

Decentralized LoRa infrastructure using blockchain

MASTER THESIS

ANDREA RAR
September 2021

Thesis supervisors:

Prof. Dr. Jacques PASQUIER
and
Arnaud DURAND
Software Engineering Group

Acknowledgements

My gratitude goes to Prof. Dr. Jacques Pasquier and to Arnaud Durand for supervising this master thesis. They have provided me with the opportunity to work on this awesome subject and always pointed me in the right direction. Arnaud Durand has taken a lot of his time to always give me some very relevant advises during our really numerous and interesting discussions. I would like to especially underline his reactivity and his hard work. I also want to thank my family and friends for their continuous support during the entire duration of the project.

Abstract

Decentralization is a very popular topic due to the surge of blockchain technologies which have permitted to bring the concept in areas where it was previously unbelievable, like for example in the field of finance. This master thesis presents a proof-of-work architecture and implementation to bring decentralization into the existing LoRa infrastructure. This is achieved thanks to the development of a new protocol, that we call LoRa-MAC, which replaces the existing LoRaWAN protocol. This new protocol is built on top of existing softwares and hardware for convenience. The decentralization aspect of LoRa-MAC is made possible thanks to the deployment of a smart contract on the Ethereum blockchain and thanks to the use of asymmetric cryptography which permits to provide non-repudation. Furthermore, an extension of the project has been developed to demonstrate the new decentralized use-cases that are now allowed. This extension consists in the exchange of micropayments between the components of the LoRa-MAC architecture in a totally decentralized way in order to allow remuneration in crowd-sourced networks. A comparison of micropayments enabling technologies for the Ethereum blockchain is in addition realized.

Keywords: LoRa, LoRaWAN, LoRa-MAC, CBOR, COSE, Blockchain, Ethereum, Smart contract, Micropayment channels, Plasma, OMG Network

Table of Contents

1. Introduction	1
1.1. Motivation	1
1.2. Goal	1
1.3. Notations and Conventions	2
2. Context	3
2.1. LoRa	3
2.2. LoRaWAN	4
2.3. Deployment options	5
2.3.1. Telecommunication companies	5
2.3.2. The Things Network (TTN)	5
2.3.3. Helium	6
2.4. Decentralization	7
2.4.1. What is decentralization ?	8
2.4.2. Why is decentralization important ?	8
2.5. Blockchain as a decentralization solution	9
2.5.1. Ethereum	9
3. Background	11
3.1. Cryptography	11
3.1.1. ECDH	12
3.1.2. HKDF	12
3.1.3. AES	12
3.1.4. ECDSA	13
3.1.5. SHA	13
3.2. CBOR	13
3.3. COSE	14
3.4. UDP Packet Forwarder	17
3.5. LoPy and MicroPython	19
3.6. Asynchronous programming	20

3.7. React	21
4. Solution	22
4.1. Overview	22
4.2. LoRa-MAC protocol	23
4.2.1. Protocol architecture	23
4.2.2. Message structure	25
4.2.3. Packet structure	26
4.2.4. Blockchain	26
4.3. Remuneration using Micropayments	27
4.3.1. Ethereum layer 2 scaling solutions	28
5. Implementation	32
5.1. Introduction	32
5.2. Architecture	33
5.3. End Device	35
5.3.1. Raspberry Pi	35
5.3.2. LoPy	37
5.4. Gateway	38
5.4.1. UDP Packet Forwarder	39
5.4.2. Forwarding Network Server (FNS)	39
5.5. Blockchain	43
5.6. Server	45
5.6.1. Home Network Server (HNS)	45
5.6.2. Application Server (AS)	47
5.7. Extension: Micropayment	48
5.7.1. Forwarding Network Server (FNS)	50
5.7.2. Blockchain	52
5.7.3. Payment service	53
5.7.4. Home Network Server (HNS)	54
5.8. Outlook	55
5.9. Pycose library	56
6. Evaluation	57
6.1. Introduction	57
6.2. LoRa-MAC packet size	57
6.3. Antennas	59
6.4. Micropayment Evaluation	60
6.4.1. Use-cases	61
6.4.2. Transaction fees	61
6.5. Issues	63

6.6. Limitations	64
7. Future Work	65
8. Conclusion	66
A. LoraResolver Smart Contract	67
B. Application Server web site	70
C. Common Acronyms	74
D. License of the Documentation	76
References	77
Referenced Web Resources	79

List of Figures

2.1. OSI model of LoRa and LoRaWAN [14]	4
2.2. A typical LoRaWAN infrastructure [1]	5
3.1. The basic COSE structure [34]	14
3.2. The COSE_Encrypt0 structure [34]	15
3.3. The COSE_Sign1 structure [34]	15
3.4. System schematic of the UDP Packet Forwarder [35]	17
3.5. Sequence diagram for the upstream protocol of the packet forwarder [35]	18
3.6. Sequence diagram for the downstream protocol of the packet forwarder [35]	19
3.7. The LoPy4 [6]	20
4.1. The LoRa-MAC protocol architecture	23
5.1. Sequence diagram of the LoRa-MAC protocol	33
5.2. Picture of the <i>Raspberry Pi</i> and the <i>LoPy4</i>	35
5.3. Pictures of the <i>Gateway</i>	38
5.4. Sequence diagram with the Micropayment extension highlighted	49
5.5. Sequence diagram of the Micropayment extension	49
5.6. Final sequence diagram of the entire thesis	55
6.1. Size of the different components of a LoRa-MAC packet	58
6.2. Dimensions of three different antennas	59
B.1. Devices page	70
B.2. Modal to add a new device	71
B.3. Modal to modify the name of an existing device	71
B.4. Modal to send a message to a device	72
B.5. Messages page	72
B.6. Modal to respond to a message	72
B.7. Down page	73
B.8. Modal to modify the payload of a down message	73

List of Tables

2.1. Helium vs telecommunication companies [20]	7
4.1. Cryptographic primitives [7]	23
4.2. Message structure	25
4.3. Message type (MType)	25
4.4. Ethereum layer 2 scaling solutions [42]	28
5.1. Possible values for the fourth byte of a UDP packet	40
6.1. Size and overhead of LoRa-MAC and LoRaWAN packets	58
6.2. RSSI values measured in dBm	60
6.3. Transaction fees to send any amount of ETH on the Ethereum Mainnet .	62
6.4. Transaction (Tx) fees for micropayment channels	62
6.5. Transaction (Tx) fees for the OMG Network	63

Listings

1.1. Example of a Python code	2
3.1. COSE_Encrypt0 message containing a COSE_CounterSignature	16
3.2. CDDL fragment describing the Countersign_structure [5]	16
3.3. Example of a JSON payload in a PUSH_DATA	18
4.1. Final COSE_Encrypt0 message	26
5.1. PUSH_DATA packet	40
5.2. PULL_DATA packet	40
5.3. PULL_RESP packet	42
A.1. LoraResolver smart contract	67

1

Introduction

1.1. Motivation	1
1.2. Goal	1
1.3. Notations and Conventions	2

1.1. Motivation

LoRa is a proprietary technology but uses license-free radio frequencies. Large deployments of the network are mainly provided by big telecommunication companies which have a monopoly and represent a single point of failure in the network. Therefore, some concerns could be made about the security and the confidentiality of such a system.

The new hype about decentralization brings some new hopes for resolving this concerns. Of course, with these hopes, come new questions. Is it possible to deploy and use a device in a secure way without using a centralized network ? Is it possible for enthusiasts to create a network of gateways to compete with the big telecommunication companies ? Is it possible for the members of such a network to get paid for their participation ?

The master thesis will try to address this interrogations by creating a new MAC routing protocol on top of the LoRa modulation technique in order to manage the communications between gateways and end-node devices in a decentralized way. This new protocol, that we call LoRa-MAC, has the goal to provide a viable solution to replace the existing LoRaWAN MAC protocol. Thanks to the decentralized aspect of the new LoRa-MAC protocol, it is possible to foresee new decentralized use-cases for the LoRa technology.

1.2. Goal

The goals of the thesis were to:

1. Learn about the LoRa protocol.
2. Get familiar with the LoRa hardware and software. Besides the LoRa protocol stack, there are three important pieces of software in a LoRaWAN network: the firmware executed by the end devices, the packet router ran by the gateways that

encapsulates LoRa messages into IP packets and the application servers that receive and handle the LoRa messages.

3. Learn about compact data format standards for constrained devices such as CBOR (a compact alternative to JSON) and COSE.
4. Design a LoRa-based MAC protocol with the following features:
 - a) Basic LoRaWAN features such as device identifiers and messages types (confirmed and unconfirmed up and down messages). Upstream packets represent a communication from an end device to a server and downstream packets represent a communication from a server to an end device.
 - b) MAC-layer or transport-layer security. The design focuses on efficiency and allow digital signatures. Only valid packets should be forwarded in the network.
5. Implement the proposed protocol stack for the LoRa roles: *End Device*, *Packet Forwarder* and *Application Server*. The protocol should keep the compatibility with the existing hardware and softwares.
6. Create simple smart-contract(s) to demonstrate the use of the LoRa-MAC protocol for decentralized use-cases such as remuneration in crowd-sourced networks or integration with decentralized blockchain applications.

1.3. Notations and Conventions

- Formatting conventions:
 - Abbreviations and acronyms as follows Hypertext Transfer Protocol (HTTP) for the first usage and HTTP for any further usage;
 - All the web pages and the servers are running on localhost.
 - Code is formatted as follows:

```
1 def division(x, y):  
2     result = x / y  
3     return result
```

List. 1.1: Example of a Python code

- The work is divided into eight chapters that are formatted in sections and subsections. Every section or subsection is organized into paragraphs, signaling logical breaks.
- Figure s, Table s and Listing s are numbered inside a chapter. For example, a reference to Figure *j* of Chapter *i* will be noted *Figure i.j*.
- As far as gender is concerned, the masculine form is systematically selected due to simplicity. But, both genders are meant equally.

2

Context

2.1. LoRa	3
2.2. LoRaWAN	4
2.3. Deployment options	5
2.3.1. Telecommunication companies	5
2.3.2. The Things Network (TTN)	5
2.3.3. Helium	6
2.4. Decentralization	7
2.4.1. What is decentralization ?	8
2.4.2. Why is decentralization important ?	8
2.5. Blockchain as a decentralization solution	9
2.5.1. Ethereum	9

2.1. LoRa

Long Range (LoRa) is a proprietary low-power, wide area network, Radio Frequency (RF) modulation technology [14]. The radio modulation used by LoRa is based on spread-spectrum modulation techniques derived from the Chirp Spread Spectrum (CSS) technology [15]. The technology was developed in 2009 by Cycleo of Grenoble in France and acquired in 2012 by Semtech which is the founding member of the LoRa Alliance. Even if the technology is patented and thus proprietary, LoRa uses license-free sub-gigahertz radio frequency bands like 433 MHz, 868 MHz in Europe, 915 MHz in Australia and North America, between 865 MHz and 867 MHz in India and 923 MHz in Asia. Since the frequencies are license-free, it is possible for anyone to create private LoRa networks which are not linked to any operator. This can be done by deploying a gateway to transmit LoRa packets which has the advantage to not require any subscription.

The goal of LoRa is to enable long-range transmissions with low power consumption at a low cost. The long-range communications can go up to five kilometers in urban areas and up to 15 kilometers or more in rural areas (line of sight) [14]. The trade-off to this long-range is the data rate which is comprised between 0.3 kbit/s and 27 kbit/s depending

on the spreading factor [15]. The low power consumption at a low cost permits to have small battery-operated devices that can last many years.

The technology covers the Physical (PHY) layer of the Open Systems Interconnection (OSI) model while other technologies and protocols, such as LoRaWAN, cover the upper layers.

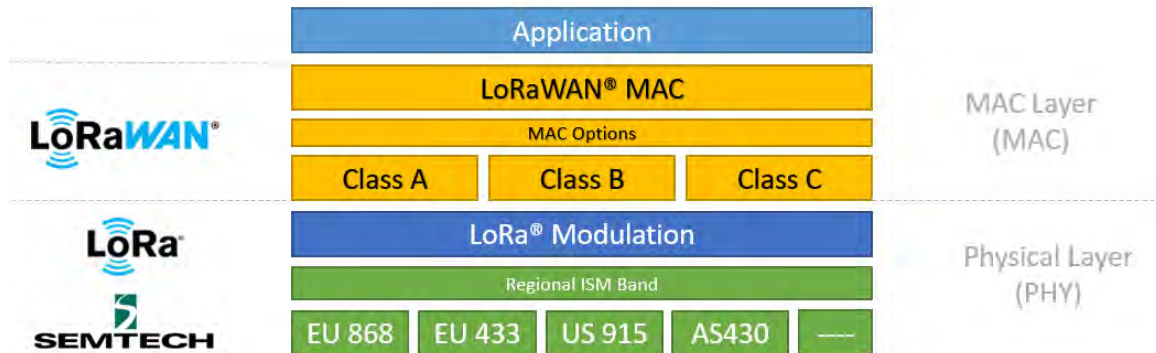


Fig. 2.1.: OSI model of LoRa and LoRaWAN [14]

2.2. LoRaWAN

Long Range Wide Area Network (LoRaWAN) is one of the several protocols that has been developed to define the upper layers of the LoRa network but it has the advantage to be maintained by the LoRa Alliance. LoRaWAN covers the Medium Access Control (MAC) layer of the OSI model. The protocol is mainly used as a network layer protocol for managing communications between Low Power Wide Area Network (LPWAN) gateways and end-node devices [15]. The end-node devices which are mainly sensors, are asynchronous and transmit only when they have data available, therefore increasing their battery life. Thanks to its attributes, the protocol is used in smart cities, industrial monitoring and agriculture.

To sum up, LoRaWAN defines the communication protocol and the system architecture of the network, while the LoRa PHY layer enables the long-range communication link. Therefore, LoRaWAN is responsible for managing the communication frequencies, the data rates and the power of all the devices. Furthermore, the protocol guarantees the confidentiality of the communication thanks to the use of symmetric cryptography in order to provide confidentiality and authenticity. Accordingly, Message Integrity Code (MIC) and end-to-end encryption of the communication using AES128 are provided. Two keys are used to provide this functionalities: the Application Key (AppKey) which is used for data confidentiality and the Network Key (NwkKey) which is used for the message integrity and the confidentiality [1].

LoRaWAN uses a star topology which implies that the data transmitted by an end-node device are received by multiple gateways. These gateways then forward the data packets without inspection to a centralized network server through the Internet and thus participate in the Internet of Things (IoT) [15]. The data are then forwarded from the central network server to the appropriate application server but, this is outside of the scope of the LoRaWAN specification [1]. When an uplink packet is received by an application server,

a downlink response packet can be sent back to the end-node by following the same route as the uplink packet. The following figure shows the just explained architecture of LoRaWAN.

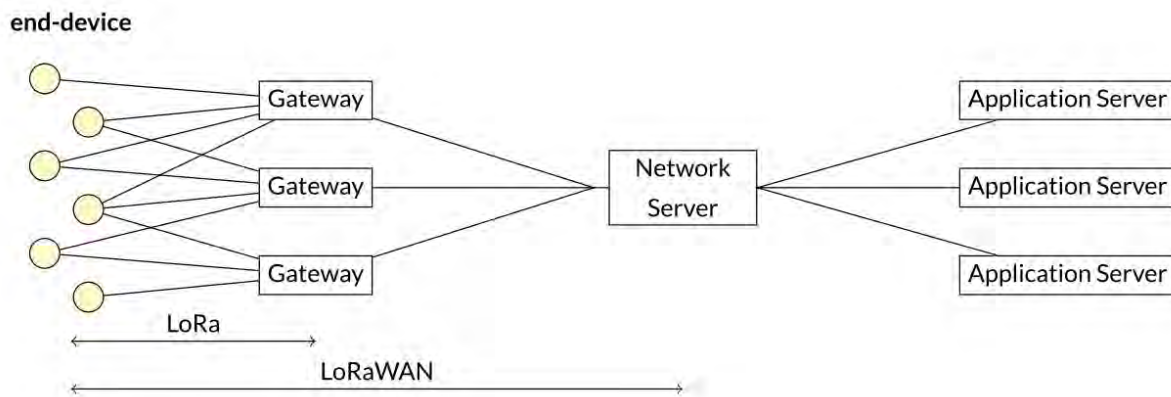


Fig. 2.2.: A typical LoRaWAN infrastructure [1]

2.3. Deployment options

The LoRa Alliance announces 156 LoRaWAN Network Operators in 171 countries as of 2021 [16]. In many countries, the network is provided by telecommunication / telephone companies like for example in Switzerland by Swisscom¹. On a global level, there exists also the network developed by The Things Network (TTN). Finally, solutions using the blockchain such as Helium have emerged.

2.3.1. Telecommunication companies

In many countries, telecommunication companies are the only local solution for users and companies who want to use LoRaWAN. These companies deploy their own gateways through the country and sell their coverage to consumers. This kind of network is totally centralized, making the users totally dependent from the gateways of the telecommunication company.

2.3.2. The Things Network (TTN)

The TTN² initiative which was started in 2015 [17] is an open community-based LoRaWAN network that is partially available in 151 countries based on a community of over 140'000 members and has over 20'000 gateways up and running. It is the world's largest networked IoT which processes several million of data sets every day, serving millions of people. The network can be used without commercial or private constraints.

Software developers of TTN develop open source LoRaWAN solutions and provide open programming tools. In addition, they also manage a global community to create IoT

¹<https://www.swisscom.ch/en/business/enterprise/offer/iot/lpn.html>

²<https://www.thethingsnetwork.org/>

applications and build collaborative IoT networks. Volunteers take on the provision, construction and support of LoRaWAN gateways [18].

Even if the TTN network is mostly free for a personal usage and the community is very large, this does not make it a decentralized system. In fact, TTN works as follow: users can deploy their own LoRa gateways and devices to be registered on the TTN web site. Once a LoRa packet is received by a gateway, it is transferred from this one to the TTN servers which will either save the packet on their database or transfer the packet to a server / application owned by the user. The user can consult the packet by login in on the TTN web site or by accessing it on his server / application. So, to sum up, the gateways are crowd sourced but the architecture / infrastructure of TTN is totally dependent from the TTN servers, thus making TTN a centralized service. In addition, the keys of the network are also managed by TTN.

2.3.3. Helium

Helium claims to be a "global, distributed network of Hotspots that create public, long-range wireless coverage for LoRaWAN-enabled IoT devices" [19]. The network was founded in 2013 with a mission to make it easier to build connected devices. At the time of writing, there are 111'702 total hotspots and thousands of ready-to-use devices. Hotspots registered in the Helium network create The People's Network. These hotspots produce and are compensated in HNT, the native cryptocurrency of the Helium blockchain. The Helium blockchain is a new blockchain built from the ground up to incentive the creation of decentralized, public wireless networks. This blockchain uses a novel work algorithm called Proof of Coverage (PoC) that is used to verify that hotspots are located where they claim to be. Put in another way, PoC tries to verify, on an ongoing basis, that hotspots are honestly representing their location and the wireless network coverage they are creating from that location. PoC has been created because the Helium network is a physical wireless network that succeeds based on the amount of reliable coverage it can create for users deploying connected devices on it. As such, it requires a work algorithm that is built for this use-case. The "challenge" is the discrete unit of work for PoC. To date, there have been 10s of millions of challenges issued and processed by the Helium blockchain. With each new challenge, the blockchain records more data about the quality of the network [19].

Like TTN, volunteers of the community can decide to host a gateway. The difference from TTN resides in the fact that there is no central server to which gateways have to deliver packets they hear over the radio spectrum. In Helium, users publish lists of interested devices they want to pay for. This list is kept on the blockchain and is constantly updated. When a gateway receives a LoRa packet, it consults the blockchain to determine where to route the packet, then it negotiates with the destination server to determine whether it wants the packet or not. In contrast to the other deployment options, this implies that participants run their own network server in addition to the application server.

The Helium packet transfer protocol is a deliver first, punish later if needed. Meaning that gateways with potential packets to deliver will contact the packet owner and ask the owner if it would like to purchase the packet. The owner can accept the packet in which case the gateway will expect a future payment to be sent. If this payment is never received, the gateway has the option to simply mark that server as unfair and refuse to deliver any future packets to it. The same protocol can work in reverse: if a gateway

repeatedly delivers fake packets that match real packets the server is interested in, the server can punish the gateway by refusing to pay for any future packets, on the belief that they will be fake. After all, the server is able to tell if the full packet is fake by checking the MIC, which can only be correctly filled in by the real device in question. The ability to memorize past transgressions is all possible because every server and gateway in the Helium network has a public identity. Gateways cannot invent new identities due to the way the onboarding process works. So, there is an incentive not to get a gateway identity banned by servers interested in purchasing packets.

Helium uses two units of exchange: HNT and Data Credits. Data Credits are bought by users in order to pay to receive packets. They are created by burning HNT. The Helium blockchain rewards hotspots in HNT for providing wireless coverage and verifying the Helium network. Every epoch, the current consensus group mines approximately 30 blocks on the blockchain. For each block, hotspots perform various types of work and are awarded. At the end of the epoch, the HNT mined are distributed to hotspots in proportion to the Data Credits received and the awarded tasks performed.

The subsequent table presents the advantages of Helium in comparison to some of the telecommunication companies that provide the LoRaWAN network in the US.

	Helium	ATT	Verizon	T-Mobile
Ability to pay based on device usage	✓	✗	✗	✗
Connectivity without sim cards or related fees	✓	✗	✗	✗
Ability to pool data across devices	✓	✗	✓	?
Unlimited data usage (no caps)	✓	✓	✗	✗
Charges for data overage	✗	✗	✓	N/A*

* No Overages Allowed

Tab. 2.1.: Helium vs telecommunication companies [20]

2.4. Decentralization

As suggested, the master thesis aims at bringing decentralization to the LoRa network in order to escape from the control of some of the network providers just mentioned. To achieve this goal, it is first necessary to understand what decentralization is, what are its benefits and what are the issues with centralization. This is why, this section will provide an explanation of this concepts and argue why they are important with the help of examples. Moreover, the section will also suggest a solution to bring decentralization in the project. The approach is based on blockchain technologies and more precisely on the Ethereum blockchain.

The informations used for this section have been retrieved mainly from the Ethereum.org web site [21] and in particular from an article about decentralization on the web site [22].

2.4.1. What is decentralization ?

"In a decentralized system, no individual or group makes decisions unilaterally. In basic terms: nobody is in charge, but things still work. Rules without rulers" [22]. The concept of decentralization is quite complex and there exist several ways to define it. But, it is nevertheless possible to list what are the desired attributes of decentralized systems:

- **Resistance to censorship:** it is not possible to unilaterally control what gets public.
- **High security:** open systems are built to resist inevitable bad actors.
- **Permissionless participation:** very basic criteria for participation are defined, often outside of the control of the system.
- **Pseudonymity or anonymity:** participants in the system can usually interact without exposing their identities.
- **Community control:** everyone who uses, operates, and maintains the system has a significant input.
- **Rules that everyone can see:** everyone can audit the system which is open source. People can copy the rules, modify them, and start new systems.
- **Deterministic:** given the same input, the rules produce the same output.
- **Public records:** a history of how the system has operated is stored by multiple participants.

2.4.2. Why is decentralization important ?

To understand why decentralization is important, it is first needed to know what are some serious risks that are associated with centralized systems:

- **Governance risk:** corporate leadership teams might take poor decisions.
- **Technical failure:** it is more likely to have problems if there is a single point of failure.
- **Economically feasible attacks:** fewer targets cost generally less to attack.

Decentralized systems can reduce or eliminate some of the risks inherent to centralized ones. Here are some examples of how:

- **Diversity:** by including diverse members and structural elements, a failure is more unlikely.
- **Dispersion:** eliminating single points of failure increases the cost of attacks.
- **Mutuality:** exploitation of other participants' results for the losses of the exploiter.
- **Distribution:** making it more difficult to obtain full control of resources and to take important decisions.
- **Integrity:** building networks of people who work towards principles of decentralization.
- **Resistance:** making unwanted behaviors impossible.

Accordingly, the goal of bringing decentralization to the LoRa network is to create a community of enthusiasts who want to provide the best possible infrastructure because all the members, not only the central authority, control and benefit from the network.

