University of Zurich^UZH

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

The topic of this thesis is the design and implementation of a Cloud Radio Access Network (C-RAN) architecture for Long Range (LoRa) networks. LoRa is a popular modulation scheme based on the chirp spread spectrum modulation technique. It is used in Low Power Wide Area Networks (LP-WANs) for Internet of Things (IoT). Traditionally, in a LoRa Wide Area Network (LoRaWAN), gateways receive signals on the uplink from end-devices to demodulate and decode the analog signal directly on the gateway. On the downlink, gateways receive a digital message from the network server and modulate and encode this message as an analog signal to send it over the air as radio waves to end-devices. The gateways, however, can be simplified by moving the demodulation and decoding in the receiver (resp. modulation and encoding in the sender) from the gateway to general purpose hardware running a software Base-Band Unit (BBU). This leaves the gateway as a Remote Radio Head (RRH) implementing only limited capabilities, while the BBUs can be centralized and virtualized on a remote server. In this thesis, the C-RAN architecture is implemented with the help a Software Defined Radio (SDR) for handling radio signals as well as Docker for the virtualization of BBUs. To design and implement the LoRa C-RAN, this thesis studies the network requirements and the effects of network and processing delay on the C-RAN. Finally, this work also also introduces a software modulator and encoder for a LoRa to emit signals that can be received by real hardware on the downlink.

Thema dieser Arbeit ist das Design und die Implemtierung einer Cloud Radio Access Networks (C-RAN) Architektur für LoRa. LoRa ist eine weit verbreite Modulierungstechnik basierend auf dem Chirp Spread Spectrum (CSS) Modulierungsverfahren. Es wird of benutzt in Low Power Wide Area Networks für Internet of Things Geräte. Herkömmlicherweise empfangen Gateways in LoRa Wide Area Networks ein uplink Signal von Endgeräten und demodulieren und dekodieren das Signal auf dem Gateway selbst. Für downlink Signal empfangen die Gateways eine digitale Nachricht von einem Netzwerk Server. Diese Nachricht wird dann von den Gateways moduliert und enkodiert als ein anologes Signal, dass dann über die Luft als Radiowellen zu den Endgeräten gesendt wird. Die Gateways können jedoch vereinfacht werden indem das Demodulieren und Dekodieren, bzw. das Moduliern und Enkodieren in Software implementiert wird welche auf Allzweck Hardware läuft. Dies ist die Baseband Unit (BBU). Das gateway fungiert dann lediglich noch als Remote Radio Head (RRH). Die BBU's können auf einem Server zentralisiert und virtualiesert werden. Die C-RAN Architektur in dieser Arbeit ist mit einem Software Defined Radio (SDR) und mit Docker für die Virutalisierung der BBUs implementiert. Desweiteren untersucht diese Arbeit Netzwerk Anforderungen und die Auswirkungen von Netzwerk -und Verarbeitungsverzögerung auf das C-RAN. Ausserdem stellt diese Arbeit einen Softwaremodulator für downlink LoRa signal vor die von echter Hardware empfangen werden können.

Acknowledgments

I would like to first thank Dr. Eryk Schiller for advising me in this project and Prof. Stiller, who is the head of Communication Systems Group, to welcome me in his lab and allow me to run this work under his supervision. Eryk's office door was always open. He provided me with meaningful feedback, whenerver I ran into a trouble or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction when I needed it.
Secondly, I owe a debt of gratitude to my parents and family for providing me with unfailing support and continuous encouragement throughout the years of my study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you

Silas Weber

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 power requirements. Battery powered LPWAN devices can run for years. For instance, a node sending 100B once a day lasts for 17 years [?]. LoRa (short for Long Range) is a spread spectrum modulation technique, a wireless radio frequency (RF) technology for long-range and low power platforms and has become the de facto technology for IoT networks worldwide [?]. 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 [?]. 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 notable LoRa Alliance members [?]. The Things Network (TTN), 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 network's reach. At the time of writing, TTN has 95'208 members, 9'786 gateways, and is present in 147 countries [?]. LoRaWAN operates in the unlicensed Industrial, Scientific and Medical (ISM) radio bands. Therefore no government license is required to operate LoRa devices and gateways. This allows hobbyists, 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 be scheduled on the network server which 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 the technological stack on the gateway can be reduced by running the signal processing functions not on the gateway itself but in a cloud / centralized environment on general purpose hardware. Such a Cloud / Centralized Radio Access Network (C-RAN) has been previously shown to be beneficial in the 3rd Generation Partnership Project (3GPP) Long Term Evolution (LTE) [?]. 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. This is called the Remote Radio Head (RRH). On the upstream, the RRH receives LoRa radio signals which it converts into an In-phase and quadrature (I/Q) sample stream with the ADC and simply forwards them to the cloud signal processing unit over Ethernet. This virtualized component is the Baseband Unit (BBU). On the downstream the BBUt streams a LoRa signal as I/Q samples to the RRH which converts it with the DAC to an analog signal and propagates it out over the air. Signals are encoded and decoded on the BBU. There are many advantages in such a setup, but they come at a cost. The 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 BBU. Streaming the I/Q samples between RRH and BBU 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 traditional setup. Extracting part of the LoRa gateways and replacint it with software also brings the advantages of setting the base for Software Defined Networking (SDN) and Network Function Virtualization (NFV) by centralizing the BBUs that were before distributed on the individual gateways. The goal of this work is setting up a C-RAN architecture for LoRa by simplifying the gateways as described above and moving the signal processing out of the gateway into software component that can be virtualized with for example 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 C-RAN for LoRa, gives an architectural overview as well as the implementation details. It evaluates the architectural and network related requirements by conducting an experiment. We developed a simple protocol in raw LoRa, meaning not compliant with the LoRaWAN standard. In this protocol physical IoT device has a queue of packets to transmit. Then depending on wether it requires an acknowledgment, it waits for a few seconds for a response or just transmit the next packet in the queue in an interval. If the packet requires to be acknowledged but no acknowledgment is received, the same packet will put as first item in the queue. We investigate network utilization and effects of network delay and processing delay on this setup.
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 a various degree of success. They all focused first on decoding LoRa signals transmitted by a real LoRa hardware. For the C-RAN 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 extended an existing experimental uplink signal encoder with the ability

to also generate downlink signals in software.

1.2 Thesis Outline

The rest of the thesis is structured the following way. In chapter 2 an introduction to LoRa and LoRaWAN is given and where it fits compared to other wireless technology. Then a more in depth explanation is given for LoRa signals, such as the modulation scheme and key factors like spreading factor coding rate and packet structure.

In the following chapter and overview of current software defined radio implementations for LoRa is given, how they are implemented as well as their level of sophistication. In this context the GNU Radio framework is introduced as well.

In chapter 4 the C-RAN architecture for cellular networks is introduced to show what steps are needed to move from a traditional architecture to a C-RAN architecture and what benefits it can bring.

The next chapter continues with the main part of the thesis, namely the C-RAN for LoRa. The architectural overview is given as well as the implementation details for all involved components. Then the C-RAN experiment is described where network utilization, network delay and processing delay are investigated. At the end of the chapter the results are presented.

Chapter 6 presents some tools that were developed during this thesis for encoding and signal visualization that may be helpful for further developing LoRa processing in software.

This ties directly into the second to last chapter where the future work for a C-RAN for LoRa may lead to. It also lists some limitations of this current C-RAN architecture that can be improved and further developed.

Finally, the last chapter summarizes the key aspects and concludes this work.

Chapter 2

LoRa and LoRaWAN

LoRa is a modulation technique derived from chirp spread spectrum technology[?]. Originally developed by Cycleo, a french company, LoRa has been acquired by Semtech [?]. 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 signals 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 a 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 [?].

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 an experiment during a TTN conference, LoRa signals from a low orbit satellite were received [?]. 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 [?].

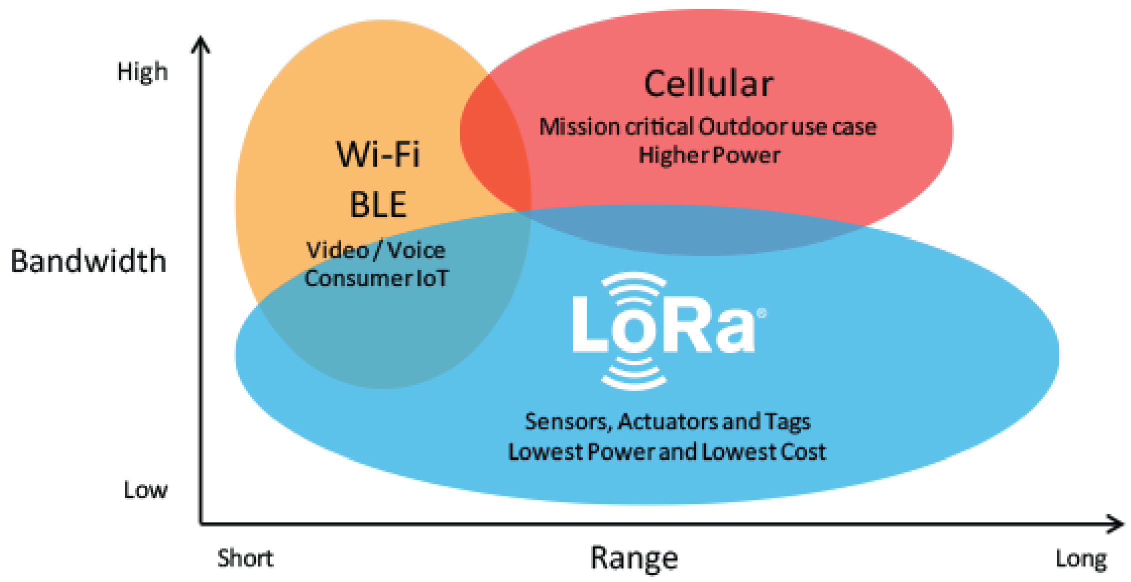

High
Bandwidth
Low
Short
Range
Long
Wi-Fi
BLE
Video / Voice
Consumer IoT
Cellular
Mission critical Outdoor use case
Higher Power
LoRa
Sensors, Actuators and Tags
Lowest Power and Lowest Cost

Figure 2.1: LoRa vs other wireless technology[?]

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 [?].

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 [?]. 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 message to the network server which does the complex things such as filtering redundant packages, security checks, forwarding the messages to the right application server etc. [?]. 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.

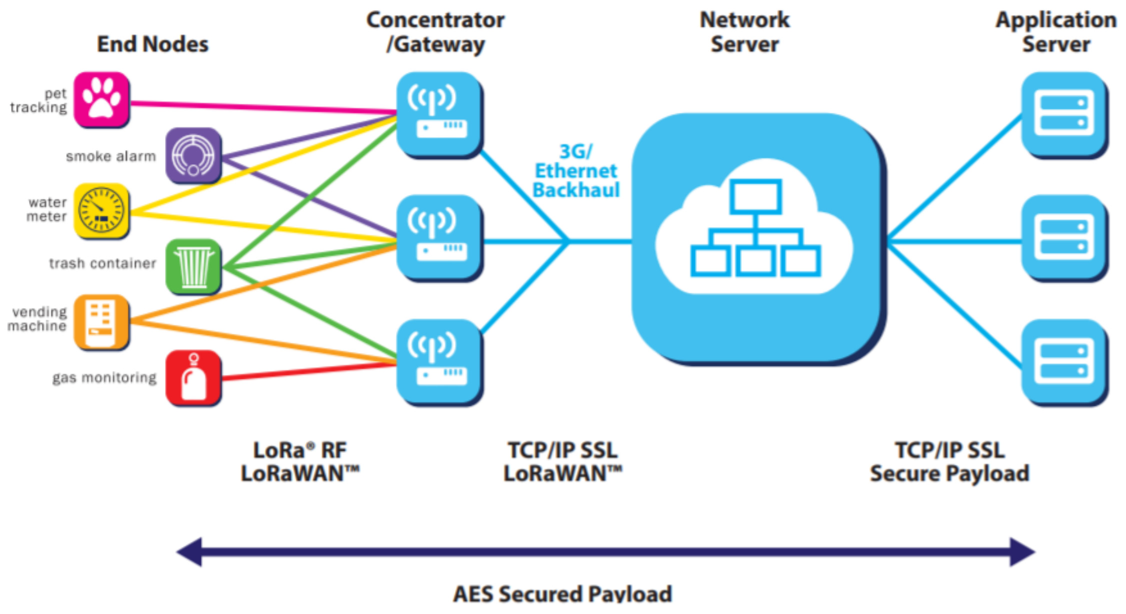


Figure 2.2: LoRaWAN network architecture [?]

