University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Cloud Radio Access Network in LoRa

Silas Weber
Zurich, Switzerland
Student ID: 14-704-845

Supervisor: Eryk Schiller
Date of Submission: February 3, 2019

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Das ist die Kurzfassung...

Acknowledgments

Optional

Contents

Chapter 1

Introduction and Motivation

Scalability and improvement of Internet of Things (IoT) devices and protocols are important research questions. Low Power Wide Area Networks (LPWANs) technology offers long-range communication with low poser requirements. Battery powered LPWAN devices can run for years. For instance, a node sending 100B once a day lasts for 17 years [1]. LoRa (short for Long Range) is a spread spectrum modulation technique, a wireless radio frequency technology for long range and low power platforms and has become the de facto technology for IoT networks worldwide [2]. LoRaWAN is the open standard backed by the LoRa Alliance. It is a communication protocol and Medium Access Control (MAC) protocol built on the physical LoRa layer. LoRaWAN is designed from the bottom up to optimize LPWANs for battery lifetime, capacity, range, and cost [3]. There are 142 countries with LoRaWAN deployments, 121 network operators, and 76 LoRa Alliance member operators. Swisscom, Amazon, IBM, CISCO are merely a few of the notables LoRa Alliance members [4]. TTN (The Things Network), also a LoRa Alliance member, provides a worldwide LoRaWAN network for and from the community. Anyone with a LoRa gateway can register their gateway on TTN, thereby extending the networks reach. At the time of writing, TTN has 95'208 members, 9'786 gateways, and is present in 147 countries [5]. As LoRaWAN operates in the unlicensed ISM (Industrial, Scientific and Medical) radio bands. Therefore no government license is required to operate LoRa devices and gateways. This allows hobbyist, enthusiasts, and developers to quickly get started and open networks such as TTN to grow rapidly.
In a typical LoRaWAN use case, an IoT device such as a sensor sends data out over the air. Then a LoRa gateway picks the signal up, decodes it, and forwards it over the Internet to the network server which then can send the packet to the application server. If needed, a response message can scheduled on the network server who then chooses the best gateway to send the response back to the IoT device. LoRa gateways carry the full implementation of the LoRa PHY (the physical layer), the LoRaWAN protocol, as well as the packet forwarder. This architecture of the LoRa gateway can be separated and technological stack on the gateway can be reduced by running the signal processing functions not on the gateway itself but in a cloud environment. Such a Cloud Radio Access Network (CRAN) has been previously shown to be beneficial in the 3rd Generation Partnership Project (3GPP) Long Term Evolution (LTE) [6]. The gateway then is left with only minimal functionality it has to support. As the decoding does not take place on the gateway

itself, it does not need do have any LoRa specific hardware e.g the SX1276 transceiver chip found on LoRa devices and gateways. Rather, the gateway is equipped with an antenna, an amplifier as well as digital to analog (DAC) and analog to digital (ADC) converters. On the upstream, the gateway receives LoRa radio signals which it converts into Inphase and Quadrature (I/Q) sample stream with the ADC and simply forwards them to the cloud signal processing unit via the internet. On the downstream the cloud unit streams a LoRa signal as I/Q samples to the gateway which converts it with the DAC to an analog signal and propagates it out over the air. Signals are encoded and decoded on the cloud unit, the Radio Cloud Center (RCC). There are many advantages in such a setup but they come at a cost. First advantage is that the gateway can be kept at a much simpler design resulting in significant manufacturing cost reduction. Also, modifications to the LoRa PHY or LoRaWAN are easier to introduce as the physical layer is implemented in software. Gateways that are once deployed do not need to be physically replaced in case of an upgrade as they are agnostic to the underlying protocol and just convert and transceive (transmit and receive) I/Q samples. Updates to the protocol can be realized with just updating the software implementation. A Low Power Network (LPN) provider saves cost by not having to drive out to the deployed gateways throughout the country to upgrade their versions. The disadvantage is the high throughput of the I/Q samples stream between the gateway and the RCC. Streaming the I/Q samples between gateway and RCC has significantly higher bandwidth requirements than just demodulating the signal on the gateway and forwarding the decoded LoRa packet as it is done in the non cloudified setup. Cloudifying the LoRa gateways also brings the advantages of setting the base for Software Defined Networking (SDN) and Network Function Virutalization (NFV) by centralizing the resources in the RCC that were before distributed on the individual gateways. Goal of this work is setting up a CRAN architecture for LoRa by simplifying the gateways as described above and moving the signal processing out of the gateway into a cloud ready environment i.e., Docker.

1.1 Description of Work

This work first gives a general introduction to LoRa, LoRaWAN and its applications, then dives into more details regarding the LoRa physical layer. Then it gives an overview over existing software implementations of the LoRa PHY. There are two main contributions. First, this work implements a CRAN for LoRa, gives an architectural overview as well as the implementation details. It evaluates the architectural and network related requirements. We developed a simple protocol in raw LoRa, meaning not compliant with the LoRaWAN standard, where a hardware IoT device has a queue of packets to transmit then, depending on wether it requires an acknowledgment, waits for a few seconds for a response or just transmit the next packet in the queue in an interval. If the packet required to be acknowledged but no acknowledgment is received, the same packet will put as first item in the queue. We use this protocol to analyze our CRAN for LoRa architecture.
Second, as the LoRa PHY is closed source, there is no official documentation on how the LoRa PHY is implemented. The existing implementations are all reverse engineering attempts with various degree of success. They all focused first on decoding LoRa signals transmitted by a real LoRa hardware. For the CRAN to work, not only is it necessary

to decode signals but also the encoding of downstream LoRa gateway signals is required. To achieve this we developed a tool that allows the generation of downstream signals in software.

1.2 Thesis Outline

Chapter 2

LoRa and LoRaWAN

LoRa is a modulation technique derived from chirp spread spectrum technology[2]. Originally developed by Cycleo, a french company, LoRa has been acquired by Semtech [7]. LoRa signals spread over multiple frequencies using the whole available bandwidth. This makes the signal more resilient against noise on a disrupting frequency. As LoRa signal are sent over the unlicensed ISM bands, this resilience is an important factor. While LoRa is the modulation technique on the physical layer, LoRaWAN on the other hand is an open communication protocol backed by the Lora Alliance. LoRaWAN specifies packet format, duty cycles, key exchanges and many more things needed for an efficient and cooperative LoRa network. A LoRa network is and LPWAN where battery powered devices can stay operating up to 17 years, making LoRa a popular choice for IoT devices as shown in the example given in the introduction in chapter 1. The TTN network for example is used for cattle tracking, smart irrigation as well as smart parking applications [5].