2.5. Blockchain as a decentralization solution

Since the invention of blockchain in 2008, the technology has been largely used to bring decentralization in new areas where it was previously unbelievable. In fact, "a blockchain enables trusting the output of a system without trusting anyone in particular" [1]. The technology was initially used to create the Bitcoin which is the first digital currency which does not depend on a trusted authority or a central server [23]. Afterwards, blockchains were used in other fields of finance to bring, among other things, decentralized loans. From there, the doors were open for many other projects in order to bring decentralization in the wildest domains. This is why, the blockchain technology has been retained as the solution to bring the decentralization concept to LoRa.

Before using blockchain technologies, it is mandatory to understand what they are, how they work and what are their implications. "Blockchains are tamper evident and tamper resistant digital ledgers implemented in a distributed fashion (i.e., without a central repository) and usually without a central authority (i.e., a bank, company, or government). At their basic level, they enable a community of users to record transactions in a shared ledger within that community, such that under normal operation of the blockchain network no transaction can be changed once published" [2]. The functioning of blockchain can be described in the following manner: "blockchains are a distributed ledger comprised of blocks. Each block is comprised of a block header containing metadata about the block, and block data containing a set of transactions and other related data. Every block header (except for the very first block of the blockchain) contains a cryptographic link to the previous block's header. Each transaction involves one or more blockchain network users and a recording of what happened, and it is digitally signed by the user who submitted the transaction" [2].

Accordingly, one of the largest benefit of the blockchain technology is its decentralization. However, it is important to note that not all the blockchains are decentralized. Indeed, blockchain / crypto does not equal decentralization. The fact is that most systems have aspects of centralization and decentralization. These design concepts can and do co-exist. One example of blockchain technology that has proven itself as a decentralized system is Ethereum which is the blockchain technology selected for this project.

2.5.1. Ethereum

"Ethereum is a decentralized, open-source blockchain with smart contract functionality" [24]. Ether (ETH) is the native cryptocurrency of the platform. At the time of writing, Ethereum uses a Proof of Work (PoW) consensus and is aiming to transit to a Proof of Stake (PoS) consensus. The smart contract functionality of Ethereum permits to developers to deploy permanent and immutable computer programs into the blockchain, with which users can interact. These smart contracts run on the Ethereum Virtual Machine (EVM) and can be written using the Solidity language. The strength of smart contracts is that they keep running in all cases even if the servers hosting a front-end portal to them go down. Furthermore, Ethereum has the advantage of being a trustless network. This means that there is no counterparty risk (no third party that could violate a deployed contract).

Ethereum has been selected—for this project—by taking into consideration many arguments. Indeed, first things first, nobody owns Ethereum, even the Ethereum Foundation

which provides support, but so do tons of other individuals and organizations. Therefore, Ethereum is built to succeed without centralized control or central points of failure. This is possible because it operates accordingly to rules set out in the protocol (rules that no single person or organization has the authority to change). Moreover, the technology is decentralized in many other important ways: people manage nodes across the world, the network is built to resist spammers, provide clear consensus, and render attacks economically non-viable. The activities of an user are not associated with his personal identity. Furthermore, the users do not need any permission to participate in the network and they can issue transactions instantly from anywhere only with an Internet connection. Finally, an important argument in its favor is that Ethereum is the most actively used blockchain and ETH is the largest cryptocurrency by market capitalization after Bitcoin.

3

Background

3.1. Cryptography	11
3.1.1. ECDH	12
3.1.2. HKDF	12
3.1.3. AES	12
3.1.4. ECDSA	13
3.1.5. SHA	13
3.2. CBOR	13
3.3. COSE	14
3.4. UDP Packet Forwarder	17
3.5. LoPy and MicroPython	19
3.6. Asynchronous programming	20
3.7. React	21

3.1. Cryptography

Secure communications between parties are very important. Thus, making sure that no unintended party can read or alter a message in transit is fundamental and involves the use of multiple cryptographic concepts and algorithms. This section will discuss the cryptographic primitives used in the implementation and the algorithms selected to achieve secure communications between parties.

When two entities want to send messages between them, there are usually three requirements that they want to be fulfilled:

1. **Confidentiality:** the messages are not made available or disclosed to unauthorized people.
2. **Integrity:** the messages have not been accidentally modified during the transmission.
3. **Authentication:** the receiver can be sure that the messages originate from the sender.

In network communications, confidentiality is usually achieved by using encryption. The algorithm selected to provide encryption in the implementation is Advanced Encryption Standard (AES). To ensure integrity and authentication, there are various ways that can be employed. Digital signatures will be used for this purpose because—for this project—it is important to provide non-repudation in addition to integrity and authentication. The reason for the need of non-repudation and the selection of digital signatures will be discussed in section 4.1. For now, it is just needed to know that the algorithm selected for this purpose is Elliptic Curve Digital Signature Algorithm (ECDSA).

The algorithms described in this section will require keys for their usage. Elliptic-curve keys have been chosen over non elliptic-curve keys because Elliptic-curve cryptography (ECC) allows to use smaller keys to provide an equivalent level of security to non-EC cryptography. "ECC is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields" [25]. There exist ECC versions of the famous cryptographic algorithms which permit to Elliptic curves to be used for key agreement, digital signatures, pseudo-random generators and other tasks. Furthermore, Elliptic curves can even be used indirectly for encryption by combining the key agreement with a symmetric encryption scheme [25].

3.1.1. ECDH

Elliptic-curve Diffie-Hellman (ECDH) is a variant of the Diffie-Hellman protocol that uses elliptic-curve cryptography. "It is a key agreement protocol that allows two parties, each having an elliptic-curve public-private key pair, to establish a shared secret over an insecure channel. This shared secret may be directly used as a key, or to derive another key. The key, or the derived key, can then be used to encrypt subsequent communications using a symmetric-key cipher" [26].

To generate the shared secret, a party needs to use his own private key and the public key of the other party. By doing this process, both parties are, at the end, in the possession of the same shared secret.

3.1.2. HKDF

"HKDF is a simple Key Derivation Function (KDF) based on HMAC message authentication code. The main approach HKDF follows is the "extract-then-expand" paradigm, where the KDF logically consists of two modules: the first stage takes the input keying material and "extracts" from it a fixed-length pseudorandom key, and then the second stage "expands" this key into several additional pseudorandom keys (the output of the KDF)" [27]. Among other usages, HKDF permits to convert (normalize) shared secrets exchanged via Diffie-Hellman or ECDH into key material suitable for a utilization in encryption, integrity checking or authentication.

3.1.3. AES

"The Advanced Encryption Standard (AES), also known by its original name Rijndael is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001." AES is a symmetric-key algorithm which means that the same key is used for both encryption and decryption of the

data. "AES is available in many different encryption packages, and is the first (and only) publicly accessible cipher approved by the U.S. National Security Agency (NSA) for top secret information when used in an NSA approved cryptographic module" [28].

3.1.4. ECDSA

"Elliptic Curve Digital Signature Algorithm (ECDSA) offers a variant of the Digital Signature Algorithm (DSA) which uses elliptic curve cryptography" [29]. The algorithm uses an elliptic-curve key pair consisting of a public key and a private key. The private key is used to generate a digital signature of a content, and such a signature can be verified only by using the corresponding public key. Digital signatures permit to provide authentication, integrity and especially non-repudiation [30].

"As with elliptic-curve cryptography in general, the bit size of the public key believed to be needed for ECDSA is about twice the size of the security level, in bits. For example, at a security level of 80 bits, meaning an attacker requires a maximum of about 2^{80} operations to find the private key, the size of an ECDSA private key would be 160 bits, whereas the size of a DSA private key is at least 1024 bits. On the other hand, the signature size is the same for both DSA and ECDSA: approximately $4t$ bits, where t is the security level measured in bits, that is, about 320 bits for a security level of 80 bits" [29].

3.1.5. SHA

"The Secure Hash Algorithms (SHA) are a family of cryptographic hash functions published by the NIST as a U.S. Federal Information Processing Standard (FIPS)" [31].

"A Cryptographic Hash Function (CHF) is a mathematical algorithm that maps data of arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function which is practically infeasible to invert or reverse the computation" [32].

The number that is used after the SHA name as for example SHA256, indicates that for any input passed to the SHA algorithm, the output has a fixed length of 256 bits.

3.2. CBOR

Concise Binary Object Representation (CBOR) is "a binary data serialization format loosely based on JSON. Like JSON, it allows the transmission of data objects that contain name-value pairs, but in a more concise manner" [33]. Because of the more compact representation which results in smaller message size, it is sometimes used instead of JSON in constrained environments. This characteristic permits to increase the processing and transfer speeds but at the cost of human readability. CBOR is defined in IETF RFC 8949 [3]. Among other usages, it is the recommended data serialization layer for the Constrained Application Protocol (CoAP) IoT protocol suite and the data format on which CBOR Object Signing and Encryption (COSE) messages are based. CBOR has been developed due to an increased focus on small, constrained devices that make up the IoT. Among other advantages, CBOR uses a schema-free decoder.

3.3. COSE

CBOR Object Signing and Encryption (COSE) is a data format for concise representations of small messages and is described in RFC 8152 [4]. COSE messages can be encrypted, MAC'ed and signed. In fact, COSE describes how to create and process signatures, Message Authentication Code (MAC) operations, and encryption using CBOR for serialization. In addition, it describes a representation for cryptographic keys. Like CBOR, it is as well a more lightweight alternative to an existing solution, in this case, Javascript Object Signing and Encryption (JOSE). Since COSE is based on CBOR, it is a data format designed for small code size and small message size. In summary, COSE permits to have basic security services defined for the CBOR data format.

The basic structure of a COSE message consists of two information parts and the payload. The protected header field of the message contains information that needs to be protected. This information is taken into account during the encryption, calculation of the MAC or the signature. The unprotected header contains some information that is not protected by the cryptographic algorithms. The last element is the payload which is protected by the cryptographic algorithms.

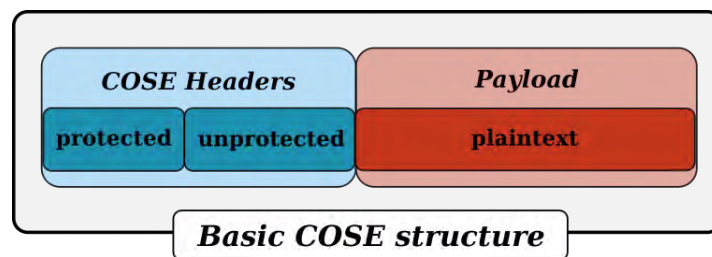


Fig. 3.1.: The basic COSE structure [34]

There are 6 different types of COSE messages [34]:

1. **Encrypt0:** an encrypted COSE message with a single recipient.
2. **Encrypt:** an encrypted COSE message that has multiple recipients.
3. **MAC0:** an authenticated COSE message with one recipient.
4. **MAC:** an authenticated COSE message that has multiple recipients.
5. **Sign1:** a signed COSE message with a single signature.
6. **Sign:** a COSE message that has been signed by multiple entities.

As it will be described later in the thesis, one of the goal of the project is to give the ability to the gateways to verify the digital signatures of the messages they receive in order to transmit only valid messages. This property needs to be respected in addition to the encryption of the payload. This can be achieved on the sender side only by encrypting first the payload and then by signing the ciphertext (encrypt-then-sign). In fact, by doing the inverse (signing first the payload and then encrypting it along with the signature (sign-then-encrypt)), the gateway should also be in the possession of the symmetric key in addition to the public key. However, to guarantee the confidentiality between the sender and the receiver, the symmetric key used to encrypt should only be in the possession of the sender and the receiver, and not the gateway. Thus, sign-then-encrypt is not feasible and encrypt-then-sign had to be selected.

Accordingly, the payload needs to be first encrypted and then signed. For this purpose, the COSE_Encrypt0 encryption structure is used for the encryption process because the payload will be accessed only by a single recipient.

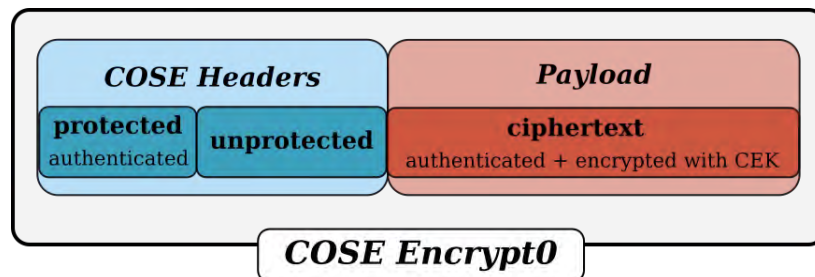


Fig. 3.2.: The COSE_Encrypt0 structure [34]

Once the COSE_Encrypt0 message has been created, a signature has to be appended to it. This cannot be done by simply passing the result of the COSE_Encrypt0 message as plaintext to a COSE_Sign1 message. In fact, this would result in having duplicated COSE elements and thus generating longer than needed messages. This element is even more important since LoRa is designed for transmitting messages as short as possible.

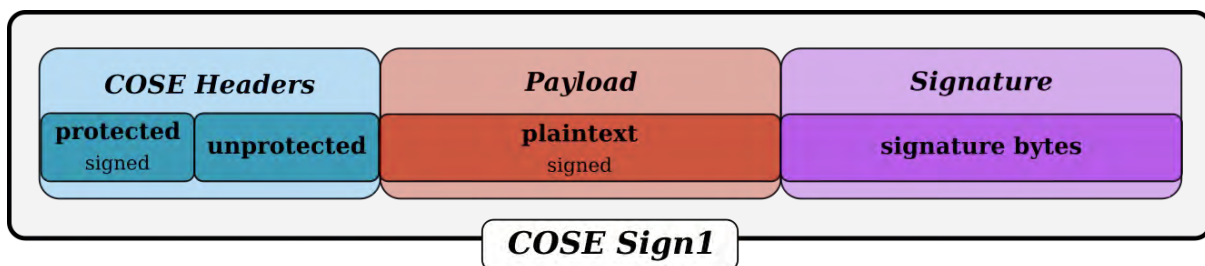


Fig. 3.3.: The COSE_Sign1 structure [34]

Therefore, in order to avoid to have some duplicated COSE elements, the COSE_Sign1 procedure cannot be applied. On the opposite, a COSE_CounterSignature needs to be appended to the existing COSE message. Unfortunately, COSE_CounterSignature is not already a final standard and is not described in RFC 8152 [4]. Instead, it is defined in the Internet-Draft COSE_CounterSign [5]. A countersignature is normally used as a second signature that confirms a primary signature. Thus, applying a COSE_CounterSignature to either a COSE_Signature or a COSE_Sign1 object match this traditional use-case. But, the Internet-Draft extends the context of a countersignature to allow it to be applied to all of the security structures defined previously. It is important to note that the countersignature needs to be considered as a separate operation even if it is applied by the same user who produced the original COSE message. In this project, the COSE_CounterSignature is appended to a COSE_Encrypt0 message. This is done by adding the COSE_CounterSignature structure inside the unprotected part of the header of the COSE_Encrypt0 message, as it is shown in the following listing.


```

1 COSE_Encrypt0(
2   [
3     protected {
4       alg : A128GCM,
5       iv : ... ,
6       reserved : header
7     },
8     unprotected {
9       kid : ... ,
10      COSE_CounterSignature : [
11        protected {
12          alg : Es256
13        },
14        unprotected {
15          kid : ... ,
16        },
17        signature
18      ]
19    },
20    ciphertext
21  ]
22 )

```

List. 3.1: COSE_Encrypt0 message containing a COSE_CounterSignature

The COSE_CounterSignature structure is almost the same as a COSE_Sign1 structure except that it does not contain a payload field since this one is already present in the COSE message to which it is appended. The signature field of the COSE_CounterSignature structure is computed over a well-defined byte string called the Countersign_structure which is a CBOR array. The fields of this Countersign_structure are in order [5]:

1. A context text string identifying the context of the signature. In this case, the context text string is "CounterSignature".
2. The protected attributes from the target structure encoded in a byte string type.
3. The protected attributes from the countersignature structure encoded in a byte string type.
4. The externally supplied data from the application, if any, encoded in a byte string type.
5. The payload to be signed encoded in a byte string type.
6. An array of all the byte string fields after the second if there are more than two byte string fields in the target structure.

```

1 Countersign_structure = [
2   context : "CounterSignature" / "CounterSignature0" /
3     "CounterSignatureV2" / "CounterSignatureOV2" /,
4   body_protected : empty_or_serialized_map,
5   ? sign_protected : empty_or_serialized_map,
6   external_aad : bstr,
7   payload : bstr,
8   ? other_fields : [ + bstr ]
9 ]

```

List. 3.2: CDDL fragment describing the Countersign_structure [5]

Once the `Countersign_structure` has been created and encoded to a CBOR array using the CBOR tag 11, it is passed as parameter along with a private key to the signature creation algorithm. This result in the signature field of the `COSE_CounterSignature` structure.

3.4. UDP Packet Forwarder

The program that we call the UDP Packet Forwarder is in fact the LoRa network packet forwarder project which is "a program running on the host of a LoRa gateway that forwards RF packets received by the concentrator to a server through a IP / UDP link, and emits RF packets that are sent by the server" [35]. The source code can be found on GitHub¹. Accordingly, the UDP Packet Forwarder can transmit either uplink packets (radio packets received by the gateway, with metadata added by the gateway, forwarded to the server) or downlink packets (packets generated by the server, with additional metadata, to be transmitted by the gateway on the radio channel). The packet forwarder program has been developed by Semtech and—for this project—is running on a Raspberry Pi 3 Model B² with a RAK831 LPWAN Gateway Concentrator Module³ mounted on top communicating through SPI. The RAK831 concentrator uses the Semtech SX1301 chip and is able to receive up to 8 LoRa packets simultaneously. Additionally, it can send with different spreading factors on different channels. The concentrator acts as the radio communication module and is the central piece for receiving and transmitting LoRa radio messages. Multiple antennas with the correct frequency for the LoRa radio have been tested. A detailed comparison of these antennas can be found in section 6.3.

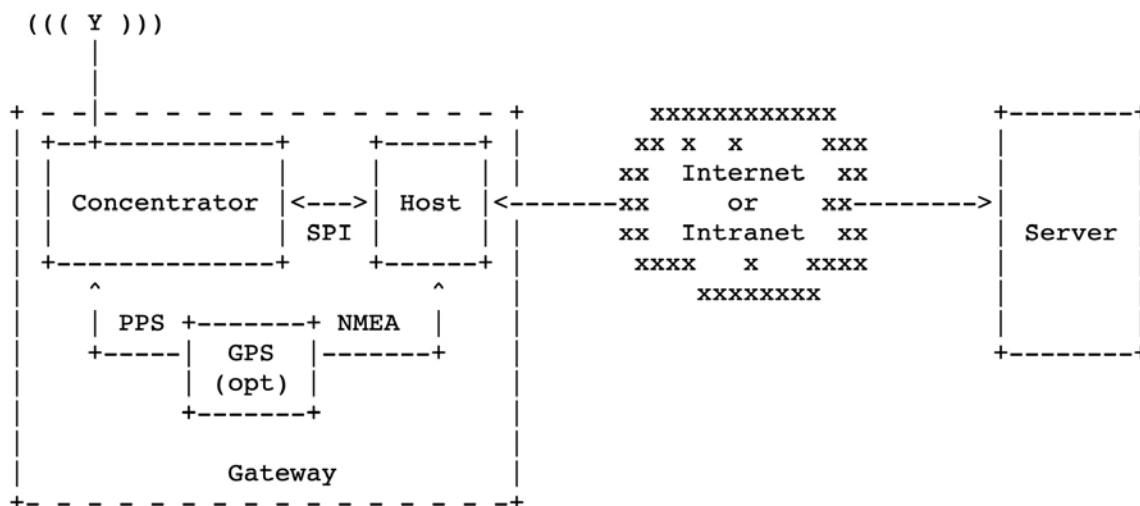


Fig. 3.4.: System schematic of the UDP Packet Forwarder [35]

The server to which the UDP Packet Forwarder transmits the packets can be deployed remotely or on the same host like in the case of this project. To configure the packet

¹https://github.com/Lora-net/packet_forwarder

²<https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>

³<https://store.rakwireless.com/products/rak831-gateway-module>

forwarder, a file called `global_conf.json` must be provided in the `lora_pkt_fwd` directory. It is important to use the configuration file that fits with the platform, region and features needed.

The protocol followed by the packet forwarder to transmit an upstream packet to a server is described by the following sequence diagram.

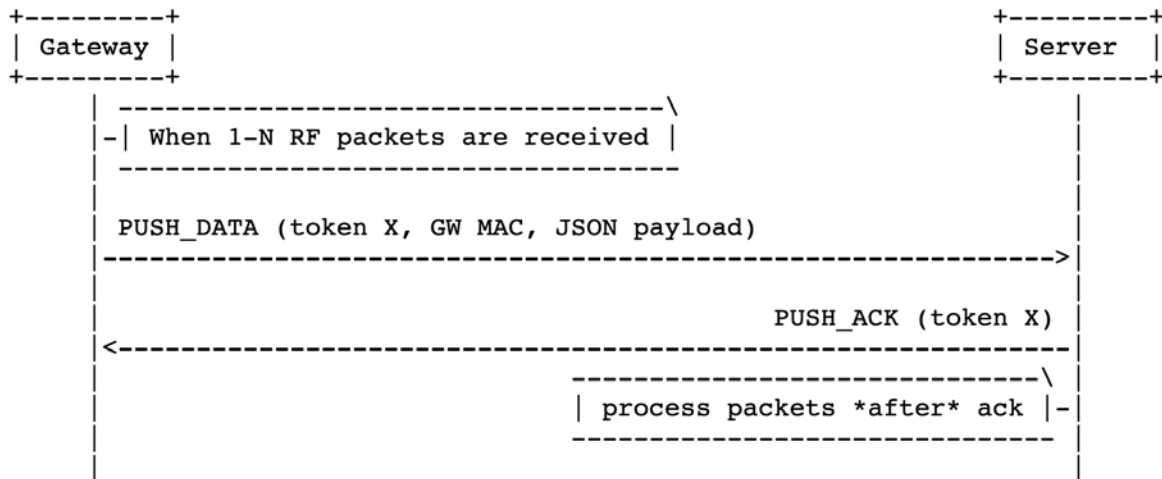


Fig. 3.5.: Sequence diagram for the upstream protocol of the packet forwarder [35]

The JSON payload in the `PUSH_DATA` contains all the major informations about a packet, including, among other things, the time, the frequency, the size of the payload and the payload.

```

1 {"rxpk": [
2   {
3     "tmst" : 1525043316,
4     "chan" : 5,
5     "rfch" : 0,
6     "freq" : 867.500000,
7     "stat" : 1,
8     "modu" : "LORA",
9     "datr" : "SF12BW125",
10    "codr" : "4/7",
11    "lsnr" : 5.8,
12    "rssi" : -15,
13    "size" : 5,
14    "data" : "aGVsbG8="
15  }
16 ]}
  
```

List. 3.3: Example of a JSON payload in a `PUSH_DATA`

The downstream protocol that a server has to follow to transmit a UDP packet back to the packet forwarder which will then emit it through LoRa is described in the underneath sequence diagram.

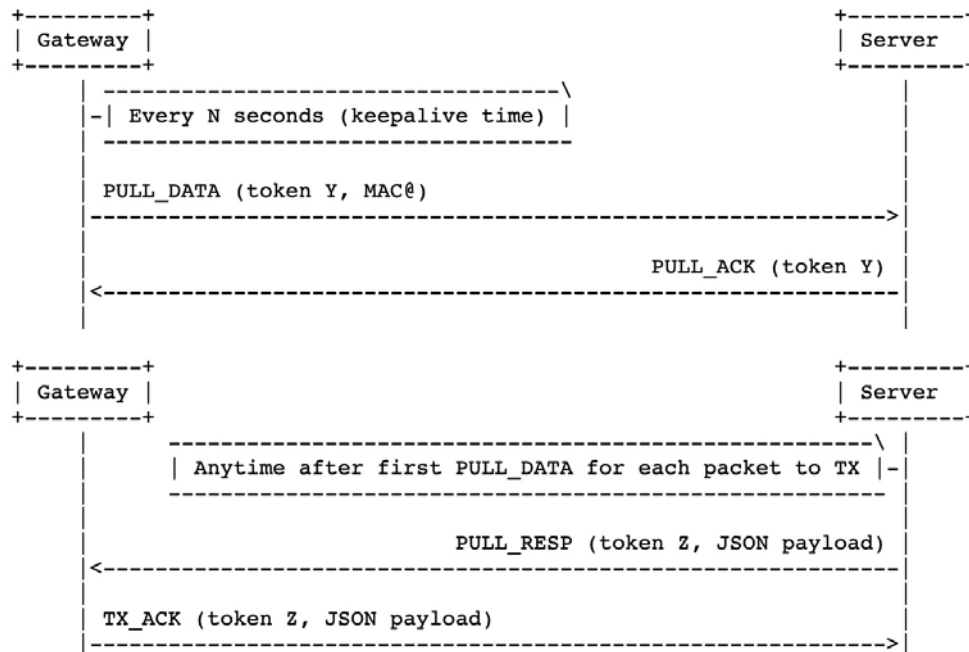


Fig. 3.6.: Sequence diagram for the downstream protocol of the packet forwarder [35]

The JSON payload in the PULL_RESP has to follow the same structure as the JSON payload in a PUSH_DATA packet.