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 asset 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 [?]. With those measures LoRaWAN prevents eavesdropping. Spoofing is prevented by a MIC (Message Integrity Code) in the MAC payload, and replay attacks are prevented by utilizing frame counters [?].

2.2 End-node Classes

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

- 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 signal bandwidth is 125 kHz with one exception. 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.

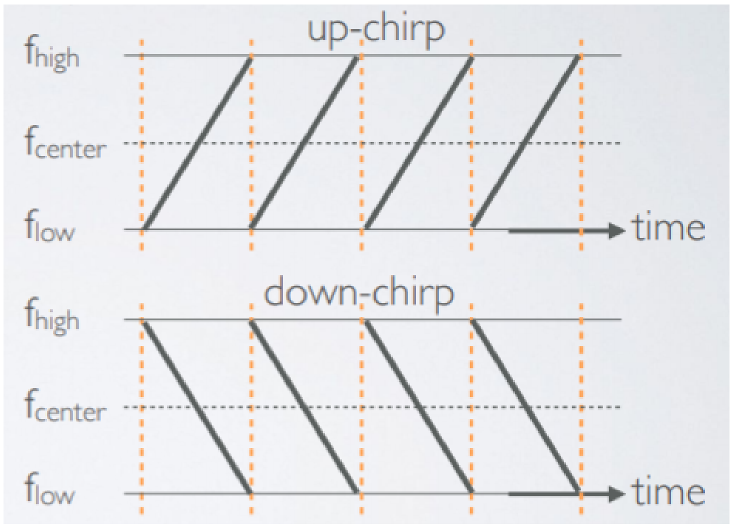


Figure 2.3: Up- and down chirps [?]

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.

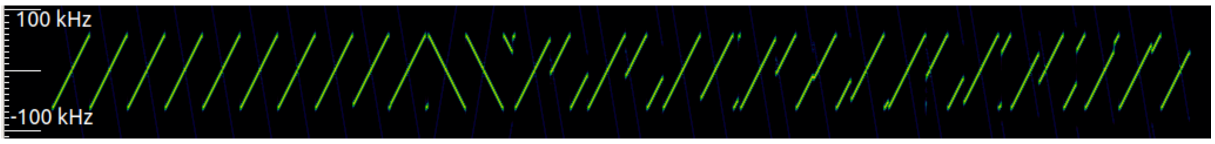


Figure 2.4: Own recording of uplink transmission by arduino equipped with a LoRa shield

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) [?]. 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.

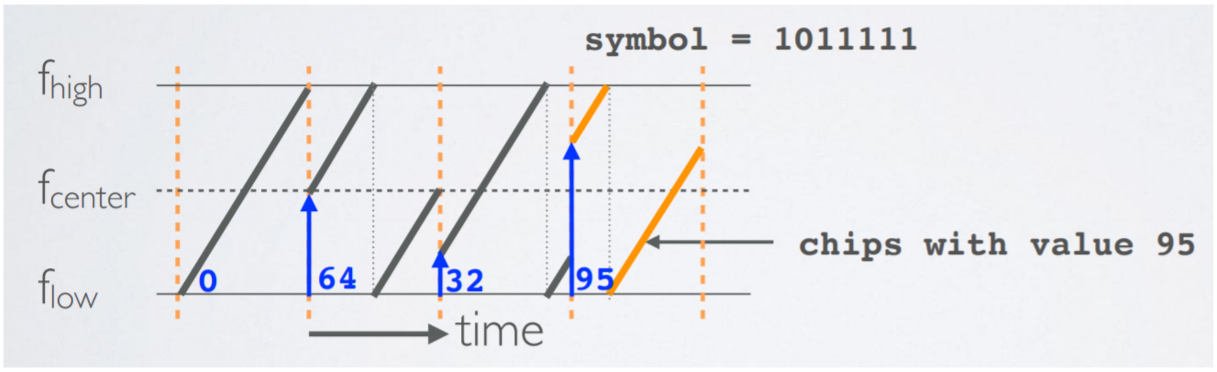


Figure 2.5: Chips and symbols value [?]

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

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

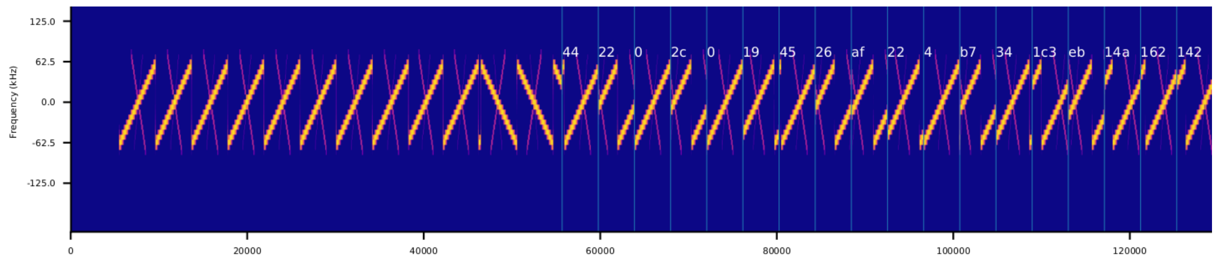


Figure 2.6: Running the signal through our toolchain, matching symbols with samples

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[?, ?]. 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 in implicit header mode where 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 [?].

The symbol time is defined in the LoRa Design guide by $T_{sym} = \frac{2^{SF}}{BW}$ [?]. 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 [?]. 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 is used and 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. Let's assume there are multiple devices near a gateway that transmit with SF 12. This occupies the bandwidth for devices 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 [?]:

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

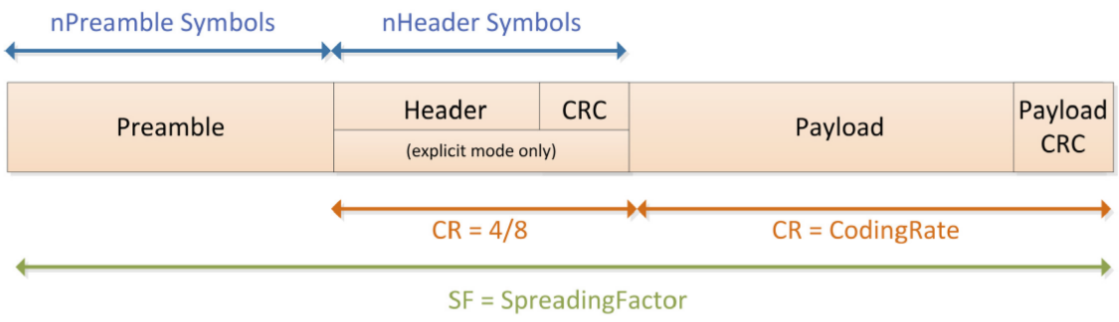


Figure 2.7: LoRa packet structure [?]

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

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 shows the structure of an uplink packet. 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 2.9 shows.

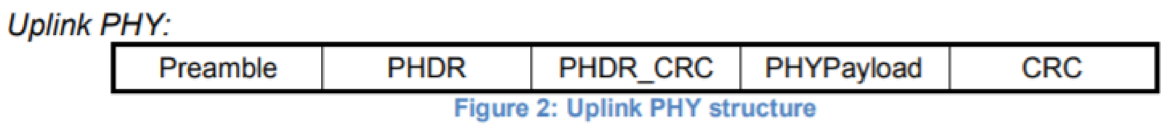


Figure 2.8: LoRa uplink packet structure [?]

2.4 LoRa signal (downlink)

LoRa downlinks signals are the inverse of an uplink signal. This lets end-devices or gateways easily differentiate between uplink and downlink. An Arduino end-device can then ignore all uplink signals as it will only ever expect a downlink signal it may need to process. Gateways on the other hand can ignore downlink signals as they are only interested in uplink signals. Figure 2.10 shows a downlink signal with SF 12 and coding rate 4/5 sent from LoRa gateway. It encodes the payload "ACK". A difference is that

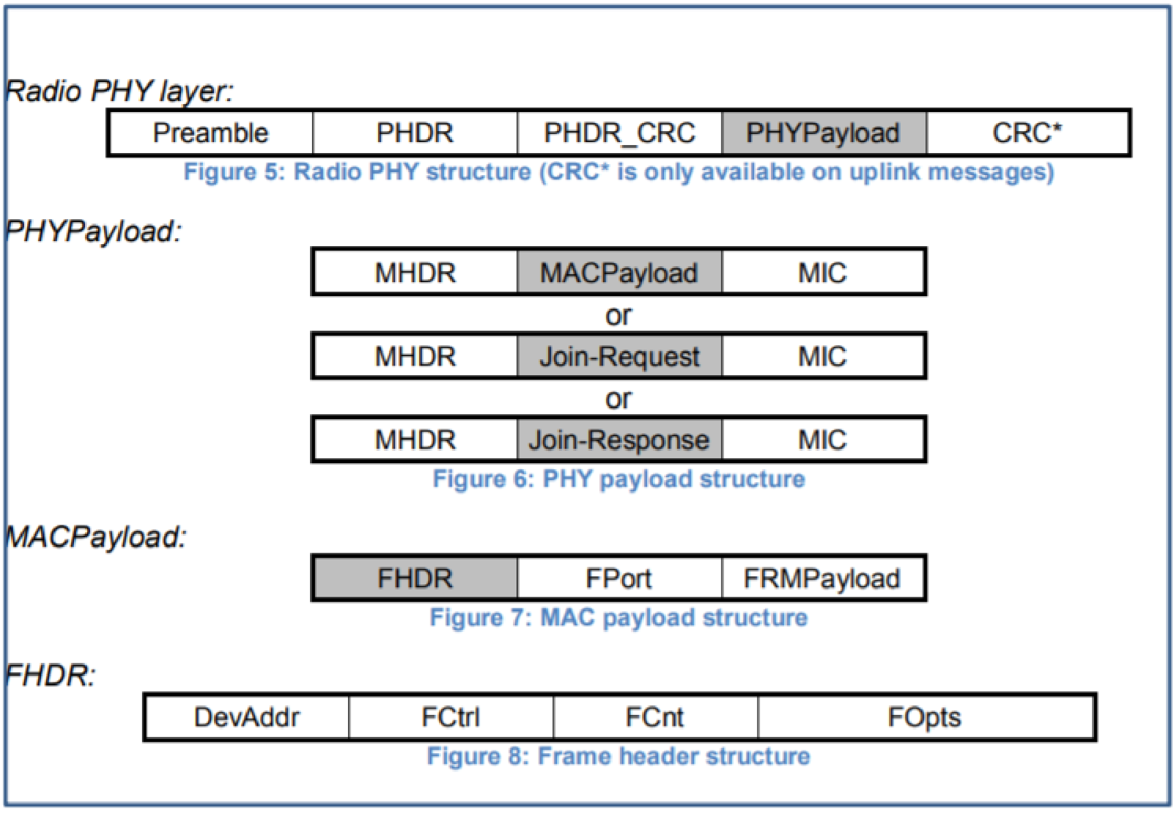


Figure 2.9: LoRaWAN packet [?]

downlinks signals do not have a CRC after the payload as shown in Figure2.11.

The 10 preamble chirps that are up chirps in an uplink signal are now down chirps.

