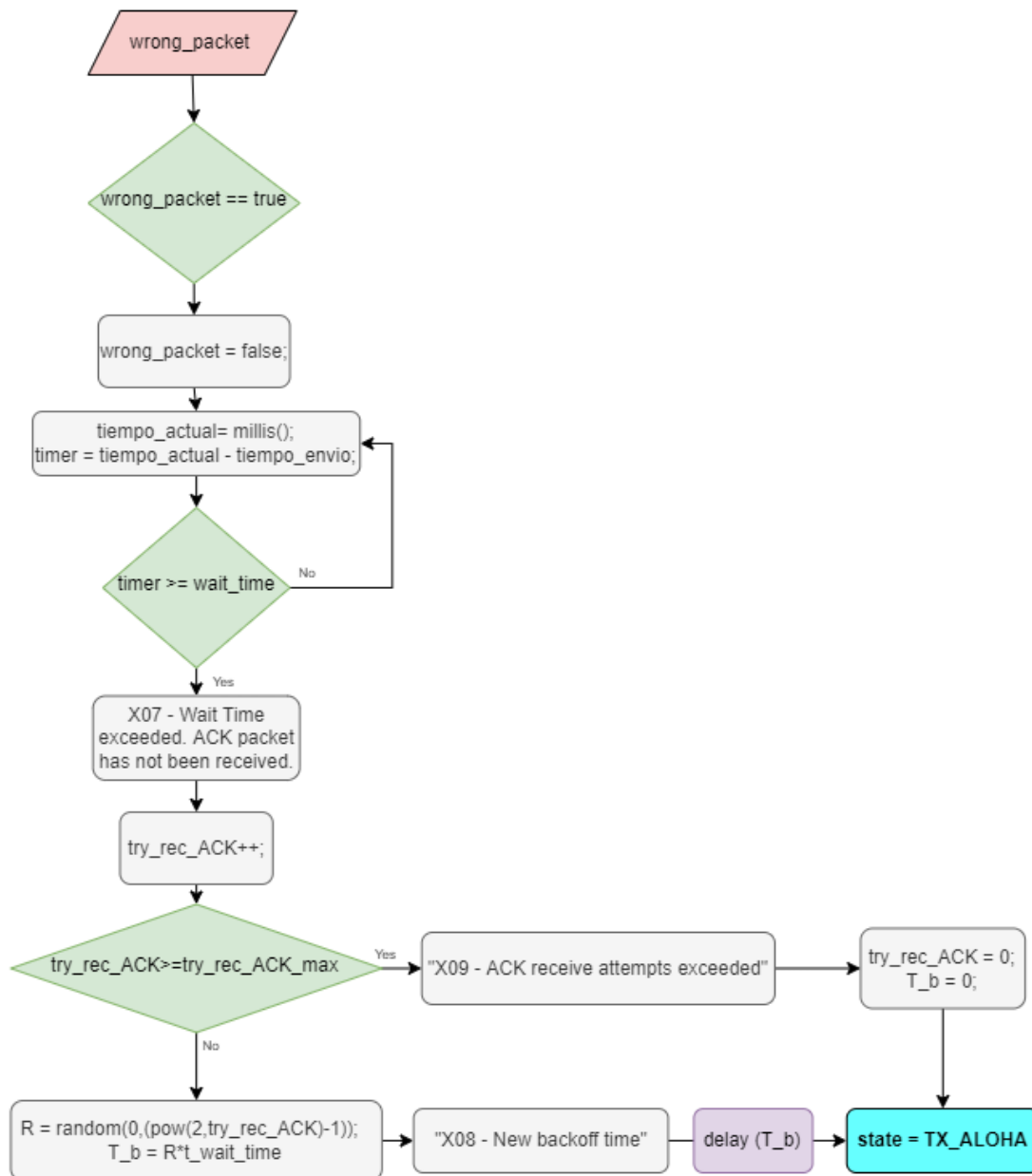


Fig. 3.34: Wrong packet case on the ground nodes code

3.2.2.4. CSMA/CA design

The following section will explain in detail how the CSMA/CA protocol has been implemented. Throughout this section you will see two different codes, the code of the algorithm implemented on the ground nodes, and the code of the receiving station, in our case the drone. To explain the implementation of the algorithm of the CSMA/CA protocol, an ideal case will be presented where the event line is fulfilled in an orderly manner. During the explanation, the different casuistic will be explained for the different cases.

Communications are initiated when the beacon is received. Figure 3.35 shows the procedure used to send the beacon from the drone. The *TX_CSMA_CA* state is configured after receiving the CSMA/CA protocol command, at this moment the nodes will be listening to the channel waiting to receive the beacon. As can be seen in the figure 3.35, the first step is to check that the experiment time is not finished, then a variable called "*time_lapse*" is calculated. The result of this variable is determined by the subtraction of the current time ($t_{actual} = \text{millis}()$) minus the time in which the last beacon was sent (t_{send_beacon}). This is done to then compare if the elapsed time (*time_lapse*) is greater than the beacon sending periodicity time (t_{beacon}). If this happens (which is not the case for the first send), the boolean *beacon_sended*, CTS and *ACK* are set to false. Then, the boolean *beacon_sended* is detected to be false and the beacon is sent. With this sending, two booleans are activated (*min_one beacon_sended* and *beacon_sended*) and the relative time in which it was sent is saved in the t_{send_beacon} variable. Once the beacon is sent, the program is directed to the "*void OnTxDone()*" function shown in Figure 3.24. This function sets the next state of the loop to *RX_START*. In this state, the drone remains listening to the channel until it receives the next package.

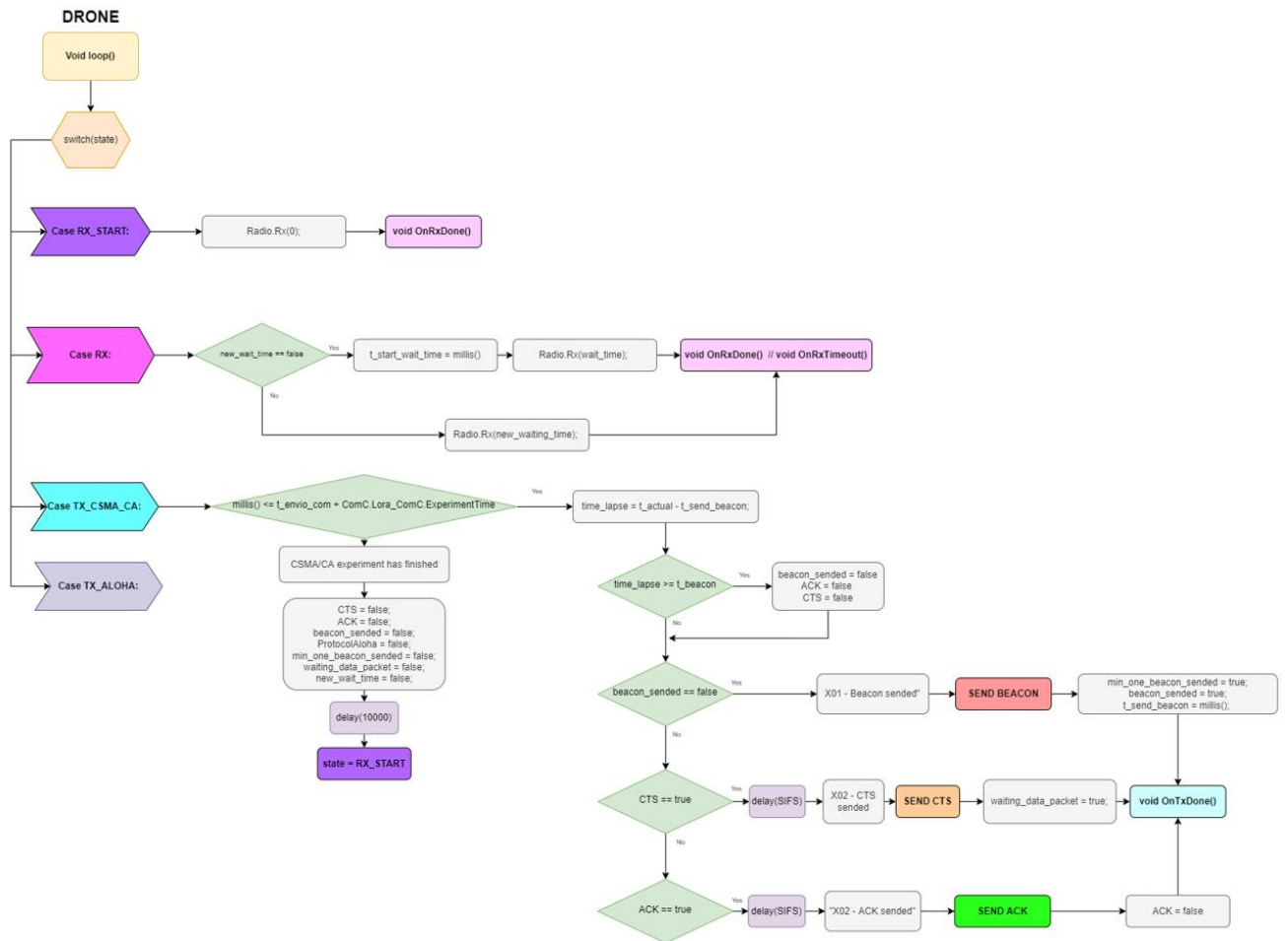


Fig. 3.35: Structure of the different states of the CSMA/CA protocol in the *void loop()* function of the drone payload.

The nodes are listening to the channel until they receive the beacon. Once it is received, the “*void OnRxDone()*” function is executed. Since a Beacon has been received, the *Packet* equals *P_BEACON* in the *Flag Determination* and the switch selects this case. Figure 3.36 explains in detail what happens when a beacon is received at the ground nodes. First, it is recorded that a beacon has been received, then the relative time of the node is saved in the time variable *tiempo_exp*, which is then used to calculate *Dif_t*. Next, the Boolean *sensing_channel* is configured as true to sense the channel. RTS and *send_datapacket* booleans are configured as false. Finally, one of the variables used for the backoff is set to 0, thus restarting the number of attempts to receive the ACK. The status is then changed to *RX_CSMA_CA* to sense the channel.

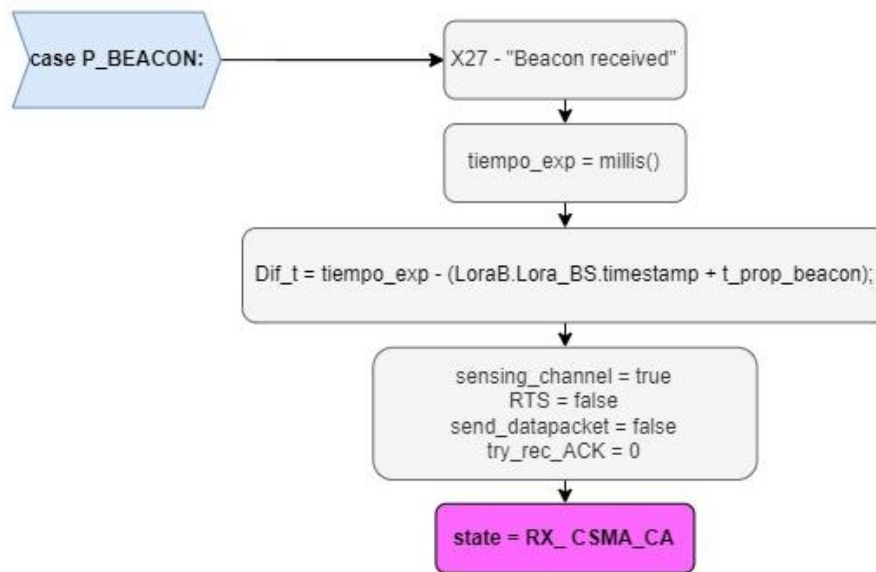


Fig. 3.36: Case where a beacon is received in the “*void OnRxDone()*” function in the ground nodes code.

Once the nodes receive the beacon, the loop state changes to *RX_CSMA_CA*. First, it is checked that the experiment time is not finished. Next, it is checked if the Boolean *sensing_channel* is true to decide which action to execute. Since the Boolean is true because it has just been activated previously, the channel is sensed for a certain time. The way to sense the channel corresponds to the nonpersistent method explained above. If during this time, no packets are received, the program executes the “*void OnRxTimeout()*” function. However, if a packet is received, the program executes the “*void OnRxDone()*” function. If a packet is received, it implies that there is communication between some other node and the drone, so the node will have to wait for a certain time to sense the channel again. Figure 3.37 shows the structure of the “*void loop()*” function.

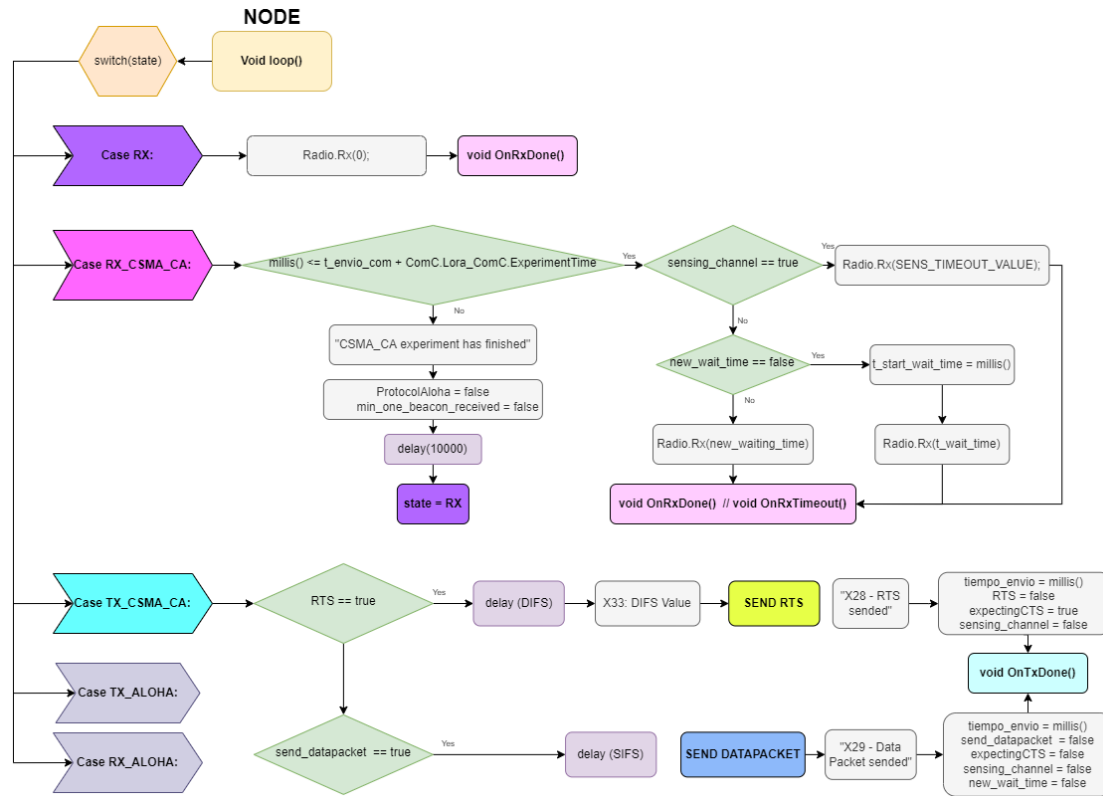


Fig. 3.37: Structure of the *void loop()* function of the ground nodes code.

If a packet has been detected while sensing the medium, the node acts in the following ways depending on the received packet:

- **RTS:** if the received packet is an RTS packet, the following steps are followed. First, the received packet is recorded. Then, a waiting time defined by the NAV of the RTS package is configured. After this waiting time, the *RX_CSMA_CA* state is reconfigured to sense the channel again.

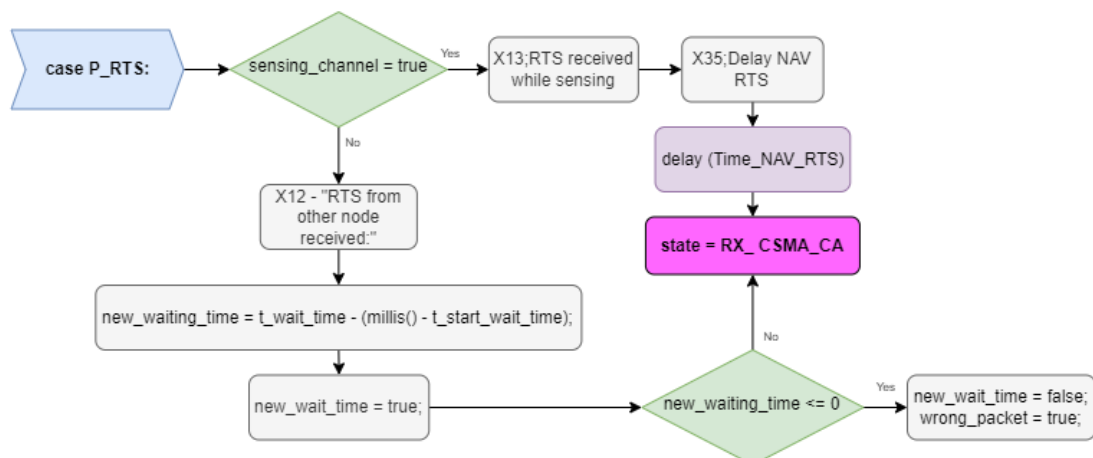


Fig. 3.38: Case where a RTS is received in the “*void OnRxDone()*” function in the ground nodes code.

- **CTS:** if the received packet is an CTS packet, the following steps are followed. First, the received packet is recorded. Then, a waiting time defined by the NAV of the CTS package is configured. After this waiting time, the *RX_CSMA_CA* state is reconfigured to sense the channel again.

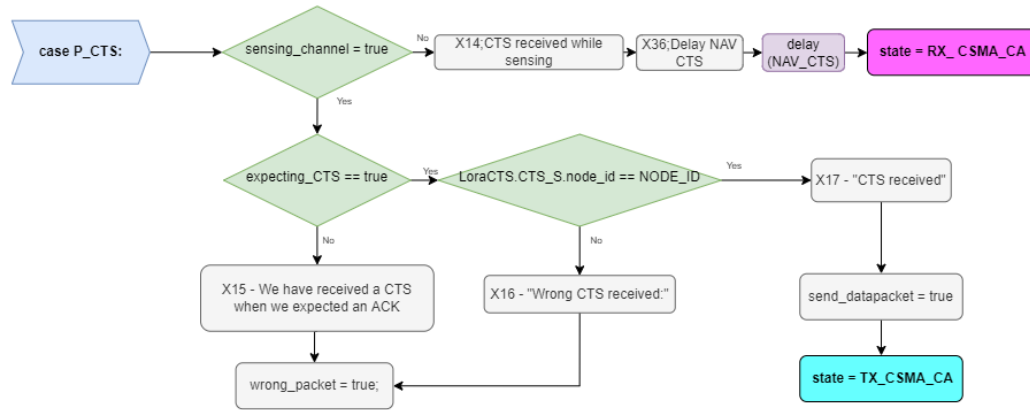


Fig. 3.39: Case where a CTS is received in the “*void OnRxDone()*” function in the ground nodes code.

- **ACK:** if the received packet is an ACK packet, the following steps are followed. First, the received packet is recorded. Then, a random waiting time defined by the nonpersistent method is configured. The method to calculate this value of time will be explained in section 3.2.3. After this waiting time, the *RX_CSMA_CA* state is reconfigured to sense the channel again.

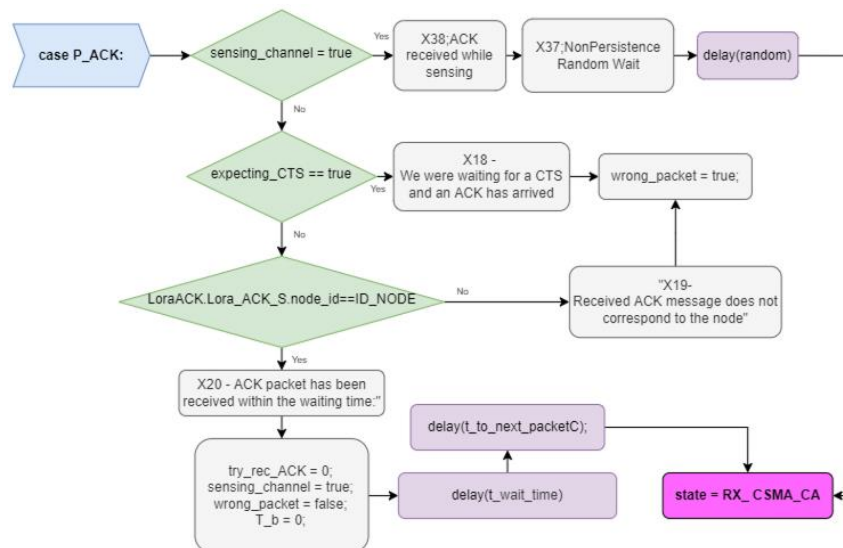


Fig. 3.40: Case where a ACK is received in the “*void OnRxDone()*” function in the ground nodes code.

- **Data Packet:** if the received packet is a Data Packet, the following steps are followed. First, the received packet is recorded. Then, a random waiting time defined by the nonpersistent method is configured. The method to calculate this value of time will be explained in section 3.2.3. After this waiting time, the *RX_CSMA_CA* state is reconfigured to sense the channel again

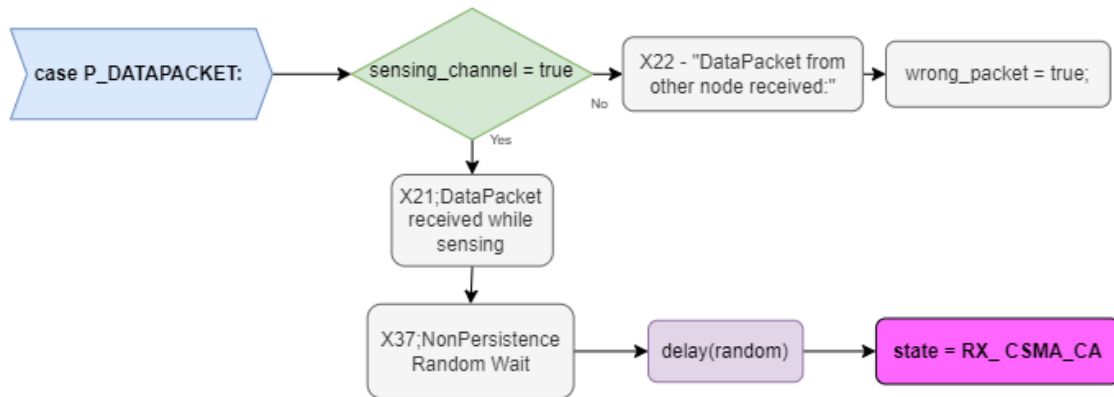


Fig. 3.41: Case where a Data Packet is received in the “*void OnRxDone()*” function in the ground nodes code.

If after sense the channel, no packet is received, it means that the channel is free. In that case, you can start the communication by sending the first RTS packet. Figure 3.42 shows the case where the sensing time has elapsed, and no package has been received. In that case, the *sensing_channel* boolean is set to false and the RTS boolean is set to true. The status is then changed to *TX_CSMA_CA* to send the RTS. The procedure for sending the RTS in the *TX_CSMA_CA* state can be seen in Figure 3.37. First, it is checked if the boolean *RTS* is true, given that it is just configured as true the following process is executed. First a waiting time determined by the DIFS is performed. Then the RTS packet is sent and the sent packet is recorded. Finally, three Booleans are configured and the time in which the packet has been sent is saved in the variable *tiempo_envio*. The first Boolean to configure is the *RTS*. This is set as false given that the package has already been sent. The second boolean to be configured is *expectingCTS*. This is set to true since the next packet expected to be received is the CTS. Finally, the third boolean that is configured is the *sensing_channel*. This is set as false given that the next step is to listen to the channel to receive the CTS, not to sense the medium.

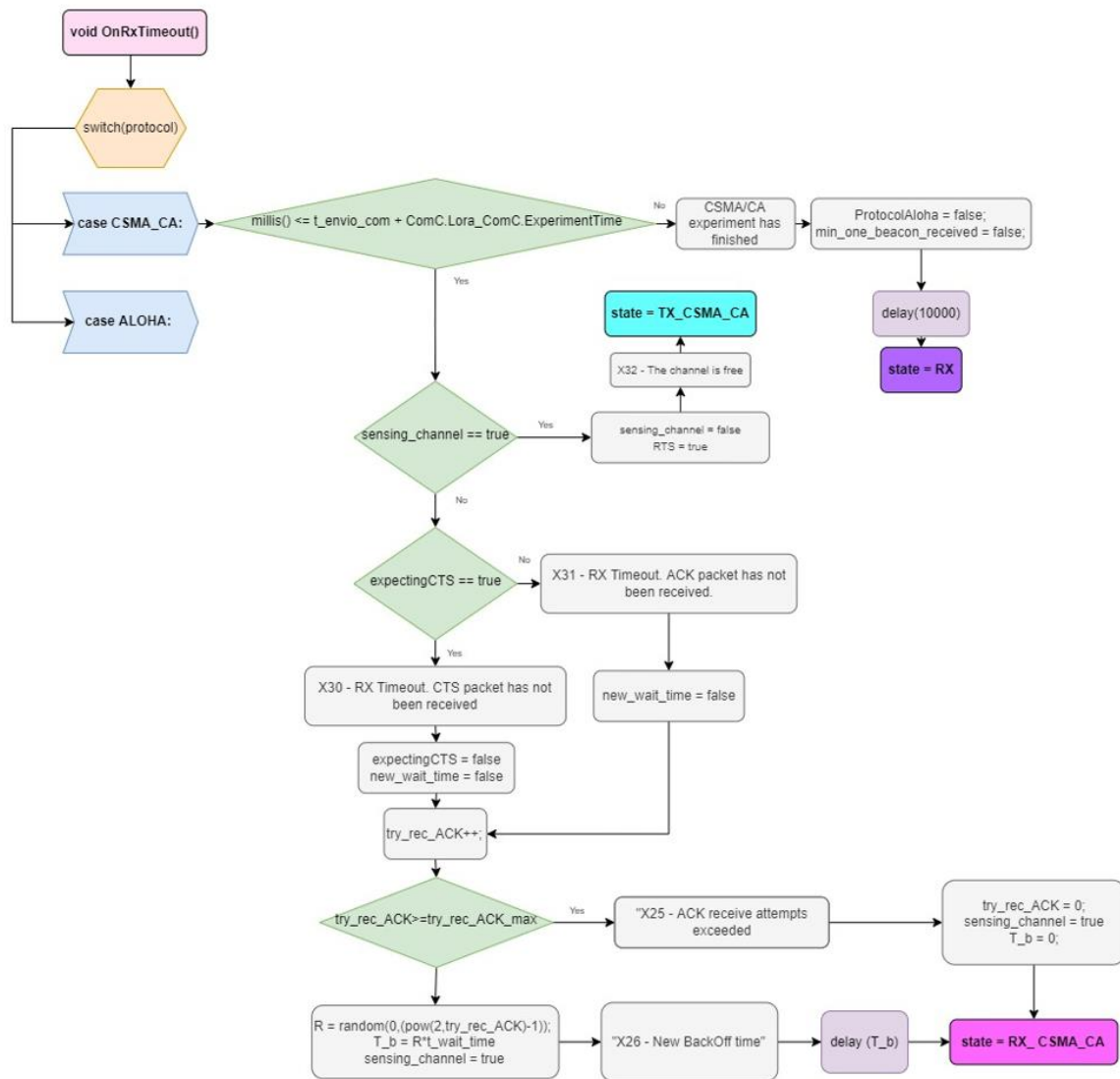


Fig. 3.42: “*void OnRxTimeout()*” function of the ground nodes code

In the drone, when an RTS packet is received, the next step is to send the CTS. However, several processes are performed before sending it. The schema of receiving an RTS packet in the drone can be seen in Figure 3.43. The first step is to check that the package is addressed to the drone by comparing the identifiers. The received RTS packet is then recorded and the status of the *waiting_data_packet* boolean is checked, which is false in this case. Then the CTS boolean is configured as true and the *TX_CSMA_CA* state is configured.

The program executes the “*void loop()*” function again as shown in Figure 3.35. After checking that the CTS Boolean is true, a waiting time determined by the SIFS is performed and then the CTS packet is sent in addition to configuring the *waiting_data_packet* Boolean as true. The program then executes the “*void OnTxDone()*” function as shown in figure 3.24. In this function, since the CTS is true, it sets the *RX* state. Again, figure 3.35 shows how this state is executed in the “*void loop()*” function. First, it is checked whether the boolean *new_wait_time*

is false. If it is false, the time at which the medium is started to listen is saved in the variable $t_start_wait_time$. The waiting time to receive the data packet is done given that the channel has been reserved for that communication. So, the drone expects to receive the data packet from the node. It will only wait for a time defined by the $wait_time$. It is possible that during this waiting time some RTS packet will be received, however the drone should ignore it and continue to wait for the data packet for the remaining time of the $wait_time$.

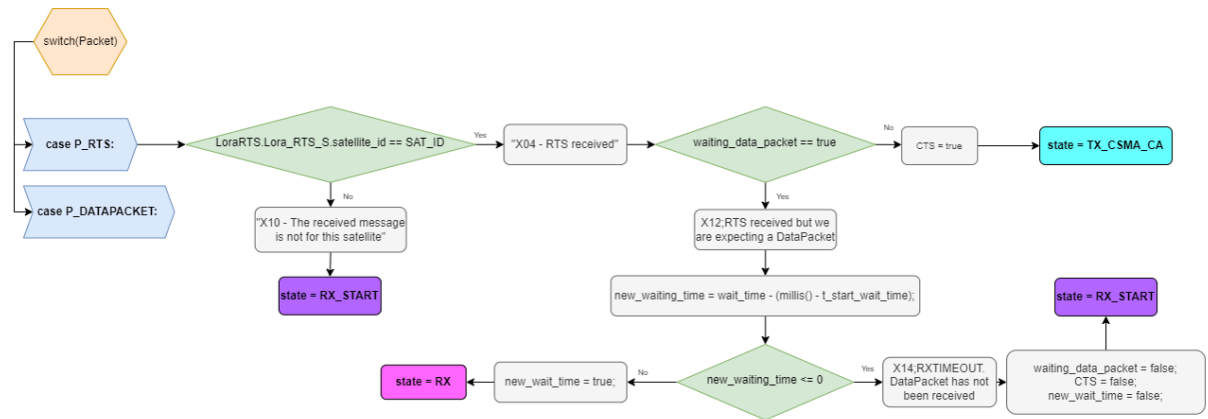


Fig. 3.43: Structure of the case where a RTS packet is received in the “void OnRxDone()” function of the drone payload.

In the ground nodes, after sending the RTS package five cases may occur:

The first case is that the RTS package does not reach the drone correctly. It could also collide with another RTS packet sent from another node because both nodes have detected the free channel. In this case the node timeout ends and the “void OnRxTimeout()” function is executed. As can be seen in Figure 3.42, the process to follow is as follows. First it is checked that the experiment time is not finished. It then checks whether the channel should be sensed (which is not the case). Next, the state of the *expectingCTS* boolean is checked, which has been set to true when sending the RTS. Therefore, the X30 message is printed saying that the CTS package has not been received after the waiting time. In addition, a reception attempt is added to the *try_rec_ACK* variable. Finally, the *backoff* process is performed if the number of ACK reception attempts has not been exceeded. Before setting the state to *RX_CSMA_CA*, the *sensing_channel* boolean is set to true.