The packet forwarder and the Raspberry Pi 3 have been selected because—for this project—it is important to use standard and common components in order to keep the compatibility with existing hardware and softwares used in the LoRaWAN protocol.

3.5. LoPy and MicroPython

A LoPy is a compact MicroPython enabled development board commercialized by Pycom. The model used—for this project—is a LoPy4⁴ which is a quadruple network board (LoRa, Sigfox, WiFi, Bluetooth). The LoPy4 is equipped with the Espressif ESP32 chipset and the Semtech LoRa transceiver SX1276. It can act as a LoRa nano gateway and a multi-bearer development platform suitable for all LoRa and Sigfox networks around the world. One of the main advantage of the LoPy4 is that it can be configured in raw LoRa mode to send packets directly between LoRa compatible devices. Another advantage is its ultra-low power usage. To connect a LoPy via USB to a computer, it is possible to mount it on an expansion board [6]. Before using a LoPy, it is recommended to first upgrade its firmware to the latest update. A getting started guide can be found on the Pycom documentation [36].

⁴<https://pycom.io/product/lopy4/>

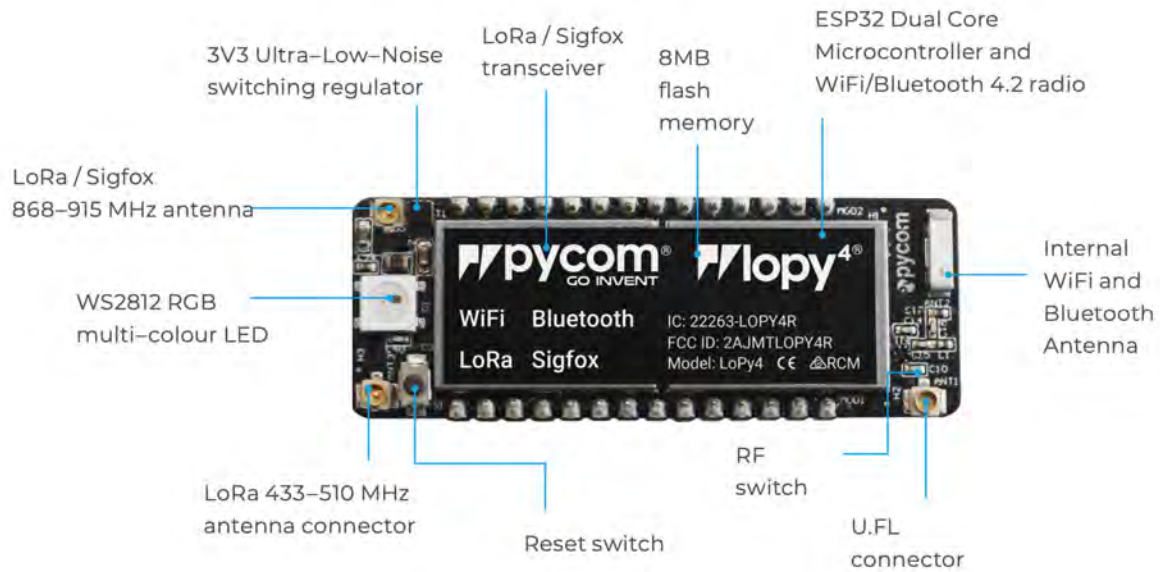


Fig. 3.7.: The LoPy4 [6]

The LoPys are programmable with the MicroPython programming language and the Pymark plugins for fast IoT application development and easy programming in-field. MicroPython is a software implementation of a programming language largely compatible with Python 3, written in C, that is optimized to run on a microcontroller [37]. MicroPython consists in a full Python compiler and runtime. The user can either access it through an interactive prompt (the REPL) to execute supported commands immediately or by uploading MicroPython files (.py files) on the microcontroller. A selection of core Python libraries are included in MicroPython. Additionally, MicroPython includes modules which give the programmer access to low-level hardware. A certain number of MicroPython libraries including a LoRa library and a Serial library, have been developed by Pycom in order to be used on the LoPys.

3.6. Asynchronous programming

Since the project aims to create an architecture / infrastructure with interactions between multiple components, it is important to develop this components using asynchronous, non-blocking programming. Asynchronous programming is a style of concurrent programming which means making many things at once. The way to do it is by using the processor at its maximum by liberating it while slower tasks are taking place. In fact, tasks release the CPU during waiting periods so that other tasks can use it. This permits to asynchronous code to wait until a response to a request is returned before it tries to do anything else while some other code could be executed in the meantime [7].

Asyncio is a Python library to write concurrent code using the `async / await` syntax. The library is used as a foundation for multiple Python asynchronous frameworks that provide for example high-performance network and web-servers, database connection libraries, distributed task queues, etc [38].

In JavaScript, there are three main types of asynchronous code style: callbacks, promises and finally, async functions and the `await` keyword which have been added in ECMAScript

2017 [39]. The JavaScript code developed for this project will use the last type of asynchronous code style which basically acts as syntactic sugar on top of promises, making asynchronous code easier to write and to read afterwards [39].

3.7. React

"React (also known as React.js or ReactJS) is a free and open-source front-end JavaScript library for building user interfaces or UI components. It is maintained by Facebook and a community of individual developers and companies" [40]. Each web page developed with React can be implemented in its own component. A component can be, for example, a `.jsx` file which contains both JavaScript and HTML code inside of it. To define the style of one or multiple components, it is possible to put the Cascading Style Sheet (CSS) code inside a separated CSS file.

4

Solution

4.1. Overview	22
4.2. LoRa-MAC protocol	23
4.2.1. Protocol architecture	23
4.2.2. Message structure	25
4.2.3. Packet structure	26
4.2.4. Blockchain	26
4.3. Remuneration using Micropayments	27
4.3.1. Ethereum layer 2 scaling solutions	28

4.1. Overview

LoRaWAN security model provides confidentiality and authenticity but does not provide non-repudiation. This makes it unsuitable for some decentralized use-cases such as remuneration in crowd-sourced networks or integration with decentralized blockchain applications.

Accordingly, one of the goal of the project is to add non-repudiation in addition to authentication to the LoRa network. Authentication has been defined in chapter 3.1 and permits to ensure to the receiver that the messages originates from the sender. Non-repudiation on the other hand, permits to ensure to a third party that the message really originates from the sender.

As shown in Tab. 4.1, non-repudiation can be achieved using asymmetric keys and digital signatures instead of symmetric keys and MAC (Message Authentication Code) which are actually used in LoRaWAN. In addition, asymmetric cryptography can also provide confidentiality by using for example HKDF, which thanks to the private key of the sender and the public key of the receiver can generate a symmetric key. Using asymmetric cryptography and digital signatures in the LoRa network is possible only by building a new MAC protocol that replaces the existing LoRaWAN protocol. This new protocol, that we call LoRa-MAC, is going to be described in more details in this chapter.

Thanks to the fact that LoRa will now have a MAC protocol that uses asymmetric cryptography, it is possible to imagine a fully decentralized version of LoRa communications.

In fact, combining this new property with the use of blockchain will permit to go even further in the decentralization aspect than Helium. It will, for example, be possible for gateways to verify that they forward only valid LoRa packets from which they will really be paid and, it will be possible for servers to ensure in advance that they only pay for valid packets. As with Helium, the users of LoRa-MAC will have to run their own network server in addition to the application server.

Security Goal	Cryptographic primitive		
	Hash	MAC	Digital signature
Integrity	✓	✓	✓
Authentication	✗	✓	✓
Non-repudiation	✗	✗	✓
Kind of keys	None	Symmetric keys	Asymmetric keys

Tab. 4.1.: Cryptographic primitives [7]

4.2. LoRa-MAC protocol

4.2.1. Protocol architecture

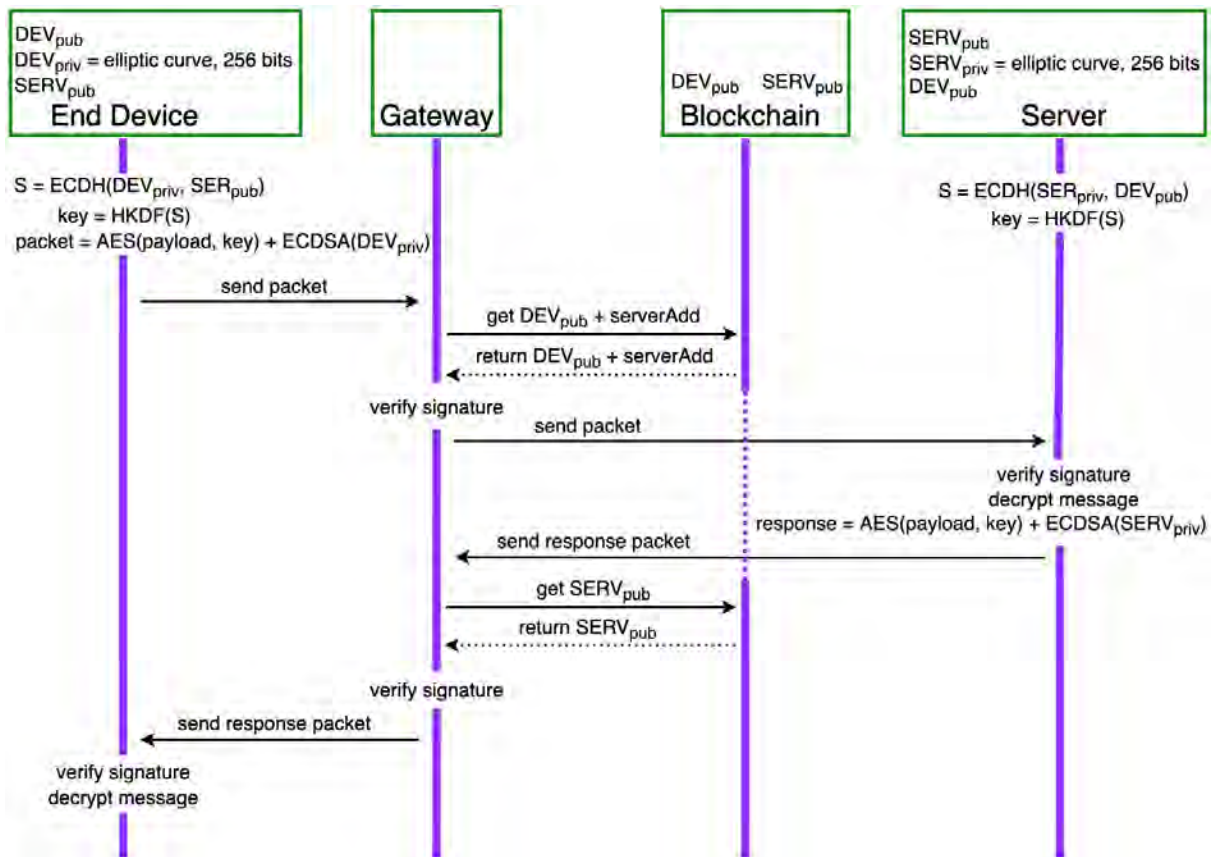


Fig. 4.1.: The LoRa-MAC protocol architecture

This subsection will analyze the architecture of the LoRa-MAC protocol shown in Fig. 4.1. The architecture of the protocol is divided into four components: the *End Device*, the *Gateway*, the *Blockchain* and the *Server*. The *End Device* and the *Server* need to generate a new set of public-private key pair each (DEV_{pub} and DEV_{priv} on the *End Device* and $SERV_{pub}$ and $SERV_{priv}$ on the *Server*). Each private key which is 256 bits long, is generated from an elliptic curve and must be kept secret on the device that generated it. The two public keys need to be provided on the *Blockchain* and exchanged between both entities. Furthermore, the address of the *Server* needs to be provided on the *Blockchain* as well. The *End Device* and the *Server* have to share a secret / symmetric key in order to encrypt the payload of the messages they will be sending to each other. This secret key S is generated on each side by applying ECDH on the DEV_{priv} and the $SERV_{pub}$ on the *End Device* side and by applying ECDH on the $SERV_{priv}$ and DEV_{pub} on the *Server* side. Then, each side needs to normalize the symmetric key S by applying HKDF to it. This last step produces the final secret / symmetric key used for encrypting and decrypting the payload of the LoRa-MAC messages.

A new communication between both ends always starts by the *End Device* wanting to send a packet to the *Server*. In fact, the *End Device* is asynchronous and transmits data only when there are some available. Thus, it is impossible for the *Server* to send a packet to the *End Device* without receiving a packet first. This mechanism permits to save battery life on the end devices and is imperative for the goal of LoRa to provide a wide-area network for low-power consumption devices.

The *End Device* generates a new packet by first encrypting the payload of the message using the AES algorithm and the secret / symmetric key. The packet is then signed by using the ECDSA algorithm and the private key of the component. The signature produced is appended to the packet. A more detailed explanation of the content and the creation of a packet will be explained in the following subsections.

The packet generated on the *End Device* is then sent using the LoRa technology to the *Gateway*. The latter gets the DEV_{pub} of this *End Device* and the address of the corresponding server on the *Blockchain*. The *Gateway* then verifies the validity of the signature of the packet by using the DEV_{pub} . If the signature of the packet is valid, the packet is forwarded to the *Server*.

On reception of the packet, the *Server* verifies its signature and if it is valid, it is decrypted in order to get its content. The *Server* can then generate a response packet by encrypting the content of the response message thanks to the use of AES and the secret / symmetric key. The packet is then signed by using ECDSA and the $SERV_{priv}$ key. Afterwards, the packet is sent to the *Gateway*.

The *Gateway* verifies again the validity of the received packet. This time, the $SERV_{pub}$ key is requested to the *Blockchain*. If the signature of the packet is valid, the packet is sent to the *End Device* by using the LoRa technology.

The *End Device* verifies the signature of the packet by using the public key of the *Server*. Afterwards, if the signature is valid, the *End Device* can decrypt the packet by using the secret / symmetric key in order to obtain the payload.

4.2.2. Message structure

A message is comprised of three elements: the Header, the Ciphertext and the Signature.

Message	Header	Ciphertext	Signature
Header	MType	Counter (Nonce)	Device Address
Ciphertext	Payload		

Tab. 4.2.: Message structure

The Header is itself composed of three elements: the message type (MType), the Counter and the device address. The Counter is used as a nonce to ensure that each message sent by a device is unique. The device address represents either the address of an end device or the address of a server. Finally, the message type is a binary number with the following meaning.

MType	Meaning
0011	DataConfirmedUp
0100	DataUnconfirmedUp
0101	DataConfirmedDown
0110	DataUnconfirmedUp
0111	ACKUp
1000	ACKDown

Tab. 4.3.: Message type (MType)

- **DataConfirmedUp** is used for a message that is sent from an end device to a server and for which the end device is waiting for a response from the server.
- **DataUnconfirmedUp** is used for a message that is sent from an end device to a server and for which the end device is not waiting for a response from the server.
- **DataConfirmedDown** is used for a message that is sent from a server to an end device and for which the server is waiting for a response from the end device.
- **DataUnconfirmedUp** is used for a message that is sent from a server to an end device and for which the server is not waiting for a response from the end device.
- **ACKUp** is used for an acknowledgment message sent from an end device to a server.
- **ACKDown** is used for an acknowledgment message sent from a server to an end device.

The Ciphertext is the Payload encrypted by using the AES algorithm with the secret / symmetric key.

4.2.3. Packet structure

The packets sent between an end devices and a server are in reality COSE_Encrypt0 messages to which a COSE_CounterSignature structure is appended in the unprotected part of the header.

The protected field of the COSE_Encrypt0 header contains the name of the algorithm used for the encryption, the Initialization Vector (IV) value and a reserved section which contains the header of the message structure. The unprotect field of the COSE_Encrypt0 header contains the Key Identifier (KID) and the COSE_CounterSignature structure. The KID value is used to give a hint about which key to use. Finally, the COSE_Encrypt0 message contains the Ciphertext.

The COSE_CounterSignature structure is divided between the protected part which contains the name of the algorithm used to generate the signature, the unprotected part which contains the KID and lastly, the signature itself.

The algorithm used for the encryption of the payload is A128GCM which stands for AES-GCM mode with a 128 bits key. The algorithm used to generate the signature is ES256 which means ECDSA with SHA256.

```

1 COSE_Encrypt0(
2   [
3     protected {
4       alg : A128GCM,
5       iv : ... ,
6       reserved : header
7     },
8     unprotected {
9       kid : ... ,
10      COSE_CounterSignature : [
11        protected {
12          alg : Es256
13        },
14        unprotected {
15          kid : ... ,
16        },
17        signature
18      ]
19    },
20    ciphertext
21  ]
22 )

```

List. 4.1: Final COSE_Encrypt0 message

4.2.4. Blockchain

For each new *End Device* created by a user, the *Server* has to register it on the *Blockchain*. This is done by calling a smart contract on the *Blockchain* from the *Server* in order to store the address and the public key of the *End Device* and the address of the *Server*. In addition, the *Server* has to register only once its own public key on the smart contract as well.

As explained in section 2.5.1, the blockchain technology selected—for this project—is

Ethereum. Thus, a smart contract has been written using the default Ethereum programming language, Solidity. The smart contract has been deployed on the Rinkeby Test Network and is accessible at the address: `0x4a9ff7c806231ff7d4763c1e83E8B131467adE61`¹. The smart contract will be discussed in more details in chapter 5.5.

4.3. Remuneration using Micropayments

One of the goal of the master thesis was to develop a decentralized use-case in order to show some of the potential of the new created infrastructure, LoRa-MAC. The decentralized solution that has been explored as an extension is about remuneration in crowd-sourced networks thanks to the use of micropayments. The final goal of this Micropayment extension is to give the ability to everyone to deploy their own gateway and to get rewarded for transferring messages in the LoRa-MAC protocol.

The extension is focused about micropayments because, if the developed solution wants to stay competitive against other deployment options presented in section 2.3 while keeping its decentralization advantage, the cost of transferring a packet in the system should be really small, in the order of a few cents.

The major difficulty that surround micropayments are the fees. Indeed, it is challenging to transfer a few cents between two wallets without having to pay more in fees than the actual amount transferred. Furthermore, this is especially the case on the Ethereum blockchain due to its raising popularity which has led to an explosion in the transaction fees called gas. "Gas is a unit of account within the EVM used in the calculation of a transaction fee, which is the amount of ETH a transaction's sender must pay to the miner who includes the transaction in the blockchain" [24].

As the number of people using Ethereum has grown substantially, the blockchain has reached certain capacity limitations. This is why, a need for scaling solutions has grown. The main goal of this scaling solutions is to increase the transaction speed (faster finality), and the transaction throughput (high number of transactions per second), without sacrificing decentralization or security. Conceptually, there are two categories of scaling solutions: on-chain and off-chain scaling.

On-chain scaling requires changes to the Ethereum protocol (layer 1 Mainnet). For this solution of scaling, the main focus is currently on sharding which is the process of splitting a database horizontally to spread the load [41]. Unfortunately, on-chain scaling is an active field of research and thus, other solutions need to be explored.

Off-chain scaling solutions are implemented separately from the layer 1 Mainnet. They require no changes to the existing Ethereum protocol. This is why, they are called layer 2 solutions. The main principle of most layer 2 solutions is to submit transaction to some layer 2 nodes instead of submitting them directly on the layer 1. The layer 2 instance then batches the transactions into groups before anchoring them on the layer 1, after which they are secured by the layer 1 and cannot be altered [41]. Tab. 4.4 gives an overall comparison of various Ethereum layer 2 scaling solutions. Since the main challenge of the Micropayment extension is about transaction fees as explained previously, the line of Tab. 4.4 comparing the costs of transaction is the most relevant. There are only two solutions that receive the mention "very low" for the cost of transaction: state channels and plasma. Thus, this two solutions have been explored in more details and have then

¹<https://rinkeby.etherscan.io/address/0x4a9ff7c806231ff7d4763c1e83E8B131467adE61>

been implemented in order to be able to compare them in full knowledge of the facts. The other solutions of layer 2 scaling will not be discussed in more details in the thesis but a starting point to have some more detailed informations about them can be found on the Ethereum.org web site [41].

	State channels	Sidechains ⁰	Plasma	Optimistic rollups	Validium	zkRollup
Security						
Liveness assumption (e.g. watch-towers)	Yes	Bonded	Yes	Bonded	No	No
The mass exit assumption	No	No	Yes	No	No	No
Quorum of validators can freeze funds	No	Yes	No	No	Yes	No
Quorum of validators can confiscate funds	No	Yes	No	No	Yes ¹	No
Vulnerability to hot-wallet key exploits	High	High	Moderate	Moderate	High	Immune
Vulnerability to crypto-economic attacks	Moderate	High	Moderate	Moderate	Moderate	Immune
Cryptographic primitives	Standard	Standard	Standard	Standard	New	New
Performance / economics						
Max throughput on ETH 1.0	1..∞ TPS ²	10k+ TPS	1k..9k TPS ²	2k TPS ³	20k+ TPS	2k TPS
Max throughput on ETH 2.0	1..∞ TPS ²	10k+ TPS	1k..9k TPS ²	20k+ TPS	20k+ TPS	20k+ TPS
Capital-efficient	No	Yes	Yes	Yes	Yes	Yes
Separate onchain tx to open new account	Yes	No	No	No	No	No ⁵
Cost of tx	Very low	Low	Very low	Low	Low	Low
Usability						
Withdrawal time	1 confirm.	1 confirm.	1 week ⁴ (?)	1 week ⁴ (?)	1..10 min ⁷	1..10 min ⁷
Time to subjective finality	Instant	N/A (trusted)	1 confirm.	1 confirm.	1..10 min	1..10 min
Client-side verification of subjective finality	Yes	N/A (trusted)	No	No	Yes	Yes
Instant tx confirmations	Full	Bonded	Bonded	Bonded	Bonded	Bonded
Other aspects						
Smart contracts	Limited	Flexible	Limited	Flexible	Flexible	Flexible
EVM-bytecode portable	No	Yes	No	Yes	Yes	Yes
Native privacy options	Limited	No	No	No	Full	Full

⁰ Some researchers do not consider them to be part of L2 space at all, see <https://twitter.com/gakonst/status/1146793685545304064>

¹ Depends on the implementation of the upgrade mechanism, but usually applies.


² Complex limitations apply.

³ To keep compatibility with EVM throughput must be capped at 300 TPS

⁴ This parameter is configurable, but most researchers consider 1 or 2 weeks to be secure.

⁵ Depends on the implementation. Not needed in zkSync but required in Loopring.

⁷ Can be accelerated with liquidity providers but will make the solution capital-inefficient.



Tab. 4.4.: Ethereum layer 2 scaling solutions [42]

4.3.1. Ethereum layer 2 scaling solutions

State channels - micropayment channels

To take part in a channel, participants have to lock a portion of Ethereum's state, like an ETH deposit for example, into a multisig smart contract which is a type of contract that requires the signature (and therefore the agreement) of multiple private keys to be executed. Locking a portion of Ethereum's state is the first transaction and opens up the channel. The participants can then transact quickly and freely off-chain. When the interactions are terminated, a final on-chain transaction is submitted, unlocking the

Ethereum's state. There are two types of channels currently: state channels and payment channels [41].