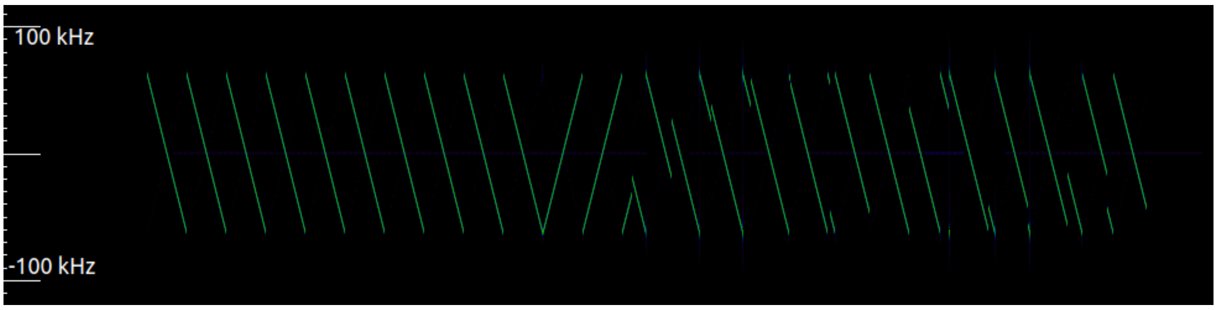
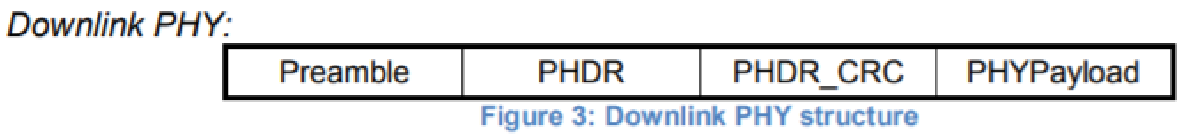


Figure 2.10: LoRaWAN packet [?]



Downlink PHY:

Preamble	PHDR	PHDR_CRC	PHYPayload

Figure 3: Downlink PHY structure

Figure 2.11: LoRa downlink packet structure [?]

The 2 start of frame delimiter chirps that are down chirps in an uplink signal are now up chirps. The question remains if the data chirps are flipped as well. In fact, the data chirps are flipped as well, as we have discovered. This will be elaborated on more in chapter 6.

Chapter 3

LoRa in SDRs

Software-defined radios (SDR) use components that are usually implemented in hardware. The most popular signal processing framework 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`

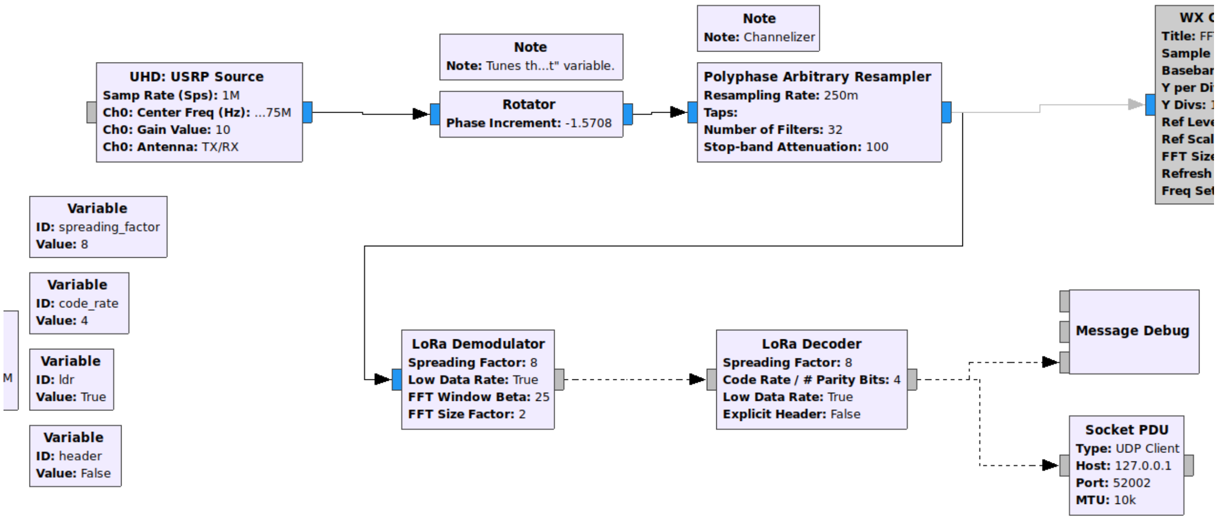
We tried Blum's implementation in the Pothos first. The Pothos projects is an open source data-flow framework supporting SoapySDR, a general framework for enabling SDR devices [?]. Unfortunately the LoRa modem demo application did not work for us at all. After spending a few unsuccessful days trying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

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 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 an 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 C-RAN implementation for LoRa on this module.


Note
Note: Channelizer

Note
Note: Tunes th...t" variable.

UHD: USRP Source
Samp Rate (Sps): 1M
Ch0: Center Freq (Hz): ...75M
Ch0: Gain Value: 10
Ch0: Antenna: TX/RX

Rotator
Phase Increment: -1.5708

Polyphase Arbitrary Resampler
Resampling Rate: 250m
Taps:
Number of Filters: 32
Stop-band Attenuation: 100

WX C
Title: FF
Sample
Basebar
Y per Di
Y Divs: 1
Ref Leve
Ref Scal
FFT Size
Refresh
Freq Set

Variable
ID: spreading_factor
Value: 8

Variable
ID: code_rate
Value: 4

Variable
ID: ldr
Value: True

Variable
ID: header
Value: False

M

LoRa Demodulator
Spreading Factor: 8
Low Data Rate: True
FFT Window Beta: 25
FFT Size Factor: 2

LoRa Decoder
Spreading Factor: 8
Code Rate / # Parity Bits: 4
Low Data Rate: True
Explicit Header: False

Message Debug

Socket PDU
Type: UDP Client
Host: 127.0.0.1
Port: 52002
MTU: 10k

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

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 difficult, some of the documentation is a lie as the PHY is not implemented in the way it is described, see Knight [?]. 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 operations which have to be reverse engineered [?].

Robyns et al. identified and implemented the following seven steps in their receiver to

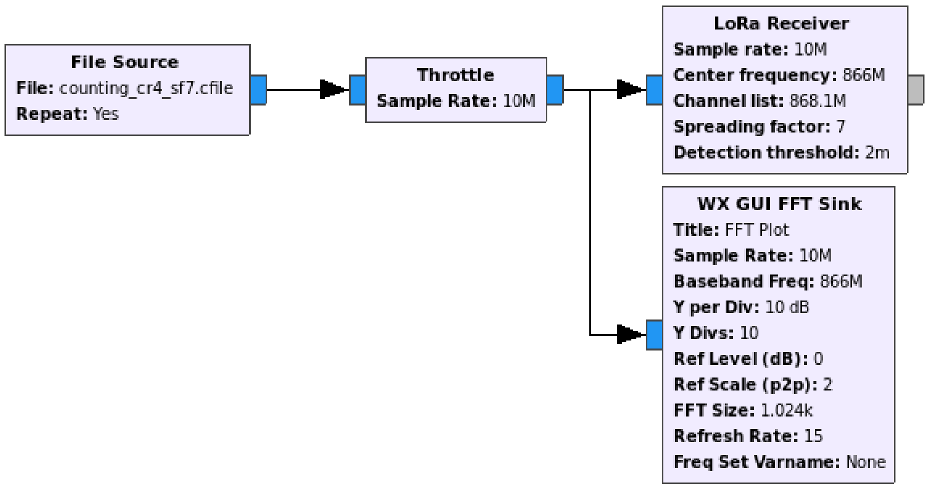


Figure 3.2: Single LoRa Receiver block (top right) [?]

receive and decode a LoRa signal: detection, synchronization, demodulation, deinterleaving, dewhitening, decoding, and packet construction [?]. 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 in 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 cellular networks

A Radio Access Network (RAN) provides the connection between and end-device e.g. a mobile phone and the core network e.g network of the telecom provider. A RAN provides access and manages resources across the radio sites. It is a major component in telecommunications. RAN components include a base station and antennas that cover a specific region. The base stations connect to the core network [?].

4.1 D-RAN

Distributed Radio Access Network (D-RAN) is the present mode of operation for many mobile network operators. In a D-RAN, the 4G radio at the site tower consists of a Baseband Unit (BBU) on the ground and a Remote Radio Head (RRH) at the top as Figure 4.1 shows. RRH and BBU are connected via the Common Protocol Radio Interface (CPRI). The BBUs in each tower are connected via ethernet to the Mobile Switching Center (MSC) [?].

4.2 Moving to C-RAN

In a Cloud / Centralized RAN (C-RAN) the BBU in each tower site are move into a centralized place, the BBU hotel, see Figure 4.2. This new architectural setup results in saving

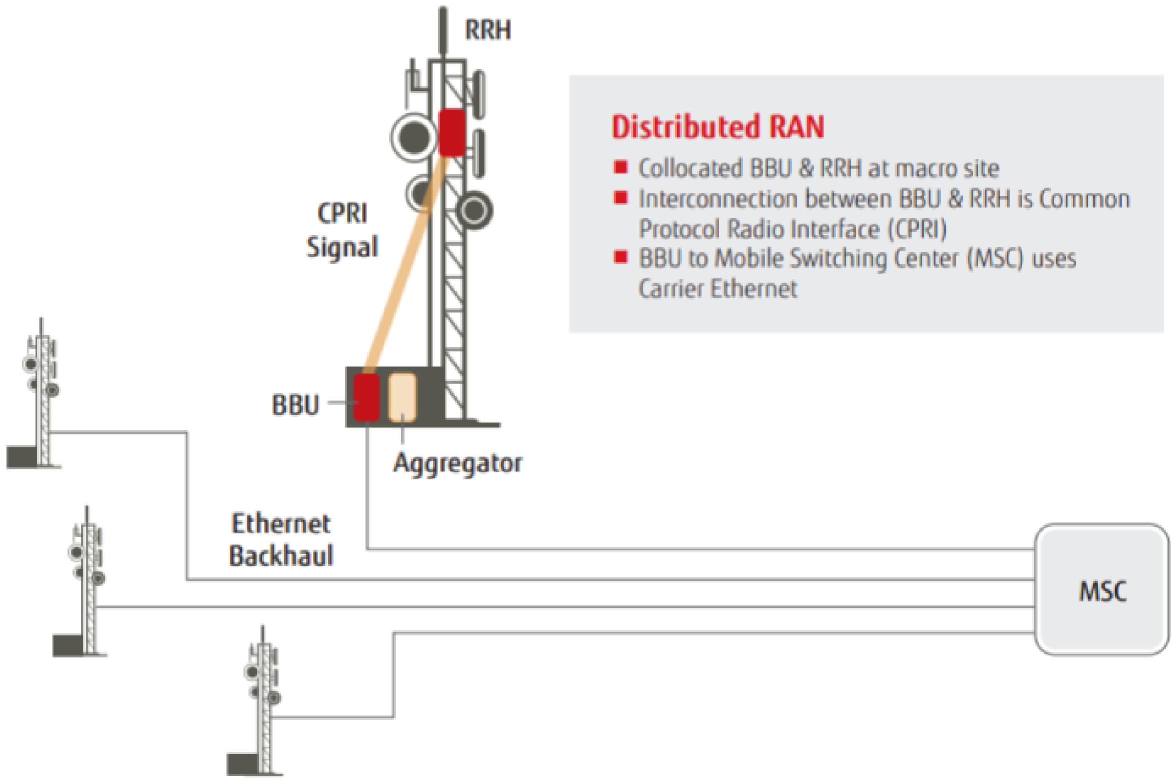

RRH
CPRI Signal
BBU
Aggregator
Ethernet Backhaul
MSC
Distributed RAN
Collocated BBU & RRH at macro site
Interconnection between BBU & RRH is Common Protocol Radio Interface (CPRI)
BBU to Mobile Switching Center (MSC) uses Carrier Ethernet

Figure 4.1: Distributed cell towers with each with a RRH and BBU [?]

costs in capital expenditures (CAPEX) as well as operational expenditures (OPEX). The many small routers in the cell towers can be replaced by one in the BBU hotel. Deployment, maintenance and scaling can be expedited as all the BBU are centralized. Also this enables the BBU for Network Function Virtualization (NFV) and the RAN for Software Defined Networking (SDN) [?].