The second case is that an RTS sent from another node is received. The process to follow if an RTS is received from another node can be seen in Figure 3.38. In this case, the first step is to identify whether the channel should be sensed, which is not true. After that, the package received is recorded and a new variable related to the waiting time is calculated, *new_waiting_time*. This variable calculates the remaining wait time after receiving a package other than the correct CTS. It is calculated using the following formula:

$$new_waiting_time = t_wait_time - (millis() - t_start_wait_time) \quad (3.2)$$

It is calculated as the subtraction between the total waiting time minus a second component. This second component is calculated as the time difference between the current time and the time when the waiting time was started. By performing this subtraction, we obtain the remaining time which we must continue to wait to receive the CTS. It is important to wait for the CTS, as it is the package that reserves the channel. Next, the *new_wait_time* boolean is configured as true. The result of *new_waiting_time* is then checked to be not less than zero or zero. If so, the *new_wait_time* boolean is set to false and the *wrong_packet* boolean to true. How this last boolean affects is explained below. If *new_waiting_time* is greater than 0, the state *RX_CSMA_CA* is set and the “*void loop()*” function is run again and the channel is heard during the time determined by the *new_waiting_time*.

The third case is that the node receive a CTS directed to another node. The process to follow if a CTS is sent to another node can be seen in Figure 3.39. In this case, the node ID is simply compared to the node ID contained within the CTS package and if they are not equal the *X16* message is printed and the *wrong_packet* boolean is set to true.

The fourth case is where the waiting time ends and no packets are received. In these cases the program executes the “*void OnRxTimeout()*” function as shown in Figure 3.42. In this case, after checking whether the time of the experiment has ended or if the channel should be felt, it is checked if the Boolean *expectingCTS* is true, which is true. In this case, a message is recorded that the CTS has not been received after the waiting time and *new_wait_time* and *expectingCTS* are configured as false. Subsequently, the *backoff* process is performed in addition to configuring the *sensing_channel* boolean as true. Finally, the *RX_CSMA_CA* state is configured to sense the channel and retry.

The fifth case is where the CTS packet sent by the drone is received correctly and corresponds to the node identifier. This case can be seen in Figure 3.39. In these cases, the received CTS packet is recorded and the *send_datapacket* boolean is configured as true. Finally, the *TX_CSMA_CA* state is configured to send the data packet to the drone. Figure 3.37 shows the process for sending the data packet. The first step is to wait a certain time by SIFS. The data packet is then sent and recorded. The time at which the packet was sent is then saved in the time variable *tiempo_envio*. Also, the following booleans are configured as false: *send_datapacket*, *expectingCTS*, *sensing_channel* and *new_wait_time*. Finally, the program executes the “*void OnTxDone()*” function which can be seen in Figure 3.18, and reconfigures the *RX_CSMA_CA* state to listen to the channel waiting to receive the ACK.

Again, in the drone, after sending the CTS and initiating the waiting time, three different cases can occur.

The first case is that the node receives an RTS sent from another node. The procedure can be seen in figure 3.43. In this procedure, the state of the

waiting_data_packet boolean is checked. Since it is true, the following process is executed. First, *new_waiting_time* is calculated as explained above. This time is then ensured not to be less than or equal to 0. Finally, the *new_wait_time* boolean is set to true and the state is changed to *RX*. The channel is then heard again during the time determined by *new_waiting_time*. For the remaining time, the data packet may be received.

In this function, the first thing to be checked is that the experiment time is not finished. Then, the *X13* message is printed, which informs that the data packet has not been received after the waiting time. The following booleans are configured as false: *waiting_data_packet*, *CTS*, *new_wait_time*. Since the data packet has not been received after the waiting time, the channel is no longer reserved, and the drone listens to the medium again waiting to receive RTS from other nodes.

The third case is where the data packet is received before the waiting time ends. The procedure to be performed in this case can be seen in the figure 3.44. First, it checks that the satellite's identifier is the same. The received data packet is then recorded and the following three Booleans are configured. The *waiting_data_packet* boolean and *CTS* are configured as false. The *ACK* boolean is configured as true, so that the ACK can be sent. Finally, the state is changed to *TX_CSMA_CA* and the "*void loop()*" function is executed again. Figure 3.35 shows the procedure for sending the ACK. As can be seen, before sending the ACK a waiting time determined by the *t_wait_time* is performed.

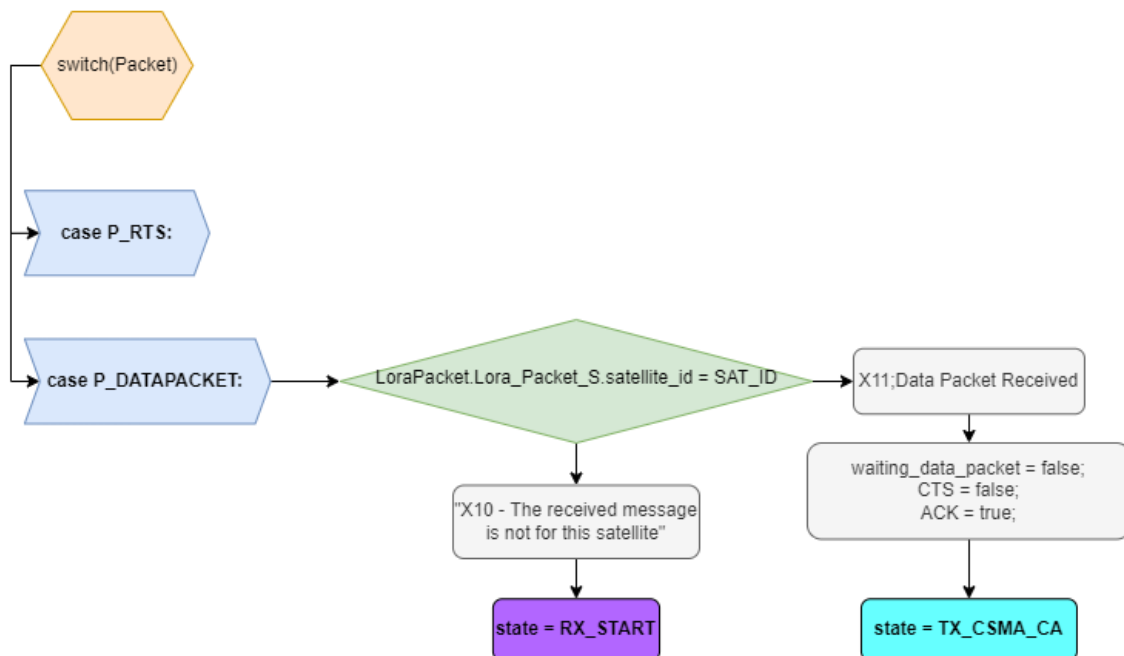


Fig 3.44: Structure of the case where a Data Packet is received in the *void OnRxDone()* function of the drone payload.

After sending the ACK, the ground node receives it. This case can be seen in Figure 3.40, where the steps to follow after receiving an ACK are detailed. As can be seen, after checking two Booleans (*sensing_channel* and *expectingCTS*), the identifiers of the ACK packet are compared with that of the node. If they match, the X20 message is printed, and various variables are configured. First, *try_rec_ACK* and *T_b* are set to zero, as it was in an initial state. Second, the *sensing_channel* and *wrong_packet* booleans are configured as true and false, respectively. Finally, the node has a waiting time equal to the *SIFS* before transmitting again. This is done to prevent the same node from occupying the channel just after receiving the ACK. Also, it is added the time (*t_to_next_packetC*) determined by the user that indicates the waiting time before restarting a communication after receiving the ACK. The *RX_CSMA_CA* state is configured and the "void loop()" function is run again to restart the process.

Finally, it remains to clarify what happens when a wrong packet is received and the Boolean *wrong_packet* is activated after passing through the Switch. This process is controlled by an "if", so it is only executed if the *wrong_packet* Boolean is true. This event can be seen when a data packet is received from another node. Or, when the node receives an ACK or CTS that doesn't match the node ID. Also, when the node receives an ACK and was expecting a CTS. Finally, it can also be given when the *new_waiting_time* calculation in the case of receiving an RTS gives 0 or less. In these cases, the boolean *wrong_packet* is set to true and the procedure in Figure 3.45 is executed. First, the Boolean *wrong_packet* is set as false again. Next, the timer variable is calculated. This variable is calculated as the subtraction between the node's current time minus the time the data packet was sent. If this result is not equal to or greater than the waiting time, it is calculated again until it is. Once the waiting time has elapsed, it is checked what type of package was expected. This is done by looking at the state of the *expectingCTS* boolean. If the boolean is false, it means that an ACK was expected. If true, the package expected to be received was a CTS. In both cases, the *backoff* process explained above is performed again. In this process, the *sensing_channel* boolean is configured as true. Finally, the state *RX_CSMA_CA* is configured, and the medium is sensed again.

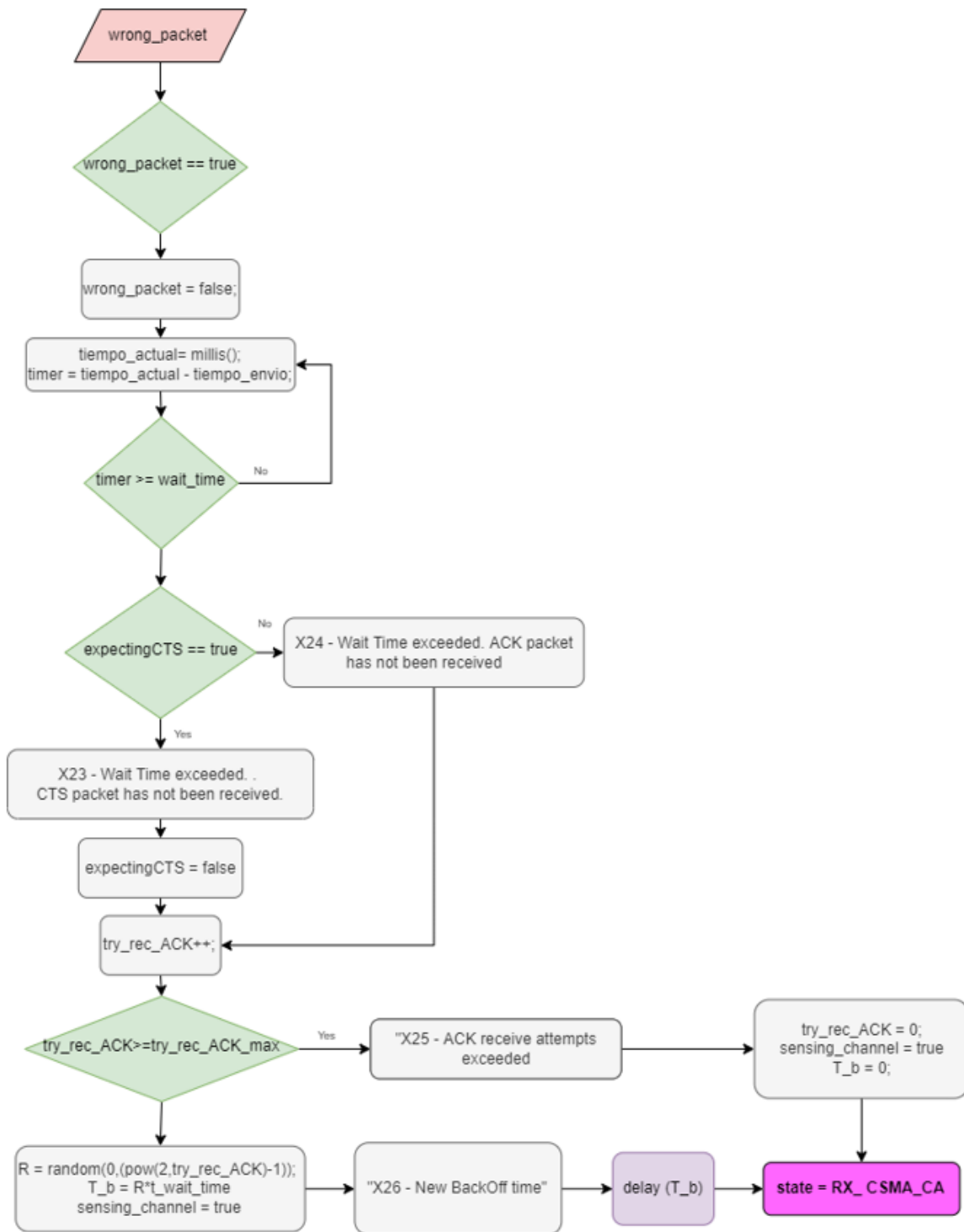


Fig. 3.45: Wrong packet case in the CSMA/CA protocol on the ground nodes code

3.2.3. Calculation of the adjustable parameters of both protocols

This section details the formulas and numerical values of each of the variables of both protocols. To do this, it is necessary to consider the formulas detailed above in paragraphs 2.4.1 and 2.4.2, where the pure ALOHA protocol and the CSMA/CA protocol were explained. Also, it is necessary to consider the LoRa Time On Air of each package sent for certain numerical calculations.

3.1.1.1. ALOHA protocol parameters

In the pure ALOHA protocol it is only necessary to calculate two values, the waiting time and the backoff time, which also depends on the waiting time. Here is how these two times are calculated:

$$T_{waitpacket} = 2 \cdot (T_{tx} + T_{prop} + T_{proc}) \quad (3.2)$$

$$T_{backoff} = T_{waitpacket} \cdot R \quad (3.3)$$

$$R = [0, \dots, 2^K - 1] \quad (3.4)$$

To calculate the waiting time in the pure ALOHA protocol ($T_{waitpacket}$), the maximum transmission time (T_{tx}), the maximum propagation time (T_{prop}) and the processing time of the transceiver (T_{proc}), in our case the CubeCell, must be taken into account. The $T_{waitpacket}$ is multiplied by two to account for the round trip of the package. When sending the package, must be considered the processing time to send it, the transmission time, and the propagation time. Once the packet reaches the destination, must be considered the processing time to send the packet back, the transmission time, and the propagation time. Therefore, it is multiplied by two, thus obtaining the waiting time once a package is sent.

The maximum propagation time (T_{prop}) is determined by the distance between the furthest node and the drone, divided by the propagation speed. In the case of the experiment, the nodes are not more than 400 meters apart from the starting point. If the drone raises until the maximum allowable distance (120 meters), the distance between the drone and the furthest node is 417,61 meters. The figure 3.46 shows the scenario.

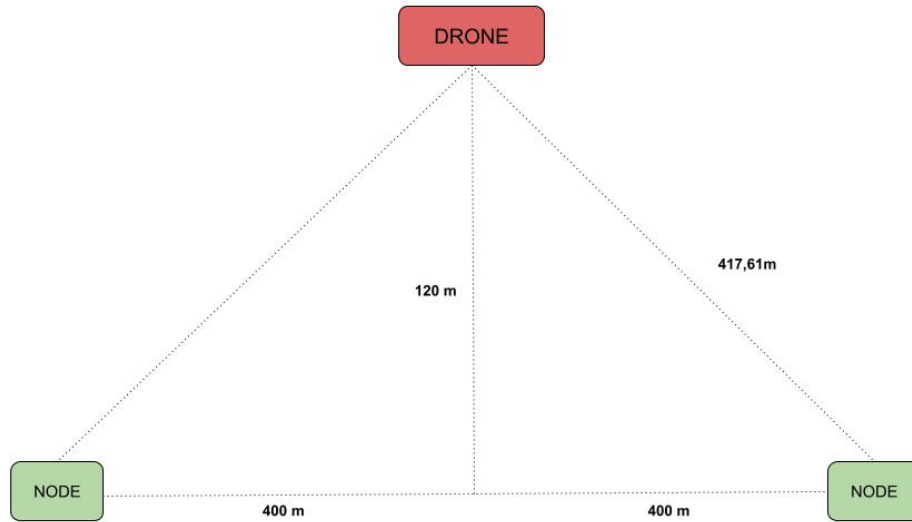


Fig. 3.46: Further location of nodes and drone in the experiment scenario

Considering that the maximum distance is 417,61 meters and the transmission speed is the same as the speed of light, the propagation time is 1,39 ms. Making approximations we establish that the T_{prop} is 1,4 ms for the calculation of the $T_{waitpacket}$.

The transmission time of the package is determined by the Time On Air. To calculate the time the package is transmitting, the LoraWAN airtime calculator is used [20]. To perform this calculation, it is necessary to enter four variables. The first variable is the size in bytes of the message to send, the second the spreading factor, the third the frequency band used, and finally the fourth is the bandwidth. The only variable that changes between the different types of packets to be sent is the number of bytes of each packet's content. The rest of the variables remain the same. In the spreading factor, as previously mentioned, an SF8 is used. In the frequency band of the region, the European (EU868) must be configured. Finally, the bandwidth is set to 125 KHz. In previous sections, we have been able to detail the content of the different types of packages. Specifically, the packets involved in the pure ALOHA protocol are only the Data Packet and the ACK packet. The Data Packet consists of 30 bytes, while the ACK package consists of 18 bytes. By inserting these values into the LoraWAN airtime calculator, we get that the ToA of the Data Packet is 164,4 ms and the ToA of the ACK is 133,6 ms. Therefore, the maximum transmission time (T_{tx}) for the pure ALOHA protocol is 164,4 ms.

Finally, to calculate the processing time (T_{proc}) of the CubeCell, different tests were performed to determine how long it takes to switch to the transmission state once a packet is received. In the vast majority of tests, the results obtained ranged between 5 and 10 ms. Therefore, being the most pessimistic case, the processing time of the transceiver is 10 ms.

Taking into account the numerical values of the different variables, the result of the $T_{waitpacket}$ is as follows:

$$T_{waitpacket} = 2 \cdot (164,4 \text{ ms} + 1,4 \text{ ms} + 10 \text{ ms}) = 351,6 \text{ ms} \quad (3.5)$$

On the other hand, the backoff time ($T_{backoff}$) is calculated as the multiplication of $T_{waitpacket}$ by R, where R is a random value between 0 and $2^K - 1$ (where K is the number of retries to receive the ACK).

3.1.1.2. CSMA/CA protocol parameters

In the CSMA/CA protocol it is necessary to perform several previous calculations before defining the value of the different variables involved. The different variables involved and calculated below are: $T_{waitpacket}$, $T_{backoff}$, $SIFS$, $DIFS$, NAV_{RTS} , NAV_{CTS} , $T_{waitsensing}$ and $T_{sensing}$

The waiting time is calculated in practically the same way as in the pure ALOHA protocol. The only difference of the waiting time in the CSMA/CA protocol is that the SIFS time must be added. The SIFS time is the timeout performed before sending any packets. Therefore, the formula for calculating the waiting time in the CSMA/CA protocol is as follows:

$$T_{waitpacket} = 2 \cdot (T_{tx} + T_{prop} + T_{proc}) + SIFS \quad (3.6)$$

Where the values of T_{tx} , T_{prop} and T_{proc} are calculated in the same way as before. The SIFS value is set by the maximum delay of a transmitted packet to reach the most distant node. This implies that SIFS is the sum of transmission time, propagation time, and processing time. All these variables have been calculated previously, so the value of SIFS is as follows:

$$\begin{aligned} SIFS &= T_{tx} + T_{prop} + T_{proc} \\ SIFS &= 1,4 \text{ ms} + 164,4 \text{ ms} + 10 \text{ ms} = 175,8 \text{ ms} \end{aligned} \quad (3.7)$$

Therefore, the $T_{waitpacket}$ value in the CSMA/CA protocol is:

$$T_{waitpacket} = 2 \cdot (164,4 \text{ ms} + 1,4 \text{ ms} + 10 \text{ ms}) + 175,8 \text{ ms} = 527,4 \text{ ms} \quad (3.8)$$

The way to calculate the backoff time ($T_{backoff}$) is the same as in the pure ALOHA protocol. The backoff time ($T_{backoff}$) is calculated as the multiplication of $T_{waitpacket}$ by R, where R is a random value between 0 and $2^n - 1$ (where n is

the number of retries to receive the ACK). The backoff time is also used to calculate the DIFS time value. This value is calculated as the sum of SIFS plus the backoff time:

$$DIFS = SIFS + T_{backoff} \quad (3.9)$$

How the NAV_{RTS} and NAV_{CTS} values are calculated is detailed below. The theoretical formulas of how to calculate these variables are shown in section 2.4.2.1. However, due to the limitations of the equipment, processing times in addition to transmission time should also be considered.

$$NAV_{RTS} = 3 \cdot (SIFS + T_{tx} + T_{proc}) + T_{propCTS} + T_{propDP} + T_{propACK} \quad (3.10)$$

$$NAV_{CTS} = 2 \cdot (SIFS + T_{tx} + T_{proc}) + T_{propDP} + T_{propACK} \quad (3.11)$$

The different transmission times are calculated again using the LoRaWAN airtime calculator. The following table compiles the different times:

Table 3.1: Time On Air of the different packets in the CSMA/CA protocol.

Type of packets		
Packet	Size (bytes)	Time On Air (ms)
Beacon	8	102,9
ACK	18	133,6
Data Packet	30	164,4
RTS	18	133,6
CTS	18	133,6

Therefore, the NAV_{RTS} and NAV_{CTS} values are calculated as follows:

$$NAV_{RTS} = 3 \cdot (175,8 + 1,4 + 10) + 133,6 + 164,4 + 133,6 = 993,2 \text{ ms} \quad (3.12)$$

$$NAV_{CTS} = 2 \cdot (175,8 + 1,4 + 10) + 164,4 + 133,6 = 672,4 \text{ ms} \quad (3.13)$$

Finally, it is explained how the time values related to the sense of the channel have been calculated ($T_{waitsensing}$ and $T_{sensing}$). The sensing type used in this protocol follows the nonpersistent method scheme. In this method, if the channel is busy, wait a random amount of time and sense the channel again. If the channel is free, it transmits immediately. This random time between sensing and sensing

($T_{waitsensing}$) has been defined as a random variable between one and two times the SIFS time. This random time is executed when an ACK or data packet is received while the channel is being sensed. So, the approximate time for the channel to be free again, hovers between one and two times the SIFS time.

$$T_{waitsensing} = [T_{tx} + T_{prop} + T_{proc}, \dots, 2 \cdot (T_{tx} + T_{prop} + T_{proc})] \quad (3.14)$$

$$T_{waitsensing} = [SIFS, \dots, 2 \cdot SIFS] = [175,8, \dots, 351,6] \text{ ms} \quad (3.15)$$

The time during which the channel is sensed is determined by the variable $T_{sensing}$. If the sense were instantaneous, it is highly likely that the node detected the free channel. This is because there are times when nothing is being transmitted. This is the cause of SIFS time between transmissions. That is why it should be considered a time large enough not to detect the free channel when it is not. The figure 3.47 shows the outline of the CSMA/CA communication showing the propagation times of each package.

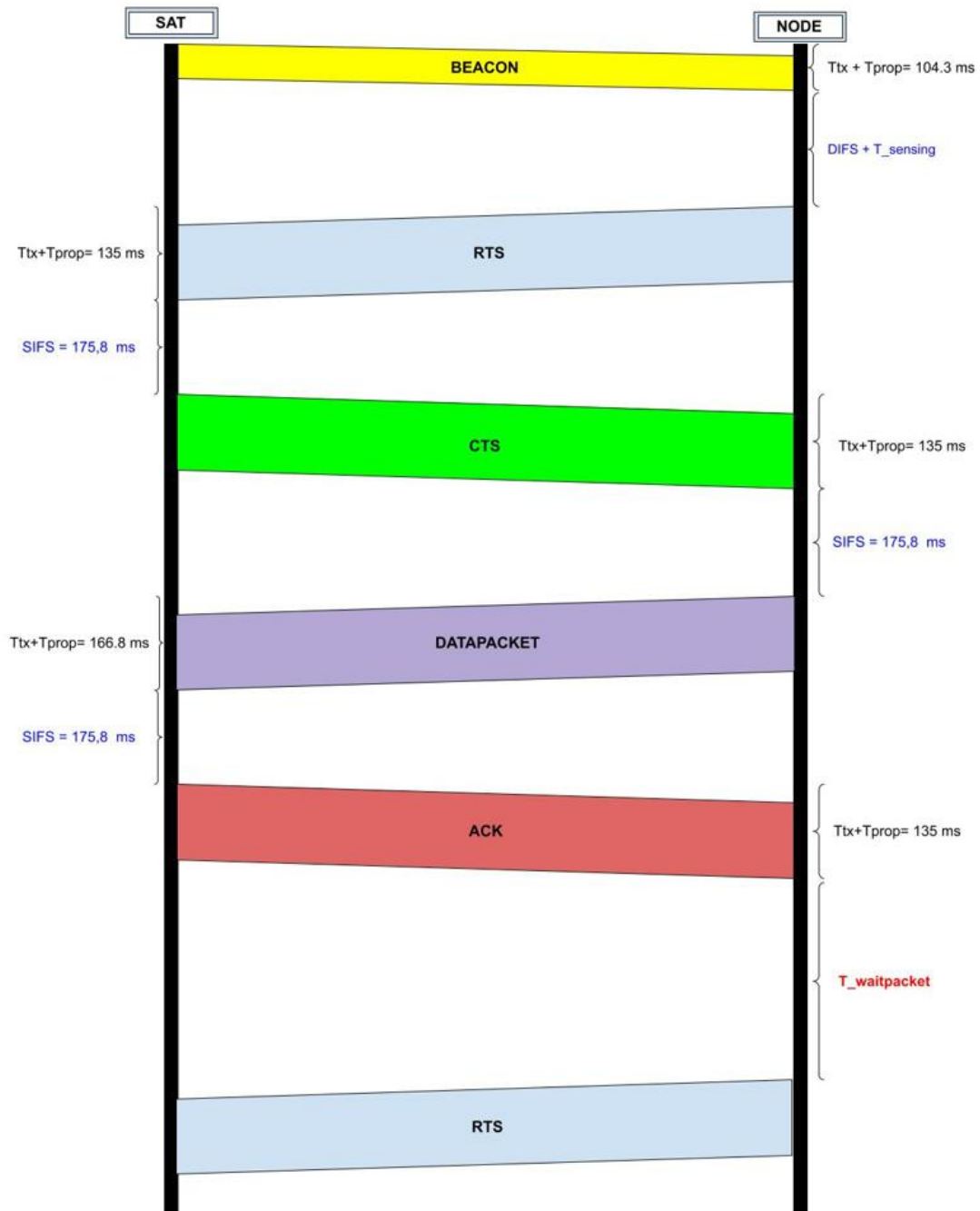


Fig. 3.47: Outline of the CSMA/CA communication

As can be seen, if one node senses the channel while another is silent waiting for SIFS to transmit, it will detect that the channel is free. To avoid this, it has been determined that the channel sensing time will be the same as the wait time. This is because the wait time is also the time between transmissions after receiving the ACK. Therefore, the sensing time is calculated as:

$$T_{sensing} = 2 \cdot (164.4 \text{ ms} + 1.4 \text{ ms} + 10 \text{ ms}) + 175.8 \text{ ms} = 527.4 \text{ ms} \quad (3.16)$$

3.3. Methodology applied in the hardware design

This section explains the methodology applied to develop the whole structure of the LoRa communications experiment proof-of-concept. To enable communications between IoT nodes and the drone, it has been necessary to develop the structure of several ground nodes with different devices. On the other hand, it has also been necessary to develop and design a drone-based miniaturized payload that hosts the hardware of this experiment and the GNSS-R experiment developed by other NanoSat Lab members.

3.3.1. Ground node design

This section details all the steps taken to realize the structure of IoT ground nodes. To carry out the experiment, the design and construction of 22 nodes were proposed. This number is determined by the number of CubeCells we have. We certainly had a total of 23 CubeCells. However, one of them is required to use for the drone payload, so it cannot be counted to generate another node. Due to hardware limitations which will be detailed later, only 13 nodes have been used.

The structure of each node is formed by a waterproof IP67 box, and inside are distributed the different devices. The main element is the CubeCell, which performs all communication. In the node are also two sensors which take temperature and humidity data. The data obtained by these sensors are stored in the content of the Data Packet and sent to the drone within the different communication processes. To record the different types of packets sent and received by the Cubecell, a RaspberryPi is used which reads the CubeCell UART. In addition, the RaspberryPi is used to power the CubeCell and moisture sensor. The other sensor is fed through the CubeCell, later the connections will be detailed. Finally, inside the node there is also a battery regulator and a battery to power the RaspberryPi.

The humidity and temperature sensors used are the "*Capacitive Soil Moisture Sensor v1.2*" sensor and *HDC1080* sensor, respectively. Below are some technical details of these sensors:

- **Capacitive Soil Moisture Sensor v1. 2:**

Capacitive Soil Moisture Sensor v1. 2 is an analog capacitive soil moisture sensor which measures the volumetric content of water inside the soil. The way to measure the different levels of soil moisture is done through the capacitive sensing. The capacitive sensing is based on measuring the capacitance between two electrodes inserted inside the ground, the capacitance between the electrodes will depend on the soil moisture, so for a very wet soil the capacitance will be very low and for a very dry soil the capacitance will be very high. To measure the differences between different types of terrain, capacitive sensors have a timer chip that is used to generate a square wave. This wave is modified according to the capacitance obtained. This difference in waves is compared by

the sensor, resulting in a small output differential voltage varying between 1.2V-3V that can be measured by a microprocessor. The CubeCell has an ADC pin, which will read the different voltages recorded and convert them to percentages. The "*Sensor calibration*" section details the procedure. It should be noted that the soil moisture sensor has an operating current of 5mA.

One of the advantages of using capacitive sensors is that they are made of a material that is resistant to corrosion. This offers superior service life compared to other types of sensors, such as resistive soil moisture sensors, which are made of corrosive material. Figure 3.48, shows the capacitive soil moisture sensor used in the experiment and the different pins it has:



Fig. 3.48: Capacitive Soil Moisture Sensor v1.2

As can be seen in the figure 3.48, the sensor has 3 pins: GND, VCC and AOUT. The first two are for feeding the sensor, the last one is the analog output pin. Concerning the power, according to the technical specifications, the operating voltage is between 3.3 and 5.5 VDC. This power would be sufficient if the original sensor chip timer (TLC555C), which has a minimum supply voltage of 2V, was used. However, because the manufacturer of the purchased sensors has used a different chip timer (NE555) to reduce structural costs, a minimum supply voltage of 4.5V is required.

Apart from having another timer chip compared to the original, the capacitive soil moisture sensor v1. 2 also have two other structural problems.

The first problem is related to the lack of the 662K voltage regulator. This component regulates any input voltage at 3.3V. Since the output voltage of the analog signal depends on the supply voltage, sensor measurements can be affected if the sensor is not powered by a constant voltage. If the 3.7-volt lithium-ion battery were used, we would have this problem. This is because the battery charge is discharged over time, causing the supply voltage to vary. This changing supply voltage would also confuse the output voltage of the sensor and thus the humidity readings. To solve this problem, many capacitor soil moisture sensors have a 662K voltage regulator on board. But some manufacturers have chosen to forget this regulator in order to save a few cents and have simply bridged the solder pads instead. The figure 3.49 shows the comparison of two sensors, one where the voltage regulator is correctly implemented and another where it is not:

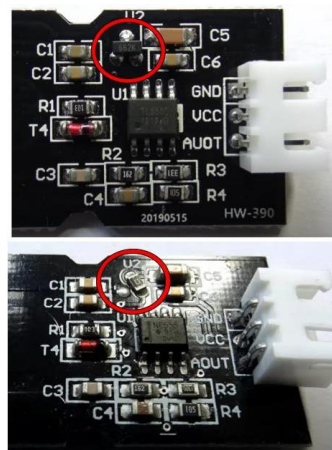


Fig. 3.49: Missing voltage regulator in the capacitive soil moisture sensor.

This problem can be solved if the humidity sensor is fed at a constant voltage. Therefore, the 5V pin of the Raspberry Pi is used to feed it as will be explained in the following sections.

