UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

Deployment and Analysis of a LoRaWAN Network

Supervisor

Renato Lo Cigno, Leonardo Maccari

Student

Alessandro Sartori

Academic Year 2018/19

Acknowledgments

I wish to dedicate all the effort put into this work to my parents, who have always been so attentive to my education.

Secondly, my gratitude goes to the closer relatives and their encouragements about my personal interests and growth.

Lastly, I wish to mention the continuous support received from all my friends, from those who I have known for a life, to those recently met here in Trento.

Contents

Abstract

LoRaWAN and its physical layer counterpart, LoRa, are relatively new protocols in the Low-Power/Wide-Area Networks (LPWAN) family, an expanding and fundamental sector of the Internet of Things (IoT) industry.

Currently, an IoT developer or researcher can chose among an incredibly vast range of possibilities spanning many performance and price ranges, both considering hardware platforms and software technologies. Communication protocols make no exception: every one of them provides different features at different computational and power costs, which can lead the choice to be far from trivial, especially when experimental data and real-world performance are rarely reported on scientific articles.

This however is justifiable, since new-born and evolving protocols like LoRa initially receive little attention from the most. Consequently, this thesis proposes to resume and discuss LoRa and LoRaWAN's characteristics by presenting them in their entirety. A deployment of an experimental network with commodity hardware will help illustrate their infrastructure and principles of working by describing, installing, and configuring each component of the system individually on a Raspberry Pi 3 and on two Pycom LoPy development boards. Thereafter, the previous technical discussion will be resumed in order to allow a thorough explanation of LoRa's performance capabilities and limitations, motivating facts with a presentation of how this innovative modulation operates, and with calculations applied to data derived from the official Semtech specification.

To go one step further and provide the reader with what LoRa is currently missing — real world data —, many scripts were developed to serve a number of experiments set in different locations and situations, leading to results that sometimes consistently diverged from theoretical expectations. Specifically, LoRa was found to be somewhat underachieving in urban environments: 1 km represented its transmission distance limit out of the 20 km officially believed. Rural settings, on the other hand, seemed much more promising even if exhaustive enough tests could not be put in place; a maximum distance of only 4 km, for instance, found a place in this work, but many other different and interesting experiments were instead included, such as penetrating a hill, embankments and walls. These last cases in fact, proved that LoRa is highly vulnerable to urban-like obstacles such as series of walls and buildings, but considerably less so to natural elements like trees, vegetation, or even masses of terrain spanning from small embankments to kilometer-wide hills.

One last section discusses security, assessing for every protocol component its possible vulnerabilities. As many other authors were found to agree, LoRaWAN exposes no obvious security flaw, provided that basic common-sense precautions are taken. Being a wireless protocol, however, makes LoRa inevitably susceptible to intentional jamming and other Denial-of-Service attacks, some of which are actually caused by frequent misimplementations of the standard.

Resuming all the analyzed characteristics and collected data, interesting applicability conclusions can be inferred, particularly on how performance can quickly degrade in improper environments and how these need to be carefully considered when planning a deployment that exceeds the hundreds of meters. Moreover, the slight unreliability of transmissions combined with a considerable vulnerability to DoS attacks make this protocol unsuitable for critical applications such as remote control of alarm systems. More feasible use cases could be remote reading of environmental or agricultural sensors (which could also take advantage of LoRa's minimal power consumption by freely running on batteries), or in general applications were a disruption of the service would not cause critical consequences.

1 LoRa and the Internet of Things

The Internet of Things (IoT) is among the roots of the revolution in communications that we are witnessing today, and LoRa, subject of this thesis, is just one of the innumerable protocols and technologies supporting its growth.

The goal of this thesis is to analyze the deployment and infrastructure of a LoRa network using two Pycom's LoPys transceivers. An overview of which technologies surround LoRa and their applications will follow in the succeeding paragraphs, after which physical characteristics and infrastructure layout will be presented in Chapter 2, while Chapter 4 will focus on throughput and range measurements. Chapter 3 dives into the hardware and software section of the experiment, documenting how the framework was set up and configured. Lastly, in Chapter 5 the overall security of the system is audited and compared to other authors' opinions and findings.

1.1 The Internet of Things

The Internet of Things tightly surrounds our lives, and it is what powers all those "smart objects" we encounter so often. Technically, "smart" is a prefix used to indicate objects embedded with processors and sensors, and arranged to exchange data with other machines or humans. This allows not only for enhanced interactions with them, but also the analysis of collectible data, offering solid advantages both for users and for companies.

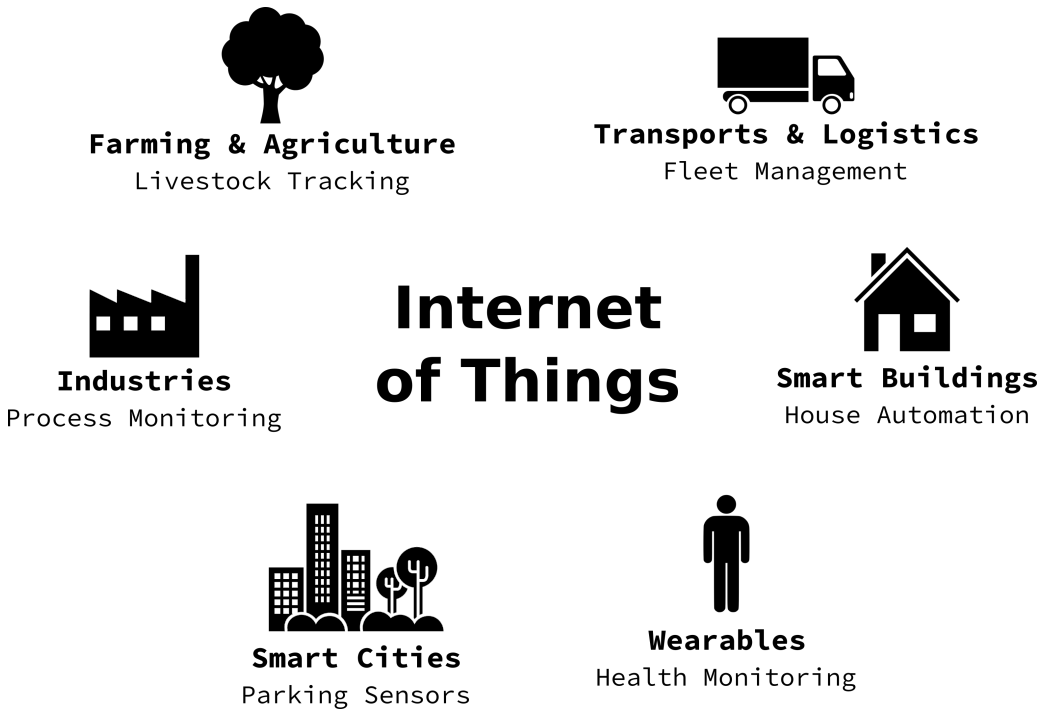


Figure 1.1: Example applications of the *Internet of Things*

Although this trend began to gain traction only in the late 2000s, it has actually been around since the early 70s as a concept, and since 1982 with the first ever connected device, a Coke machine. The actual birth of IoT, however, is estimated to have happened between 2008 and 2009, when the number of connected objects surpassed that of people [19].

At that point, many common-use objects started being equipped with Internet connections and remote control capabilities, letting people gradually coexist and appreciate this technology as part of their daily lives. With home automation, born in 2000 with the first smart-refrigerator, the growth of

IoT reached its peak, and major industries all over the world begun experimenting and investing in its potential to improve their businesses, increase production processes efficiency, and deliver enhanced customer services.

As an example of a prensent-day substantial deployment, Trenitalia, the primary Italian railway service provider, embedded its fleet of nearly 30,000 locomotives with a number of sensors aimed at preventing failures and maintenance downtime. While improving customer experience, this also set up a dynamic maintenance model: equipment such as motors, batteries, or brakes were no more replaced on the basis of scheduled times or ran kilometers, but on the basis of life-cycle models built from *Big Data*, boosting reliability and reducing costs of 8% to 10% [17].

In few years IoT even reached the interest of central companies such as Google, which is still researching to this day on futuristic topics like self-driving cars and advanced smart glasses (*Google Glass*).

1.2 Overview of Existing Technologies

The Internet of Things spans such a vast number of different use cases that in the course of its development a lot of protocols were born, covering many different communication ranges and providing various transmission data rates and versatility. Below, the most widespread standards are briefly presented, in order to let the reader picture what distinguishes LoRa from the other protocols and their ideal use cases.

As a side note, many of these technologies are classified as **Low Power Wide Area Networks (LPWAN)**, highlighting their key design characteristics:

- **Long Range**: IoT devices rarely have real-time needs. Hence, trading off high data rates for longer ranges is often convenient, especially in rural or urban applications.
- **Low Power**: low power requirements allow for longer operating times even on cheap battery packs. 10 or 20 years is not an unusual lifespan for an average LPWAN transceiver.
- **Low Cost**: affordable costs are reached through the use of license-free frequency bands and simplified hardware designs, thanks to lightweight protocols.

Moreover, unlike other common types of networks where down-link communications exceed up-link traffic, in LPWANs up-link messages are to dominate.

1.2.1 Comparison of Protocols

A Media Access Control (MAC) protocol, along with its physical carrier, is probably the most fundamental choice for an IoT project design. It will have the most critical impact on communication range, data rate, and ultimately on power consumption, an aspect that can become central for battery operated devices like remote sensors or actuators. In contrast to the ubiquitous ISO/OSI model though, in IoT the separation between layers is often blurry, and some of the following standards actually span the entire network stack:

- **WiFi** is easily the most obvious choice for a huge number of applications: its infrastructure has a pervasive presence, and the trade-off between data rate, range, and power consumption is nearly optimal.
- **Bluetooth and Bluetooth Low Energy (BLE)** are almost as ubiquitous as WiFi. BLE especially, is designed to absorb minimal amounts of power and deliver smaller chunks of data. These characteristics, plus a reduced range compared to WiFi, make Bluetooth and BLE ideal for wearable technology.
- **Near Field Communication (NFC)** is a short range (approx. 4cm) RF technology that enables simple and safe interactions between devices with a touch. Essentially, it extends the functionality of contactless cards to smartphones or RFID readers.

- **Cellular Networks (2G/3G/4G)**, at the price of a considerable power consumption, offer high data rates of up to 10 Mbps and long ranges (30 - 200 Km). Special implementations of these technologies aimed at the IoT market exist, and are known under the names of LTE-M and NB LTE-M.

- **ZigBee**, like Bluetooth and WiFi, has an already vast and present infrastructure, but it is often employed in more industrial settings. Despite the low power requirements, it offers high robustness, scalability, and security. Data rates are somewhat low (20 - 200 kbps), and the range of operation typically covers a home or building (100 m).

- **Z-Wave**, an RF communication technology, was primarily designed for small home-automation products such as lamps or sensors. Since it operates in a sub-1GHz band, it does not suffer from 2.4 GHz interference like WiFi, Bluetooth, or ZigBee. Furthermore, it is considered to be very scalable and low latency, with data rates up to 100 kbit/s.

- **LoRa** targets Wide Area Networks (WANs), offering low-troughput bi-directional connections scalable to millions of devices. Its long range capabilities (2 - 15 km) and low power requirements make this standard suitable for industrial, mobile and smart cities applications. Another important distinguishing feature is the use of ISM[1] frequency bands, which avoid the need to acquire a license to transmit.

- **Sigfox**, much like LoRa, provides low-troughput low-power wireless communications via ISM bands, but with a slightly longer range: 3 - 50 km. Moreover, Sigfox is a business operator; consequently, to deploy a network one needs to wait for their area to be covered instead of autonomously setting up their own infrastructure.

1.2.2 Additional Protocols: IPv6 Integration

Home automation and similar use cases made the integration with existing Internet connections a priority, and several standards operate in this field:

- **6LoWPAN** is a key IP-based technology. It offers encapsulation and IPv6 features thanks to the inclusion of this stack. Clearly, it is independent from frequency bands, physical layer and communication platform, such as Ethernet, Wi-Fi, or sub-1GHz ISM radio frequencies.

- **Thread** is a relatively new network protocol based on 6LoWPAN. It supports full mesh networks of up to 250 nodes with high levels of authentication and encryption.

1.2.3 Additional Protocols: Data Transfer

For complex systems, a solid way to represent, transfer and manage data is a essential. APIs require efficient designs, and communications may have authentication or encryption needs.

- **Message Queuing Telemetry Transport (MQTT)** enables an extremely lightweight publish/-subscribe messaging infrastructure, which fits very well in low bandwidth applications.

- **Extensible Messaging and Presence Protocol (XMPP)**, unlike MQTT, aims at real time communications, from instant messaging to voice and video calls.

- **Constrained Application Protocol (CoAP)** presents the peculiar feature of a RESTful design, allowing for easy integration with the web via HTTP. A client subscribes to a resource and receives push notifications.

[1] ISM bands are portions of the radio spectrum internationally reserved for non commercial usage: **I**ndustrial, **S**cientific, and **M**edical purposes.

2 LoRa & LoRaWAN

LoRa and **LoRaWAN** are, respectively, a wireless modulation technique and its Medium Access Control (MAC) layer. Their name comes from **Lo**ng **Ra**nge, and they are relatively new protocols in the LPWAN family: in 2012 LoRa made its appearance as a proprietary technology by Semtech, while LoRaWAN followed in 2014 as an open standard, released by the LoRa Alliance. The latter, though, is only one of several stacks that can use LoRa as a basis and provide MAC layer features such as addressing, message integrity checks, encryption/decryption, and support for additional layers.

2.1 LoRa PHY

Being a license-free protocol, LoRa operates in the sub-1GHz ISM frequencies, a set of radio bands reserved for scientific or other non-commercial purposes. The exact center frequency and bandwidth, however, vary for each country depending on its specific regulations; these working frequencies are 868 MHz for Europe and 915 MHz for North America, while supported bandwidths are 125 kHz, 250 kHz and, only for North America, 500 kHz.

LoRa's radio modulation takes advantage of a particular Spread Spectrum technology called **Chirp Spread Spectrum (CSS)**, which allows a significant sensitivity of up to 20 dB below noise level. Transmissions can occur on 6 "virtual" channels created using as many *spread factors* (sweep rate of the chirp), from `SF7` to `SF12`. Thanks to the Spread Spectrum modulation, these 6 channels are orthogonal and do not interfere with one another if transmissions overlap, therefore allowing for simultaneous processing of messages by transceivers.