4.3 Virtual RAN

In virtual RAN the execution environment is abstracted i.e. virtualized. Radio function applications operate in a virtualized environment and interact with physical resources directly or through hardware emulation. Such a cloud environment enables RAN as a service [?]. For C-RAN this means instead of having a separate standalone physical device for each BBU in the BBU hotel, the BBUs now can run on a single physical server with each BBU in a virtualized environment.

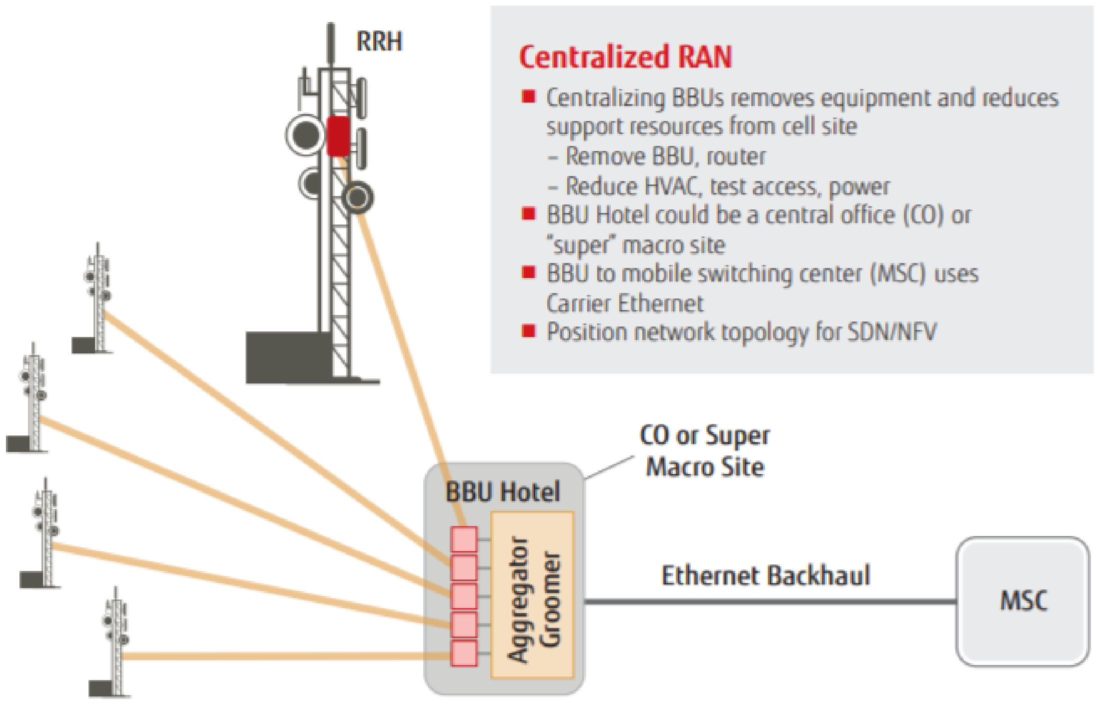

RRH
Centralized RAN
■ Centralizing BBUs removes equipment and reduces support resources from cell site
– Remove BBU, router
– Reduce HVAC, test access, power
■ BBU Hotel could be a central office (CO) or "super" macro site
■ BBU to mobile switching center (MSC) uses Carrier Ethernet
■ Position network topology for SDN/NFV
CO or Super Macro Site
BBU Hotel
Aggregator Groomer
Ethernet Backhaul
MSC

Figure 4.2: BBUs are centralized in the BBU hotel [?]

Nikaein et al. describe two main steps for realizing a C-RAN namely:

- **Commoditization and Softwarization** which essentially is the implementation of network and RF function in software.
- **Virtualization and Cloudification** which is the execution of network functions and management of physical resources by a cloud OS [?].

Nikaein et al. found that containers e.g. Docker proved to be more adequate in their RAN as a service experiment as they offer near bare metal performance and provide direct access to the RF hardware. Virtual machines in, particular KVM, also gave them good performance, but require low latency mechanisms to access I/O resources.
For the C-RAN for LoRa, described in chapter 5, Docker is used for the virtualization layer.

Chapter 5

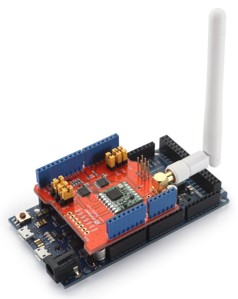
C-RAN for LoRa

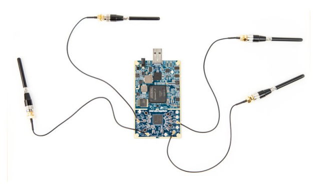
5.1 Goal

The goal is to set up a minimal working environment for a LoRa C-RAN. The gateway's functionality should separated into an RRH and BBU component and the BBU component should be virtualized and run on general purpose hardware. A simple network server should process the uplink message and schedule a response if required. From this setup basic network requirements can be derived and measured as well as costs estimated.

5.2 Hardware

An Arduino Mega 2560 with the Dragion LoRa Shield v1.4 is the end-device, see Figure 5.1a. A LimeSDR mini, Figure 5.1b, serves as the receiver for the RRH. The Raspberry Pi, Figure 5.1c with a LoRa hat was used for testing up -and downlink signals, but not for the experiment itself. Its hat is the iC880A LoRaWAN concentrator for the 868MHz frequency.


(a) Arduino with LoRa shield


(b) LimeSDR mini

(c) Raspberry Pi with LoRa hat

Figure 5.1: Hardware devices

5.3 Sending Uplink Signals

Uplink signals are sent with an Arduino device equipped with a LoRa shield. The Arduino is controlled with an adapted form of the IBM LoraMAC-in-C (LMIC) library, modified to run on Arduino devices. Using this library, we implemented a simple communication protocol where a queue of packets is sent out in an interval. See section 5.6.3 for detailed information.

5.4 RRH and BBU

For splitting up the LoRa gateway's functionality we use two laptops. One laptop has the LimeSDR mini plugged in and serves as the RRH. The other decodes the LoRa signal in software, processes the signal and also generates LoRa signals in software and sends those downlink I/Q samples back to the RRH. The processes on the second laptop run in virtualized environment with Docker, more specifically Docker compose, Docker's orchestration tool.
The LoRa OOT module by Robyns et al. has a branch called ”encoder” where they began the implementation of modulating an uplink LoRa signal in software. It is able to generate a specific test packet but the modulated signal has errors as we saw when we inspected the data payload on the LoRa gateway. Having an uplink signal generator was a nice starting

point, but we needed something to generate downlink signal. In the end we extended the existing implementation by adding a downlink signal generation ability, see section 6. Based on these considerations, as a first workaround we set up a private LoRaWAN network, scheduled a downlink, and recorded it to a file. Now we can stream that file as a response by streaming its content, which are I/Q samples, to the RRH.

5.5 Architecture

5.5.1 Overview

This section aims to first provide a high level overview of the architecture and then give a more detailed architectural overview of each component. Figure 5.2 shows the high level architecture with all the hardware components involved. There are two laptops in the same network connected by an ethernet cable. One laptop serves as an RRH. It has the LimeSDR mini plugged into its USB port so it can send and receive signals. Incoming

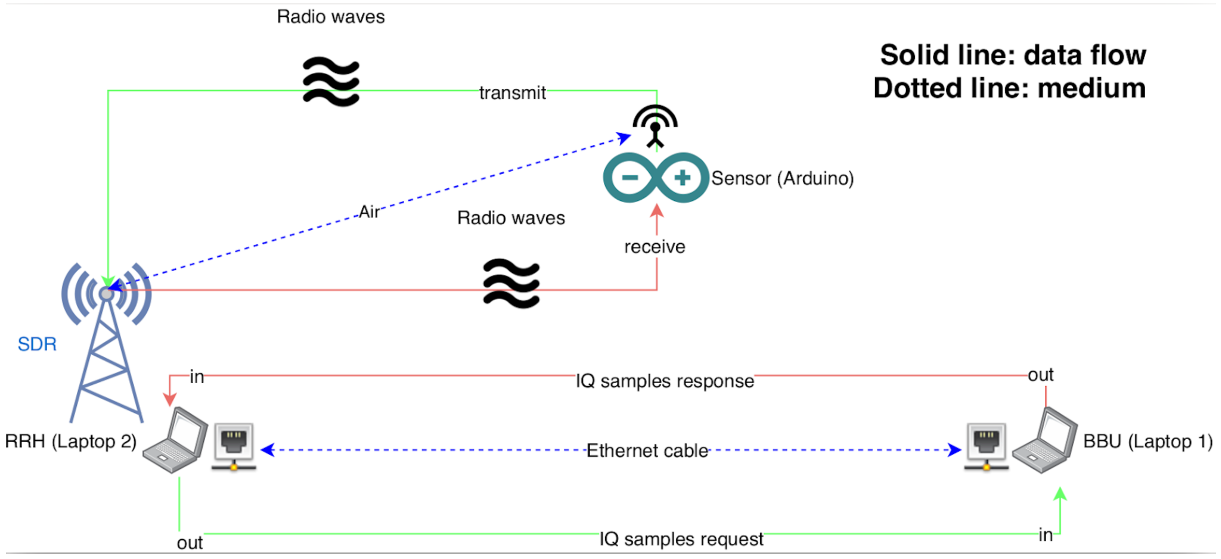


Figure 5.2: High level architecture

signals come from the Arduino device. The Arduino transmits packets over the air as radio waves. Those get picked up by the RRH which converts the analog signals and sends them as digital 32bit floats as I/Q samples over ethernet to the second laptop. This second laptop is the host for the virtual BBU that runs in a Docker container. There the signals get demodulated and decoded. Then the decoded signal gets processed. In case a response is requested, a response signal is generated and sent as I/Q samples back to the RRH. From there it gets transmitted back to the Arduino.

5.5.2 Containerization and Orchestration

Containerization is done with Docker. Docker containers are more lightweight than full virtual machines. Docker also has a command line tool for container orchestration, called Docker compose. With compose, container configuration is stored in docker-compose.yaml files. In such a file multiple container configurations can be specified and with a single command i.e docker-compose up, all containers defined in that file start up. Docker compose also lets the user scale services by spawning new containers on demand. This is the basis for a LoRa C-RAN as a service that could be build with an open cloud computing platform software like Openstack.

5.5.3 RRH and BBU

The RRH is the simplest component. It has an antenna for input and one for output. In Figure 5.3 the RRH is composed by the two components ”SDR RX” and ”SDR TX”. They correspond to the physical RX and TX slots ond the SDR device. The BBU is the ”Lora Decoder” component. It runs in a virtualized environment. Decoded messages get passed to the ”Python Script” component. This acts as a network server that schedules an acknowledgement message back to the Arduino. It could run on a third laptop connected via ethernet to the BBU laptop, but for our purposes it runs on the same laptop as the BBU but in a separate Docker container.