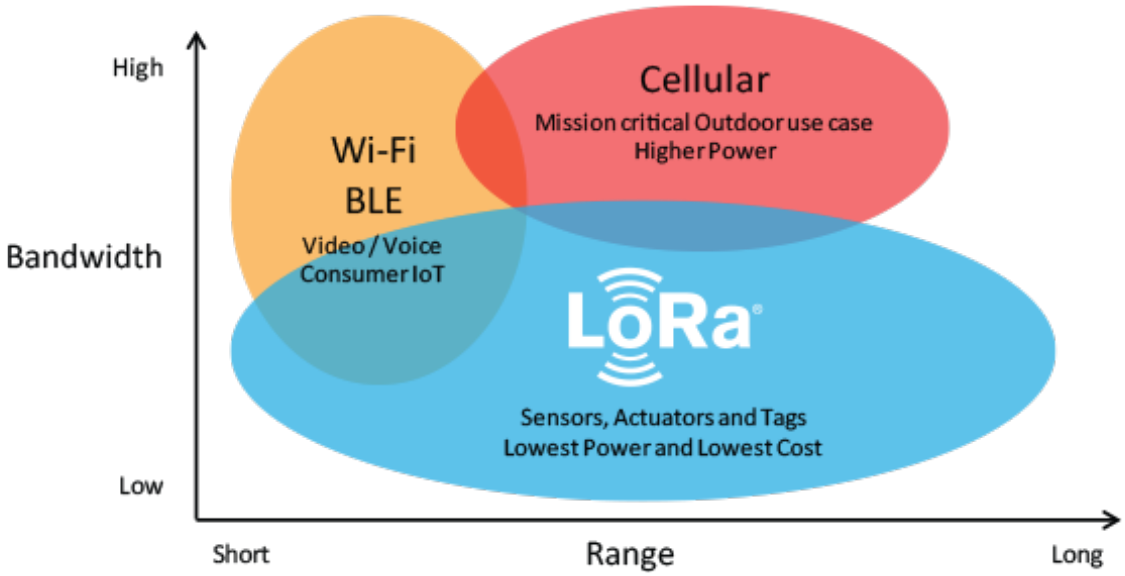

Figure 2.1: LoRa vs other wireless technology[8]

Figure 2.1 shows LoRa compared two other wireless technologies, Wi-Fi and cellular. Both

Wi-Fi and cellular are high in bandwidth with cellular having a longer range than Wi-Fi. They both have a much higher power consumption compared to LoRa. LoRa has lower bandwidth but a high range. In a experiment during a TTN conference LoRa signals from a low orbit satellite were received [9]. On the other hand, as LoRa is designed for long range and low power, only few bytes are transmitted per day while Wi-Fi and cellular are capable of video streaming. In urban areas LoRa has a range of 2-5 km and 15 km in suburban areas [7].
LoRaWAN is not the same all around the world. There are regional parameters that come into play, one is for example the frequency band. In Europe LoRaWAN operates on the in the 863-870MHz and 433MHz ISM band and in North America the 902-928MHz ISM band. Also channel bandwidth and maximum transmission settings are regulated by the government and thus are not the same for all regions [10].

2.1 LoRaWAN architecture

A LoRaWAN network architecture is a star-of-stars topology. The gateways relay the messages between the end-devices and a central network server. Gateways are connected to the network server via IP connections, converting the RF packets to IP packets and vice versa [11]. Network nodes are not associated with a specific gateway, rather messages sent by a node can be received by multiple gateways. Each gateway will then forward the the message to the network server who does the complex things such as filtering redundant packages, security checks, forwarding the messages to the right application server etc. [3]. As network communication is bidirectional, the network server is also responsible for scheduling responses to the end-nodes. There are different classes of end-nodes which will be described in the next section.

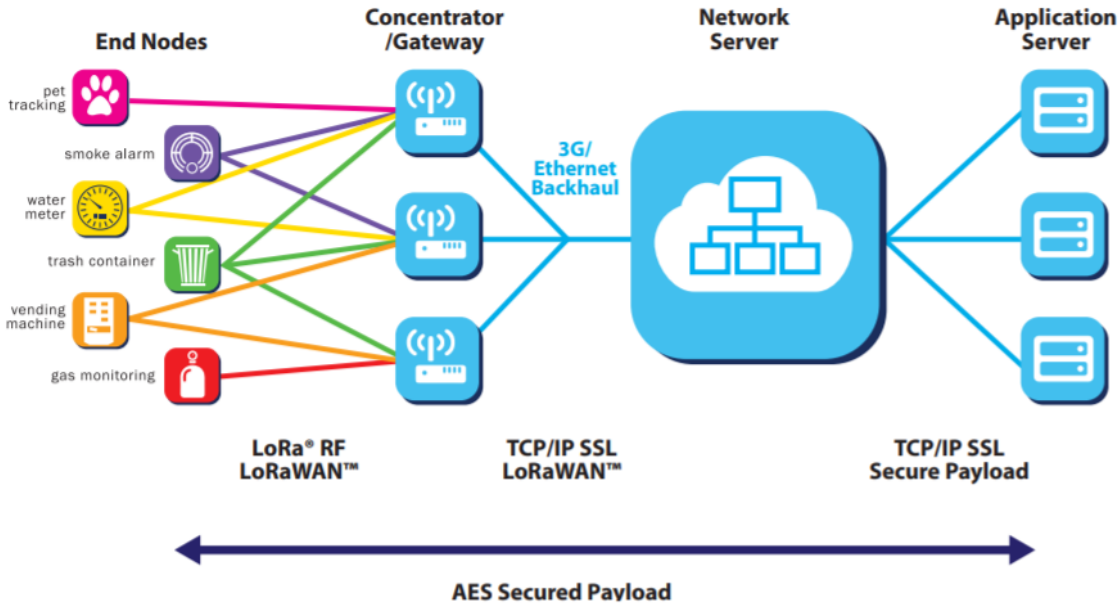

Figure 2.2: LoRaWAN network architecture [3]

As depicted in Figure 2.2, the packets sent by end-devices (on the far left) such as alarms, tracking devices and monitoring devices, can be received by multiple gateways. As the end-nodes are not linked to a particular gateway, the can be moved freely which is an important requirement for assets tracking.
The Figure also shows how security is built into LoRaWAN. The payload is end-to-end encrypted from the end-nodes to the applications server. A unique 128-bit network session key is shared between the end-device and the network server and another 128-bit application session key is shared end-to-end at the application level [11]. With those measures LoRaWAN prevents eavesdropping. Spoofing is prevented by a MIC (Message Integrity Code) in the MAC payload, and replay attacks are prevent by utilizing frame counters [12].

2.2 End-node Classes

There are three classes of end-devices. The following description is adapted from the LoRa Alliance guide [11, 2]:

- Class A, Lowest power, bi-directional end-devices:

  This is the default class, supported by all LoRaWAN devices. It is always the end-node that initiates the communication. After an uplink two downlink windows open for the end-device to receive a response, enabling bi-directional communication. Either the first is used, or the second, but not both receive windows. The end-device can rest in low-power sleep mode, wake up when it needs to send a packet, receive a response in the downlink window, then go back to seep. This is an ALOHA-type of protocol. Class A devices have the lowest power consumption. Downlinks from the server have to wait for an uplink from end-device and cannot be initiated directly.