Achievable data rates and transmission ranges depend on all of the parameters presented above, and will be measured and discussed in Chapter 4.

2.2 LoRaWAN

Given the large communication range achievable by LoRa, a convenient network configuration is indeed a star topology, or, more precisely, a star-of-stars topology. **End Nodes (ED)**, like LoRa sensors and such, are represented by the leaves of the three, and transmit directly to AC-powered gateways. These **Base Stations (BS)**, in turn, are linked to **Network Servers (NS)** which coordinate their transmissions and forward packets to **Application Servers (AS)**, thus forming the "bigger star". Application Servers are the other endpoint of the infrastructure, where the actual user applications reside and are possibly accessed by APIs to provide external services.

In order to provide developers with the choice of trade-offs between power consumption and message delivery time, LoRa Alliance defined three classes of end nodes:

- **Class A** is the most power efficient. After each up-link transmission, the end node opens two short down-link receive windows in which the gateway can communicate with it. The obvious drawback is that if a message needs to be delivered from the gateway to the end node, the gateway needs to keep it queued until the next up-link frame is received.
- **Class B** devices open receive windows at regular times: every 128 seconds all the BSs send a beacon signal at the same time and assign a time-slot to every ED, in which these will come up and listen for down-link traffic.
- **Class C** is nearly continuously open to down-link messages, except when transmitting. It has the most power absorption, but also the lowest BS-to-ED latency, making it suitable primarily for AC-powered applications.

Class B, however, is sometimes not implemented, as only classes A and C are mandatory.

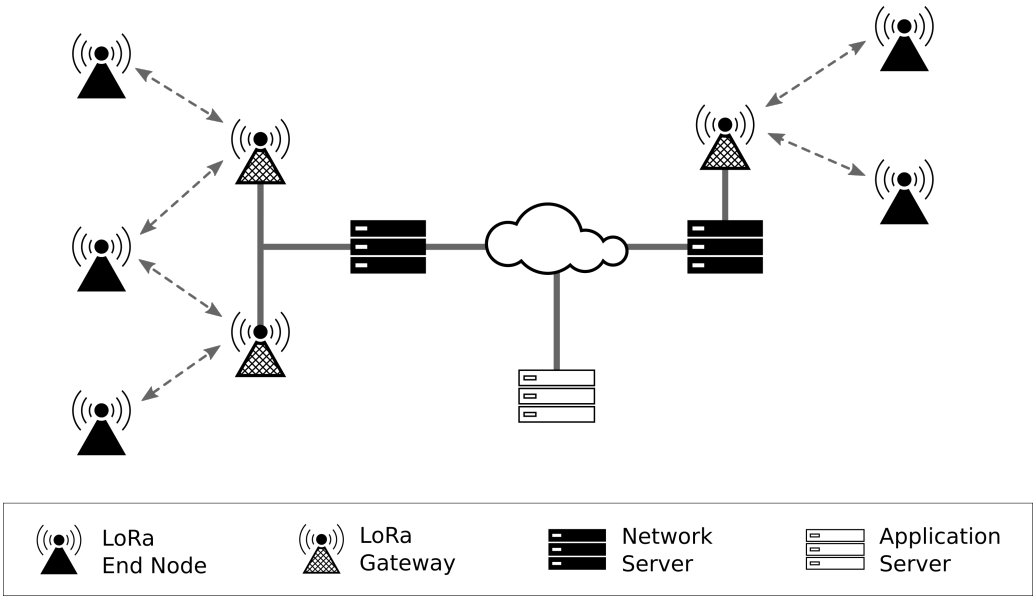


Figure 2.1: Example infrastructure of a LoRaWAN network, highlighting the ”star-of-stars” topology. Note how, as shown on the left side, one device might be in reach of multiple gateways.

2.3 Device Activation

Since LoRa's radio modulation does not provide any security feature, LoRaWAN introduces what is called *Device Activation*, a process that lets Network Servers negotiate cryptographic keys with the end-devices before these are allowed to participate in the network. Although the latest LoRaWAN specification is version 1.1, this document will contemplate version 1.0.2, because of compatibility restrictions on the chosen hardware.

Activation can happen with two different procedures, **Activation By Personalization (ABP)** and **Over The Air Activation (OTAA)**: the former (also referred to as ”static activation”) requires the session keys to be explicitly memorized in the end nodes, while the latter (dynamic activation) derives the **Network Session Key** and **Application Session Key** from an **Application Key** that is sent through a Join Procedure and uniquely identifies every device.

Specifically, the Network Session Key (NwkSKey) ensures the integrity of messages exchanged between EDs and NSs with Message Integrity Codes (MICs), while the Application Session Key (AppSKey) is used to encrypt and decrypt the payload of these packets, granting a private session with the AS [18].

The mentioned Join Procedure with which these keys are generate simply consists in a Join-Request message containing the device's Application Key, its address (DevEUI), and a random DevNonce to prevent replay attacks. If this request is validated by the NS, it generates a pair of NwkSKey and AppSKey and sends back a Join-Accept message, with another DevNonce to let the end node generate its keys.

3 Deployment of an Experimental Network

In order to compare LoRaWAN's specification with its real world behavior, two LoRaWAN nodes were acquired by the laboratory, one to act as a gateway and the other as an end device. As shown in Chapter 2, a LoRaWAN network consists of several components, and their individual setup and configuration will be described below, after an overview of the chosen hardware.

3.1 Employed Hardware

3.1.1 LoRa Transceivers

Pycom's LoPy[1] was found to be a valid product for its costs and characteristics. Other than a LoRa interface (a Semtech SX1272 chipset), its core (a ubiquitous ESP32 SoC) supports numerous GPIO[2] pins and WiFi/BLE connectivity, all with ultra low power requirements: deep sleep mode can absorb as little as 25uA to 5uA, while normal operation requires between 35mA and 100mA of current [15]. Moreover, LoPys have native support for MicroPython (a Python variant for micro-controllers), which keeps the development of software easy and efficient. Lastly, thanks to an Expansion Board, their serial (UART) interface can be accessed via USB, and updating or running new scripts on the board is a matter of few clicks in PyMakr, a specific development extension made by Pycom to ease various interaction processes.

3.1.2 Network Backend

This network, being of experimental purpose, was expected to carry very little load, and configuring separate a server for each component seemed a useless complication. Instead, it was chosen to host them all on a single device, a Raspberry Pi 3. Its affordable costs and reduced form factor deliver unexpectedly good performances, with enough computing power to run each piece of software.

3.2 Software Installation and Configuration

The final structure of the network is therefore composed of three devices:

- One **LoPy** acting as a LoRa ED, asking to join the LoRaWAN network and then transmitting dummy packets with a sequence number in their payloads.
- One other **LoPy** acting as a LoRa BS, relaying the received packets to the NS.
- The **Raspberry Pi** hosting the Network and Application Servers.

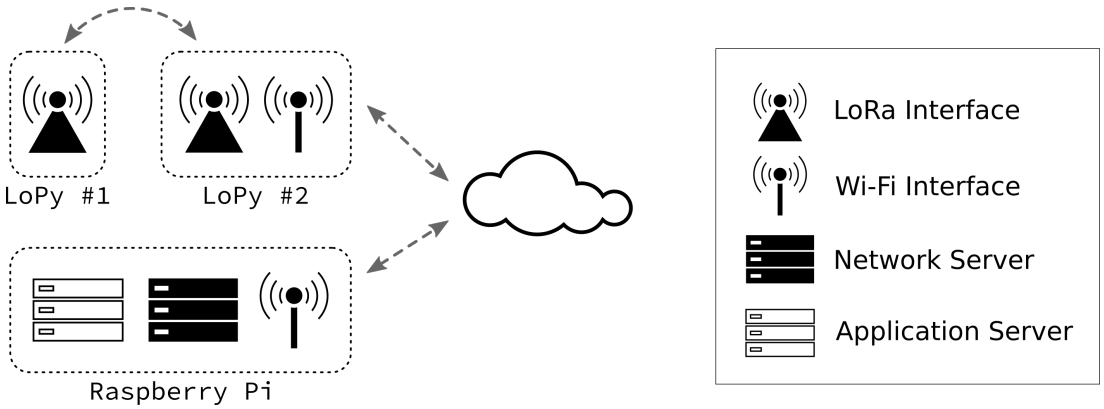


Figure 3.1: Schematic representation of the network, highlighting the separation of roles and their corresponding host devices.

3.2.1 LoRaWAN End Node

As anticipated, this device will simply send periodical join requests to the network and wait until one is accepted. At this point it will enter a `while` loop and symbolically send dummy packets with a delay between them. The actual code is listed in Appendix A.1, and was uploaded to the LoPy via the aforementioned development plugin PyMakr. As can be seen in lines 11 to 13, the join procedure requires three authentication parameters:

- `dev_eui` is the unique device address. It can either be the MAC address or a random hexadecimal string generated by the Application Server.

[1] https://docs.pycom.io/datasheets/development/lopy/
[2] General Purpose Input Output

- `app_eui`, or `join_eui`, is an unnecessary parameter only required when using an external Join Server, a scenario which is not contemplated by this work since handled by the AS. When this parameter is mandatory, as for the LoPy LoRaWAN library, it can be set to a dummy value like `0000000000000000`.
- `app_key` is the 128 bit key introduced in section 2.3 from which the security keys NwkSKey and AppSKey are derived. As for the `dev_eui`, it is generated by the AS when registering the device the first time.

Consequently, before uploading the code to the ED, it was necessary to conclude the installation process, register the device, and enter the generated keys in the code.

3.2.2 LoRaWAN Gateway / Packet Forwarder

A Base Station relays messages from those devices in its range to a specified Network Server and vice-versa (see Chapter 2). A basic implementation of a LoRaWAN Gateway (referred to as `packet-forwarder`, for its function) is provided by the makers of the development boards, Pycom, at their public GitHub Repository[3] under the name of `lorawan-nano-gateway`. While the complete code is split into three files (`main.py`, `config.py`, `nanogateway.py`), only a portion of `config.py` is shown and discussed below, because relevant to the network configuration:

```
1  # Address of the Network Server
2  SERVER = 'loraserver.local'
3  PORT = 1700
4
5  # WiFi parameters
6  WIFI_SSID = '...'
7  WIFI_PASS = '...'
8
9  # European frequency
10 LORA_FREQUENCY = 868100000
11 LORA_GW_DR = "SF7BW125" # DR_5
12 LORA_NODE_DR = 5
```

Listing 3.1: Relevant lines of the gateway configuration file

In order for the BS to communicate with the NS, it is needed to specify the address at which the server will respond and, of course, a network where this will be reachable (LoPys have a WiFi chipset but not a LAN interface). The last three lines are essential to set the right frequency, Spread Factor and Bandwidth on which the BS will listen on. Normally, Gateways are able to listen an all 8 channel simultaneously, but this is just an implementation adapted to a Pycom board which cannot, forcing us to operate on only one DR.

3.2.3 Network Backend

Raspberry Pis, being of widespread use, have been the focus of a project called *LoRa Server*, aiming to build an operating system that would aggregate every needed piece of software in a single image. Unfortunately, at the time of writing, this OS was not found to be stable enough, therefore raising the need to install and configure components one by one. On the positive side, this approach will ensure a thorough comprehension of the system by the reader, who will find the interaction between actors of the network to be more explicit.

Operating System

Among the vast variety of Linux distributions adapted to the ARM architecture, *Raspbian*[4] is the most favorited. It derives from Debian and is officially supported by the Raspberry Pi Foundation, plus it is available in different images equipped with more or less software packages, depending on one's

[3] https://github.com/pycom/pycom-libraries/tree/master/examples/lorawan-nano-gateway
[4] https://www.raspberrypi.org/downloads/raspbian/

needs. As this project requires very little prerequisites it was preferred to use a lightweight headless image, Raspbian Lite.

The command to flash the `.img` file to a microSD card (suppose `/dev/mmcblk0`) is straightforward, and takes advantage of `dd`'s block-by-block copying function:

```
dd if=raspbian-lite.img of=/dev/mmcblk0
```

To avoid the set up of a separate workstation with a keyboard and monitor to interact with the Raspberry Pi, its UART serial interface was used instead. The RPi 3 has two UART interfaces, but the primary one is used by the Bluetooth chipset; it is possible to either swap these interfaces or to redirect the system TTY, but the most simple solution is to disable the Bluetooth functionality since it will not be needed. This action can be carried out with a single command, after mounting the `boot` partition in a temporary folder:

```
mkdir tmp
sudo mount /dev/mmcblk0p1 tmp
sudo echo "dtoverlay=pi3-disable-bt" >> tmp/config.txt
```

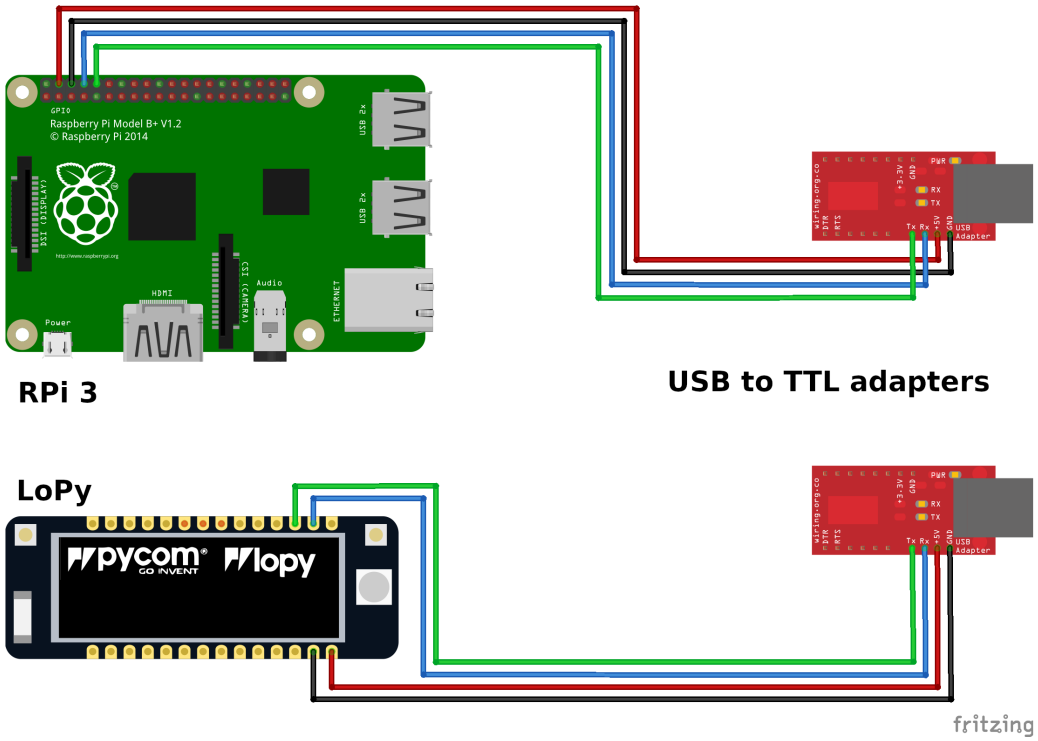


Figure 3.2: Wiring of the RPi and LoPy's serial interfaces

The Operating System was then ready to be booted and used. Connecting the USB/TTL adapter powers the device, and the system TTY is accessible via `screen` or other equivalent tools:

```
screen /dev/ttyUSB0 115200
```

