

# Bachelorarbeit

**Titel:** Entwicklung eines LoRaWAN fähigen NOx Sensors

**Gutachter:**

- Prof. Dr. Georg Hartung (TH Köln)
- Dr. Tobias Krawutschke (TH Köln)

**Zusammenfassung:** Die Luftverschmutzung ist ein gravierendes Problem in Köln und anderen städtischen Gebieten Deutschlands. Durch genaue Bestimmung der Herkunft der Schadstoffe ist es möglich das Problem effektiver zu bekämpfen. Dazu wird in dieser Arbeit ein NOx Sensors entwickelt, der über LoRaWAN seine Daten über hohe Entfernungen senden kann. In Zukunft kann dieser dabei helfen ein städtisches Netz an Luftqualitätssensoren aufzubauen. Ferner wird LoRaWAN als Medium der Datenübermittlung zu diesem Zweck untersucht und als geeignet erkannt.

**Stichwörter:** LoRaWAN, The Things Network, Internet of Things, Sensoren, Luftqualität

**Datum:** 16. Januar 2018

# Bachelor Thesis

**Title:** Development of a LoRaWAN compatible NOx sensor

**Reviewers:**

- Prof. Dr. Georg Hartung (TH Köln)
- Dr. Tobias Krawutschke (TH Köln)

**Abstract:** Air pollution is a grave problem in Cologne and other urban areas in Germany. By accurately determining the source of the pollutants, it is possible to combat the problem more effectively. For this purpose, a NOx sensor is developed in this thesis, which can send data over long distances via LoRaWAN. In the future, this can help build an urban network of air quality sensors. Furthermore, LoRaWAN as a medium of data transmission for this purpose is examined and recognized as suitable.

**Keywords:** LoRaWAN, The Things Network, Internet of Things, Sensors, Air Quality

**Date:** January 16, 2018

# Table Of Contents

<b>Preface</b>	<b>3</b>
<b>1 Background</b>	<b>4</b>
1.1 OpenAir Cologne	4
1.2 Communication Technologies	5
1.2.1 Wireless LAN	5
1.2.2 LoRa	6
1.2.3 LoRaWAN	6
1.2.4 The Things Network	7
1.2.5 SigFox	8
1.3 SensorCloud	8
<b>2 Examining LoRaWAN And The Things Network</b>	<b>9</b>
2.1 Device Classes	9
2.2 Activation Methods Of End Devices	10
2.2.1 Over-The-Air Activation (OTAA)	11
2.2.2 Activation By Personalization (ABP)	12
2.3 LoRaWAN Packet Structure	13
2.3.1 Uplink Messages	13
2.3.2 Downlink Messages	14
2.4 Security	14
2.5 LoRa Service Range	15
2.6 Estimating Battery Life Time	19
2.7 Estimating Packet Time On Air And Bit Rate	21
2.8 The Things Network Backend Architecture	26
<b>3 Developing The Distributed NOx And CO Measurement System</b>	<b>28</b>
3.1 End Device	30
3.1.1 Market Research Devices/End Devices	30
3.1.2 Measuring NOx And CO Data	32
3.1.3 Registering And Connecting An End Device To The Things Network	36
3.2 Gateway	38
3.2.1 Market Research Gateways	38
3.2.2 Building The Gateway	40
3.2.3 Configuration Of The Gateway	42
3.2.4 Registering A Gateway To The Things Network	42

3.2.5 Security	44
3.2.6 Protection Against Environmental Influences	45
3.3 Network	45
3.3.1 Configuring The Things Network	45
3.3.2 Payload Formats	46
3.4 Application	47
3.4.1 Registering An Application Using The Things Network	47
3.4.2 Communication To And From The End Device	49
3.4.3 The Things Network MQTT API	49
3.4.4 Uploading Data To The SensorCloud	51
<b>4 Instructions For Use</b>	<b>54</b>
4.1 End Device	54
4.2 Gateway	55
4.3 Application	55
<b>Abstract And Outlook</b>	<b>57</b>
<b>Appendix</b>	<b>59</b>
Acronyms And Abbreviations	59
CD Contents	60
<b>List Of Figures</b>	<b>61</b>
<b>List Of Tables</b>	<b>62</b>
<b>Bibliography</b>	<b>63</b>

# Preface

Today, more people live in cities and urban areas as ever. The high concentration of people activities greatly influences the air quality. A considerable source for pollutants are combustion motors, especially in transportation. The resulting smog depicts a worldwide problem and is a big source for health issues. While the output of particulates could greatly be reduced by the use of particle filters in Germany, the concentration of nitrogen oxide in Cologne and other German cities is among the highest of the world. [1]

Nitrogen oxide, or NO<sub>x</sub>, is a combination of nitrogen monoxide and nitrogen dioxide. While nitrogen monoxide is regarded as relatively harmless for humans, nitrogen dioxide can be the source for respiratory illnesses. Therefore the concentration of nitrogen dioxide needs to be analyzed in long term measurement. [1]

In the course of this bachelor thesis the prospects of air quality sensors transmitting data via LoRaWAN is investigated.

The goal of this bachelor thesis is to build a LoRaWAN network to read NO<sub>x</sub> and other air quality data. This includes a sensor, able to read NO<sub>x</sub> and transmitting them via LoRaWAN, a gateway integrated to The Things Network, and an application that receives the sensor values from The Things Network and transferring them to the SensorCloud, a database for sensor values build by the University of Applied Science Cologne, to be saved persistently.

The first chapter gives a overview of the technologies that were used. The second chapter examines LoRaWAN, as well as The Things Network, in a greater detail to descry the suitability of these technologies. Chapter three describes the approach of building the sensor itself and the backbone to support the LoRaWAN capabilities. The fourth chapter contains user references to operate or extend the LoRaWAN system that was built for this thesis.

# 1 Background

## 1.1 OpenAir Cologne

OpenAir Cologne is a collaboration project of the OKLab Cologne, Everykey, the city of Cologne, and the University of Applied Science Cologne under the slogan of ‘Citizen Science’ or ‘Citizen partake in Science’. Citizen get active and take part in duties and responsibilities which are originally assigned to government instances. Not in the sense of establishing a kind of rivalry, but more under the aspect of considering different viewpoints. The goal is to establish a collaboration in the fields of Open Data and civic participation where an idea originating from the civic community is brought up to the administration and commonly implemented under scientific assistance. [2]

The example model the OpenAir Cologne community is the collection of nitrogen oxide (NO<sub>x</sub>) data by volunteers within the rural area of Cologne. OpenAir Cologne also acts as a platform teaching the necessary knowledge of the functionality and operation of sensors in workshops and other events to interested people.

The first 35 NO<sub>x</sub> sensors were distributed to the public in October 2016. These sensors were MiCS-4514 Gas Sensors attached to an Adafruit Feather that communicated over Wireless LAN with a database to store the data. As of November 2017, seven of them are still running, measuring and sending data. The first 30 sensors of the second generation, the OpenAir Node V1.1, were distributed in November 2017 to interested members of the community, with the goal to have distributed 70 more by the end of March 2017. Like the previous generation, the OpenAir Node V1.1 is equipped with a MiCS-4514 Gas Sensor, which can be seen in Figure 1 (A), but has an ESP32 with Wi-Fi chip (B). In addition to Wireless Lan capabilities, the circuit board has connectors for a Hoppe RF chip for future LoRa capabilities, which are implemented in scope of this thesis.

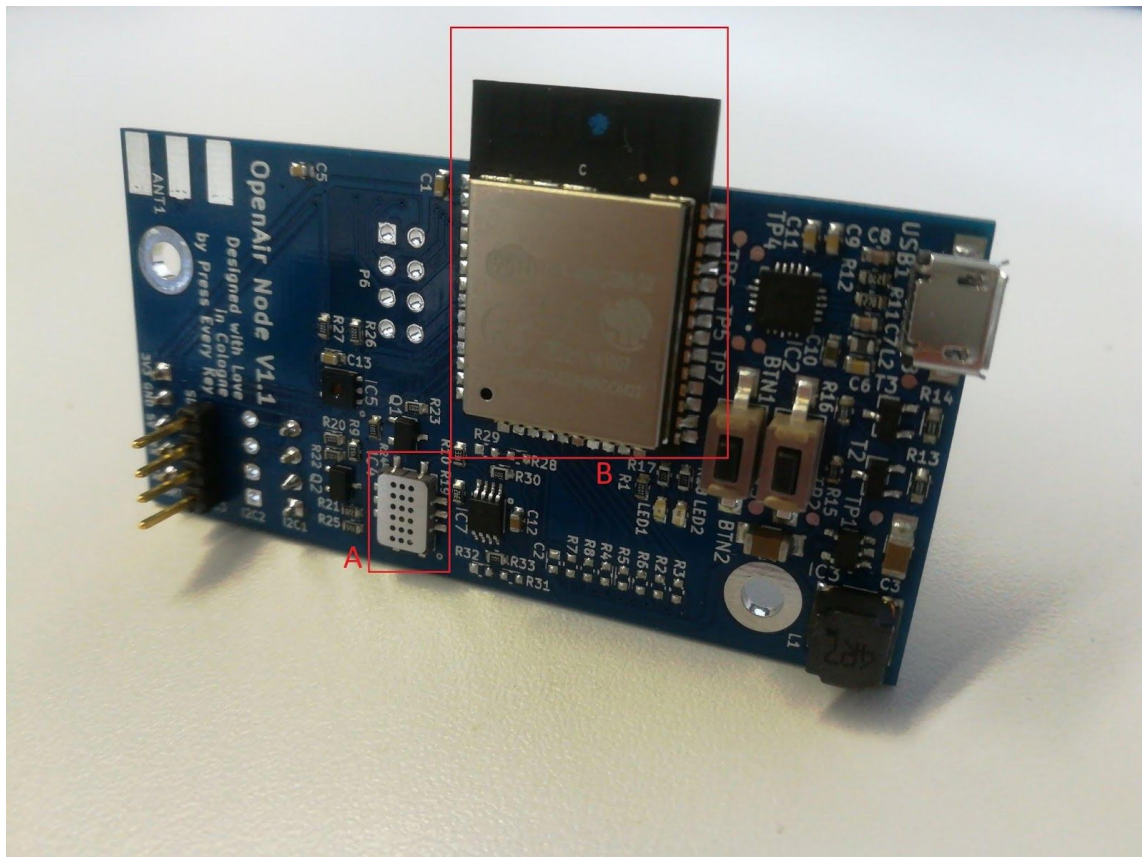


Figure 1: OpenAir Node V1.1 by Press Every Key

## 1.2 Communication Technologies

There are many communication technologies available today. In the scope of this thesis, the most important communication is the ‘first-mile’, meaning from the sensor to the router or gateway. With the upcoming of LPWAN technologies in the last few years, which offer high range and low power consumption, these technologies are the most interesting to look at. The drawback of these technologies, the low bitrate, has to be evaluated.

### 1.2.1 Wireless LAN

Wireless LAN big upside is its considerably fast data rate. In Germany the mean internet speed is about 15,3 Mbit/s, which is far more than is needed to send the air measurement data, which is around 10 bytes, every minute [3].

Currently, the OpenAir Node V1.1 uses Wireless LAN for communication. This comes with three problems: availability, security, and customization. With Wireless LAN, the sensors can only be deployed where wireless networks are available and where one of the wireless network owners is willing to hand out the network information. This leads right to the second problem which is security and trust: the sensor has to store the SSID

and the password to the wireless network. As the third problem, the stored network information must be changed when the sensor moves to another network, or if the network information changes. Wireless LAN modules also consume a considerable amount of energy, which makes it infeasible to use these in battery operated mode.

### 1.2.2 LoRa

LoRa (*Long Range*) is a physical layer to create a long range communication link. It uses a spread spectrum modulation technology, which maintains very low power characteristics with a significantly increased communication range compared to many legacy wireless systems.