- Class B, Bi-directional end-devices with deterministic downlink latency:

  Additionally to Class A receive windows, a Class B device opens extra receive windows at scheduled times. This is achieved by time-synchronized beacons from the gateway to the end-device to notify the end-device to open a receive window.

- Class C, Lowest latency, bi-directional end-devices:

  Devices of this class have always open receive windows, except for when they are themselves transmitting. A downlink transmission can be initiated by the network server at any time (assuming the device is not currently transmitting) resulting in no latency. Class C devices however use the most energy. They are more suitable for plugged in devices rather than battery powered devices.

2.3 LoRa signal (uplink)

2.3.1 Chirps

A LoRa signal is a series of so called chirps as LoRa is derived from the Chirp Spread Spectrum modulation (CSS) technique. There are up-chirps and down-chirps. In CSS chirps are deliberately spread across the available bandwidth. Up-chirps go from low frequency to high frequency and down-chirps go from high frequency to low frequency. In Europe the LoRaWAN bandwidth for is 125 kHz. Assuming a center frequency of 868.5 MHz, which is in the european ISM band, a full up-chirp, so called sweep, would go from 868.4375 MHz to 868.5625 MHz.

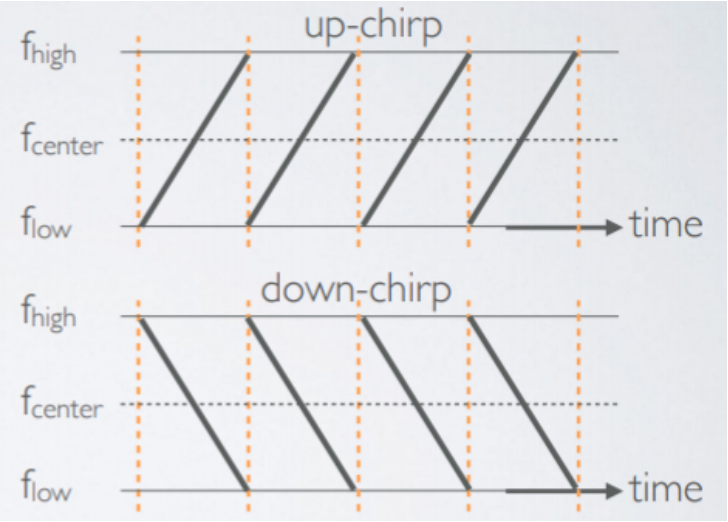

Figure 2.3: Up- and down chirps [13]

Figure 2.3 shows the linear frequency increase resp. decrease over time over the full bandwidth for up-chirps and down-chirps. Data is encoded by frequency jumps in the chirps.

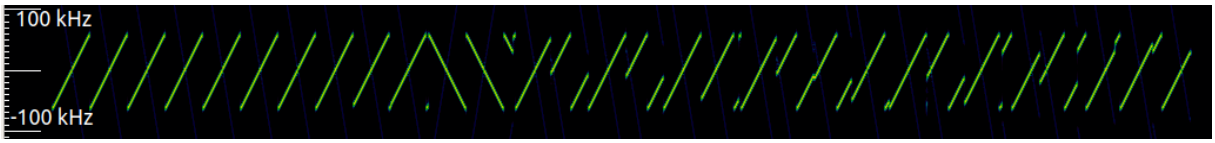

Figure 2.4: Own recording of uplink transmission by arduino equipped with a LoRa shield

The LoRa signal shown in 2.4 carries the message ”Goodbye !”. This message was sent with a spreading factor (SF) of 9 and coding rate of 4/5. The terms spreading factor and coding rate will be discussed later on.
As one can see, a typical LoRa signal start with a so called preamble, which are the 10 up-chirps at the beginning. Those are followed by two down-chirps, which signify the end of the preamble and the start of the actual payload. In this payload is a header, the actual encoded message followed by a Cyclic Redundancy Check (CRC). The CRC is used for error correction.

2.3.2 Symbol and Spreading Factor

A LoRa signal holds various symbols. A symbol encodes one or more bits of data. The spreading factor determines the number of encoded bits in a symbol. In the shown recording one symbol holds 9 bits of data as the spreading factor of that signal was set to 9. It follows that a symbol has 2^{SF} values. Those values range from 0 to 511 in case of SF 9. A sweep signal of SF 9 thus has 512 chips (no to be confused with chirps) [14]. The chips go linearly from low to high and then wrap around once the maximum frequency is reached.
In Figure 2.5 a fictional symbol with SF 7 is shown. This particular arrangement of chips highlighted in orange would denote the symbol "1011111". Those 7 bits correspond to the decimal value 95.

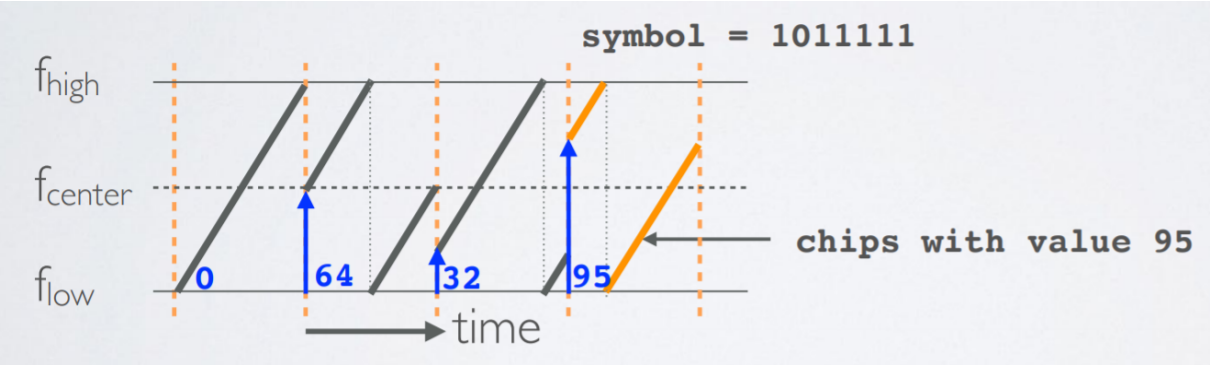

Figure 2.5: Chips and symbols value [14]

In Figure 2.6, a real world example is shown. The same LoRa signal as in Figure 2.4 with SF 9 with the message "Goodbye !" run through modified version of the LoRa decoder by Robyns et al. [15] and then through a python script where we match the samples to the symbols and their values. The last symbols encodes the hex value 142 which corresponds to these 9 bits "101000010". In a SF 9 signal each symbol encodes 9 bits.