State channels use multisig contracts that permit to participants to transact quickly and freely x number of times off-chain while only submitting two on-chain transactions to the Ethereum layer 1 network. This allows for extremely high transaction throughput by minimizing network congestion, fees, and delays [41]. Payment channels are a simplified version of state channels that only deals with payments (e.g. ETH transfers). They permit off-chain transfers between two participants, as long as the net sum of their transfers does not exceed the deposited token amount [41].

Micropayment channels are an example implementation of payment channels. They use cryptographic signatures that can be sent off-chain (e.g. via email) to make repeated transfers of ETH between the same parties secure, instantaneous, and without transaction fees. The process can be seen as similar to writing checks [43].

The case of a simple unidirectional micropayment channel between two parties like the one this project will be using is now described. More informations about micropayment channels can be found from the Solidity documentation [43]. Imagine Alice wants to send a quantity of ETH to Bob, i.e. Alice is the sender and Bob is the recipient. Alice and Bob use signatures to authorize transactions, which is possible thanks to smart contracts on Ethereum. Alice has to build a smart contract that lets her transmit ETH to Bob. The smart contract works as follow:

1. Alice deploys the smart contract, attaching enough ETH to cover the payments that will be made. This "opens" the payment channel. Furthermore, Alice has to specify the intended recipient and a maximum duration for the channel to exist.
2. Alice signs messages with her private key in order to specify how much ETH is owed to the recipient. For this task, Alice does not need to interact with the Ethereum network. The process can be done completely offline and is repeated for each payment. Each message includes the following informations:
 - The address of the smart contract which is used to prevent cross-contract replay attacks.
 - The total amount of ETH that is owed to the recipient so far.

The message is then hashed and signed.

3. Alice sends the cryptographically signed message to Bob. The message does not need to be kept secret, and the mechanism for sending it does not matter.
4. Bob claims its payments by presenting the last signed message of Alice to the smart contract which verifies the authenticity of the message (the message has to contain a valid signature from the sender which matches the given parameters) and then releases the funds to their owners. This "closes" the payment channel and destroys the smart contract. A payment channel is closed only once, at the end of a series of transfers. Because of this, only one of the messages sent is redeemed. This is why each message specifies a cumulative total amount of ETH owed, rather than the amount of the individual micropayment. A recipient naturally chooses to redeem the most recent message because it has the highest total. It is critical for Bob to perform his own verification for each message. Otherwise, there is no guarantee that he will be able to get paid in the end.

Thus, a recipient should verify each message using the following process:

- Verify that the contact address in the message matches the payment channel.
- Verify that the new total is the expected amount.
- Verify that the new total does not exceed the amount of ETH stored in the smart contract.
- Verify that the signature is valid and comes from the payment channel sender.

Closing the contract before the expiration is only possible for the recipient because if the sender was allowed to do it, he could provide a message with a lower amount and cheat the recipient out of what he is owed.

Only steps 1 and 4 require Ethereum transactions. This means that only two transactions are required to support any number of transfers.

Bob is guaranteed to receive his funds because the smart contract escrows the ETH and honors a valid signed message. The smart contract also has a determined expiration. Therefore, Alice is guaranteed to eventually recover her funds even if the recipient refuses to close the channel. It is up to the participants in a payment channel to decide how long to keep it open. It is important for Bob to close the channel before the expiration is reached otherwise he can no longer receive any ETH.

Plasma - OMG Network

Plasma refers to a framework co-conceived by the creator of Ethereum, Vitalik Buterin, that allows the creation of separate blockchains which are anchored to the main Ethereum chain. These separated blockchains are usually called child chains because they are essentially smaller copies of the Ethereum Mainnet [41]. The offload of bandwidth from the parent chains is achieved by performing transactions on the child chain instead of on the Ethereum main chain. This relieves pressure on the Ethereum blockchain by condensing all the transactions that happen across multiple child chain blocks into a single, validated Merkle proof that is submitted to the root chain. Therefore, these child chains occasionally write a fingerprint of their state to the root chain. Users can enter the child chain through a smart contract. Thus, interactions with the Ethereum root chain are only required for depositing funds to the child chain or exiting funds from the child chain back to the root chain [8]. The plasma chains can operate at a different speed and under a different consensus mechanism than the main chain. Furthermore, each child chain has its own mechanism for block validation. If a user wishes to exit a child chain, a certain “challenge period” is required, in which any potential attempts of fraud can be prevented [9]. There exist different implementations of plasma and one of them is the OMG Network. The thesis has selected this implementation of plasma as one of the solutions to perform micropayments because at the time of the implementation, OMG Network was the most popular plasma solution.

Initially called OmiseGO (OMG) upon launch in 2017, "the OMG Network creates a value transfer layer on top of Ethereum that bundles together Ethereum transactions and validates them through a speed-optimized child chain before sending them back to the Ethereum blockchain for confirmation. The OMG Network is able to process thousands of transactions per second, and can reduce the costs of operating on Ethereum by one third. The OMG Network's native OMG token can be used to pay for transaction fees" [10]. The protocol of OMG Network uses an updated version of plasma, called More Viable

Plasma (MoreVP), to significantly increase transaction throughput. More specifically, the OMG Network works by bundling transactions together, compressing them into one transaction, and verifying them on the OMG Network's child chain. The child chain then returns confirmed transactions to the Ethereum main chain in order to guarantee the confirmations on the blockchain. This bundling technique enables the OMG Network to process thousands of transactions per second. Transaction costs are largely inferior to what they would cost on Ethereum because gas fees are paid on the bundled, compressed transaction instead of each individual transaction [10].

The OMG Network architecture consists of three components interacting together [44]:

- Deployed on the Ethereum network, **the PlasmaFramework smart contract** can be seen as a top-level contract containing other smart contract level functionalities like deposits, exits, and the receipts of block transactions from the child chain.
- **The OMG Network child chain** is controlled by a single block producing node called an operator, which maintains the network state. The operator receives transactions from the users and bundles them into blocks that are submitted to the PlasmaFramework smart contract on the Ethereum main chain.
- **Watchers** are applications continuously monitoring the child chain, validating its behavior and reporting any inconsistency or malicious behavior to subscribed users. Anyone can deploy and run its own watcher because the biggest the number of running watchers, the safer the network gets.

While Ethereum is totally decentralized and runs on many nodes, OMG Network has only one which is run by the OMG Foundation itself. However, decentralization of the security is achieved through the network of watchers which monitors the operator and the network for suspicious activities. The watchers check all the blocks that are submitted to the root chain by pulling the Merkle proof tree from the operator, and then validating that the transactions that are supposed to have happened, have indeed happened [10]. In this system, the network is semi-decentralized through the watchers that can identify bad behaviors of the operator, including if it becomes unavailable or if transactions are manipulated. If this become the case, the watchers have the ability to launch a mass exit which flags all the funds on the plasma chain to be withdrawn back to Ethereum. Watchers are able to do this because they have visibility to data on both the layer 1 and layer 2 blockchains. The consensus mechanism behind this system is called Proof of Guarantee (PoG) [8]. Additionally, the security is guaranteed because the child chain is non-custodial, which means that funds of the users never leave the Ethereum network at any time. As a result, users can be ensured that they will always be able to recover their funds, even if the OMG Network child chain goes offline. This feature is also known as trustlessness [10].

5

Implementation

5.1. Introduction	32
5.2. Architecture	33
5.3. End Device	35
5.3.1. Raspberry Pi	35
5.3.2. LoPy	37
5.4. Gateway	38
5.4.1. UDP Packet Forwarder	39
5.4.2. Forwarding Network Server (FNS)	39
5.5. Blockchain	43
5.6. Server	45
5.6.1. Home Network Server (HNS)	45
5.6.2. Application Server (AS)	47
5.7. Extension: Micropayment	48
5.7.1. Forwarding Network Server (FNS)	50
5.7.2. Blockchain	52
5.7.3. Payment service	53
5.7.4. Home Network Server (HNS)	54
5.8. Outlook	55
5.9. Pycose library	56

5.1. Introduction

This chapter contains in depth details about the final implementation of each component of the architecture described in section 4.2.1. The source code of the implementation can be found on the GitHub repository [45]. During the progress of the project, several changes were made concerning the technologies used. These changes were mainly at the level of the *End Device* and the means of communication between the *Gateway*

and the *Server*. Subsequently, the implementation of the Micropayment extension is also presented in this chapter but is explained in its own section. The source code of the extension can be found at the same address as the rest of the implementation. Since the project is vast and multiple sequence diagrams are presented throughout the thesis, a final sequence diagram including all the components and the interactions between them is included. Finally, this chapter also explains in details how the `pycose` Python library has been modified to support `COSE_CounterSignature`. The source code of the `pycose` library can be found on GitHub [46].

5.2. Architecture

The subsequent sequence diagram shows in more detailed the architecture of the LoRa-MAC protocol presented in section 4.2.1.

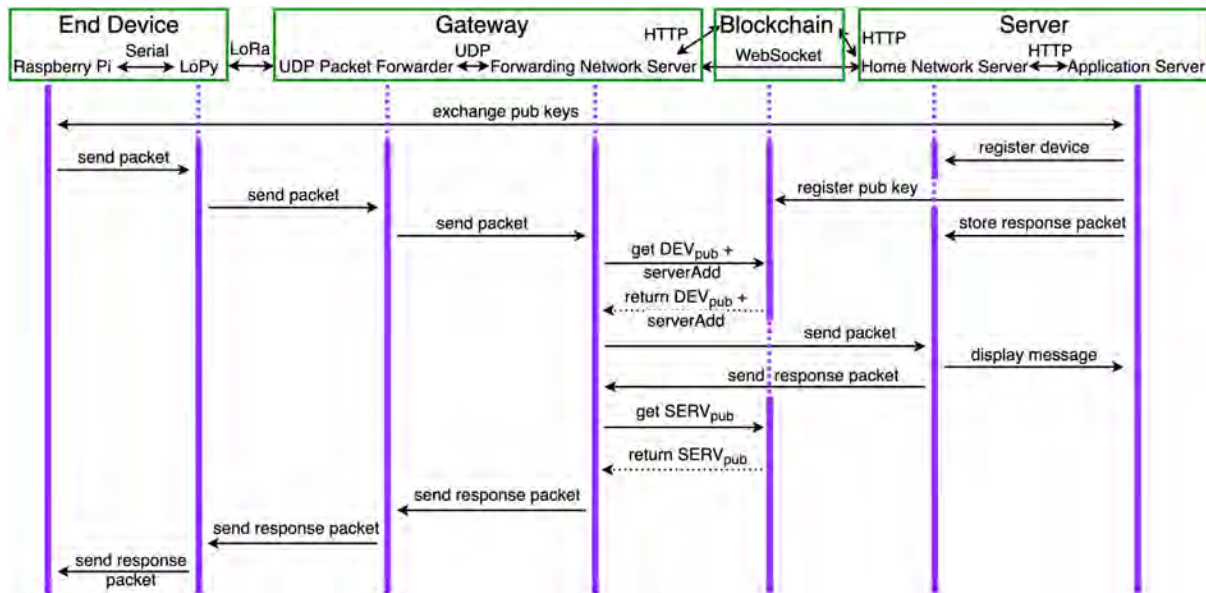


Fig. 5.1.: Sequence diagram of the LoRa-MAC protocol

The *End Device* is divided into two components: the *Raspberry Pi* and the *LoPy*. The communication between these two components is made through serial. The reason why two devices are used to compose the *End Device* can be found in section 6.5.

The *Gateway* is also divided into two components: the *UDP Packet Forwarder* and the *Forwarding Network Server (FNS)*. These two components communicate through UDP packets. The *FNS* communicates with the Blockchain by making HTTP requests and with the *Server* through WebSockets. WebSockets allow sending message-based data, similar to UDP, but with the reliability of TCP. They use HTTP as the initial transport mechanism, but they keep the TCP connection alive after the HTTP response is received. This functionality makes them a good choice for sending messages between a client and a server [11].

The *Server* is divided into two components as well: the *Home Network Server (HNS)* and the *Application Server (AS)*. The *AS* calls the *HNS* by making HTTP requests. The *HNS*

also uses HTTP to communicate with the *Blockchain* and WebSockets to communicate with the *FNS*.

The first step in the sequence diagram is the exchange of the public keys between the *Raspberry Pi* and the *AS*. Furthermore, the address of the *End Device* needs to be provided to the *Server* as well. This exchange permits to create a new device in the *AS*. The newly created device in the *AS* is then registered in the *HNS* which stores it in its database. The *AS* then calls the *Blockchain* in order to store the public key and the address of the device along with its own address in the deployed smart contract. If this is not already done, the *AS* also stores its own public key in the smart contract. Finally, on the *AS* side, a programmed response that will be sent to the *End Device* when an upstream packet is received from it, is stored on the *HNS*.

On the *End Device* side, the *Raspberry Pi* generates a message and a packet to be sent. The content of the message is encrypted by using the secret / symmetric key that the *End Device* and the *Server* share. The packet is also signed with the private key of the *End Device*. Then, the packet is sent through serial to the *LoPy* which forwards it through LoRa to the *Gateway*. On the *Gateway*, the packet is received by the *UDP Packet Forwarder* that transmits the packet to the *FNS* by using a UDP packet.

On reception of the packet, the *FNS* analyzes it and extracts the address of the device. Thanks to this address, it can make a request to the smart contract on the *Blockchain* to obtain the public key of the *End Device* that has sent the packet and the address of the *Server* to which the packet has to be sent. The *FNS* can now verify the signature of the packet by using the public key of the *End Device* so that only valid messages are transmitted to the *Server*. If the packet is valid, the *FNS* can open a WebSocket with the *HNS* and send the packet to it.

The *HNS* analyzes the packet, extracts the device address and gets the corresponding public key of the *End Device* in its database. The *HNS* then verifies the signature of the packet and if the signature is valid, it generates the corresponding secret / symmetric key by using its private key and the public key of the *End Device*. With this symmetric key, the *HNS* can decrypt the content of the payload and store this payload along with other data (like the header of the message and the time of reception) on its database such that the user can consult them on the *AS* later. Finally, the *HNS* can either send the response packet stored previously by the *AS* or can generate a response packet if none have been provided. The response packet is sent through the same WebSocket that has been opened between the *FNS* and the *HNS*.

Again, the *FNS* has to verify the validity of the packet. For this purpose, it has to get the public key of the *Server* on the smart contract in the *Blockchain*. Once the public key is obtained, it can verify the validity of the packet and if everything is correct, the packet can be sent to the *UDP Packet Forwarder* such that the response is sent to the *End Device*.

The response packet is received by the *LoPy* and immediately forwarded to the *Raspberry Pi* through serial. The *Raspberry Pi* verifies the signature of the packet too. Afterwards, the response packet can be decrypted using the secret / symmetric key so that its content can then be processed by the *Raspberry Pi*.

Each component of the architecture and their sub-components will be described in more details in the next sections of this chapter.

5.3. End Device

As explained previously, the *End Device* is in fact itself divided into two components: the *Raspberry Pi* and the *LoPy*. Both components should be connected through serial as shown in the following picture.

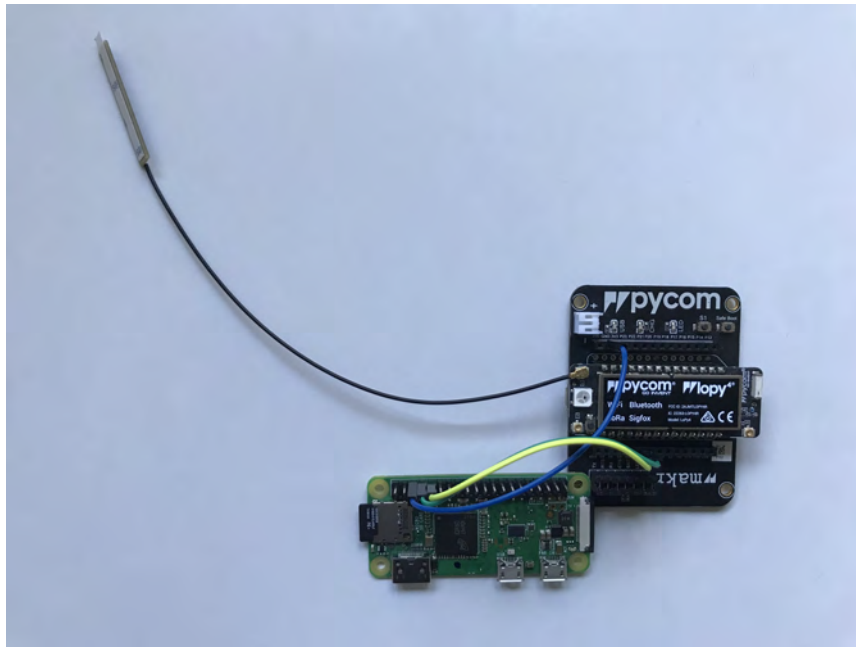


Fig. 5.2.: Picture of the *Raspberry Pi* and the *LoPy4*

The *End Device* can be used only in a manual / interactive mode where the user has to manually run a program on both the *Raspberry Pi* and the *LoPy* simultaneously. The program that runs on the *Raspberry Pi* asks to the user to provide the content of the payload he wants to send to the *Server*. It is recommended to first start the program on the *LoPy* and then the one on the *Raspberry Pi*. The program to be run on the *Raspberry Pi* is written in Python while the one for the *LoPy* is written in MicroPython. The "brain" of the *End Device* is the *Raspberry Pi* while the *LoPy* is mainly used just to transmit the LoRa RF packets. This is why, most of the work and the logic is done on the *Raspberry Pi*.

5.3.1. Raspberry Pi

The program to be run on the *Raspberry Pi* can be found on the `LoRa_device` directory of the GitHub repository¹. During the development of the thesis, the program was run on a Raspberry Pi Zero W² running Python 3.7.3. The required packages can be found in the `requirements.txt` file. The modified version of the `pycose` library [46] is also required for the program to run properly. Some more detailed informations about how to install all the required packages and the modified version of `pycose` can be found in the `README` file of the `LoRa_device` directory. The `README` file also explains how to start

¹https://github.com/inefix/Decentralized-LoRa/tree/master/LoRa_device

²<https://www.raspberrypi.org/products/raspberry-pi-zero-w/>

the program. The user needs to provision, among other variables, the public key of the *Server* in a `.env` file. This public key needs to be divided between the X and the Y values.

The main file of the *Raspberry Pi* program is the `device.py` file. The program can be run for two different use-cases. If the user wants to generate a new *End Device* meaning that he needs a new address for the device and a new pair of public-private keys, he can start the program followed by a `"-n"` argument. Accordingly, the program generates all the required data and stores them in the `keys.txt` file. Additionally, the `keys.txt` file also stores the private value of the private key and the public key of the *Server*. The second use-case can be used by the user when he has already registered his device and he only wants to send a new packet. Hence, the second use-case does not need to generate any of the previous generated data since it uses the one stored in the `keys.txt` file. For this second use-case, the `device.py` program has to be run without any argument.

Once running, the program starts by either generating a new address for the device and a new set of public-private keys or by reading the ones stored in the file `keys.txt`. The address of the device is a randomized 64 bits identifier expressed in hexadecimals. The length of the address is long enough such that collisions are very unlikely. In fact, the number of different addresses is 2^{64} . The key pair is generated by using the `cryptography` Python library. The elliptic curve used to generate the private key is the `secp256r1` also called NIST P-256.

When starting the program, the user is asked to write on his terminal the payload of the message he wants to send. Afterwards, the program generates the symmetric key that will be used to encrypt the payload. This symmetric key is also generated by using the `cryptography` Python library. For this purpose, the program first uses the ECDH algorithm with the private key of the device and the public key of the *Server* as parameters. Then, the ECDH key is normalized thanks to the HKDF algorithm by using the following parameters: SHA256 as algorithm, a length equal to 16 (since the resulting key will be used for encryption with AES128), a salt equal to none and "handshake data" as the info. As the parameter length equals to 16 can suggest, the symmetric key generated has a length of 16 characters.

Subsequently, the payload is encrypted. This procedure is achieved by creating an `Enc0Message` thanks to the standard `pycose` library. The protected part of the header contains first the name of the algorithm used to encrypt, which is A128GCM and correspond to AES in GCM mode with a 128 bits key. The second element of the protected header is the IV which is "000102030405060708090a0b0c" and finally the last element is the header of the message which is comprised of the MType, the counter and the address of the device. For each new message sent, the counter is read, incremented by one and stored in a file called `counter.txt`. The payload is encrypted using the AES128 algorithm with the symmetric key and placed in the adequate part of the `Enc0Message`.

The newly created `Enc0Message` has to be signed with the private key. This is done by appending a `COSE_CounterSignature` structure in the unprotected part of the header of the `Enc0Message` thanks to the modified `pycose` library. Indeed, this functionality is not developed in the standard `pycose` library. The `COSE_CounterSignature` structure contains the name of the algorithm which is ES256 (ECDSA with SHA256) in its protected header. The signing of the encrypted payload is performed by using the ECDSA algorithm with the private key of the device. The resulting signature is placed in the adequate part of the `COSE_CounterSignature` structure.

The packet has now been created and is ready to be sent. The program opens a se-

rial communication with the *LoPy* by using the asynchronous `serial_asyncio` Python library. As explained in section 3.6, it is important to use asynchronous programming when sending packets to the *LoPy* so that the program can release the processor while waiting on the responses. Before sending, the packet which is in byte string format, it is converted in its hexadecimal representation in order to be able to add an end of line character ("`\n`") to it. Indeed, the end of line character is used to mark the end of a communication between the *Raspberry Pi* and the *LoPy*. Once the packet is sent, the program has to wait until a response packet is sent back from the *LoPy*.

Once a response packet is received from the *LoPy*, it is converted in binary data (byte string) from its hexadecimal representation in order to obtain an `Enc0Message`. The signature of the response packet present in the `COSE_CounterSignature` structure of the `Enc0Message` is then verified with the public key of the *Server* and if it is valid, the ciphertext of the `Enc0Message` is decrypted with the symmetric key previously generated. During the entire run of the *Raspberry Pi* program, it is important to catch the eventual errors that can easily happen due to the radio wave nature of the LoRa packets.

5.3.2. LoPy

The program to be run on the *LoPy* can be found on the `LoPy` directory of the GitHub repository³. During the development of the master thesis, the program was executed on a *LoPy*⁴ running Pycom MicroPython 1.20.2.r4. It is important to update the *LoPy* before running the program. Otherwise, there could be some issues to receive the downstream packet, as for example, no reception at all. The program is developed to be run on a *LoPy* equipped with a LoRa module and an antenna. The user must be careful to never start the program on a *LoPy* if the antenna is not connected. Some more detailed informations about how to check the firmware version and how to start the program can be found in the `README` file of the `LoPy` directory.

The main file of the *LoPy* program is the `device.py` file. When starting it, it first makes sure that `Pybytes` is disabled by modifying the file `/flash/pybytes_config.json` present on the *LoPy* in order to ensure that downstream messages are received by the *LoPy*. This procedure could imply a reboot of the *LoPy*.

Afterwards, the program initializes the LoRa MicroPython library developed by Pycom in LORA mode (raw LoRa). Some really precise parameters need to be provided so that the LoRa packets sent are received by the *Gateway*. This include to set the region to EU868, the spreading factor (sf) to 12 (the spreading factor should be a value between 7 and 12, the higher it is, the less there is a risk of corrupted bits on the reception side), the coding rate to 4/7 (4/5 and 4/6 have a higher chance of corrupted bits on the reception side and 4/8 has a higher chance of the message not being received on the *Gateway* side), the power mode to always on, the frequency to 867.5 MHz and the bandwidth to 125 KHz.

The program then creates a raw LoRa socket for sending and receiving LoRa messages. Subsequently, the program opens a serial communication with the *Raspberry Pi* by using the `uart` MicroPython library developed by Pycom.

The initialization part is now terminated and the *LoPy* program starts listening to packets sent by the *Raspberry Pi* through the serial communication. When a packet is received,

³<https://github.com/inefix/Decentralized-LoRa/tree/master/LoPy>

⁴<https://pycom.io/product/lopy4/>

it is converted in binary data (byte string) from its hexadecimal representation since it had to be converted for the serial transmission as explained in the previous subsection. The packet is then sent to the *Gateway* through LoRa.

The program waits for a response packet from the *Gateway*. This is done by checking during 2 minutes every second if a packet has been received. The one second break between each check is very important since it permits to the LoPy to listen to eventual incoming LoRa messages. Indeed, if the program were to continuously check if a packet has been received, it will not let the possibility to the LoPy to listen for incoming LoRa messages and thus none would be received. The 2 minutes time interval acts as a protection to avoid to the program to get stuck waiting indefinitely.