Once access to the system terminal was gained, the following steps were executed in order to prepare the dependencies for the software to be later installed:

```
# Configure a network connection
sudo raspi-config

# Update the system
sudo apt-get update
sudo apt-get dist-upgrade

# Install dependencies needed later
sudo apt-get install redis-server postgresql mosquitto mosquitto-clients
   python3-pip
```

```
sudo pip install paho-mqtt

# Keep our work in a new directory
mkdir loraserver
cd loraserver
```

Gateway Bridge

A Gateway Bridge is a service that converts RF packets received from the Gateway's packet-forwarder to a format compatible with Network Servers, which in our case is JSON. Therefore, the interaction between Base Stations and Network Servers is actually mediated by this bridge, which encapsulates UDP packets into JSON messages and vice-versa; these JSON messages are then published on an MQTT channel, where Network Servers can securely exchange packets. As for the rest of components, the Gateway Bridge was available in a pre-compiled package for the ARMv7 architecture, letting us download and extract it instead of compile its source code. The MQTT broker, instead, was already installed as a dependency.

```
wget https://artifacts.loraserver.io/downloads/lora-gateway-bridge/lora-gateway-
    bridge_2.7.1_linux_armv7.tar.gz
tar -zxf lora-gateway-bridge_2.7.1_linux_armv7.tar.gz
```

Next, a default configuration was written to the proper file, ensuring that `udp_bind` was equal to `0.0.0.0:1700`, meaning every available NIC on the default port to which the packet-forwarder sends its data. Alternatively, the UDP server could have been bound to a specific interface address and different port, with care taken to always match it with the packet-forwarder's parameters.

```
./lora-gateway-bridge configfile > lora-gateway-bridge.toml
vim lora-gateway-bridge.toml
```

A test run of the script printed a number of log messages showing a successful connection to the MQTT broker, indicating a correct installation.

Network Server

Similarly to the Gateway Bridge, the Network Server only needed to be downloaded and extracted:

```
wget https://artifacts.loraserver.io/downloads/loraserver/loraserver_2.8.1
    _linux_armv7.tar.gz
tar -zxf loraserver_2.8.1_linux_armv7.tar.gz
```

This time, however, some extra work was necessary to correctly configure the application, as the NS and AS both need a Redis and PostgreSQL database to run. The Redis database did not need any set up, while the following instructions were issued to create a dedicated PostgreSQL user called `loraserver`, owner of a database named `loraserver_ns`.

```
sudo -u postgres psql # Opens the PostgreSQL console

# Inside the shell:
> create role loraserver with login password '...';
> create database loraserver_ns with owner loraserver;
> \c loraserver_ns # Change DB
> create extension pg_trgm;
> \q # Quit
```

Lastly, the server needed to be instructed on how to reach this table by updating the database connection string in the default configuration file, generated as before:

```
./loraserver configfile > loraserver.toml
vim loraserver.toml
```

In the text editor, the following attributes were filled in:

```
[postgresql]
dsn="postgres://loraserver:lorasecret@localhost/loraserver_ns"

[redis]
url="redis://localhost:6379"

[network_server.api]
bind="0.0.0.0:8000"
```

A test run of the script confirmed successful connections to the databases and the MQTT broker.

Application Server

Again, the installation was a matter of two trivial commands, followed by the creation of another dedicated PostgreSQL database named `loraserver_as`. A default configuration file was generated in the same manner and updated to match the parameters listed below.

```
# Install the Application Server
wget https://artifacts.loraserver.io/downloads/lora-app-server/lora-app-server_2
    .6.1_linux_armv7.tar.gz
tar -zxf lora-app-server_2.6.1_linux_armv7.tar.gz

# Create loraserveer_as
sudo -u postgres psql # Opens the PostgreSQL console

# Inside the shell:
> create database loraserver_as with owner loraserver;
> \c loraserver_as # Change DB
> create extension pg_trgm;
> \q # Quit

# Generate default configuration file
./lora-app-server configfile > lora-app-server.toml
vim lora-app-server.toml
```

```
[postgresql]
dsn="postgres://loraserver:lorasecret@localhost/loraserver_as"

[redis]
url="redis://localhost:6379"

[application_server.api]
bind="0.0.0.0:8001"

[application_server.external_api]
bind="0.0.0.0:8080"
jwt_secret="XUm85ptcR5ekeU4cLSpl5UXLeKOR1HxZlopZGEAjuVA="
```

The `jwt_secret` attribute is required to authenticate users accessing the web API, and was generated with OpenSSL: `openssl rand -base64 32`. A correct installation and configuration could be verified with a test run of the script, which exposes a web interface at the device's address on port `8080`, as per configuration section `[application_server.external_api]`.

3.3 Set Up of the Devices

Once all three services were running, the AS's interface could be reached on port `8080` through a browser. Figure 3.3 shows how it presented.

Network Server and Servcice-profile

First, a Network Server was added by clicking on "Network-servers" on the top left, then "Add", and filling in the requested information, i.e. a symbolic name and a `hostname:port` address. Care should

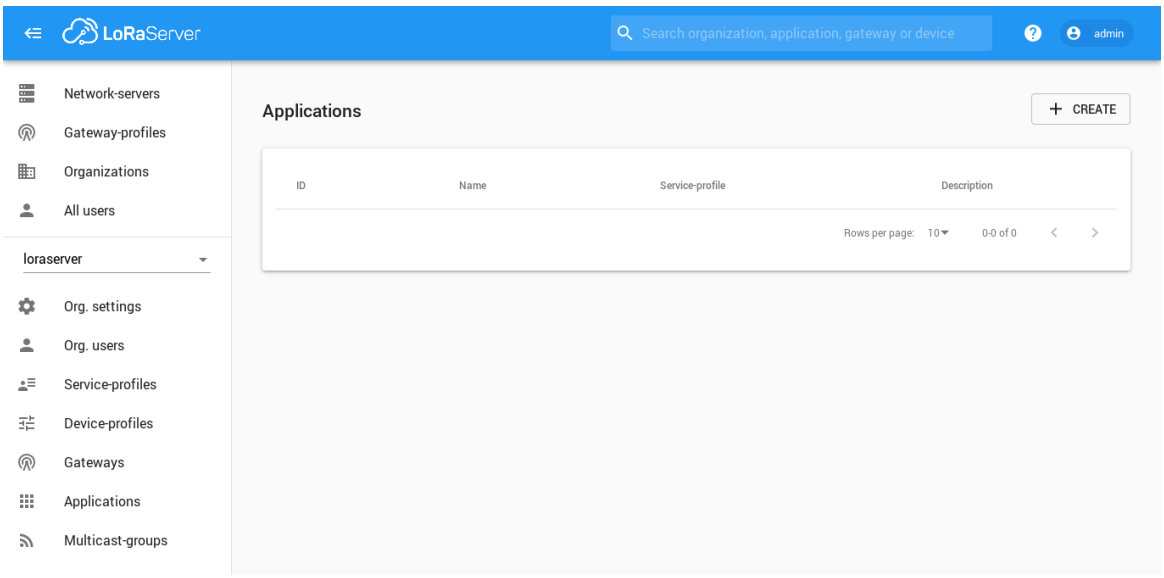


Figure 3.3: Home page of the LoRaServer web interface.

be taken to match the address and port with those entered in the NS's configuration file, which in our case were `localhost:8000`.

At this point an additional step was required before proceeding with the gateway. LoRa App Server implements what it defines "service-profiles", an abstraction of a contract between a user and the network, specifying which features are available to each organization. Hence, it was necessary to make the Network Server available to the default organization being used in this test environment, by adding a new service-profile associated to the newly created NS.

As before, it was a matter of filling in a form with trivial details, reachable from the relative left column button named "Service-profiles".

Gateway

In order to register the LoPy Gateway, its ID had to be retrieved first. Reading through the source files, the following lines were found to calculate this unique ID, and further down in the code a print statement indicated that it would be logged at start-up.

```
1 # Set  the Gateway ID to be the first 3 bytes of MAC
2 # address + 'FFFE' + last 3 bytes of MAC address
3 WIFI_MAC = ubinascii.hexlify(machine.unique_id()).upper()
4 GATEWAY_ID = WIFI_MAC[:6] + "FFFE" + WIFI_MAC[6:12]
5
6 [...]
7
8 self._log("Starting LoRaWAN nano gateway with id: {}", self.id)
```

In fact, starting up the device and monitoring its serial interface revealed the needed ID, and the Gateway could be registered with its proper identification code.

```
[     1.878] Starting LoRaWAN nano gateway with id: b'240AC4FFFE020984'
```

End Device, Device-profile, and Application

In order to let the LoPy End Device join the network, an "Application" had to be created to welcome the device's join requests. At this point, for convenience, a "device-profile" was created, aggregating all the boot features of the device to better manage same-brand nodes. One critical parameter during the registration of a node is the `Device EUI`, which represents a unique address. It can either be randomly generated, or it can be a node's MAC address itself.

3.3.1 Testing

To test the correct operation of the system, all three devices were powered on, and their serial output monitored for the confirmation that the device successfully joined and started publishing packets. Furthermore, the web interface started displaying statistics and live data from the node, visible in Figure 3.4.

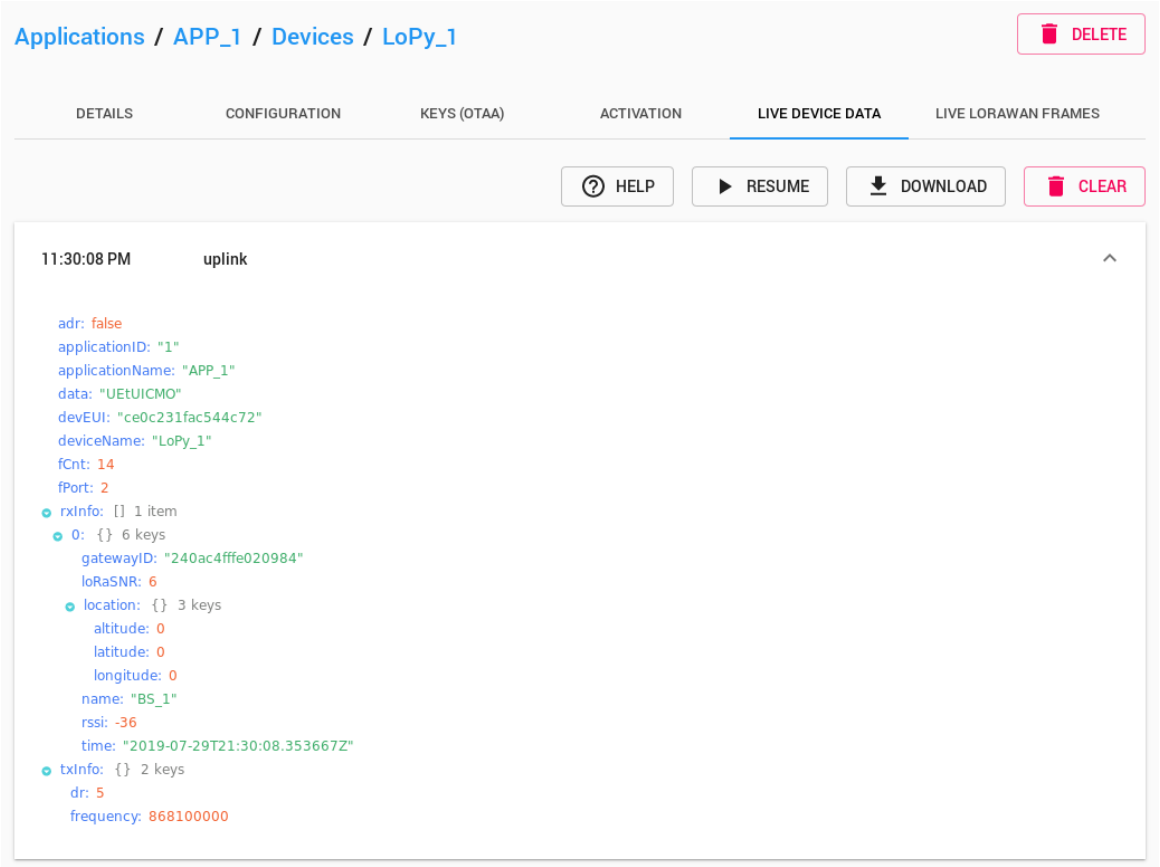


Figure 3.4: Live packet data from the test device.

4 Capacity Analysis

4.1 Achievable Throughput

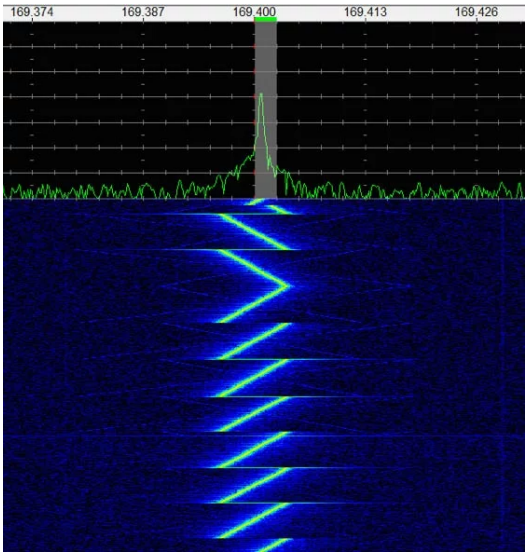
In Chapter 2, LoRa's modulation scheme was introduced, and will be here discussed in more depth in order to formally expound its bitrate and later compare these results with experimental tests. Although LoRa is a proprietary specification, a document by Semtech [7], was found to explain basic modulation concepts, and was here used to derive formulas to calculate a supposed throughput.