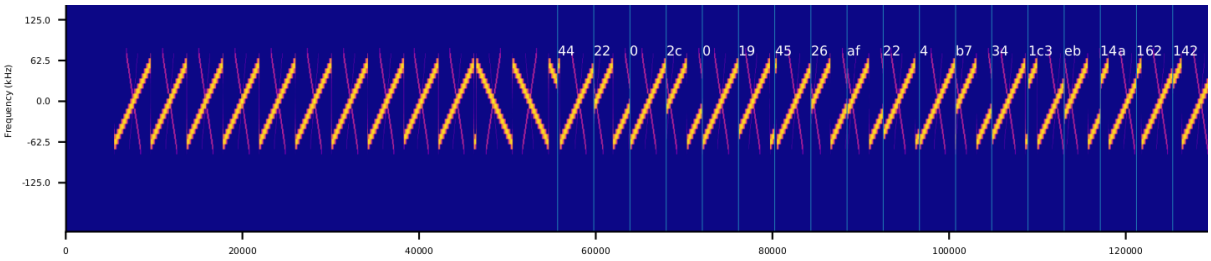

Figure 2.6: Running the signal through our toolchain, matching symbols with samples

2.3.3 Coding Rate

LoRa signals are encoded with a coding rate (CR). The CR denotes the proportion of how many bits carry actual information. The bits that do not carry information are used for Forward Error Correction. The formula for coding rate is $CR = 4/(4 + CR)$ where CR $\in \{1, 2, 3, 4\}$. A CR of 1 is thus the proportion of 4/5 of actual information over bits used for error correction[16, 17]. With FEC, corrupted bits e.g. due to interference can be corrected. With CR of 4, corresponds to $4/8 = 1/2$, half the transmitted bits carry information, the other half is for FEC. The higher the CR (from 1-4) the more bits can get corrupted and corrected by FEC. On the other hand, the higher the CR the more bits need to be transmitted which drains the battery more.

2.3.4 Spreading Factor & Time on Air

The longer the packet, the longer the transmission time. LoRa packets can be shortened by sending packets with implicit header mode where the no header is sent and the settings that would have been specified in the header have to be predefined manually on the end device.
Assuming constant packet size and same bandwidth, varying the spreading factor increases resp. decreases the time on air. The higher the SF, the longer the time on air. Higher SF means longer range. The spreading factor goes from 7 to 12. SF 7 has the shortest range, SF 12 the longest. The spreading factor essentially sets the duration of a chirp, a full sweep [18].
The symbol time is defined in the LoRa Design guide by $T_{sym} = \frac{2^{SF}}{BW}$ [16]. It follows as stated above, that the higher the SF the longer the symbol duration. Also, the higher the bandwidth (BW) the shorter the symbol duration. In Europe the BW is 125 kHz, while in North America a BW of 500 kHz is allowed. It also follows that with an increase in SF by 1 the symbol duration is doubled. The bit rate R_b is then defined by $R_b = SF * \frac{[\frac{4}{4*CR}]}{[\frac{2^{SF}}{BW}]}$ with CR being the coding rate for the error correction scheme [19]. It follows from the formula that the higher the coding rate the lower the bit rate as with a higher CR more redundancy is added for the error correction scheme. Highest data rate for $BW = 125\,kHz$ and $CR = 1$ is achieved with SF 7 resulting in a data rate of 5.5 kbits/s and the lowest data rate is achieved with SF 12 resulting in a data rate 0.29 kbits/s.
The spreading factors are orthogonal to each other, meaning signals on different spreading factors do not interfere with each other. This is Code Division Multiple Access (CDMA) where a shared medium i.e. the bandwidth is optimized for multiple access.
To optimize network capacity LoRaWAN employs a method called Adaptive Data Rate (ADR). With ADR the network server signals the end-device which spreading factor to use according to some measurements including the signal to noise ratio. Assuming there are multiple devices near a gateway that transmit with SF 12. This occupies the bandwidth for device that are farther away and actually need SF 12. The network server detects that the nearby devices do not need a spreading factor of 12 and signal them to use a lower SF such as SF 7 or SF 8. The ADR setting has to be enabled on the end-devices and can be disabled.

2.3.5 Packet structure

The base form of a LoRa packet starts with the preamble, followed by the optional header with a header CRC, followed by the payload and finally the payload CRC. The number of payload symbols is calculated by the following formula [16]:

$$payloadSymbNb = 8 + max(ceil(\frac{8PL-4SF+28+16-20H}{4(SF-2DE)}) * (CR+4), 0)$$

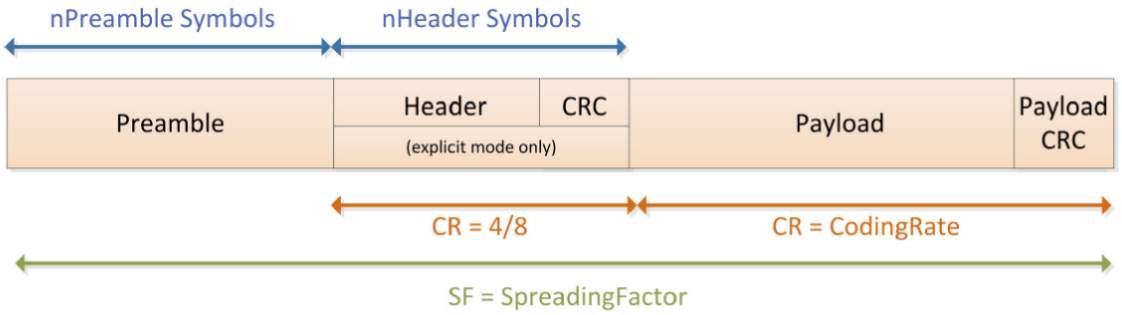

Figure 2.7: LoRa packet structure [16]

With:

1. PL being the number of payload bytes
2. SF being the spreading factor
3. H = 0 if header is enabled and H = 1 if no header
4. DE = 0 if low data rate optimization is enabled and DE = 0 if disabled
5. CR being the coding rate

This website `https://www.loratools.nl/#/airtime` has an online tool for calculation the airtime.

As Figure 2.7 shows, the header is always encoded with the highest coding rate, $CR = 4$. This is because the header contains crucial information such as the packet length.

Figure 2.8 and Figure 2.9 show the structure of an uplink resp. a downlink packet. There is no CRC in downlink packets. PHDR stands for PHY header. Those are ”raw” LoRa packets. LoRaWAN packets have additional fields such as MAC header (MHDR) and frame header (FHDR). Those are all in PHY payload of the ”raw LoRa” packet as Figure 2.10 shows.