The next problem is related to the lack of grounding connectivity of a $1\text{M}\Omega$ resistance. The figure 3.50 shows the schematic of the humidity sensor. The outline part of the circuit is used to convert the waveform signal from the sensor into a constant voltage that can be read by other hardware.

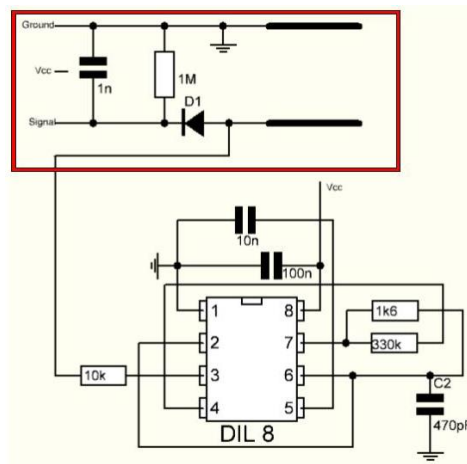


Fig. 3.50: capacitive soil moisture sensor schematic [24]

As can be seen, there is a $1\text{M}\Omega$ resistor that should be connected to the analog output on one side and to ground on the other. However, the check with the multimeter shows that this is not the case for the ground side. The reason why this connection is not being made is to locate a via hole where it should not be located. In the figure 3.51 it can be seen the different locations of the via hole.

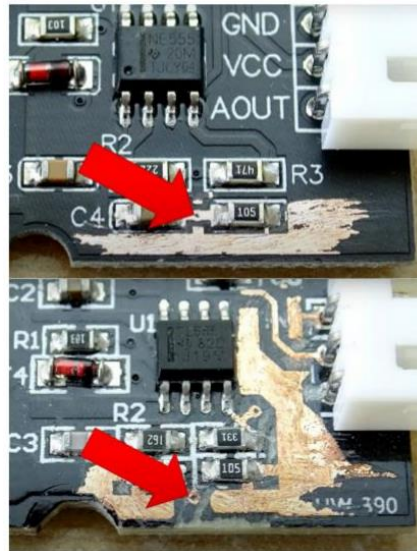


Fig. 3.51: Comparison of the different locations of the via hole in the capacitive soil moisture sensor

The image above corresponds to the sensor whose schematic is correct and works correctly. For the working sensor, the multimeter confirms the connection of the resistor to ground by the copper path. In the other hand, as can be seen in the image below, the same via hole is located a little further outside, which interrupts the connection of the 1MΩ resistor to ground. In practice, this means that the sensor becomes extremely unresponsive and the measured value changes only very slowly.

This problem can be solved quickly by welding a cable between the 1MΩ resistor and the ground pin of the connector. The following image shows how this problem should be solved:

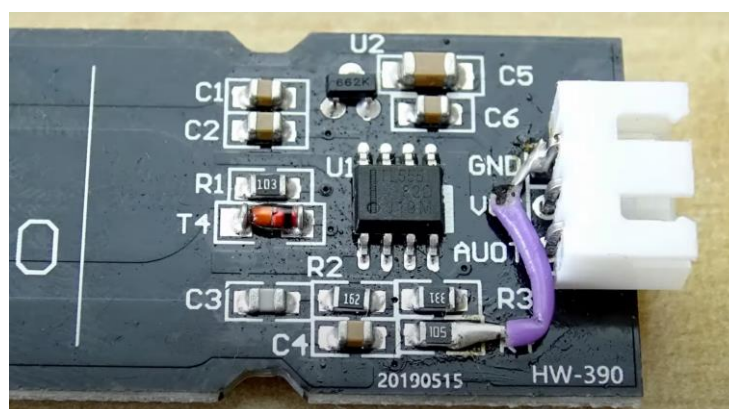


Fig. 3.52: Solution to the unresponsive problem in the capacitive soil moisture sensor

▪ HDC1080 Temperature sensor:

The HDC1080 is a digital temperature and humidity sensor with excellent accuracy and very low consumption. This sensor is compatible with Arduino/CubeCell thanks to the I2C communication protocol.

In this sensor, the only measurement that will be taken is the temperature. The most important characteristics are the following:

Table 3.2: Important characteristics of the HDC1080 sensor [25]

Feature	Value
Temperature accuracy	$\pm 0.2^{\circ}\text{C}$
Temperature resolution	11-bit and 14-bit
Sleep mode consumption	100nA
Consumption mode measuring	1.3 μA
Supply voltage	2.7V to 5V
Communication	I2C
Temperature range	-40°C to 125°C

As can be seen in the above table, the accuracy of the HDC1080 sensor is not uniform. So, it can vary by $\pm 0.2^{\circ}\text{C}$. The figure 3.53 shows the accuracy range as a function of temperature:

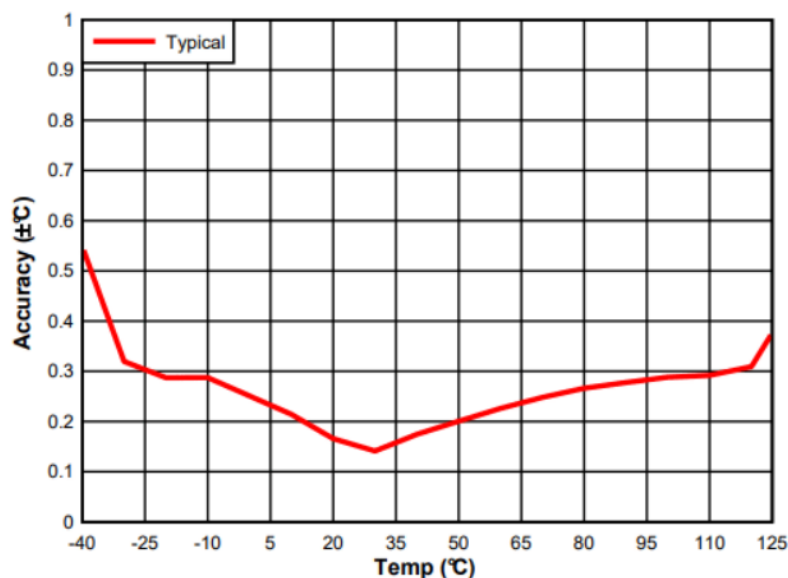


Fig. 3.53: Temperature Accuracy vs. Temperature [25]

Depending on the temperature of the region where IoT ground nodes are located, the temperature accuracy may vary. However, the experiment to be performed will be done in an environment with a standard temperature (25-30 °C), so the accuracy of the measurements will be quite accurate.

One of the most interesting features of the HDC1080 sensor is its low consumption. The sensor is able to operate with low energy consumption thanks to its two operating modes: sleep mode and measuring mode. When the sensor turns on, it automatically enters sleep mode consuming an average of 100nA. In this mode, the sensor waits for any command that comes through the I2C protocol to wake up. When it receives a command to perform a measurement, it switches from sleep mode to measuring mode. Once the measurement is completed, the sensor returns to sleep mode. Thanks to all this done automatically, very low consumption is achieved. This makes it a very suitable device when low energy consumption is required, as is the case with IoT projects.

Related to consumption, it is also important to look at the response times that the sensor has. The response time determines how long it takes to take the measurement and return to low-power sleep mode. Response times depend on the resolution the sensor is working with and the ADC clock cycles. The following table shows the response times of the sensor to obtain the temperature measurement according to the ADC resolution:

Table 3.3: Conversion time in function of the resolution of the HDC1080 sensor [25]

Resolution	Time (ms)
11-bit	3,65 ms
14-bit	6,35 ms

The longer the sensor response time, the higher the consumption. When the sensor is measuring, the consumption is 1300nA, thirteen times more than in sleep mode. So resolution is a parameter to consider in IoT nodes that require low power consumption.

Finally, the following image shows the electrical diagram of the HDC1080 sensor:



Fig. 3.54: HDC1080 sensor

As can be seen in the figure 3.54, the sensor consists of 4 pins. The connection between the sensor and the CubeCell is explained later in the section “*Connection between devices*”.

- GND: reference to 0V
- SCL: clock signal I2C
- SDA: data signal I2C
- VCC: supply voltage (between 2.7V and 5.5V)

After explaining and detailing some of the technical details of the humidity sensor and temperature sensor, below is a picture of the first proposal of the ground node. It shows all the elements already integrated:

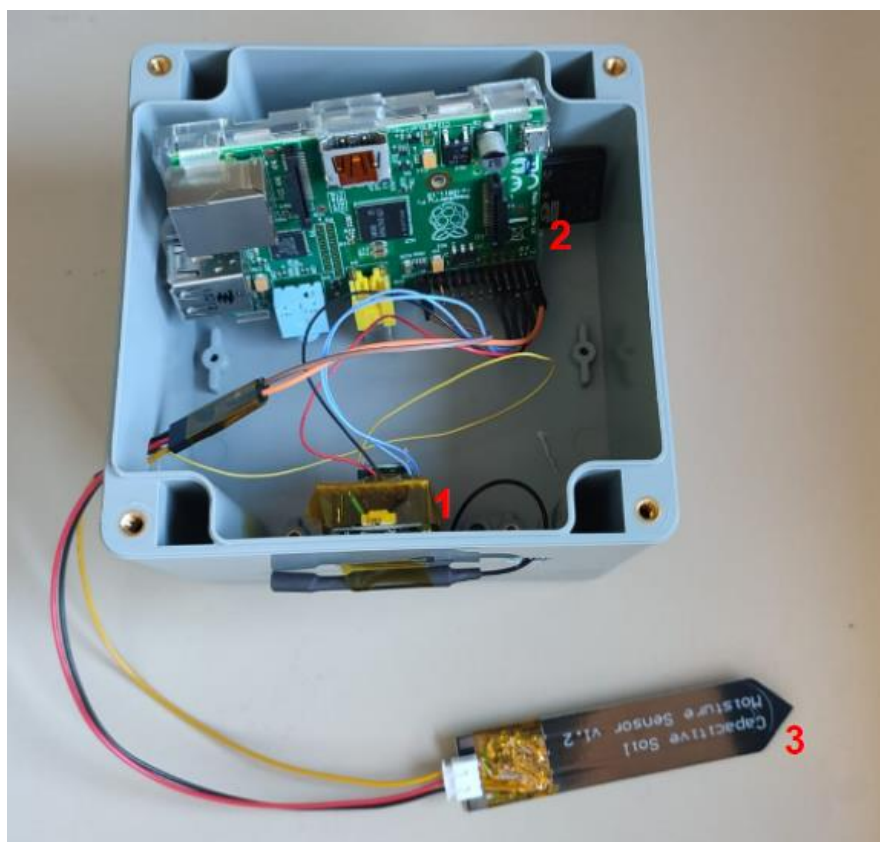


Fig. 3.55: First design proposed for the ground node

In the above image, three groups of elements can be seen. The first one is the CubeCell with the antenna located outside the box to improve the link budget. The CubeCell has an integrated stripboard to make the different connections without damaging the pins of the device. In addition, the temperature sensor is located on top of the integrated CubeCell stripboard. Secondly, we can see the Raspberry Pi with the SD and the various connections. The battery regulator and battery are located behind the Raspberry Pi. Finally, the third device seen in Figure 3.55 is the capacitive soil moisture sensor. This sensor is designed to be

nailed to the ground where the measurement is to be taken. For this reason, it is necessary to make a hole in the box to be able to pass the sensor and the wiring.

The final design of the ground node has slight modifications compared to Figure 3.55. One of the modifications that has most affected the initial design has been the way of recording the CubeCell data. Initially, it was designed to be recorded in the SD of the Raspberry Pi. However, the reading of the data through the Raspberry Pi presented many losses and there were certain limitations with some of the models used. Finally, the data is read through the CubeCell-PC connection, where they are recorded on the PC by reading the CubeCell UART. The details of these changes are explained later in the "connection between devices" section and the final model of the ground node is presented.

3.3.1.1. CubeCell modifications required

The most important equipment of the ground nodes is the CubeCell HTCC-AB01, which is part of the "CubeCell" series. It is a wireless communication card based on the ASR605x chip that integrates an MCU with a LoRa module. Without this equipment communications would not be possible. However, proper communication is not the only objective of the work. The CubeCell must communicate with the different sensors and read the data they provide. In order to make this possible, CubeCell hardware had to be modified.

First, the CubeCell Pinout Diagram HTCC-AB01 is shown in Figure 3.56:

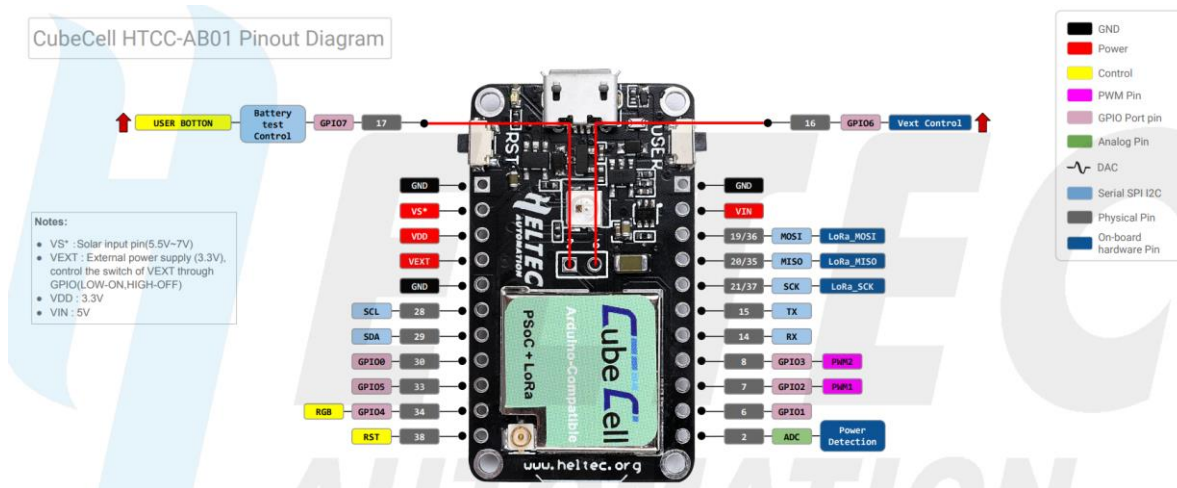


Fig. 3.56: CubeCell HTCC-AB01 Pinout Diagram [26]

The ASR6501 chip has only one ADC pin input. This pin is used by default for battery voltage reading. In the following image we can see the schematic of the chip AO7801, which contains the head of the ADC hooked to the D1.

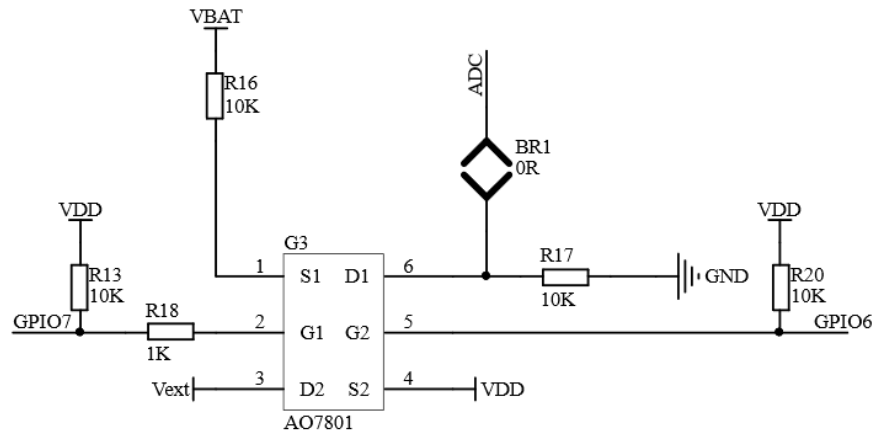


Fig. 3.57: Schematic of the AO7801 chip [26]

In order to read other analog signals, it is necessary to remove the BR1 resistance, so the ADC header would be free. The following image shows that resistance is the one that must be extracted to enable ADC.

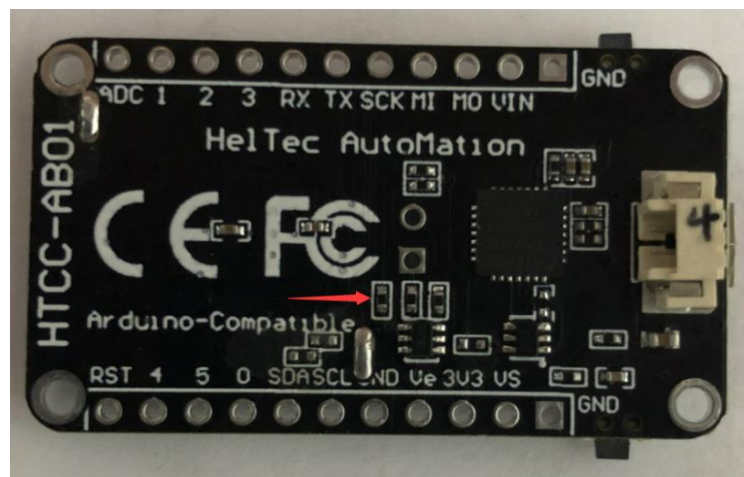


Fig. 3.58: Resistance BR01 to remove [26]

By extracting this resistance, it is possible to make the analog reading from the ADC. However, a problem arises with the maximum voltage that can be entered by the ADC pin. The CubeCell ADC has an internal reference voltage of 1,2V. This means that the ADC input cannot exceed 2,4V due to electrical specifications. Previously, we have been able to detail that the analog output of the humidity sensor is between 1,2 and 3V. Therefore, it is not possible to connect the analog output of the humidity sensor to the CubeCell ADC pin. That is why an external voltage divider has been implemented in the integrated stripboard into the CubeCell that is seen in the next section.

3.3.1.2. Design of the stripboard integrated into the CubeCell

This section explains the design of the stripboard integrated into CubeCell. The reason why an additional element was designed to integrate it into the CubeCell is to protect the equipment and facilitate the process of connecting the different devices between them. In the model design, the HDC1080 sensor is integrated. The HDC1080 sensor needs certain connections with the CubeCell, so having it fixed and anchored to it prevents further wiring. In addition, it is also necessary to include two resistors in order to make a voltage divider that reduces the maximum voltage offered by the capacitive soil moisture sensor. The value of these two resistors depends on the maximum analog output value. According to specifications, the maximum analog output voltage is 3V. However, this value depends on the supply voltage. In the final model, the sensors are powered with 5V via the Raspberry Pi. To determine the maximum analog output voltage ($V_{AO_{max}}$), we used the 13 sensors to be used in the measurement campaign and measured their maximum analog output when powered at 5V. The following results were obtained:

Table 3.4: Maximum analog output voltage ($V_{AO_{max}}$) of the capacitive soil moisture sensors

Number of capacitive soil moisture sensor	Maximum analog output voltage ($V_{AO_{max}}$)
1	4,11
2	4,13
3	4,15
4	4,08
5	4,03
6	4,06
7	4,12
8	4,07
9	4,04
10	4,05
11	4,11
12	4,08
13	4,15

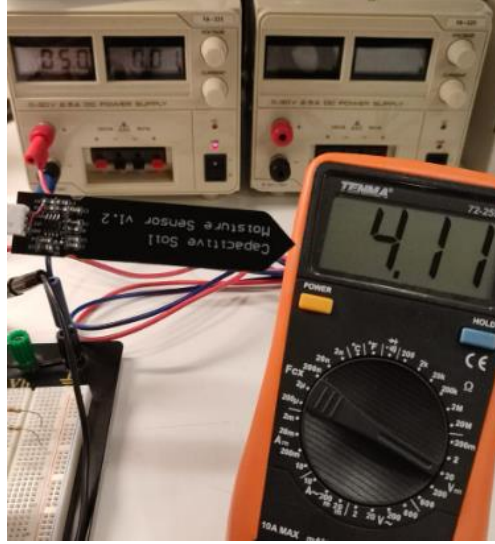


Fig. 3.59: Experimental $V_{AO_{max}}$ measurements of the capacitive soil moisture sensor

Therefore, the maximum analog output value is 4,15V. The voltage value should be reduced using a voltage divider because the maximum input value of the CubeCell ADC pin is 2,4V. We use the following expression to find two commercial resistances that meet the criteria:

$$V_{out} = \frac{R_2}{R_1 + R_2} \cdot V_{in} \quad (3.17)$$

Whereas $V_{out} = 2,4V$ and $V_{in} = 4,15V$, the expression is as follows:

$$\frac{V_{out}}{V_{in}} = \frac{R_2}{R_1 + R_2} \rightarrow \frac{V_{out}}{V_{in}} (R_1 + R_2) = R_2 \quad (3.18)$$

$$\frac{V_{out}}{V_{in}} R_1 + \frac{V_{out}}{V_{in}} R_2 = R_2 \rightarrow \frac{V_{out}}{V_{in}} R_1 = R_2 \left(1 - \frac{V_{out}}{V_{in}}\right) \quad (3.19)$$

$$R_1 = \frac{R_2 \left(1 - \frac{V_{out}}{V_{in}}\right)}{\frac{V_{out}}{V_{in}}} = \frac{R_2 \left(1 - \frac{2,4}{4,15}\right)}{\frac{2,4}{4,15}} = \frac{35 \cdot R_2}{48} = 0,729 \cdot R_2 \quad (3.20)$$

So, $R_1 = 0,729 \cdot R_2$. Considering the commercial resistance values, $R_2 = 51K\Omega$ is chosen:

$$R_1 = 0,729 \cdot 51K\Omega = 37187,5\Omega \quad (3.21)$$

The closest commercial value is $39K\Omega$. Taking these two values into account, the result of V_{out} is:

$$V_{out} = \frac{51K\Omega}{39K\Omega + 51K\Omega} \cdot 4,15V = 2,35V \quad (3.22)$$

Value practically adjusted to the maximum 2,4V that allows the ADC pin. Adjusting it to the maximum voltage is necessary to not lose ADC dynamic range margin. After determining the values of the resistors to perform the voltage divider, the stripboard integrated to the CubeCell is explained below.

The final model of the integrated CubeCell board was designed with a stripboard and has all the necessary elements. It has integrated the HDC1080 sensor and the two previously determined commercial resistors. To make the required 13 models, many members of the NanoSat Lab helped the process of welding the tracks and welding the different elements and cables. The final result is shown in Figure 3.60 below:

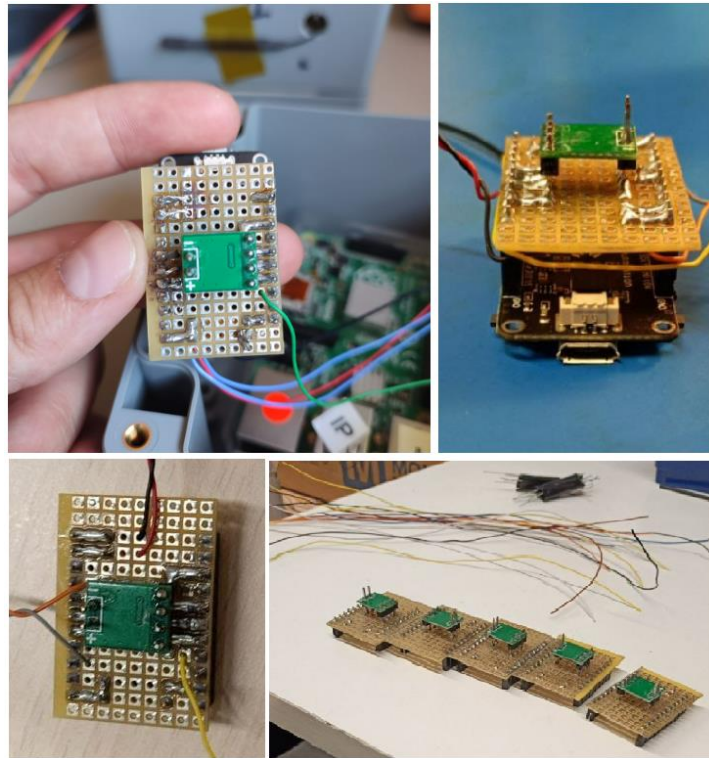


Fig. 3.60: Final model of the stripboard integrated into the CubeCell

As can be seen in the image above, the HDC1080 sensor remains on the outside of the stripboard, while the two resistors of the voltage divider remain on the inside. The different connections are as follows:

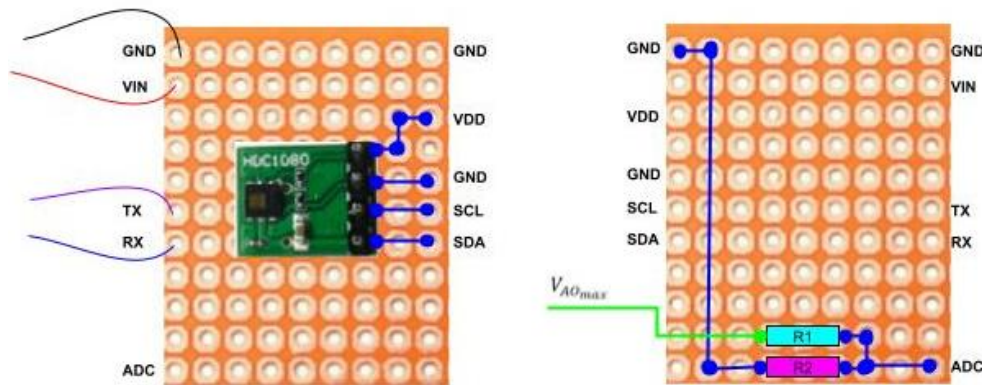


Fig. 3.61: Connections of the stripboard integrated into the CubeCell

The HDC1080 sensor is located on the outside of the board, which has 4 pins: VDD, GND, SCL and SCA. In image 3.61 it can be seen how the CubeCell pins have been written on the edges of the stripboard. To connect the HDC1080 sensor simply connect these 4 pins to the VDD, GND, SCL and SDA pins of the CubeCell. The GND and VIN pins have two cables which are used to power the CubeCell via the Raspberry Pi. The serial port TX and RX cables are also directed to the Raspberry Pi. On the inside of the stripboard, you can see the voltage divider, where $R_1 = 39K\Omega$ and $R_2 = 51K\Omega$. The green cable that reaches R1 is directly connected to the analog output of the capacitive soil moisture sensor.

3.3.1.3. Sensor calibration

This section explains the procedure performed to configure the HDC1080 sensor, calibrate the capacitive soil moisture sensor, and the code implemented to make the sensors read.

The HDC180 sensor does not require calibration as it is a digital sensor and is ready to operate when it is connected. However, it is necessary to include a library in the code (`#include "HDC1080.h"`) and initialize it in the `"void setup()"` function (`hdc1080.begin(0x40)`).

To calibrate the capacitive soil moisture sensor, it is necessary to measure the CubeCell ADC analog signal in two scenarios. The first scenario is where the sensor is in a dry environment. To do this, the measurement is taken while the sensor is in the air without anything being in contact with it. The second scenario is where the sensor is in a wet environment. For this, an environment where humidity is 100%, such as water, was chosen. The sensor was placed in a glass of water, protecting the circuit part. The measurements were taken with the help of CubeCell. A program that does the analog reading of the ADC (`analogRead(ADC)`) was loaded. The data obtained by the sensors was then recorded. For a dry environment the measurement of the sensor was 3250, while

for a wet environment it was 1900. These two values have been saved within the ground node code as two *const int*: *AirValue* and *WaterValue*. However, the result we want to obtain is the percentage of soil moisture. For that reason, the following code was implemented:

```
//Reading of the Temperature sensor - HDC1080
SensorTemperature = hdc1080.readTemperature();
LoraPacket.Lora_Packet_S.temperature = SensorTemperature;

//Reading of the Soil Moisture sensor
soilMoistureValue = analogRead(ADC);
soilmoisturepercent = map(soilMoistureValue, AirValue, WaterValue, 0, 100);
if(soilmoisturepercent >= 100){
    LoraPacket.Lora_Packet_S.soilmoisture=100;
}
else if(soilmoisturepercent <=0){
    LoraPacket.Lora_Packet_S.soilmoisture=0;
}
else if(soilmoisturepercent >0 && soilmoisturepercent < 100){
    LoraPacket.Lora_Packet_S.soilmoisture = soilmoisturepercent;
}
```

Fig. 3.62: Code for reading soil moisture and temperature sensors

As can be seen in the figure 3.62, the first line is for reading the temperature sensor. In the second, simply save this information within the content of the Data Packet to send. From the fourth line, the procedure to make the reading of the capacitive soil moisture sensor is observed. First, the ADC is read as previously discussed. Then, the percentage of the measurement is obtained using the "*map()*" function of Arduino, which Re-maps a number from one range to another. The data entered are those from *AirValue* and *WaterValue*, which were obtained in the calibration process. Finally, the measurement is checked to be within the range and saved within the content of the Data Packet.

3.3.1.4. Connections between devices

This section details the different limitations encountered, the connections between the different devices, and shows the final model of the IoT ground node. Initially, the experiment was intended to have a total of 22 independent nodes, each located at a different site. However, due to hardware limitations, it has not been possible to perform that number of nodes. The variations or limitations in the hardware that have been found are the following three:

Firstly, we have IP67 boxes of different sizes, so it will not be possible to make all nodes independent. In some nodes, given the size of the IP67 box, the equipment has been located in duplicate to optimize space. On the other hand, given the limited number of IP67 boxes available, space was only available for 18 CubeCells. So, the first limitation leaves us with only 18 ground nodes.

Second, the greatest limitation and variation is related to how CubeCell data is captured. In an initial scenario, CubeCell data would be captured by Raspberry Pi using UART communication. From the CubeCell all this information would be sent through the TX serial port and in the Raspberry Pi would be captured through the RX serial port. An added limitation is that we only had 15 Raspberry Pi with different models. To test the different models of Raspberry Pi that we had different tasks were done. The first one was to flash all the SDs that we had using the *Raspberry Pi Imager* program by selecting the *RASPBERRY PI OS LITE (32-BIT)* operating system. An OS without *Graphical User Interface (GUI)* was chosen in order to optimize the programming process of each Raspberry Pi. The next task was to program a code to read the UART of the Raspberry Pi automatically when it was switched on. In this way, it would not be necessary to load the code manually on all Raspberry Pi every time the port was read. In this process, it was important to enable serial communication of the Raspberry Pi to receive the messages and determine on which serial port the data was received (*ttyAMA0* or *ttyS0*). It was also important to set the baud rate of communication. The third and final task was to test the reading of the 15 Raspberry Pi with the CubeCell connected.

The results of this process showed that 8 out of the 13 Raspberry Pi were either recording corrupted data on the SD card, losing data or not recording at all. In some, corrupted data and significant data loss were obtained. Meanwhile, in Raspberry Pi model 3 there were large data losses. This was checked by looking at the files where the data was stored. The following image shows the case where there is corrupt data:

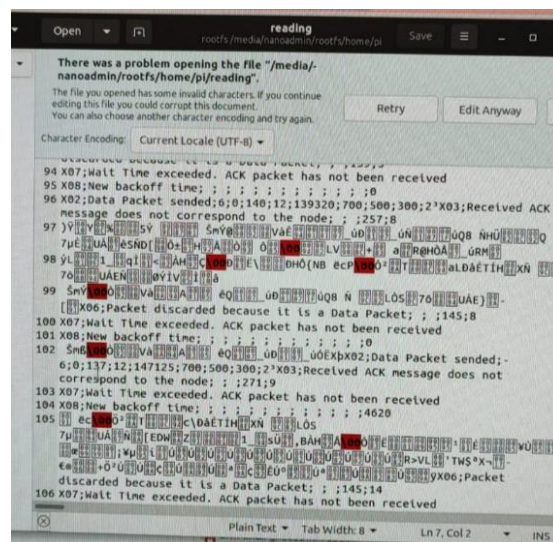


Fig. 3.63: Corrupt data and data loss of the SD reading

On devices that failed to make a correct reading of the SD, serial ports were changed. After making these changes and re-testing, no changes were obtained. The rest of the nodes worked correctly, except for the Raspberry Pi model 3 which occasionally had large data losses that were not tolerable. Noting that there were

only 5 nodes that saved the data correctly, it was decided to change the structure of how the CubeCell data was going to be captured.