According to the LoRa Alliance, LoRa's main advantage is the technologies long range capabilities. This allows to cover an entire city with just a single Gateway. This is due to the great link budget LoRa and LoRaWAN have. The link budget, as the name implies, is the maximum difference between different gains and losses between the transmitter and the receiver, and is the primary factor in determining the range of a wireless network environment. But the range is also highly dependent on the surroundings and obstructions in a given location. [4]

The LoRa spread spectrum modulation scheme is a proprietary derivative of Chirp Spread Spectrum (CSS), which was developed for radar applications in the 1940's. It generally trades data rate for sensitivity within a fixed channel bandwidth to provide a low-cost, low-power, and still robust communication technique. [5]

In CSS, a chirp is a signal of continuously increasing or decreasing frequency in a finite bandwidth. It either increases or decreases until it hits the end of the band and then wraps around to the beginning. In contrast to other radio technologies, where the data is modulated on a fixed frequency carrier, in LoRa the information is modulated on these chirps. [5]

### 1.2.3 LoRaWAN

LoRaWAN (LoRa wide-area network) is a LPWAN (*Low Power Wide Area Network*) communication protocol and system architecture for the network built on the LoRa physical layer. Communication protocol and system architecture have the most influence in determining the battery lifetime of an end device, as well as the network capacity, the quality of service, the security, and the variety of applications served by the network. [4]

LoRaWAN utilizes a star network architecture, as seen in Figure 2. In contrast to a mesh network, where the individual end devices forward the information to other end devices to increase communication range, LoRa's long range connectivity is used for battery lifetime preservation. [4]

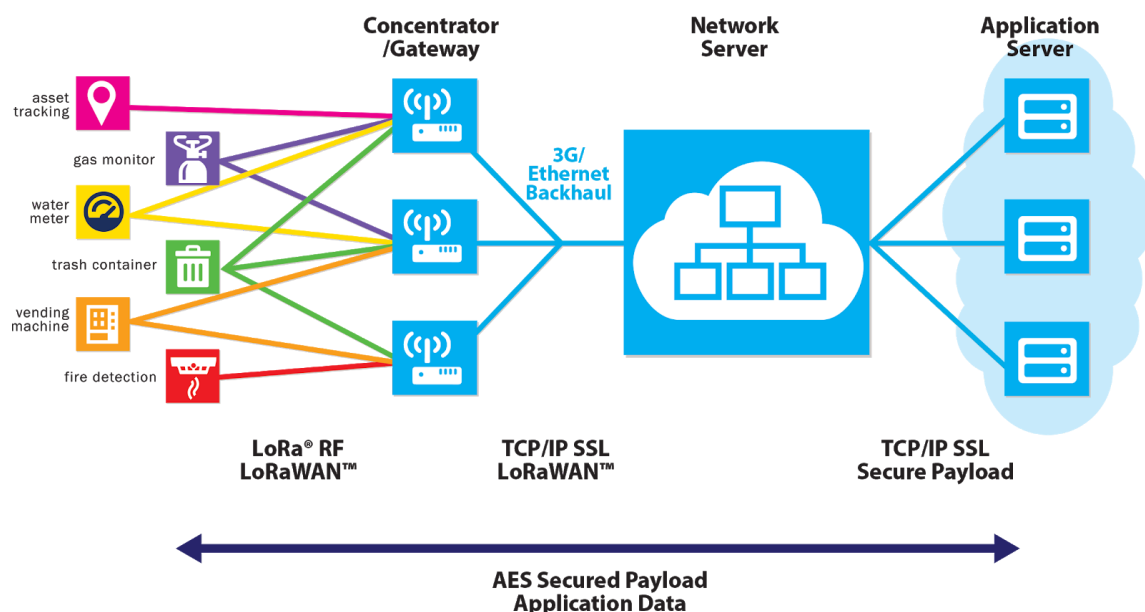


Figure 2: LoRaWAN network architecture [4]

The end devices are not associated to a specific gateway. In fact, the data transmitted by an end device can be received by multiple gateways. Each of those gateways will then forward the received messages to a cloud based network server (for example The Things Network or a privately hosted server). These network servers perform the more complex tasks like filtering redundant received packets, perform security checks, and schedule acknowledgements. This means, that the sensor can be deployed anywhere where it can reach a gateway and be moved without the need for reconfiguration. [4]

LoRaWAN end devices work asynchronous and send the message when they have data ready to send, whether this is scheduled, or event driven. They don't have to synchronize with the network to check for downlink messages. This reduces the 'awake' time of the end device and significantly reduces energy consumption. The drawback with this is, that the messages to be send to an end device must wait in the network server until the end device opens a downlink receive window, which is usually timed shortly after the end device send an uplink message, as seen in Figure 3 in [chapter 2.1](#). [4]

#### 1.2.4 The Things Network

The Things Network (TTN) is a crowdfunded open LoRaWAN network, a cloud service, and a community, whose goal it is to build a global network of open LoRaWAN Gateways. As of January 2018, there are more than 1900 gateways in over 87 countries connected to TTN, all of which are run and financed by members of the TTN community. Connection to and usage of the TTN cloud services are free of charge. [6]

The Things Network runs a backend system that is positioned between the gateways and the application and takes care of routing and processing in a decentralized way. To



achieve this, the backend is split up in four components: router, broker, network server, and handler. In the LoRaWAN network architecture, TTN provides network server and application server functions. [6]

### 1.2.5 SigFox

SigFox is a French telecommunication company that operates a proprietary communication network with the same name. Just like LoRaWAN, SigFox is a LPWAN designed for low-power and long-range communication. In Europe it operates in the 868 MHz band and in the 915 MHz band in North America with an extremely narrow bandwidth of 100 Hz. Its range is up to 10 km in urban and 50 km in rural environments. SigFox is based on Random Frequency and Time Division Multiple Access (RFTDMA) and has a hard restriction on the allowed amount of data send. Each data packet may not exceed the size of 12 byte and the limit is 14 packets per end device a day. [7]

In contrast to LoRaWAN, SigFox is not open source and the SigFox company owns and operates the network. Coverage includes most western European countries and parts of the United States, New Zealand, and South Africa. [7]

## 1.3 SensorCloud

SensorCloud is a research project concluded in 2014 that is employed in the OpenAir Cologne project. It deals with the development of technologies which allow cloud applications to gather and control sensor data and actors. Also, it appropriately processes the data collected by a plethora of sensors. The SensorCloud project is a corporation of the University of Applied Science Cologne, the RWTH Aachen, the telecommunication company QSC, and the software company Symmedia and is financed by the Federal Ministry for Economic Affairs and Energy of Germany in line with the research program 'Trusted Cloud'. [8]

## 2 Examining LoRaWAN And The Things Network

This chapter examines the functionality of LoRaWAN and The Things Network as well as reviewing the advantages of LoRaWAN as advertised by the LoRa Alliance, most notably long range capabilities and low power consumption. [4]

First it looks at the LoRaWAN specifications and outlines the three different device classes, the methods of activation with which an end device joins the network, the LoRaWAN packet structure, and the encryption of LoRaWAN packets. Then the service range is estimated in the example of the city of Cologne, the energy consumption of one of the two example end devices is measured, and the time on air of LoRaWAN packets and the resultant bit rate is calculated. Finally, it reviews the structure, properties, and functions of The Things Network.

### 2.1 Device Classes

End devices in a LoRaWAN environment serve different purposes and therefore may have different requirements. To accommodate this, the LoRaWAN specification describes three different classes of devices, that trade off downlink capabilities for battery lifetime [9]. All classes higher than Class A are based on Class A and thus implement at least this functionality. Figure 3 displays the receive windows of the different device classes.

**Bi-directional end-devices (Class A):** This class of devices is designed for battery operated end devices, with maximum battery lifetime in mind. After each end devices uplink message, it opens two short downlink receive windows. Therefore, the end device is only able to receive downlink messages after it has send an upload. Downlink messages arriving at the network server are queued and scheduled to be send after the next uplink message arrives at the network server. [10]

**Bi-directional end-devices with scheduled receive slots (Class B):** Class B devices open downlink receive windows at scheduled times in addition to the receive windows opened after an uplink message (Class A). At the scheduled times, the gateway broadcasts beacon for end devices to sync. This way the end devices can keep track when to open receiving windows. [10]

**Bi-directional end-devices with maximal receive slots (Class C):** Class C devices have almost always an open downlink receive window. It is only closed when the device is transmitting an uplink message. As this is a huge drain on batteries, because the chip must always be active, this class is only meant for devices with a constant power line connection. [10]

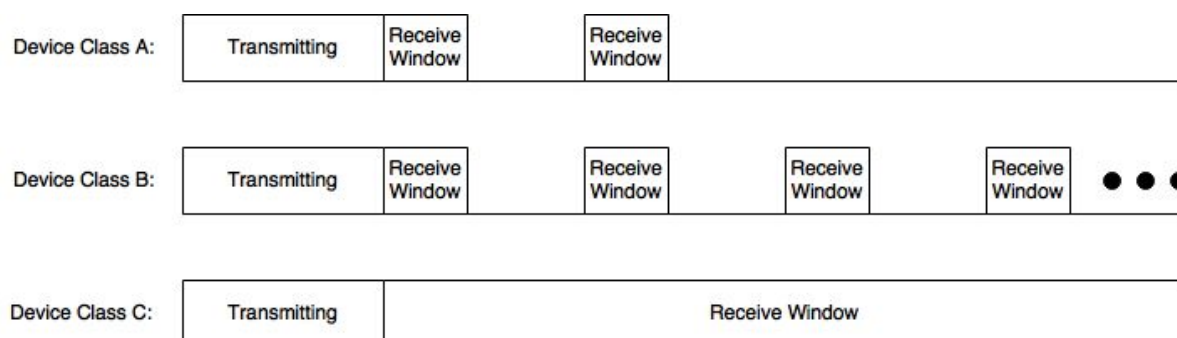


Figure 3: Receive windows in different device classes

## 2.2 Activation Methods Of End Devices

The LoRaWAN specification provides two methods for an end device to join the network: over-the-air activation (OTAA) and activation by personalization (ABP). Both have different advantages and disadvantages with regarding security and complexity. [10]

After a successful joining of an end device, it has the following information stored: a device address (DevAddr), an application identifier (AppEUI), a network session key (NwkSKey), and an application session key (AppSKey). [10]

- **DevAddr:** The DevAddr is a 32 bit logical address, equivalent of an IP address. It is formatted as in Figure 4.

Bit #	[31..25]	[24..0]
DevAddr bits	NwkID	NwkAddr

Figure 4: LoRaWAN device address

The most significant 7 bits are used as a network identifier (NwkID), which vary in different regions and companies (respectively network managers) [10]. For example, The Things Network uses the 0x26 and 0x27 prefix, while KPN, the Dutch telecommunication agency, holds the 0x14 and 0x15 prefix [11]. The least significant 25 bits are the network address (NwkAddr) of the end device, which are arbitrarily assigned by the network manager. [10]

The DevAddr is not unique to a single end device. In fact, The Things Network can assign the same DevAddr to hundreds of different end devices. It can

differentiate between the end devices by matching the message integrity code (MIC). [10]

- **AppEUI:** This is a 64 bit global application identifier which is used to group end devices. The AppEUI uniquely identifies one application, though it is possible to assign more than one AppEUI a single application. [10]
- **NwkSKey:** The NwkSKey is an AES 128 bit key specific to a single end device. It is used to encrypt the payload between the end device and the network server. The NwkSKey is used both by the end device and the network server to calculate the message integrity code (MIC) in order to verify and ensure data integrity. [10]
- **AppSKey:** Like the NwkSKey, the AppSKey is an AES 128 bit key specific to a single end device. It is used for application layer security. Both the application server and the end device use the AppSKey to encrypt their payloads while communicating with each other. [10]

### 2.2.1 Over-The-Air Activation (OTAA)

With this method, the end device executes a join procedure before sending the first data. An end device has to repeat this step every time it loses the session information, for example after a restart. To engage in the join procedure an end device needs to be deployed with three extended unique identifiers: a device EUI (DevEUI), an application EUI (AppEUI) and an application key (AppKey). [10]

- **DevEUI:** This is a 64 bit globally unique identifier to identify one single end device. In principle it is factory set and cannot be changed, though some development boards allow for modification. [10]
- **AppKey:** The AppKey is a unique AES 128 bit key specific to an end device, that is shared between the end device and the network. It is used to derive the NwkSKey and the AppSKey. [10]

The end device sends a join-request message including the AppEUI, the DevEUI followed by a randomly generated two byte value DevNonce as shown in Figure 5.