4.1.1 Spread Spectrum Modulation

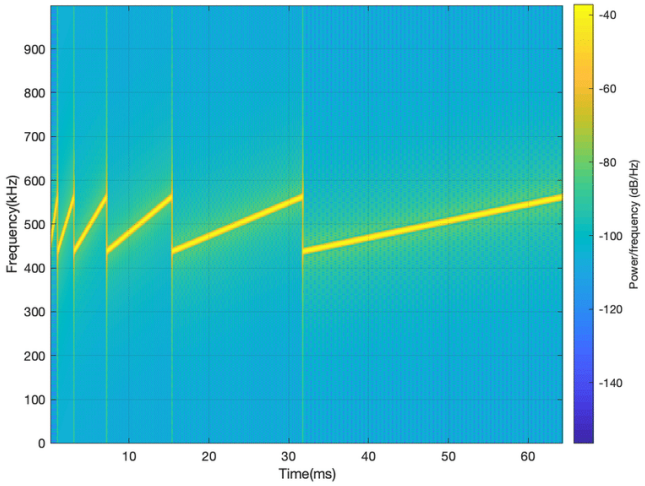
The fundamental concept of LoRa, which is also the principle of Spread Spectrum modulations, is that spreading the signal over a bandwidth much larger than the frequency content of the initial information gives a strong robustness against noise and narrowband interference, such as intentional channel jamming. Consequently, this high sensitivity allows for long ranges of transmission with minimal output power. Such a distribution of the signal can be accomplished via numerous techniques, for instance *FHSS (Frequency Hopping Spread Spectrum)* or *DSSS (Direct Sequence Spread Spectrum)*, where the transmitter continuously switches from one carrier frequency to another, following a pseudo-random sequence matched with the receiver.

The main drawback of these technologies, however, is that the transceivers need extremely accurate reference clocks in order to keep the devices synchronized, making the needed hardware often expensive. Continuous synchronization is also a problem for power-constrained devices, which cannot stay on all the time but need frequent and rapid synchronization mechanisms.

In order to overcome these issues, LoRa derives its modulation from **Chirp Spread Spectrum (CSS)**, in which the carrier frequency linearly sweeps from the lowest frequency of the band to the highest (up-chirps) or vice-versa (down-chirps). While still maintaining resistance to in-band interference, CSS also allows for quick synchronization of the devices, and ensures resistance towards multi-path fading and Doppler effect as well, making this technology ideal for mobile low-power applications. Figure 4.1a reports an example spectrogram analysis of a LoRa transmission, where the use of up-chirps and down-chirps is clear. Figure 4.1b instead, shows the relationship between the 6 different Spread Factors, which are fundamentally related to the sweep rate of the chirps, and directly influence the trade-off between data rate and channel sensitivity. Note, in particular, how this rate changes exponentially with base 2, e.g. `SF12` takes twice the air time of `SF11`.



(a) Waterfall diagram of LoRa modulation showing both up-chirps and down-chirps [12].



(b) Comparison of the 6 available Spreading Factors [9]. Notably, `SF7` (leftmost) requires much less air time than `SF12` (rightmost), therefore increasing the symbol rate but losing channel sensitivity.

Figure 4.1: Details of LoRa modulation: CSS chirps (left) and SFs (right).

Information Encoding and Bitrate

As for FHSS and DSSS, information is encoded with frequency hops; every T_s seconds, a symbol is encoded with a hop to one of the possible 2^{SF} bandwidth subdivisions, where SF is the *Spreading Factor* just discussed. As can be inferred, since a symbol has 2^{SF} different states, it basically carries SF bits of data. Its aforementioned period T_s, and its reciprocal, symbol rate R_s, can be calculated using the following formulas:

$$T_s = \frac{2^{SF}}{BW} \qquad\qquad R_s = \frac{BW}{2^{SF}}$$

Where BW is for bandwidth. The **modulation bitrate** R_b can now be determined as the symbol rate times the number of bits carried per symbol:

$$R_b = SF \cdot R_s = SF \cdot \frac{BW}{2^{SF}}$$

However, to further improve the robustness of the communication, LoRa includes in a packet a variable number of Forward Error Correction (FEC) bits, referred to as *Code Rate (CR)*. This parameter

varies from 1, where for every 4 bits of data 1 additional bit is added, to 4, where for every 4 bits of data 4 additional ones are calculated. Clearly, when computing the effective bitrate of a transmission, this overhead needs to be taken into account, reformulating R_b as:

$$R_b = SF \cdot R_s \cdot \frac{4}{4+CR} = SF \cdot \frac{BW}{2^{SF}} \cdot \frac{4}{4+CR}$$

Table 4.1 lists the 8 possible combinations of Spreading Factor and Bandwidth, referred to as *Data Rates (DR)*, that can be used by LoRaWAN. For each of them, modulation bitrate and nominal bitrate for each Coding Rate have been calculated.

Configuration				**Modulation**	**Nominal Bitrate** (bit/s)			
DR	**Modulation**	**SF**	**BW** (kHz)	**Bitrate** (bit/s)	**CR=1**	**CR=2**	**CR=3**	**CR=4**
0	LoRa	12	125	366	292	244	209	183
1	LoRa	11	125	671	537	447	383	335
2	LoRa	10	125	1,220	976	813	697	610
3	LoRa	9	125	2,197	1,757	1,468	1,255	1,098
4	LoRa	8	125	3,906	3,125	2,604	2,232	1,953
5	LoRa	7	125	6,835	5,468	4,557	3,906	3,417
6	LoRa	7	250	13,671	10,937	9,114	7,812	6,835
7	GFSK	-	150	-	-	-	-	-

Table 4.1: LoRa Data Rates with their configuration and respective bitrates rounded to their integer part. Note that DR 7 adopts a different modulation technique (*Gaussian Frequency Shift Keying, GFSK*) which won't be taken into account in this work, due to large differences in the underlying bitrate calculus.

In conclusion, at the physical layer, LoRa can transmit from a mere 183 bps in a long-range scenario (SF = 12, BW = 125 kHz, CR = 4), up to 11 kbps in a short-range situation (SF = 7, BW = 250, CR = 1).

4.1.2 Frame Format and MAC Layer Overhead

In order to assess LoRaWAN's performance at the application layer, the packet's format had to be taken into account, and is depicted in Figure 4.2 in an exploded view.

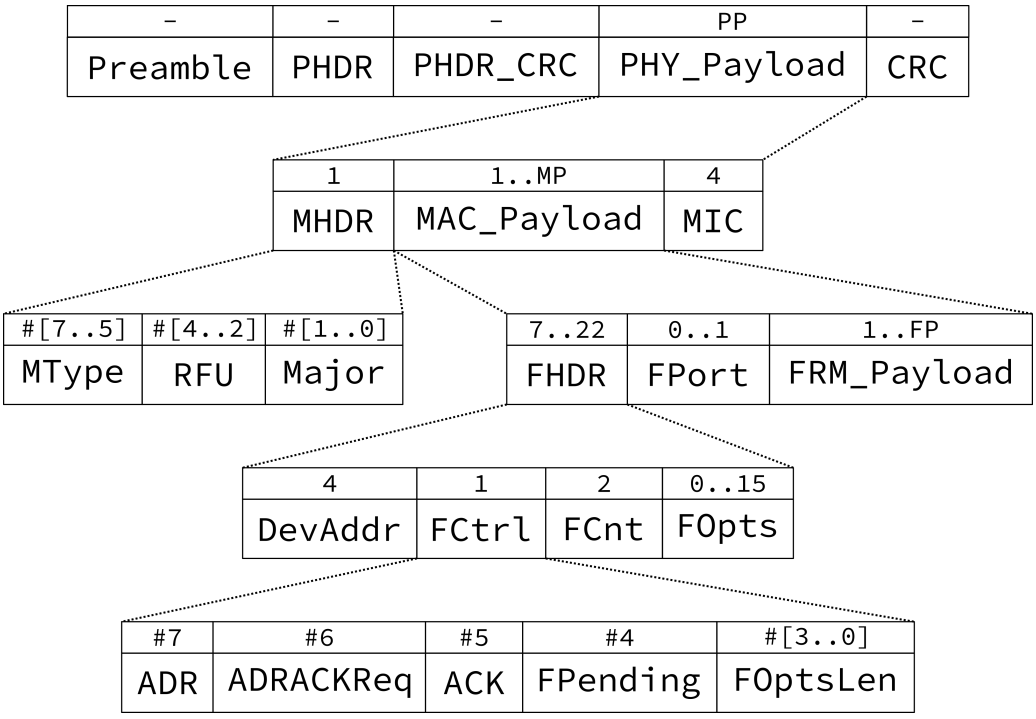


Figure 4.2: Breakdown of LoRaWAN's encapsulation stack.

Payload sizes are treated as variables because dependent on the used *Data Rate* and, possibly, on regional restrictions. For brevity and because out of scope, packets' fields won't be presented here, but LoRa Alliance's open documentation contains exhaustive descriptions [18].

For every regional band, [6] reports the maximum `MAC_Payload` (`MP`) size in relation to *Data Rates* (`DR`s), and was employed to infer the maximum `FRM_Payload` (`FP`) size in Europe by subtracting the overhead that headers can cause in two possible situations: the best case is when `FOpts` is empty, leaving room for 15 more octets, while the worst case is when this field is fully used to exchange MAC commands. For each `DR`, table 4.2 lists `MP`s, `FP`s, and the calculated throughput for every `CR`.

DR	**Size** (bytes)		**Overhead**	**Nominal Throughput** (bit/s)			
	MP	**FP**	(%)	**CR=1**	**CR=2**	**CR=3**	**CR=4**
0	59	36 - 51	13.6 - 38.9	178 - 252	149 - 210	127 - 180	111.2 - 152
1	59	36 - 51	13.6 - 38.9	328 - 464	273 - 386	234 - 331	205 - 288
2	59	36 - 51	13.6 - 38.9	596 - 843	497 - 702	426 - 602	373 - 520
3	123	100 - 115	6.5 - 18.7	1,429 - 1,642	1,194 - 1,373	1,020 - 1,174	893 - 1,024
4	250	227 - 242	3.2 - 9.2	2,838 - 3,025	2,365 - 2,521	2,026 - 2,161	1,774 - 1,888
5	250	227 - 242	3.2 - 9.2	4,965 - 5,293	4,138 - 4,411	3,546 - 3,781	3,102 - 3,304
6	250	227 - 242	3.2 - 9.2	9,928 - 10,584	8,275 - 8,824	7,094 - 7,562	6,206 - 6,616

Table 4.2: Application level throughput in relation to Data Rate and Coding Rate.

4.1.3 Verification with Experimental Data

In order to evaluate the reliability of the calculated data, effective datarates were measured by writing two Python scripts that sent a number of dummy packets using each `DR`, and estimated the bitrate by dividing the number of sent bytes by the time taken by the execution of the `send()` instruction. Although not very precise, this was the only method available from the platform API; the scripts used to gather performance data, the produced outputs, and how charts were generated, are fully included in Appendix A.2 and Appendix A.3, but a number of snippets will be here reported while explaining the experiment.

LoRa (PHY) Bitrate

For each of the seven *Data Rates*, the LoRa radio was initialized with relevant *Spreading Factor* and *Bandwidth*, to later enter a second loop that progressively updated the *Coding Rate* from 1 to 4. For each of these configurations, 10 "dummy" packets were sent and their execution time estimated. A packet consisted of a number of `0x00` bytes equal to its maximum payload size, .i.e. `MP bytes + 1 byte of MHDR + 4 bytes of MIC`. Listing 4.1 reports the main loop of the script, showing the actual procedure employed to quantify the bitrate.

```
1  for i, dr in enumerate(data_rates):
2      PKT_SIZE = dr[2]
3      print("\n[*] Setting up Data Rate DR{}: SF{}BW{} with payload of {}
           bytes".format(i, dr[0], [125, 250, 500][dr[1]], PKT_SIZE))
4
5      lora.sf(dr[0])
6      lora.bandwidth(dr[1])
7
8      for cr in [LoRa.CODING_4_5, LoRa.CODING_4_6, LoRa.CODING_4_7,
           LoRa.CODING_4_8]:
9          print("    > Coding Rate: {}".format(cr))
10         lora.coding_rate(cr)
11
12         s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
13         s.setblocking(True) # Set to blocking to measure execution time
14
15         times = []
```

```
        for i in range(PKT_N):
            t_start = time.ticks_us()
            s.send(bytes([0x00]*PKT_SIZE))
            times.append(time.ticks_diff(t_start, time.ticks_us())/1000000)

        _avg, _min, _max = (PKT_SIZE*8)/(sum(times)/PKT_N),
            (PKT_SIZE*8)/max(times), (PKT_SIZE*8)/min(times)
        print("      Throughput (bit/s):   Avg {:.2f}  -  Min {:.2f}  -  Max
            {:.2f}".format(_avg, _min, _max))
        s.close()
```

Listing 4.1: Main loop of the LoRa bitrate measurement script

The resulting figures are tabulated in 4.3a and plotted in Figure 4.3 along with the calculated bitrates. Considering the scale, deviation between measures can be considered almost negligible (especially for higher Data Rates) and imputable to software latencies. Moreover, as error bars show, variance is minimal and therefore the bitrate constant and somewhat stable.

DR	**Measured Bitrate** (bit/s)			
	CR=1	**CR=2**	**CR=3**	**CR=4**
0	183	159	140	125
1	327	283	249	222
2	729	633	560	501
3	1,505	1,279	1,115	988
4	2,869	2,415	2,085	1,837
5	5,042	4,245	3,675	3,233
6	9,953	8,396	7,261	6,388

(a) LoRa bitrate

DR	**Measured Throughput** (bit/s)			
	CR=1	**CR=2**	**CR=3**	**CR=4**
0	68			
1	81.6			
2	101.6			
3	229.6			
4	484			
5	484			
6	644.8			

(b) LoRaWAN throughput

Table 4.3: Experimentally measured data.