The change caused the CubeCells data to be read using serial communication between the computer and CubeCell. In this way, simply with a USB to micro-USB cable, the reading could be done through the program "CoolTerm". In this program, it is only necessary to configure the baud rate and the COM port. On the other hand, if the reading was done from a computer that had Linux, it was only necessary to read the port *ttyAMA0* or *ttyS0* and save the data in a file to be able to process it. With these changes, all 15 nodes were made to work correctly.

Third, there is the limitation of battery regulators available. There are only 13 battery regulators next to the 3.7V LiPo battery. The following image shows both elements joined together with *kapton*:

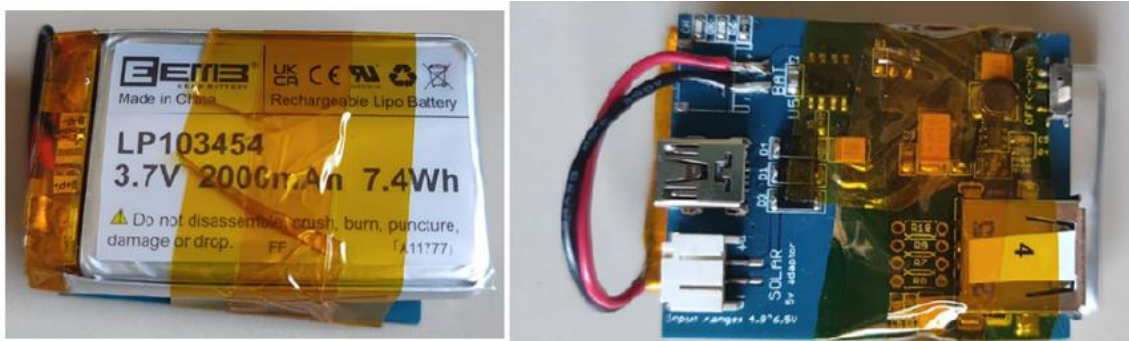


Fig. 3.64: Battery regulator with the 3,7 V LiPo battery

Therefore, after all the limitations mentioned above, it was only possible to make 13 nodes. The process of connecting and mounting between the different devices and in the different IP67 boxes is detailed below.

Different types of cables and methods have been used to connect the different equipment. Firstly, the connection of the LiPo battery with the battery controller was made by welding the cables directly as can be seen in the figure 3.64. This has been done since the battery regulator adapter was not the same as the battery connector. The connection between the battery controller and the Raspberry Pi was then made via a USB to micro-USB cable. Subsequently, the Raspberry Pi powers the capacitive soil moisture sensor at 5V, on the other hand, it is connected to the ground of the CubeCell to reference the same ground in the system. The analog output of the soil moisture sensor is the input of the "ADC Dynamic Range Adjustment" implemented in the stripboard. The output of this last module ends in the ADC pin of the CubeCell. On the other hand, there is the connection of the temperature sensor HDC1080, which is in the stripboard integrated to the CubeCell. Finally, the CubeCell is connected to the computer to read the data using a USB to micro-USB cable. The figure 3.65 shows the connection scheme between the different devices:

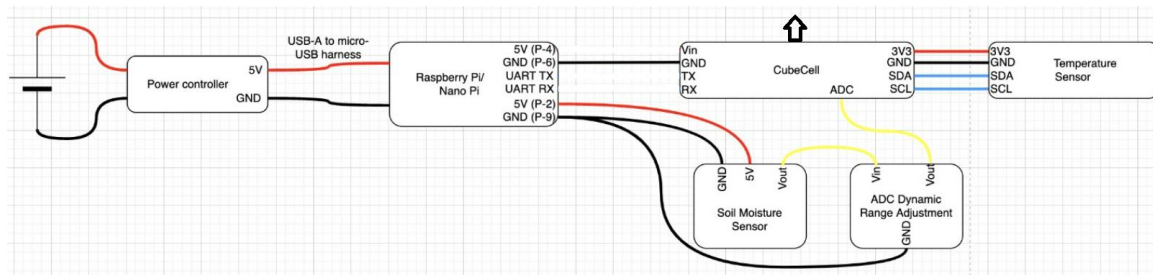


Fig. 3.65: Connections between different devices.

These connections had to be made for the different nodes in different IP67 boxes. Some elements within the IP67 boxes were subject with *kapton*, the CubeCell being one of them. The distribution of the elements was also important due to the limited space on the individual nodes. In addition, the CubeCell's USB micro harness was positioned in such a way that it was facing upwards so that the micro-USB cable could be easily connected. The following image shows on the left all the nodes with the different IP67 boxes that we had. An individual node is shown at the top right of the image. At the bottom right is an individual node closed and with the USB outbound to connect to the computer.



Fig. 3.66: Assembly process of the different nodes

3.3.2. Drone-based miniaturized payload design for LoRa communications

This section explains the drone-based miniaturized payload used for the experiment. This payload is designed to be easily integrated into the drone used for the experiment. For the measurement campaign experiment, the *Drone Condor*, a drone from the company *MDrone*, is used. Initially, however, the drone to be used was the *3D Robotics Iris+* drone, a drone belonging to NanoSat Lab. The reasons why the *3D Robotics Iris+* drone was not used for the measurement campaign are explained below. Subsequently, the design of the payload used in the *Drone Condor* is explained.

Initially, the *3D Robotics Iris+* drone was the drone to be used for the LoRa communication proof-of-concept experiment. Different payload designs were realized and tested. All of these designs included space for the GNSS-R experiment and the LoRa communications experiment. However, this drone was not used for the measurement campaign. The main reason is that this drone is not prepared to carry a payload for long periods. It was purchased in 2014 and was only designed to carry a small camera. Several tests showed that it was able to lift payloads of 800g for a limited time, but the autonomy was reduced. The autonomy of the drone considering the camera and supports is 15 minutes [27]. When tested with higher loads, the autonomy was reduced to 8-10 minutes, which is insufficient time for LoRa communications tests. In addition, in order to reduce the weight of the drone, the supports were replaced by hollow steel bars, which reduced the overall weight. Finally, it was tested in June during the GNSS-R measurement campaign with a payload reaching 650g in weight. After 8 minutes of flight, the drone overheated due to the overload and high temperatures. This caused the loss of control and the drone fell, damaging the support structure and part of the payload. After this experiment, it was decided to hire the services offered by *MDrone* who fly drones for different tests and experiments. By using a drone with better characteristics carried by professionals, the safety of the equipment is ensured, and more durable tests are guaranteed. The figure 3.67 shows the *3D Robotics Iris+* drone with the payload designed for previous experiments.



Fig. 3.67: Drone 3D Robotics Iris+ with the payload design

On the other hand, the *MDrone Condor Drone* is a much more powerful and versatile drone, capable of carrying up to 5kg. It has an autonomy of 85 minutes of flight without load and 25 minutes with a 5 Kg load. The payload designed for this experiment includes equipment from two experiments, the GNSS-R experiment and the LoRa communications experiment. Given that in a realistic case, both equipment must operate simultaneously in RITA, it is appropriate to test them together in the same test and check that there is no interference between the two experiments.

The necessary equipment to install on the drone is: a DC-DC converter, a CubeCell, a Raspberry Pi and a monopole antenna. The DC-DC converter is used to power the Raspberry Pi at 5V from the 12V of the drone batteries with an XT60 cable. The CubeCell is the module that handles all communications. This uses a monopole antenna with better characteristics than those used in the ground nodes in order to improve the downlink properties. Finally, the Raspberry Pi is used to capture the data from the CubeCell through the serial port as explained in the previous section. In this case, a Raspberry Pi Zero was chosen, which has smaller dimensions than those used in the ground nodes. Both the DC-DC converter and the Raspberry Pi Zero are shared elements of the two experiments. The DC-DC is shared because without it it would not be possible to power the Raspberry Pi. On the other hand, the Raspberry Pi runs two codes separately and stores the data in different files on the same SD. The Raspberry Pi runs the code for the GNSS-R experiment and another code to capture the data through the TX serial port of the CubeCell. Figure 3.68 shows the connection diagram of the different devices involved:

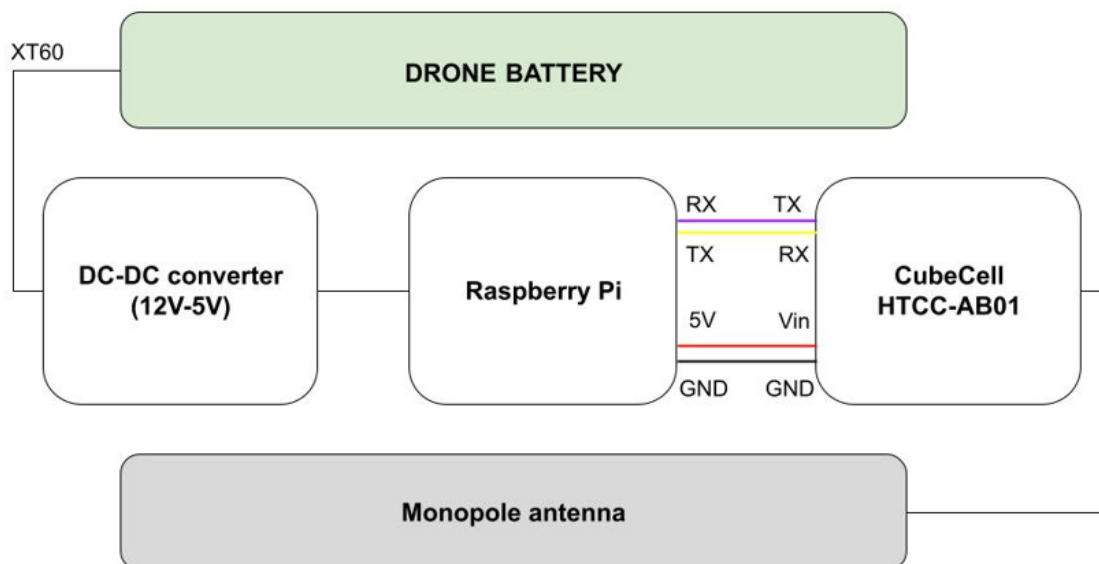


Fig. 3.68: Connection diagram of the different devices in the payload design

In addition to the devices discussed above, the GNSS-R experiment requires other elements. The final design of the payload using 3D printing with the equipment already assembled is shown below:

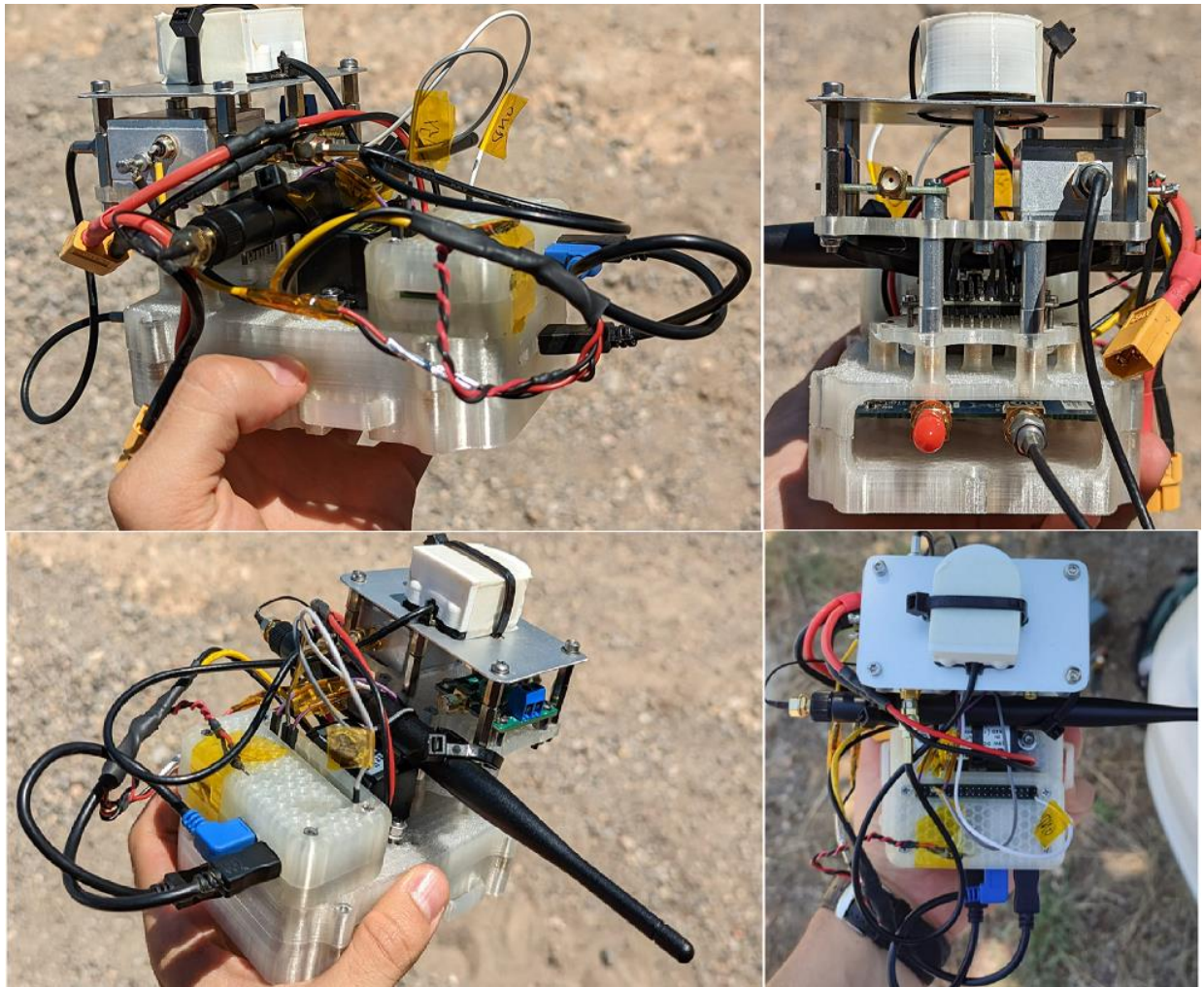


Fig. 3.69: Miniaturized drone-based payload for LoRa communications and GNSS-R

CHAPTER 4: First Measurement Campaign

The objectives and results of the first measurement campaign are detailed below. In this campaign, LoRa communications were tested using two CubeCell modules. In addition, the correct operation of the "Capacitive Soil Moisture V1.2" sensor was also tested by taking different measurements of the soil moisture.

4.1. First Measurement Campaign

The first measurement campaign was performed on the 26th of May 2022. In this first measurement campaign, there were two objectives. First, to check the correct functioning of the LoRa communications using two CubeCell modules. Secondly, to take measurements of the capacitive soil moisture sensor and analyze the results.

The first objective was successfully achieved. A different code was programmed into each device for communications between the two devices. The first CubeCell was placed inside an IP67 box powered with 5 V through a power bank. In addition, the structure of the IP67 box had two holes. One of them was to extract the CubeCell antenna and the other one was to extract the humidity sensor in order to take measurements. This first CubeCell was programmed as a transmitter. Its function was to collect the measurements from the humidity sensor and store them inside the package to be sent. It then sent the packet at a rate of 1 second. In the scenario of this experiment, there is only one transmitter, so there is no collision between the packets sent. The second CubeCell was connected to the computer. This CubeCell was programmed as a receiver, so ideally it received all the packets sent by the transmitter. Afterward, they were stored using the CoolTerm program, which reads the COM port at the indicated baud rate. The figure 4.1 shows both the receiver and the transmitter of this experiment.

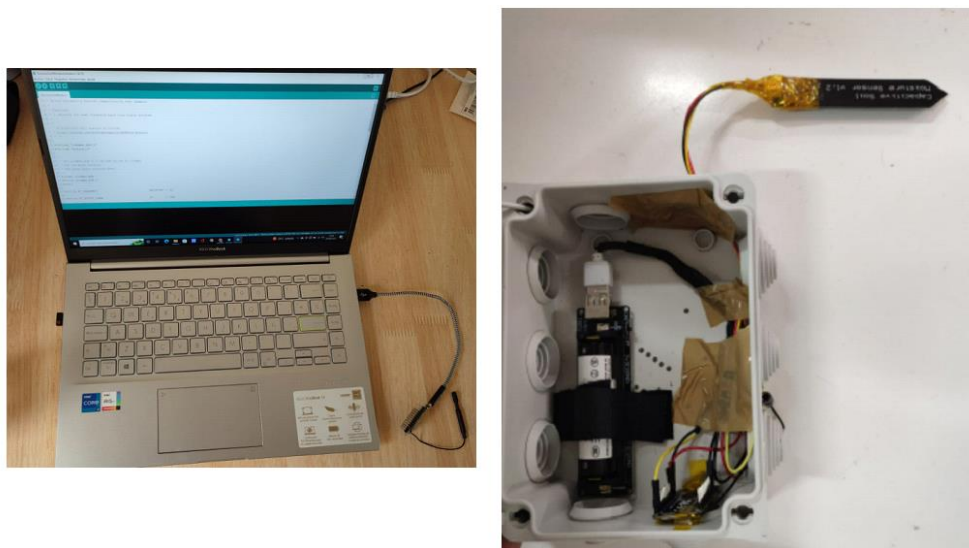




Fig. 4.1: Receiver and transmitter of the first experiment.

Second, to achieve the second objective, several measurements were taken with the capacitive soil moisture sensor. As can be seen in Figure 4.1, there are several wires covered with insulating tape. Under this material is the voltage divider to reduce the voltage of the analog output of the moisture sensor to 2,4V. In addition, the USB cable coming from the power bank was bifurcated into two, in order to power both the CubeCell and the humidity sensor. In this experiment, measurements were taken on different terrains to then check if the results were consistent. The following section shows the results obtained.

4.2. Analysis of the results of the experiment

In this experiment, different soil moisture measurements were taken in different soils. Below are some of the data captured in different soils. Each photo on the right represents the soil condition of the measurement. As can be seen in Table 4.1, the different measurements and soil states are shown gradually, from more sandy to more clayey.

Table 4.1: Soil moisture measurements

Soil moisture measurement	Soil conditions
<p>received packet "The humidity is:31.0" with rssi -114 , length 20</p> <p>received packet "The humidity is:30.0" with rssi -115 , length 20</p> <p>received packet "The humidity is:31.0" with rssi -115 , length 20</p> <p>received packet "The humidity is:32.0" with rssi -115 , length 20</p> <p>received packet "The humidity is:31.0" with rssi -115 , length 20</p> <p>received packet "The humidity is:31.0" with rssi -115 , length 20</p> <p>As we can see, the soil moisture is approximately 31%. This type of soil is sandy.</p>	
<p>received packet "The humidity is:40.0" with rssi -106 , length 20</p> <p>received packet "The humidity is:39.0" with rssi -109 , length 20</p> <p>received packet "The humidity is:40.0" with rssi -104 , length 20</p> <p>received packet "The humidity is:40.0" with rssi -106 , length 20</p> <p>received packet "The humidity is:40.0" with rssi -105 , length 20</p> <p>received packet "The humidity is:40.0" with rssi -106 , length 20</p> <p>As we can see, the soil moisture is about 40%. This type of soil is sandy loam soil.</p>	




<div>received packet "The humidity is:66.0" with rssi -112 , length 20</div> <div>received packet "The humidity is:66.0" with rssi -110 , length 20</div> <div>received packet "The humidity is:66.0" with rssi -114 , length 20</div> <div>received packet "The humidity is:65.0" with rssi -110 , length 20</div> <div>received packet "The humidity is:66.0" with rssi -109 , length 20</div> <div>received packet "The humidity is:66.0" with rssi -115 , length 20</div> <div>As we can see, the soil moisture is approximately 66%. This type of soil is loam soil.</div>	
<div>received packet "The humidity is:81.0" with rssi -114 , length 20</div> <div>received packet "The humidity is:80.0" with rssi -115 , length 20</div> <div>received packet "The humidity is:81.0" with rssi -114 , length 20</div> <div>received packet "The humidity is:82.0" with rssi -114 , length 20</div> <div>received packet "The humidity is:82.0" with rssi -112 , length 20</div> <div>received packet "The humidity is:81.0" with rssi -114 , length 20</div> <div>As we can see, the soil moisture is about 81%. This type of soil is loam - clay soil.</div>	
<div>received packet "The humidity is:92.0" with rssi -114 , length 20</div> <div>received packet "The humidity is:91.0" with rssi -115 , length 20</div> <div>received packet "The humidity is:92.0" with rssi -114 , length 20</div> <div>received packet "The humidity is:91.0" with rssi -114 , length 20</div> <div>received packet "The humidity is:91.0" with rssi -115 , length 20</div> <div>received packet "The humidity is:92.0" with rssi -114 , length 20</div> <div>As we can see, the soil moisture is approximately 91%. This type of soil is clayey soil.</div>	

Table 4.1 shows the measurements taken in the different soils. As can be seen, the measurements are not accurate since they give values of 30% moisture for sandy soil and 90% moisture for clay soil. The lack of precision is due to the way the sensors are calibrated. This problem can be solved by recalibrating the sensors taking into account the different soils and expected soil moisture values. However, the measurements provided are consistent and logical since the moisture data match the state of the soil. When the soil is arid and sandy, it shows low moisture, while when the soil is wet and clayey, it shows high moisture. The table 4.2 shows the available moisture values according to soil texture.

Table 4.2: Realistic soil moisture values [28]

Soil moisture values	
Soil texture	Soil moisture value
Sandy	9%
Sandy - Loam	23%
Loam	34%
Loam - Clayey	30%
Clayey	38%
Clayey with good structure	50%

The values in Table 4.2 show the realistic soil moisture values. If the sensors have been properly calibrated and the soil moisture has been correctly measured, the values given in Table 4.1 should correspond to these values. However, the values obtained in the experiment are far from these values, so that it is necessary to re-calibrate the sensors in order to be able to make good measurements in the second measurement campaign.

CHAPTER 5: Second Measurement Campaign

In Chapter 5, the second measurement campaign is explained. In this measurement campaign, the LoRa communications proof-of-concept experiment was performed with several IoT ground nodes and the miniaturized drone-based payload. In order to show the results, several plots have been elaborated showing the performance of each of the experiments in function of their properties. To extract these graphs, it has been necessary to process the information collected from the individual nodes. Since the data collected are separated by ";" and by rows, the data processing has been lighter since Excel has been used to separate the data and filters have been used to count the types of packets sent or received.

The first section explains how the measurement campaign was managed. This first section explains the site chosen for the experiment, the distribution of the nodes, the assembly and flight path of the drone, and the specifications of the different experiments performed. The second section analyzes the results obtained by the sensors. The third section contains all the results obtained from the experiments using the pure ALOHA protocol. On the other hand, the fourth section collects all the results obtained from the experiments using the CSMA/CA protocol. Finally, the fifth section analyzes and compares both experiments.

5.1. Second Measurement Campaign Specifications

The second measurement campaign was conducted on August 4th. In this second measurement campaign, the LoRa communications proof-of-concept experiment was performed with several IoT ground nodes and the miniaturized drone-based payload. In this measurement campaign, communications were performed using the different protocols implemented: pure ALOHA and CSMA/CA. The objective of this campaign is to capture data from the different nodes and the drone payload and then process the performance of each of the protocols.

Through the commands sent by a CubeCell, the type of experiment to be performed is controlled with the different configurable parameters: *ExperimentTime*, *TimeNextPacket* and *T_BEACON*. In addition, the values of the different protocol parameters are also sent. However, these are not modified at any time since they have been previously calculated.

The *ExperimentTime* determines the duration of the experiment, in most cases the experiments do not last more than 10 min. On the other hand, the *TimeNextPacket* determines the waiting time between each new communication attempt after receiving the ACK.

5.1.1. Location chosen for the experiment

The place chosen to perform the experiment is a field of vineyards far from the center of Barcelona. Specifically, it was carried out in Vilafranca del Penedès, near the "*La Torreta de Castellví*". Different permits were requested to fly the drone in this field. Firstly, the permissions managed by the company *MDrone* to raise the drone to a certain height. Secondly, the permission to occupy the vineyard field of a known farmer. The location where the experiment is conducted is important as there are different terrain textures, so the GNSS-R experiment could measure variations. As for the LoRa communications experiment, the diversity of ground textures is also interesting to take different measurements with the moisture sensors. Since it is performed in a vineyard field, there are certain regions that are wetter than others. Below is the map where the measurement campaign was carried out:

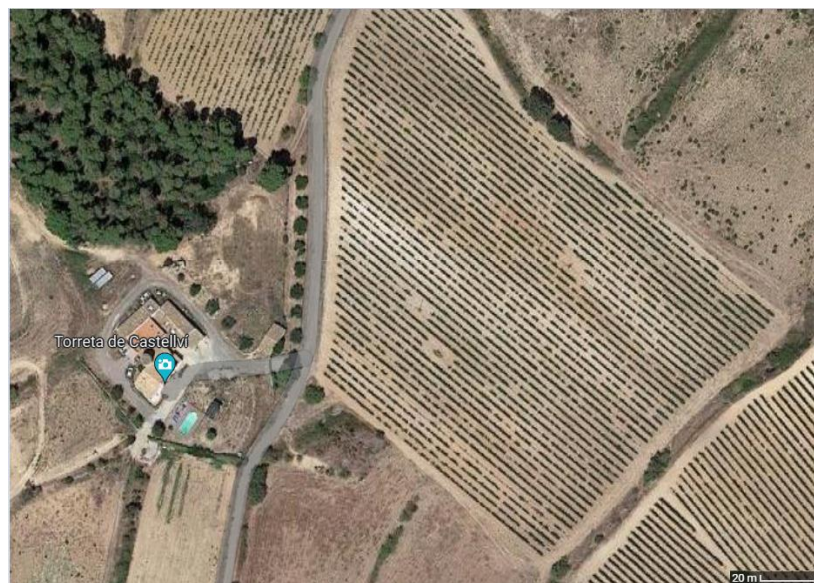


Fig. 5.1: Map of the scenario where the measurement campaign is performed

5.1.2. Specifications of the different experiments performed

In this section, the different experiments performed are specified. The measurement campaign was organized to perform different experiments with configurable variables. The first variable of the experiment is the determination of the type of MAC protocol to be executed. The second variable to be configured is the time of the experiment. Given that we have limited time since the LoRa experiment is not the only one performed during the measurement campaign, the times of the different experiments were less than 10 minutes. Nevertheless, it is enough time to collect information and process it.

A total of 6 experiments were performed, which were executed in three rounds. In each round, two different experiments were executed using firstly the pure

ALOHA protocol and secondly the CSMA/CA protocol. In all experiments, the beacon sending periodicity was set to 80 seconds. Since the experiments were not going to last more than 10 minutes, it was decided to reduce the retransmission time, previously set to 8 minutes. From the experiments performed, data were taken from 4 of them to analyze and compare data. The specifications of each experiment can be seen in Figure 5.2.

ALOHA					
Experiment Number	Time to send next packet	Beacon	Exp Time	Nodes	Number of Nodes
EXP 1	15 ms	80 s	13,4 min	2, 4, 5, 6, 8, 9, 10, 12, 13, 14, 15	11
ALOHA					
Experiment Number	Time to send next packet	Beacon	Exp Time	Nodes	Number of Nodes
EXP 3	15 s	80 s	7,93 min	2, 4, 6, 9, 10, 12, 13, 15	8
CSMA/CA					
Experiment Number	Time to send next packet	Beacon	Exp Time	Nodes	Number of Nodes
EXP 4	15 s	80 s	6,82 min	1, 4, 8, 9, 14, 15	6
CSMA/CA					
Experiment Number	Time to send next packet	Beacon	Exp Time	Nodes	Number of Nodes
EXP 6	1 ms	80 s	3,6 min	1, 4, 8, 9, 14, 15	6

Fig. 5.2: Different characteristics of the experiments

As can be seen, the " $t_{to_next_packet}$ " varies between the different experiments. In the first ALOHA experiment the $t_{to_next_packet}$ is 15 ms. This is a minimum wait, so the channel gets saturated. In the second ALOHA experiment, the $t_{to_next_packet}$ is 15 seconds, so the channel is not as saturated as in the previous case. The same happens with the CSMA/CA experiments. In the first experiment, the $t_{to_next_packet}$ is 15 seconds, while in the second experiment it is 1 ms.

5.1.3. Distribution and location of the ground nodes

As explained in previous sections, the ground nodes were designed to be connected to the computer to capture the data from the CubeCell. Since most computers only have 3 USB ports, it was necessary to use several USB HUBs for the computers brought for the experiment. During the experiment, 5 NanoSat Lab teammates helped to manage the position and control of the nodes. They were divided into three groups. Group 1, controlled nodes 2, 5, 6, 8, 9, 10, 12, 14 and 15 during the first experiment. Group 3 controlled nodes 1 and 7 in addition to managing the GNSS-R experiment. Finally, group 2 consisted only of myself. In my case, I controlled node 4 and controlled the CubeCell which sent the

commands to the rest of the nodes and the drone. Figure 5.2 shows how the different groups are separated by approximately 80 meters.

For the rest of the experiments, the distribution of nodes by clusters varied. For experiment 3, there were two clusters of nodes. The first cluster was formed by nodes 2, 4, 9, 10, 12 and 13. The second cluster was only formed by node 6. Finally, for experiments 4 and 6, the same node density and the following node clusters were used. The first cluster is formed by nodes 1, 8, 9, 14 and 15. The second cluster is formed only by node 4.

To initiate communications the command must be sent from an intermediate point in order for the command to be received by all nodes and the drone. For this reason, it is necessary to be located in an intermediate point between both groups. The departure point of the drone is where group 3 is located because it is necessary to configure the Raspberry Pi for the GNSS-R experiment before the beginning of the flight.

The figure 5.3 shows where the different groups were located with their respective nodes:



Fig. 5.3: Location of the different groups with the respective nodes

5.1.4. Assembly of the payload on the drone and flight path

The *Drone Condor* has an adaptable and flexible system to place different types of payloads of different sizes and weights very easily. The lower design of the payload includes different holes through which four M3 screws can be anchored. Also has lateral anchors through which flanges can be passed to securely fasten the payload. The figure 5.4 shows the drone with the payload assembled.



Fig. 5.4: Drone Condor with assembled payload.

Below is one of the flight paths of the experimental drone. The flight path was controlled by a pilot authorized by *MDrone*. The drone was flown at an altitude of 120 m and performed a flight path defined by the GNSS-R experiment. The flight path is based on traversing the vineyard field perpendicularly as shown in Figure 5.5. For the LoRa communications experiment, the path taken by the drone is not significant, since the communication with the different nodes is performed in the same way regardless of the path taken.



Fig. 5.5: Flight path of the experiment

5.2. Sensor data results

This section analyzes the results obtained by the sensors in the multiple experiments. To take the measurements of the sensors, the capacitive soil moisture sensor needs to be properly connected to the Raspberry Pi to be powered and to be able to extract results once it is placed in the soil. On the other hand, the temperature sensor obtains results when the CubeCell is powered, so this sensor is always operational. In experiments 1 and 2 measurements of the capacitive soil moisture sensors were taken, while in the rest of the experiments these devices were not nailed in the soil, so the measurements have no value. The following graphs show the values obtained from the capacitive humidity sensors and temperature sensors in experiment 1. It should be noted that node 6 in experiment 1 was not located in the same place because group 2 was moving during the experiment. That is why node 6 is not taken into account for the following plot.