5.6 Implementation

All communication between the components happens over sockets. We use the ZeroMQ (ZMQ) networking library. It is a good messaging library that offers N-to-N patterns as

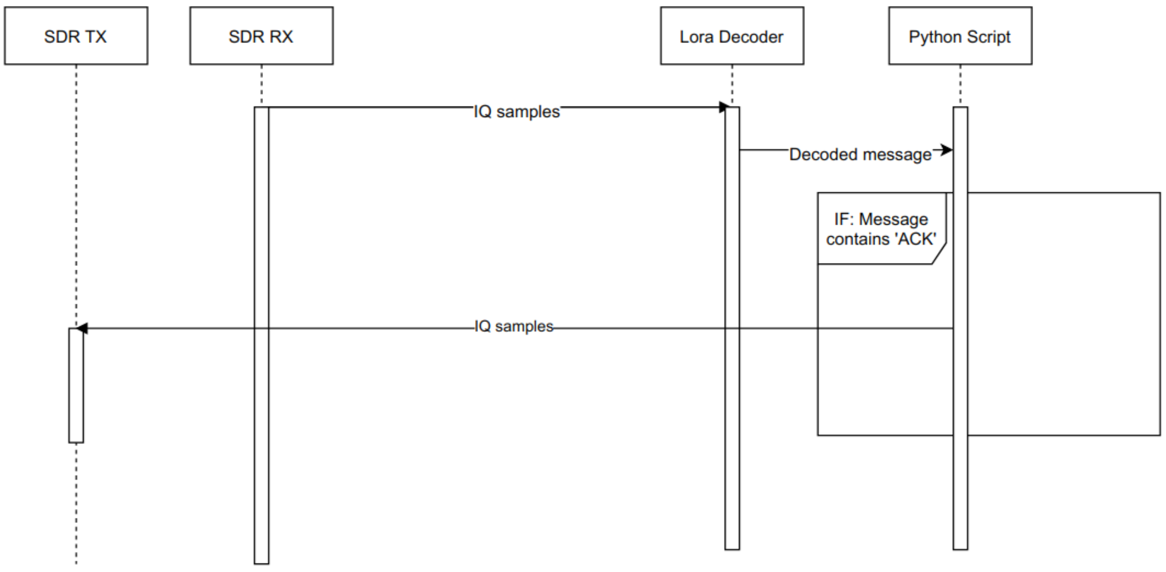


Figure 5.3: Sequence diagram

possible ways to connect the sockets, such as a request-reply pattern or pub-sub pattern and many more [?]. GNU Radio offers ZMQ blocks out of the box, The TCP sink and source blocks for socket communication are still available but deprecated. For communication between the RRH and the BBU the pub-sub (publish-subscribe) pattern is used. Figure 5.3 shows the ”PUB” and ”SUB” blocks and the data flow. The RRH has a PUB socket that takes as input the I/Q sample stream generated by the RX of the SDR. The SUB socket in the BBU subscribes to this publishing socket. As this socket connection happens with TCP, the I/Q samples arrive in the order they are sent and can be directly passed to the LoRa Decoder. Robyns' et al. implementation of LoRa Decoder sends the decoded message out on a UDP socket. The ”Python Script” block, which is our LoRa network server, takes the decoded messages sent over on this UDP socket, and then streams out the I/Q samples of the response message over a ZMQ publishing socket to which the RRH's TX slot subscribes to. This closes the cycle. One of the advantages of using ZMQ is that the sockets can be given the option to not time out or close on a disconnect. This means a subscribing socket can be started before a publishing socket without issues. The subscribing socket can wait for the publishing socket to get instantiated. For our architecture this means the docker containers for the RRH and BBU can be started in any order and more instances of the BBU can be added at runtime. The sub-pub pattern allows new subscribers and publisher to join.

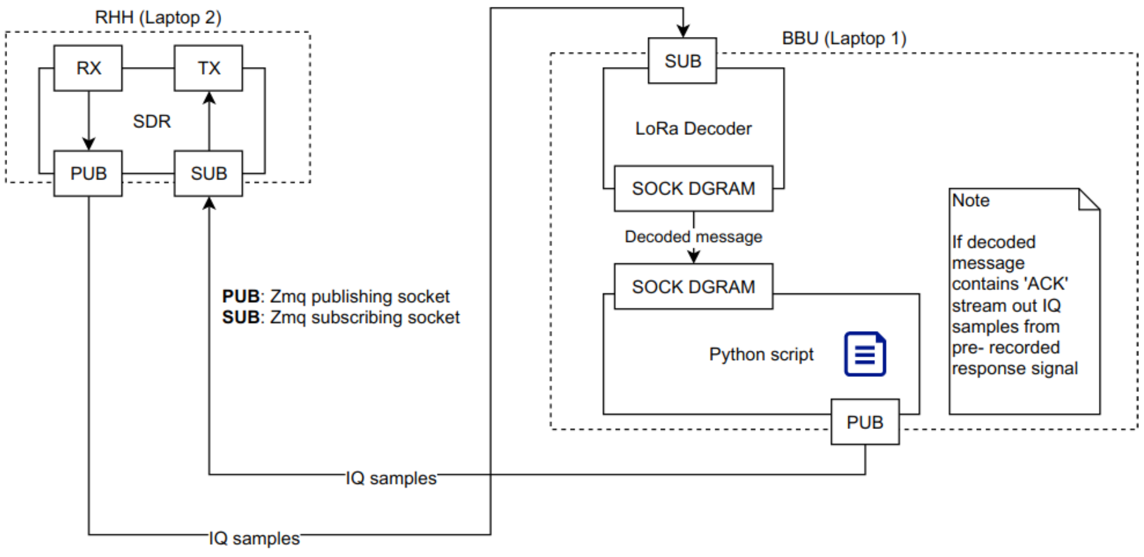

RHH (Laptop 2)
RX
TX
SDR
PUB
SUB
BBU (Laptop 1)
SUB
LoRa Decoder
SOCK DGRAM
Decoded message
SOCK DGRAM
Python script
PUB
Note
If decoded message contains 'ACK' stream out IQ samples from pre- recorded response signal
PUB: Zmq publishing socket
SUB: Zmq subscribing socket
IQ samples
IQ samples

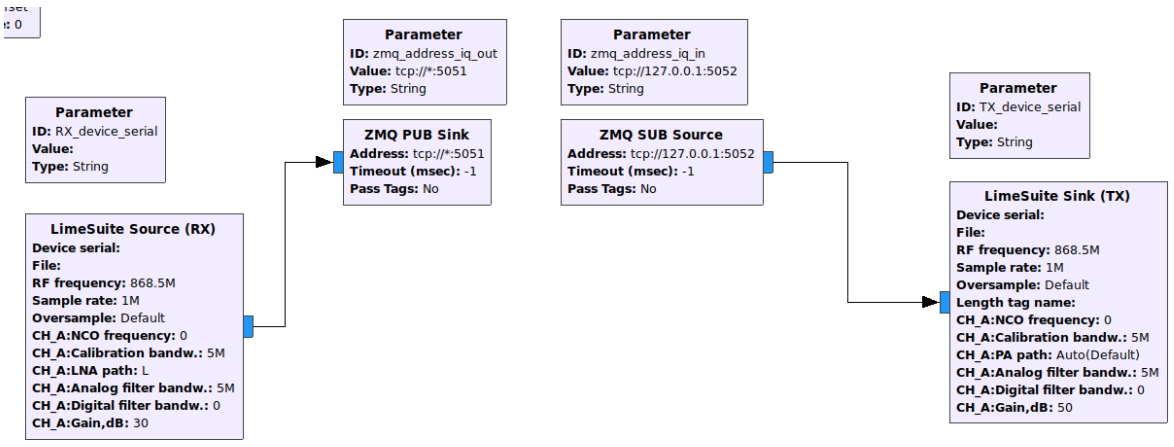
Figure 5.4: Socket communication between components

5.6.1 RRH

The RRH implementation is straightforward. Figure 5.5 shows the necessary GNU Radio blocks. On the left is the RX block of the LimeSDR that streams the incoming signals to the PUB socket. The response message from the networks server to send out comes through a SUB socket which streams directly to the TX block of the LimeSDR on the far right of the Figure. The parameter blocks allow the passing of command line arguments to the resulting application to configure the socket addresses if necessary. As there is an OOT module needed for GNU Radio to work with the LimeSDR, the RRH also comes containerized to quickly get started, as the necessary dependencies have all been installed in that container.

5.6.2 BBU

The BBU, as shown in Figure 5.6, takes in the RX stream of the RRH on a SUB socket, passes the I/Q samples to the LoRa decoder. The decoder decodes LoRa signal and outputs them as a message on the message socket sink, shown on the far right in the Figure.



Parameter
ID: RX_device_serial
Value:
Type: String

LimeSuite Source (RX)
Device serial:
File:
RF frequency: 868.5M
Sample rate: 1M
Oversample: Default
CH_A:NCO frequency: 0
CH_A:Calibration bandw.: 5M
CH_A:LNA path: L
CH_A:Analog filter bandw.: 5M
CH_A:Digital filter bandw.: 0
CH_A:Gain,dB: 30

Parameter
ID: zmq_address_iq_out
Value: tcp://*:5051
Type: String

ZMQ PUB Sink
Address: tcp://*:5051
Timeout (msec): -1
Pass Tags: No

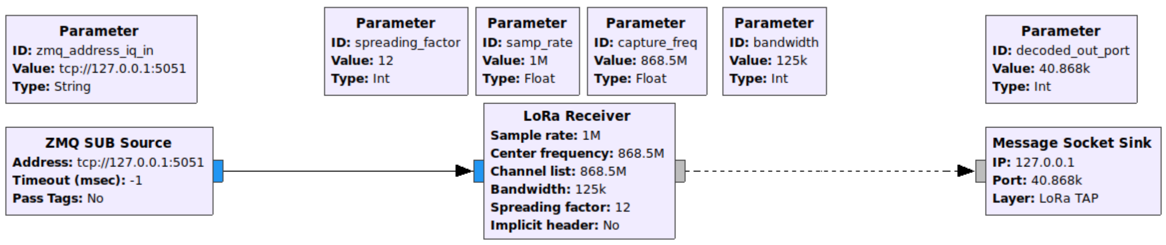
Parameter
ID: zmq_address_iq_in
Value: tcp://127.0.0.1:5052
Type: String

ZMQ SUB Source
Address: tcp://127.0.0.1:5052
Timeout (msec): -1
Pass Tags: No

Parameter
ID: TX_device_serial
Value:
Type: String

LimeSuite Sink (TX)
Device serial:
File:
RF frequency: 868.5M
Sample rate: 1M
Oversample: Default
Length tag name:
CH_A:NCO frequency: 0
CH_A:Calibration bandw.: 5M
CH_A:PA path: Auto(Default)
CH_A:Analog filter bandw.: 5M
CH_A:Digital filter bandw.: 0
CH_A:Gain,dB: 50

Figure 5.5: GNU Radio blocks for the RRH



Parameter
ID: zmq_address_iq_in
Value: tcp://127.0.0.1:5051
Type: String

ZMQ SUB Source
Address: tcp://127.0.0.1:5051
Timeout (msec): -1
Pass Tags: No

Parameter
ID: spreading_factor
Value: 12
Type: Int

Parameter
ID: samp_rate
Value: 1M
Type: Float

Parameter
ID: capture_freq
Value: 868.5M
Type: Float

Parameter
ID: bandwidth
Value: 125k
Type: Int

LoRa Receiver
Sample rate: 1M
Center frequency: 868.5M
Channel list: 868.5M
Bandwidth: 125k
Spreading factor: 12
Implicit header: No

Parameter
ID: decoded_out_port
Value: 40.868k
Type: Int

Message Socket Sink
IP: 127.0.0.1
Port: 40.868k
Layer: LoRa TAP

Figure 5.6: GNU Radio blocks for the BBU

The Python script, which acts as our network server, inspects the message payload, and if an acknowledgement is required by the sender it generates the response signal. Listing 1 shows an excerpt of that Python script. The acknowledgement message is a recording of a LoRa downlink signal. Its I/Q samples get read into memory. The script connects to the UDP socket. As the BBU component and this Python script run on the same host, it connects to 127.0.0.1, the port is passed as argument to the script. As the focus lies on the split between RRH and BBU we decided to hard code the IP address to localhost for the LoRa decoder and the network server because they run on the same machine as depicted in Figure 5.4. Then, in an endless loop, data gets received from the socket. The buffer size is 1024 bytes. Whenever "ACK" is in the message payload, the acknowledgement signal gets streamed out over a ZMQ publishing socket.

The BBU and the network server run each in a container started with Docker compose.

```python
import zmq
import socket
...
with open (dir_path + "/ACK_DOWN_SF12_CR4.raw") as f:
    ack = f.read()
...
zmq_socket = context.socket(zmq.PUB)
...
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(("127.0.0.1", udp_port))
print ('listen for decoded lora packages on udp port ' + str(udp_port))
while True:
    data, addr = s.recvfrom(1024)

    if ("ACK" in data):
        zmq_socket.send(ack)
    else:
        print ("received package requests no ACK")
```

Listing 1: Excerpt of the Python script that functions as the network server

5.6.3 Communication Protocol Arduino

The LMIC library runs a loop that executes jobs scheduled to run at a specified time. In the *setup()* function that runs once when the Arduino starts, settings such as the spreading factor, frequency and coding rate are set. The frequency is given in Hertz and the datarate is set with a predefined enum from the LMIC library. Also, the initial job for the LMIC is initialized there as well, see Listing 2 last line.