Size [byte]	8	8	2
Join-request bits	AppEUI	DevEUI	DevNonce

Figure 5: join-request payload

These three values are signed with a four byte message integrity check (MIC) which is calculated from the AppKey, AppEUI, DevEUI, and the DevNonce. The join-request message is not encrypted. [10]

When receiving a join-request message the network server in turn calculates the MIC autonomously with the AppKey it has stored and, if valid, may answer the end device

with a join-accept message like in Figure 6. If the join-request is not accepted, no acknowledgement is send by the network server. For the join-accept message, the network server randomly generates a three byte value called AppNonce. With the AppNonce the end device calculates the app session key (AppSKey) and the network session key (NwkSKey). Additionally, the join-accept message includes a network identifier (NetID), the device address, which the network server assigns to the end device (DevAddr), a DLSettings field that contains information about RX-offset and -data rate, a delay between TX and RX (RxDelay), and an optional list of channel frequencies for the network the end device is joining (CFList). [10]

Size [byte]	3	3	4	1	1	16 (optional)
Join-accept bits	AppNonce	NetID	DevAddr	DLSettings	RxDelay	CFList

Figure 6: join-accept payload

With the AppNonce, the AppKey, the NetID, and the DevNonce, the end device is able to calculate the two session keys NwkSKey and AppSKey. The least significant 7 bits of the NetID are the network identifier and match the most significant 7 bits of the DevAddr. Finally, the join-accept message is encrypted with the AppKey. [10]

This method is regarded as the more secure, since the NwkSKey and the AppSKey are not permanently saved on the device and renegotiated each time the node is activated. [10]

### 2.2.2 Activation By Personalization (ABP)

This method skips the complete join-request and join-accept procedure. The device address (DevAddr), the network session key (NwkSKey), and the application session key (AppSKey) are directly stored into the end device. No device identifier (DevEUI), application identifier (AppEUI), or application key (AppKey) is needed. [10]

Since the end device is already equipped with the NwkSKey and AppSKey, it can only connect to a specific LoRaWAN network which has to be configured at the time of writing the end devices software (Note: this does not mean only one specific gateway). [10]

This approach is regarded as the easier method, because the join-request and join-accept procedure can be skipped. But as the NwkSKey and AppSKey are hard coded into the end device it may be possible to readout that information and then for other devices to impersonate that end device. [10]

## 2.3 LoRaWAN Packet Structure

This section takes a look at the message structure of LoRaWAN uplink and downlink messages.

### 2.3.1 Uplink Messages

Uplink messages are sent by end devices to the network server and picked up by one or more gateways. The physical message, meaning the LoRa radio signal as defined by the LoRaWAN specification is shown in Figure 7. It contains a preamble, the LoRa physical header (PHDR), plus a header cyclic redundancy check (PHDR\_CRC). These are followed by the payload (PHYPayload), which is also protected by an own cyclic redundancy check (CRC). [10]

Preamble	PHDR	PHDR_CRC	PHYPayload	CRC
----------	------	----------	------------	-----

Figure 7: PHY layer uplink structure

The PHYPayload contains a 1 byte MAC header (MHDR), the MAC payload (MAC payload), and an 8 byte message integrity code (MIC). As Figure 8 shows, the MACPayload is subsidized with join-request or join accept data during the activation process. [10]

MHDR	MACPayload	MIC
------	------------	-----

MHDR	join-request	MIC
------	--------------	-----

MHDR	join-accept	MIC
------	-------------	-----

Figure 8: PHYPayload structure

The join-request and the join-accept are described earlier in [chapter 2.2](#). The maximum length of the MACPayload is region specific and can be found in the LoRaWAN Regional Parameters [12]. For example, in the EU 863 - 870 MHz ISM Band, used across Europe, the maximum MACPayload size is between 51 and 222 bytes, depending on the spreading factor and the bandwidth. [10]

Figure 9 shows the MHDR. It contains the message type (MType) in the three most significant bits and the major version number (Major) of the frame format in the two least significant bits. The remaining bits 4 to 2 are reserved for future use. [10]

Bit #	[7..5]	[4..2]	[1..0]
MHDR bits	MType	RFU	Major

Figure 9: MAC header structure

The MType distinguishes between six different message types: join-request or join-accept, unconfirmed data up or down, and confirmed data up or down. As of now, the LoRaWAN specification defines only one major version of the frame format. [10]

The MACPayload field which is shown in Figure 10 can also be called data frame. It contains a frame header (FHDR) followed by the port field (FPort) as well as the frame payload field (FRMPayload). The former two of these are optional. [10]

FHDR	FPort	FRMPayload
------	-------	------------

Figure 10: MACPayload structure

The FHDR contains information about the address of the end device (DevAddr), a frame control byte (FCtrl), two frame counter bytes (FCnt), and an optional, up to 15 bytes long frame options field (FOpts). [10]

Size [byte]	4	1	2	[0..15]
FHDR	DevAddr	FCtrl	FCnt	FOpts

Figure 11: FHDR structure

The 1 byte sized FPort lets the end device categorize the data. The application is able to handle the data differently depending on the port the data is sent to. Finally, the FRMPayload contains the data, the end device has sent. [10]

### 2.3.2 Downlink Messages

A downlink message is sent by the network server to a single end device, forwarded by a single gateway. The LoRaWAN specification defines them similar to the uplink message. It contains a preamble, a LoRa physical header (PHDR), a header CRC (PHDR\_CRC), and a payload (PHYPayload). A payload integrity check is omitted at this level. This is necessary to keep messages as short as possible to not strain any duty-cycle limitations of the ISM bands while transmitting. [10]

Preamble	PHDR	PHDR_CRC	PHYPayload
----------	------	----------	------------

Figure 12: PHY layer downlink structure

The PHYPayload field contains the same data structure as described in in [chapter 2.3.1](#).

## 2.4 Security

Security is a big issue in a radio protocol, since every device able to receive the radio signal is potentially able to read it. As shown in [chapter 2.3.1](#), each packet is secured with two 128 bit security keys: the network session key (NwksKey) and the application

session key (AppSKey). If the end device is dynamically activated with over-the-air activation (OTAA), both keys will be newly generated with every device activation. If the end device is activated with ABP, these keys are hard coded on the end device. [10]

The NwkSKey is used for encryption and decryption of the MAC payload between the end device and the network. It is further used by the network server as well as the end device to calculate the message integrity code (MIC) for data integrity. [10]

The AppSKey is used by both, the application server and the end device to encrypt and decrypt the application payload field. Unlike the MAC payload, the application payload is not integrity checked. This means, that the network server may be able to alter the application payload. While these network servers are considered to be trusted in, true end-to-end encryption is not in the scope of the LoRaWAN specification. [10]

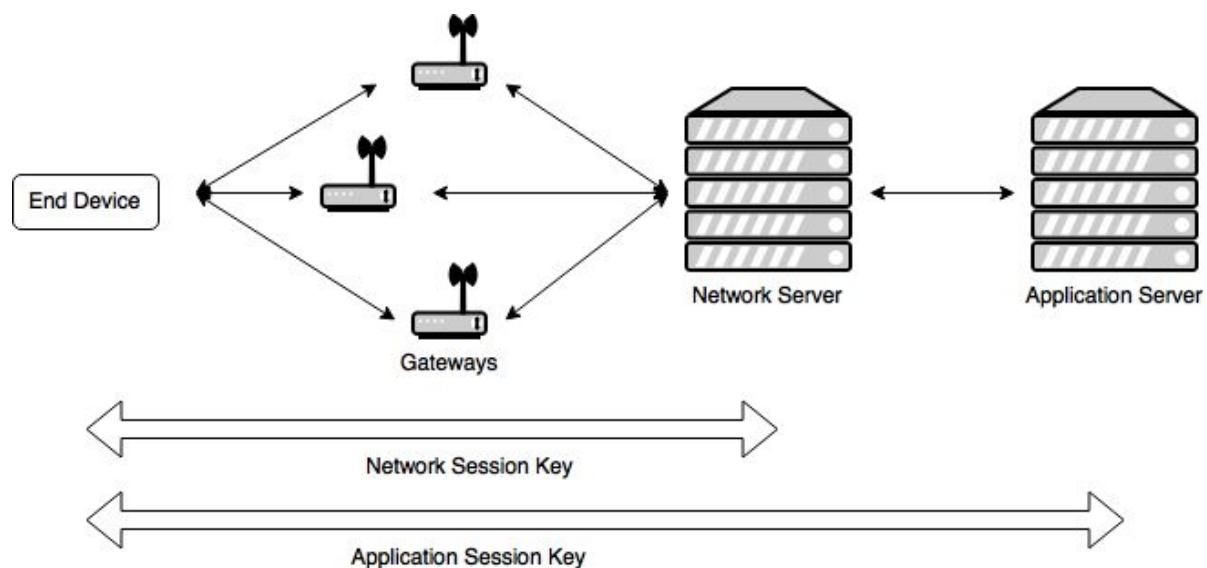


Figure 13: LoRaWAN security

In The Things Network (TTN), the application server is the handler component. It is possible to run private application servers and connect them to TTN to implement end-to-end security this way.

## 2.5 LoRa Service Range

One of the great benefits of LPWAN and LoRaWAN is its high service range. The LoRa Alliance advertises a range of more than 15 km in suburban environments and up to 5 km in urban environments [4]. If the coverage of one gateway is known, it is easier to install further gateways with lesser overlap.

Before installing the gateway and measuring the service range in the field, it is a good first step to roughly calculate the service range. With the wide range LoRa is supposed to cover, it is easier to check the true range out in the open if an estimate is available.



The Okumura-Hata Model is a radio wave propagation model to calculate the path loss of cellular transmissions in urban environments [13]. It is based on the Okumura Model for radio propagation in urban environments, but is able to differentiate between rural and suburban environments, as well as small, medium, and big cities. Possible parameters of the model are transmission frequencies of 150 MHz to 1500 MHz, point-to-point as well as broadcast communication, a mobile station antenna height between 1 m and 10 m, base station antenna heights of 30 m to 200 m, and link distances from 1 km to 10 km.

$$L_U = 69.55 + 26.16 \log_{10} f - 13.82 \log_{10} h_B - C_H + [44.9 - 6.55 \log_{10} h_B] \log_{10} d \quad (2.1)$$

Where:

- $L_U$ : path loss in urban areas in [dB]
- $h_B$ : height of base station antenna in [m]
- $h_M$ : height of mobile station antenna in [m]
- $f$ : frequency of transmission in [MHz]
- $d$ : distance between the base and mobile stations in [km]
- $C_H$ : antenna height correction factor, for medium sized cities<sup>1</sup>:

$$C_H = 0.8 + (1.1 \log_{10} f - 0.7) h_M - 1.56 \log_{10} f \quad (2.2)$$

In this experimental LoRa network the gateway will be placed on the ninth floor on the west wing of the IWZ Deutz building of the TH Cologne. For mobile stations to erect, an ideal position is on top of street lights. In Germany, the vertical structural clearance on main roads is 4.50 m. Any structure craning partly or completely into the clearance limit have to be labeled as such [14]. So, it can be assumed overarching street lights are at least that height.

In Europe, LoRa uses the 868 MHz ISM-Band. So, the following three values are given:

$$h_B = 30 \text{ m}$$

$$h_M = 4.5 \text{ m}$$

$$F = 868 \text{ MHz}$$

Accordingly, the antenna height correction factor (2.2) is:

$$\begin{aligned} C_H &= 0.8 + (1.1 \log_{10} 868 - 0.7) 4.5 - 1.56 \log_{10} 868 \\ C_H &= 0.8 + (1.1 \cdot 2.94 - 0.7) 4.5 - 1.56 \cdot 2.94 \\ C_H &= 7.61 \end{aligned}$$

---

<sup>1</sup> It is debatable if Cologne is a medium or big city. Cologne is indeed a big city, but there are no skyscrapers (buildings 150 m or higher) in Cologne and the administration forbids any new buildings higher than 22,50 m in the inner city.