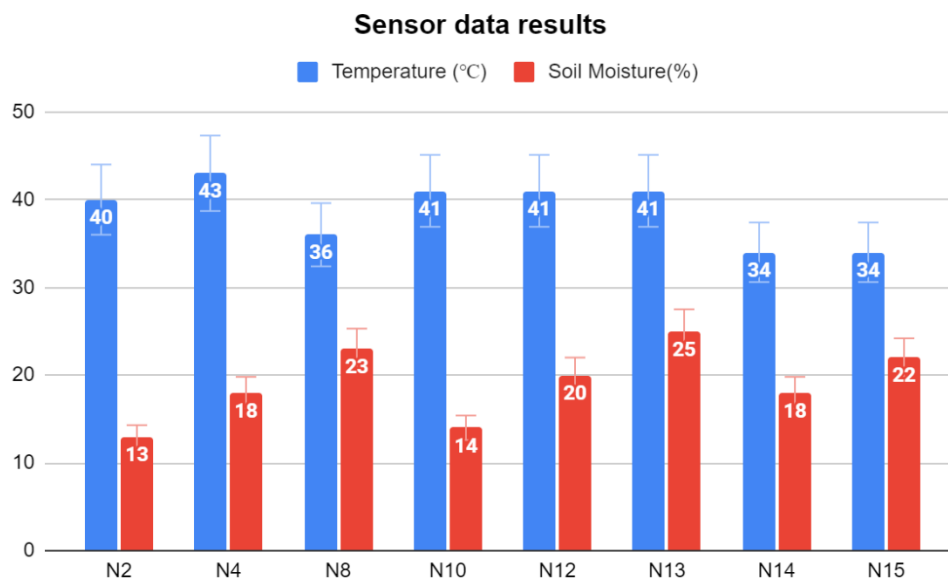


Fig. 5.7: Sensor data results

Figure 5.7 shows the average of the measurements taken by both sensors at the different ground nodes. The measurements are the average of the measurements and the bracket is the variance between them. The temperature sensors show different results between them considering that all the nodes are in the same region. However, it should be considered that due to the high temperatures on the day of the measurement campaign, some nodes were placed under the shade of trees to protect the computers of the different members of the team. It can be observed that the temperature of nodes 12 and 13 are the same, this is due to the fact that both sensors are located inside the same IP67 box. The same occurs when the temperatures of nodes 14 and 15 are analyzed.

Regarding the values of the capacitive soil moisture sensors, it can be seen that after the recalibration of the first measurement campaign, they now show more coherent results regarding the soil moisture values in table 4.2 of chapter 4.

5.3. Results of the pure ALOHA experiment

In this section, the results obtained in the pure ALOHA experiments are discussed. The results are then divided into 3 sections. The first section analyzes the success rate of the packets sent. Since the channel is saturated, many of the sent packets collide. The second section analyzes the packets received during waiting time. The third section analyzes the results obtained from successful communications. In this last section, different graphs are shown regarding the reception of the ACK after sending the Data Packet and the percentage of successful communications obtained according to the characteristics of the experiment performed. All the graphs of the results correspond to the average of all the nodes involved in the specific experiment. More information on the particular behavior of each node can be found in Annex A.

Since two pure ALOHA experiments have been performed, both experiments are compared with each other and the differences due to the different properties of each one are observed. Experiment 1, has a *t_to_next_packet* of 15 ms, so the channel is more saturated than experiment 3, which has a *t_to_next_packet* of 15 s.

5.3.1. Analysis of packages transmitted and received

Given that the pure ALOHA protocol is a MAC protocol that accesses the medium randomly, it is normal that the higher the density of nodes and the higher the transmission rate, the greater the saturation of the channel and the greater the packet loss. Two subsections are shown below where the percentage of packets transmitted and received is analyzed. In the first subsection, the percentage of data packets received versus data packets sent by the nodes is analyzed. In the second subsection, the percentage of ACK received versus ACK sent by the drone is analyzed.

Percentage of Data Packet received versus Data Packet sent by the nodes

The figure 5.8 show the results obtained from experiments 1 and 3. The value that can be seen in the following graphs is the average of the values of all the nodes of the experiment.

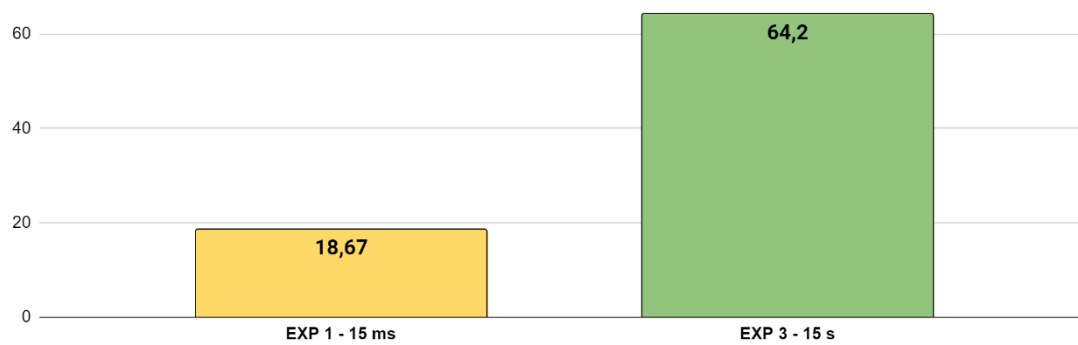


Fig. 5.8: Pure ALOHA – Average percentage of Data Packets received versus Data Packet sent by the nodes (%).

As can be seen, in experiment 1 the average is 18.67%. On the other hand, in experiment 3 the average is 64.20%. In experiment 1 there is a higher density of nodes. In addition, the packet sending rate after receiving the ACK is 1000 times lower than in experiment 3. Since the channel is much more saturated, many of the transmitted data packets do not reach the drone because they collide with the rest of the packets. In experiment 3, since the node waits 15 seconds before restarting a communication after receiving the ACK, the medium is much less saturated, allowing a larger number of data packets to be received by the drone. After receiving the ACK, the node initiates a wait to restart communications. During this wait, other nodes are more likely to be able to receive the ACK since the node density is lower. In addition, the nodes remain synchronized, since after the 15-second wait of the first node to receive the ACK, it is very likely that the rest of the nodes are still in their waiting time ($t_{to_next_packet}$) before restarting communications. Therefore, the longer the wait between the restart of communications after receiving the ACK, the greater the synchronization between the different nodes and the higher the percentage of data packets received by the drone.

Percentage of ACK received versus ACK sent by the drone

The figure 5.9 show the results obtained from experiments 1 and 3. The information that is observed is generated with the average in percentage of the ACKs received by the nodes versus the ACKs sent by the drone.

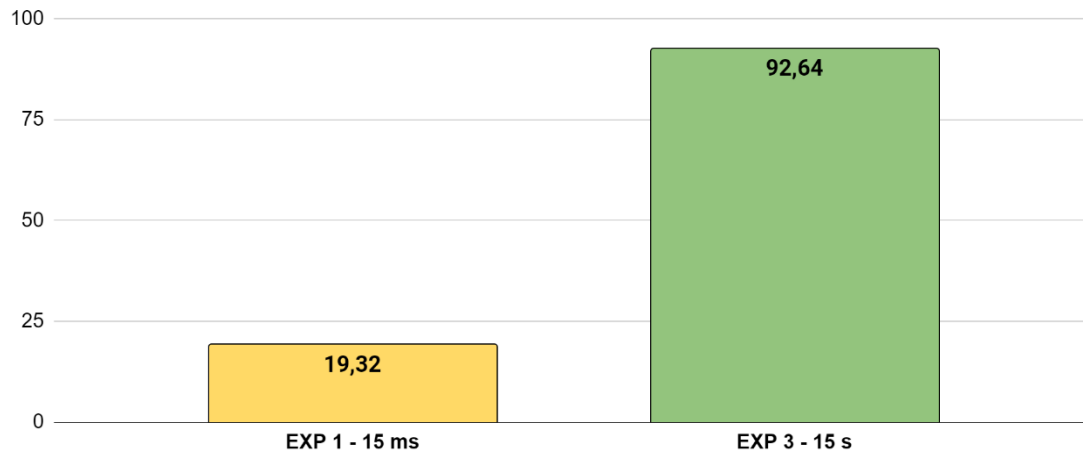


Fig. 5.9: Pure ALOHA – Percentage of ACK received versus ACK sent by the drone (%).

In this case, the packet sent is the ACK, which is sent by the drone. And the above plots show the percentage of ACK packets received by the different nodes compared to the total sent by the drone. In experiment 1, the average percentage of ACK packets received is 19.32%. On the other hand, in experiment 3 the average percentage of ACK packets received is 92.64%. Because the $t_{to_next_packet}$ is much larger in experiment 3, the channel is much less saturated than in experiment 1, allowing a large number of ACK packets to reach the indicated node without colliding with other packets.

5.3.2. Analysis of packages received during the waiting time

Since the pure ALOHA protocol is a MAC protocol that accesses the medium randomly (RA), it is normal that the higher the density of nodes and the higher the transmission rate, the higher the saturation of the channel. This section analyzes the packets received during the waiting time to receive the ACK. During the waiting time, it is likely that other packets will be received instead of the ACK or that nothing is received at all. Four cases are possible. The first case is where a data packet is received from another node that has initiated communication with the drone. The second case is that an erroneous ACK is received which has been transmitted to another node. The third case is that no packet is received during the timeout period. Finally, the fourth case is that the correct ACK is received. The figure 5.10 show the average percentage of packets received during the waiting time to receive the ACK of the nodes experiment 1 and 3.

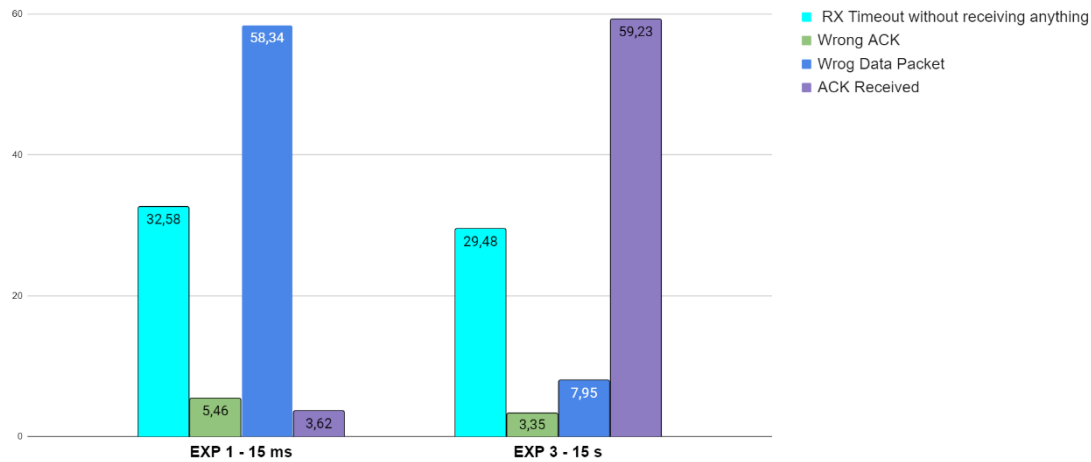


Fig. 5.10: Pure ALOHA – Percentage of the average packages received during the waiting time.

For experiment 1, it can be seen that 58.34% of the packets received are data packets from other nodes which have initiated communications at the time when the waiting time was performed. On the other hand, in 32.58% of the cases, no packet is received by the node. As a result, the waiting time ends without any data being received. In 5.46% of the cases, an erroneous ACK is received which is destined for another node. Finally, in only 3.62% of cases is the correct ACK received.

In experiment 3, the data are much more favorable. In 59.23% of the cases, the correct ACK is received, so the percentage of successful communications in experiment 3 is much larger due to a lower node density and a longer wait between communication initiations after receiving the ACK. The next dominant case is the case where no packet is received. Finally, in 7.95% of the cases, an erroneous packet is received and in 3.35% of the cases, an erroneous ACK is received.

5.3.3. Analysis of successful communications

In this section, the results obtained in experiments 1 and 3 are analyzed. In the first experiment, it has been observed in the previous results how the channel saturation is evident. On the other hand, the channel saturation in experiment 3 is lower, allowing the different transmitted packets not to collide and reach their destination. Next, three subsections are presented where different graphs of the experiments are analyzed. The first subsection shows the percentage of ACKs received versus Data Packets sent. The second subsection shows the percentage of successful communications versus failed communications after K_{\max} attempts. In all experiments, the maximum number of retransmissions to receive the ACK is 5. Finally, the third subsection shows the percentage of attempts needed for successful communications.

Percentage of ACK received versus data packets sent

If there were no collisions and all packets arrived before the end of the waiting time, all sent data packets would receive an ACK. However, in a scenario where there is a certain density of nodes and the channel is saturated, this does not occur. The average percentage of ACK packets received versus data packets transmitted is shown below.

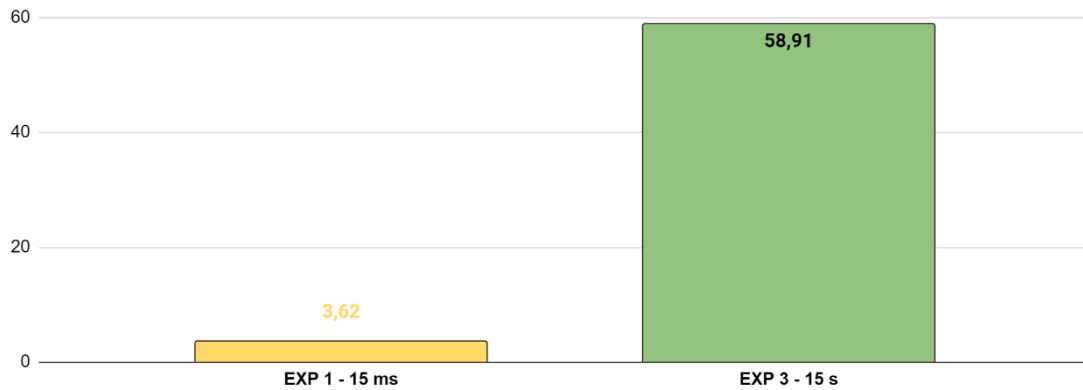


Fig. 5.11: Pure ALOHA – Average percentage of ACK received versus data packets sent.

As can be seen in the previous figures, the percentage of ACK packets received versus Data Packets sent is higher in experiment 3. In experiment 1, on average only 3.62% of the cases, the ACK is received. On the other hand, in experiment 3, the ACK is received in 58.91% of the cases. This is due to the reasons discussed above. The lower the density of the nodes and the longer the waiting time between retransmissions after receiving the ACK ($t_{to_next_packet}$), the higher the probability that the packet reaches the destination without collision.

Percentage of successful communications versus failed communications after Kmax attempts

The following subsection shows the results obtained after analyzing the number of times where the communication has been declared successful and the number of times where the communication is considered a failure. A communication is considered successful if the ACK has been received before exceeding the maximum number of attempts allowed to receive it. On the other hand, a communication is considered a failure when it exceeds this limit. The figure 5.12 show the results obtained from both experiments.

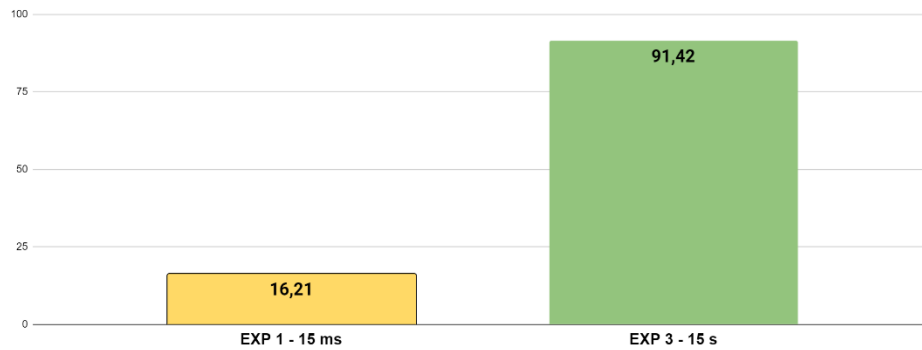


Fig. 5.12: Pure ALOHA – Average percentage of successful communications versus failed communications after Kmax attempts

As can be seen, the percentage of successful communications with 5 retransmission attempts is higher than the values in the previous subsection. In the case of experiment 1, in 16.21% of the cases, successful communication is achieved. On the other hand, in experiment 3, in 91.42% of the cases, successful communication is achieved.

Percentage of attempts needed for successful communications

After sending a Data Packet, it waits for a period until the ACK is received. If after this time the ACK has not arrived, an attempt to receive the ACK is added and the backoff process is performed. After waiting the backoff time, the Data Packet is sent again. In this subsection the number of attempts required to receive the ACK is shown as a percentage. The figure 5.13 show the results obtained in both experiments.

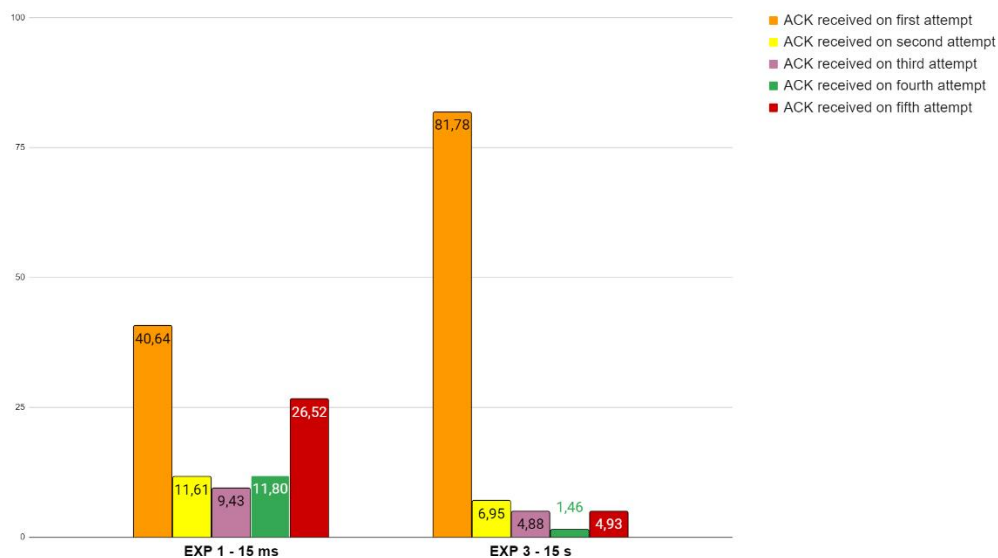


Fig. 5.13: Pure ALOHA – Average percentage of attempts needed for a successful communication.

As can be seen in the previous figures, the ACK is usually received on the first attempt. In experiment 1, in 40.64% of the cases the ACK is received on the first attempt. This is due to the fact that after a successful communication the node only waits 15 ms to initiate the next communication, so it is very likely that the medium is still free and therefore the next transmission is successful. The medium is free because most of the nodes are waiting for the backoff time after failing to receive the ACK. However, when the backoff time of the rest of the nodes is over, they retransmit the Data Packet and interrupt the node's communications. On the other hand, it can be seen that in 26.52% of the cases the ACK is received on the fifth attempt. Since the margin of the backoff time is larger the higher the number of attempts, the probability that all nodes calculate the same backoff time is reduced. Therefore, the probability of successful transmission after the backoff is higher.

In the second experiment, the ACK is received on the first attempt in 81.78% of the cases. Since the retransmission waiting time after receiving the ACK is 15 seconds, the nodes are naturally synchronized. When the first node receives the ACK, it waits 15 seconds until it restarts communications. The rest of the nodes in these seconds manage to communicate with the drone and receive the ACK, so they also wait 15 seconds. Once all the nodes have communicated and are waiting, the first node retransmits the Data Packet when the rest of the nodes are still waiting. For this reason, there is a high percentage of ACK reception on the first attempt.

5.4. Results of the CSMA/CA experiment

In this section, the results obtained in the experiments using the CSMA/CA protocol are analyzed. The results are then divided into 3 sections. The first section analyzes the success rate of the packets sent. The second section analyzes the packets received during waiting time. The third section analyzes the results obtained from successful communications. In this last section, different graphs are shown regarding the reception of the ACK after the communication process. All the graphs of the results correspond to the average of all the nodes involved in the specific experiment. More information on the particular behavior of each node can be found in Annex B.

Since two CSMA/CA experiments have been performed, both experiments are compared with each other and the differences due to the different properties of each one are observed. Experiment 6, has a $t_{to_next_packet}$ of 1 ms, so the channel is more saturated than experiment 4, which has a $t_{to_next_packet}$ of 15 s. In this case, both experiments have the same node density.

5.4.1. Analysis of packages transmitted and received

This section shows the graphs obtained by comparing the number of packets sent versus the number of packets received. The CSMA/CA protocol initiates communications after sensing the medium. If it detects that the medium is free,

then it sends the first RTS packet. Subsequently, this packet reaches the drone, which replies with a CTS, thus reserving the channel. Since it is possible for different nodes to detect the free medium at the same time, most collisions can occur in the process of reserving the channel. Once the channel is reserved by the CTS, the rest of the communications are based on sending the Data Packet from the node and the drone responds with the ACK. Next, four subsections are shown where the percentage of transmitted and received packets is analyzed. A column referring to the average generated among all the nodes in the experiment is also shown. In the first subsection, the percentage of RTS received versus RTS sent by the nodes is analyzed. In the second subsection, the percentage of CTS received versus CTS sent by the drone is analyzed. In the third subsection, the percentage of Data Packet received versus Data Packet sent by the nodes is analyzed. In the fourth subsection, we analyze the percentage of ACK received versus ACK sent by the drone.

Percentage of RTS received versus RTS sent by the nodes

The figure 5.14 show the results obtained from experiments 4 and 6. The values that can be seen are the average of the values of all the nodes of the experiment.

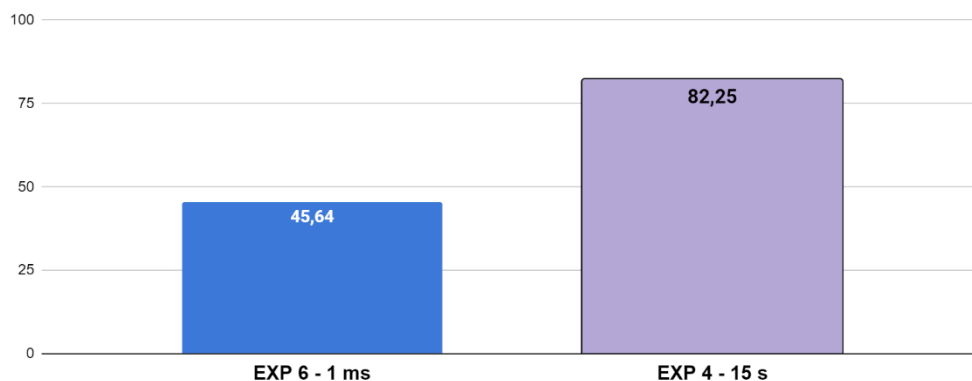


Fig. 5.14: CSMA/CA – Average percentage of RTS received versus RTS sent by the nodes

As can be seen in the figures above, the average percentage of RTSs received in experiment 6 is 45.64%. This may be due to two reasons. The first reason is that the RTS packet does not reach the drone before the waiting time expires. The second reason is that the sending of the RTS packet collides with other packets. This is the most likely case since it is possible that several nodes detect the free medium at the same time and transmit the RTS packet. In addition, it should be noted that in experiment 6 the waiting time after receiving the ACK ($t_{to_next_packet}$) is 1 ms, so the channel is saturated at any moment and the probability of collision is higher.

In experiment 4, it can be seen how the average percentage of RTS received is 82.25%. In experiment 4, the waiting time after receiving the ACK

($t_{to_next_packet}$) is 15 s, so the channel is much less saturated and the probability of collision between packets is lower.

Percentage of CTS received versus CTS sent by the drone

The figure 5.15 show the results obtained from experiments 4 and 6. The values that can be seen are the average of the values of all the nodes of the experiment.

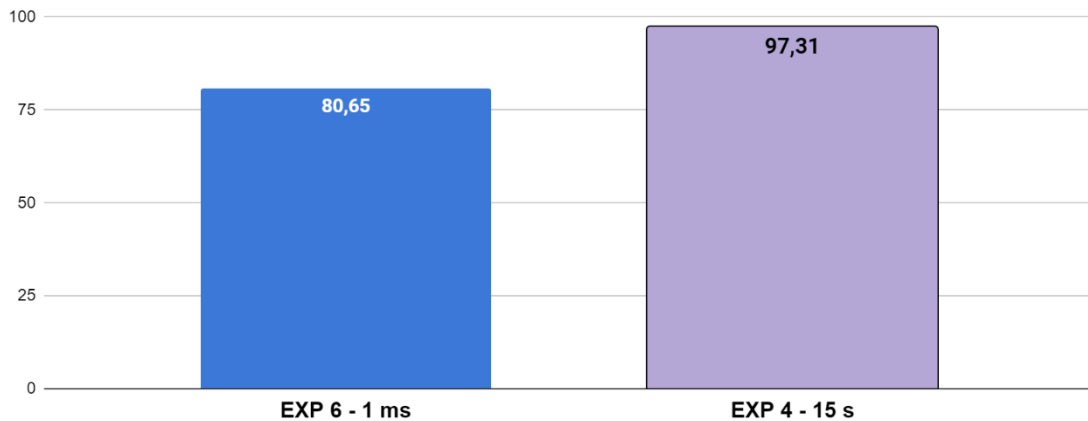


Fig. 5.15: CSMA/CA – Average percentage of CTS received versus CTS sent by the drone

As can be seen in the figures above, the average percentage of CTS received in experiment 6 is 80.65%. Once the drone sends the CTS packet, it is likely to collide with some RTS packet sent by some other node. In experiment 4, it can be observed how the average percentage of CTS received is 97.31%. The probability of collision between packets in this experiment is lower because the $t_{to_next_packet}$ is 15 seconds, so the channel is not as saturated as in experiment 6.

Percentage of Data Packet received versus Data Packet sent by the nodes

The figure 5.16 show the results obtained from experiments 4 and 6. The values that can be seen are the average of the values of all the nodes of the experiment.

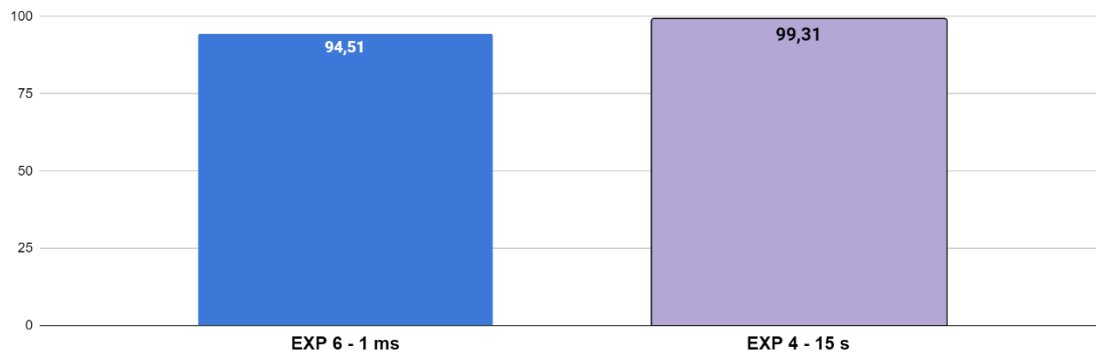


Fig. 5.16: CSMA/CA – Average percentage of Data Packet received versus Data Packet sent by the nodes

As can be seen in the figures above, the average percentage of Data Packets received in experiment 6 is 94.51%. On the other hand, the average percentage of Data Packets received in experiment 4 is 99.31%. It can be seen that very few Data Packets are not received by the drone. This is because once the drone sends the CTS to the node, the channel is reserved for this node. If the CTS packet is received while the nodes are sensing the channel, they must wait for a timeout defined by the NAV_CTS. On the other hand, if the CTS is received by other nodes after sending the RTS (since they have detected the free channel), the backoff process is initiated and the medium is re-sensed.

Percentage of ACK received versus ACK sent by the drone

The figure 5.17 show the results obtained from experiments 4 and 6. The values that can be seen are the average of the values of all the nodes of the experiment.

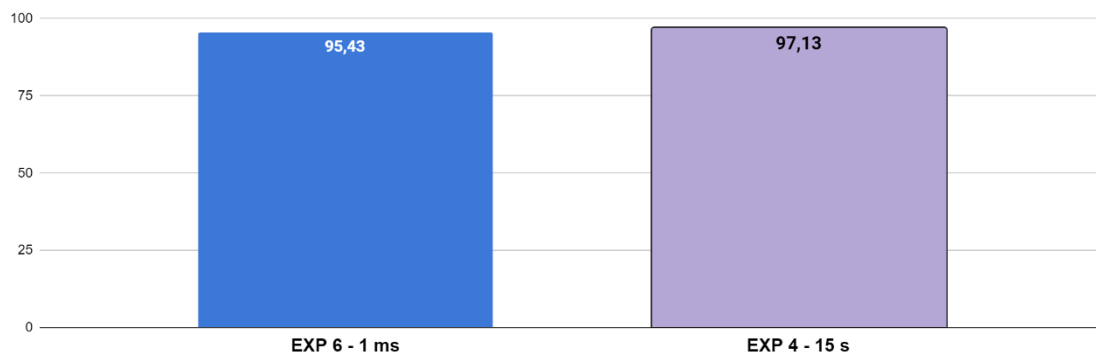


Fig. 5.17: CSMA/CA– Average percentage of ACK received versus ACK sent by the drone

As can be seen in the figures above, the average percentage of ACKs received in experiment 6 is 95.43%. On the other hand, the average percentage of ACKs

received in experiment 4 is 97.13%. The other small percentage corresponds to packets that did not arrive before the end of the waiting time.

5.4.2. Analysis of packages received during waiting times

This section analyzes the packets received during the waiting time. There are a total of three waiting times in the communication process using the CSMA/CA protocol. The first one takes place when the RTS packet is sent. The node waits for a certain time to receive the CTS. The second timeout is performed on the drone after sending the CTS. The drone waits for a certain time to receive the Data Packet. If after that time it has not received the Data Packet it returns to active listening for other RTS packets from other nodes. Finally, the third timeout is performed on the nodes after sending the Data Packet. During the first waiting time, other RTS packets are likely to be received instead of the correct CTS. However, the node ignores these packets and continues waiting to receive the CTS during the remaining waiting time. In the second timeout, it is also possible that other RTS packets are received instead of the Data Packet, however, the drone ignores them and continues to wait to receive the Data Packet for the remaining time. In the third timeout, the node waits for a certain amount of time to receive the ACK. If after this waiting time the ACK packet has not been received, the backoff process is initiated and the average is re-sensed to restart the communication process.

The results obtained in both experiments are shown below, divided into 4 subsections. The first subsection analyzes the percentage of times the CTS was received versus the number of times the waiting time expired. The second subsection analyzes the percentage of packets received during the CTS waiting time. The third subsection analyzes the percentage of times the Data Packet has been received versus the times the waiting time has expired. Finally, the fourth subsection analyzes the percentage of times the ACK has been received versus the number of times the waiting time has expired.

Percentage of times the CTS has been received against times the wait time has expired

In the figure 5.18, the results obtained from experiments 4 and 6 are shown. The values that can be seen are the average of the values of all the nodes of the experiment.