LoRaWAN (MAC/Application) Throughput

In order to perform the same evaluations at a higher level, the same strategies used for LoRa were adopted. This time however, the procedure required to use both devices, as the End Device being tested needed a Gateway with a network backend to connect to and exchange LoRaWAN frames; hence, a join procedure precedes the main loop of the script in order to let the device be part of this network.

Listing 4.2 reports this main loop, and as for the previous one, a nested loop that crafts and sends dummy packets can be seen.

```
for dr in data_rates:
    print("[*] Testing DR{}".format(dr[0]))
    PKT_SIZE = dr[1]

    s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
    s.setsockopt(socket.SOL_LORA, socket.SO_CONFIRMED, False)
    s.setsockopt(socket.SOL_LORA, socket.SO_DR, dr[0])
    s.setblocking(True)

    times = []
    for i in range(PKT_N):
        t_start = time.ticks_us()
        s.send(bytes([0x00]*PKT_SIZE))
        times.append(time.ticks_diff(t_start, time.ticks_us())/1000000)
        time.sleep(0.5)

```

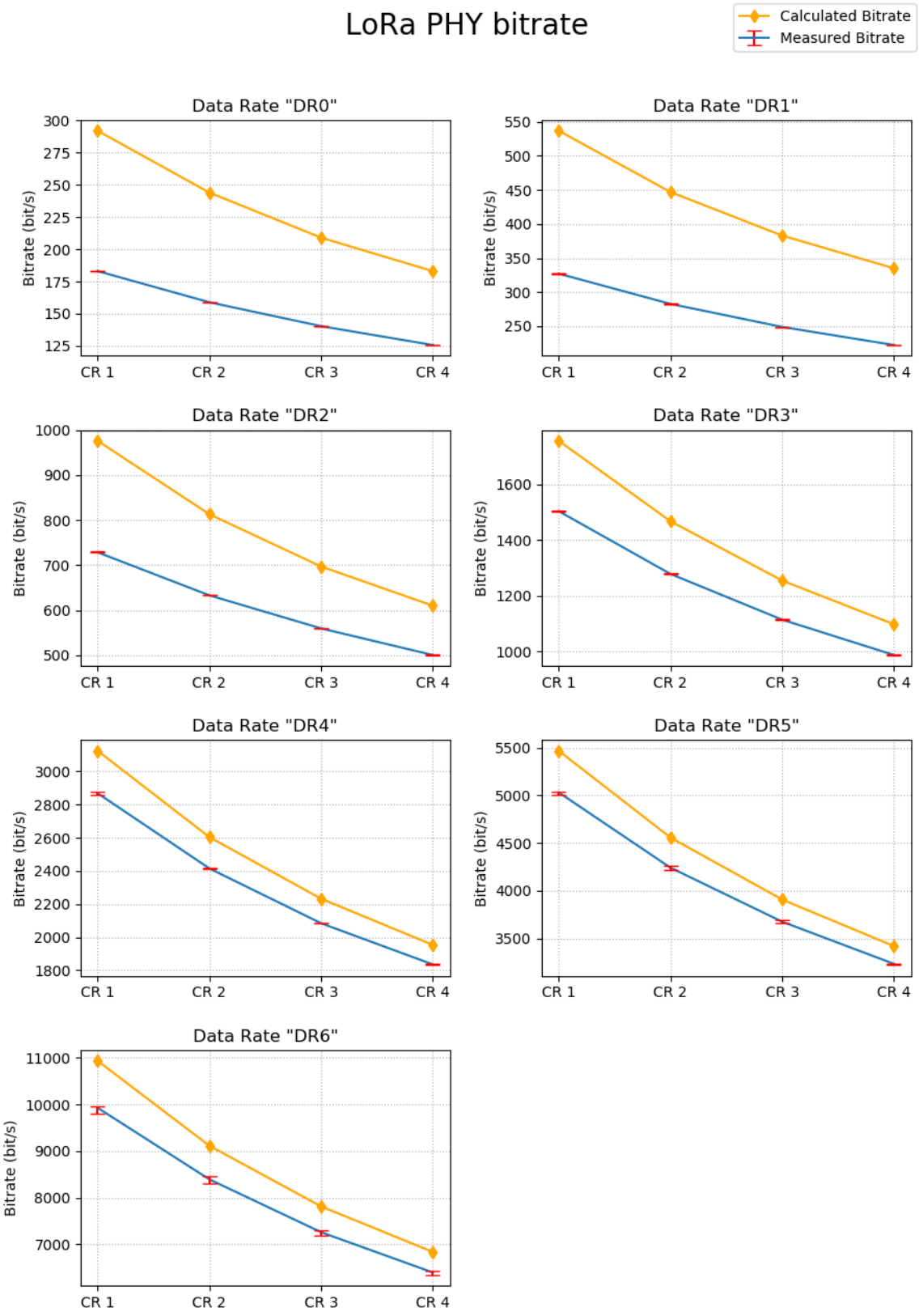


Figure 4.3: Comparison of calculated and measured LoRa bitrate for every DR and CR. Each graph additionally highlights how FEC overhead impacts performance when using higher CRs.

```
17      _avg, _min, _max = (PKT_SIZE*8)/(sum(times)/PKT_N),
            (PKT_SIZE*8)/max(times), (PKT_SIZE*8)/min(times)
18      print("      Min: {:.2f} bit/s  -  Max: {:.2f} bit/s  -  Avg: {:.2f}
            bit/s".format(
19          _min, _max, _avg
20      ))
21      s.close()
```

Listing 4.2: Main loop of the LoRaWAN throughput assessment script

However, this script doesn't enumerate the four Coding Rates, for the LoRaWAN stack internally overrides the CR radio setting that could be set as before. More specifically, network functions are available through sockets, which can only be created after initializing the **lora** class. This class directly interfaces with the radio, and among all the modulation parameters to configure, it can be set to either `LoRa.LORA` mode or `LoRa.LORAWAN` mode. Obviously, many parameters are contextual to the operating mode, meaning that for example, a LoRa raw socket will make no use of parameters such as `device_class` or `adr` (Adaptive Data Rate), as they are a LoRaWAN feature. Conversely, a LoRaWAN socket won't process raw radio parameters, like `power_mode` which schedules the sleep/listen cycle, because handled by the stack itself.
Pycom's firmware documentation [16], in fact, specifies exactly as said. For the `lora.coding_rate([coding_rate])` method, it states:

> Get or set the coding rate in **raw LoRa** mode (`LoRa.LORA`). The allowed values are: `LoRa.CODING_4_5` (1), `LoRa.CODING_4_6` (2), `LoRa.CODING_4_7` (3) and `LoRa.CODING_4_8` (4).

This method, as a matter of fact, is specifically affirmed to only work in raw LoRa mode, and no alternative is provided to operate when in LoRaWAN mode. As a result, any provided Coding Rate will be ignored when creating the network socket, or better, since this parameter won't be considered at all by the class, it can be set and read with no effect, therefore hiding the real in-use value. As a consequence of this misimplementation, resulting figures could not be attributed to any Coding Rate, and Table 4.3b thus lists them in rows with merged columns.

Unfortunately, this was not the only inconsistency found during the process, as Figure 4.4 illustrates: measurements were far from close to the calculations, despite numerous changes made to the environmental circumstances. Many settings were tried, both inside and outside of buildings in urban and rural areas. Distance between the devices seemed not to influence performance either, as well as weather or temperature. This fact however, let us conclude that this inconsistency was most certainly due to other factors, probably to firmware limitations or hardware malfunctions of the LoPys, hypothesis in fact supported by the insignificant variance of the measures.

One other factor lowering even more the actual reachable performance is the legal restriction of 1% Duty Cycle in the EU868MHz band. This tight restriction highly limits daily transmissible data, especially if considering that gateways undergo the same rule. For instance, during a day (`86,400` seconds), a gateway can transmit or receive on each channel, using theoretical datarates, at most

$$10,584\ (bit/s)\ \cdot\ 86,400\ (s)\ \cdot\ 1\%\ \approx\ 1\ \text{Mb}$$

with DR6 and CR1, or with DR0 and CR4, a mere

$$111.2\ (bit/s)\ \cdot\ 86,400\ (s)\ \cdot\ 1\%\ \approx\ 11.7\ \text{Kb}$$

As experimentally proven before, LoPys offer even poorer performances, allowing for daily quotas between 7 Kb and 68 Kb.

4.2 Radio Coverage

Official LoRa coverage is stated to be more than 10 km, sometimes approximated to 15 km, sometimes to 20 km. In order to shed more light on these vague declarations, a number of transmission tests were carried out, sending packets with different radio and environmental settings, and recording

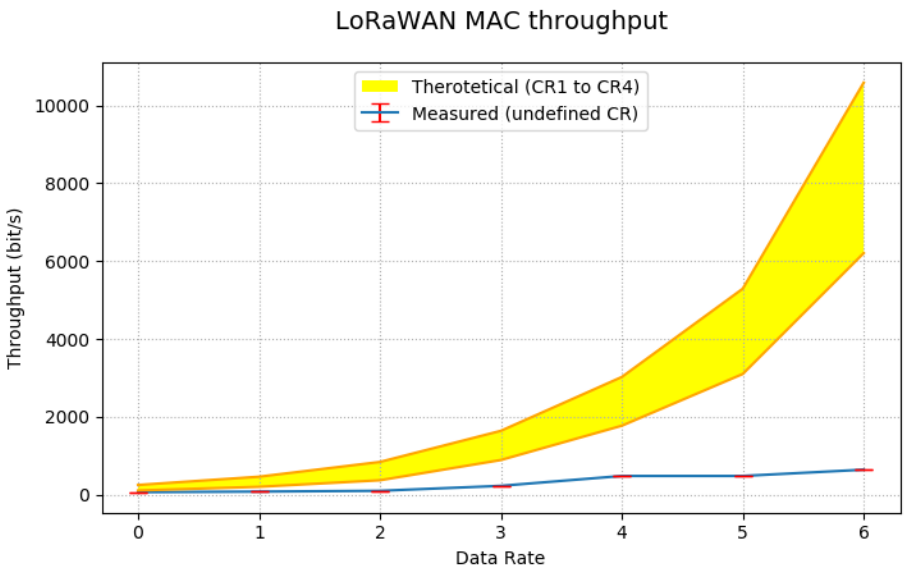


Figure 4.4: Comparison of calculated and measured MAC layer throughput.

corresponding reception rates. Two scripts were developed, one to run on a first LoPy acting as continuous transmitter, and the other to listen and record data on a second LoPy, positioned in progressively different conditions for every test; as usual, these scripts are reported in Appendix A.4. The low number of packets being sent for each DR/CR setting (10) is due to LoRa's prolonged transmission times, which if applied to more exhaustive experiments of 50 or 100 messages, would cause them to last too much time: at the current value of 10 packets per test, almost 300 transmissions are actuated, and consequently, reception rates are multiples of 10%.

4.2.1 Multipath Fading

Multipath fading occurs when signals reach a receiver via different paths due to reflections (i.e. walls) at different distances, and their relative magnitude and phases change, possibly interfering with one another and degrading the signal.

LoRa supposedly offers high resistance against this phenomenon. In order to test this claim, the two devices were placed far apart in an extended room (approx. 80 m^2), surrounded by walls irregular in shape and covered in many different materials including glass, wood and fabric surfaces. The outcome of the experiment is shown in Table 4.5 beside a representative map of the room, and numbers seem to confirm their claim.

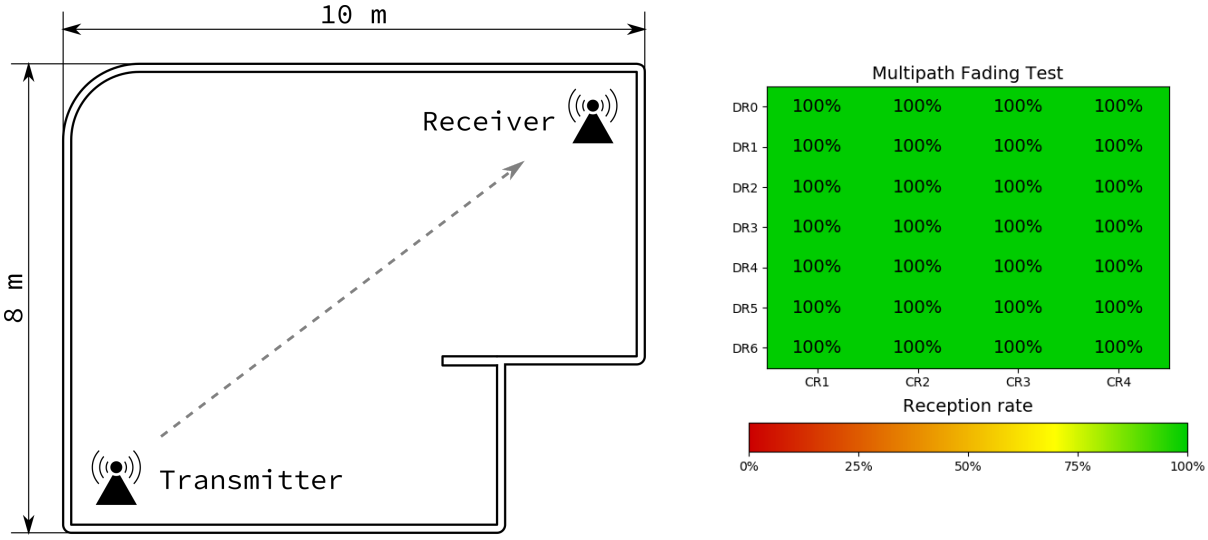


Figure 4.5: Reception rates for the multipath fading resistance tests.

4.2.2 Urban Obstacles

As urban applications would face many concrete obstacles, a set of transmissions was repeated multiple times while maintaining the distance between the devices as similar as possible, but varying the number of walls standing in between. Figure 4.6 illustrates the four studied situations. The first three are set on one location, while the last one is set on another. This is because the first three experiments had 100% reception rate for every case (apart from extremely sporadic packet losses that did non persist across repetitions), so one further test was planned with five walls, three of which were load-bearing walls (drawn thicker), where LoRa showed the first real losses on the least sensitive Data Rate, DR7.