If a LoRa downstream packet is received, it is converted from its byte string format to its hexadecimal value and forwarded back to the *Raspberry Pi*. If no LoRa packet is received, an empty response is forwarded to the *Raspberry Pi*. Of course, an end of line character ("`\n`") is added to the hexadecimal packet to mark the end of the communication between the *LoPy* and the *Raspberry Pi*.

5.4. Gateway

The *Gateway* is divided into two components: the *UDP Packet Forwarder* and the *Forwarding Network Server (FNS)*. Both components are in fact programs that are run simultaneously on the same device, a Raspberry Pi 3 Model B with a RAK831 LPWAN Gateway Concentrator Module mounted on top through SPI and an antenna connected to it as shown in the following pictures.

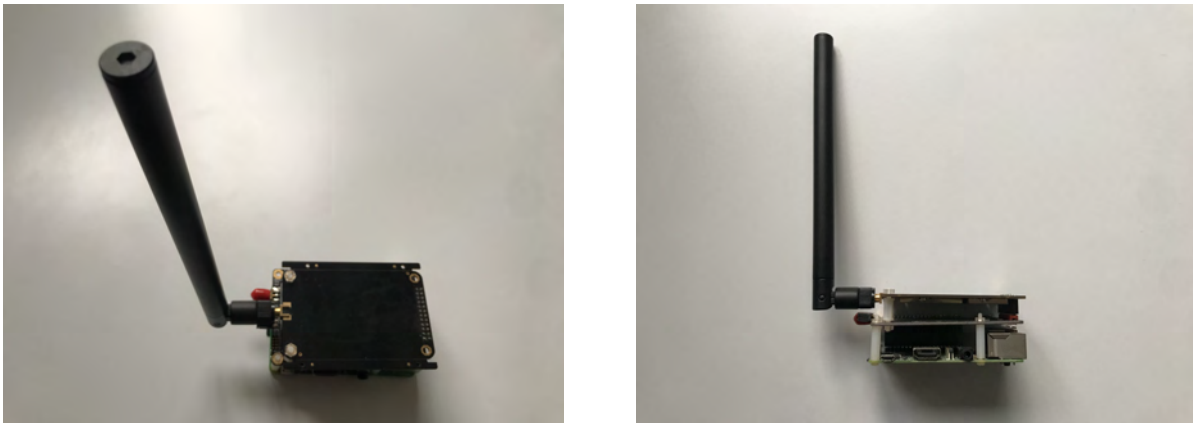


Fig. 5.3.: Pictures of the *Gateway*

The *UDP Packet Forwarder* is a standard program that permits to interact with the concentrator to either receive or send LoRa messages. The program is mainly used in the LoRaWAN protocol in conjunction with TTN. Since one of the goal—of this project—is to use standard LoRa hardware and softwares as they are without any modification, there was a need to develop a program on top of the *UDP Packet Forwarder* to bring the new functionalities of the LoRa-MAC protocol to the gateways. This functionalities include among others, decentralization of the gateways using the blockchain and the ability to be remunerated as a gateway. This is why, the *FNS* as been developed to provide this additional functionalities.

5.4.1. UDP Packet Forwarder

The *UDP Packet Forwarder* has already been described in many details in section 3.4. It is worth mentioning that the `global_conf.json` configuration file used on the Raspberry Pi 3 can be found on GitHub⁵. Since the project aims to develop an alternative to LoRaWAN, in order to ensure that all messages are effectively received on the *Gateway* side, the options to forward LoRa packets containing Cyclic Redundancy Check (CRC) errors or not having a CRC at all have been set to true. The CRC permits to detect if a LoRa message is corrupted. The *UDP Packet Forwarder* is configured to forward the LoRa packets that are received to the *FNS* on port 1700 of the address 0.0.0.0 using the UDP protocol. The *UDP Packet Forwarder* is also configured to listen to incoming UDP packets on port 1700 of the address 0.0.0.0.

5.4.2. Forwarding Network Server (FNS)

The program to be run as the *FNS* has been written in Python and can be found on the `ForwardingNetworkServer` directory of the GitHub repository⁶. The program has been tested on Python 3.7.3. The required packages can be found in the `requirements.txt` file. The modified version of the `pycose` library [46] is also required for the program to run properly. Some more detailed informations about how to install all the required packages and the modified version of `pycose` can be found in the `README` file of the `ForwardingNetworkServer` directory. The `README` file also explains how to start the program.

To configure the program, the user needs to provide some variables in a `.env` file, including MongoDB⁷ credentials. In fact, MongoDB is the database solution selected for all the database needs for the entire thesis because it is an easy solution to use and is quite popular. In addition, in order to connect to the Ethereum blockchain, a node connected to the blockchain is needed. Therefore, the user needs to provide the address of such a node. There exists many alternatives which are mainly subdivided into two categories: providing a node hosted by the user himself (like for example Geth⁸) or using a node provider. The node provider category is itself divided in two categories: companies that abstract away the node management and just provide APIs (like Infura⁹ which is the solution tested during the implementation) and companies that deploy dedicated or shared nodes (like BlockDaemon¹⁰) [47].

The program starts with an initialization phase where different clients are initialized. This include the client to connect to the MongoDB database (using the `motor` Python library) and the client to connect to the Ethereum blockchain (using the `web3s` Python library which is an asynchronous version of the famous `web3` library). After the initialization of the `web3s` client is done, it is important to initialize the `web3` object that will permit to interact with the smart contract deployed on the *Blockchain*. This is done by passing the address and the Application Binary Interface (ABI) of the smart contract as parameter

⁵https://github.com/inefix/Decentralized-LoRa/blob/master/ForwardingNetworkServer/global_conf.json

⁶<https://github.com/inefix/Decentralized-LoRa/tree/master/ForwardingNetworkServer>

⁷<https://www.mongodb.com/>

⁸<https://geth.ethereum.org/>

⁹<https://infura.io/>

¹⁰<https://blockdaemon.com/>

to the object. The smart contract ABI is an interface that defines a standard scheme in JSON of how to call functions in a smart contract and get data back. The smart contract ABI is designed for external use such as enabling application-to-contract interactions by defining the function names and the argument data types of the smart contract [12]. The `motor` and the `web3` clients are asynchronous and permit to the program to run in an asynchronous way such that the *Gateway* can listen to incoming packets at all time. Furthermore, an asynchronous UDP server is started on port 1700 of the address 0.0.0.0 in order to listen for incoming packets sent by the *UDP Packet Forwarder*. Packets which are received by the UDP server, are of the following forms.

```

1 \x02\xa0\x9d\x00\xb8'\xeb\xff\xfe\xa8o\b9{"rxpk": [
2   {
3     "tmst" : 1525043316,
4     "chan" : 5,
5     "rfch" : 0,
6     "freq" : 867.500000,
7     "stat" : 1,
8     "modu" : "LORA",
9     "datr" : "SF12BW125",
10    "codr" : "4/7",
11    "lsnr" : 5.8,
12    "rssi" : -15,
13    "size" : 5,
14    "data" : "aGVsbG8="
15  }
16 ]}]

```

List. 5.1: PUSH_DATA packet

```

1 \x02\x90\xff\x02\xb8'\xeb\xff\xfe\xa8o\b9

```

List. 5.2: PULL_DATA packet

It is first necessary to analyze this packets in order to know what they actually mean. The packets are in a string byte format and the first 12 bytes are hexadecimal values. The first byte is the protocol version. This byte is always equal to 2. The next two bytes are a random token. The fourth byte, which is the most important of the first 12 bytes, represents the type of the packet. The bytes that follow until the `rxpk` JSON object are the gateway unique identifier (MAC address). The following table describes what each possible value for the fourth byte means.

Value	Type of packet
0	PUSH_DATA
1	PUSH_ACK
2	PULL_DATA
3	PULL_RESP
4	PULL_ACK

Tab. 5.1.: Possible values for the fourth byte of a UDP packet

The *FNS* has to filter all the packets send by the *UDP Packet Forwarder* based on their fourth byte and keep only the ones with a value equals to 0 or 2 for this byte. In fact,

when an upstream packet is sent from a device to a server, the *UDP Packet Forwarder* sends two UDP packets to the *FNS* to represent this single upstream packet. The first UDP packet is a `PUSH_DATA` packet and contains the actual content of the upstream packet. The second UDP packet is a `PULL_DATA` packet that contains no data but it can be used by the *FNS* to respond to in order to send a downstream response packet from the gateway to the device. Both UDP packets are sent by the same address of the *UDP Packet Forwarder* but from different ports. Afterwards, the *FNS* program applies some more filters to the `PUSH_DATA` packets such that only messages sent in the context of the LoRa-MAC protocol are taken into consideration. This filters include, among others, ensuring that the frequency used to send the packet is 867.5 MHz and that the coding rate is 4/7. Both of this filter data can be found in the `rxpk` JSON payload. Finally, the data field of the `rxpk` JSON payload of the `PUSH_DATA` packets that passed all this filters, needs to be base64 decoded in order to get the actual content of the upstream packets. Once the actual content of the LoRa packet is actually decoded, an acknowledgment UDP packet of type `PUSH_ACK` can be sent from the *FNS* to the *UDP Packet Forwarder*. Afterwards, the decoded content of the upstream packet which is an `Enc0Message` can be processed. It is possible to extract the header of the message and thus obtain the address of the device (`deviceAdd`) that has sent the packet. With the `deviceAdd`, it is possible to get all the data stored for this device on the blockchain. Therefore, the program requests this data to the *Blockchain* by using the `web3` object to call the `Devices` mapping of the smart contract by passing the `deviceAdd` as parameter. The *FNS* program gets as response the IP address and the port of the server to which the packet should be forwarded and the public key of the device. This public key is divided between the *X* and the *Y* values. Now that the program is in the possession of the public key of the device, it can check if the signature present in the `COSE_CounterSignature` structure of the `Enc0Message` is valid. If this is the case, the packet can be forwarded. There is just one last step before. The address of the server returned by the smart contract needs to be converted so that it can actually be used.

In fact, the address of the server can either be an IPv4 or an IPv6 address or a domain name. In the case of an IPv4 or an IPv6 address, the address should be converted from its decimal format to its dotted-decimal format before being used. In the case of a domain name, if it ends with `.eth`, this means that it is an Ethereum Name Service (ENS)¹¹ decentralized domain name and that a special procedure needs to be applied to get the IP address. Otherwise, if it is a "normal" domain name, getting the IP address associated to it is much more straight forward. "The ENS is a distributed, open, and extensible naming system based on the Ethereum blockchain" [48]. ENS has similar goals to DNS and is mostly used to map human-readable names to machine-readable identifiers such as Ethereum addresses. But, it can also be used like DNS to map human-readable names to simple IP addresses. ENS is rather popular and is implemented in the `web3` and `web3s` libraries but unfortunately, the mapping to standard IP addresses is a relatively new feature and thus is not already implemented in Python. Thereby, the project has overcome this limitation by calling directly the ENS smart contract on the blockchain by using its smart contract address and ABI, in order to obtain the corresponding IP address. To perform this call, the human-readable name that is used as a parameter has to be converted to a namehash. This operation is performed in the `namehash.py` file¹².

¹¹<https://ens.domains/>

¹²<https://github.com/inefix/Decentralized-LoRa/blob/master/ForwardingNetworkServer/namehash.py>

Once the program is in the possession of all the details for sending the packet (the address and the port of the server), this one can be sent by opening a WebSocket communication to the *Server*.

In the case where the *Server* wants to respond to the *End Device* with a downstream packet, the *Server* sends a downstream packet back to the *FNS* by using the same opened WebSocket. This time, the *FNS* receives the packet directly in the correct LoRa-MAC format. The *FNS* needs to verify the validity of the signature present in the packet. Thus, it will again call the smart contract in the *Blockchain* by providing the address of the server in the header of the downstream packet to get the public key of the server. Once it gets the public key (the *X* and the *Y* values), it can verify the validity of the signature. If the signature is valid, the downstream packet is programmed to be sent when a PULL_DATA packet is received from the *UDP Packet Forwarder*. Indeed, it is not possible to respond directly to the *UDP Packet Forwarder* on the port from which the PUSH_DATA packet was received because PULL_DATA packets are received on a different port. Thus, a FIFO queue has been implemented to store the downstream packets programmed to be sent. Once a PULL_DATA packet is received, the program checks if there is a downstream packet stored in the FIFO queue to be sent. If the queue is not empty, the downstream packet at the head of the queue is sent to the *UDP Packet Forwarder*. Before that, the *FNS* program has to send an acknowledgment UDP packet of type PULL_ACK to the *UDP Packet Forwarder*. It then has to create an UDP packet containing the downstream packet. An example of such an UDP packet is shown in the following listing.

```

1 \x02\x00\x00 \x03{"txpk":
2   {
3     "imme" : true,
4     "rfch" : 0,
5     "freq" : 867.500000,
6     "powe" : 14,
7     "modu" : "LORA",
8     "datr" : "SF12BW125",
9     "codr" : "4/7",
10    "prea" : 8,
11    "ipol" : false,
12    "size" : 5,
13    "ncrc" : true,
14    "data" : "aGVsbG8="
15  }
16 }
```

List. 5.3: PULL_RESP packet

The first byte of the UDP packet is the protocol version which is always equal to 2. The next two bytes are a random token. The *FNS* always assigns the value 0 to this two bytes. Next comes the byte representing the type of the packet. This byte is set to 3 (PULL_RESP). Then comes the *txpk* JSON object which is very similar to the *rxpk* JSON object. It permits to set all the various parameters of the LoRa packet that will be sent to the device including among others the frequency and the coding rate. Furthermore, the *txpk* JSON object also contains of course the downstream LoRa-MAC packet, encoded in base64, which is placed in the data field. It is important to note that the calculation of the size field is not straight forward and that a special procedure needs to be followed. Finally, the UDP packet containing the downstream LoRa-MAC packet is

sent to the *UDP Packet Forwarder* signaling the end of the LoRa-MAC communication on the *FNS* side.

As with the *Raspberry Pi* program on the *End Device*, during the entire run of the *FNS* program, it is important to catch the eventual errors that can easily happen due to the radio wave nature of the LoRa packets.

5.5. Blockchain

As explained in section 2.5.1, the blockchain technology selected—for the project—is Ethereum and the smart contracts running on it can be written with the Solidity language. The full Solidity code of the deployed smart contract can be found in Appendix A.1. The smart contract is called `LoraResolver` and requires a version of the Solidity compiler comprised between 0.7 and 0.9. Data are always stored in the smart contract by the *AS* and read by the *FNS*. Thus, the smart contract is basically used as a database between the two entities but with the advantage of being totally decentralized. In fact, once deployed on the blockchain, no one can control or alter the smart contract in a bad way and it will run forever or at least as long as Ethereum exists.

The `LoraResolver` smart contract starts by defining a struct called `Device`. This struct permits to store all the required data of an *End Device* such that it is then possible for a gateway to verify that a LoRa-MAC packet has been signed by this *End Device*. Furthermore, the `Device` struct also contains all the informations required for the routing of upstream packets to the server that owns the *End Device*. In more details, it is possible to store the following data in the `Device` struct: the IPv4 address of the *Server*, the IPv6 address of the *Server*, the domain name of the *Server*, the IPv4 port of the *Server*, the IPv6 port of the *Server*, the port associated to the domain name of the *Server*, the Ethereum address of the *Server* that owns the *End Device* and finally the *X* and the *Y* values of the public key of the *End Device*. The IPv4 and IPv6 addresses are stored in decimal format and not in dotted-decimal format. The domain is stored as a string. All the different ports are stored as unsigned integers of 16 bits. The Ethereum address of the owner is stored as an address. The *X* and the *Y* values of the public key of the *End Device* are stored as unsigned int of 256 bits. The Ethereum address of the owner of the *End Device* is mainly used to ensure that only the owner of an end device can modify it once it has been created and stored.

The `LoraResolver` smart contract defines another struct which is called `Server`. This struct permits to store all the required data of a server such that it is possible for a gateway to verify that a LoRa-MAC packet has been signed by this *Server*. In more details, it is possible to store the following data in the `Server` struct: the *X* and the *Y* values of the public key of the *Server* and the Ethereum address of the user that owns this *Server*. As with the `Device` struct, the Ethereum address of the owner of the *Server* is mainly used to ensure that only the owner of a server can modify it once it has been created and stored.

Afterwards, a mapping for the devices is created. This mapping permits to assign a specific `Device` struct to a certain unique integer. This unique integer is the address of the device (`deviceAdd`) which is an unsigned integer of 64 bits.

The mapping for the servers is more complicated since there does not exist a unique format for the address of a server. Thus, there is an `ipv4Servers` mapping for when the address of the server is an IPv4 address, an `ipv6Servers` mapping for the case of an IPv6

address and a `domainServers` mapping for when the address of the server is a domain name.

The smart contract contains six methods that work more or less the same way. Three of them permit to register or update an end device with an IPv4 server address or an IPv6 server address or using a domain name as server address. The other three permit to register or update a server with an IPv4 server address or an IPv6 server address or using a domain name as server address. To simplify the reading and the comprehension, only one function of each group will be analyzed in more details.

The `registerIpv4Device` public function takes as argument a `deviceAdd`, an IPv4 server address and a port, and the X and the Y values of the public key of the device. The function first performs a series of tests to ensure the proper functioning of the smart contract. The first check consists in verifying that the device is not already stored in the `devices` mapping or that the user calling the function is the actual owner of the device. This is a very important check because it ensures that no one can modify a device that is not its property and also that multiple devices with the same `deviceAdd` do not exist. Afterwards, the function performs some tests to ensure that the parameters of the function are not null. Finally, the function stores all the given parameters in the `Device` struct that maps to the provided `deviceAdd`.

The `registerIpv4Server` public function takes as argument an IPv4 server address and the X and the Y values of the public key of the server. Again, the function will first perform a series of tests to ensure the proper functioning of the smart contract. The first check consist in verifying that the server is not already stored in the `ipv4Servers` mapping or that the user calling the function is the actual owner of the server. Again, this is a very important check because it ensures that no one can modify a server that is not its property and that multiple servers with the same IPv4 server address do not exist. The function performs then some tests to ensure that the parameters of the function are not null. Finally, the function stores all the given parameters in the `Server` struct that maps to the provided IPv4 server address.

There is no need for getter functions that return the different mappings of the smart contract because all the mappings are declared as public and can thus be accessed without needing getter functions.

To deploy the `LoraResolver` smart contract, many different solutions exist. Two of them which have been tested are: the Truffle Suite¹³ and the Remix Ethereum IDE¹⁴. For testing locally the interactions of the *FNS* and the *AS* with the *Blockchain*, the Truffle Suite is practical to use. But, for deploying the smart contract on the Ethereum Mainnet or on an Ethereum Testnet, the Remix Ethereum IDE is recommended.

The `LoraResolver` smart contract has been deployed on the Rinkeby Test Network because the OMG Network test network is running on Rinkeby. This permits to use a single Ethereum test network for the entire project. The deployed `LoraResolver` smart contract can be found at this address: `0x4a9fF7c806231fF7d4763c1e83E8B131467adE61`¹⁵.

¹³<https://www.trufflesuite.com/>

¹⁴<https://remix.ethereum.org/>

¹⁵<https://rinkeby.etherscan.io/address/0x4a9fF7c806231fF7d4763c1e83E8B131467adE61>

5.6. Server

As explained, the *Server* is itself divided into two components: the *Home Network Server (HNS)* and the *Application Server (AS)*. Both components are in fact programs that are run simultaneously on the same server. In very short, the *HNS* acts as a back-end component and the *AS* as a front-end / web site component. The *HNS* is programmed in Python and the *AS* in React. Both components are running in a Docker Compose instance which permits to run multi-container Docker applications. This permits to start both components with a single command which is really useful since both components run on the same server simultaneously. The Docker Compose instance is configured by using a YAML file called `docker-compose.yml` which can be found on GitHub¹⁶. During the development of the thesis, both programs were run on a hosted Ubuntu VM at the University of Fribourg. Some more detailed information about how to install all the required packages and the modified version of `pycose` can be found in the `README` file of the `Server` directory. This `README` file also explains how to start the Docker Compose instance. The user needs to provide various variables in a `.env` file.

5.6.1. Home Network Server (HNS)

The program to be run as the *HNS* can be found on the `Server/HomeNetworkServer` directory of the GitHub repository¹⁷. The program has been tested on Python 3.7. The required packages can be found in the `requirements.txt` file. The modified version of the `pycose` library [46] is also required for the program to run properly.

The main file of the *HNS* program is the `server.py` file. This file contains two mains components: an HTTP server that is used as back-end for the server and a WebSocket server which is used to communicate with the *FNS*.

To configure the program, a user needs to provide some MongoDB credentials and the address and port of the server. Furthermore, a private-public key pair is also needed. A detailed explanation of how this pair of keys is generated is already provided in section 5.3.1. Finally, the user also has to provide his Ethereum address.

The program starts with an initialization phase where a MongoDB client (the `motor` Python library), a WebSocket server (the `websockets` Python library) and an HTTP server (the `aihttp` Python library) are initialized. All this initialized components work in an asynchronous way and are described in more details.

The WebSocket server manages the communication with the *FNS*. It is started on port 8765 of address 0.0.0.0. When the WebSocket server receives a LoRa-MAC packet from a gateway which is in fact a `Enc0Message`, it starts by extracting the header from it. The program has to first verify that it has not already received this LoRa-MAC packet from this device. This is achieved by comparing the counter of the received packet and the counter of the last received packet stored in the database. If the new packet is actually a new one, the program verifies that the gateway sending the packet is the same as the one who sent the last received packet stored in the database. This procedure will be discussed in more details later but could permit to the *HNS* to not open micropayment channels with too many different gateways. Thus, if a packet is received from another gateway than the usual one, the program waits 10 seconds to see if a packet is received from the

¹⁶<https://github.com/inefix/Decentralized-LoRa/blob/master/Server/docker-compose.yml>

¹⁷<https://github.com/inefix/Decentralized-LoRa/tree/master/Server/HomeNetworkServer>

usual gateway. Otherwise, the program considers this new gateway as the new usual one and processes its packet.

The processing of the received LoRa-MAC packet starts by getting the public key of the *End Device* in the database of the *HNS*. The program can then verify if the signature present in the `COSE_CounterSignature` structure of the `Enc0Message` is valid. If this is the case, the program generates the symmetric key with the public key of the *End Device* and the private key of the *Server* in order to be able to decrypt the ciphertext of the packet. The decrypted payload is then stored in the database such that the *AS* will be able to access it and display it to the user.

Afterwards, a response packet is sent to the *FNS* through the same opened `WebSocket`. This response packet has either been partially pre-stored by the *AS* or is totally created by the *HNS*. So, the program has to first verify if a pre-stored payload is present in its database. If this is the case, the program constructs a LoRa-MAC downstream packet by providing an appropriate header (using the right `MType`, counter and device address), encrypting the payload and signing the packet. The procedure is explained in more details in section 5.3.1. If no pre-stored payload is present in the database, the *HNS* generates a payload and an header and constructs a downstream packet. In both cases, the counter value to be included in the header of the message is stored in a file called `counter.txt`, as it is done in the *Raspberry Pi* program (section 5.3.1).

The HTTP server acts as a back-end RESTful service for the *AS*. It is running on port 8080 of address 0.0.0.0. The HTTP server exposes URL paths that are callable by the *AS* by using HTTP requests with different HTTP methods (`GET`, `POST`, `PATCH`, `DELETE` methods). For each URL path and its HTTP method, a corresponding operation on the database is performed. In fact, the HTTP server is mainly used to permit to the *AS* to access the database of the *HNS*. The database is divided in four collections: `device`, `msg`, `gateway`, `down`. The `device` collection, as the name suggest, stores all the required data at the creation of a new device (public key, `deviceAdd`, server address and port, etc). It is important to note that this collection does not contain any of the private keys of the devices because they have been created on the *End Device* side and the private keys should not be used outside of the *Raspberry Pi* program. The `msg` collection stores all the received messages from the *FNS*. A message stored in this collection consists of the header (the `MType`, the counter and the `deviceAdd`), the payload, the Ethereum address of the owner and the gateway, if the message has been payed, etc. The `gateway` collection is used for the micropayment channels and permits to store all the data about a micropayment channel (the address of the deployed micropayment smart contract, the amount already payed, the total amount stored in the smart contract and the expiration time of the smart contract). Finally, the `down` collection stores all the downstream messages programmed by the *AS* to be sent as response packets to the end devices. These down messages stored contain the `deviceAdd`, the payload, the date and time and if they have been transmitted to the *End Device*.