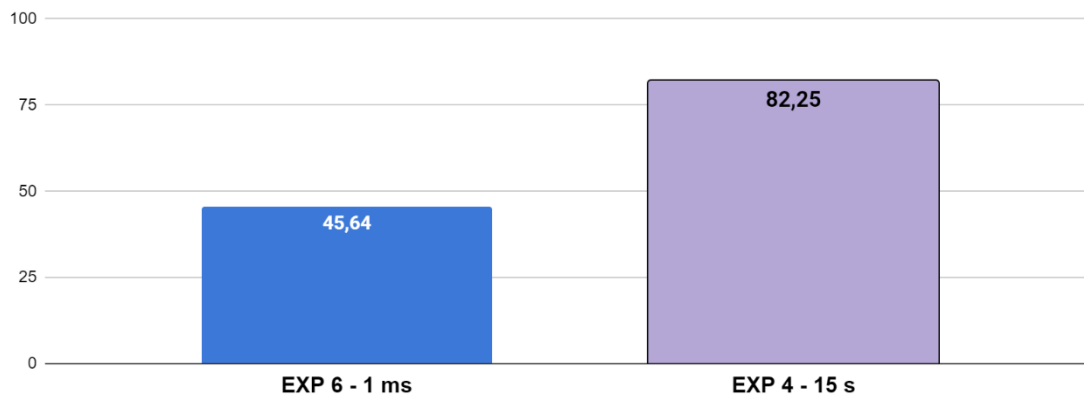


Fig. 5.18: CSMA/CA – Average percentage of times the CTS has been received against times the wait time has expired

For experiment 6 it can be observed that in 29.34% of the cases the CTS has been received. On the other hand, for experiment 4, in 76.42% of the cases, the CTS is successfully received. In the remaining cases, three things may have happened. The first one is that the RTS has not arrived correctly to the drone since it has collided with an RTS packet from another node. Second, the CTS sent by the drone collided with an RTS packet from another node. The third is that the waiting time expires, and the CTS does not arrive on time. Since in experiment 4 the waiting time between sending packets after receiving the ACK ($t_{to_next_packet}$) is longer than in experiment 6, the channel is less congested and there are not so many collisions. For this reason, the data from experiment 4 are more favorable.

Percentage of packages received during the waiting time to receive the CTS

This subsection analyzes the types of packets received during the waiting time of the different nodes. Five cases are possible. The first case is where an RTS is received from another node that has initiated communication with the drone. The second case is where an erroneous CTS is received. The third case is when a Data Packet sent by another node is received. The fourth case is that an erroneous ACK is received which has been transmitted to another node. Finally, the fifth case is that no packet is received during the waiting time.

The figure 5.19 show the average percentage of packets received during the waiting time period. These graphs do not take into account the CTS packets received, so only the cases discussed above are considered.

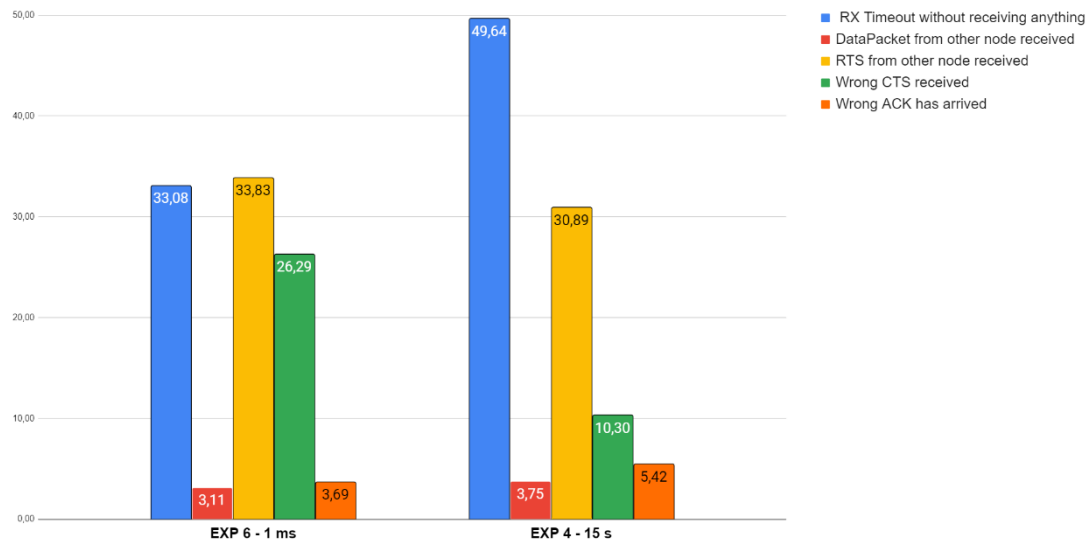


Fig. 5.19: CSMA/CA – Average percentage of packages received during the waiting time to receive the CTS

In experiment 6, we have previously seen that in only 29.34% of the cases the CTS is received before the waiting time expires. In the remaining 70.66% of cases, the following packages are received. On average, in 33.83% of the cases, an RTS is received from another node. In 33.08% of the cases, no packet is received. In 26.26% of cases, an erroneous CTS is received and addressed to another node. In 3.11% of the cases, a Data Packet is received from another node which is addressed to the drone. Finally, in 3.69% of the cases, an erroneous ACK is received that corresponds to another node.

On the other hand, in experiment 4, we have previously seen that in 76.42% of the cases the CTS is received before the waiting time expires. In the remaining 23.58% of the cases, the next packets are received. On average, in 49.64% of the cases, no package is received. In 30.89% of the cases, an RTS is received from another node. In 10.30% of the cases, an erroneous CTS is received which is addressed to another node. In 3.75% of the cases, a Data Packet is received from another node which is addressed to the drone. Finally, in 5.42% of the cases, an erroneous ACK is received that corresponds to another node.

Percentage of times the Data Packet has been received against times the wait time has expired

The figure 5.20 show the results obtained from experiments 4 and 6. The values that can be seen are the average of the values of all the nodes of the experiment.

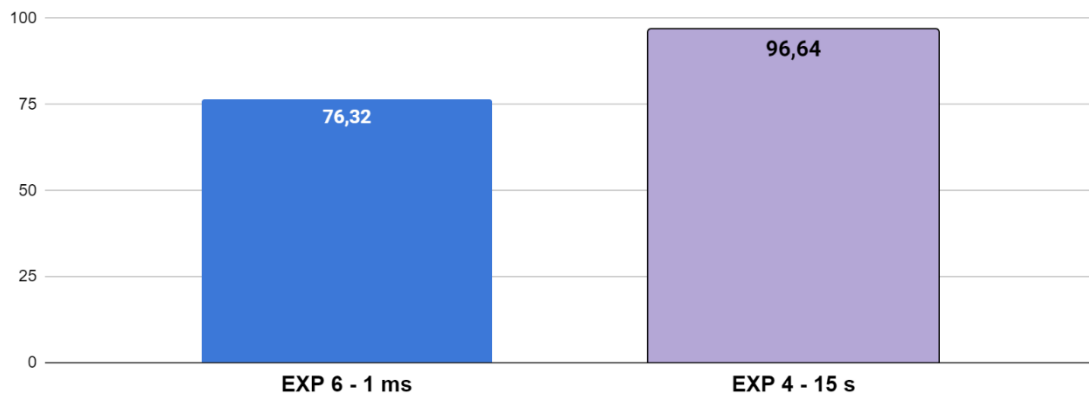


Fig. 5.20: CSMA/CA – Average percentage of times the Data Packet has been received against times the wait time has expired

For experiment 6, it can be observed that in 76.32% of the cases the Data Packet has been successfully received by the drone. On the other hand, for experiment 4, in 96.64% of the cases, the Data Packet is successfully received. In the remaining cases, two things may have happened. First, the Data Packet did not arrive correctly at the drone because it collided with an RTS packet from another node. The second, is that the waiting time expired and the Data Packet did not reach the drone in time. Since in experiment 4 the waiting time between sending packets after receiving the ACK ($t_{to_next_packet}$) is longer than in experiment 6, the channel is less congested and there are not so many collisions. For this reason, the data from experiment 4 are more favorable.

Percentage of times the ACK has been received against times the wait time has expired

In the figure 5.21, the results obtained from experiments 4 and 6 are shown. The values that can be seen are the average of the values of all the nodes of the experiment.

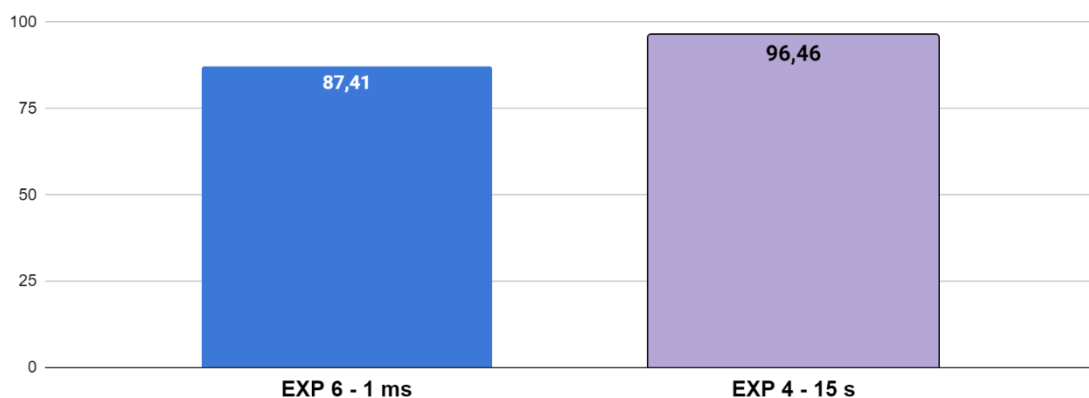


Fig. 5.21: CSMA/CA – Average percentage of times the ACK has been received against times the wait time has expired

For experiment 6 it can be observed that in 87.41% of the cases the ACK has been successfully received by the drone. On the other hand, for experiment 4, in 96.46% of the cases, the ACK is successfully received. In the rest of the cases, the waiting time ends, and the ACK is not received. For these cases, it is most likely that the waiting time has expired without any other packet being received. However, there is a possibility that some node after sensing the medium before this communication started, and after having had a very long DIFS time, sends the RTS just at the time when the ACK is sent and then both packets collide. Since in experiment 4 the waiting time between sending packets after receiving the ACK ($t_{to_next_packet}$) is longer than in experiment 6, the channel is less congested and there are not so many collisions. For this reason, the data from experiment 4 are more favorable compared to experiment 6.

5.4.3. Analysis of successful communications

In this section, the results obtained in experiments 6 and 4 are analyzed. In experiment 6, it has been observed in the previous results that the channel saturation is higher than in experiment 4. In the following, three subsections are presented where different plots of the experiments are analyzed. The first subsection shows the percentage of ACKs received versus RTSs sent. The second subsection shows the percentage of successful communications versus failed communications after K_{max} attempts. In all experiments, the maximum number of retransmissions to receive the ACK is 5. Finally, the third subsection shows the percentage of attempts needed for successful communications.

Percentage of ACK received versus RTS sent (%Success)

If there were no collisions and all packets arrived before the end of the waiting time, all sent RTSs would receive an ACK after the CSMA/CA protocol communication process. However, in a scenario where there is a certain density of nodes and the channel is saturated, this does not happen. The value that can be seen in the following graphs is the average of the values of all the nodes of the experiment.

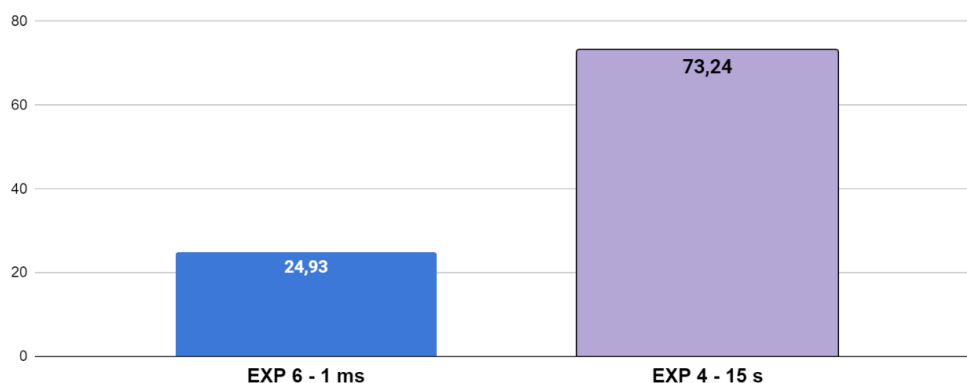


Fig. 5.22: CSMA/CA – Average percentage of ACK received versus RTS sent.

As can be seen in the figure 5.22, the percentage of ACK packets received versus RTS sent is higher in experiment 4. In experiment 6, on average, ACK is received in only 24.93% of the cases. On the other hand, in experiment 4, the ACK is received in 73.24% of the cases. This is because the longer the waiting time between retransmissions after receiving the ACK ($t_{to_next_packet}$), the higher the probability that the packet reaches the destination without collision.

Percentage of successful communications versus failed communications after K_{max} attempts

The following subsection shows the results obtained after analyzing the number of times the communication has been declared successful and the number of times the communication is considered a failure. A communication is considered successful if the ACK has been received before exceeding the maximum number of attempts allowed to receive it. On the other hand, a communication is considered a failure when it exceeds this limit. The value that can be seen in the following graphs is the average of the values of all the nodes of the experiment.

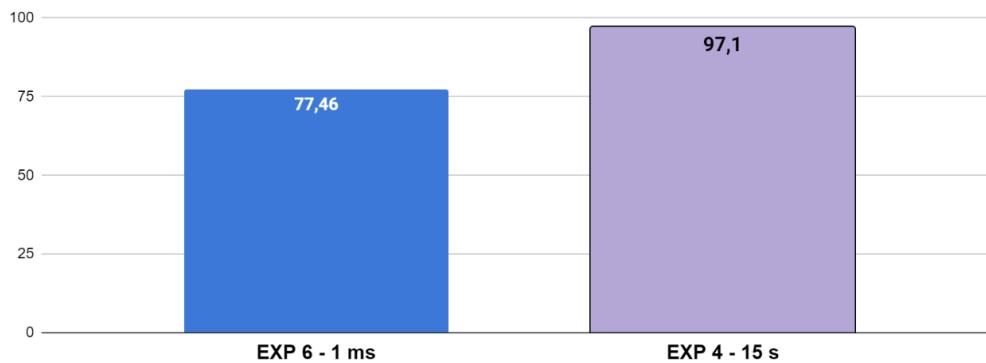


Fig. 5.23: CSMA/CA – Average percentage of successful communications versus failed communications after K_{max} attempts

As can be seen, the percentage of successful communications with 5 retransmission attempts is higher than the values in the previous subsection. This is due to the fact that the greater the number of attempts to receive the ACK, the greater the probability of successful communication. In the case of experiment 6, successful communication is achieved in 77.46% of the cases. On the other hand, in experiment 4, successful communication is achieved in 97.10% of the cases.

Attempts needed for a successful communication

After sending an RTS, the nodes wait for a certain period to receive the CTS. If after that time the CTS has not been received, a reception attempt is added and the backoff process is performed. After waiting the backoff time, the communication starts again. If, on the other hand, the CTS is received, the node sends the Data Packet and starts a waiting time again. If, after this waiting time, the ACK packet has not been received, a reception attempt is added and the

backoff process is executed. This subsection shows the number of attempts required to receive the ACK in percentage. The value that can be seen in the following graphs is the average of the values of all the nodes of the experiment.

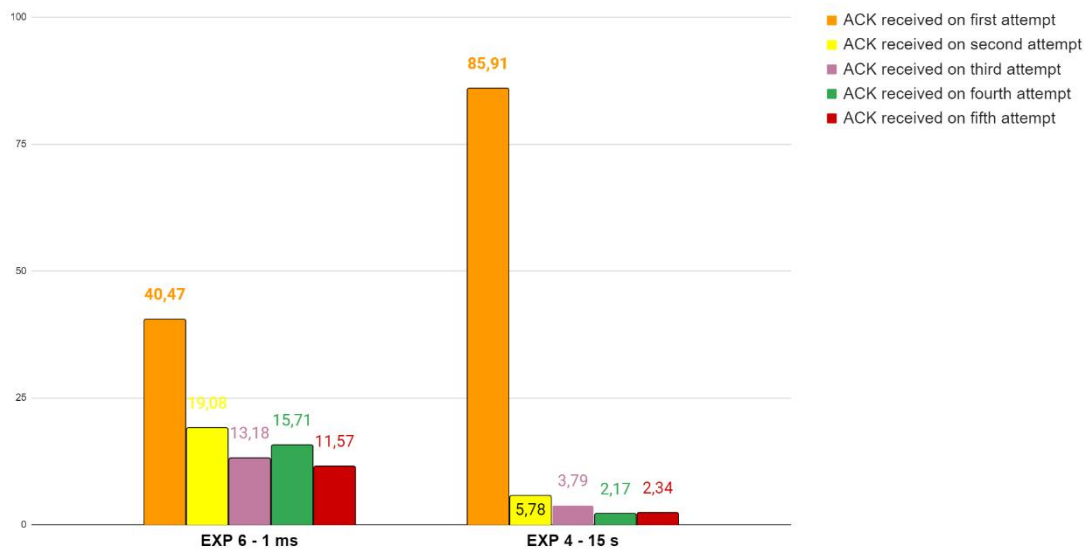


Fig. 5.24: CSMA/CA – Average percentage of attempts needed for a successful communication

As can be seen in the previous figures, the ACK is usually received on the first attempt in practically all nodes. In experiment 6, in 40.47% of the cases, the ACK is received on the first attempt. This is due to the fact that after a successful communication the node waits only 1 ms to initiate the next communication, so it is very likely that the medium is still free and therefore the next transmission may be successful. The medium may be free because most of the nodes are waiting for the backoff time after failing to receive the ACK. Or, they are waiting for a time determined by the NAV to re-sense the medium and start the communication process. However, when the backoff time or the NAV of the rest of the nodes is over, they sense the medium to detect whether it is free or not. If several nodes sense the medium at the same time, it is likely that several will detect that it is free, which may cause collisions when sending the RTS. As can be seen in the graphs, in 19.08% of the cases, the ACK is received on the second attempt. In 13.18% of the cases, the ACK is received on the third attempt. In 15.71% of cases, the ACK is received on the fourth attempt. And finally, in 11.57% of the cases, the ACK is received on the fifth attempt.

In the second experiment, the ACK is received on the first attempt in 85.91% of the cases. Since the retransmission time after receiving the ACK is 15 seconds, the nodes are naturally synchronized. When the first node receives the ACK, it waits 15 seconds to restart communications. The rest of the nodes in these seconds manage to communicate with the drone and receive the ACK, so they also wait 15 seconds. Once all the nodes have communicated and are waiting, it is probable that the first node will sense the free medium and send the RTS when the rest of the nodes are still in the waiting process. For this reason, there is a high percentage of ACK reception on the first attempt.

5.5. Analysis and comparison of the performance of both experiments

In the following, the performance of both protocols in the different experiments is analyzed. In addition, the experiments whose waiting times after receiving the ACK are the same are analyzed together. In other words, experiments 1 and 6 are analyzed and compared first. And second, experiments 3 and 4 are analyzed and compared.

Experiment 1 corresponds to the pure ALOHA protocol with a $t_{to_next_packet}$ of 15 ms. Experiment 6 corresponds to the CSMA/CA protocol with a $t_{to_next_packet}$ of 1 ms. Next, a comparison of the results obtained from the different experiments is shown and it is analyzed which protocol performs better in the case where the channel is saturated. The figure 5.25 shows the percentage of average of all the nodes involved in each experiment. Specifically, it shows the percentage of ACK received versus Data Packet / RTS sent. In the same graph, also shows the percentage of successful communications versus failed communications after K_{max} attempts.

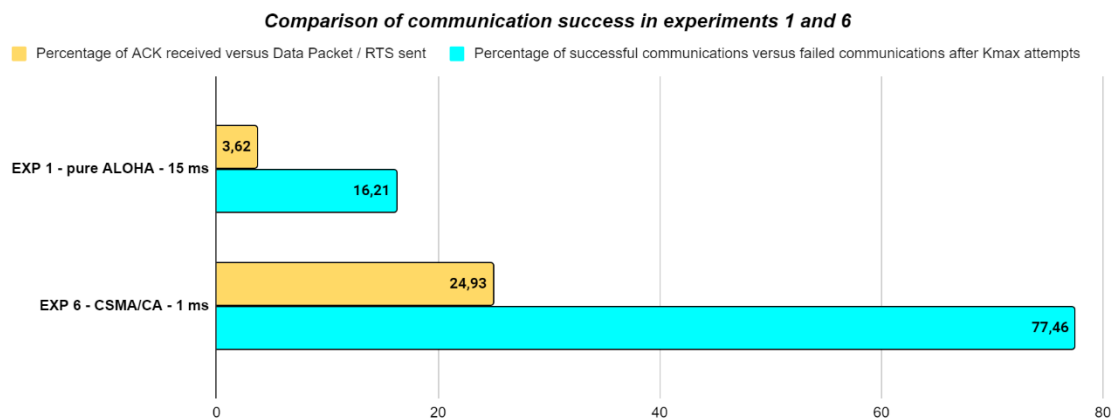


Fig.5.25: Comparison of communication success in experiments 1 and 6

In both experiments, the channel is saturated because after each successful transmission the node transmits again with practically no delay. In the case of the pure ALOHA experiment, it can be seen that in only 3.62% of the cases is it possible to receive the ACK after sending the Data Packet. On the other hand, in the CSMA/CA experiment, in 24.93% of the cases, the ACK is received after sending the RTS. If the maximum number of attempts (K_{max}) is taken into account, the pure ALOHA experiment has a 16.21% probability of successful communication. On the other hand, the CSMA/CA experiment has a 77.46% chance of successful communication.

In the pure ALOHA protocol, when the channel is saturated, it is very likely that the packets sent collide with others. To have a successful communication, it is necessary that the Data Packet is correctly received in the drone and that the ACK arrives in time to the node without colliding with any other packet. In the

case of the CSMA/CA protocol, the nodes must sense the medium before transmitting, which means that there are not so many collisions since many of the nodes wait for a NAV to initiate communications if they detect that there is a communication taking place. However, a collision-free space is not possible. Especially in an environment where the channel is saturated.

The comparison between experiment 3 and experiment 4 is shown below. In both experiments the channel is much less saturated because the waiting time between the start of a new communication after receiving the ACK is a thousand times bigger. In this case, the $t_{to_next_packet}$ is 15 seconds, so a node stops accessing the channel during this time after a successful communication.

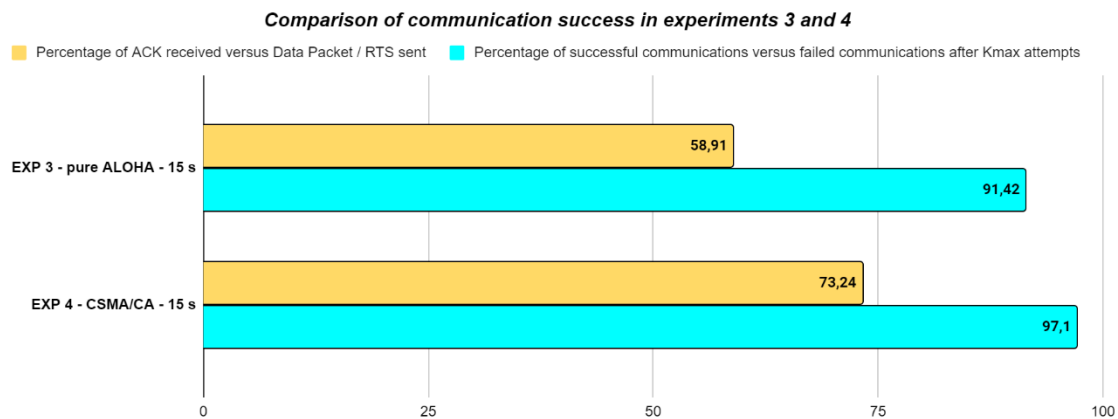


Fig. 5.26: Comparison of communication success in experiments 3 and 4

As can be seen in the previous figure, in the case of the pure ALOHA experiment, 58.91% of the cases of receiving the ACK after sending the Data Packet can be observed. On the other hand, in the CSMA/CA experiment, in 73.24% of the cases it is possible to receive the ACK after sending the RTS. If the maximum number of attempts (K_{max}) is taken into account, the pure ALOHA experiment has a 91.42% probability of successful communication. On the other hand, the CSMA/CA experiment has a 97.1% probability of having a successful communication. In this case, the probability of successful communication is much higher.

Since the retransmission waiting time after receiving the ACK is 15 seconds, the nodes are naturally synchronized. When the first node receives the ACK, it waits 15 seconds until it restarts communications. The rest of the nodes in this time are more likely to communicate with the drone and receive the ACK since the node density is lower. Once they receive the ACK, they wait 15 seconds, reducing again the density of nodes. Once all the nodes have communicated and are in the waiting time to restart communications, the first node that started the wait retransmits the Data Packet with a high probability of having a free channel. For this reason, there is a high percentage of reception of the ACK when sending the Data Packet or RTS. For CSMA/CA, there is the added complexity that the nodes

sense the medium before transmitting, so the probability of collision is drastically reduced. Finally, if we consider the successful communications after 5 attempts, it can be seen how both protocols have an excellent performance, highlighting the 97.1% provided by the CSMA/CA protocol with a waiting time between retransmissions of 15 seconds.

CHAPTER 6: Conclusions and future development

In this project, the LoRa communications experiment proof-of-concept has been performed using several ground IoT nodes and a miniaturized drone-based payload. The communications have been performed using two MAC protocols that are compatible with an IoT scenario: pure ALOHA and CSMA/CA with RTS/CTS. In both protocols, the useful information to be sent is the data contained in the Data Packet. In this packet, the data obtained by the capacitive soil moisture sensor and the temperature sensor are stored.

Chapter 1 provides an introduction to the aims and objectives of the project. Chapter 2 (SoA) determines the technology used in this experiment. This chapter defines the MAC protocols to be used and corroborates why LoRa technology has better properties than other LPWAN technologies. Chapter 3 explains the methodology applied to develop the experiment, both in the software and hardware parts. In chapter 4, the results obtained in the first measurement campaign in which data were taken from the capacitive soil moisture sensor are analyzed. Finally, Chapter 5 explains how the second measurement campaign was performed and details the results obtained.

In this project, all the objectives mentioned in section 1.2 have been achieved despite all the inconveniences and setbacks. It should be noted that initially this project was only designed to perform the ground nodes design and software implementation and test it with the RITA LoRa module. However, since the RITA LoRa module was not prepared for the campaign, the gateway code of both protocols had to be implemented in the structure designed for the drone. To accomplish all these objectives has been necessary to implement the pure ALOHA protocol and the CSMA/CA protocol together in the same code controlled by a command in the CubeCell transceiver of the IoT devices and the drone. In addition, it has been necessary to design the COTS structure of the ground nodes and the payload of the drone. Finally, both the software implementation and the hardware design have been tested in a measurement campaign where data have been stored and subsequently analyzed in order to determine which protocol obtains the best performance according to its characteristics. In the following paragraphs, some conclusions obtained after this study and possible improvements for the future development of the work are presented.

The first conclusion obtained after the measurement campaign is that both the LoRa communications experiment and the GNSS-R experiment can be executed simultaneously in the same payload with shared equipment without interference or problems.

In an IoT environment, it is required that all devices used should be low-power, including sensors. The following is an analysis of whether the sensors used in the campaign fulfill these objectives. The HDC1080 temperature sensor has a power consumption of 1.3 μ A during measurements, while in the sleeping mode it has a minimum power consumption of 100 nA. On the other hand, the capacitive soil moisture v1.2. sensor has a power consumption in measurements of 5 mA and has no sleeping mode state. Since the ground nodes are designed to be low-

power IoT devices, the soil moisture sensor is not compliant with the low-power requirements, which is why in future implementations this sensor should be replaced by a low-power digital sensor. On the other hand, the temperature sensor meets the requirements of low power consumption for IoT nodes, therefore it is concluded that the temperature sensor HDC1080 is suitable for this type of scenario.

In chapter 5 the different data obtained in the experiments performed using the pure ALOHA and CSMA/CA protocols have been analyzed. A total of two experiments have been performed for each protocol. The first one with a short time between packet transmissions after receiving the ACK ($t_{to_next_packet}$) and the second one where the $t_{to_next_packet}$ is three orders of magnitude bigger.

In conclusion, it has been observed that with a small $t_{to_next_packet}$ (between 1 ms and 15 ms in the experiments) the channel is highly saturated, which causes a high probability of collision between packets. On the other hand, when the $t_{to_next_packet}$ is larger (15 seconds), the channel is not so saturated, and the probability of collision is reduced. In these cases, the nodes are naturally synchronized. This is because once a node receives the ACK, it initiates a wait determined by the $t_{to_next_packet}$, ceasing to occupy the channel and thus reducing the density of nodes trying to communicate with the drone. The rest of the nodes are more likely to receive the ACK, and as they receive the ACK, they also start the waiting time leaving the channel free for the next node to communicate with the drone. Therefore, it is concluded that the higher the $t_{to_next_packet}$ and the lower the density of nodes in the scenario, the lower the percentage of collisions between packets and the higher the percentage of successful communications. In addition, if the possible K_{max} attempts are considered, the percentage of successful communications increases.

By comparing the results obtained by each protocol, it can be concluded that for any type of scenario, the best performance is offered by the CSMA/CA protocol. In a scenario where the channel is saturated, the performance of the CSMA/CA protocol outperforms that of the pure ALOHA protocol. On the other hand, in a scenario where the channel is not highly saturated, the percentages of successful communications are similar, but it is still preferable to use the CSMA/CA protocol for communications.

As a future development in this project, it would be convenient to make the IoT ground nodes independent and achieve remote telemetry recording without the nodes being connected to the computer. In addition, a future proof-of-concept should be performed where the IoT ground nodes are tested in a more realistic case. In this experiment, the nodes should be placed more widely spaced and with different densities using other types of low-power sensors for IoT environments.