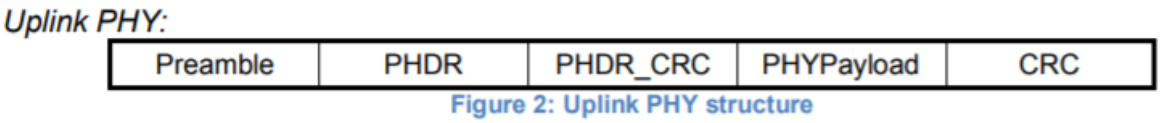

Figure 2.8: LoRa uplink packet structure [20]

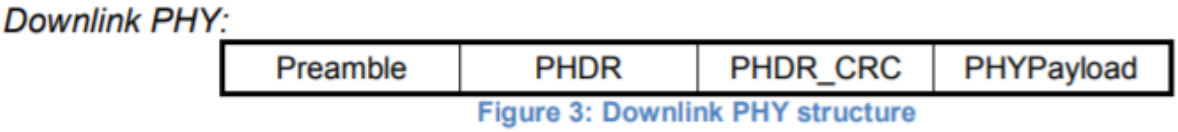

Figure 2.9: LoRa downlink packet structure [20]

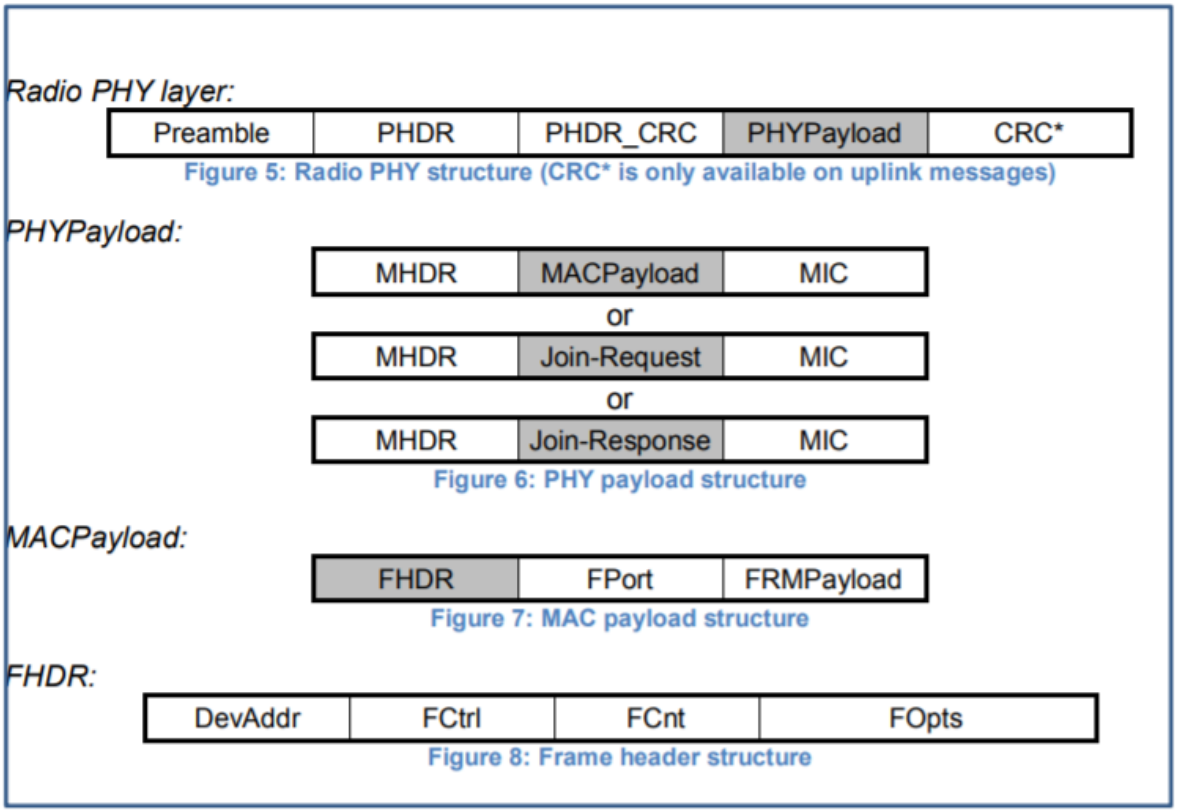

Figure 2.10: LoRaWAN packet [20]

Chapter 3

LoRa in SDRs

Software-defined radios (SDR) implement components that are usually implemented in hardware. The most popular signal processing frameworks is GNU Radio For LoRa we were looking for an existing implementation that demodulates, and also modulates LoRa signals.

3.1 Existing implementations

There are three existing implmentations we looked at:

- Josh Blum's LoRa Mod -and Demodulator for LoRa in the Pothos framework `https://myriadrf.org/news/lora-modem-limesdr/`
- Matt Knight's GNU Radio Module `https://github.com/BastilleResearch/gr-lora`
- Robyns et al. LoRa Module for GNU Radio `https://github.com/rpp0/gr-lora`

We tried Blum's implementation in the Pothos first. The Pothos projects is an open source data-flow framework supporting SoapySDR, a general framework for supporting SDR devices [21]. Unfortunately the LoRa modem demo application did not work for at all. After spending a few unsuccessful days trying to get to the issue we moved to Knight's application.

Matt Knight held a great talk on reverse engineering LoRa at the GNU Radio conference 2016 `https://www.youtube.com/watch?v=-YNMRZC6v1s`. GNU Radio, as Pothos, is a framework for signal processing.
From the GNU Radio website:

> GNU Radio is a free & open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with

> readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used in research, industry, academia, government, and hobbyist environments to support both wireless communications research and real-world radio systems. [22]

GNU Radio already comes with a wide set of blocks. Extensions are called Out Of Tree modules (OOT) as they are not in the standard tree of blocks. Knight's implementation did not work well for us. If we got an output from the decoder, it was not what was expected. A reason could be that in his examples the signal source block is an USRP SDR while we had a LimeSDR mini at our disposal. Simply switching the source blocks probably is not enough. We did not investigate the compatability between LimeSDR mini and and USRP further but moved on to the final implementation. His blocks are in modular fashion. The demodulator and decoder are separate blocks. Channelization and fine tuning must be done explicitly before passing the stream to the demodulator block.

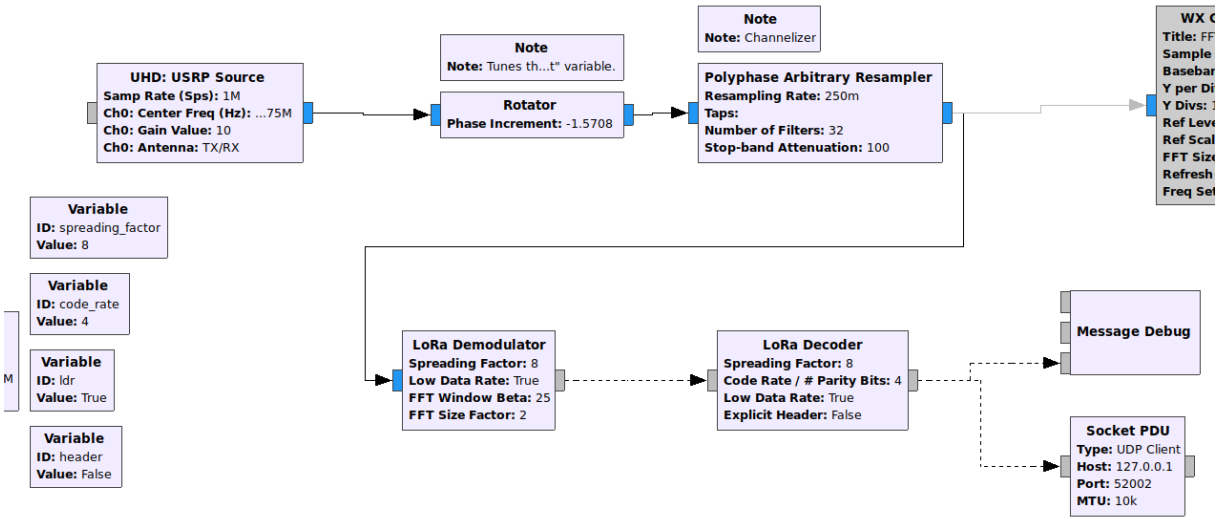

Figure 3.1: Knight's GNU Radio gr-lora OOT RX example [23]