The *HNS* also has a really important role in the Micropayment extension but this will be explained in section 5.7.4.

5.6.2. Application Server (AS)

The program to be run as the *AS* can be found on the `Server/ApplicationServer` directory of the GitHub repository¹⁸. The program has been tested on Node 14. The *AS* is a front-end web site developed with React. It permits, among other features, to register new devices on the blockchain, to consult the content of the upstream packets sent from an *End Device* to the *Server* and to program downstream packet responses to be sent from the *Server* to the *End Device*. For registering new devices on the web site, it is mandatory for the user to have installed a browser cryptocurrency wallet software that can be used to interact with the Ethereum blockchain. An example of such a browser software is the MetaMask¹⁹ browser extension.

The *AS* web site consists of three web pages: the `Devices` page, the `Messages` page and the `Down` page. Each page is developed in one `.jsx` file inside the `src/component` directory of the `ApplicationServer` directory. This directory contains also some other useful files like the `abis.js` file which contains the ABI of the `LoraResolver` smart contract and the `Style.css` file which defines the CSS style of the three web pages. All three pages and their source code are going to be detailed.

The `Devices` page lists all the devices registered by the server which are stored in the `device` collection in the database of the *HNS*. This web page acts as a nice web interface for interacting with this database collection through HTTP requests (GET, POST, PATCH, DELETE requests). In fact, HTTP requests are sent to the *HNS* back-end which performs a corresponding operation on the database and returns the result as a JSON object to the web page. This procedure of HTTP requests is the same for the two other web pages and thus will not be mentioned anymore. For each device, the web page displays a card containing the `deviceAdd`, the public key and the name of the device. It is possible to define a name for each end device to facilitate the reading and the comprehension of the end user. For registering a new device, the user needs to click the blue button on the top right corner with the text "Add device". Once this button has been clicked, a modal form is presented to the user which lets him register his new device with a nice interface. The user has to complete the different fields. The name field is not mandatory but the `deviceAdd` and the public key ones are. The data to complete this fields can be generated with the *Raspberry Pi* program for example. Once the modal form has been completed, the user can register his new device by clicking the corresponding button. By clicking this button, the user is asked to pay some ETH for the transaction to be submitted on the blockchain. This permits to register a new device on the `LoraResolver` smart contract. Once the user has payed for the transaction fees on his cryptocurrency wallet, he needs to wait a few seconds for the transaction to be processed on the blockchain. Afterwards, the newly registered device is shown along with the other devices registered on the page. Each card containing a device has three buttons: `send`, `modify`, `delete`. The `send` button opens a modal form that permits to write and program a payload that will be sent in response to an upstream packet received by the *Server* from this *End Device*. The `modify` button also permits to open a modal form but this time to modify the name of the device. Finally the `delete` button, as the name suggest, permits to delete a device. Moreover, the page contains a `reload` button on the top right corner that permits to reload the devices that are stored in the `device` collection. Finally about the `Devices` page, each time the web page is loaded, the web site first checks if the server hosting it

¹⁸<https://github.com/inefix/Decentralized-LoRa/tree/master/Server/ApplicationServer>

¹⁹<https://metamask.io/>

is already registered in the blockchain and if it is not the case, a transaction to register the public key is performed.

The **Messages** page lists all the upstream messages received by the server from all the devices. All this messages are stored in the **msg** collection in the database of the *HNS*. For each message, the web page displays a card containing the date and time at which the message has been received, the payload and the header of the message. Each message card has two buttons: **respond** and **delete**. The **respond** button has the same purpose as the **send** button in the **Devices** page which opens a modal form that permits to write and program a payload that will be sent in response to an upstream packet. The **delete** button permits to delete a message. Finally, the **Messages** page also contains a **reload** button on the top right corner that permits to reload the messages received by the *HNS*. The **Down** page lists all the downstream messages programmed by the *AS* for all the end devices. All this downstream messages are stored in the **down** collection in the database of the *HNS*. This web page is very similar to the **Messages** page. For each downstream message, the web page displays a card containing the date and time at which the message has been stored in the database collection, the payload of the message, the **deviceAdd** of the *End Device* to which the downstream message will be sent to and an indication about if the downstream message has already been sent. Each downstream message card has two buttons: **modify** and **delete**. The **modify** button opens a modal form that permits to modify the payload programmed for this downstream message. The **delete** button permits to delete a downstream message. Finally, the **Down** page contains a **reload** button on the top right corner like the other two pages that permits to reload the downstream messages which are stored in the **down** collection.

Screenshots of the three web pages just described and their functionalities can be found in Appendix B.

5.7. Extension: Micropayment

The section will discuss how the two solutions of Ethereum layer 2 scaling were used and implemented in order to enable micropayments between the *Server* and the *Gateway*. It is important to recall that payments are made only unidirectionaly from a server to a gateway. The section will also discuss which changes had to be done to the initial sequence diagram such that remuneration is actually feasible and makes sense. The following sequence diagram highlights where the Micropayment extension takes place in the original sequence diagram.

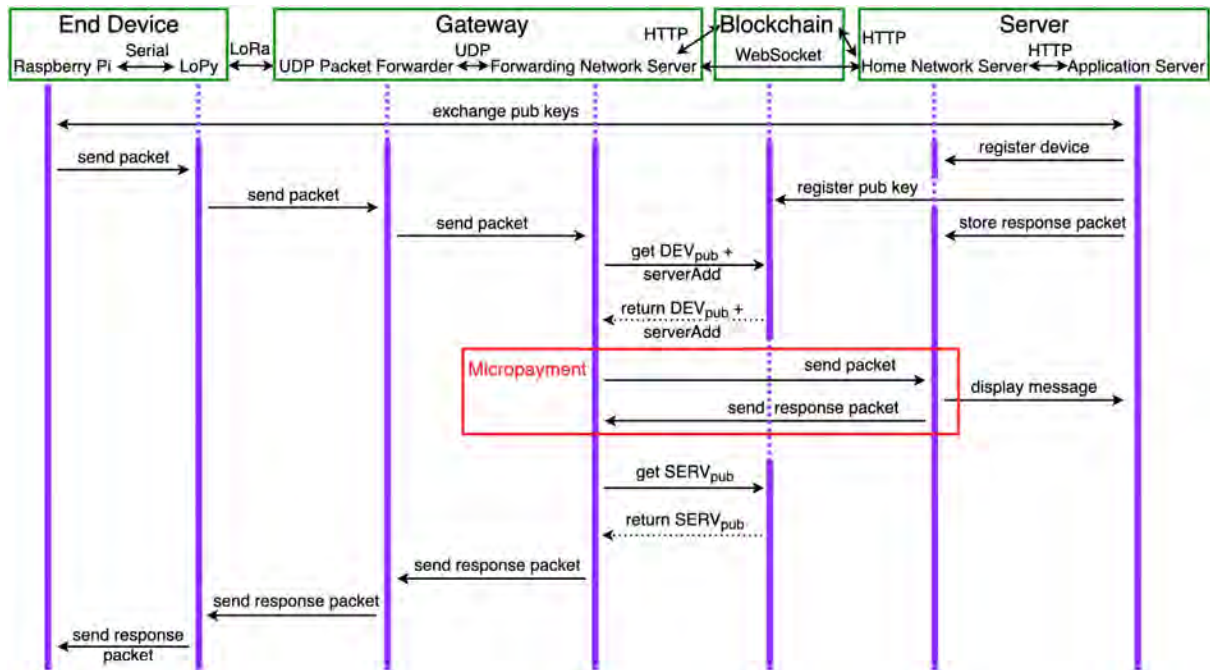


Fig. 5.4.: Sequence diagram with the Micropayment extension highlighted

The next sequence diagram describes in details the interactions between the *Gateway*, the *Blockchain* and the *Server* in the Micropayment extension.

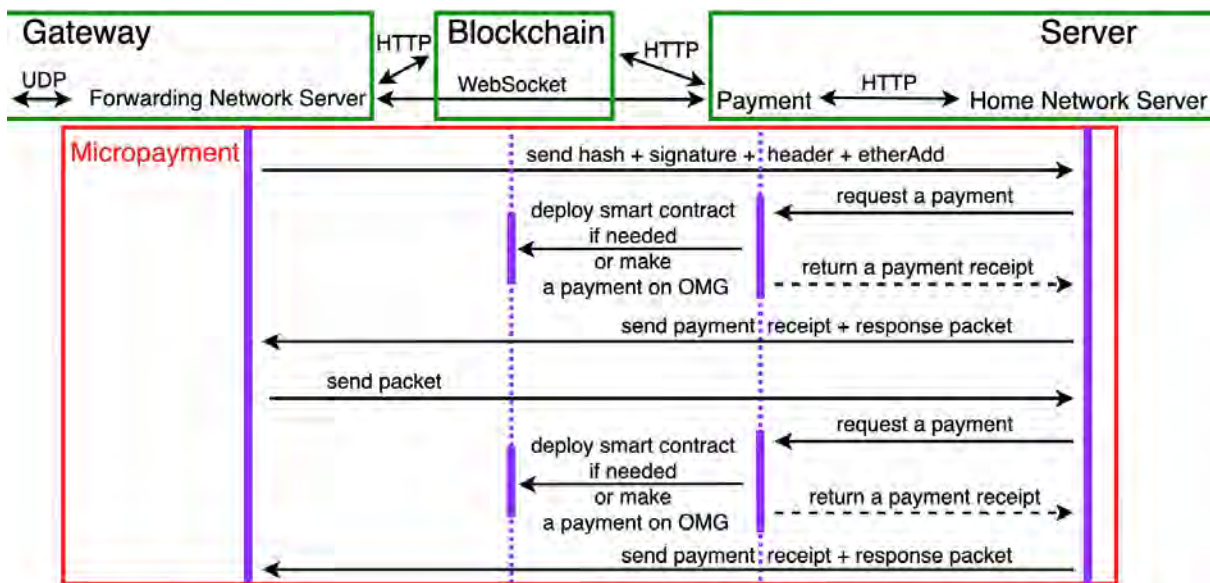


Fig. 5.5.: Sequence diagram of the Micropayment extension

The components of the *Gateway* remain the same but a new component is added to the *Server*: the *Payment* service. This new component is used by the *Server* to perform all the payment related tasks. This service has been developed in JavaScript because the *web3* library is much more complete and standardized in JavaScript than in Python and because the library to interact with the *OMG Network* is only available in JavaScript. The major change in the original communication between the *Gateway* and the *Server*

is that now the *Gateway* does not send the upstream packet directly when it is received from the *End Device*. In fact, if the *Gateway* would have forwarded the entire upstream packet as previously, there would be no incentive for the *Server* to pay for that packet since this last one would already be in possession of the entire packet. This is why, on reception of an upstream packet on the *Gateway* side, the *Gateway* now performs a hash of all the protected elements of the packet that the signature authenticates. This hash is then sent along with the signature to the *HNS* such that this one can still verify the signature of the packet but over the hash. This procedure permits to the *Gateway* to prove that an upstream packet as really been received but without giving the actual content for "free". This procedure also permits to the *Server* to be sure that an upstream packet has really been received by the *Gateway* since the signature sent can only be generated by the private key of the *End Device*. Along with the hash and the signature of the upstream packet, the *Gateway* also sends the header of the message and its Ethereum address in order to get paid. The header of the message can be sent by the *Gateway* because it does not contain any hint about the payload of the message. Furthermore, the `deviceAdd` is useful for the *Server* to know about which device the packet comes from and the counter permits to verify that the message has not already been received on the *Server* side.

On the *Server* side, if the signature received is valid, the *HNS* requests for a payment to the *Payment* service. The *Payment* service performs the payment on the blockchain either using a micropayment channel or using the OMG Network. A receipt of the payment is sent back to the *HNS* which forwards it to the *FNS*. This last one verifies the payment receipt and if the money has been received, it sends the entire upstream packet to the *HNS*. On reception, the *HNS* performs the usual procedure to verify the signature of the packet and decrypt the ciphertext. If the *HNS* has a downstream packet to send back to the *End Device*, it has to pay for the transmission of this downstream packet as well. Thus, a payment is requested to the *Payment* service which processes it and returns the payment receipt. Finally, the payment receipt and the response packet can be sent to the *FNS* which again has to verify the payment before forwarding the packet to the *UDP Packet Forwarder* if everything is as expected. This last step concludes the sequence diagram of the Micropayment extension.

Of course, for performing micropayments on the blockchain, the user needs to have ETH associated to its Ethereum address. Furthermore, if micropayments are performed on the OMG Network, the user needs to deposit some of his ETH on the OMG Network. This can be done by using the OMG Network Web Wallet²⁰. Last precision about the OMG Network, it is advised for each user deploying a gateway to first connect his Ethereum wallet to the OMG Network Web Wallet as well. However, no deposit of ETH on the web wallet is needed for these users.

The followings subsections will discuss in more details the implementation of the Micropayment extension for each component involved.

5.7.1. Forwarding Network Server (FNS)

To configure the *FNS* for the Micropayment extension, the user needs to provide his Ethereum address and the private key associated. The private key is used on the gateway side only to close micropayment channels.

On reception of an upstream packet from the *UDP Packet Forwarder*, the *FNS* verifies

²⁰<https://webwallet.mainnet.v1.omg.network/>

the signature of the packet as previously explained. Once the signature is verified, the `Countersign_structure` is constructed from the packet. This structure is then hashed using the SHA256 algorithm because the signature of the packet is performed on the *End Device* side by using the ES256 algorithm which is, in more details, the ECDSA algorithm applied to an hash realized by the SHA256 algorithm. This results in the hashed structure which is sent to the *HNS* along with the extracted signature and header from the packet and the Ethereum address of the user who runs the *Gateway*.

Afterwards, the *FNS* receives a payment receipt from the *HNS*. This payment receipt can be of two different forms depending on the micropayment solution used. If the OMG Network has been used, the payment receipt is in fact the transaction hash of the transaction that has been submitted on the OMG Network. This transaction hash can be verified by the *FNS* only after more or less two minutes. This is one of the restrictions of the OMG Network which will be discussed in more details in section 6.4. This two minutes waiting time implies that the *HNS* has to send, along with the payment receipt, also the downstream response packet that has to be forwarded from the *Gateway* to the *End Device*. In fact, a two minutes waiting period is a too long waiting time for the *End Device* to keep listening for incoming packets. Indeed, the *FNS* program will process the downstream packet as usual by verifying the signature and constructing the UDP packet to be sent to the *UDP Packet Forwarder*. Once the downstream packet sent, this thread of the *FNS* program sleeps for two minutes before verifying the transaction hash. In fact, the *FNS* program opens a new thread for each new upstream packet received such that even if a thread needs to wait, the *FNS* program can still continue its work by receiving / processing new incoming upstream and downstream packets. After the two minutes sleep, the thread of the *FNS* can verify the transaction hash of the OMG Network transaction. Since the OMG Network only provides a library for the JavaScript language and none for Python, the complete process of transaction hash verification has to be totally implemented. This is achieved by making a RPC call containing the transaction hash to a OMG watcher which returns various data about the transaction on the OMG Network. It is then possible to extract the sender, the currency, the amount, the receiver and the metadata associated to this OMG transaction. All this data can then be verified by the *FNS* to ensure that it has been paid for forwarding the LoRa-MAC packet. It is important to mention that if the *Server* wanted to send a downstream packet, it had to pay for two packets: the upstream and the downstream packet. This element is also verified by the *FNS* and if there is any issue with the transaction data, the upstream packet is simply not sent by the *FNS* to the *HNS*. Otherwise, the upstream packet is forwarded to the *HNS* and this conclude the communication between the two components.

On the other hand, if the micropayment solution used is a micropayment channel, the payment receipt contains a signature, the address of the smart contract of the micropayment channel and the amount payed for the upstream packet. The program has to perform different checks to ensure that the signature and the smart contract are valid such that it will then be possible to recover the funds stored in the smart contract. The *FNS* program stores the ABI of the `SimplePaymentChannel` smart contract such that it can open a `web3` client by providing the contract address in order to interact with the smart contract deployed by the *Payment* service. In addition, the *FNS* program also stores the runtime bytecode of the `SimplePaymentChannel` smart contract such that it is possible to ensure that the *Payment* service as really deployed the correct `SimplePaymentChannel` smart contract without any modifications. The runtime bytecode is a low-level programming language which is compiled from a high-level programming language such as Solidity [12].

Since the bytecode is derived from the Solidity code of the smart contract, if there is a single modification in it, the bytecode does not match anymore. In addition, the program also verifies the expiration time of the deployed smart contract. Afterwards, the program gets the data stored about the payment channel in its database. The data mainly consist of the previous signature stored and the amount of ETH owned by the *Gateway* in the smart contract. A check is then performed to ensure that there are enough available funds in the smart contract in order to pay for transferring this new packet. If all checks succeed, the *FNS* program can verify the signature sent by calling the `isValidSignature` function of the smart contract. If the signature is valid, the program checks if it is time to close the payment channel. In fact, the user can configure a threshold of balance (percent of used funds) and time (number of second before the expiration) that if exceeded, will close the payment channel automatically. If one of the two threshold is exceeded, the *FNS* program calls the `close` function of the smart contract by creating an Ethereum transaction signed by the private key. This transaction contains the last signature stored and the total amount of ETH owned. Accordingly, owned funds are sent to the Ethereum address of the *Gateway* and the remaining funds, if any, are sent to the address of the *Server*. In this case, the *FNS* program has to later notify the *HNS* of the closure of the smart contract. Otherwise, if no threshold is exceeded, the program considers the payment as valid and stores the new signature and the new total amount of ETH in its database. The upstream packet can now be sent to the *HNS* along with an eventual notification of smart contract closure. With micropayment channels as type of payment, it is possible for the *HNS* to respond with a downstream packet and a new payment receipt. If this is the case, the *FNS* program verifies again the payment receipt by following the same procedure as the one just described and if the payment is valid, the downstream packet is sent to the *UDP Packet Forwarder* by following the same procedure as usual.

5.7.2. Blockchain

To open a micropayment channel, the *Payment* service of the *Server* has to deploy a smart contract called `SimplePaymentChannel` on the Ethereum blockchain. This smart contract can be found on GitHub²¹. The smart contract and the procedure for creating micropayment channels have been taken from the official Solidity documentation [43]. The `isValidSignature` function of the `SimplePaymentChannel` smart contract has been modified to make it public such that it is now callable from the *FNS* in order to be able to verify a signature received by the *HNS*. This change is not required since the verification can be done totally offline but, it is more convenient to do it in this way. In addition, since it is a view function (i.e., it does not modify the state of the contract), calling it does not consume any gas. The most important methods of the smart contract will be explained in more details. When deploying the `SimplePaymentChannel` smart contract, the *Payment* service has to pass as argument the Ethereum address which will receive the payments, the amount of ETH stored in the contract and the duration of the payment channel. All this parameter are stored in the smart contract.

The `isValidSignature` function permits to verify the validity of a signature by passing as argument a signature to verify the amount value which has been used at the computation of the signature. The function starts by constructing the payment message by first

²¹<https://github.com/inefix/Decentralized-LoRa/blob/master/Server/Payment/SimplePaymentChannel.sol>

concatenating and encoding the address of the contract and the amount. The resulting payment message is then hashed with the keccak256 algorithm. It is then possible to use the `ecrecover` build-in function in Solidity that permits to return the address (public key) that was used to sign the payment message (the payment message and the signature have to be passed as argument). The `isValidSignature` function returns if the address obtained is the one used to deploy the smart contract. This permit to verify that a signature has been generated by the correct Ethereum address and that this signature has effectively been computed over the correct address of the smart contract and the correct amount value.

The `close` function has to be called by the Ethereum address which will receive the payment. This function takes has parameter an amount value and a signature. It first checks the validity of the signature by using the `isValidSignature` function. If the signature is valid, funds equals to the amount value passed are sent to Ethereum address which called the `close` function and the remaining funds of the smart contract are sent back to the address that deployed it.

The `extend` function can be called to extend the expiration of the micropayment channel. This function can only be called by the address which has deployed the smart contract. Finally, the `claimTimeout` function can only be called by the address which has deployed the smart contract if the payment channel has expired. This function sends all the ETH present in the smart contract back to address that deployed it.

The procedure to generate the micropayment signatures is described in the next subsection.

5.7.3. Payment service

The program to be run as the *Payment* service can be found on the `Server/Payment` directory of the GitHub repository²². The program has been tested on Node 14 and is an HTTP server (using the Koa JavaScript library) that runs on port 3000 of the localhost address of the *Server*. The service also runs on the same Docker Compose instance as the HNS and the AS. It exposes three URLs that can be called by the *HNS* to perform all the necessary payment related functionalities of the *Server*. This three URLs redirect to three functions: `payment`, `signPayment` and `deploy`. To configure the *Payment* service program, the user needs to provide his Ethereum address and the private key associated to it. The private key is used on the *Payment* service to sign transactions on the OMG Network, to deploy `SimplePaymentChannel` smart contracts for opening micropayment channels and to sign micropayment transactions.

The `payment` function can be called to send a payment on the OMG Network thanks to the OMG Network JavaScript library, `omg-js`²³. The function needs the address of the receiver of the payment, the amount to transfer and the metadata to include in the transaction. The function constructs the transaction, signs it using the private key and submits it on the OMG Network. The transaction hash of the submitted transaction is then returned by the function.

The `signPayment` function can be called to generate a signature used as proof of payment in micropayment channels. The function takes as arguments the address of the smart contract and the amount of the payment. The procedure starts by constructing

²²<https://github.com/inefix/Decentralized-LoRa/tree/master/Server/Payment>

²³<https://github.com/omgnetwork/omg-js>

the payment message by first concatenating and encoding the address of the contract and the amount, and then by hashing the result with the keccak256 algorithm. The payment message is then signed using the private key. The function then returns the computed signature.

The `deploy` function can be called to deploy a new `SimplePaymentChannel` smart contract and thus to create a new micropayment channel. The function takes as arguments the address of the receiver of the payment, the amount to initialize the smart contract with and the duration of the payment channel. A transaction is created for deploying the smart contract by using the ABI and the bytecode of the contract. The arguments provided to the function are also added in the transaction which is signed using the private key and then deployed on the Ethereum blockchain. The function returns the address of the newly deployed smart contract.

5.7.4. Home Network Server (HNS)

When receiving the hash of the `Countersign_` structure, the signature and the header of the upstream packet along with the Ethereum address of the *FNS*, the *HNS* starts by verifying the hash and the signature. This is done by first getting the public key of the device in the database and then using a function to verify the digest (the hash) by using the signature and the public key. If the verification succeed, the *HNS* asks for a payment to the *Payment* service.

For a payment on the OMG Network, the *HNS* has to provide the metatada which include the counter present in the header of the message and the `deviceAdd`. Furthermore, the *HNS* has to provide to the *Payment* service the Ethereum address of the *Gateway* and the amount to send (this amount can be the price of a single message or the one of two messages). The *HNS* sends an HTTP request to the *Payment* service which returns the transaction hash as payment receipt.

On the other hand, for a payment using micropayment channels, the *HNS* program first verifies if a micropayment channel has already been opened with this *Gateway* and otherwise opens a new one. Since opening a new micropayment channel implies to deploy a smart contract on the blockchain which cost a lot of gas, it is important to have a mechanism to try to use as much as possible existing, already opened micropayment channel. This is where the functionality to wait 10 seconds when a packet is received from a new *Gateway* takes a lot of sense. If it is still needed to open a new micropayment channel, the *HNS* program has to call the *Payment* service with an HTTP request by providing the address of the *Gateway*, the amount of ETH to lock in the smart contract and the duration of the payment channel. This last two parameters can be configured by the user when starting the *HNS* program. The *Payment* service returns the smart contract address that has to be stored along with the data sent as parameters in the database of the program. Once a payment channel exists between two addresses, the *HNS* program has to provide to the *Payment* service only the address of the smart contract and the total amount owned by the *Gateway* (the amount already owned plus the price of the new packet). The *Payment* service returns the signature of the payment as payment receipt to this HTTP request. It is important for the *HNS* program to always update the database with the last amount owed.