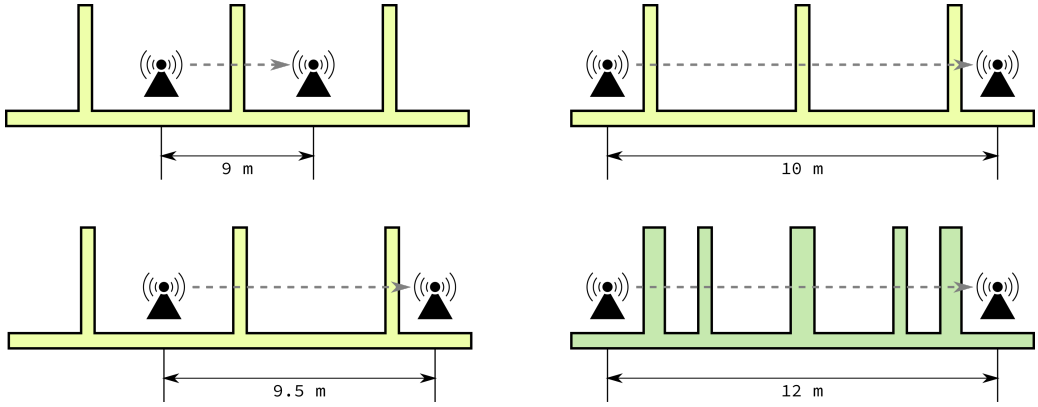


Figure 4.6: Relative placement of walls and transmitters during the "wall penetration" experiment.

Results are not tabulated because very repetitive: the only relevant information is a 10% to 20% packet loss when using DR7 with any Coding Rate.

In conclusion, LoRa demonstrated to be ideal at least for small deployments involving a contained number of buildings, such as school campuses or industrial areas.

4.2.3 Urban Range

Secondly, applicability in urban environments was evaluated with distance coverage tests. Figure 4.7 illustrates five locations where a receiver was placed to record data (blue pins) sent from a fixed receiver (red pin). Received packets were then compared to sent ones in order to discover their progressive reception rates, reported in Figure 4.8 as charts.

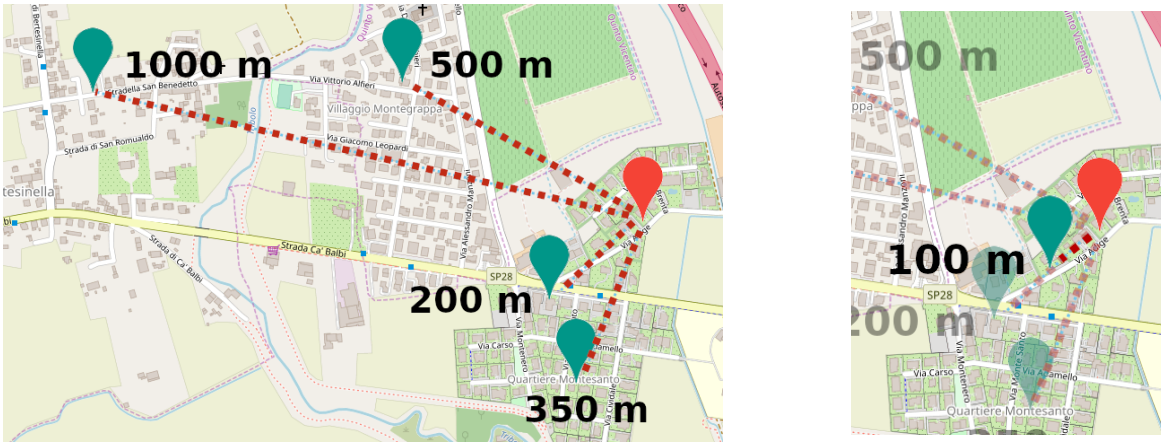


Figure 4.7: Map of the 5 locations where LoRa's coverage was tested.

Clearly, urban locations represent a tough environment for low-power radios such as LoRa, especially if using ISM frequency bands which can be contaminated by many other users.

Interestingly, the difference between 500m and 1000m is indeed more incisive than expected, and is probably attributable to interferences local to the place of measurement. This leads to the conclusion that even though LoRa is capable of far higher ranges, urban areas are filled with unexpected,

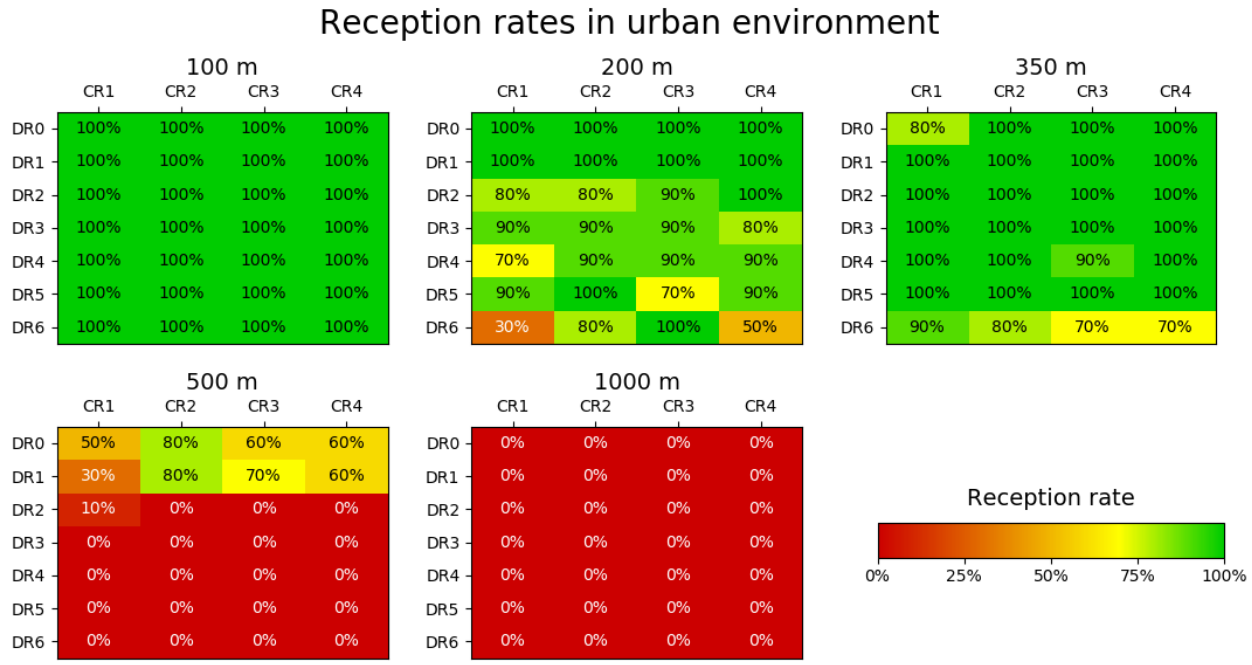


Figure 4.8: Reception rates of Figure 4.7's tests.

unpredictable, and possibly intermittent obstacles to radio transmissions. Thus, this phenomenon must be taken into account when planning such applications, likely requiring to invest in on-field tests to asses the feasibility of the project.

4.2.4 Rural Obstacles

Rural applications are likely to cross woods or other natural elements. In order to asses their impact on a typical deployment, two experiments were put in place.

The first of these aimed at simulating small obstacles such as contained irregularities of the landscape. Transmitting from an underground room to the garden above, and from another underground room to the end of an adjacent embankment (both illustrated in Figure 4.9), the impact of 5 and 10 meters of terrain was evaluated to be fairly negligible, as no packets were lost out of almost 300 sent, over various Data Rates and Coding Rates. These results are however not tabulated because repetitive and without valuable information.

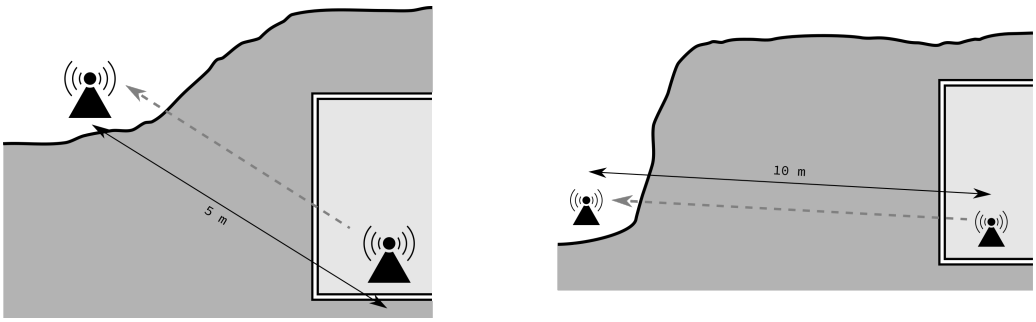
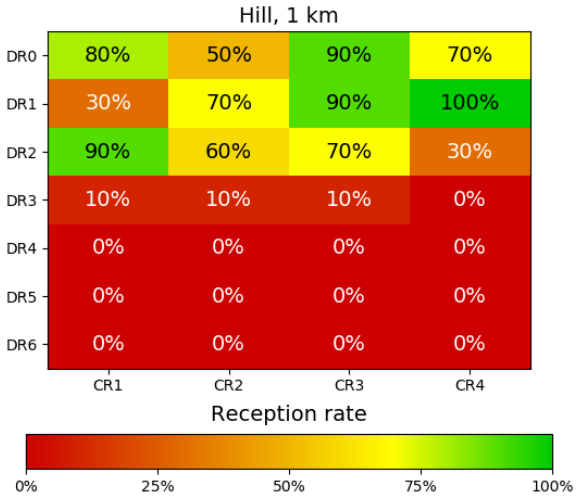


Figure 4.9: Representation of the two underground rooms used for the first "rural obstacles" test.

The second experiment consisted instead in a more concrete simulation, where a number of transmissions where sent over a real hill of little more than 1 km in width, considered a somewhat typical size. Figure 4.10a shows its location, and Figure 4.10b reports reception rates.

Results are overall promising and in accordance to expectations: consistent packet loss only occurred when using least sensitive Data Rates (DR3 to DR6), while more sensitive ones (DR0 to DR2) succeeded almost completely.

(a)



(b)

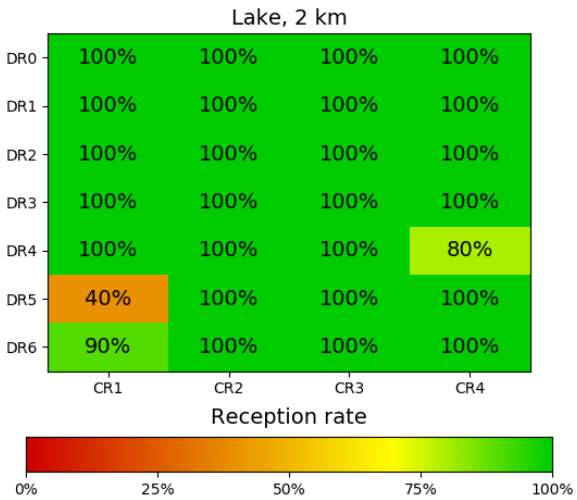
Figure 4.10: Experiment setting and results when crossing a 1 km hill.

4.2.5 Rural Range

After witnessing LoRa's limitations when operating in urban environment, and seeing the promising results of the "hill" experiment, radio coverage was tested directly on larger distances, 2 km and 4 km. The first test was set across a lake, depicted in Figure 4.11a beside the related results, while Figure 4.12a and 4.12b illustrate the 4 km experiment, which crosses mainly agricultural crops or minimally populated streets, with only some houses at the start of the transmission path. In this case results were a bit under expectations, but it is probably attributable to the low height at which transceivers could be placed: at about 1 meter above ground a lot of vegetation was in the way, i.e. crops, bushes, and trees.

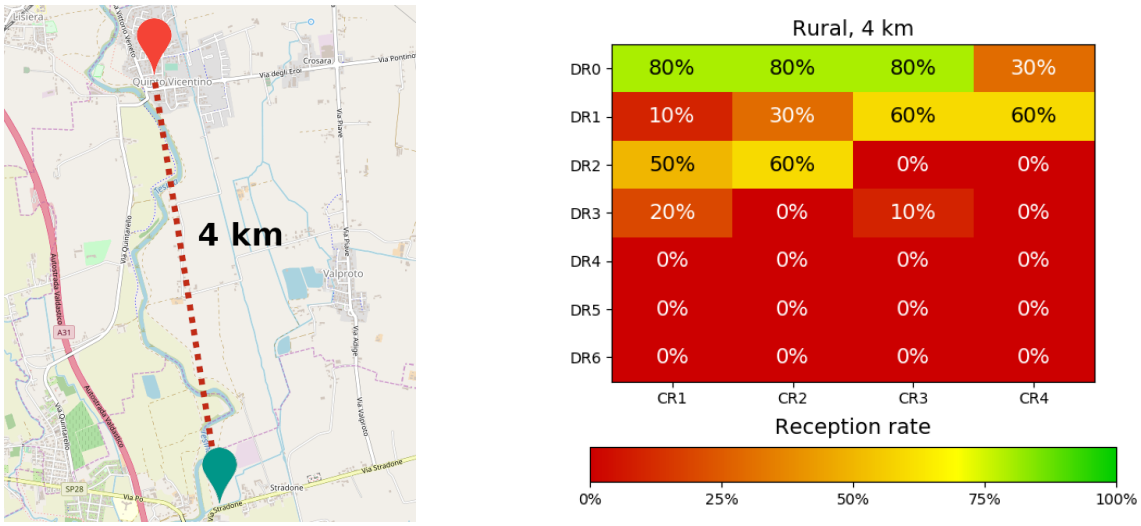
LoRa's performance is, in conclusion, demonstrated to be considerably better in less contaminated environments, were obstacles are contained in number and composed of more penetrable elements like dirt and wood. Reception rates decreased to quite small numbers during the 4 km experiment, but if using more sensitive Data Rates like DR0, LoRa could be still considered reliable enough to operate in non-critical applications.

(a)



(b)

Figure 4.11: "Lake" experiment setting and results.



(a) (b)

Figure 4.12: Transmission path and results when crossing 4 km of crops and some vegetation.

5 Security Considerations

After an overview of the objective capabilities of LoRa and LoRaWAN, it is proper to discuss their potential by also analyzing the level of security they can offer. LoRaWAN networks can be considered a worthwhile target for attackers for a number of reasons, from frequency bands being free of license, to the nature of the protocol itself, which due to its long range makes potentially many nodes reachable to a well positioned attacker.