Figure 3.1 shows the typical flow of a GNU Radio flowgraph where data is passed from block to block as a stream (blue connection ports) or as message blocks (grey connection ports).
Robyns' implementation is also an OOT module for GNU Radio. It has a elaborate installation and usage guide. A docker environment is also provided for testing the decoding of a LoRa signal, which is a big plus. Unlike Knight's implementation, this module has demodulation, decoding and channelization all in one single block as shown in Figure 3.2. The module has been tested with various SDR devices but not with the LimeSDR mini. Nevertheless it worked well for us and we based the CRAN implementation for LoRa on this module.

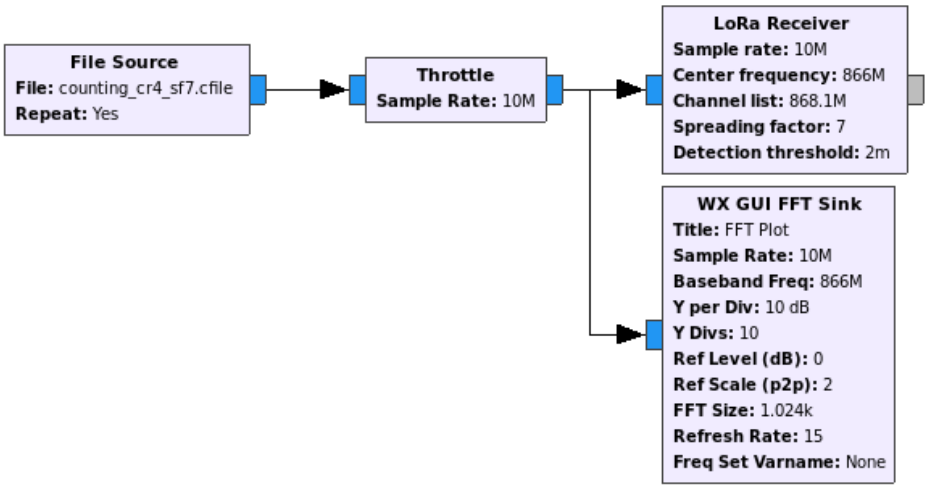

Figure 3.2: Single LoRa Receiver block (top right) [24]

3.2 LoRa decoding

The difficulty with reverse engineering LoRa is that its proprietary and there is no official documentation on the PHY. To reverse engineer, information hints on the PHY layer have to be taken from various official LoRaWAN documents, from patents, and the rest is guesswork. To make it more difficulty, some of the documentation is a lie as the PHY is not implement in the way it is described. complete lie, see Knight [25]. The data is encoded before it is sent over the air to make it more resistant against interference. Thus after demodulating the signal with a Fourier Transform, the data has to be decoded to make it usable. Semtech's european patent hints at the following four steps:

1. Symbol gray indexing. This adds error tolerance
2. Data whitening. This induces randomness.
3. Interleaving. This scrambles the bits within a frame
4. Forward Error Correction. This adds correction parity bits.

Those are 4 distinct operation which have to be reverse engineered [25].
Robyns et al. identified and implemented the following seven steps in their receiver to receive and decode a LoRa signal: detection, synchronization, demodulation, deinterleaving, dewhitening, decoding, and packet construction [24]. They also provide a detailed description of the packet structure, especially the header. They deduce that because the minimum SF is SF 7, and the header is always transmitted mit CR = 4, it must fit in an interleaving matrix of a certain size which results int the header heaving a size of 40 bits. The header contains important data as the payload length, thus it makes sense that the header is always sent with the highest coding rate. At the time of Knight's talk, he did not decode the header. Robyns implementation is quite complete except for CRC checks of the payload and header as well as decoding multiple channels simultaneously.

Chapter 4

C-RAN in LTE advantages graphics

pdflatex

Chapter 5

C-RAN for LoRa

5.1 Goal

5.2 Methods

5.2.1 Sending uplink signals

5.2.2 Sending downlink signals

getting a downlink signal recording from thethingsnetwrok recording from private networks manipualting private gateway offline generation of downlink signal see chapter

5.2.3 Transmission protocol

5.3 Architecture

5.3.1 BBU

5.3.2 RRH

5.3.3 Network

5.4 Implementation

5.5 Results

Chapter 6

Future work

6.1 Limitations

6.2 Improvements

Chapter 7

Summary and Conclusions

Bibliography

[1] Elodie Morin, Mickael Maman, Roberto Guizzetti, Andrzej Duda, *Comparison of the Device Life-time in Wireless Networks for the Internet of Things*. IEEE Access, IEEE, 2017, 5, pp.7097 - 7114. 10.1109ACCESS.2017.2688279. hal-01649135.

[2] "What is lora." `https://www.semtech.com/lora/what-is-lora`, last visited 27.12.19.

[3] "What is lorawan." `https://lora-alliance.org/sites/default/files/2018-04/what-is-lorawan.pdf`, last visited 27.12.19.

[4] "Lora alliance." `https://lora-alliance.org`, last visited 27.12.19.

[5] "The things network." `https://www.thethingsnetwork.org`, last visited 27.12.19.

[6] B. Sousa, L. Cordeiro, P. Simões, A. Edmonds, S. Ruiz, G. A. Carella, M. Corici, N. Nikaein, A. S. Gomes, E. Schiller, T. Braun, and T. M. Bohnert, "Toward a Fully Cloudified Mobile Network Infrastructure," vol. 13, no. 3, pp. 547–563, 2016.

[7] F. Adelantado, X. Vilajosana, P. Tuset-peiro, B. Martinez, J. Melià-seguí, and T. Watteyne, "Understanding the Limits of LoRaWAN," no. September, pp. 34–40, 2017.

[8] "Lora vs other wireless technology." `https://www.semtech.com/uploads/images/LoRa_Why_Range.png`, last visited 27.12.19.

[9] "Lora signals from a low orbit satellite." `https://twitter.com/telkamp/status/956900631985475586?lang=en`, last visited 27.12.19.

[10] "Lorawan regional parameters." `https://lora-alliance.org/sites/default/files/2018-04/lorawantm_regional_parameters_v1.1rb_-_final.pdf`, last visited 27.12.19.

[11] "Ablut lorawan." `https://lora-alliance.org/about-lorawan`, last visited 27.12.19.

[12] "Lora alliance." `https://lora-alliance.org/sites/default/files/2019-05/lorawan_security-faq1.pdf`, last visited 27.12.19.

[13] "Lora / lorawan tutorial 12, modulation types and chirp spread spectrum." `https://www.mobilefish.com/download/lora/lora_part12.pdf`, last visited 27.12.19.