CHAPTER 7: Bibliography

- [1] CubeSat. Cubesat. Accessed: Nov. 2019. [Online]. Available: <http://www.cubesat.org/>
- [2] Mekki, K.; Bajic, E.; Chaxel, F.; Meyer, F. *A comparative study of LPWAN technologies for large-scale IoT deployment*. ICT Express 2019, 5, 1–7.
- [3] Sigfox. *Sigfox: The Global Communicator Service Provider*. Available on: <https://www.sigfox.com/en>
- [4] 3GPP. Release 13. Available on: <https://www.3gpp.org/release-13>
- [5] LoRa Alliance. Available on: <https://lora-alliance.org/>
- [6] Qu, Z.; Zhang, G.; Cao, H.; Xie, J. *LEO Satellite Constellation for Internet of Things*. IEEE Access 2017, 5, 18391–18401.
- [7] Adelantado, F.; Vilajosana, X.; Tuset-Peiro, P.; Martinez, B.; Melia-Segui, J.; Watteyne, T. *Understanding the Limits of LoRaWAN*. IEEE Commun. Mag. 2017, 55, 34–40.
- [8] Ochoa, M.N.; Suraty, L.; Maman, M.; Duda, A. *Large Scale LoRa Networks: From Homogeneous to Heterogeneous Deployments*. In Proceedings of the 2018 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), Limassol, Cyprus, 15–17 October 2018; pp. 192–199.
- [9] Lee, J.; Jeong, W.C.; Choi, B.C. *A Scheduling Algorithm for Improving Scalability of LoRaWAN*. In Proceedings of the 2018 International Conference on Information and Communication Technology Convergence (ICTC), Jeju Island, Korea, 17–19 October 2018; pp. 1383–1388.
- [10] T. Ferrer, S. Cespedes, and A. Becerra, “Review and Evaluation of MAC Protocols for Satellite IoT Systems Using Nanosatellites,” *Sensors*, vol. 19, p. 1947, Apr 2019.
- [11] A. Perez, P. Fabregat, M. Badia, M. Sobrino, C. Molina, J.F. Munoz-Martin, L. Fernandez, L. Rayon, and J. Ramos; 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. Available on:
[10.1109/IGARSS39084.2020.9324458](https://doi.org/10.1109/IGARSS39084.2020.9324458)

- [12] A. Perez-Portero; P. Fabregat; M. Badia; M. Sobrino; C. Molina; L. Fernandez; L. Rayón; A. Rodríguez; J. F. Munoz-Martin; *RITA: a 1u multi-sensor payload for the grsssat contributing soil moisture, vegetation analysis and RFI detection. IEEE*, 1-4 (2021)
- [13] 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.
- [14] 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 on: <http://www.sciencedirect.com/science/article/pii/S2405959517302953>
- [15] M. De Sanctis, E. Cianca, G. Araniti, I. Bisio, and R. Prasad, "Satellite communications supporting Internet of remote Things," *IEEE Internet Things J.*, vol. 3, no. 1, pp. 113–123, Feb. 2016.
- [16] Lara Fernandez; Joan Adria Ruiz-De-Azua; Anna Calveras; Adriano Camps; *Assessing LoRa for Satellite-to-Earth Communications Considering the Impact of Ionospheric Scintillation. IEEE Access (Volume 8)*, 165570 - 165582 (2020). Available on: [10.1109/ACCESS.2020.3022433](https://doi.org/10.1109/ACCESS.2020.3022433)
- [17] M. Bor and U. Roedig, "LoRa transmission parameter selection," 2017.
- [18] LoRa Modulation Bases. Available on: <https://www.mouser.com/pdfdocs/an120022.pdf>
- [19] A. Augustin, J. Yi, T. Clausen, and W. M. Townsley, "A study of LoRa: Long range & low power networks for the internet of things," *Sensors*, vol. 16, no. 9, p. 1466, 2016.
- [20] LoRaWAN airtime calculator. Available on: <https://www.thethingsnetwork.org/airtime-calculator/>
- [21] Forouzan, B.A. *Data communications and networking*. 5th ed. New York: McGraw-Hill, 2013. ISBN 0071254420.

- [22] Supriya S. Sawwashere; S.U. Nimbhorkar; RTS/CTS Frame Synchronization to Minimize the Hidden Node Problem in Wireless Network. [https://www.researchgate.net/publication/275885844 RTSCTS Frame Synchronization to Minimize the Hidden Node Problem in Wireless Network](https://www.researchgate.net/publication/275885844_RTSCTS_Frame_Synchronization_to_Minimize_the_Hidden_Node_Problem_in_Wireless_Network)
- [23] Serial connection. Available on: https://heltec-automation-docs.readthedocs.io/en/latest/general/establish_serial_connection.html
- [24] Capacitive Soil Moisture sensor schematic. Available on: <https://how2electronics.com/wp-content/uploads/2019/11/Capacitive-Soil-Moisture-Sensor-Schematic-1.png>
- [25] Texas Instrument. HDC1080 DataSheet. Available on: <https://www.ti.com/lit/ds/symlink/hdc1080.pdf>
- [26] Heltec. HelTec CubeCell Dev-Board HTCC-AB01 frequently asked questions. Available on: https://heltec-automation-docs.readthedocs.io/en/latest/cubecell/frequently_asked_questions.html
- [27] Manual 3DRobotics Iris +. Available on: <https://www.manualpdf.es/3dr/iris/manual?p=3>
- [28] Soil moisture values. Available on: <https://www.traxco.es/blog/tecnologia-del-riego/humedad-en-suelos-de-diferente-textura>
- [29] Peyravi, H. *Medium Access Control Protocols for Space and Satellite Communications*; Kent State University Kent, OH, USA, 2004.
- [30] 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>.

CHAPTER 8: Appendices

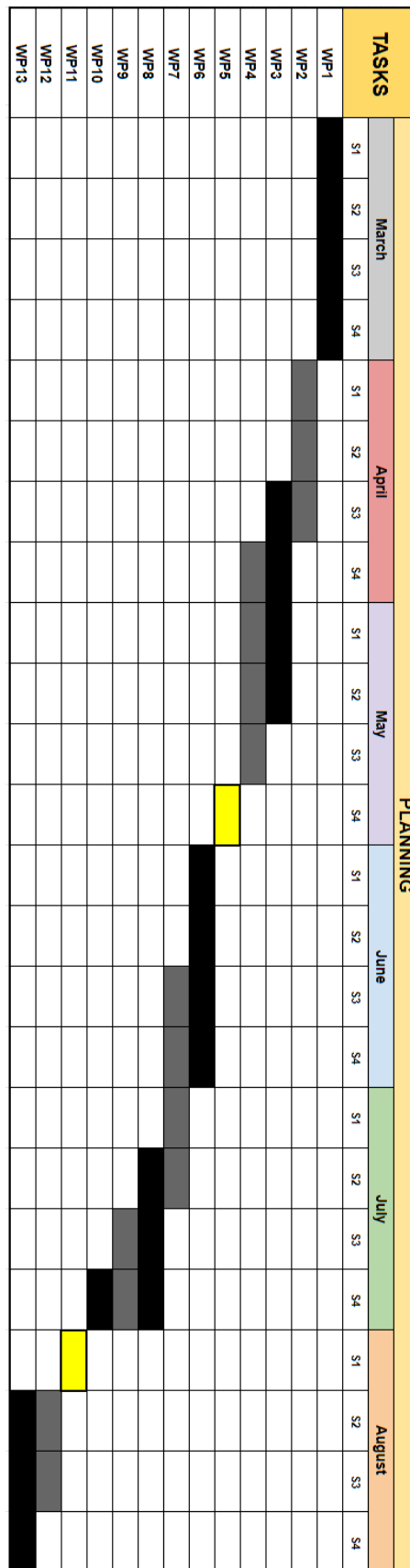
8.1. Work plan

8.1.1. Work packages

This work is divided into different work packages that have been performed over the last months. A summary of the work done in these work packages is presented below, followed by a Gantt diagram showing the time consumed in each of the work packages.

- **WP1:** Research. In this first work package research on the technologies to be applied in the project and the objectives to be achieved is performed.
- **WP2:** design of the payload of the 3D IRIS+ drone.
- **WP3:** Modifications and calibrations of the sensors. In this work package the hardware for the first measurement campaign is unified and the IP67 box for the experiment is assembled.
- **WP4:** Code implementation for the first measurement and test campaign and tests.
- **WP5:** First measurement campaign and analysis of the results.
- **WP6:** Implementation of the pure ALOHA protocol.
- **WP7:** Implementation of the CSMA/CA protocol.
- **WP8:** Design and assembly of the ground nodes.
- **WP9:** Implementation of the command code to control the experiments to be performed.
- **WP10:** Testing with the nodes and the drone payload.
- **WP11:** Second measurement campaign.
- **WP12:** Analysis of the results. In this work package the results obtained in both experiments are processed, analyzed and graphed.
- **WP13:** Documentation. Write the different sections of this work.

8.1.2. Gantt diagram



8.2. Annex A

This appendix shows particular information about the behavior of the nodes in experiment 1 and 3 using the pure ALOHA protocol.

8.2.1. Analysis of packages transmitted and received

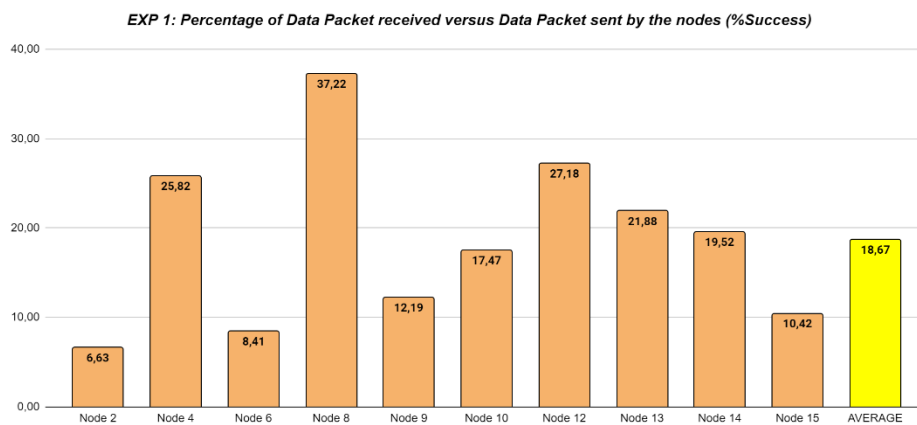


Fig. 8.1: Experiment 1 – pure ALOHA – 15 ms: Percentage of Data Packets received versus Data Packet sent by the nodes (%).

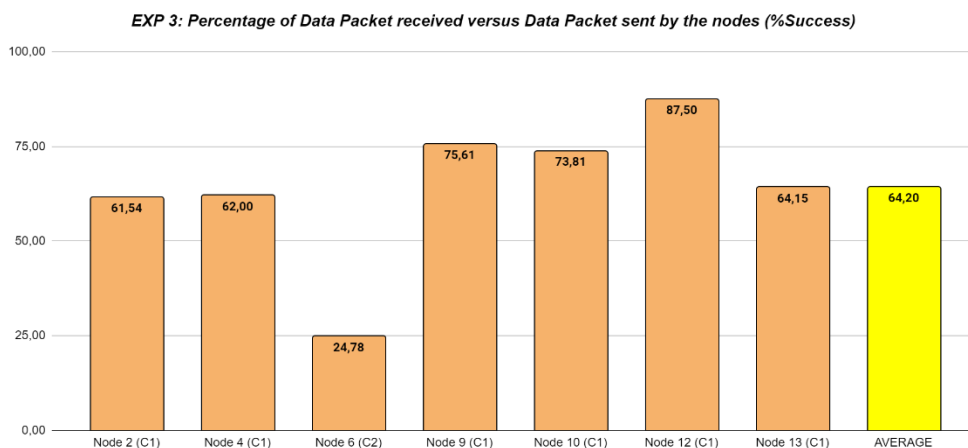


Fig. 8.2: Experiment 3 – pure ALOHA – 15 s: Percentage of Data Packets received versus Data Packet sent by the nodes (%).

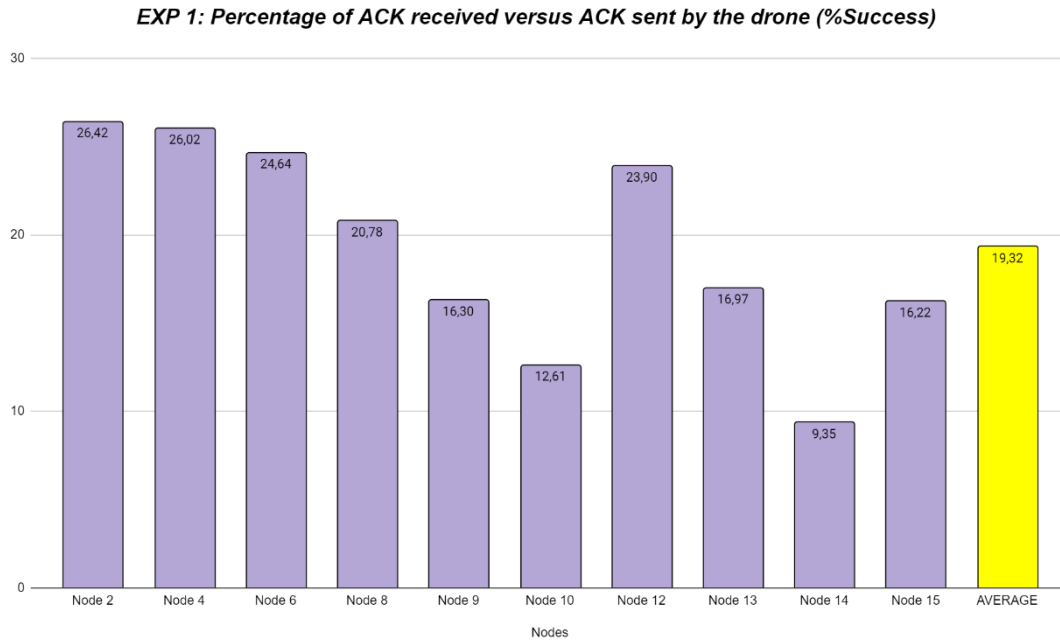


Fig. 8.3: Experiment 1 – pure ALOHA – 15 ms: Percentage of ACK received versus ACK sent by the drone (%).

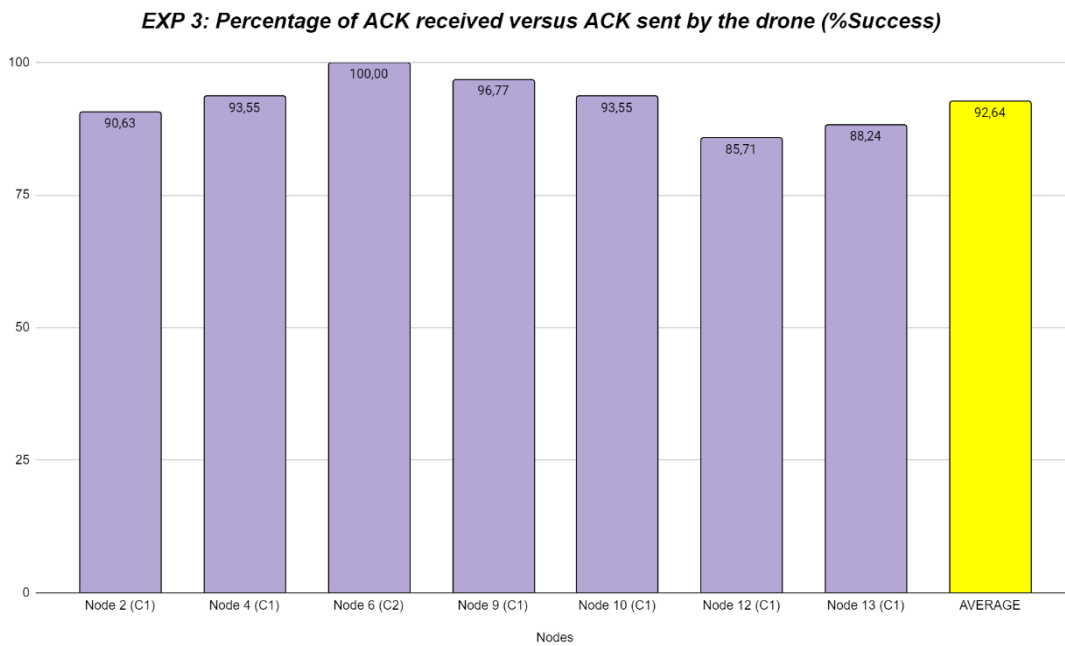


Fig. 8.4: Experiment 3 – pure ALOHA – 15 s: Percentage of ACK received versus ACK sent by the drone (%).

8.2.2. Analysis of packages received during the waiting time

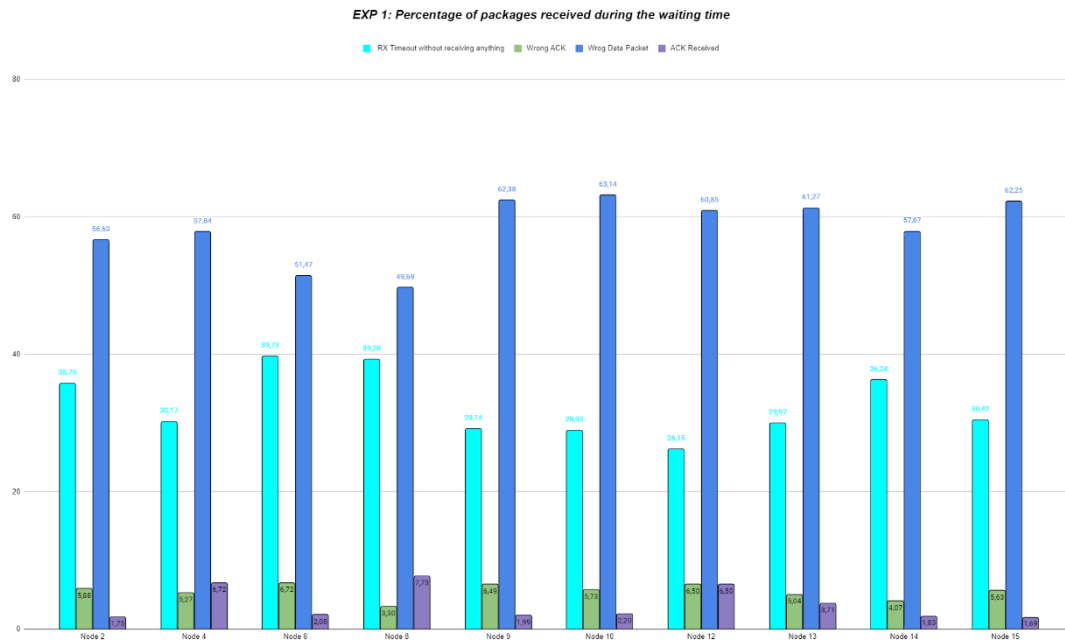


Fig. 8.5: Experiment 1 – pure ALOHA – 15 ms: Percentage of packages received during the waiting time.

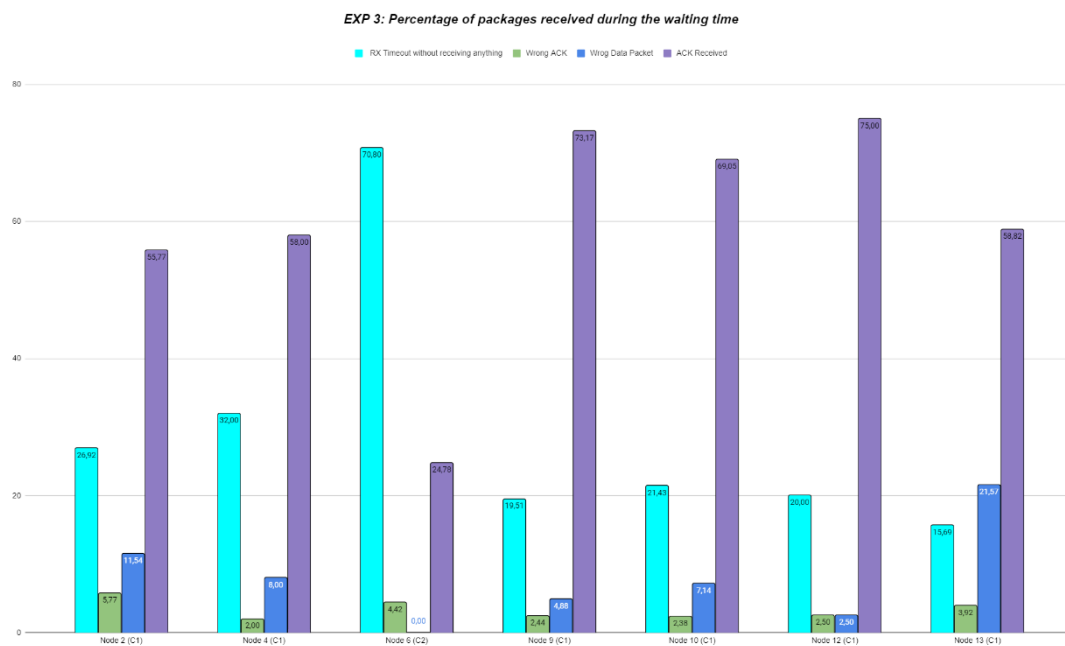


Fig. 8.6: Experiment 3 – pure ALOHA – 15 s: Percentage of packages received during the waiting time.

8.2.3. Analysis of successful communications

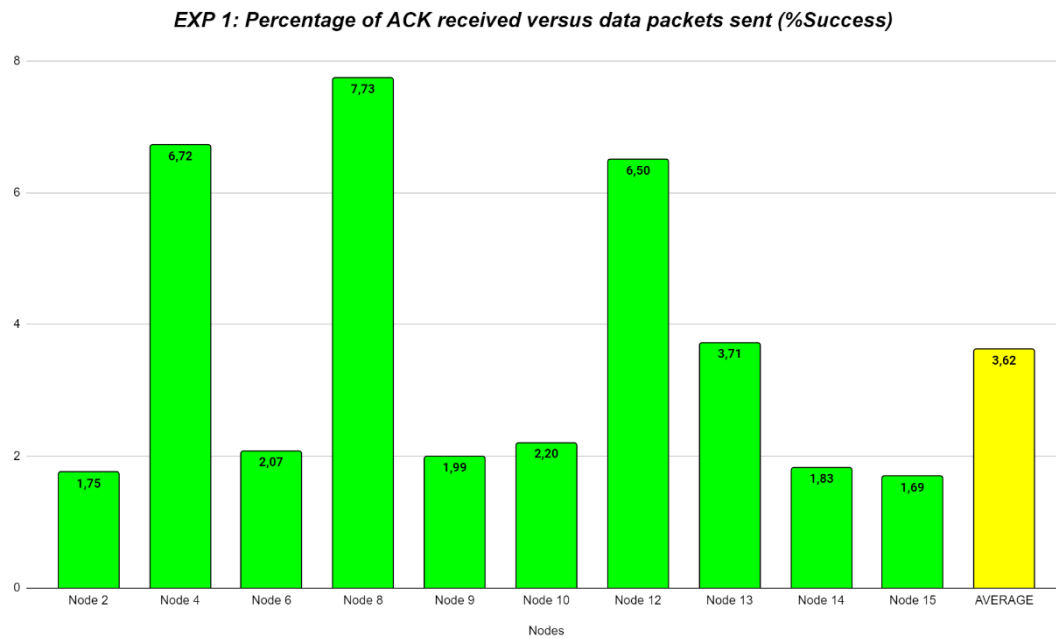


Fig. 8.7: Experiment 1 – pure ALOHA – 15 ms: Percentage of ACK received versus data packets sent.

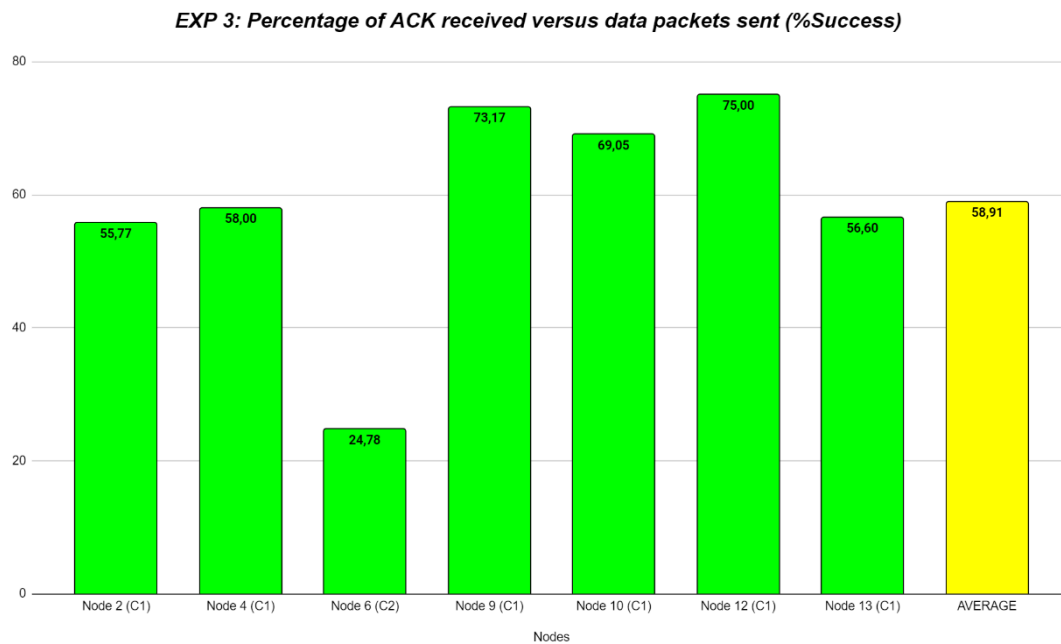
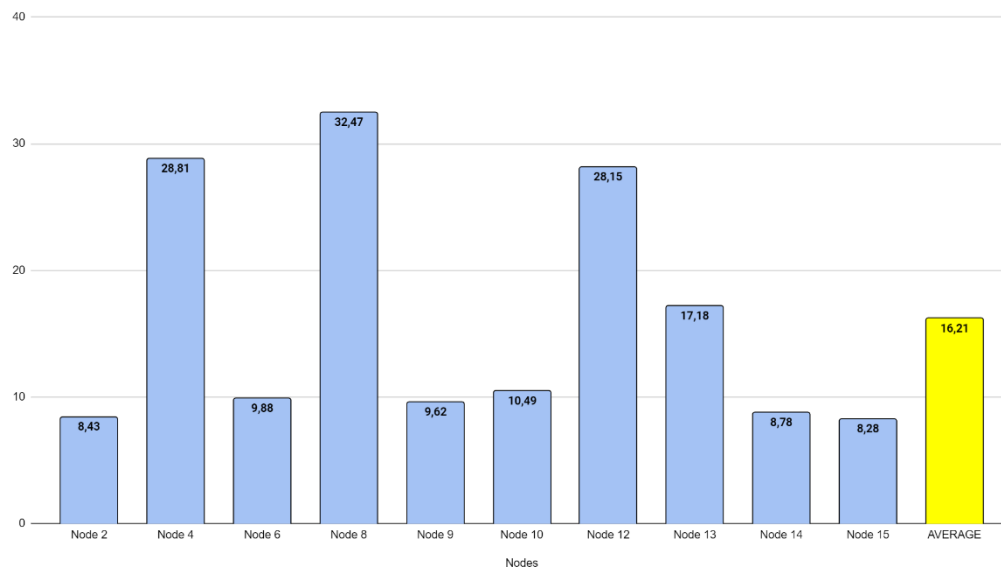
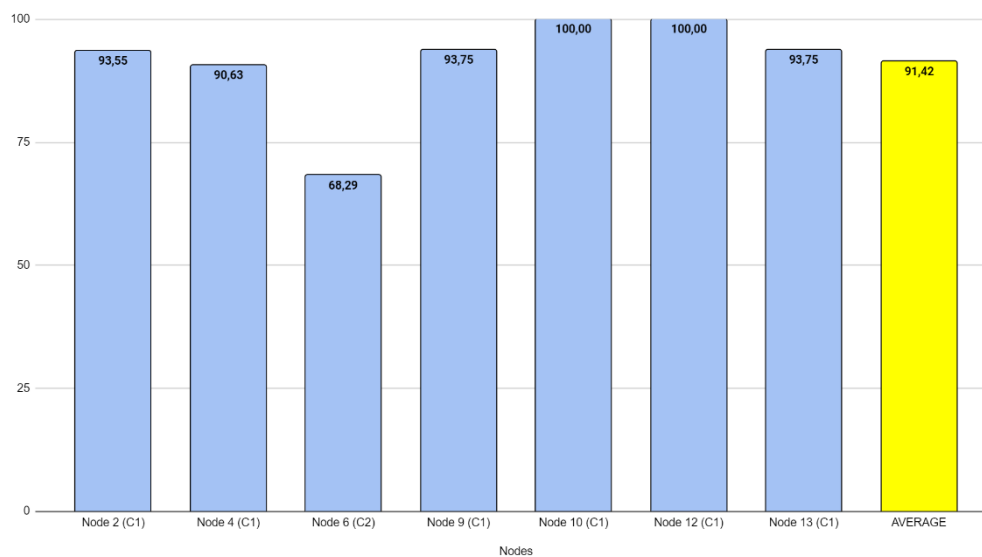


Fig. 8.8: Experiment 3 – pure ALOHA – 15 s: Percentage of ACK received versus data packets sent.

EXP 1: Percentage of successful communications versus failed communications after Kmax attempts**Fig. 8.9:** Experiment 1 – pure ALOHA – 15 ms: Percentage of successful communications versus failed communications after Kmax attempts*EXP 3: Percentage of successful communications versus failed communications after Kmax attempts***Fig. 8.10:** Experiment 3 – pure ALOHA – 15 s: Percentage of successful communications versus failed communications after Kmax attempts

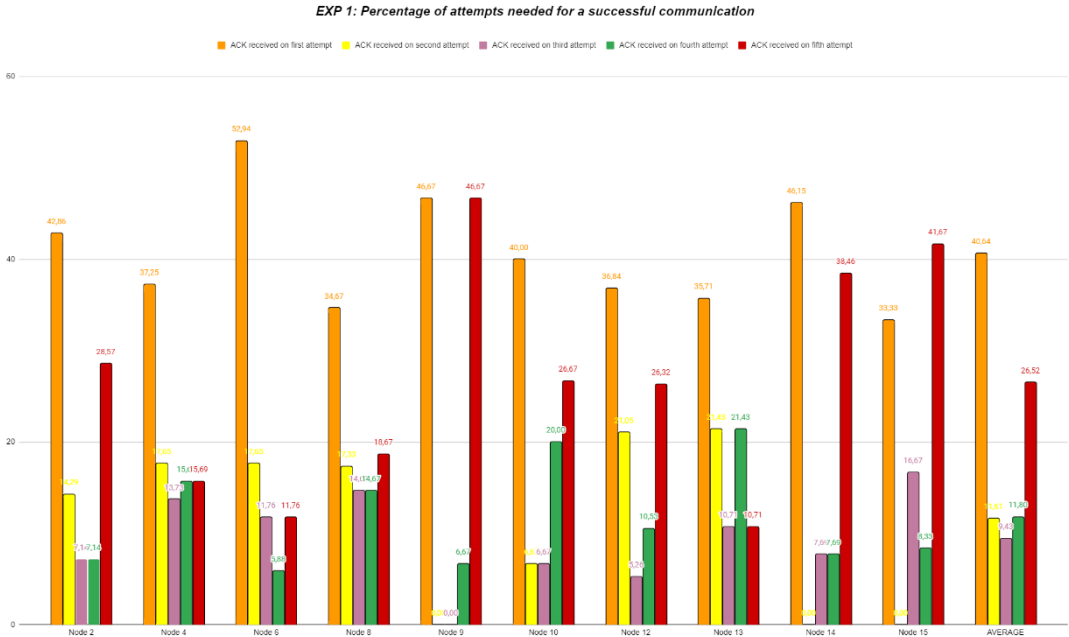


Fig. 8.11: Experiment 1 – pure ALOHA – 15 ms: Percentage of attempts needed for a successful communication.

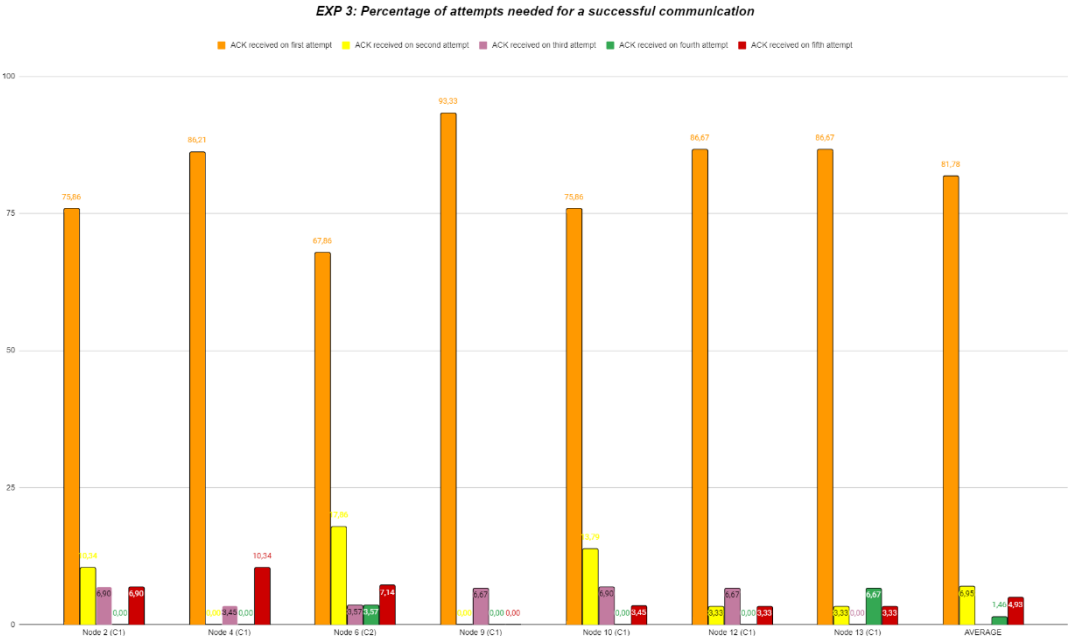


Fig. 8.12: Experiment 3 – pure ALOHA – 15 s: Percentage of attempts needed for a successful communication.