Afterwards, the payment receipt can be sent to the *FNS*. In the case of using the OMG Network, it is also possible to send a downstream packet programmed by the *AS* due to

the 2 minutes waiting time restriction described previously. In the case of micropayment channels, the payment receipt is sent along with the smart contract address and the price of one message. If the verification of the payment succeeded on the *FNS* side, the *HNS* program receives the desired upstream packet along with a possible notification of micropayment channel closure. If the notification is present, the *HNS* verifies that the smart contract balance is effectively equal to zero in order to delete this payment channel from its database.

The *HNS* can then finally process the received upstream packet as usual. In the case of using micropayment channels, it is then possible to respond with a downstream packet either programmed by the *AS* or generated by the *HNS*. The eventual downstream packet has to be paid using the same procedure such that it can be sent with the payment receipt to the *FNS*.

5.8. Outlook

The underneath sequence diagram gathers the original sequence diagram of the project and the one from the Micropayment extension into a single one. This final sequence diagram permits to have a great understanding and overview of the entire implementation that has been realized for the thesis.

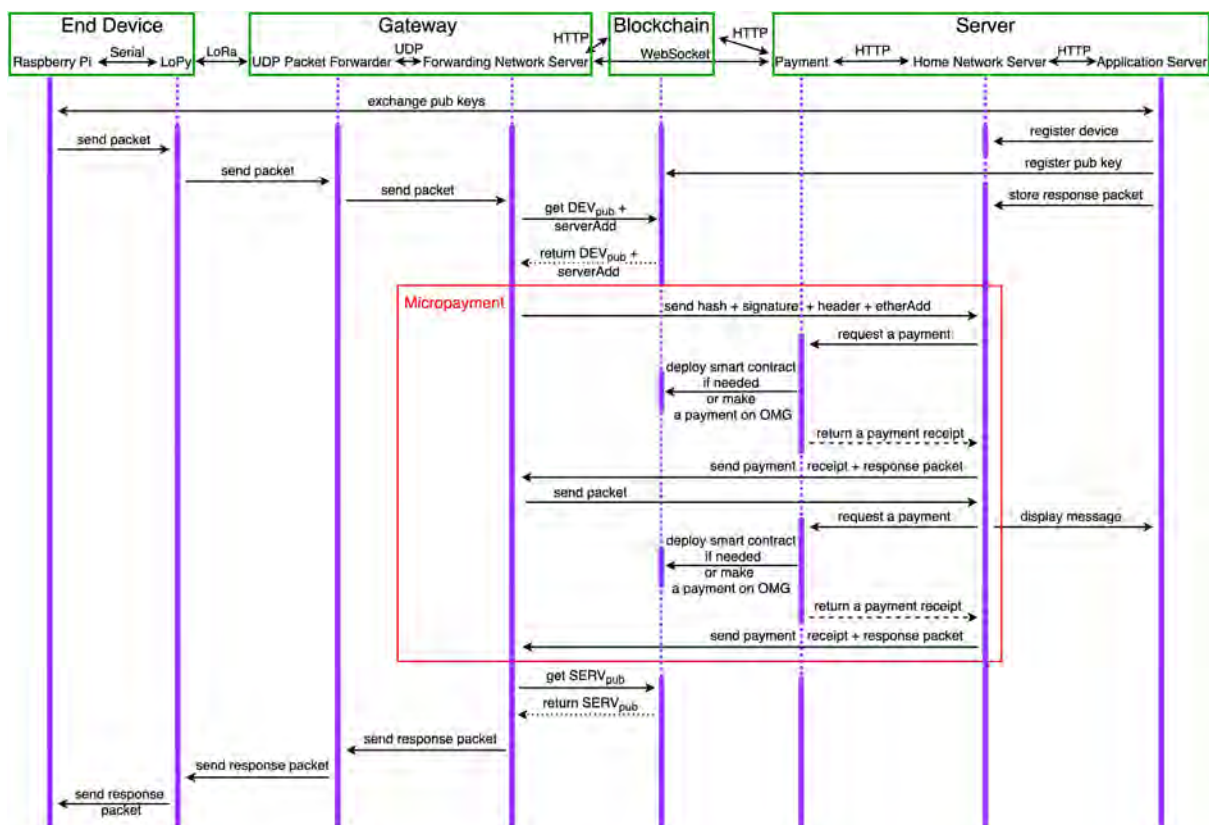


Fig. 5.6.: Final sequence diagram of the entire thesis

5.9. Pycose library

The `pycose` library [49] is a Python implementation of COSE which is available in RFC 8152 [4]. The library permits to create various COSE messages including: `Sign`, `Sign1`, `Mac`, `Mac0`, `Encrypt` and `Encrypt0`. Unfortunately, the library does not support the addition of a `COSE_CounterSignature` structure to the provided COSE messages. Since this project wants to use countersignatures in order to have COSE messages as short as possible, a countersignature implementation in Python is needed. Thus, a library that permits to add `COSE_CounterSignature` structure has been developed. This library is a fork of the `pycose` library [46].

The fork mainly consist in the addition of a file called `countersignmessage.py`²⁴ in the `cose/messages` directory. The implementation of the countersignature feature respects the way COSE messages are created by the original `pycose` library in order to keep a certain consistency across the new library. However, this new functionality differs from the other features of the library because it modifies a COSE message instead of creating a new one. The `countersignmessage.py` file and its most important methods are explained in more details.

The `countersignmessage.py` starts by defining the CBOR tag associated with the `COSE_CounterSignature` structure. This tag has the value 11 as the examples in the Internet-Draft `COSE_CounterSign` [5] suggest. The file defines a `CountersignMessage` class which contains a few variables mainly used to store the different fields of the COSE message to which the `COSE_CounterSignature` structure is added. This variables will be used to reconstruct the full COSE message at the end.

The `__init__` function permits to initialize the fields of the `COSE_CounterSignature` structure which are: the protected header, the unprotected header and the signature. In addition, this function stores the different fields of the COSE message in the previous mentioned variables.

The `_sig_structure` function is used to construct the `Countersign_structure` and to return it. Since the process to generate the signature is explained in details in section 3.3 and the code mostly follows the explanations provided, the process is not discussed again.

The `encode` function is used to reconstruct the full COSE message (the original COSE message with the `COSE_CounterSignature` structure appended in the unprotected part of the header of the COSE message).

Finally, the `verify_signature` function, as the name suggest, is used to verify the signature present in a COSE message. This is done by extracting the signature from the `COSE_CounterSignature` structure and then verifying it with the public key provided.

The fork of the library also implies the addition of the `CountersignMessage` object in the `__init__.py`²⁵ file in the `cose/messages` directory and in the modification of the identifier value from 7 to 11 in the `CounterSignature` class of the `headers.py`²⁶ file in the `cose` directory.

²⁴<https://github.com/inefix/pycose/blob/master/cose/messages/countersignmessage.py>

²⁵https://github.com/inefix/pycose/blob/master/cose/messages/__init__.py

²⁶<https://github.com/inefix/pycose/blob/master/cose/headers.py>

6

Evaluation

6.1. Introduction	57
6.2. LoRa-MAC packet size	57
6.3. Antennas	59
6.4. Micropayment Evaluation	60
6.4.1. Use-cases	61
6.4.2. Transaction fees	61
6.5. Issues	63
6.6. Limitations	64

6.1. Introduction

This chapter presents an evaluation of the implementation. The chapter starts by a comparison of the size of the different components of a LoRa-MAC packet. Then, a section comparing various antennas highlights the gaps that exists between different models. Afterwards, a comparison of both micropayment solutions describes the advantages and the disadvantages along with the use-cases of each solution. This section also contains a comparison of the transaction fees of both solutions. The chapter then presents some issues that were encountered. Finally, a section containing some precisions about the limitations of the implementation is presented.

6.2. LoRa-MAC packet size

The LoRa-MAC protocol aims at providing the highest level of security. Unfortunately, this comes at the expense of message size. In fact, sending a "helloWorld" payload which has a length of 10 characters / bytes results in a final LoRa-MAC packet having a length of 188 characters. Of course the packet is longer because it has an header but the header is only 35 characters long thanks to the use of a tag for the MType.

To provide the highest level of security, the payload has to be encrypted in the ciphertext which is 26 characters long. Furthermore the ciphertext has to be signed. This results

in a signature of 64 characters. Someone could argue that the signature is not needed since the ciphertext is encrypted with AES-GCM which is an authenticated mode for AES. Unfortunately, AES-GCM does not provide non-repudation since the ciphertext is encrypted with the symmetric / private key. The following tree sums up the size of the different components of a LoRa-MAC packet.

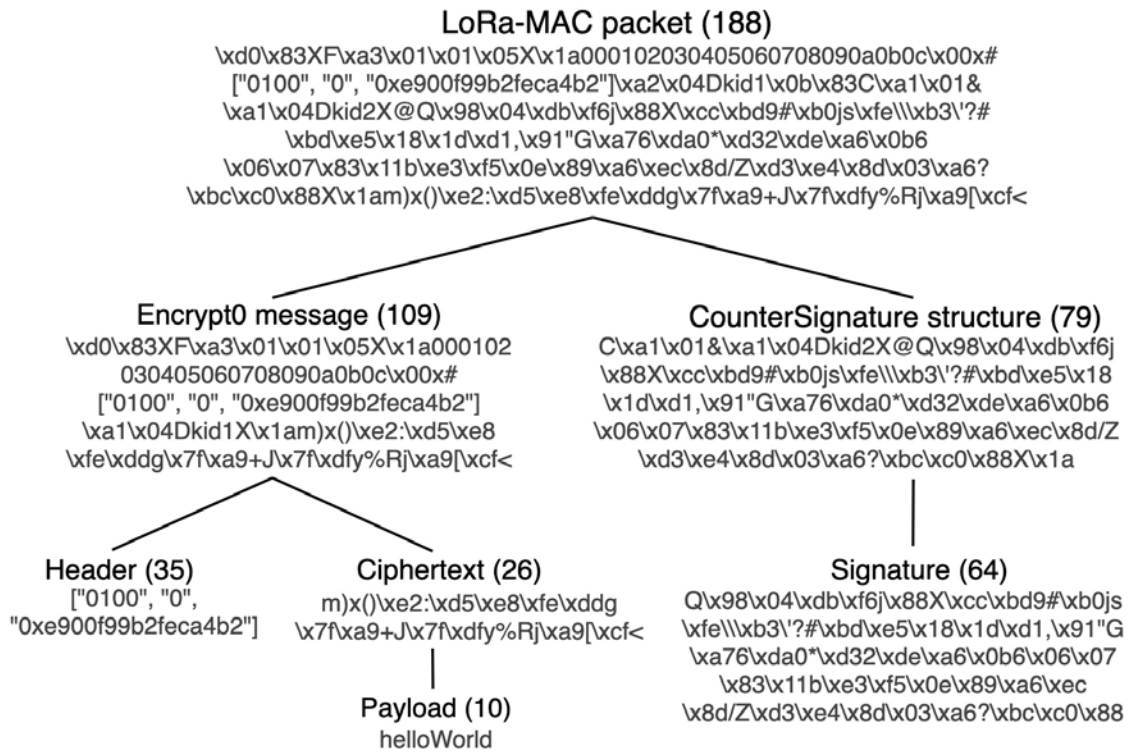


Fig. 6.1.: Size of the different components of a LoRa-MAC packet

The subsequent table shows the size in characters / bytes and the overhead of the LoRa-MAC and LoRaWAN packets obtained from payloads of different sizes. The size of the LoRaWAN packets are much smaller since LoRaWAN does not provide non-repudation through the use of digital signatures. In fact, the integrity and the authenticity of messages in LoRaWAN are ensured by a Cipher-based MAC (CMAC) algorithm (AES-CMAC [13]) and the MIC associated is the first 4 bits of the calculated CMAC. In addition, this can also be explained with the fact that LoRaWAN is more compact and less verbose than LoRa-MAC.

Payload Size (bytes)	LoRa-MAC		LoRaWAN	
	Size (bytes)	Overhead (%)	Size (bytes)	Overhead (%)
4	181	4525	17	425
8	186	2325	21	262.5
16	194	1215.5	29	181.25
32	209	653.125	45	140.625

Tab. 6.1.: Size and overhead of LoRa-MAC and LoRaWAN packets

The larger packet size of LoRaWAN can result in higher power consumption for the devices and decreased reliability of communication. In fact, higher power consumption

is due to longer time spent on cryptographic operations as well as on transmission and decreased reliability results from higher chance of message collision and / or corruption [1]. Finally, it is interesting to note that the overhead percentage decreases as the size of the payload increases.

6.3. Antennas

The quality and the size of the antennas used by the end devices and the gateways make an important difference in the quality of the communication in the entire LoRa-MAC protocol. In fact, having a poor quality antenna can result in packet being corrupted or not being received at all. This is why, multiple antennas have been tested on the *Gateway* side in order to find a suitable one. This section contains the results obtained when comparing three different antennas on the gateway side. The subsequent picture shows the dimensions of the three antennas tested. The smallest one is the base antenna provided when buying the RAK831 LPWAN Gateway Concentrator Module¹, the medium² and the biggest³ one can be bought on Digi-Key.

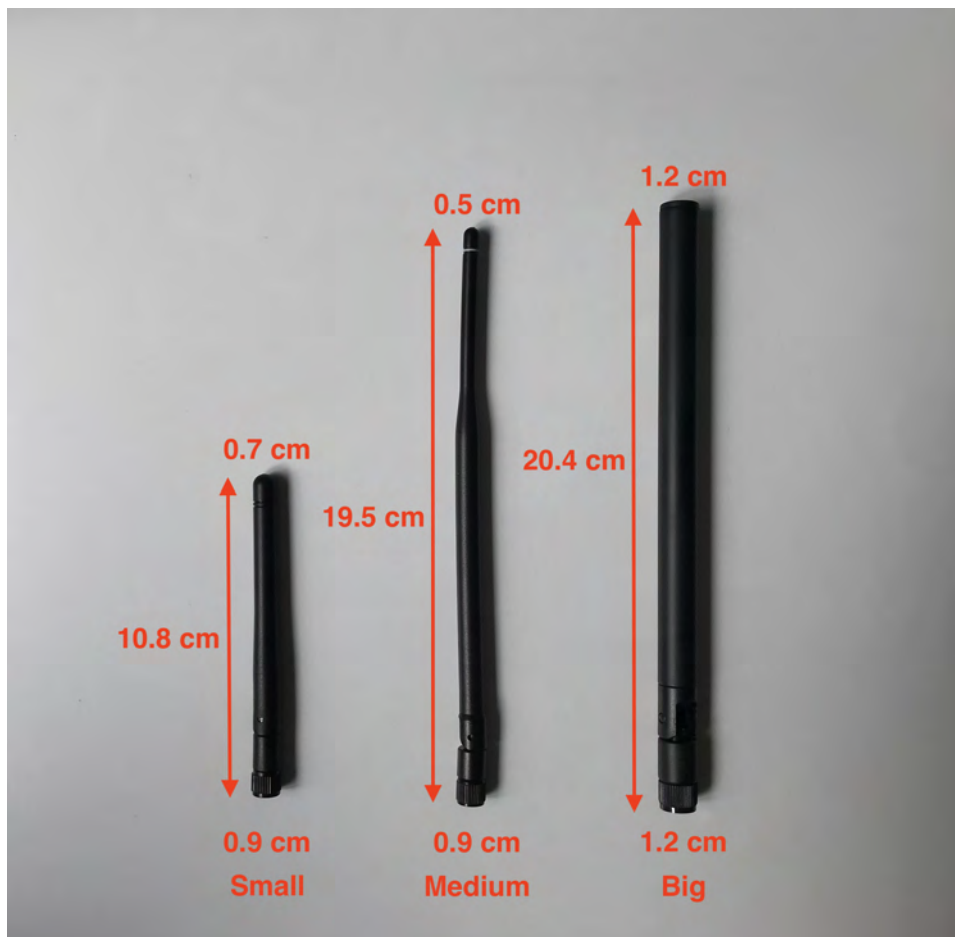


Fig. 6.2.: Dimensions of three different antennas

¹<https://store.rakwireless.com/products/rak831-gateway-module>

²<https://www.digikey.ch/product-detail/en/linx-technologies-inc/ANT-868-OC-LG-RPS/343-ANT-868-OC-LG-RPS-ND/12158019>

³<https://www.digikey.ch/product-detail/en/nearson-inc/S1551AH-868S/730-1067-ND/7320173>

The subsequent table contains the Received Signal Strength Indication (RSSI) values measured in dBm at the reception of LoRa packets sent by a device at various distances in meter from the *Gateway*. All the packets are sent using the same parameters as the ones in the LoRa-MAC protocol, which means using a frequency of 867.5 MHz, a spreading factor of 12, a coding rate of 4/7 and a bandwidth of 125 KHz.

		1 m			3 m			5 m			10 m		
Antennas	Small	-46	-46	-47	-62	-66	-66	-67	-59	-61	-51	-52	-52
	Medium	-28	-28	-29	-46	-44	-49	-49	-52	-52	-43	-43	-43
	Big	-29	-28	-28	-37	-36	-36	-40	-41	-39	-49	-49	-49

Tab. 6.2.: RSSI values measured in dBm

"The RSSI is the received signal power in milliwatts and is measured in dBm. The value can be used as a measurement of how well a receiver can "hear" a signal from a sender" [50]. The RSSI is expressed as a negative value and the closer it gets to 0, the better the signal is. Typical LoRa RSSI values are: -120 dBm as minimum which means that the signal is weak and -30 dBm which signifies that the signal is strong [50]. It is in addition important to mention that the scale is logarithmic which implies that differences between values are much more important than what they seem.

Accordingly, it is possible to observe that the differences between the antennas are relatively important especially when receiving packets from short distances. However, when the distances start to become larger, the differences are reduced. This is especially important since the LoRa technology has been designed for long range transmission.

The antenna that seems to have the best results is the biggest one⁴. Hence, this is the antenna that has been used for the *Gateway* during the final tests and demonstrations of the project.

6.4. Micropayment Evaluation

This section will start by comparing, from the point of view of the usability in the project, both micropayment solutions implemented and will then list some use-cases along with the best solution for each one accordingly to their advantages and disadvantages. Furthermore, a comparison of the transaction fees to send a micropayment is provided.

Micropayment channel

- Advantage:
 - Since the payments are done totally off-chain (without using any blockchain), they are instantaneous and can be sent through WebSockets.
- Disadvantages:
 - The *Server* is required to open a micropayment channel and lock ETH in advance for each gateway wanting to send a packet to it.
 - The *Server* has to deploy a new smart contract for each micropayment channel.

⁴<https://www.digikey.ch/product-detail/en/nearson-inc/S1551AH-868S/730-1067-ND/7320173>

- For both the *Gateway* and the *Server*, managing multiple micropayment channels can be painful especially if a lot of them are opened. In fact, the various smart contracts have to be monitored based on some thresholds and closed at the right time so that no money is lost on both sides.

OMG Network

- Advantage:
 - ETH can be locked only once by the *Server* for all the transactions with the gateways.
- Disadvantages:
 - OMG Network is considered as semi-decentralized because, even if everyone can run its own watcher to verify the security of the network, the child chain is run only by a single node managed by the OMG Foundation.
 - Due to the 2 minutes waiting period between each transaction sent from one account / wallet, compromises have to be made in the implementation. They include having to forward downstream packets without the payment being verified on the *Gateway* side and not having the ability to send automatic responses on the *Server* side.
 - Due to the way the plasma framework is implemented, there is a long waiting period to withdraw funds from the Ethereum layer 2 solution back to the main layer. This waiting period can go up to one week.

6.4.1. Use-cases

If a user receives messages from his devices at predetermined times and there are no urgent data to process and to answer to, the OMG Network solution could make sense. This could be the case for example for a farmer who wants to get the temperature of one of his field every 15 minutes.

On the other hand, if a user receives a lot of messages from different devices and these messages have to be processed instantly, the micropayment channel solution is more adequate. This could be the case for example when monitoring fire alarms in buildings.

6.4.2. Transaction fees

The transaction fees to send any amount of ETH on the Ethereum Mainnet at the time of writing (16 August 2021) are shown in Tab. 6.3. The price of 1 ETH is around 3200 USD. All three transaction fees assume a gas limit of 21'000 gas. The gas price in GWEI (1 GWEI = 0.000000001 ETH) for the slow solution is set to 40, for the average solution to 42 and for the fast solution to 49. The gas limit is the maximum amount of units of gas a user is willing to spend for a transaction and the gas price specifies the amount of ETH a user is willing to pay for each unit of gas.

	Gas limit	Gas price (GWEI)	Transaction fees (ETH)	Transaction fees (USD)
Slow	21'000	40	0.00084	2.76
Average	21'000	42	0.00088	2.89
Fast	21'000	49	0.00103	3.38

Tab. 6.3.: Transaction fees to send any amount of ETH on the Ethereum Mainnet

The transaction fees to send a micropayment using a micropayment channel have to be calculated by taking into consideration all the different steps involved with a transaction in a micropayment channel. A micropayment channel starts by the deployment of a smart contract on the *Server* side. This costs exactly 524'876 gas. Sending micropayments is then totally free so no gas are used. Closing the payment channel costs 46'397 gas to the *Gateway*. The addition of both gas fees equals 571'273 gas. If the same gas price is used as before, the total transaction fees and the fees for each participant of the transaction would be the followings:

		Gas limit	Gas price (GWEI)	Tx fees (ETH)	Tx fees (USD)
Slow	Server	524'876	40	0.02099	67.18
	Gateway	46'397	40	0.00186	5.94
	Total	571'273	40	0.02285	73.12
Average	Server	524'876	42	0.022045	70.54
	Gateway	46'397	42	0.001949	6.24
	Total	571'273	42	0.023993	76.78
Fast	Server	524'876	49	0.025719	82.30
	Gateway	46'397	49	0.002273	7.28
	Total	571'273	49	0.027992	89.58

Tab. 6.4.: Transaction (Tx) fees for micropayment channels

The transaction fees to send a micropayment using the OMG Network have to be calculated by taking into consideration all the different steps of a transaction in the OMG Network. This implies first depositing ETH on the OMG Network for the *Server*. Sending micropayments is not free and the fees should be considered. Finally, the fees for the *Server* and the *Gateway* to recover the funds on the Ethereum Mainnet should be taken into consideration. Depositing and withdrawing any amount on the OMG Network using the OMG Network Web Wallet⁵ requires a gas limit of 168'533 gas and costs 33 GWEI for the slow solution, 36 for the average solution and 39 for the fast one. The transaction fees to send a payment on the OMG Network is 0.07157 OMG tokens⁶ which equals to 0.00013 ETH (0,42 USD)⁷. The price of 1 OMG token is 5.88 USD at the time of writing (16 August 2021). The total transaction fees and the fees for each participant of the transaction would be the followings:

⁵<https://webwallet.mainnet.v1.omg.network/>

⁶<https://blockexplorer.mainnet.v1.omg.network/fees>

⁷<https://coinmarketcap.com/fr/converter/>

		OMG Network (ETH)	Micropayment (ETH)	Tx fees (ETH)	Tx fees (USD)
Slow	Server	2 * 0.00556	0.00013	0.01125	36
	Gateway	0.00556	0.00013	0.00556	17.80
	Total	3 * 0.00556	0.00013	0.01681	53.80
Average	Server	2 * 0.00607	0.00013	0.01226	39.24
	Gateway	0.00607	0.00013	0.00607	19.41
	Total	3 * 0.0061	0.00013	0.01833	58.66
Fast	Server	2 * 0.00657	0.00013	0.01327	42.48
	Gateway	0.00657	0.00013	0.00657	21.03
	Total	3 * 0.00657	0.00013	0.01985	63.51

Tab. 6.5.: Transaction (Tx) fees for the OMG Network

Of course, when using both micropayment solutions, it does not make any sense to send only one micropayment transaction since the interactions with the Ethereum main chain cost a lot of ETH. Accordingly, starting a micropayment channel is convenient when at least 27 transactions⁸ are exchanged between parties. For using the OMG Network, it is after at least 25 transactions⁹ that the solution becomes interesting. By taking the same example as previously of a farmer that receives LoRa packets every 15 minutes, both solutions are profitable in comparison to sending transactions with the Ethereum Mainnet after only a few hours. Unfortunately, even with the optimization of using layer 2 solutions, the fees of using Ethereum remain on the higher range in confront to the cost of the other options presented in section 2.3. However, it is important to mention that these fees are so high and prohibitive because the price of ETH is at an all-time-high and the consensus used by Ethereum is PoW. With the shift to a PoS consensus and the various optimizations that will come along, it is possible to imagine a drastic diminution in these fees.