The last line schedules the initial job *txjob* with the *my_tx_func* function that gets run on execution of the job, see Listing 4. It takes the index of the packet to send, Listing 3, and checks if the packet has "ACK" appended. Then the transmit function *tx(packet,*

```
void setup() {
  Serial.begin(9600);
  ...
  // initialize runtime env
  os_init();
  ...

  LMIC.freq = 868500000;
  LMIC.txpow = 27; // Maximum TX power
  LMIC.datarate = DR_SF9;
  // This sets CR 4/5, BW125 (except for DR_SF7B, which uses BW250)
  LMIC.rps = updr2rps(LMIC.datarate);

  Serial.println("Started");
  Serial.flush();

  // setup initial job
  os_setCallback(&txjob, my_tx_func);
}
```

Listing 2: Arduino setup() function

callback) gets executed. It takes a packet and a callback function that gets executed after the transmission is finished. If an ”ACK” is present, the callback function is set to *my_txdone_func*, if not it is set to *my_txdone_no_ack_func*. Finally, in line 16 in Listing4, schedules itself to be run again after *TX_INTERVAL* which is 4 seconds.

The difference between the two callback functions passed to the transmission function is that one waits to receive an acknowledgement while the other simply increases the *currentPacketIndex* so the next packet gets sent the next time *my_tx_func* gets called. The callback to the RX function gets executed every time a packets is received. Listing 5 shows the implementation. The response signal sent by the network server has the payload

```
#define TX_INTERVAL 4000


int currentPacketIndex = 0;
const int numOfPackets = 3;
char *myPackets[numOfPackets] = {
    "This is packet 1ACK",
    "This is packet 2ACK",
    "This is packet 3",
};
```

Listing 3: Packets 1 and 2 have ”ACK” appended in their payload, while packet 3 does not

”ACK”, which is three bytes. If the *LMIC.dataLen* is three bytes, it is assumed it is the expected acknowledgement signal. The *currentPacketIndex* gets incremented and the next transmission job gets set up. Else, the *currentPacketIndex* does not get incremented so the same packet gets rescheduled for transmission and the RX function gets called again.

5.7 Study Subjects

5.7.1 Network Utilization

For measuring network traffic, the tool ”bmon” is used which stands for bandwidth monitor. It estimates the bits per second on all available network interfaces, ingoing as well as outgoing.

5.7.2 Impact of Delay

Speed is essential for most communication systems. LoRaWAN specifies two receive windows for class A devices. One 1 second after transmission, and if not received in this window, another one 2 seconds after transmission. The protocol for this experiment is more lenient. After transmission, the device starts listening for at least 4 seconds before retransmitting the unacknowledged packet.

5.7.3 Network Delay

Using the command line tool *netem*, which stands for network emulator, we introduce some delay between the RRH and the BBU. The command is: tc qdisc add dev *network_interface_name* root netem delay *number_of_milliseconds*ms. In which *interface_name* is something like eth0.
This instructs tc (traffic control) to modify the qdisc (queuing discipline) by adding (add) a new rule to the device (dev) e.g. eth0 to modify the outbound traffic scheduler (root) using the network emulator (netem) with a delay of e.g 200ms [**?**].
This setup can be easily seen an tested in action:

- On the RRH add a delay of 8000ms
- On the BBU, in a terminal, type *nc -l 4040*
- On the RRH, in a terminal, type *nc* <IP_ADDRESS_OF_THE_BBU> *4040*

On the RRH on the same terminal on, type a message and send with enter. It should arrive on the BBU's terminal after 8 seconds, which was the case.

5.7.4 Processing Delay

Not only are network delays of importance, but also the processing time it takes for decoding incoming messages and scheduling downlink messages. In the python script that sends the downlink, a call to *time.sleep()* can simulate longer processing times.

5.8 Results

5.8.1 Network Utilization

Monitoring the network utilization with bmon yields 335 bytes per second on the ethernet network interface when idle. Once the C-RAN gets started the network utilization rises to 8MiB/s (Mebibytes) which are equal to 67'108'864 bits. Bmon only gives an estimation. The theoretical value can be derived the following way:

- The LimeSDR had the sample rate set to 1 million samples per second by default
- The type is of complex type of 32 bit, (I/Q) => 2 x 32 bit, which gives 64 million bits/s
- MTU (Maximum Transmission Unit) on most systems is 1500 bytes.
- Overhead for TCP and IP is 20 bytes each => 40 bytes
- This gives a MSS (Maximum Segment Size) of 1460 bytes
- 64M bits $\div$ (1460 bytes$\times$8) = 5479.45 packets needed
- Total overhead: 5480 packets x (40 bytes x 8) = 175360 bits
- 64'000'000 bits + 1'753'600 bits = 65'753'600 bit/s = 7.8 MiB/s

The RRH is constantly sending samples to the BBU. It does not matter if the Arduino sends 1 signal every 4 seconds or 40 signals per second, the network utilization stays the same as the sample rate stays the same. The experiment worked without error with the 10 gigabit ethernet connection, but it did not work over Wi-Fi connection. The response signal sent from the BBU to the RRH did not arrive on the RRH.

Optimization: The load on the network can be reduced. According to the Nyquist-Shannon sampling theorem [**?**] a sufficient sampling rate f_s for a signal with bandwidth B is given by:

$$f_s > 2 * B$$

The signal bandwidth of the LoRa signal sent with the Arduino is set to 125 kHz. Thus the minimum sampling frequency according to the formula above is 250'000 samples per second which is a quarter of the previous sample rate of 1 million. This also quarters the network utilization from 8 MiB/s to 2 MiB/s, see Table 5.1. Lowering the sampling rate below the Nyquist-Shannon limit results in the signal not getting successfully decoded. With the sample rate of 125'000 samples per second, the maximum allowed signal bandwidth to reconstruct that signal is 62.5 kHz, so the Arduino signal with 125 kHz bandwidth cannot get decoded as is depicted in the last line in the Table.

Samples per second	Max. Signal Bandwidth	Network Utilization	Decode Success in Experiment
1'000'000	500 kHz	8 MiB/s	yes
500'000	250 kHz	4 MiB/s	yes
250'000	125 kHz	2 MiB/s	yes
125'000	62.5 kHz	1 MiB/s	no

Table 5.1: Sampling rates and network utilization

LoRaWAN defines different regional parameters. In the EU SF 8 to SF 12 signals are sent with 125 kHz bandwidth, while SF 7 can be sent with 125 kHz or 250 kHz bandwidth. In the US on the other hand SF 7 to SF 12 can also be sent with a 500 kHz bandwidth [?]. This has to be taken into consideration when designing a LoRaWAN compliant C-RAN architecture. The network load for decoding a 500 kHz bandwidth US signal is four times higher than a EU 125 kHz bandwidth signal.

5.8.2 Cost

LoRa gateways come in various prices. The low cost indoor commercial gateway by TTN costs around 70$ while their higher end gateway sells for 300$. Cheaper gateways can

be built with and Arduino and a LoRa shield such as the Dragino shield (22$). The issue is that those are often based on the SX1272/SX1276 chips. While cheaper, those gateways are so called single channel gateways. They can only listen on one channel with one spreading factor. They are not LoRaWAN compliant [?]. LoRaWAN compliant gateways can be built with a Raspberry Pi with a iC880A concentrator (130$) which use the SX1301/SX1257 chips. They can receive packets sent with different spreading factors on up to 8 channels in parallel.
The LoRa Decoder block of Robyns et al. does not yet support multichannel decoding. Spreading factor and center frequency need to be set explicitly, making it a single channel gateway. The advantage of having a decoder in software is that new instances of the LoRa Decoder block can be added without additional hardware costs. As the BBU is containerized, starting up a new instance is as easy as instructing Docker to start a new instance of the decoder and pass it the desired frequency and SF of to decode for. This saves the costs of a LoRaWAN concentrator. These savings are offset by the need for general purpose hardware the containers can run on and the higher bandwidth requirements.

Using the amazon aws cost calculator `https://calculator.s3.amazonaws.com/index.html` the following three fields are of importance :

- Compute:
  - Amazon EC2 Instance
- Data Transfer:
  - Data Transfer In
  - Data Transfer Out

For the EC2 instance we select the *a1.medium* which comes with 1 vCPU and 2 GiB of memory at *0.0255$* per hour.

Data Transfer In has to be 8 MiB/s to accommodate the maximum possible signal bandwidth of 500 kHz in the US or 4 MiB/s for the maximal signal bandwidth of 250 kHz in EU. In this cost estimation example we proceed with 4 MiB/s for Europe. 4 MiB per second are 363 GB per day. Data input is free on AWS giving a daily cost of 0$.

Data Transfer Out on the other hand does cost. The amount of data that gets sent out depends on how many uplink signals require a downlink signal. In our example a fixed 3 byte payload response is sent. In LoRaWAN, downlink messages can have variable length, multiple bytes long. Sticking with our example, the fixed downlink response signal with a payload of 3 bytes (excluding preamble, header and crc), SF 7 and coding rate 4/5, yields an I/Q sample stream of 248 kB (including preamble, header and crc) after modulation. How many downlink signals will be required? That depends on wether the end-node requires a confirmed uplink. However, there is an upper limit. The EU specifies a 1% duty cycle for the 868.0 - 868.6 MHz frequency plan [?]. The downlink signal has an airtime of 25.86 ms. With a duty cycle of 1% a downlink signal can be sent every 3 seconds. This means a maximum of 28'800 downlink signals of this type can be sent by one gateway a day. Data Transfer Out is thus limited to 7 GB a day if duty cycles are respected, $28800 * 248\ kB = 7GB$. TTN encourages end-devices to not use confirmed uplinks as most gateways are not designed to send and receive simultaneously. We assume for our experiment that about 30% of total possible downlink signals need to be sent which gives a Data Transfer Out of 2.1 GB/day.

Total monthly cost amounts to 24.07$:

- Compute:
  - Amazon EC2 Instance: 18.67$
- Data Transfer:
  - Data Transfer In: 0$
  - Data Transfer Out: 5.40$

As the modulated signals have to ben sent out over TCP/IP from the BBU to the RRH, a TCP/IP overhead of 2.7% could theoretically be added, but it is negligible as it is an estimation for comparison and not for absolute numbers.

In the traditional LoRa setup where the gateway is not split into and RRH and BBU component, the decoded packet get sent to the network server and not the I/Q samples. Assuming the same setup as before, instead of calculating $28800 * 248\ kB = 7GB$, the following has to be calculated:

- payload size: 3 bytes
- header size: 4 bytes
- CRC size: 0 bytes, downlink messaged do not have a CRC

This yield the following $28800 * 7\ Bytes = 0.0002016GB$, which is an insignificant daily amount for AWS as it gives already 0$ as cost estimation before we even apply the 30% usage assumptions. Thus, sending the I/Q samples from the cloud to the RRH is infinitely more expensive compared to the traditional setup where the signal is modulated in the gateway. The absolute cost however is still low.

In a local cloud setup where RRH and BBU are on the same network or in a network architecture, the data does not have to be routed over the Internet and the BBU can still be virtualized and centralized in a BBU hotel for much cheaper. In both setups hardware and operational costs can be saved as outlined in section 4.2. For each gateway at least 130$ costs for the LoRa concentrator could be saved by doing its job in software. For a hobbyist who operates a single gateway it may not be useful as now he needs to invest in general purpose hardware that is fast enough to do the decoding in software. Though, for an operator with multiple gateways, centralizing and virtualizing the decoding has similar benefits as a C-RAN for LTE when taking into account the OPEX additionally to the CAPEX.

5.8.3 Delays

Various network delays for the outgoing sample stream from the RRH to the BBU were set. Table 5.2 summarizes the results.