[14] "Lora / lorawan tutorial 13, symbol, spreading factor & chip." https://www.mobilefish.com/download/lora/lora_part13.pdf, last visited 27.12.19.

[15] P. Robyns, P. Quax, W. Lamotte, and W. Thenaers, "A multi-channel software decoder for the lora modulation scheme," pp. 41–51, 01 2018.

[16] "Sx1272/3/6/7/8: Lora modem, designer's guide." https://www.rs-online.com/designspark/rel-assets/ds-assets/uploads/knowledge-items/application-notes-for-the-internet-of-things/LoRa%20Design%20Guide.pdf, last visited 27.12.19.

[17] "Lora / lorawan tutorial 14, coding rate and forward error correction." https://www.mobilefish.com/download/lora/lora_part14.pdf, last visited 27.12.19.

[18] "Exploratory engineering, data rate and spreading factor." https://docs.exploratory.engineering/lora/dr_sf/, last visited 27.12.19.

[19] "An1200.22 lora modulation basics." http://wiki.lahoud.fr/lib/exe/fetch.php?media=an1200.22.pdf, last visited 27.12.19.

[20] "Lorawan specification 1.0.3." https://lora-alliance.org/sites/default/files/2018-07/lorawan1.0.3.pdf, last visited 27.12.19.

[21] "Pothosware." http://www.pothosware.com/#about, last visited 27.12.19.

[22] "About gnu radio." https://www.gnuradio.org/about/, last visited 27.12.19.

[23] "Gnu radio oot module implementing the lora phy, based on https://github.com/matt-knight/research/tree/master/2016_05_20_jailbreak." https://github.com/BastilleResearch/gr-lora, last visited 27.12.19.

[24] "Gnu radio blocks for receiving lora modulated radio messages using sdr." https://github.com/rpp0/gr-lora, last visited 27.12.19.

[25] "Grcon16 - reversing and implementing the lora phy with sdr, matt knight." https://www.youtube.com/watch?v=-YNMRZC6v1s, last visited 27.12.19.

[26] "Arduino-based library for dragino lora shield v1.4 https://github.com/arduino-org/arduinolibrary-lora-node-shield, last visit."

[27] M. Coates, A. Hero, R. Nowak, and B. Yu, "Internet tomography," May 2002. http://www.qqq.com, 15.12.2019.

[28] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, ninth dover printing, tenth gpo printing ed.

Abbreviations

AAA	Authentication, Authorization, and Accounting

Glossary

Authentication

Authorization Authorization is the decision whether an entity is allowed to perform a particular action or not, e.g. whether a user is allowed to attach to a network or not.

Accounting

List of Figures

List of Tables

Appendix A

Installation Guidelines

For Docker, see C.1.1. For manual installation, see C.2.1

Appendix B

Contents of the CD

Appendix C

README.md

Below is the README.md file converted to LaTeX. It is recommended to view the file in a markdown viewer e.g. VS Code or on GitHub `https://github.com/mustard123/master-thesis`

C.1 C-RAN for LoRa

An arduino with a LoRa shield sends out packets over the air in an interval. Some packets require an acknowledgment (ACK). If an ACK is required, the arduino waits for a certain amount of time for the ACK. If the ACK arrives in time, the arduino starts transmitting the next packet. If not, the arduino will resend the packet and again wait for the ACK.

The RRH (Remote Radio Head) receives radio waves with a LimeSDR. The RRH streams the IQ samples over the network the the BBU (Base Band Unit).

The BBU decodes the message. If the message says it require and ACK, the BBU send out IQ samples of the ACK message over the network to the RRH which transmits them back over the air to the arduino.

C.1.1 Run with Docker

1. Clone the repo
2. Go to the docker directory

Info

- The container run in privileged mode to easily access plugged in USB devices

- The container run in network mode host (No NAT or Bridge has to be considered). This means the containers have the ip address of the host machine. If RRH and BBU run on different machines, find out their respective IP with *ifconfig* and pass the address as arguments in the docker-compose.yml, see below.

C.1.2 RRH

In the RRH directory run:

```
docker-compose up
```

This starts the Remote Radio Head. The RRH looks for a LimeSDR, it prints errors if it cannot find one. You can plug one in after the container has started and it should get detectet. By default it uses the first LimeSDR it can find.

Parameters

There are various parameters which you can specify in the *docker-compose.yml* file.

Run this to see what the possible params are:

```
./zero_mq_split_a.py -h
```

Output:

```
Usage: zero_mq_split_a.py: [options]

Options:
  -h, --help            show this help message and exit
  --RX-device-serial=RX_DEVICE_SERIAL
                        Set RX_device_serial [default=]
  --TX-device-serial=TX_DEVICE_SERIAL
                        Set TX_device_serial [default=]
  --capture-freq=CAPTURE_FREQ
                        Set capture_freq [default=868.5M]
  --samp-rate=SAMP_RATE
                        Set samp_rate [default=1.0M]
  --zmq-address-iq-in=ZMQ_ADDRESS_IQ_IN
                        Set zmq_address_iq_in [default=tcp://127.0.0.1:5052]
  --zmq-address-iq-out=ZMQ_ADDRESS_IQ_OUT
                        Set zmq_address_iq_out [default=tcp://*:5051]
```

Param	Explanation
RX-device-serial	By default, the program will use the first LimeSDR it can find for receiving and transmitting signal. If you have two devices you can specify which should receive by passing the device Serial (See section **Help** for more info)
TX-device-serial	By default, the program will use the first LimeSDR it can find for receiving and transmitting signal. If you have two devices you can specify which should transmit by passing the device Serial (See section **Help** for more info)
capture-freq	The frequency in Hz at which the RRH listens for signals. Default value is 86850000

Param	Explanation
samp-rate	How many samples per second. Default value is 1000000. Must be at least double the bandwidth of the expected signal see *Nyquist-Shannon principle*
zmq-address-iq-in	ZMQ address to which the RRH subscribes to receive an IQ samples stream (from the BBU) to then send out (TX). Default value is tcp://127.0.0.1:5052 meaning the IQ samples are expected to come from localhost on port 5052. Normally RRH and BBU are on different devices but on the same network

Param	Explanation
–zmq-address-iq-out	ZMQ address on which the RRH streams out the IQ samples (to the BBU) it receives (RX). Default is tcp://*:5051 meaning it publishes the stream on all interface on port 5051

C.1.3 BBU

The BBU has two components: * LoRa_Decoder: receives a stream of IQ samples from the RRH, decodes the LoRa signal and sends the decoded message out on a UDP socket * LoRa_Responder: receives the messages from that UDP socket and, depending on message content, streams response IQ samples to the RRH or does not give a response