The result then can be put inside the Okumura-Hata Model (2.1):

$$\begin{aligned}
 L_U &= 69.55 + 26.16 \log_{10} 868 - 13.82 \log_{10} 30 - 7.61 + [44.9 - 6.55 \log_{10} 30] \log_{10} d \\
 L_U &= 69.55 + 26.16 \cdot 2.94 - 13.82 \cdot 1.48 - 7.61 + [44.9 - 6.55 \cdot 1.48] \log_{10} d \\
 L_U &= 118.40 + 35.22 \cdot \log_{10} d \\
 \log_{10} d &= \frac{L_U - 118.40}{35.22}
 \end{aligned} \tag{2.3}$$

One of the more common The Things Network gateways, as well as the one installed at the TH Köln for this thesis, is a Raspberry Pi with an IMST iC880A-LoRaWAN concentrator ([see chapter 3.2.1](#)). It is specified with a sensitivity of down to -138 dBm [15], which is the minimum received signal strength indication (RSSI) in this system. The SX1272MB2DAS used as a test object in the scope of this thesis, has an effective radiated power (ERP) of +17 dBm [16]. These two values help to calculate the maximum path loss  $L_U$  possible, enable the device to receive a data packet.

$$\begin{aligned}
 L_U &= ERP - RSSI \\
 L_U &= 17 \text{ dBm} + 138 \text{ dBm} \\
 L_U &= 155 \text{ dBm}
 \end{aligned} \tag{2.4}$$

This data put in the shortened Okumura-Hata Model (2.3) above results in:

$$\begin{aligned}
 \log_{10} d &= \frac{155 - 118.40}{35.22} \\
 \log_{10} d &= 1.04 \\
 d &= 10^{1.04} \\
 d &= 10.94 \text{ km}
 \end{aligned}$$

The outcome according to the Okumura-Hata Model is that an end device at 4.50 m height in urban environment is able to send packets to a gateway up to 10.94 km distance, if that gateway is mounted in a height of at least 30 m.

The calculated distance is off the model's capacity of link distance by almost 1 km or one tenth of the maximum distance, so the result may be imprecisely. It is notable though, that the model predicts a service range over 10 km. Thus, it would be possible to cover the whole inner city of Cologne, by just one Gateway as Figure 17 shows.

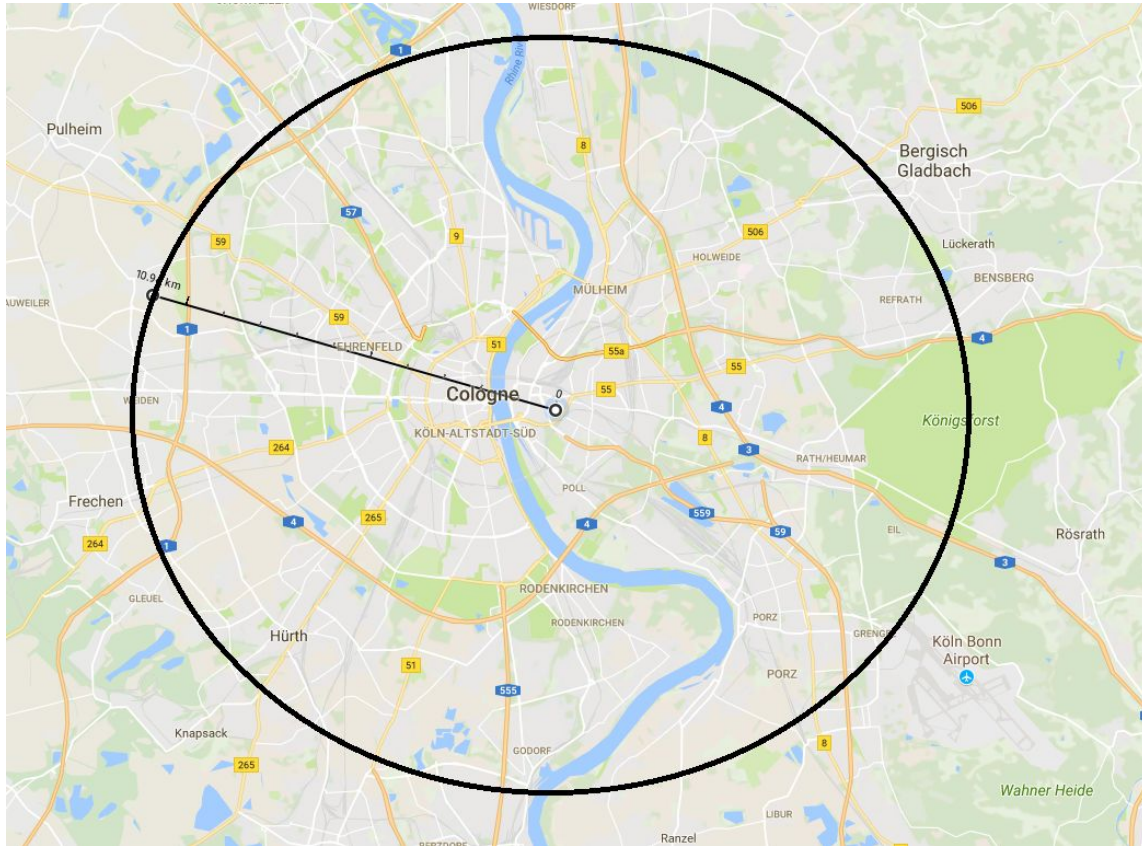


Figure 17: estimated LoRa service range in Cologne

The distance one gateway can cover, according to the Okumura-Hata Model, is highly dependent on the height of the mobile station. This makes sense, the higher the antenna of the mobile station, the less radio waves will be obscured by buildings or other common objects like trees, people, or vehicles. Table 1 shows the calculated distances depending on the height of the mobile station.

<i>Height of mobile station antenna in [m]</i>	Distance between the base and mobile stations in [km]
1	6.16
2	7.08
3	8.51
4	10.00
5	11.75
6	14.13

Table 1: LoRa range depending on the mobile station antenna height

## 2.6 Estimating Battery Life Time

Beside the long range capability, a high lifetime of battery powered end devices is one of LoRaWANs big advantages advertised by the LoRa Alliance [4]. This chapter looks at the energy consumption of the OpenAir Node V1.1 (see [chapter 3.1](#)) and calculates the battery life. The OpenAir Node has a measurement cycle of fifteen minutes. So, every fifteen minutes, it takes NOx, humidity and temperature data and sends a LoRaWAN message. Thirty seconds before measuring, the MiCS-4514 Gas Sensor has to preheat its heating plates.

To measure the energy consumption, a precise  $1\ \Omega$  resistor is put between the OpenAir Node and the power supply with an oscilloscope showing and recording the voltage over the resistor over a thirty minute period. The result is to be seen in Figure 18. Resulting Ohm's Law  $U = R \cdot I$ , the value for the current is equal to the voltage measured. The two markers are at the peaks that indicate the time where the LoRa Message is transmitted and the RFM95W consumes the most energy. Shortly before each of the markers a slightly increased energy consumption can be noticed. This is due to the heater inside of the MiCS-4514 Gas Sensor.

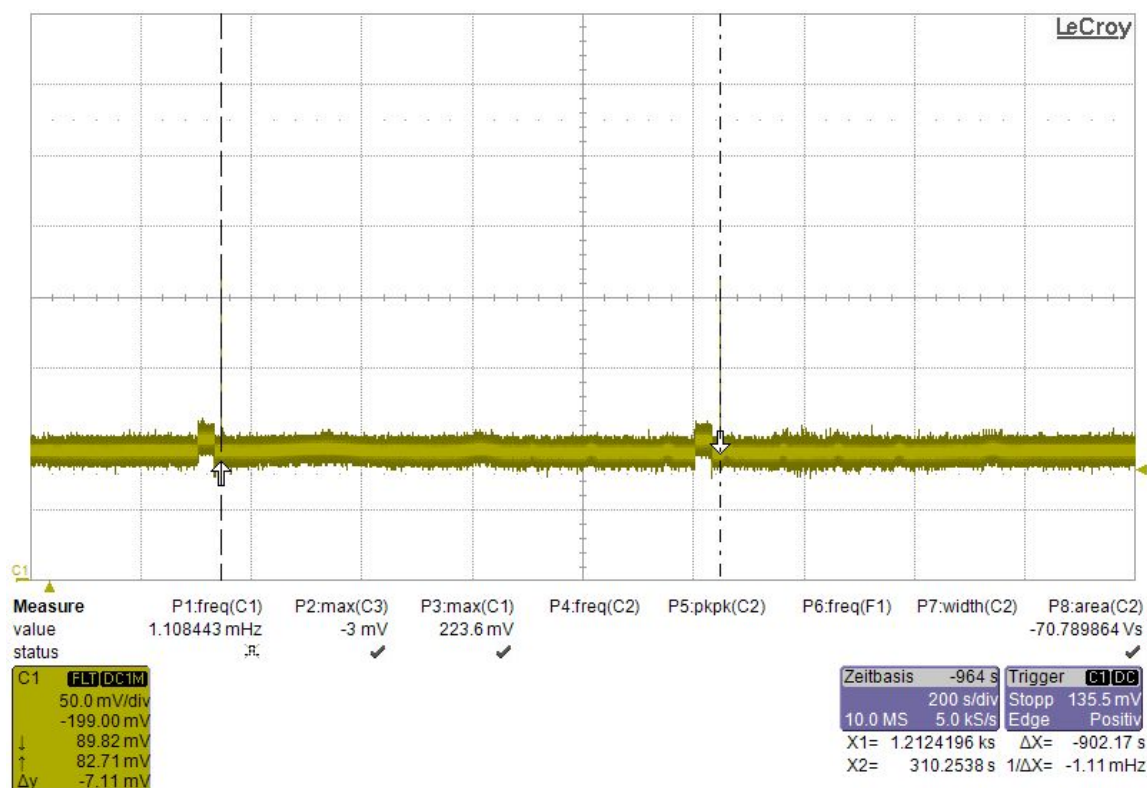


Figure 18: Thirty minutes measurement of energy consumption

Each horizontal div is 50 mV, each vertical div is 200 s, so the entire graph shows a measurement of about 33 minutes, or slightly longer than two measurement cycles of the OpenAir Node. In Figure 19, which zooms into the thirty second heating period, it

can be seen, that the average current running through the node is 90 mA. Measuring the area under the graph between the two markers of Figure 18 gives the electric power the OpenAir Node consumes during one measurement cycle. It can be rounded to 81000 mAs for a fifteen minute period, which results to 324000 mAs in an Hour, or 90 mAh. This means, that the OpenAir Node with a fifteen minute measurement cycle and powered by three commercially available AA Mignon batteries with 2700 mAh each, will last for about 27 hours, or just a little bit longer than a day.