8.3. Annex B

This appendix shows particular information about the behavior of the nodes in experiment 4 and 6 using the CSMA/CA protocol.

8.3.1. Analysis of packages transmitted and received

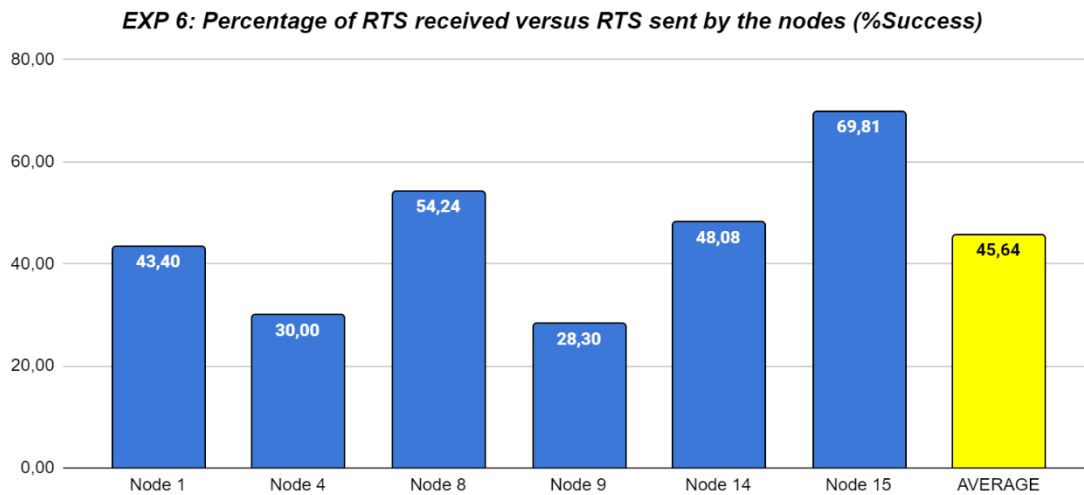


Fig. 8.13: Experiment 6 – CSMA/CA – 1 ms – Percentage of RTS received versus RTS sent by the nodes

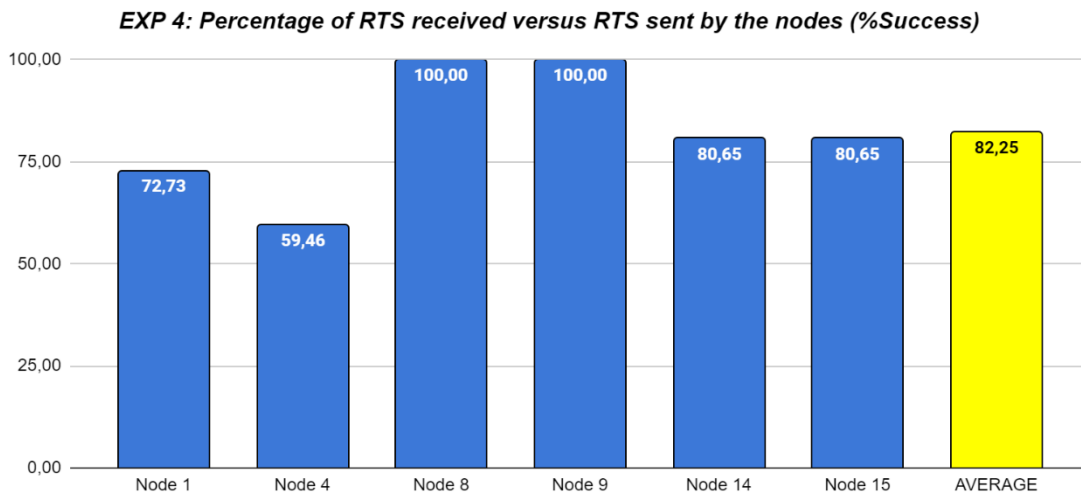


Fig. 8.14: Experiment 4 – CSMA/CA – 15 s – Percentage of RTS received versus RTS sent by the nodes

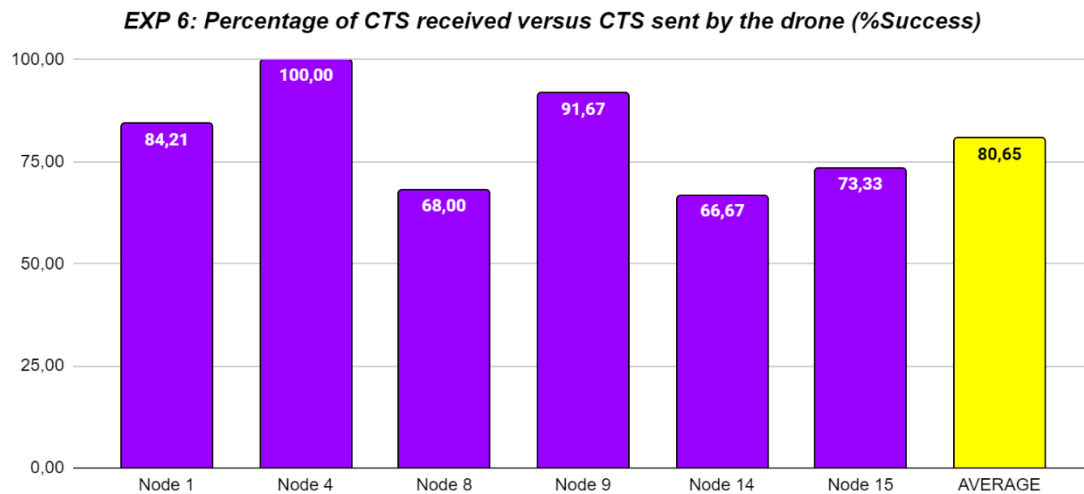


Fig. 8.15: Experiment 6 – CSMA/CA – 1 ms – Percentage of CTS received versus CTS sent by the drone

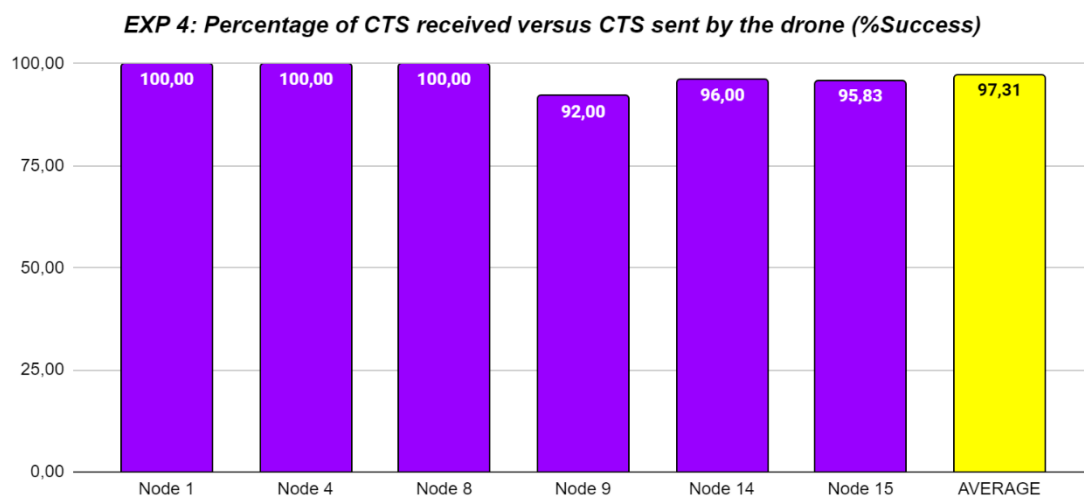


Fig. 8.16: Experiment 4 – CSMA/CA – 15 s – Percentage of CTS received versus CTS sent by the drone

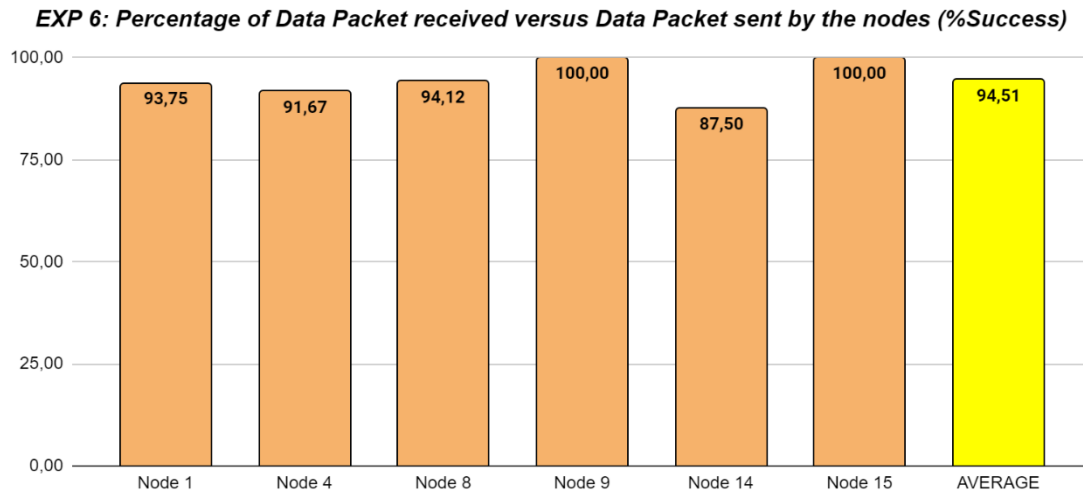


Fig. 8.17: Experiment 6 – CSMA/CA – 1 ms – Percentage of Data Packet received versus Data Packet sent by the nodes

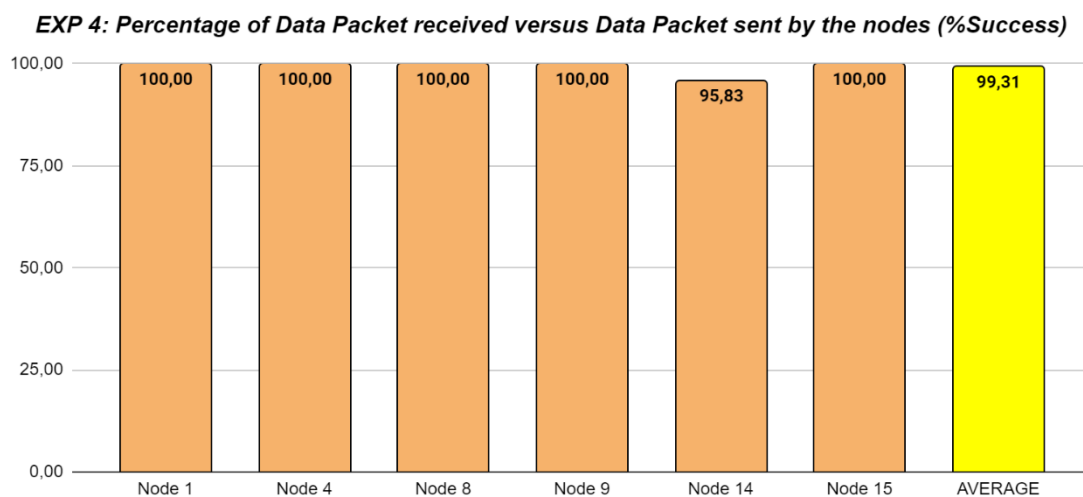


Fig. 8.18: Experiment 4 – CSMA/CA – 15 s – Percentage of Data Packet received versus Data Packet sent by the nodes

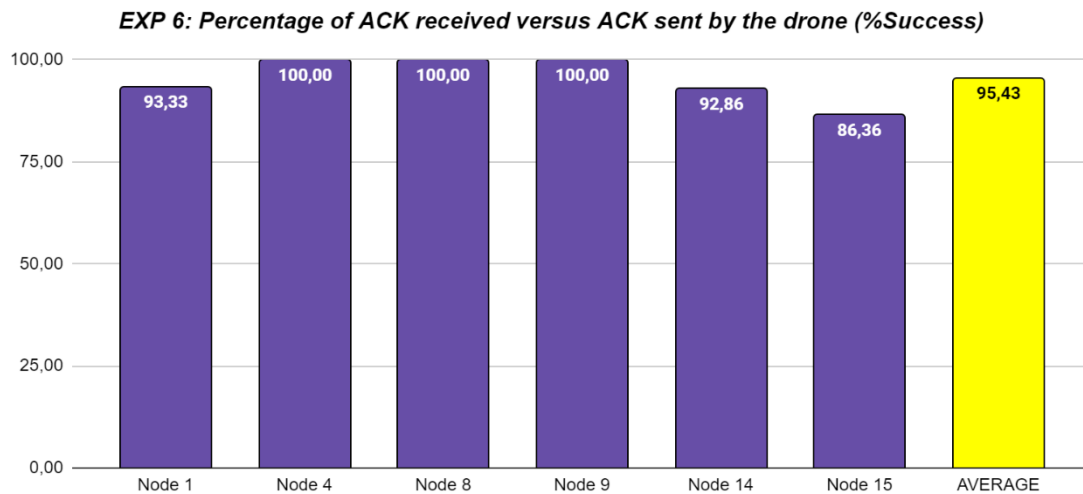


Fig. 8.19: Experiment 6 – CSMA/CA – 1 ms – Percentage of ACK received versus ACK sent by the drone

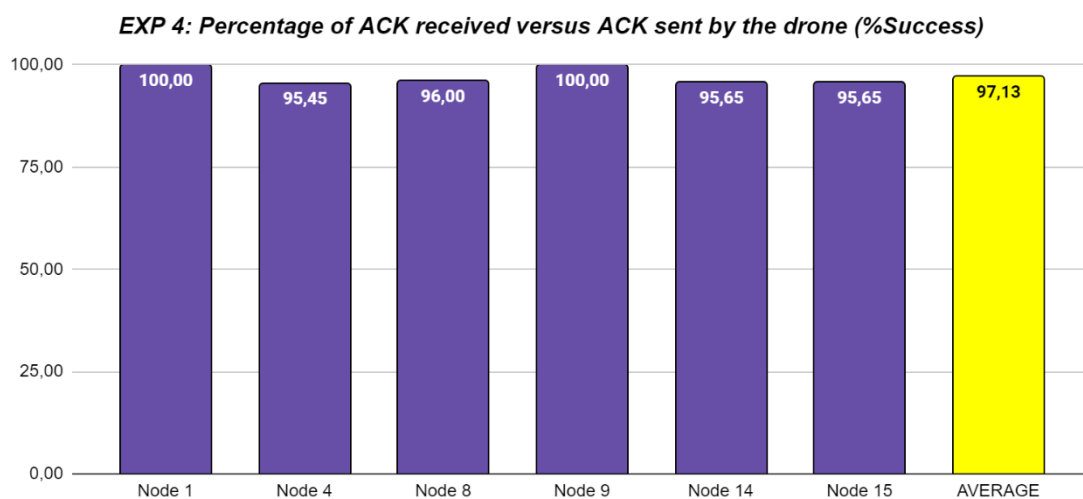


Fig. 8.20: Experiment 4 – CSMA/CA – 15 s – Percentage of ACK received versus ACK sent by the drone

8.3.2. Analysis of packages received during waiting times



Fig. 8.21: Experiment 6 – CSMA/CA – 1 ms – Percentage of packages received during the waiting time to receive the CTS

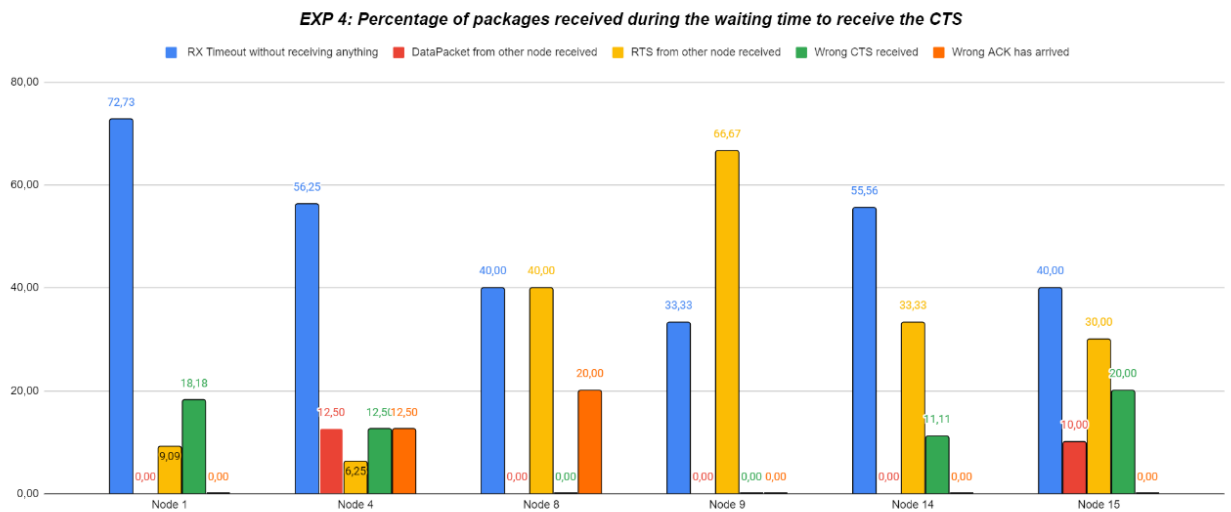


Fig. 8.22: Experiment 4 – CSMA/CA – 15 s – Percentage of packages received during the waiting time to receive the CTS

8.3.3. Analysis of successful communications

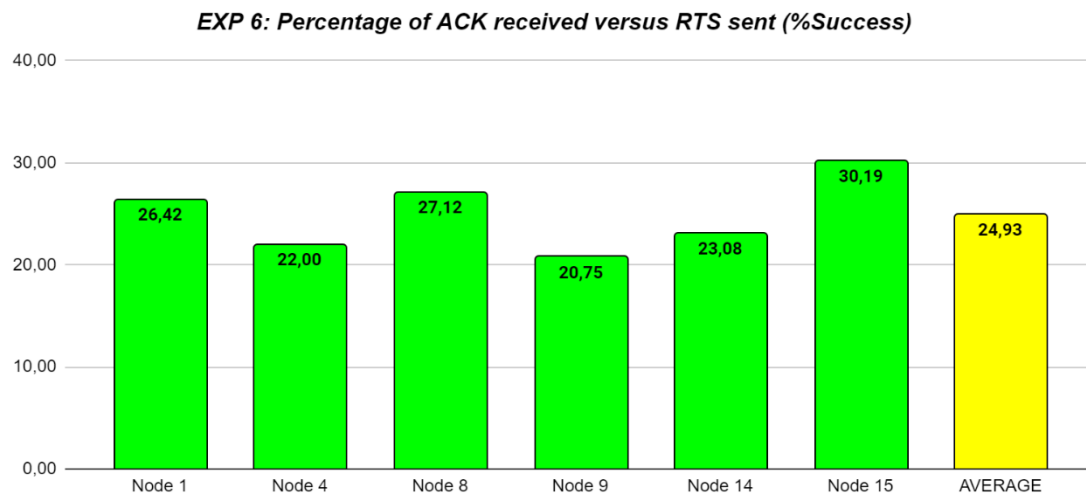


Fig. 8.23: Experiment 6 – CSMA/CA – 1 ms: Percentage of ACK received versus RTS sent.

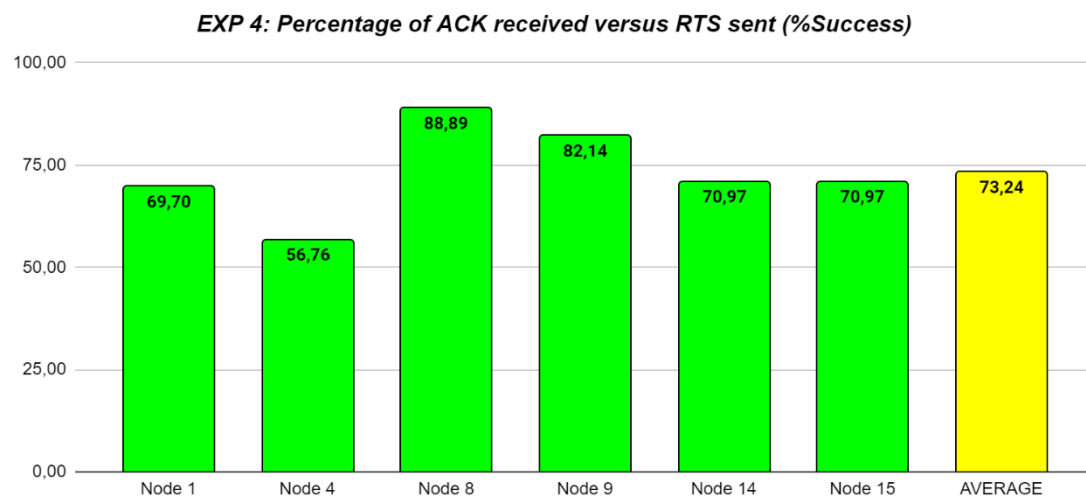


Fig. 8.24: Experiment 4 – pure ALOHA – 15 s: Percentage of ACK received versus RTS sent.

EXP 6: Percentage of successful communications versus failed communications after Kmax attempts

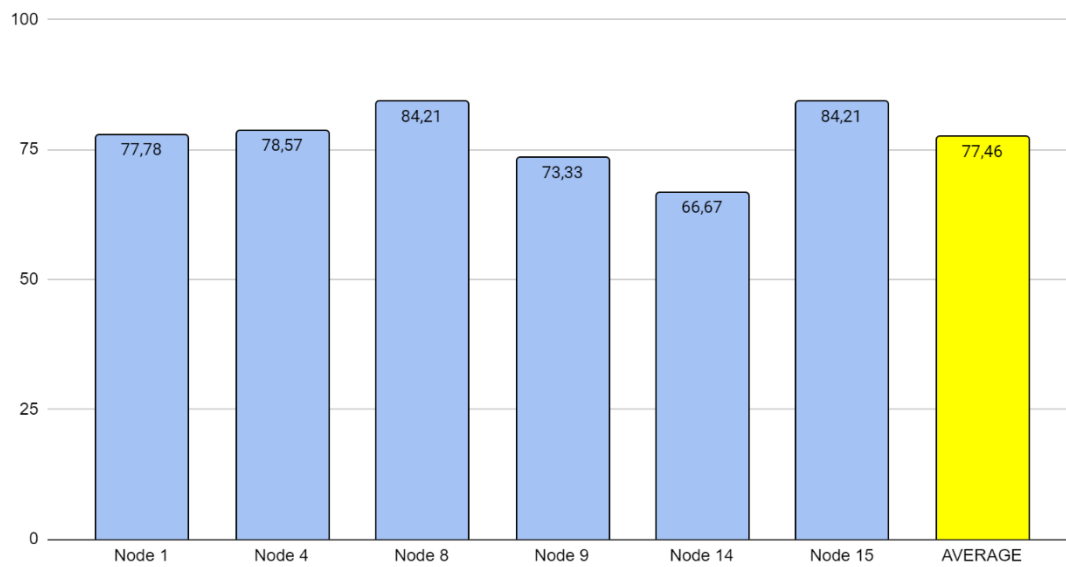


Fig. 8.25: Experiment 6 – CSMA/CA – 1 ms: Percentage of successful communications versus failed communications after Kmax attempts

EXP 4: Percentage of successful communications versus failed communications after Kmax attempts

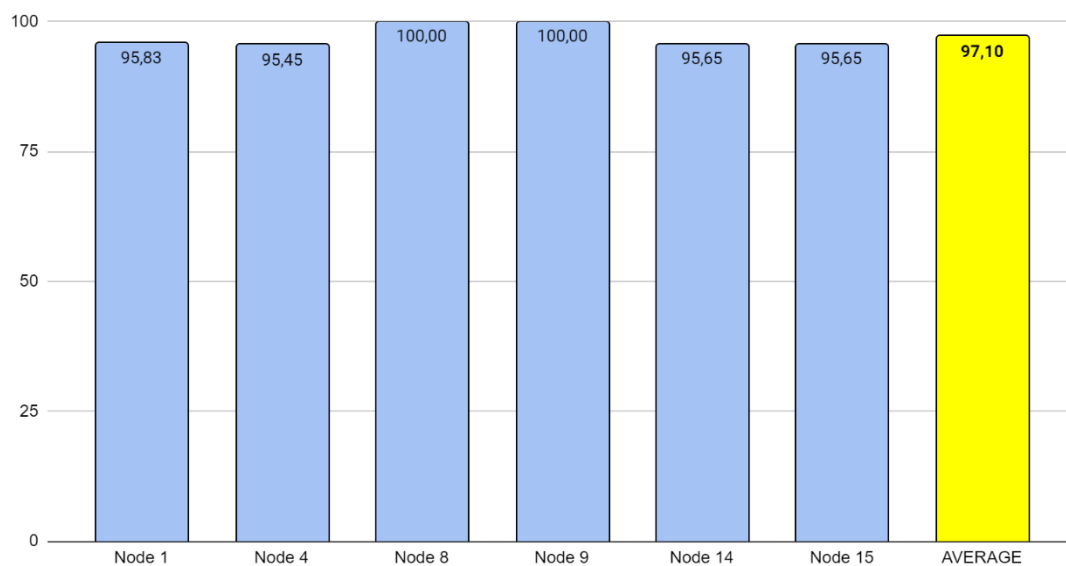


Fig. 8.26: Experiment 4 – CSMA/CA – 15 s: Percentage of successful communications versus failed communications after Kmax attempts

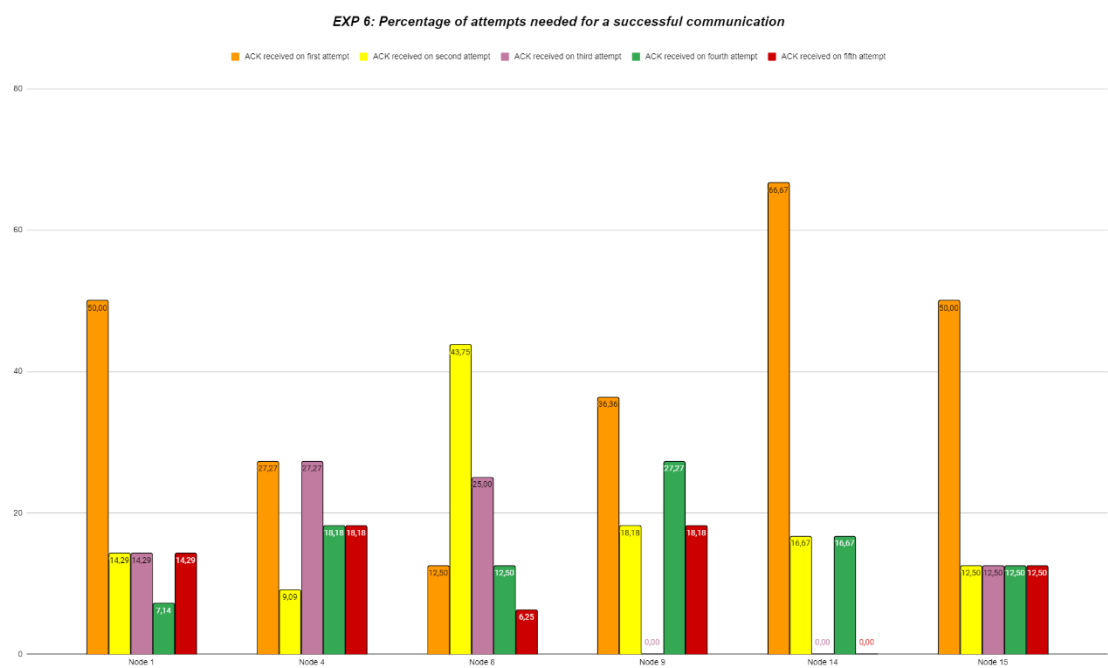


Fig. 8.27: Experiment 6 – CSMA/CA – 1 ms: Percentage of attempts needed for a successful communication

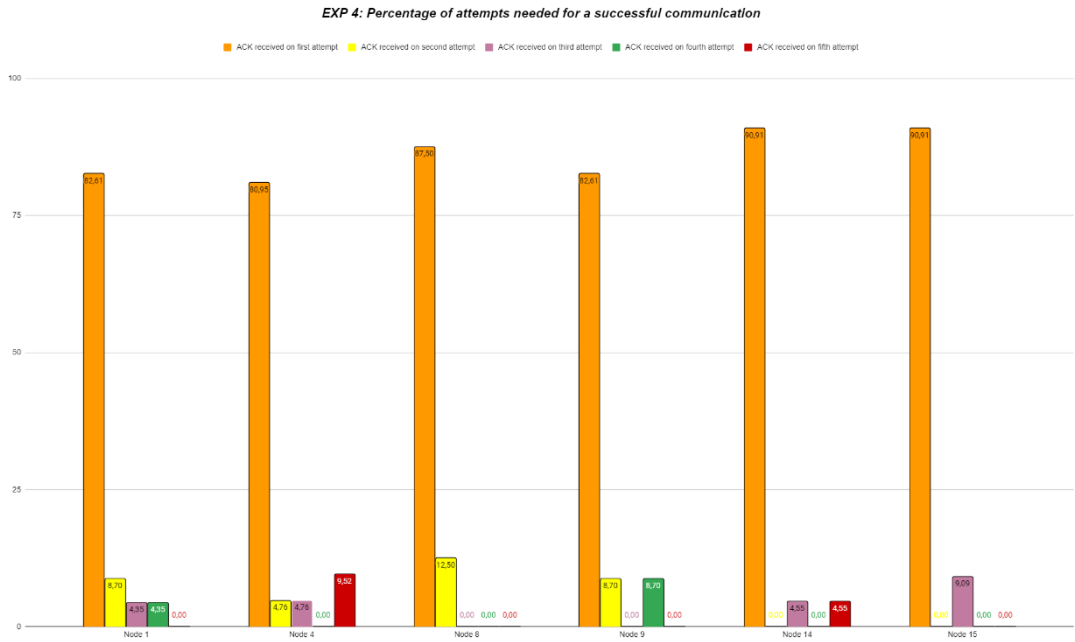


Fig. 8.28: Experiment 4 – CSMA/CA – 15 s: Percentage of attempts needed for a successful communication

8.4. Code

The code is uploaded on GitHub. In GitHub you can find 4 files (*.ino*). ALOHA_COM.ino and CSMA_CA_COM.ino are the commands used to initialize the experiments with pure ALOHA and CSMA/CA in addition to sending the characteristics of each one. CODE_Of_THE_DRONE.ino is the code implemented in the CubeCell of the drone. And Ground_Code_Node_1.ino is the code implemented in the different nodes of the experiment. Each node is loaded with the same code only varying certain attributes of the node, such as the identifier and the position.

https://github.com/diegoth99/TFG_CODE.git