In the BBU directory run:

```
docker-compose up
```

This starts both components of the BBU

Params

The LoRa_Decoder has the following params:

```
Usage: zero_mq_split_b.py: [options]

Options:
  -h, --help            show this help message and exit
  --bandwidth=BANDWIDTH
                        Set bandwidth [default=125000]
  --capture-freq=CAPTURE_FREQ
                        Set capture_freq [default=868.5M]
  --decoded-out-port=DECODED_OUT_PORT
                        Set decoded_out_port [default=40868]
```

```
--samp-rate=SAMP_RATE
                        Set samp_rate [default=1.0M]
--spreading-factor=SPREADING_FACTOR
                        Set spreading_factor [default=12]
--zmq-address-iq-in=ZMQ_ADDRESS_IQ_IN
                        Set zmq_address_iq_in [default=tcp://127.0.0.1:5051]
```

Param	Explanation
bandwith	The bandwidth in Hz of the LoRa signal. Default is 125000.
capture-freq	The frequency in Hz of the LoRa signal. The RRH of course must also listen on this frequeny. Default is 868500000.
decoded-out-port	On which port the decoded messages will be sent out. Localhost only. The LoRa_Responder needs to be configured to listen on this port. Default is 40868.
samp-rate	How many samples per second to expect from the RRH. Default is 1000000
spreading-factor	The spreading factor of the incoming LoRa signal. From [7-12] inclusive. Default is 12

Param	Explanation
–zmq-address-iq-in	ZMQ address to which the BBU subscribes to receive an IQ samples stream (from the RRH) to decode. Default value is tcp://127.0.0.1:5051 meaning the IQ samples are expected to come from localhost on port 5051. Normally RRH and BBU are on different devices but on the same network

---

The LoRa_Responder has the following params:

```
usage: lora_socket_server.py [-h] [-o OUT_PORT] [-i INPUT_PORT]

Connect to udp port for receiving decoded LoRa signals, if an ACK is required
publish ACK iq samples via zmq socket for Remote Radio Head to receive and
send out (TX).

optional arguments:
  -h, --help            show this help message and exit
  -o OUT_PORT, --out-port OUT_PORT
                        zmq port to publish downstream (i.e ACK) iq samples
                        (default: 5052)
  -i INPUT_PORT, --input-port INPUT_PORT
                        UDP port to connect for receiving decoded lora
                        messages (default: 40868)
```

Param	Explanation
out-port	Publish the response IQ samples on all interface on this port. Default is 5052. (The response is 3 bytes long (“ACK”) and SF 12. This is hardcoded for now)
input-port	UDP port to receive the decoded messages sent by the LoRa_Decoder. Default is 40868

To pass the parameters you have to specify them in the docker-compose.yml file.

Example:

To have the LoRa_Decoder send the decoded messages out on port 30300 and the Lora_Responder to listen on port 30300 accordingly pass the arguments like below to the respective command field:

docker-compose.yml

```
version: '3'
services:
        lora_decoder:
                build: ./LoRa_Decoder
                network_mode: host
                tty: true
                command: ["--decoded-out-port", "30300"]
        lora_responder:
                build: ./LoRa_Responder
                network_mode: host
                tty: true
                command: ["--input-port", "30300"]
```

C.1.4 LimeSDR

- Plug in the antennas on the LimeSDR board on *RX1_L* and *TX1_1*

C.1.5 Help

- LimeSDR calibration/gain error:
- Download LimeSuite Toolkit to calibrate the LimeSDR
- LimeSDR find device serial:
- With LimeSuite installed run *LimeUtil –find*
- Or run *lsusb -v* and look for the LimeSDR device

C.2 Arduino

The arduino-lmic library is required Instructions here

1. Go to the arduino directory.
2. Compile and upload the code to the arduino
3. The arduino runs the protocol in the manner described at the beginning.
4. It send packets with SF12 and expects the ACK response to be SF12 as well.
5. After 3 packets the arduino has finished.
6. Look at the Serial output for details. Baud rate 9600

Info

PlatformIO was used to compile and upload the image to the arduino.

C.2.1 Manual installation Ubuntu

Visit this guide for installing LimeSDR Plugin for GNU Radio for more detail. This guide only has the short version.

Install dependencies for signal processing:

```
sudo apt-get update && sudo apt-get install -y gnuradio=3.7.11-10 libboost-all-dev swig
libcppunit-1.14-0 libfftw3-bin libvolk1-bin liblog4cpp5v5 python libliquid1d libliquid-d
&& pip install numpy && pip install scipy
```

Install LimeSuite

```
sudo add-apt-repository -y ppa:myriadrf/drivers && sudo apt-get update \
&& sudo apt-get install -y limesuite liblimesuite-dev limesuite-udev limesuite-images \
soapysdr-tools soapysdr-module-lms7
```

Clone and install LimeSDR Plugin for GNU Radio:

```
git clone https://github.com/myriadrf/gr-limesdr && cd gr-limesdr && mkdir build && cd b
```

Clone and install rpp0's LoRa decoder for gnuradio

```
git clone https://github.com/rpp0/gr-lora.git && cd gr-lora && git checkout b1d38fab9032
&& mkdir build && cd build \
&& cmake .. && make && sudo make install \
&& cd .. && rm -rf build \
&& git checkout -b encoder origin/encoder && git checkout 3c9a63f1d148592df2b715496c67cc
&& mkdir build && cd build \
&& cmake .. && make && sudo make install && sudo ldconfig
```

With pip for python2 install the zmq package:

```
pip install pyzmq==18.1.0
```

Then open the *zero_mq_split_a.grc* and the *zero_mq_split_b.grc* file in the docker/RRH directory resp. in the docker/BBU/LoRa_Decoder directory. Or run the *zero_mq_split_a.py* resp. the *zero_mq_split_b.py* script in those directories with your shell. Also run the *lora_socket_server.py* sript inside docker/BBU/LoRa_Responder with your shell.

C.3 Tools

In the tools directory in the Encode and Decode directory are multiple usefuls scripts for encoding and decoding lora without gnuradio

1. First, after you recorded a signal trim the signal with a tool like audacity. Else if you want to visualize it with plot_signal.py the signal is shrunk too much to make it fit in the plot.

2. After trimming, channelize the signal else the decoder cannot properly decode the signal. Run channelizer.py -h to see the options. It takes an signal recording via the –input-file option and outputs the channelized file as “channelized.raw”. Don’t forget to specify bandwidth and sample rate if they differ from the set default values.

3. The channelized signal can the be passed to the decoder. The decoder prints out the decoded signal and generates a csv file (words.csv) containing the words at each sample. Don’t forget to specify bandwidth and sample rate etc if they differ from the set default values.

4. This csv file can be passed to plot_signal.py which draws the signal and the words in the csv file to a pdf (rawframe.pdf). Don’t forget to specify bandwidth and sample rate if they differ from the set default values.

Use the encoder to generate samples for the test_packet[] uint8 array in the code. The samples are written to the fiel “output.bin”

Use the two scripts decoder_build.sh and encoder_build.sh to compile the encode.cc and decode.cc files.

Use VsCode to open the directory “Encode and Decode” to have predefiend debug configurations. The folder ‘.vscode’ has been commited in this repo.

All recorded uplink signals have been recorded with sample rate 1Million and transmitted with a bandwidth of 125’000

The decoder only works for signals with an explicit header.