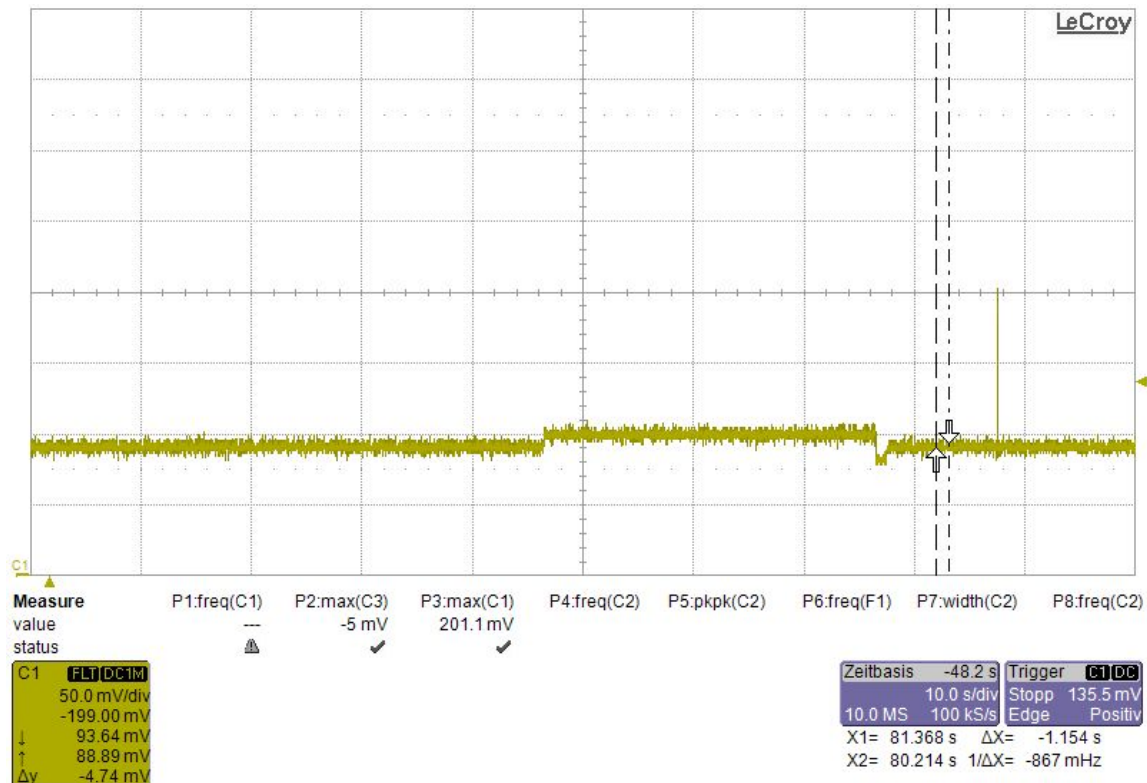


Figure 19: Zoom on heating period

Hoperf, the manufacturer of the LoRaWAN transceiver, states the energy consumption of their RFM95W in the datasheet with 120 mA while transmitting, which is verified with this measurement [17]. Figure 19 shows the peak in energy consumption on the far right, which is the RFM95W chip transmitting the LoRa packet. The examined packet shown in the Figure had a low spreading factor of 7 (see [chapter 2.6](#)). It had a time on air of about 60 ms, so the increased energy consumption of 12 mAs while transmitting can be neglected. Bigger packets with the highest spreading factor, for example, have a time on air of about 2800 ms which results in an increase of energy consumption of about 550 mAs per cycle. This increases the energy consumption by 2200 mAs, or 0.61 mAh per hour. While not in transmitting or receiving mode, the RFM95W is in idle mode, with a power consumption of 1.5  $\mu$ A. So most of the time, the power consumption of the transceiver module can be neglected [17].

Another point to make it that the reprogrammaning of the OpenAir Node was not optimized for battery run time as it was merely a proof of concept for LoRaWAN capabilities. The usage of the deep sleep mode in the ESP32 could drastically lower the power consumption during the time the OpenAir Node is inactive.

## 2.7 Estimating Packet Time On Air And Bit Rate

LoRa trades high range and low power consumption for low data rate [4]. For the planning of different applications, it may be important to know how many bytes can be send and how often it is allowed to send them. This chapter discusses the formulas to calculate the time on air as well as the bit rate and shows them in an example.

LoRas spread spectrum modulation transmits packets with a spreading factor (SF) between 7 and 12, from best to worst conditions [5]. The spreading factor is the quotient of the chip rate and the symbol rate (the symbol rate is also known as baud rate). A chip in digital communications is basically a message bit multiplied by a waveform carrier. So, a higher SF means more chips are being used to represent one symbol, resulting in higher battery consumption, higher range, and higher packet transmission time or time on air. The standard for the EU 868 ISM band, where LoRaWAN operates in the European Union, the duty cycle of end devices is restricted to 1% [18]. This means that the maximum transmission time of end devices is restricted to 1% of time, or 36 seconds per hour. Semtech provides a formula for calculating the time on air of LoRa packets [19]:

$$T_{packet} = T_{preamble} + T_{payload} \quad 8 \quad (2.5)$$

The time on air  $T_{packet}$  for a packet is the sum of the time on air for the preamble and the payload. The elements of a LoRa packet is shown in Figure 20.

Preamble	PHDR	PHDR_CRC	PHYPayload	CRC
----------	------	----------	------------	-----

Figure 20: Elements of a LoRa packet

The formula to calculate the time on air for the preamble  $T_{preamble}$  is as follows:

$$T_{preamble} = (n_{preamble} + 4.25) \cdot T_{sym} \quad (2.6)$$

Where  $n_{preamble}$  is the number of preamble symbols and  $T_{sym}$  is the symbol duration. The number of preamble symbols is defined in the LoRaWAN Regional Parameters as 8 symbols in all regions, although when using plain LoRa this may differ [12]. The symbol duration is the time it takes to send  $2^{SF}$  chips at the given bandwidth:

$$T_{sym} = \frac{2^{SF}}{BW}$$

(2.7)

According to the LoRaWAN Regional Parameters, the data rate limits in the EU 863-870 MHz ISM band region is a bandwidth of 125kHz with a spreading factor (SF) of 7 to 12 or a bandwidth of 250kHz with a SF of 7, totaling to seven different data rate configurations [12]. The time on air for the payload  $T_{\text{payload}}$  is, like  $T_{\text{preamble}}$ , the number of symbols in the payload multiplied by the symbol duration:

$$T_{\text{payload}} = n_{\text{payload}} \cdot T_{\text{sym}} \quad (2.8)$$

$n_{\text{payload}}$  is, according to the Semtech LoRa Modem Designer's Guide defined as [19]:

$$n_{\text{payload}} = 8 + \max(\text{ceil}(\frac{8PL-4SF+28+16-20H}{4(SF-2DE)})(CR+4), 0) \quad (2.9)$$

Where:

PL: number of payload bytes

SF: the spreading factor

H: is 0 when header is enabled, else 1

DE: is 1 when low data rate optimization is enabled, else 0

CR: coding rate

One has to keep in mind, that PL in this case describes the number of PHYPayload bytes. This means for LoRaWAN packets it is important to add 13 bytes to PL to account for the LoRaWAN header. The LoRaWAN specification also states, that LoRaWAN always uses explicit headers, which means when accommodating for an LoRaWAN environment, H is 0 [10]. An enabled DE is advisable at a SF of 11 or 12, but not mandatory. The coding rate is an indicator for the proportion of redundancy in telecommunication messages for error correction. It is represented by a number between 1 and 4, where 1 is the lowest amount of error correction.

As an example, the time on air for a LoRaWAN packet can be calculated. With an assumed data rate configuration of SF12 / 125kHz and enabled low data rate optimization, which would be the longest time on air configuration. The assumed MACPayload is 10 bytes, subsequently the PHYPayload will be 23 bytes. First the symbol duration (2.7) is calculated:

$$\begin{aligned} T_{\text{sym}} &= \frac{2^{12}}{125\text{kHz}} \\ T_{\text{sym}} &= \frac{4096}{125000\frac{1}{s}} \\ T_{\text{sym}} &= 0.0328\text{s} \end{aligned}$$

With the 8 preamble symbols,  $T_{\text{sym}}$  can be used in (2.6) to calculate the preamble time on air:



$$T_{preamble} = (8 + 4.25) \cdot 0.0328s$$

$$T_{preamble} = 0.4018s$$

The second half of the packet time on air is the payload. For this,  $n_{payload}$  must be calculated 2.9:

$$n_{payload} = 8 + \max(\text{ceil}(\frac{8 \cdot 23 - 4 \cdot 12 + 28 + 16 - 20 \cdot 0}{4(12 - 2 \cdot 1)})(1 + 4), 0)$$

$$n_{payload} = 8 + \max(\text{ceil}(4, 5)(5), 0)$$

$$n_{payload} = 8 + 25$$

$$n_{payload} = 33$$

Inside 2.7,  $n_{payload}$  gets  $T_{payload}$ :

$$T_{payload} = 33 \cdot 0.0328s$$

$$T_{payload} = 1.0824s$$

This gives the total packet time on air of 2.5:

$$T_{packet} = 0.4018s + 1.0824s$$

$$T_{packet} = 1.4842s$$

In this example, the LoRaWAN packet has an calculated time on air of just under 1,5s.

Figure 21 shows a 10 byte LoRaWAN packet, captured by software defined radio (SDR). The timestamps on the left roughly confirm the calculations above

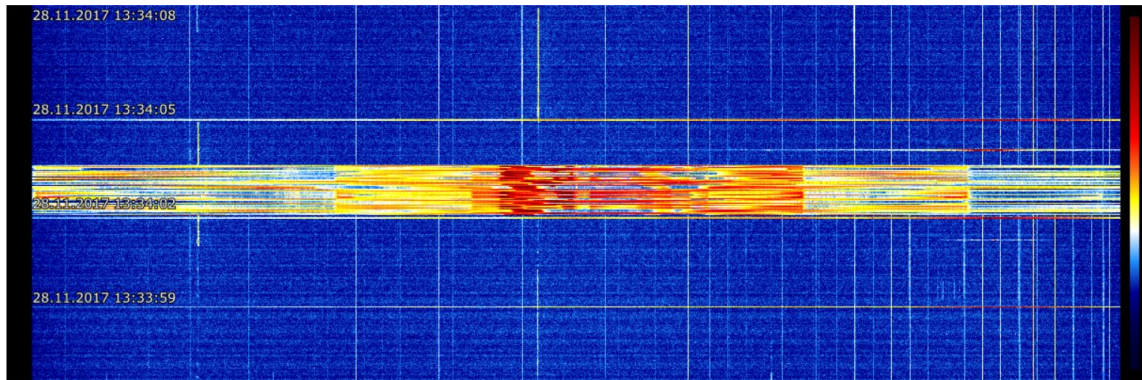


Figure 21: LoRaWAN packet captured with SDR



The time on air can also be seen in the TTN Gateway tab in the TTN console (see [chapter 3.2.4](#)). Figure 22 shows a 23 byte packet, resulting on 10 byte MACPayload plus 13 byte LoRaWAN header, arriving at the gateway. This proves the calculation correctness to 1/100 s.

time	frequency	mod.	CR	data rate	airtime (ms)	cnt		
15:04:16	868.3	loro	4/5	SF 12 BW 125	1482.8	11	dev addr: 26 01 29 16	payload size: 23 bytes

Figure 22: LoRaWAN packet captured by a gateway

To stay under the maximum duty cycle of 1%, this example end device is only allowed to send 24 packets per hour, or one packet every 148 seconds or two-and-a-half minutes per hour, which results in a bandwidth of 243 bytes on average. Of course, the time on air is highly dependent on the payload size as Figure 23 demonstrates. This means that the bit rate is directly depending on the payload size.

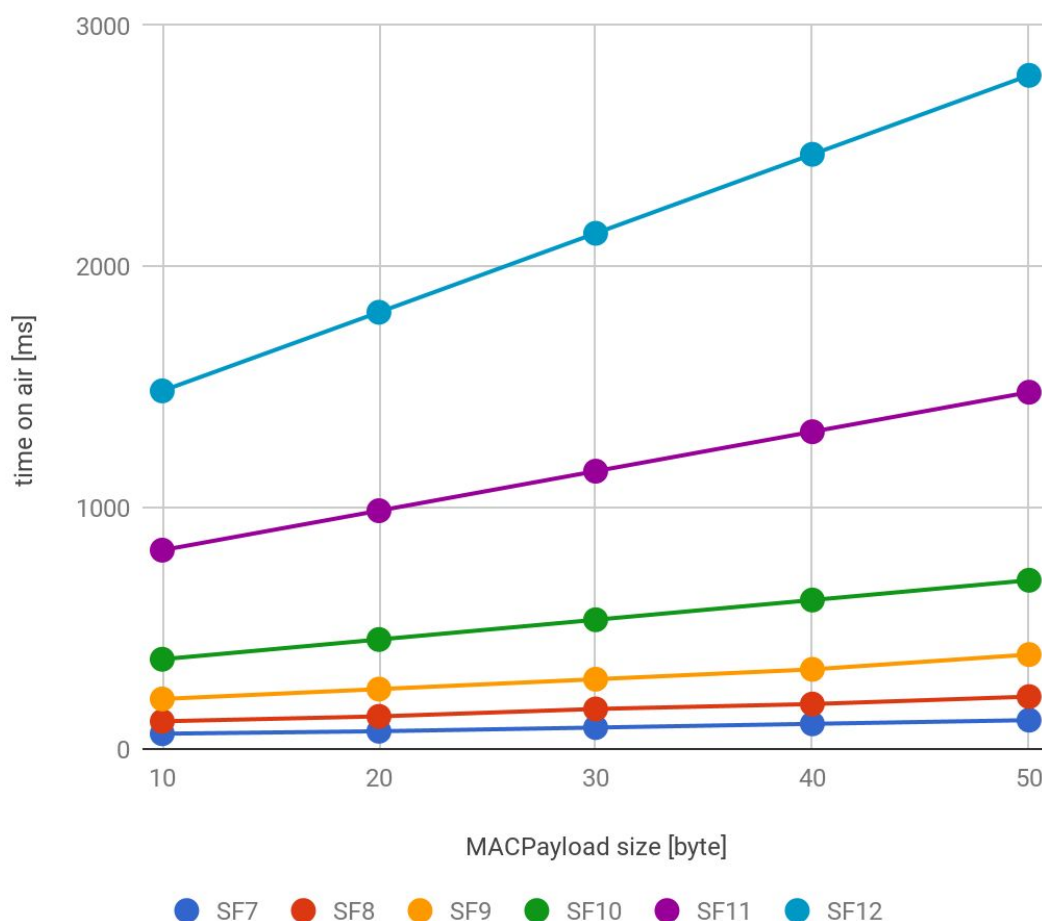


Figure 23: Time on air of LoRaWAN packets

In 2016 The Things Network specification was updated. Thomas Telkamp, a network architect for TTN introduced a fair access policy, with the goal that all TTN users are able to use the network reliably[20]. The stated golden rule is that a single end device may not have more than 30 seconds of time on air per day (up- and downlink), instead