The higher the delay the less network utilization was measured with bmon. But only after

delay, namely 300ms. Without delay, RX resp. TX on the ethernet ports of the RRH and BBU is 8 MiB/s. With a 800ms delay it is about 2.32 MiB/s. With a 400 ms delay it is 5.5 MiB/s. With a delay of 300ms or less bmon measured still 8 MiB/s.
The expectation was that the signals would get decoded on the BBU the same as without delay just later. However this was not the case. There was a difference between only a single signal being picked up by the RRH and sent to the BBU and the Arduino sending multiple signals in an interval and the RRH sending I/Q samples of multiple signals to the BBU.
For a delay of 500 ms and below the following held true in the experiment:

- The Arduino sends a single signal then stops. The RRH receives the signal and sends the I/Q sample stream to the BBU. On the BBU the signal gets successfully decoded with a delay of 2 to 10 times the delay set with Netem.
- The Arduino sends signals in a 4 second interval. The signals get decoded in order on the BBU.

For a delay of 500 ms and up the following held true in the experiment:

- The Arduino sends a single signal then stops. The RRH receives the signal and sends the I/Q sample stream to the BBU. On the BBU the signal does not get decoded.
- The Arduino send signals in a 4 second interval. Some signals get decoded after the delay times 10. Some signals do not get decoded. In an experiment with 800 ms delay, the first three signals did not get decoded, the fourth signal did after a 10 seconds delay.
  Generally, the first signals never got decoded, only the 4th or later signals got eventually decoded by the BBU. A possible cause could be that either the LorRa decoder block or the GNU Radio scheduler has an issue with some sort of buffer not being filled fast enough when a single signal is sent, and if multiple signals are

sent the buffer has time to fill up which leads to some decode success when multiple signals are sent in an interval. This is but pure speculation and would need to be investigated further.

Netem Delay	Network Traffic	Eventual Decode Success	
(ms)	(MiB/s)	Single Signal	Signal Interval
0	8	yes	yes
20	8	yes	yes
50	8	yes	yes
100	8	yes	yes
200	8	yes	yes
300	8	yes	yes
400	5.5	yes	yes
500	4.25	no	yes
600	3.6	no	yes
800	2.3	no	yes
1000	1.92	no	yes

Table 5.2: Effect of delay on network traffic on decode process

5.8.4 Processing Delay

Processing delays on the LoRa network server were straightforward and as expected. Sleep is called in the Python script that sends the downlink. If this is more than the time the Arduino is set to listen for downlink signals, the downlink is missed. For LoRaWAN class A devices this is 1 resp. 2 seconds after uplink.

In this example the Arduino sends in an interval of 4 seconds which is more than enough time under good conditions. However if network delay and processing delays occur at the same time, this changes. With a Netem delay of 400 ms, the decoded message arrives about 3 seconds later, which leaves only 1 second for the downlink to be sent in our Protocol. For LoRaWan there is still 600ms left to make the first receive window and 1600ms to make the second receive window. Comparing this to LTE where the requirements impose an upper-bound of less than 3 ms for TX/RX processing [**?**], LoRaWAN, at least for Class A devices, offers much more leeway.

```
static void my_tx_func(osjob_t *job) {
if (currentPacketIndex < numOfPackets) {
    ...
    char lastThree[3];
    memcpy(lastThree, &myPackets[currentPacketIndex][length - 3], 3);
    const char ack[] = {'A', 'C', 'K'};
    if (!memcmp(lastThree, ack, 3)) {
        // send and start rx for receiving ACK
        Serial.print("transmitting packet with ACK, packet: ");
        tx(myPackets[currentPacketIndex], my_txdone_func);
    } else {
        // send and schedule next packet
        Serial.print("transmitting packet without ACK, packet: ");
        tx(myPackets[currentPacketIndex], my_txdone_no_ack_func);
    }
    os_setTimedCallback(&txjob, os_getTime() +
                                ms2osticks(TX_INTERVAL), my_tx_func);
    } else {
        Serial.println("No more packets to send, done");
    }
}
```

Listing 4: *my_tx_fun* function

```
static void my_rx_func(osjob_t *job)
{
    if (LMIC.dataLen == 3)
    {
    Serial.println("Got ACK");
    // if we get our ACK, start with next transmission,
    // reschedules transmission at half TX_INTERVAL
    currentPacketIndex++;
    os_setTimedCallback(&txjob, os_getTime()
                                + ms2osticks(TX_INTERVAL / 2), my_tx_func);
    }
    else
    {
    Serial.println("NOT AN ACK");
    // resend packet if no ACK received within 3*TX_INTERVAL,
    // reschedules transmission in 3* TX_INTERVAL
    os_setTimedCallback(&txjob, os_getTime()
                                + ms2osticks(3 * TX_INTERVAL), my_tx_func);
    // listen again
    rx(my_rx_func);
    }
}
```

Listing 5: RX function that checks for the response or reschedules the packet transmission

Chapter 6

LoRa Tools

6.1 Getting a Downlink signal

With the Arduino we could generate raw LorRa signal, record them with the SDR and save them to a file for inspection. However, getting a downlink signal was not that straightforward. In this section, three methods of getting a downlink signal are presented that were tried. The last method is clearly the superior method.

1. Schedule a downlink over TTN
2. Manipulate the packet forwarder of the gateway and schedule a downlink over TTN
3. Manipulate the packet forwarder of the gateway in the private LorRa network

The first idea was to connect the gateway to TTN. The Arduino LMIC library provides the LoRaWAN implementation for communicating with a LoRaWAN server like TNN. After setting up the gateway and Arduino for TTN correctly, the gateway is visible in the TTN dashboard as well as the messages that the Arduino sends. Then, in the TTN dashboard a downlink can be scheduled for when the next uplink message is received. There are two issues with this method. First, the downlink message will be a LoRaWAN packet. That means when a 3 byte message ”ACK” is sent downlink, the actual signal has all the LoRaWAN fields, as shown in 2.9, and is encrypted with an AES (Advanced

Encryption Standard). The raw LorRa packet thus has then a length of around more than 3 bytes.
Second, the gateway is free to send the downlink on any valid LoRaWAN frequency with any valid SF it deems reasonable. To get the signal on the desired frequency and SF, multiple downlinks have to be scheduled and one has to wait for the network server to cycle to the desired frequency for each new downlink.

Second, to avoid the varying frequencies and SF, the gateway can be manipulated to send exactly what we want by modifying the source code and recompiling. Changes have to be done to this file: `https://github.com/TheThingsNetwork/packet_forwarder/blob/legacy/poly_pkt_fwd/src/poly_pkt_fwd.c`. Listing 6 shows the following necessary changes:

- Set the frequency to 8685000000 instead of letting the server decide
- In any case set the SF to 12
- Override the packet size to 3
- Copy the desired payload into the downlink packet payload i.e. ”ACK”

This makes the gateway to send ”ACK” instead of the actually scheduled response.

This already gives significantly more control and convenience than the first approach. Though, there are two other issues that have to be addressed. First, there is no guarantee that TTN decides to send the downlink over your manipulated gateway. The TTN network server could decide to send the message over another gateway close enough to your end-device. In this case the downlink message would not get modified.
Second, your gateway is essentially a trap for the other TTN users. If their downlinks get sent over this manipulated gateway, the payload their end-devices receive is not what they specified. The end-devices will probably just drop the packet as it is not LoRaWAN conform anymore and also not signed with their key. Nevertheless, this modified gateway is a disruptive factor to the TTN network, thus this approach should be avoided.

The third method methods is to use a private LoRa server with the modified gateway. The ChirpStack project, `https://www.chirpstack.io/`, provides open source components of the LoRaWAN stack. Once installed, the gateway can be configured to forward the packets to this local LoRaWAN server instead of TTN. The ChirpStack application server provides a similar web interface to TTN where gateways and devices and application can be registered. Now downlinks signals can be scheduled like in a TTN dashboard. The difference is that the downlink will always be sent over the modified gateway as it is the only gateway registered in this local LoRaWAN network.

6.2 Generating a Downlink Signal in Software

Currently, the Python script in the LorRa C-RAN architecture sends a pre-recorded downlink signal as a response to the uplink messages. This signal was obtained the way described above. Ideally, the downlink signal could be generated on demand in software, enabling dynamic payload instead of the static pre recorded ”ACK” signal.
Robyns et al. decoder repository has branch called encoder which would send out a predefined packet (uplink) as many times as the GNU Radio scheduler was able to handle. The packet log on the Raspberry Pi gateway showed the incoming messages, but the payload of each subsequent message was different than the one before. The branch was in early development. There was most likely an error in the implementation that propagated with each call to GNU Radios schedular. So we took the code out of the GNU Radio and refactored this OOT module into a single c++ file that can be executed without GNU Radio. It is hereafter referred as encoder. Instead of streaming the I/Q samples out to the SDR's TX, the encoder simply writes the generated samples to a binary file. We then provided the file as input source to the SDR to stream out. Now the gateway showed the same payload for every signal received. This means the modulation and encoding process is deterministic for a given input, and the issue was indeed with how it was implemented in the GNU Radio framework.
The encoder encodes up and down chirps. Full up chirps for preamble, full down chirps for start of frame delimiter and various length up chirps to encode the data. To get a downlink signal we added a flag, and a bit of refactoring, that signals the *transmit_packet*

function to flip all down chirps to up chirps and vice versa. This generated downlink signal then gets successfully received by the Arduino.

6.3 Improving the Encoder & Chirp Visualization

Interestingly, the generate uplink signals by the encoder cannot be decoded by the LorRa decoder block, but only by real hardware. According to the author, there are some artifacts at the symbol boundaries for which real hardware can compensate for but the software decoder cannot. This is explained in issue 48 on GitHub, `https://github.com/rpp0/gr-lora/issues/48`.

To visualize the symbol values of the chirps, the LoRa decoder needs to be modified to save the symbol values at each sample to a file. This file can then be the input to the Python visualization script. To make this process easier, we take the LorRa decoder functionality out of the GNU Radio and put it into a single c++ file. In GNU Radio the GNU Radio scheduler fed a number of input items in each call of the *work()* function to the decoder and the decoder told the scheduler how many output items it produced with the *consume_each(int number_of_items)* function. The functions *work* and *consume_each()* are GNU Radio framework functions we needed to emulate. See `https://wiki.gnuradio.org/index.php/BlocksCodingGuide#Basic_Block` for a basic example.

To achieve this, the *work* function was replaced with a while loop and the *consume_each* function with a pointer arithmetics. Listing 7 shows the relevant code snippets.

In the program's main function the file with the I/Q samples is read into a buffer. While there are still samples left to read, the work function is called.

In the work function all calls to consume_each are replaced by incrementing the read samples and moving the pointer forward by how many samples were consumed.

Whenever the decoder calls the demodulate function, we print the value and at which sample it occurred to a csv file.

This file and the I/Q sample file are the inputs to the visualization script which then produces a spectrogram chart.

Figure 6.1 shows an uplink signal sent with the Arduino, Figure 6.2 the signal generated with the encoder. In the original implementation the number of bytes to transmit was set to match only the tespacket. We added the implementation of the formula in 2.3.5 to adjust the number of bytes to transmit dynamically. This fixed the issue that before, the signal from the encoder did not have 18 symbols like the signal from the Arduino. Now the number of symbols in the encoder generated signal corresponds to the number of symbols in the Arudino signal, however the symbol values do not.