Moreover, it is profitable to maximize the use of the micropayment solutions, especially micropayment channels, once they have been initialized. Therefore, the *Server* needs to try as much as possible to not open new payment channels with every gateway it receives packets from. This is why, the 10 seconds waiting time when receiving a LoRa packet not from the usual *Gateway* on the *Server* side, makes a lot of sense and has therefore been implemented.

6.5. Issues

During the implementation of the master thesis, different issues were encountered. On the software side, there were mainly due to the fact that blockchain development is relatively new and much more advanced in the JavaScript libraries than in the Python ones. Thus, there were some cases where a functionality has to be totally implemented in Python while in JavaScript, the functionality was already implemented and it is only necessary to use a provided library. This is why, at the time of selecting the programming language to implement the *Payment* service, JavaScript was selected without any doubt. The same could be said about COSE which does not get much attention in Python and this is why COSE_CounterSignature structures had to be implemented on top of the existing library. On the hardware side, there have been numerous issues with the devices to be run as the

⁸ $76.78/2.89 \approx 27$

⁹ $3 * 0.0061 + 0.00013 * X = 0.00088 * X$, with X is equal to the number of transactions to break-even
 $X \approx 25$

End Device. In fact, at the beginning, downstream LoRa packets were not received at all by the LoPy4. It is only after a countless amount of experiments and time that a solution was finally found. On the less brighter side, a will was to use a Adafruit LoRa Radio Bonnet¹⁰ connected through SPI on top of the Raspeberry Pi Zero W. Unfortunately, despite all the work and the researches done, it has been impossible to send raw LoRa packets with modulation parameters that are compatible with the LoRaWAN gateway. This is why, another solution had to be found which finally consisted in connecting the Raspberry Pi Zero W to the LoPy4 through serial.

6.6. Limitations

The master thesis presents a proof-of-concept architecture and implementation and thus should not be expected as a fully production solution. To try the softwares developed, it is imperative to respect the instructions provided and to use the hardware specified.

¹⁰<https://www.adafruit.com/product/4074>

7

Future Work

Since the subject is so vast, there exist several interesting opportunities to extend or improve the implementation. This include extensions on the LoRa-MAC protocol itself as well as on the *End Device* side or even on the Micropayment extension side:

- Using ephemeral keys and thus switching to Elliptic-curve Diffie-Hellman Ephemeral (ECDHE) instead of ECDH could result in an even higher level of security. It is possible to imagine a solution where the *End Device* and the *Server* would generate some new keys using their existing keys after a certain number of messages exchanged to ensure that an attacker could not have access to the entire communication even if he gets access to some of the keys. This would require more extensive key exchanges between both entities.
- Optionally, using symmetric MAC (Message Authentication Code) instead of digital signatures for producing shorter messages without non-repudiation.
- Developing a MicroPython library for the LoPy4 in order to be independent from the Raspberry Pi. This library should contain all the cryptographic functions needed as well as CBOR and COSE.
- Adding a logic (for example: Node-RED¹) to generate automatic downstream packets on the *Server* side. This would allow to analyze the packets received by the devices and create the payloads of the response downstream packets. It is also possible to imagine the development of an API to make things even more convenient to use.
- Changing the plasma solution selected from OMG Network to Polygon² previously called Matic Network which seems more promising to even further reduce the cost of using the Ethereum blockchain.
- Trying other layer 2 solutions to enhance the Ethereum layer 1 could result in some even more detailed comparisons between the various layer 2 solutions.
- Using a single smart contract for all the micropayment channels instead of deploying a new one each time, could result in a significant reduction of the transaction fees.

¹<https://nodered.org/>

²<https://polygon.technology/>

8

Conclusion

The project has shown how the use of asymmetric cryptography could be used to provide confidentiality and especially non-repudiation in addition to integrity and authentication already provided with symmetric cryptography. The decentralization of the LoRa infrastructure using the blockchain has been provided by creating a new LoRa-MAC protocol that depends on a smart contract deployed on the Rinkeby Test Network of Ethereum. To interact with the newly created protocol and smart contract, the needed softwares which permit an use of the new infrastructure with the standardized LoRa softwares on the existing hardware, have been developed. In addition, as it was one of the goal of the project, the proposed LoRa-MAC protocol implements the basic features of LoRaWAN as described in the thesis. A special focus as been followed during the entire project in order to have LoRa packets as short as possible without compromising the highest level of security. This is why, special encoding techniques in the name of CBOR and COSE have been used. Furthermore, it was important to show that by providing non-repudiation and decentralization, a new world of decentralized uses-cases is now open. To illustrate that, the project focused on bringing remuneration in crowd-sourced networks which would allow to anyone to deploy their own gateway and to get compensated for forwarding packets to the Internet. Transferring micropayments has been made possible thanks to the use of layer 2 solutions on Ethereum that helped bringing down the fees of transactions. Accordingly, two layer 2 solutions (plasma and micropayment channels) have been explored which have given the ability to compare the fees of the Ethereum main chain with the two solutions.

A

LoraResolver Smart Contract

```
1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >0.7.0 <0.9.0;
4
5 contract LoraResolver {
6
7     struct Device {
8         uint32 ipv4Addr;
9         uint128 ipv6Addr;
10        string domain;
11        uint16 ipv4Port;
12        uint16 ipv6Port;
13        uint16 domainPort;
14        address owner;
15        uint256 x_pub;
16        uint256 y_pub;
17    }
18
19    struct Server {
20        uint256 x_pub;
21        uint256 y_pub;
22        address owner;
23    }
24
25    mapping(uint64 => Device) public devices;
26
27    mapping(uint32 => Server) public ipv4Servers;
28    mapping(uint128 => Server) public ipv6Servers;
29    mapping(string => Server) public domainServers;
30
31    function registerIpv4Device(uint64 loraAddr, uint32 server, uint16 port, uint256
32        x_pub, uint256 y_pub) public {
33        require(devices[loraAddr].owner == address(0) || devices[loraAddr].owner == msg
34            .sender, "Device already owned");
35        require(loraAddr != 0, "loraAddr cannot be 0");
36        require(server != 0, "Server address cannot be 0");
37        require(port != 0, "Server port cannot be 0");
38        require(x_pub != 0, "x_pub cannot be 0");
39        require(y_pub != 0, "y_pub cannot be 0");
40        devices[loraAddr].ipv4Addr = server;
41        devices[loraAddr].ipv4Port = port;
42        devices[loraAddr].owner = msg.sender;
```

```
41     devices[loraAddr].x_pub = x_pub;
42     devices[loraAddr].y_pub = y_pub;
43 }
44
45 function registerIpv6Device(uint64 loraAddr, uint128 server, uint16 port, uint256
46     x_pub, uint256 y_pub) public {
47     require(devices[loraAddr].owner == address(0) || devices[loraAddr].owner == msg
48         .sender, "Device already owned");
49     require(loraAddr != 0, "loraAddr cannot be 0");
50     require(server != 0, "Server address cannot be 0");
51     require(port != 0, "Server port cannot be 0");
52     require(x_pub != 0, "x_pub cannot be 0");
53     require(y_pub != 0, "y_pub cannot be 0");
54     devices[loraAddr].ipv6Addr = server;
55     devices[loraAddr].ipv6Port = port;
56     devices[loraAddr].owner = msg.sender;
57     devices[loraAddr].x_pub = x_pub;
58     devices[loraAddr].y_pub = y_pub;
59 }
60
61 function registerDomainDevice(uint64 loraAddr, string memory domain, uint16 port,
62     uint256 x_pub, uint256 y_pub) public {
63     require(devices[loraAddr].owner == address(0) || devices[loraAddr].owner == msg
64         .sender, "Device already owned");
65     require(loraAddr != 0, "loraAddr cannot be 0");
66     require(keccak256(bytes(domain)) != keccak256(bytes("")), "Server domain cannot
67         be empty");
68     require(port != 0, "Server port cannot be 0");
69     require(x_pub != 0, "x_pub cannot be 0");
70     require(y_pub != 0, "y_pub cannot be 0");
71     devices[loraAddr].domain = domain;
72     devices[loraAddr].domainPort = port;
73     devices[loraAddr].owner = msg.sender;
74     devices[loraAddr].x_pub = x_pub;
75     devices[loraAddr].y_pub = y_pub;
76 }
77
78 function registerIpv4Server(uint32 ipv4Addr, uint256 x_pub, uint256 y_pub) public
79     {
80     require(ipv4Servers[ipv4Addr].owner == address(0) || ipv4Servers[ipv4Addr].
81         owner == msg.sender, "Server already owned");
82     require(ipv4Addr != 0, "ipv4Addr cannot be 0");
83     require(x_pub != 0, "x_pub cannot be 0");
84     require(y_pub != 0, "y_pub cannot be 0");
85     ipv4Servers[ipv4Addr].owner = msg.sender;
86     ipv4Servers[ipv4Addr].x_pub = x_pub;
87     ipv4Servers[ipv4Addr].y_pub = y_pub;
88 }
89
90 function registerIpv6Server(uint128 ipv6Addr, uint256 x_pub, uint256 y_pub) public
91     {
92     require(ipv6Servers[ipv6Addr].owner == address(0) || ipv6Servers[ipv6Addr].
93         owner == msg.sender, "Server already owned");
94     require(ipv6Addr != 0, "ipv6Addr cannot be 0");
95     require(x_pub != 0, "x_pub cannot be 0");
96     require(y_pub != 0, "y_pub cannot be 0");
97     ipv6Servers[ipv6Addr].owner = msg.sender;
```

```
90     ipv6Servers[ipv6Addr].x_pub = x_pub;
91     ipv6Servers[ipv6Addr].y_pub = y_pub;
92 }
93
94 function registerDomainServer(string memory domain, uint256 x_pub, uint256 y_pub)
95     public {
96     require(domainServers[domain].owner == address(0) || domainServers[domain].
97         owner == msg.sender, "Server already owned");
98     require(keccak256(bytes(domain)) != keccak256(bytes("")), "Server domain cannot
99         be empty");
100     require(x_pub != 0, "x_pub cannot be 0");
101     require(y_pub != 0, "y_pub cannot be 0");
102     domainServers[domain].owner = msg.sender;
103     domainServers[domain].x_pub = x_pub;
104     domainServers[domain].y_pub = y_pub;
105 }
```

List. A.1: LoraResolver smart contract

B

Application Server web site

Decentralized LoRa infrastructure using blockchain Devices Messages Down

Devices

[Reload](#) [Add device](#)

live_demo

0x46b6bc44a06627b7 [Send](#) [Modify](#)

x_pub : 23848c3497ea9ed5787888c6cd3bbfc3e62153e60b46b885c8b82b800744c513
y_pub : 3182ea477752e49032595fe08077e8b7a3800fad13b1b2f3a6b4c6eacc9f439

[Delete](#)

new

0x33366777a039afbf [Send](#) [Modify](#)

x_pub : 0c8564f11fc4b37877d33800b4457b56742c76c74c059216f7a92f45f09c2526
y_pub : 21db9fc8eb0f3c3c250a106b5cb1049b758061b93c9fb652dcb73289fa339335

[Delete](#)

démo

0x844bf9b98959d291 [Send](#) [Modify](#)

x_pub : d2605555b768324e3920c1e69d4ffe2bbe35c175b7fe8e6e7c8ffb5f4837ffb
y_pub : d209671644048be5d2d27f021a96f34613fcc44330f9eca3ba84bb096321912

[Delete](#)

Fig. B.1.: Devices page

Add a new device

Name :

deviceAdd* :

Public key* :

Fig. B.2.: Modal to add a new device

live_demo

0x46b6bc44a06627b7

Name :

Fig. B.3.: Modal to modify the name of an existing device

live_demo**0x46b6bc44a06627b7**

Message to send :

payload

Close

Send

Fig. B.4.: Modal to send a message to a device

Decentralized LoRa infrastructure using blockchain

Devices Messages Down

Messages

0 message to pay for

Reload

09-08-2021_09:38:06

hello from the device

DataConfirmedUp, 127, 0x46b6bc44a06627b7

Respond

Delete

09-08-2021_08:15:18

reconnect

DataConfirmedUp, 126, 0xb65138cf3e1d836f

Respond

Delete

09-08-2021_08:08:03

connected

DataConfirmedUp, 125, 0xb65138cf3e1d836f

Respond

Delete

Fig. B.5.: Messages page**0x46b6bc44a06627b7**

Message to send :

payload

Close

Send

Fig. B.6.: Modal to respond to a message

Decentralized LoRa infrastructure using blockchain Devices Messages Down

Down messages Reload

09-08-2021_09:34:18

hello from the server Modify Delete

0x46b6bc44a06627b7, sent : true

09-08-2021_08:04:34

hey Modify Delete

0xb65138cf3e1d836f, sent : true

29-07-2021_13:20:16

new test Modify Delete

0x844bf9b98959d291, sent : true

Fig. B.7.: Down page

0x46b6bc44a06627b7

Message to send :

hello from the server

Close

Modify

Fig. B.8.: Modal to modify the payload of a down message

C

Common Acronyms

ABI	Application Binary Interface
AES	Advanced Encryption Standard
API	Application Programming Interface
AppKey	Application Key
AS	Application Server
CBOR	Concise Binary Object Representation
CHF	Cryptographic Hash Function
CMAC	Cipher-based MAC
CoAP	Constrained Application Protocol
COSE	CBOR Object Signing and Encryption
CRC	Cyclic Redundancy Check
CSS	Cascading Style Sheet
DNS	Domain Name System
DSA	Digital Signature Algorithm
ECC	Elliptic-curve cryptography
ECDH	Elliptic-curve Diffie-Hellman
ECDHE	Elliptic-curve Diffie-Hellman Ephemeral
ECDSA	Elliptic Curve Digital Signature Algorithm
ENS	Ethereum Name Service
ETH	Ether
EVM	Ethereum Virtual Machine
FIFO	First-In, First-Out
FIPS	Federal Information Processing Standard
FNS	Forwarding Network Server
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
HNS	Home Network Server
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IV	Initialization Vector
JOSE	Javascript Object Signing and Encryption
JSON	JavaScript Object Notation

KDF	Key Derivation Function
KID	Key Identifier
LoRa	Long Range
LoRaWAN	Long Range Wide Area Network
LPWAN	Low Power Wide Area Network
MAC	Medium Access Control
MIC	Message Integrity Code
MoreVP	More Viable Plasma
NIST	National Institute of Standards and Technology
NSA	National Security Agency
NwkKey	Network Key
OMG	OmiseGO
OSI	Open Systems Interconnection
PHY	Physical
PoC	Proof of Coverage
PoG	Proof of Guarantee
PoS	Proof of Stake
PoW	Proof of Work
REST	Representational state transfer
RF	Radio Frequency
RPC	Remote Procedure Call
RSSI	Received Signal Strength Indication
SHA	Secure Hash Algorithms
SPI	Serial Peripheral Interface
TCP	Transmission Control Protocol
TTN	The Things Network
UDP	User Datagram Protocol
URL	Uniform Resource Locator
USB	Universal Serial Bus
USD	United States Dollar
VM	Virtual Machine

D

License of the Documentation

Copyright (c) 2021 Andrea Rar.

Permission is granted to copy, distribute and / or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

The GNU Free Documentation Licence can be read from the GNU.org web site [51].

References

- [1] A. Durand, P. Gremaud, and J. Pasquier, “Decentralized LPWAN infrastructure using blockchain and digital signatures. *Concurrency Computat Pract Exper.* 2019:e5352.” <https://doi.org/10.1002/cpe.5352> (accessed August 17, 2021). vi, 4, 5, 9, 59
- [2] D. Yaga, P. Mell, N. Roby, and K. Scarfone, “Blockchain Technology Overview. NISTIR 8202.” <https://doi.org/10.6028/NIST.IR.8202> (accessed August 18, 2021). 9
- [3] C. Bormann and P. Hoffman, “Concise Binary Object Representation (CBOR). RFC 8949, RFC Editor, December 2020.” <https://datatracker.ietf.org/doc/html/rfc8949> (accessed August 18, 2021). 13
- [4] J. Schaad, “CBOR Object Signing and Encryption (COSE). RFC 8152, RFC Editor, July 2017.” <https://datatracker.ietf.org/doc/html/rfc8152> (accessed August 18, 2021). 14, 15, 56
- [5] J. Schaad and E. R. Housley, “CBOR Object Signing and Encryption (COSE): Countersignatures. Internet-Draft draft-ietf-cose-countersign-05, 23 June 2021.” <https://datatracker.ietf.org/doc/html/draft-ietf-cose-countersign> (accessed August 18, 2021). viii, 15, 16, 56
- [6] Pycom, “Pycom Specsheets LoPy4.” https://docs.pycom.io/gitbook/assets/specsheets/Pycom_002_Specsheets_LoPy4_v2.pdf (accessed August 18, 2021). vi, 19, 20
- [7] J. Pasquier, A. Durand, and P. Gremaud, “Advanced Software Engineering course. University of Fribourg.” vii, 20, 23
- [8] pwolf88 and Joel Foster, “OMG Network, what we know so far, part 2. OMGpool, Medium, 12 Jun 2020.” <https://medium.com/omgpool/omg-network-what-we-know-so-far-part-2-6e9db557c165> (accessed August 18, 2021). 30, 31
- [9] D. R. Huber, “Scaling the Second Layer. 9 June 2020.” <https://www.bitcoinsuisse.com/research/decrypt/scaling-the-second-layer> (accessed August 18, 2021). 30
- [10] Cryptopedia Staff, “What Is OMG Network and How Does It Work? Cryptopedia, 8 March 2021.” <https://www.gemini.com/cryptopedia/what-is-omg-network> (accessed August 18, 2021). 30, 31

-
- [11] K. Sookocheff, “How Do Websockets Work? 4 April 2019.” <https://sookocheff.com/post/networking/how-do-websockets-work/> (accessed August 18, 2021). 33
- [12] eiki, “Explaining Ethereum Contract ABI & EVM Bytecode. Medium, 15 July 2019.” <https://medium.com/@eiki1212/explaining-ethereum-contract-abi-evm-bytecode-6afa6e917c3b> (accessed August 18, 2021). 40, 51
- [13] J. Song, R. Poovendran, J. Lee, and T. Iwata, “The AES-CMAC Algorithm. RFC 4493, RFC Editor, June 2006.” <https://datatracker.ietf.org/doc/html/rfc4493> (accessed August 31, 2021). 58

Referenced Web Resources

- [14] “LoRa web page from Semtech.” <https://lora-developers.semtech.com/library/tech-papers-and-guides/lora-and-lorawan/> (accessed August 18, 2021). vi, 3, 4
- [15] “LoRa web page from Wikipedia.” <https://en.wikipedia.org/wiki/LoRa> (accessed August 18, 2021). 3, 4
- [16] “LoRa Alliance web site.” <https://lora-alliance.org/> (accessed August 18, 2021). 5
- [17] “The Things Network (TTN) web page from Wikipedia.” https://de.wikipedia.org/wiki/The_Things_Network (accessed August 18, 2021). 5
- [18] “The Things Network (TTN) web site.” <https://www.thethingsnetwork.org/> (accessed August 18, 2021). 6
- [19] “Helium documentation.” <https://docs.helium.com/> (accessed August 18, 2021). 6
- [20] “Helium web site.” <https://www.helium.com> (accessed August 18, 2021). vii, 7
- [21] “Ethereum.org web site.” <https://ethereum.org/en/> (accessed August 18, 2021). 7
- [22] “Decentralization article on the Ethereum.org web site.” <https://ethereum.org/en/decentralization/> (accessed August 6, 2021). 7, 8
- [23] “Blockchain web page from Wikipedia.” <https://en.wikipedia.org/wiki/Blockchain> (accessed August 24, 2021). 9
- [24] “Ethereum web page from Wikipedia.” <https://en.wikipedia.org/wiki/Ethereum> (accessed August 18, 2021). 9, 27
- [25] “Elliptic-curve cryptography (ECC) web page from Wikipedia.” https://en.wikipedia.org/wiki/Elliptic-curve_cryptography (accessed August 18, 2021). 12
- [26] “Elliptic-curve Diffie-Hellman (ECDH) web page from Wikipedia.” https://en.wikipedia.org/wiki/Elliptic-curve_Diffie-Hellman (accessed August 18, 2021). 12
- [27] “HKDF web page from Wikipedia.” <https://en.wikipedia.org/wiki/HKDF> (accessed August 18, 2021). 12

- [28] “Advanced Encryption Standard (AES) web page from Wikipedia.” https://en.wikipedia.org/wiki/Advanced_Encryption_Standard (accessed August 18, 2021). 13
- [29] “Elliptic Curve Digital Signature Algorithm (ECDSA) web page from Wikipedia.” https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm (accessed August 18, 2021). 13
- [30] “Digital Signature Algorithm web page from Wikipedia.” https://en.wikipedia.org/wiki/Digital_Signature_Algorithm (accessed August 18, 2021). 13
- [31] “Secure Hash Algorithms (SHA) web page from Wikipedia.” https://en.wikipedia.org/wiki/Secure_Hash_Algorithms (accessed August 18, 2021). 13
- [32] “Cryptographic hash function web page from Wikipedia.” https://en.wikipedia.org/wiki/Cryptographic_hash_function (accessed August 18, 2021). 13
- [33] “CBOR web page from Wikipedia.” <https://en.wikipedia.org/wiki/CBOR> (accessed August 18, 2021). 13
- [34] “Pycose documentation.” <https://pycose.readthedocs.io/en/latest/> (accessed August 18, 2021). vi, 14, 15
- [35] “Github repository of the packet forwarder.” https://github.com/Lora-net/packet_forwarder (accessed August 18, 2021). vi, 17, 18, 19
- [36] “Pycom documentation.” <https://docs.pycom.io/gettingstarted/> (accessed August 18, 2021). 19
- [37] “MicroPython web page from Wikipedia.” <https://en.wikipedia.org/wiki/MicroPython> (accessed August 18, 2021). 20
- [38] “Asyncio documentation.” <https://docs.python.org/3/library/asyncio.html> (accessed August 18, 2021). 20
- [39] “Mozilla documentation about asynchronous programming in JavaScript.” <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous> (accessed August 18, 2021). 21
- [40] “React web page from Wikipedia.” [https://en.wikipedia.org/wiki/React_\(JavaScript_library\)](https://en.wikipedia.org/wiki/React_(JavaScript_library)) (accessed August 18, 2021). 21
- [41] “Scaling documentation on the Ethereum web site.” <https://ethereum.org/en/developers/docs/scaling/> (accessed August 18, 2021). 27, 28, 29, 30
- [42] “zkSync web site.” <https://zksync.io/faq/intro.html#zksync-in-comparison> (accessed August 18, 2021). vii, 28
- [43] “Solidity by Example web page on the Solidity documentation.” <https://docs.soliditylang.org/en/v0.5.3/solidity-by-example.html> (accessed August 18, 2021). 29, 52
- [44] “OMG Network documentation.” <https://docs.omg.network/> (accessed August 18, 2021). 31
- [45] “Github repository of the implementation.” <https://github.com/inefix/Decentralized-LoRa> (accessed August 18, 2021). 32
- [46] “Github repository of the modified pycose library.” <https://github.com/inefix/pycose> (accessed August 18, 2021). 33, 35, 39, 45, 56

-
- [47] “The best Web3 providers online. Ethereum StackExchange.” <https://ethereum.stackexchange.com/questions/56191/the-best-web3-providers-online/56195> (accessed August 18, 2021). 39
- [48] “ENS documentation.” <https://docs.ens.domains/> (accessed August 18, 2021). 41
- [49] “Github repository of the original pycose library.” <https://github.com/TimothyClaeys/pycose> (accessed August 18, 2021). 56
- [50] “RSSI section of the LoRa documentation.” <https://lora.readthedocs.io/en/latest/#rssi> (accessed August 18, 2021). 60
- [51] “Free Documentation Licence (GNU FDL).” <http://www.gnu.org/licenses/fdl.txt> (accessed August 18, 2021). 76