of the 36 seconds per hour specified by the ETSI standard for radio equipment [18]. This ensures that each gateway is able to support 1000 end devices. If we assume that the end device sends packets like the example above, the end device is only allowed to send 20 packets per day, or roughly one packet every 72 minutes. Because the practical number of packets per day is depend on the time on air, a better connection with lower spreading factor means more packets can be send. Figure 24 shows the number of packets that can be send per day depending on the spreading factor and the payload size. As of now, the golden rule is a suggestion and not actively enforced by TTN. If more end devices are connected to TTN in the future, the enforcement will probably be stricter.

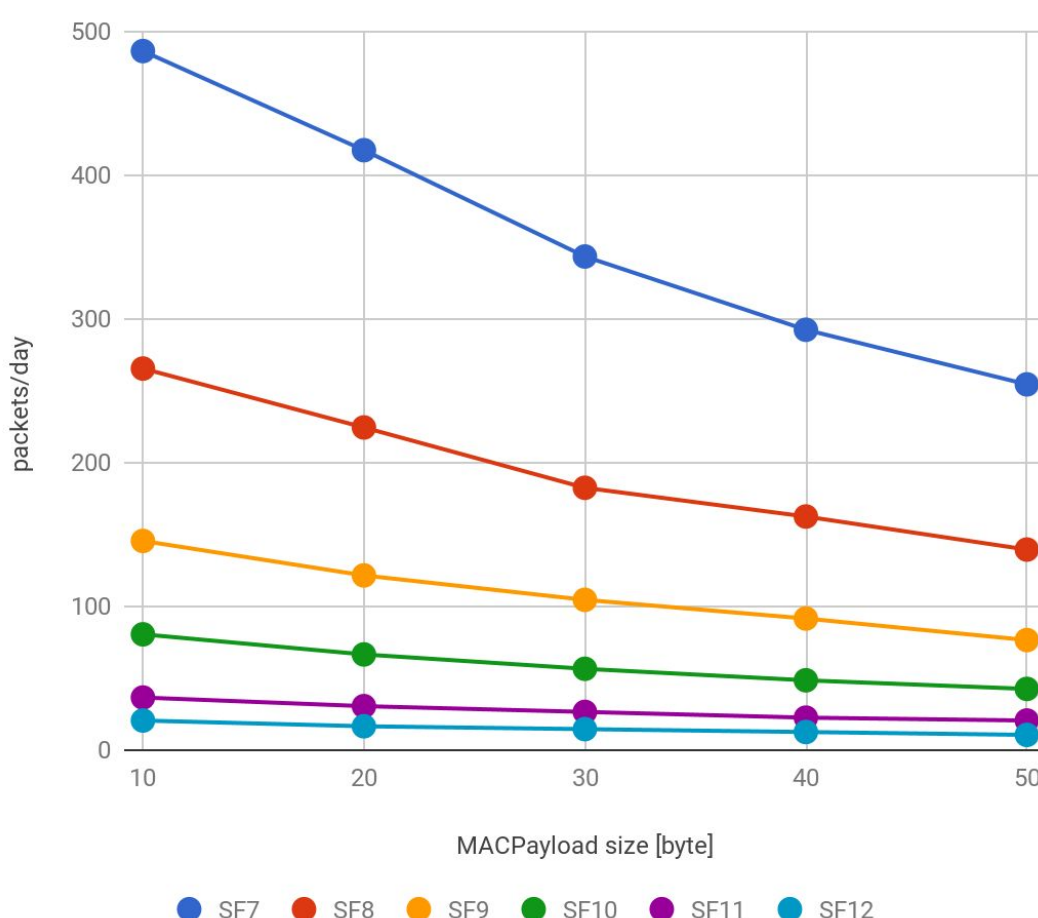


Figure 24: TTN packets per day under the fair access policy

To keep in mind though: these packets are limited by the fair access policy from TTN and do not need to be distributed evenly throughout the day. It is allowed to send all 20 packets within one hour, if the end device is silent during the rest of the day.

This calls for a smart use of packets, downlink as well as uplink. Therefore downlink packets are to be kept at a minimum and uplink packets should forgo receive confirmation unless there is no other option. It might be feasible for an end device to

first collect data over a longer period of time and then send a big chunk of data all at once. So a 5-fold increase in data, from 10 to 50 bytes, will only halve the number of packets allowed to be sent under the fair access policy. End devices also could vary the times at which they take measurements to catch the occurrence of more noticeable changes. For example, a NO<sub>x</sub> sensor could send more data during the morning and the afternoon, when there is higher traffic to gather the incline of nitrous oxide during the rush hours more detailed. During nighttime less data are to be collected and sent.

## 2.8 The Things Network Backend Architecture

The architecture of The Things Network consists of four major components as shown in Figure 25: router (R), broker (B), network server (NS), and handler (H). Although the router, broker, and handler are shown only once in the graphic, each of those may have multiple instances. Those are all connected to the discovery server (D) which in turn is used by these components to discover services within TTN. [21]

A LoRaWAN message broadcasted by a gateway is received by one or more other gateways. The gateway in between serves as a bridge, forwarding the messages. These bridges convert from LoRa radio protocol to HTTP (hypertext transfer protocol). A gateway always is connected to one router, although a router may be connected to many gateways. The router is responsible for all gateway related functions as well as region specific details. It manages the gateway's status and schedules transmissions. The routers are connected in a many-to-many relation with the brokers, which are the backbone of TTN. They are responsible to map end devices to an application, handling device addresses, and forwarding downlink messages to the correct router, which in turn forwards the messages to the correct gateway. The network server is responsible for the LoRaWAN functionality, like keeping track of device status or notifying the end device to enhance signal strength. The handler's responsibility is editing and forwarding the data of one or more applications. The handlers represent the application servers in the LoRaWAN as seen in [Chapter 1.2.3](#). They also decrypt and encrypt the messages and run the MQTT service to publish messages to the applications. [21]

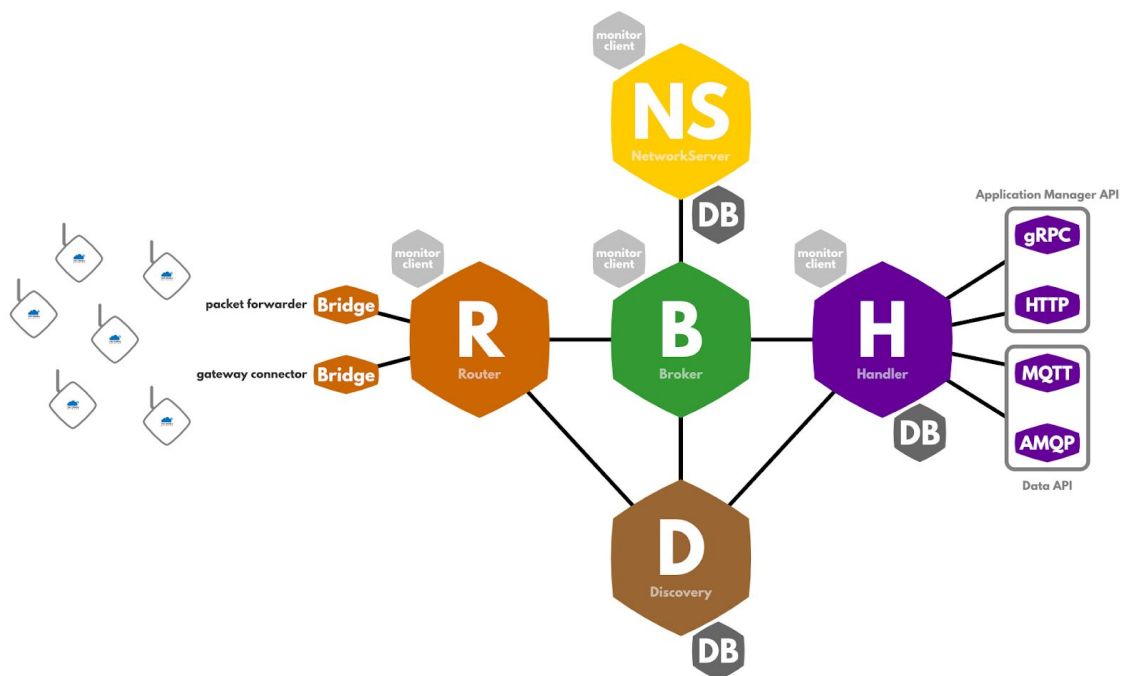


Figure 25: The Things Network Backend [22]

The program code for these components are open source. So it is possible to deploy private networks where all data remain within a private environment. Hybrid network deployment is also possible. As of now, a user can run his own handler and thus take care of encryption and decryption. A more complicated option planned but not yet implemented is a complete private network with own routers, brokers, network server, and handlers, that is able to exchange data with the public TTN network. For this, a link between private router to public brokers and vice versa is needed. By this the private network can offload traffic to the TTN network and use it as a backup. [21]

## 3 Developing The Distributed NOx And CO Measurement System

The LoRaWAN architecture is composed of four main elements: the end device, the gateway, the network server, and the application server. In this chapter each of the components will be implemented and configured to assemble a distributed NOx and CO measurement system. The end device is a STM Nucleo with an Arduino LoRa shield, equipped with a MiCS-4514 Gas Sensor measuring NOx and the OpenAir Node V1.1 measuring NOx, CO, temperature, and humidity. The gateway is a IMST iC880a LoRa board connected to a Raspberry Pi. This is one of the most popular versions of a gateway. The network server and the application server are both part of The Things Network. However, there is an application receiving the data from the application server and forwarding them to the SensorCloud. Figure 26 shows the sequence diagram of an uplink message. The notes above the objects indicate which part they represent in the LoRaWAN network architecture. Downlink messages are not intended in this system.

In the first step, the ESP32 or Nucleo poll the Sensor for a value. After they received the values, the message is broadcasted to surrounding gateways, which sends them over ethernet



Figure 26: Sequence diagram of an uplink

## 3.1 End Device

End devices are, simply put, the devices that are deployed in the open landscape. An end device may be a small actor for opening or closing the roller shutter on a window, reading the status of the garbage container in a backyard or the air quality in a neighborhood. In this thesis the NO<sub>x</sub> value of the air is measured.

Next chapters present a few common end devices and take a more detailed look at the two end devices that were implemented for this thesis.

### 3.1.1 Market Research Devices/End Devices

There are already some companies on the market offering different LoRaWAN development boards, already equipped with a microcontroller and a LoRaWAN chip on board. Also, it is possible to purchase these two separately and connect them by oneself. All the presented end devices have either a Semtech sx1272 or the identical in construction Hoperf RFM95W chip on board.

#### **The Things Uno**

The development for The Things Uno was, like the development for The Things Gateway, sponsored by a Kickstarter campaign during late 2015 to early 2016. The Things Uno is based on the Arduino Leonardo and has an added LoRaWAN module build in. It is compatible to the Arduino IDE and common Arduino shields. [23]

In August 2017, the first batch of Things Unos was shipped to Kickstarter supporters but was not yet available for purchase at the beginning of writing on this thesis.