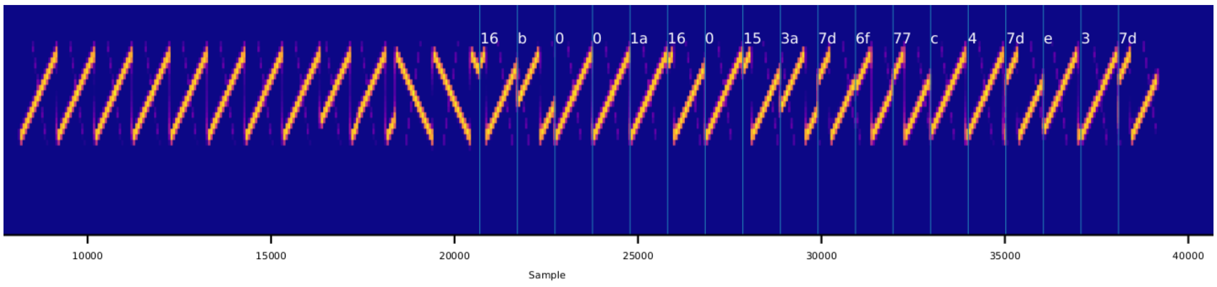


Figure 6.1: 123 SF 7, CR 4/5, sent with Arduino

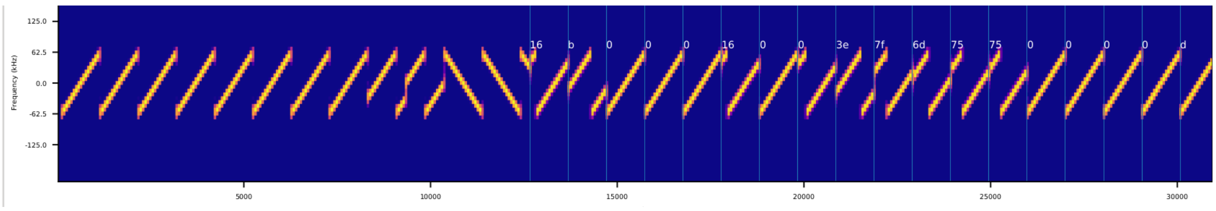


Figure 6.2: 123, SF 7, CR 4/5, sent generated with encoder

```
void thread_down(void* pic) {
    ...
    //txpkt.freq_hz = (uint32_t)((double)(1.0e6) *
     ↪  json_value_get_number(val));
    txpkt.freq_hz = 868500000;
    ...
    switch (x0) {
            // case  7: txpkt.datarate = DR_LORA_SF7;  break;
            // case  8: txpkt.datarate = DR_LORA_SF8;  break;
            // case  9: txpkt.datarate = DR_LORA_SF9;  break;
            // case 10: txpkt.datarate = DR_LORA_SF10; break;
            // case 11: txpkt.datarate = DR_LORA_SF11; break;
            // case 12: txpkt.datarate = DR_LORA_SF12; break;

            case  7: txpkt.datarate =  DR_LORA_SF12;  break;
            case  8: txpkt.datarate =  DR_LORA_SF12;  break;
            case  9: txpkt.datarate =  DR_LORA_SF12;  break;
            case 10: txpkt.datarate =  DR_LORA_SF12; break;
            case 11: txpkt.datarate =  DR_LORA_SF12; break;
            case 12: txpkt.datarate =  DR_LORA_SF12; break;
            ...
    }
    ...
    // txpkt.size = (uint16_t)json_value_get_number(val);
    txpkt.size = 3
    ...
    // i = b64_to_bin(str, strlen(str), txpkt.payload, sizeof
     ↪  txpkt.payload);
    memcpy(txpkt.payload, "ACK", 3);
    i = 3;
    ...
```

Listing 6: Changes to the polypacket forwarder

```
int main (){
    ...
    file_size = std::experimental::filesystem::file_size(abs_path);
    uintmax_t total_samples = file_size / sizeof(gr_complex);

    char buffer[BUFFER_SIZE];
    std::ifstream fin(abs_path, std::ios::in | std::ios::binary);
    fin.read(buffer, BUFFER_SIZE);
    input = (gr_complex *)buffer;
    while (read_samples < total_samples)
    {
        work(input);
    }


    ...
}

int work (gr_complex *&input){
    ...
    // consume_each(d_samples_per_symbol);
    read_samples = read_samples + d_samples_per_symbol;
    input = input + d_samples_per_symbol;
    ...
}

bool demodulate () {
    ...
    outfile << std::hex << word << ',' << std::dec << read_samples <<
     ↪  std::endl;
    ...
}
```

Listing 7: Emulate the GNU Radio schedular

Chapter 7

Future work

The design for a C-RAN for LoRa depends on level of sophistication of software processing for LoRa.
Best case scenario would haven been if software decoding, encoding and the LoRaWan protocol were already implemented for software defined radios.
As this was not the case, we first had to spend a significant amount of time to understand the inner workings of LoRa better. This is made more difficult by the fact that the PHY layer of LoRa is proprietary and no official documentation exists.

7.1 Limitations

A first limitation is that the response signal is a recording of a real downlink signal and not purely generated in software on demand. Nevertheless we developed the tools to make it possible.
Second, the LoRa protocol implemented on the Arduino is ”raw” LoRa and not compliant with the LoRaWan standard. This was necessary as decoding a LoRaWan packet is not yet implement in the decoder, in terms of destructuring the received packet according to all the LoRaWan fields as shown in Figure 2.9.
Third, the experiment, C-RAN architecture and Arduino implementation is based on the assumption of a single end-device and gateway in the network. In a LoRaWan, end-devices

ignore downlink packets not destined for them. Our Arduino however happily accepts any "ACK" downlink because of the premise that all downlinks are destined for it as it is the only device in the network.
Further in a network with multiple gateways, an uplink signal gets picked up by multiple gateways and forwarded to the network server. Then the network server discards duplicates and decides over which gateway the downlink response is sent. In our network however there is only one RRH.

7.2 Improvements

Improvements to the protocol implementation can be made. Though, instead of improving this protocol, it makes far more sense to adopt the LoRaWan protocol. The LoRaWan protocol is an open protocol. Implementing it on the decoder requires the necessary time and knowledge. The Arduino could already speak the LoRaWan protocol with the help of the LMIC library which offers a fairly complete LoRaWan Class A device implementation. On network server side, an initial improvement would be to dynamically generate the response signal by incorporating the encoder for downlink signals in the architecture.

Further, there is only one decoder / BBU running because there is only one Arduino sending on one channel. As the BBU is containerized, the experiment can be run with multiple Arduino devices on different channels and spreading factor. Witch Docker compose, multiple BBUs could be started to handle the different channels and spreading factor, they would then forward the decoded messages to the network server. This can be already done with the current implementation. The network server however could be replaced with the open implementation of the chirpstack network server, `https://www.chirpstack.io/network-server/overview/`. It handles among other things the de-duplication of LoRaWan messages and downlink scheduling. It takes LoRaWan messages from an MQTT broker. MQTT like ZMQ facilitates architectures based on a publish-subscribe strategy. So instead of the decoder sending the decoded messages out over an UDP socket and our Python script subscribing, the decoder can publish the decoded message to the MQTT broker. For downlink scheduling the chirpstack network server sends the message also over MQTT back to the gateway for modulation and transmission. For the C-RAN architecture where the downlink is modulated in software and not on the gateway anymore, this would need a redesign where the RRH simply receives the I/Q samples to transmit without having to do any LoRa modulation. Then the chirpstack application server could be added for free, giving a nice web-interface similar to the TTN interface, but instead of traditional LoRaWan network architecture a C-RAN LoRaWan architecture is running.

For further developing the centralized / cloud aspect of C-RAN, the BBUs instead of running on a single host controlled by Docker compose could be spread of multiple machines with Docker swarm or Kubernetes. Further down the road BBU and network server could be offered as a service via software such as Openstack essentially providing C-RAN as a service as demonstrated for LTE by in [**?**].

Finally, it needs to be investigated why the decoder cannot successfully decode I/Q samples of a single signal streamed over and ethernet connection as TCP/IP packets if a network delay of more than 500ms is emulated on the connection. Also, there are some open issues on the GitHub repo of the decoder and a few items that are not yet implemented such as multichannel decoding and CRC checks for payload and header. However, the decoder itself is not in scope of this thesis and is just mentioned here for completeness.

Chapter 8

Summary and Conclusions

Moving to a C-RAN approach for LoRaWAN networks is feasible. For this the LoRa gateways functionality must be split up in a into a RRH and BBU component. It shows how the BBU part can be accomplished in software and the benefits it brings by containerizing and running it in a virtualized environment. Tradeoff for this simplification of the LoRa gateway is the introduction of significant amount of data traffic between the RRH and the BBU component and a more complex architecture as the one gateway component is now split into two components. This offers more flexibility and convenience in the long run but needs to be managed and maintained nevertheless. Generally, C-RAN for LoRa brings much of the same advantages as a C-RAN for LTE does in terms of CAPEX and OPEX. For at least class A devices, the LoRaWAN protocol allows for more leeway in processing as receive windows for downlinks signals are open either 1 or 2 seconds after transmission. Data traffic between RRH and BBU can be reduced by applying the Nyquist-Shannon sampling theorem. By reducing the sampling rate to only the minium needed network utilization between RRH and BBU can be greatly reduced compared from worst to best case.

There is only one implementation for software defined LoRa decoding that worked for us but it is fairly completed. However the modulation of an uplink signal is still in the early stages. We extended the modulation by implementing the modulation of a downlink signal but the correctness and sophistication is strictly tied to the uplink modulation implementation as the downlink signals is essentially the inverse of an uplink signal in terms of chirp direction.

Having also modulation of signals defined in software, not only the decoding, would make prototyping and simulation of network conditions much simpler. The difficulty lies in the fact that the LoRa PHY is proprietary which is why reverse engineering it is so hard. Having said that, as there already exists a functioning decoder, an encoder should not be that far away. We contribute some tools we developed during to this thesis such as the downlink modulator and chirp visualizer that could possibly help for further development.

Abbreviations

C-RAN	Centralized / Cloud Radio Access Network
CAPEX	Capital Expenditures
CSS	Chirp Spread Spectrum
IOT	Internet of Things
IP	Internet Protocol
LAN	Local Area Network
LoRa	Long Range
LTE	Long-Term Evolution
OPEX	Operational Expenditures
PHY	Physical
RF	Radio Frequency
RX	Reception
SDR	Software Defined Radio
TCP	Transport Control Protocol
TTN	The Things Network
TX	Transmission
WAN	Wide Area Network

Glossary

C-RAN Cloud / Centralized Radio Access Networks centralize the BBU components of the radio towers in a BBU hotel.

LoRa a spread spectrum modulation technique derived from chirp spread spectrum (CSS) technology.

LoRaWAN the open communication protocol based on LoRa backed by the LoRa Alliance

LoRa Alliance A non-profit association committed to the development and promotion of the open LoRaWAN standard.

PHY is the physical layer of LoRa which is proprietary and belongs to Semtech. Software implementations of LoRa try to reverse engineer the PHY.

RF Technology All technology related to radio frequency.

TCP / IP The Transmission Control Protocol and Internet Protocol operate on the Transport layer resp. the Internet layer and is used in local networks as well as on the Internet. TCP controls how data is transmitted while IP is responsible for addressing and routing.

TX Concerning the transmission of data.

SDR Software Defined Radio allow the signal processing to be done in software on general purpose hardware.

Socket are system resource for receiving and sending data in a computer network.

RX Concerning the reception of data

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

The CD contains the following:

- a folder with the Arduino source code
- a folder for the docker setup
- a folder with the grc (gnuradio companion) files
- a folder with various signal recordings
- a folder with the thesis source code and the final thesis pdf
- a folder with various tools for decoding, encoding and visualization of LoRa signals
- a README.md file

All contents of the CD are available online on my GitHuB `https://github.com/mustard123/master-thesis`

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 priviledged 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)

Param	Explanation
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*

Param	Explanation
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

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

Example:

To pass a capture frequencey of 915M and a sample rate of 250k enter the params in the following way in the command field:

docker-compose.yml

```
version: '3'
services:
    rrh:
        build: .
        privileged: true
        network_mode: host
        volumes:
                - /dev/bus/usb:/dev/bus/usb
        command: ["--capture-freq", "915000000", "--samp_rate", "250000"]
```

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_Network_Server: 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.

Param	Explanation
decoded-out-port	On which port the decoded messages will be sent out. Localhost only. The LoRa_Network_Server 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_Network_Server 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)

Param	Explanation
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_Network_Server 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_network_server:
                build: ./LoRa_Network_Server
                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 SF9 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_Network_Server 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.