Unfortunately, being provided with only two LoPy devices, none of the attacks could be tested and studied to report efficiency data, apart from the first (concerning physical tampering) for which experiments are currently undergoing.

5.1 Physical Tampering

A LoRa node is composed of a radio module that communicates with a Micro Controller Unit (MCU) to exchange MAC commands and data. Usually, this happens through a UART or SPI interface between the two, which don't currently provide built-in encryption features, therefore exposing the entire communication to malicious taps. Possible attacks include interception or manipulation of the payload, or worse, extraction of the encryption keys when exchanged during module initialization.

LoPy boards, specifically, feature an ESP32 MCU communicating with a Semtech SX1276 chipset through an SPI interface, dangerously exposed on the GPIO headers via pins GPIO5 (CLK), GPIO27 (MOSI) and GPIO19 (MISO), as Figure 5.1 reports from its datasheet [15].

5.2 LoRa PHY

LoRa's physical layer offers no cryptography features, apart from its modulation's inherent complexity to be demodulated by a random radio eavesdropper, which is anyway unlikely to notice any ongoing LoRa transmission given the noise-like shape of the signal. These characteristics, however, are no coincidence: CSS is in fact of military origins, and therefore specifically designed to appear as indistinct noise, besides being resistant to interference as discussed earlier. In conclusion, as for the majority of PHY layer protocols, encryption and authentication are expected to be implemented in higher layers.

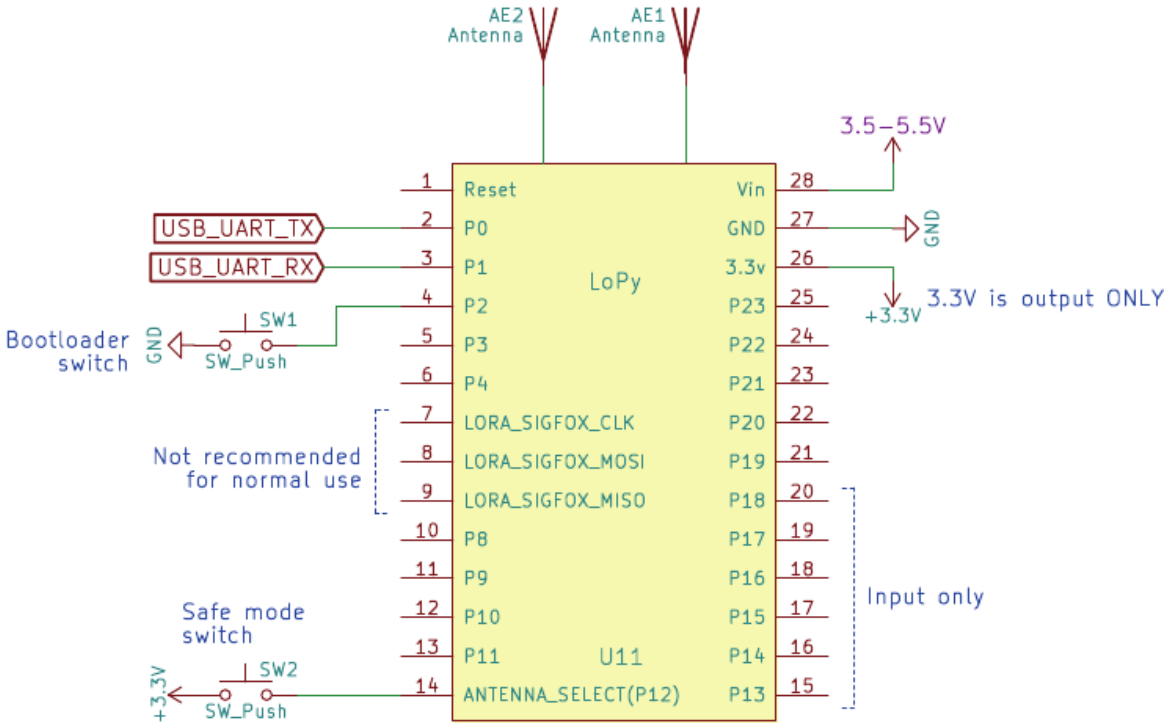


Figure 5.1: LoPy electrical schematic, where pins 7 to 9 show the exposed SPI interface.

5.3 LoRaWAN MAC

LoRaWAN's specification thoroughly explains all the encryption procedures and key material excange processes, which, provided that basic security precautions are adopted, are considered to be sound by many authors [14, 4]. However, these same authors agree on various vulnerabilities to Denial of Service (DoS) attacks, from obvious radio jamming techniques to more elaborated message replays, all reported below from [20].

5.3.1 Triggered and Selective RF Jamming

The slow modulation of LoRa causes long packet air times, which in turn give attackers the time to arrange sophisticated attacks like triggered or selective jamming. In triggered jamming, a jammer device is only activated when an ongoing LoRa transmission is detected; an experiment in [2] reported successful results of 3 packets received out of 600 (0.5%).

This vulnerability provides a good base for more advanced techniques like selective jamming, where a malicious and well-timed device is able to jam a channel during specific messages, rendering the attack very efficient and hard to detect, since to a network administrator it will appear just like a seldom loss of packets of some devices, instead of an extended down of many channels and nodes. LoRaWAN is vulnerable to this attack because of the unencrypted packet headers, which allow an attacker to read message information like device address or message type, and act on the basis of a comparison with a jamming policy.

5.3.2 Selective Jamming for a Wormhole Attack

A device capable to enact the previous attack can be combined with another transceiver to perform a Wormhole attack, where one device (close to the destination) selectively jams packets, while the other ("sniffer", close to the source) records them. These can then be replayed in a different part of the network, usually creating false routing information or routing loops to waste the energy of a mesh network.

In the case of LoRaWAN, given the absence of timing restrictions, packets can be replayed at any time, provided that no packet with a higher Frame Counter has reached the gateway (which would then cause the recorded packet to be rejected for having been replayed). This allows for instance to hide a sensor's state change, by jamming the new packets while replaying previously recorded normal-state ones [3].

5.3.3 Downlink Routing Vulnerability

Making use of the previous wormhole architecture, false node location information can be created and used to exclude it from the network.

Basically, when a Network Server needs to communicate with an end device, it knows which Gateway to utilize by searching its *Downlink Routing Path Database*, where every device is associated to the last Gateway which had it in range. A wormhole attack can be enacted to fake a node's position to be near another BS out of its range, so that next downlink messages to this node will be transmitted from this last Gateway and lost.

5.3.4 Join-Request Message Replay Attack

During the Join Procedure described in Section 2.3, it was mentioned that session keys on the Network Server are generated upon reception of a Join-Request packet, protected against replay attacks by a random DevNonce, of which NSs should keep history in order to spot reused ones. Unfortunately, implementation errors and performance compromises are numerous and part of the majority of firmwares.

For instance, an inadequate server may keep only track of the last 10 DevNonces for every device, leading to a vulnerability when a node authenticates and 11th time: if the first join-request packet is sniffed, recorded and replayed after it gets deleted from the server's history, it effectively fakes a Join Request, forcing the NS to refresh its keys and putting the End Node in a Denial-of-Service state, since its packets won't be recognized anymore by the network [10].

5.3.5 Join-Accept Message Replay Attack

Join-Accept messages do not contain any reference to the Join-Request they were generated for, and therefore suffer from replayability as well. Similarly to the previous attack, an older packet (but a Join-Accept in this case) is replayed during a later OTAA authentication, before the real Join-Accept reaches the requesting node, letting it create a security context that won't coincide with the server's, putting it once again in a DoS state.

6 Conclusions

After comparing LoRa's theoretical expectations to its real world behavior with numerous experiments, the idea of what can and cannot be achieved with it is now made much clearer.

From the throughput point of view, many applications are immediately ruled out: video streaming requires at its bare minimum 500 kilobyte/s, which is far above LoRa's capacity of at most 1.3 kilobyte/s. Audio streaming needs around 30 kilobyte/s, which is still too high even with enough quality compromises to lower its demand down to LoRa's capacity, given the 1% Duty Cycle. This last limitation restricts in general any consistent transfer of data, especially if observing how — combined with LoRa's low throughput — it allows no more than one Megabyte of traffic per day.

Feasible applications require the protocol to be used for small and infrequent chunks of data, for instance to send periodic statistics or sporadic monitoring information. Such implementations could in addition take huge advantages from LoRa's derisive power absorption, for example by operating on batteries and avoid any wiring, besides allowing for placement in rough or wild environments.

Other important considerations involve transmission range, which was found to be highly fluctuating in accordance to environment settings. Urban areas in particular, are filled with unexpected, unpredictable, and possibly intermittent obstacles to radio transmissions, limiting coverage to less than 1 km. This phenomenon must be taken into account when planning similar deployments, likely requiring to invest in on-field tests to asses the feasibility of the project. Rural areas on the other hand, were found to be much more suitable for employing LoRa: distances of over 4 km could be crossed by the signal with no issues, and even hills and terrain conformations didn't disrupt efficiency.

Such considerations restrict and clarify even more in which scenarios LoRa is a good solution and when it is not. Remote reading of sensors, for instance, might be an ideal use case provided that coverage and interference tests are carried out prior to deployment. Interference tests, in particular, are essential when operating in ISM bands since they are free and possibly occupied by other users.

Security aspects, in conclusion, rule out many other possible employments. Although cryptographically sound, LoRa is vulnerable to several Denial of Service and Message Replay attacks, compromising its reliability and employability in critical operations, such as remote control of alarm systems.

6.1 Related and Future Works

Many related publications were consulted during the writing of this work, beginning with a thesis by Allahparast [1] who set up a similar network but mainly focused on analyzing MAC layer features such as Adaptive Data Rate.

Theroretical throughput calculations were found to coincide with the more elaborated ones made in [13], which also took into consideration packet air times and assessed feasibility of many different use cases accounting for they typical payload sizes and timing requirements. One other aspect mentioned in this article is an estimation of the maximum capacity — in terms of scalability — of LoRaWAN cells and nodes.

When planning the experiments of Chapter 4, [8] was the only article found to consider rural applications, which were in addition tested with different types of antennae and several technical and other in-depth experiments.

Possible future works include implementation tests of different MAC layer protocols that could rely on LoRa, maybe enhancing some features to trade off less useful others. For instance, [11] is an already developed LoRaWAN substitute that aims at maximum reliability of the communication, or again, the authors of [5] propose an innovative multi-hop protocol to reach even further distances, at the obvious price of prolonged delivery times.

Bibliography

[1] Soroush Allahparast. Development of an open-source gateway and network server for "Internet of Things" communications based on LoRaWAN technology, 2017/2018.

[2] Emekcan Aras, Gowri Sankar Ramachandran, Piers Lawrence, and Danny Hughes. Exploring the Security Vulnerabilities of LoRa. *2017 3rd IEEE International Conference on Cybernetics (CYBCONF)*, June 2017.

[3] Emekcan Aras, Nicolas Small, Gowri Sankar Ramachandran, Stéphane Delbruel, Wouter Joosen, and Danny Hughes. Selective Jamming of LoRaWAN using Commodity Hardware. *MobiQuitous 2017*, November 2017.

[4] Ismail Batun, Nuno Pereira, and Mikael Gidlund. Analysis of LoRaWAN v1.1 Security. *SMARTOBJECTS '18*, June 2018.

[5] Martin Bor, John Vidler, and Utz Roedig. LoRa for the Internet of Things. *International Conference on Embedded Wireless Systems and Networks (EWSN)*, February 2016.

[6] LoRa Alliance Technical committee. *LoRaWAN 1.0.2 Regional Parameters*. LoRa Alliance, February 2017.

[7] Semtech Corporation. *AN1200.22 LoRa Modulation Basics*. Semtech Corporation, May 2015.

[8] Oana Iova, Amy L. Murphy, Gian Pietro Picco, Lorenzo Ghiro, Davide Molteni, Federico Ossi, and Francesca Cagnacci. LoRa from the City to the Mountains: Exploration of Hardware and Environmental Factors. *International Conference on Embedded Wireless Systems and Networks (EWSN)*, February 2017.

[9] Dong-Hoon Kim, Eun-Kyu Lee, and Jibum Kim. Experiencing LoRa Network Establishment on a Smart Energy Campus Testbed. *Sustainability*, 11:1917, March 2019.

[10] Jaehyu Kim and JooSeok Song. A Simple and Efficient Replay Attack Prevention Scheme for LoRaWAN. November 2017.

[11] Link Labs. Symphony Link. https://www.link-labs.com/symphony. Last accessed 12/09/2019.

[12] Link Labs. What is LoRa? A Technical Breakdown. https://www.link-labs.com/blog/what-is-lora. Last accessed 13/09/2019.

[13] Konstantin Mikhaylov, Juha Petäjäjärvi, and Tuomo Hänninen. Analysis of the Capacity and Scalability of the LoRa Wide Area Network Technology. February 2016.

[14] Thomas Mundt, Alexander Gladisch, Simon Rietschel, Johann Bauer, Johannes Golz, and Simeon Wiedenmann. General Security Considerations of LoRaWAN Version 1.1 Infrastructures. *MobiWac '18*, October 2018.

[15] Pycom. LoPy Datasheet. https://docs.pycom.io/gitbook/assets/specsheets/Pycom_002_Specsheets_LoPy_v2.pdf. Last accessed 13/09/2019.

[16] Pycom. Pycom LoRa API. https://docs.pycom.io/firmwareapi/pycom/network/lora/. Last accessed 13/09/2019.

[17] SAP. Trenitalia Showcases Railway Innovation. https://news.sap.com/2016/09/trenitalia-showcases-railway-innovation-with-sap/. Last accessed 13/09/2019.

[18] N. Sornin (Semtech), M. Luis (Semtech), T. Eirich (IBM), T. Kramp (IBM), and O. Hersent (Actility). *LoRaWAN Specification 1.0.2*. LoRa Alliance, July 2016.

[19] HQ Software. The History of IoT. https://hqsoftwarelab.com/about-us/blog/the-history-of-iot-a-comprehensive-timeline-of-major-events-infographic. Last accessed 13/09/2019.

[20] Eef van Es, Harald Vranken, and Arjen Hommersom. Denial-of-Service Attacks on LoRaWAN. *ARES 2017*, August 2018.