Estimate of costs: 40€

#### **LoPy**

The LoPy is a small MicroPython enabled development board with an ESP32 chip capable of LoRa, Wi-Fi, and Bluetooth connection. It has two UART and SPI interfaces, an I2C and an I2S interface, a 12 bit ADC with eight channels and 24 GPIO pins, 512 KB RAM and 4 MB external flash memory, and is supposed to use a fraction of power compared to other microcontrollers in question. [24]

Estimate of costs: 40€

#### **ESP32 with Hoperf RFM95W**

The ESP32 is a series of low cost and low power microcontrollers with integrated Wi-Fi capabilities made by Espressif Systems [25]. The ESP32 that is used in the OpenAir Node has three UART, four SPI, and two I2C interfaces, a 12 bit ADC with up to 18 channels and is capable of Bluetooth in addition to Wi-Fi communication. The Hoperf RFM95W is a low power, long range transceiver module featuring a LoRa

communication modem [17]. Together they can be built into an end device for LoRaWAN implementations.

The OpenAir Cologne community distributed 100 Wireless LAN NOx and CO sensors equipped with an ESP32 to interested citizens of the town. They have the necessary connectors for an Hoperf RFM95W on the back, but neither are they equipped with a LoRa module nor are they programmed to send data via LoRaWAN.

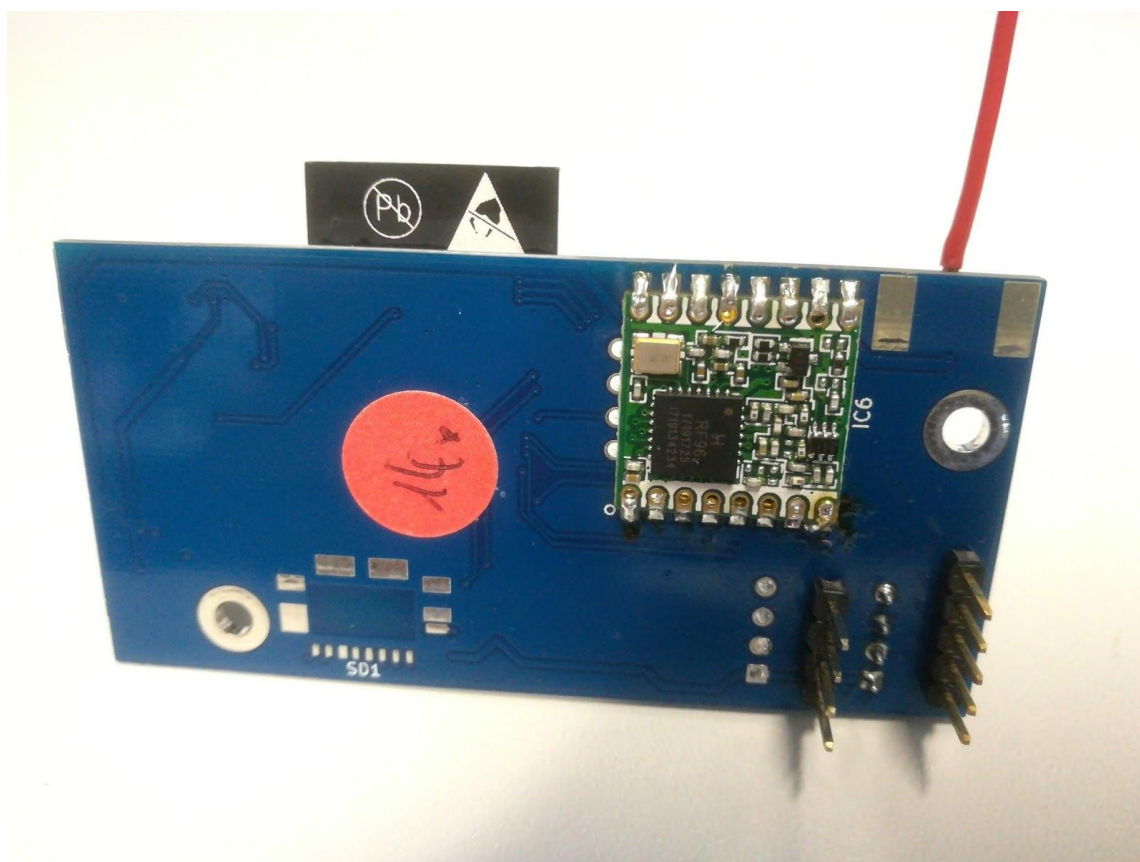


Figure 27: Hoperf RFM95W soldered on the backside of the OpenAir Node V1.1

As part of this thesis, one OpenAir Node was equipped with an RFM95W as shown in Figure 27, and reprogrammed to send data over TTN.

Estimate of costs: 25€

### **STM32 Nucleo-L073RZ with SX1272MB2DAS LoRa expansion board**

The STM32 Nucleo-L073RZ is a development board for STM32 microcontrollers. This board comes with a detachable USB interface for programming. It has a 12 channel ADC with 12 bit accuracy, a SPI and a I2C bus connection, and Arduino compatible connectors. The SX1272MB2DAS is a LoRa expansion board for Arduino. STM offers drivers and middlewares for LoRa and LoRaWAN to use with these two boards. [26]



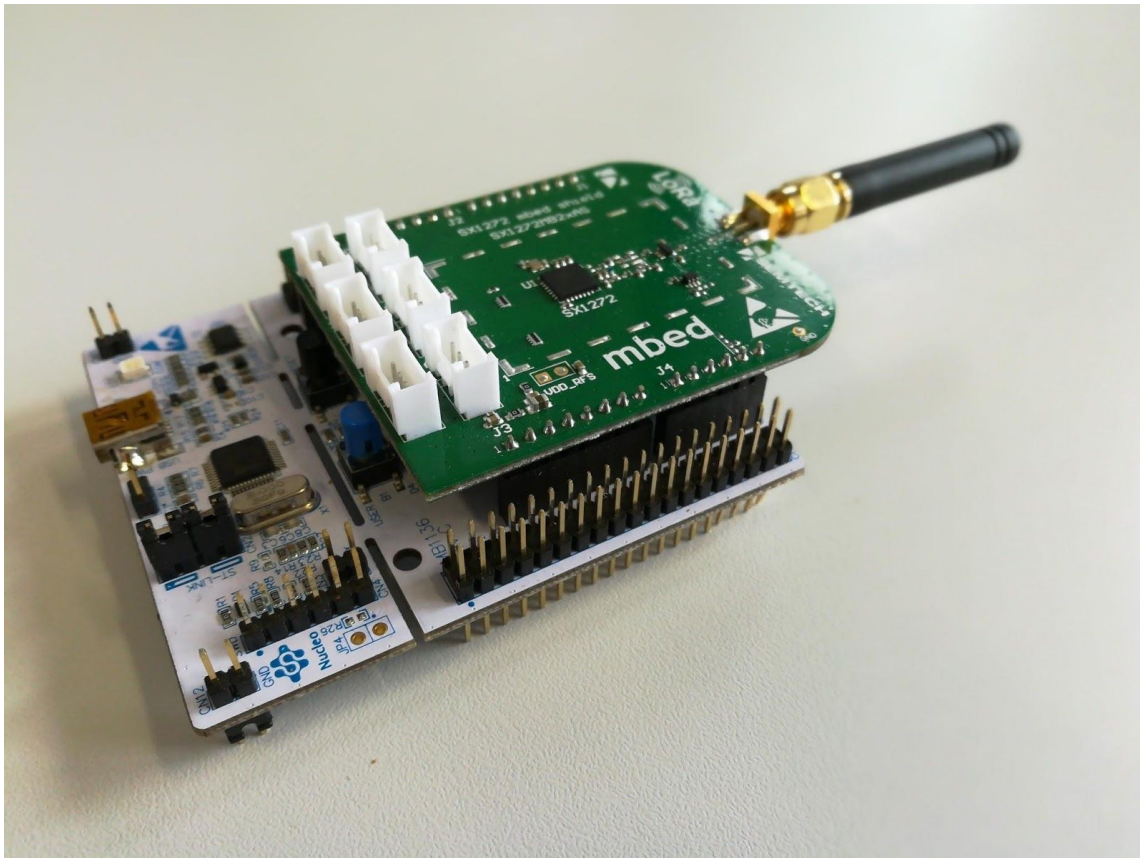


Figure 28: STM32 Nucleo-L073RZ with a SX1272MB2DAS shield

Estimate of costs: 65€

### 3.1.2 Measuring NO<sub>x</sub> And CO Data

Both end devices, the STM Nucleo and the ESP32 on the OpenAir Node, are using a MiCS-4514 Gas Sensor to measure NO<sub>x</sub> and CO values. The Gas Sensor is equipped with two sensor chips with independent heaters and sensing layers. One of those sensors chips detects oxidizing gases, which NO<sub>x</sub> is a part of, the other one detects reducing gases, which CO is a part of. When the heaters warm up the sensing layer, the layer absorbs the elements in the air which causes a change in its electrical resistance. This change then is measured by the end devices. [27]

The downside of the MiCS-4514 sensors is that each sensor has a different specific resistance in air and the dependent relationship between resistance and gas concentration also differs [28]. Therefore, the absolute gas concentration values can't be derived from these sensors directly. Thus, the measured value is nothing more than the resistance of the sensing layers. The sensors are also prone to temperature drifts. SGX Sensortech states that for its CO sensor, the resistance is about halved by a temperature increase of 25°C. The measurements however, can be used to potentially derive qualitative data of the change of air quality. But this is not in the scope of this thesis.

## Measuring On The STM Nucleo-L073RZ

The STM Nucleo-L073RZ has a 12 bit analog-to-digital converter (ADC) build in. The implemented sensor performs analog-to-digital conversion in polling mode. The microcontroller queries the ADC in continuously for a value and then waits until the ADC has converted that value.

Although there is a MiCS-4514 gas sensor connected to the Nucleo, the end device only reads and transmits NOx data. Approximately two minutes after a message was sent, the SX1272 LoRa transceiver chip on the shield scans if any radio bands are available. In positive case, the Nucleo polls the ADC for conversion of the MiCS resistance value. This value then is transmitted to the nearest gateways.

For two reasons, the end device does not send any converted data. First is to keep the LoRa packets as small as possible. The FRMPayload only needs 3 bytes to send two 12 bit values for NOx and prospectively CO resulting in a total packet size of 16 bytes. Sending the resistance values, will take considerably more space than 12 bits. Second, the value read from the sensor is not an absolute NOx value, but the resistance of the sensing layers. Plotted in a graph it shows the relative change of NOx density in the air but no absolute and comparable values. If later the sensors are calibrated, it can be done without changing the program code of the end device.

## Measuring On The OpenAir Node V1.1

The OpenAir Node is capable of measuring more values than the Nucleo. Additionally, to the MiCS-4514 Gas Sensor on the Nucleo, it has a temperature and humidity sensor integrated. The resistance of the MiCS are read with the ADC on the ESP32, while the temperature and humidity sensor is connected via I2C. The schematics for this board are available online as open source [29].

The ESP32 was programmed with the Arduino programming language, which is a dialect of C++. It can be programmed with the Arduino IDE using the help of the *Arduino core for ESP32 Wi-Fi chip*. It needs the `Wire.h` library for I2C and the `SPI.h` library for SPI communication. The `lmic.h` library and the `hal/hal.h` library are added for the LoRaWAN communication.