Appendix A Code Listings

A.1 LoRaWAN End Device

A.1.1 `lorawan_end_device.py`: join a LoRaWAN network and simulate traffic

```
from network import LoRa
import socket, binascii, time


lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

lora.add_channel(0, frequency=868100000, dr_min=0, dr_max=5)
lora.add_channel(1, frequency=868100000, dr_min=0, dr_max=5)
lora.add_channel(2, frequency=868100000, dr_min=0, dr_max=5)

dev_eui = binascii.unhexlify('ce0c231fac544c72')
app_eui = binascii.unhexlify('0000000000000000')
app_key = binascii.unhexlify('63a4687b1b1762eecab6be334640a772')


lora.join(activation=LoRa.OTAA, auth=(dev_eui, app_eui, app_key), timeout=0,
    dr=5)
print('Joining...')

while not lora.has_joined():
    time.sleep(0.5)

for i in range(3, 16):
    lora.remove_channel(i)

s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
s.setblocking(False)

# Set same DR as Gateway
s.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)

time.sleep(5)

for i in range (200):
    pkt = b'PKT #' + bytes([i])
    print('Sending:', pkt)
    s.send(pkt)

    time.sleep(4)

    rx, port = s.recvfrom(256)
    if rx:
        print('Received: {}, on port: {}'.format(rx, port))
    time.sleep(6)
```

A.2 PHY Throughput Measurement

A.2.1 phy_throughput.py: record PHY transmission timings

```
from network import LoRa
import socket, binascii, time


data_rates = [
    # (SF, BW, PP)
    (12, LoRa.BW_125KHZ, 64),
    (11, LoRa.BW_125KHZ, 64),
    (10, LoRa.BW_125KHZ, 64),
    (9,  LoRa.BW_125KHZ, 128),
    (8,  LoRa.BW_125KHZ, 255),
    (7,  LoRa.BW_125KHZ, 255),
    (7,  LoRa.BW_250KHZ, 255),
]

lora = LoRa(mode=LoRa.LORA, region=LoRa.EU868)

PKT_N = 10
DR_stats = []
DR_stats_err = []

for i, dr in enumerate(data_rates):
    PKT_SIZE = dr[2]
    print("\n[*] Setting up Data Rate DR{}: SF{}BW{} with payload of {}
        bytes".format(i, dr[0], [125, 250, 500][dr[1]], PKT_SIZE))

    lora.sf(dr[0])
    lora.bandwidth(dr[1])

    CR_stats = []
    CR_stats_err = [[], []]

    for cr in [LoRa.CODING_4_5, LoRa.CODING_4_6, LoRa.CODING_4_7,
        LoRa.CODING_4_8]:
        print("    > Coding Rate: {}".format(cr))

        lora.coding_rate(cr)
        s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
        s.setblocking(True) # Set to blocking to measure execution time

        times = []
        for i in range(PKT_N):
            t_start = time.ticks_us()
            s.send(bytes([0x00]*PKT_SIZE))
            times.append(time.ticks_diff(t_start, time.ticks_us())/1000000)

        _avg, _min, _max = (PKT_SIZE*8)/(sum(times)/PKT_N),
            (PKT_SIZE*8)/max(times), (PKT_SIZE*8)/min(times)
        print("      Throughput (bit/s):   Avg {:.2f}  -  Min {:.2f}  -  Max
            {:.2f}".format(_avg, _min, _max))
        CR_stats.append(_avg)
        CR_stats_err[0].append(_avg - _min)
        CR_stats_err[1].append(_max - _avg)

        s.close()

```

```
    DR_stats.append(CR_stats)
    DR_stats_err.append(CR_stats_err)

print(DR_stats)
print('---')
print(DR_stats_err)
```

A.2.2 phy_plot.py: generate PHY bitrate charts

```
import numpy as np
import matplotlib.pyplot as plt


calc_stats = [
    [292,   244,  209,  183 ],
    [537,   447,  383,  335 ],
    [976,   813,  697,  610 ],
    [1757,  1468, 1255, 1098],
    [3125,  2604, 2232, 1953],
    [5468,  4557, 3906, 3417],
    [10937, 9114, 7812, 6835]
]
DR_stats = [
    [183.0699, 158.8705, 140.322,  125.6519],
    [327.2082, 282.7559, 248.9364, 222.3428],
    [728.5637, 633.0759, 559.7156, 500.6125],
    [1504.223, 1278.804, 1114.559, 987.7047],
    [2868.992, 2414.922, 2083.875, 1835.113],
    [5031.446, 4240.766, 3673.361, 3230.69 ],
    [9929.534, 8393.418, 7258.546, 6394.041]
]
DR_stats_err = [
    [
        [0.0123291, 0.01039124, 0.007385254, 0.00554657],
        [0.1003723, 0.073349,   0.05751038,  0.04634857]
    ], [
        [0.03720093, 0.02789307, 0.02166748, 0.01705933],
        [0.3118896,  0.234467,   0.1820526,  0.1440887 ]
    ], [
        [0.1838989, 0.1417847, 0.1081543, 0.08752441],
        [1.548584,  1.181702,  0.9139404, 0.733429  ]
    ], [
        [0.4039307, 0.2872314, 0.2218018, 0.1752319],
        [3.337646,  2.384155,  1.808228,  1.433167 ]
    ], [
        [11.54517, 0.5090332, 0.6162109, 3.762207],
        [7.277344, 4.263916,  0.1669922, 3.944458]
    ], [
        [30.52637, 25.21094, 17.00171, 12.59326],
        [7.470703, 15.90283, 13.82788, 3.083252]
    ], [
        [118.21,   98.27344, 73.67676, 55.93994],
        [29.06934, 62.40039, 46.44825, 35.93604]
    ]
]

fig = plt.figure()
fig, axs = plt.subplots(nrows=4, ncols=2, figsize=(9, 13))

for dr in range(7):
    x = ['CR 1', 'CR 2', 'CR 3', 'CR 4']
```

```
    y = DR_stats[dr]
    yerr = DR_stats_err[dr]

    ax = axs[dr//2, dr%2]
    ax.plot(x, calc_stats[dr], c='orange', marker='d')
    ax.errorbar(x, y, yerr=yerr, ecolor="r", capsize=5, barsabove=True)
    ax.set_ylabel("Bitrate (bit/s)")
    ax.set_title("Data Rate \"DR" + str(dr) + "\"")
    ax.grid(True, linestyle=':')

fig.legend(['Calculated Bitrate', 'Measured Bitrate'], loc='upper center')
plt.subplots_adjust(top=0.930, bottom=0.039, left=0.088, right=0.974,
    hspace=0.315, wspace=0.25)
plt.show()
```

A.3 MAC Throughput Measurement

A.3.1 mac_throughput.py: record MAC transmission timings

```
from network import LoRa
import socket, binascii, time


lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

dev_eui = binascii.unhexlify('ce0c231fac544c72')
app_eui = binascii.unhexlify('0000000000000000')
app_key = binascii.unhexlify('63a4687b1b1762eecab6be334640a772')

lora.add_channel(0, frequency=868100000, dr_min=0, dr_max=6)
lora.add_channel(1, frequency=868100000, dr_min=0, dr_max=6)
lora.add_channel(2, frequency=868100000, dr_min=0, dr_max=6)

lora.join(activation=LoRa.OTAA, auth=(dev_eui, app_eui, app_key), timeout=0,
    dr=5)

print('Joining...')
while not lora.has_joined():
    time.sleep(0.5)

for i in range(3, 16):
    lora.remove_channel(i)


data_rates = [(0, 51), (1, 51), (2, 51), (3, 115), (4, 242), (5, 242), (6,
    242)]
PKT_N = 10

DR_stats = []
DR_stats_err = [[], []]

for dr in data_rates:
    print("[*] Testing DR{}".format(dr[0]))
    PKT_SIZE = dr[1]

    s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
    s.setsockopt(socket.SOL_LORA, socket.SO_CONFIRMED, False)
    s.setsockopt(socket.SOL_LORA, socket.SO_DR, dr[0])
    s.setblocking(True)
```

```
39
40      times = []
41      for i in range(PKT_N):
42          t_start = time.ticks_us()
43          s.send(bytes([0x00]*PKT_SIZE))
44          times.append(time.ticks_diff(t_start, time.ticks_us())/1000000)
45          time.sleep(0.5)
46
47      _avg, _min, _max = PKT_SIZE/(sum(times)/PKT_N), PKT_SIZE/max(times),
            PKT_SIZE/min(times)
48      DR_stats.append(_avg)
49      DR_stats_err[0].append(_avg - _min)
50      DR_stats_err[1].append(_max - _avg)
51
52      print("      Min: {:.2f} byte/s  -  Max: {:.2f} byte/s  -  Avg: {:.2f}
            byte/s".format(
53          _min, _max, _avg
54      ))
55
56      s.close()
57
58  print(DR_stats)
59  print('---')
60  print(DR_stats_err)
```

A.3.2 mac_plot.py: generate MAC throughput charts

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4
5   calc_stats = [
6        [x*8 for x in [31.5, 58,   105.4, 205.3, 378.1, 661.6, 1323 ]],
7        [x*8 for x in [13.9, 25.6, 46.6,  111.6, 221.7, 387.8, 775.8]]
8   ]
9   DR_stats = [x*8 for x in [8.4877, 10.26163, 12.78267, 28.73583, 60.46851,
        60.46923, 80.59957]]
10  DR_stats_err = [
11       [x*8 for x in [0.08555317,  0.06741142, 0.04193115, 0.006668091,
             0.02728653, 0.02172852,  0.09063721]],
12       [x*8 for x in [0.009656906, 0.6314449,  0.3620882,  0.0008487701,
             0.00334549, 0.002700806, 0.01717377]]
13  ]
14
15
16  plt.title('LoRaWAN MAC throughput', size=14, y=1.05)
17  plt.xlabel('Data Rate')
18  plt.ylabel('Throughput (bit/s)')
19  plt.fill_between(np.arange(0, 7), calc_stats[0], calc_stats[1],
        facecolor='yellow', label='Therotetical (CR1 to CR4)')
20  plt.plot(np.arange(0, 7), calc_stats[0], np.arange(0, 7), calc_stats[1],
        c='orange')
21
22  plt.errorbar(np.arange(0, 7), DR_stats, yerr=DR_stats_err, ecolor="r",
        capsize=5, barsabove=True, label='Measured (undefined CR)')
23  plt.grid(True, which='both', linestyle=':')
24
25  plt.legend(loc='upper center')
26  plt.show()
```

A.4 Range and Coverage Testing

A.4.1 `range_test_sender.py`: dummy packet sender

```
from network import LoRa
import socket, time, pycom


# Show idle status
pycom.heartbeat(False)
pycom.rgbled(0x006600)

lora = LoRa(mode=LoRa.LORA, region=LoRa.EU868)

data_rates = [
    # (SF, BW, PP)
    (12, LoRa.BW_125KHZ, 64),
    (11, LoRa.BW_125KHZ, 64),
    (10, LoRa.BW_125KHZ, 64),
    (9,  LoRa.BW_125KHZ, 128),
    (8,  LoRa.BW_125KHZ, 255),
    (7,  LoRa.BW_125KHZ, 255),
    (7,  LoRa.BW_250KHZ, 255),
]

PKT_N = 10

while True:
    print("\n[*] Starting...")

    for i, dr in enumerate(data_rates):
        PKT_SIZE = dr[2]
        print("[*] Setting up Data Rate DR{}: SF{}BW{} with payload of {}
            bytes".format(i, dr[0], [125, 250, 500][dr[1]], PKT_SIZE))
        pycom.rgbled(0x220000)

        lora.sf(dr[0])
        lora.bandwidth(dr[1])

        for cr in [LoRa.CODING_4_5, LoRa.CODING_4_6, LoRa.CODING_4_7,
            LoRa.CODING_4_8]:
            print("    > Coding Rate: {}".format(cr))
            lora.coding_rate(cr)

            s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
            s.setblocking(True)

            times = []
            print("      Progress: [" +  ' '*(PKT_N) + "] 0%", end='\r')
            for i in range(PKT_N):
                s.send(bytes([cr]*PKT_SIZE))
                print("      Progress: [{}{}] {:.0f}%".format(
                    '#'*(i+1), ' '*(PKT_N-i-1), (i+1)/PKT_N*100
                ), end='\r')

                # Show that a packet was sent
                pycom.rgbled(0x000000)
                time.sleep(0.1)
                pycom.rgbled(0x220000)

```

```
55              print('')
56              s.close()
57
58          # Show idle and give time to update receiver to next DR
59          pycom.rgbled(0x006600)
60          input("    > Press Enter to continue...")
61
62      pycom.rgbled(0x000022)
63      input("\n[*] Press Enter to restart...\n")
```

A.4.2 range_test_receiver.py: listener and logger

```
1  from network import LoRa
2  import socket, time, ubinascii
3
4
5  lora = LoRa(mode=LoRa.LORA, region=LoRa.EU868, tx_power=14)
6
7  data_rates = [
8      # (SF, BW, PP)
9      (12, LoRa.BW_125KHZ, 64),
10     (11, LoRa.BW_125KHZ, 64),
11     (10, LoRa.BW_125KHZ, 64),
12     (9,  LoRa.BW_125KHZ, 128),
13     (8,  LoRa.BW_125KHZ, 255),
14     (7,  LoRa.BW_125KHZ, 255),
15     (7,  LoRa.BW_250KHZ, 255),
16 ]
17
18 dr = 0
19 dr_stats = []
20
21 while dr <= 6:
22     lora.sf(data_rates[dr][0])
23     lora.bandwidth(data_rates[dr][1])
24     s = socket.socket(socket.AF_LORA, socket.SOCK_RAW)
25     s.setblocking(False)
26
27     print("[*] Listening on DR{}... (Ctrl-C to advance to next
           DR)".format(dr))
28     cr_stats = [0]*4 # One counter for every CR
29
30     try:
31         while True:
32             tmp = s.recv(data_rates[dr][2])
33             if tmp:
34                 # Payload is CR number
35                 cr_stats[int(tmp[0]) - 1] += 1
36             else:
37                 time.sleep(1);
38     except KeyboardInterrupt:
39         print(cr_stats)
40         dr_stats.append(cr_stats)
41         s.close()
42         dr += 1
43
44 print("\n=== STATS: ===")
45 print(dr_stats)
```