Every fifteen minutes the ESP32 polls the temperature and humidity sensor by sending a measurement command over I2C. They have to be read separately because the measurement commands differ. The ESP32 receives a two byte array for each reading. The schematic shows, that the SI7006-A20 temperature and humidity sensors are connected to the ESP32 via I2C. The master I2C pins on the ESP32 are GPIO 18 for the data signal and GPIO 19 for the clock signal. The measurement command has to be transmitted to the SI7006-A20 before the data can be requested from the I2C. The following code snippet is an example of how the reading works.

```

1      #define Addr 0x40                      //Addr: SI7006-A20 address
2      Wire.beginTransaction(Addr);
3      Wire.write(0xF3);
4      Wire.endTransmission();
5      delay(500);
6      Wire.requestFrom(Addr, 2);
7      if(Wire.available){
8          tempData[0] = Wire.read();
9          tempData[4] = Wire.read();
10     }

```

In line 1 to 3, the ESP32 opens the transmission and sends the temperature measurement command 0xF3. After a delay of 500 ms, the ESP32 requests two bytes from the sensor. If the request is successful, these two bytes are stored in the temperature data variable `tempData`. The four least significant bits are later cut off before transmitting them. This allows to store the temperature and humidity values inside an array with a length of three bytes instead of four and thus reducing one byte of payload. The loss of accuracy only affects the result by less than 1% which can be neglected. The precision of the temperature and humidity sensors are, in contrast,  $\pm 5\%$  relative humidity and  $\pm 1^\circ\text{C}$  in temperature, which is far higher than this loss of accuracy.

The value for NO<sub>x</sub> is also read every fifteen minutes, at the same time as the temperature and humidity. To reduce energy consumptions, the heating plate of the gas sensor are off 97% of the time. They need 30 seconds to heat up before the reading.

The value for CO is polled by the ADC every five seconds and added into one variable. When the other values are read, the value of the CO variable is divided by the number of added values getting an average value.

As with the Nucleo, the OpenAir Node does not transmit calculated values, but only the raw bits from the sensors. This enables the OpenAir Node to fit four sensor values into a six byte array, resulting in a LoRaWAN packet of 19 bytes in size. The values are transmitted approximately every fifteen minutes in accordance to the TTN fair use policy. This way the spreading factor can go up to SF10. The transmit function calls the reading function before transmitting the data. This may not be exactly every fifteen minutes, because sometimes the end device must wait for free radio bands.

The SPI connection to the RFM95W requires a little change in the `hal/hal.h` library. The pinout is configured in the `lmic_pinmap`. Following is the declaration of the `lmic_pins` struct, which is used by the library:

```

1      const lmic_pinmap lmic_pins = {
2          .nss = 33,
3          .rxtx = LMIC_UNUSED_PIN,
4          .rst = 32,

```

```

5         .dio = {21, 22, 23}, //use 1 pin for all 3 radio dio pins
6     };

```

The SPI bus is not connected to the standard SPI pins of the ESP32 but have to be matrixed to other pins. This is supported by the ESP32 but not natively by the `hal/hal.h` library. To initialize the SPI connection, inside the `hal.cpp` file the `SPI.init()` inside of the `hal_spi_init()` function must be modified to contain the right SPI SCK, MISO, MOSI, and NSS pins. After this, the `hal_spi_init()` looks as follows:

```

1     static void hal_spi_init(){
2         SPI.begin(25, 15, 26, 33);
3     }

```

Figure 29 shows the state machine of the end device.

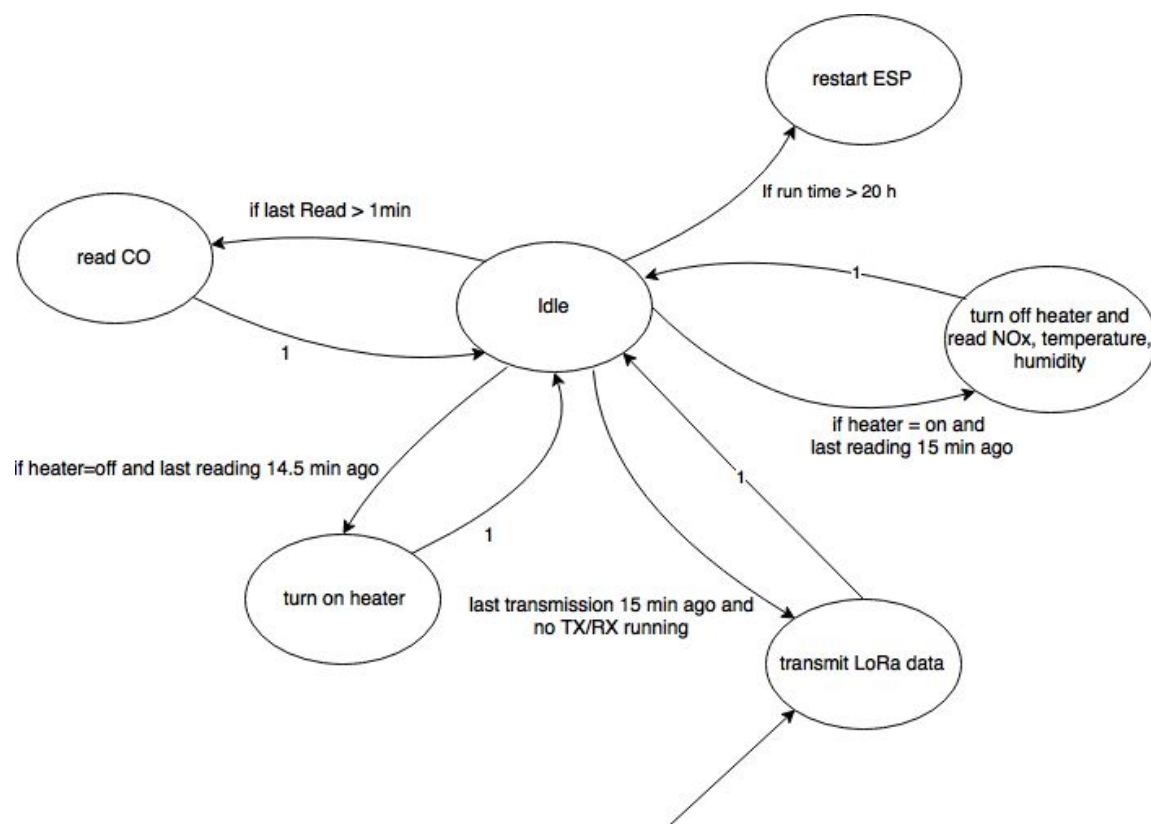


Figure 29: State machine end device

The end device must transmit a first LoRaWAN message on startup. This is the OOTA activation packet with which the end devices receives its address and keys.

The frequency of LoRaWAN messages can be configured in the `TX_INTERVAL` constant. This interval is to be specified in seconds and gives the shortest time between two messages. Since duty cycle limitations might increase the interval length, as seen in [chapter 2.7](#), it is dependent on the spreading factor, which in turn is dependent on the quality of the transmission. However, with an selected interval of fifteen minutes, duty

cycle limitations are of no concern. To set the interval to 15 minutes, the constant has to be declared as follows:

```
1    const unsigned TX_INTERVAL = 15 * 60;
```

The `do_send` method is a callback method that is registered by the `lmic.h` library to be called after the `TX_INTERVAL` time.

```
1    void do_send(osjob_t* j){
2        if (LMIC.opmode & OP_TXRXPEND) {
3            Serial.println(F("OP_TXRXPEND, not sending"));
4        } else {
5            LMIC_setTxData2(1, txData, sizeof(txData), 0);
6            Serial.println(F("Packet queued"));
7        }
8    }
```

This method checks for other TX or RX jobs already running on the device. If not, the uplink data is set in line 5. In the example, the payload is `txData` and is send to port 1.

### 3.1.3 Registering And Connecting An End Device To The Things Network

To be able to connect an end device to a LoRaWAN network, it needs to have a device EUI and an AppKey if the device is activated over OTAA, or a fixed device address, an AppSKey and a NwkSKey if the end device is activated over ABP as described in [chapter 2.2](#). Since every end device needs an AppKey or an AppSKey, the application first must be registered within TTN as described in [chapter 3.4.1](#), so that end devices can be registered within that application.

After an application is registered, a click on the application name in the application tab of the TTN console as shown in Figure 30, the application overview is presented. A click on button “register device” takes the user to the register device screen.

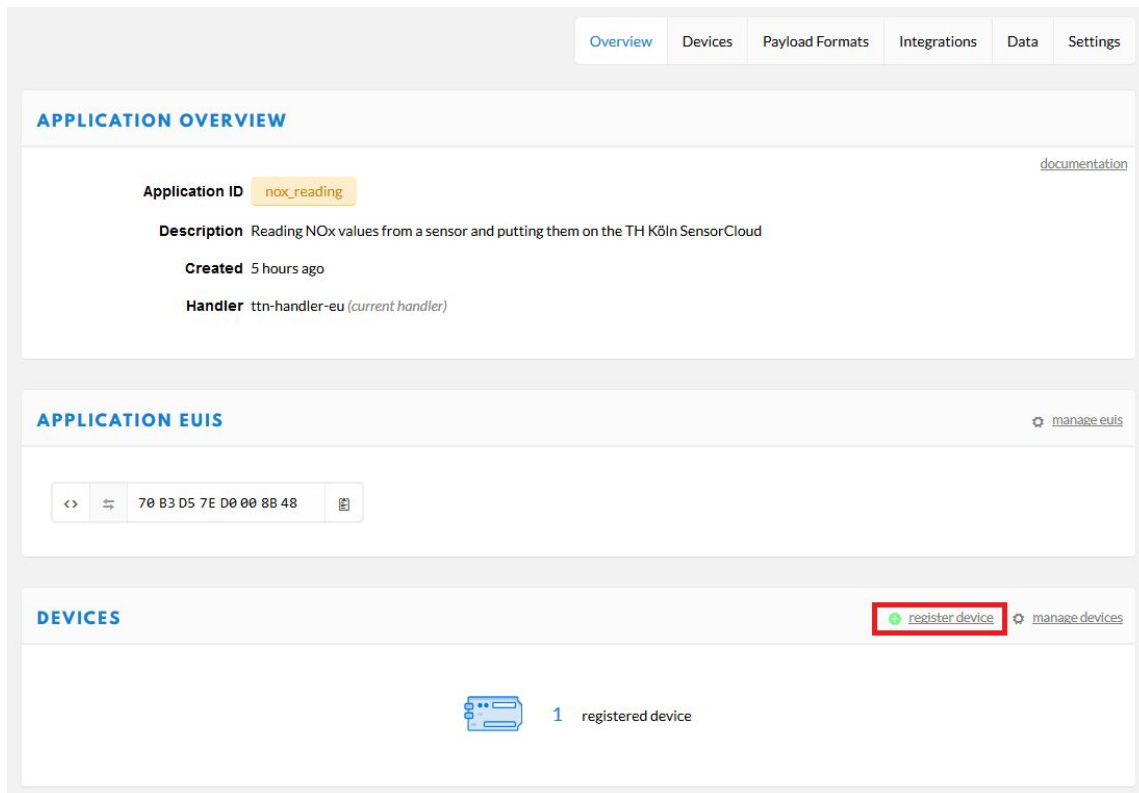


Figure 30: Step one: Click on register device

Figure 31 shows the register device form. The device ID is the identifier for the device. This is user defined and must be unique within TTN. The device EUI is generated by the TTN and is used for identification within the network. Also, it is used as the topic for the MQTT messages the handler publishes as seen in [chapter 3.4.3](#). The app key is the same AppKey that is needed during the joining process, if OTAA is used, and will be generated by TTN. The app EUI is the EUI of the app the end device will be registered to. In this example, the end device is registered to the `nox_reading` application so the app EUI is the same as shown in Figure 38 in [chapter 3.4.1](#).

**REGISTER DEVICE** [bulk import devices](#)

**Device ID**  
This is the unique identifier for the device in this app. The device ID will be immutable.

**Device EUI**  
The device EUI is the unique identifier for this device on the network. You can change the EUI later.

**App Key**  
The App Key will be used to secure the communication between you device and the network.

**App EUI**

Figure 31: Step two: fill out form and submit

## 3.2 Gateway

The gateway is the device which forwards LoRaWAN messages to the Internet. It is able to receive and send LoRa packets and is connected to a network server.

This chapter briefly looks at a few selected gateways, and exhibits how the gateway on the University of Applied Science Cologne is build, configured, and how a gateway is registered to The Things Network. Also aspects of security and protection against environmental influences are disclosed.

### 3.2.1 Market Research Gateways

The TTN community recommends five different gateways. They have set up instructions online for each one, as well as configuration and troubleshooting tips. Four gateways are completely assembled, while the fifth is a LoRa connectivity board, which can be built to act as a gateway with the help of a Raspberry Pi (or a similar embedded Linux computer).

#### The Things Gateway

The Things Gateway was exclusively developed for The Things Network. It was funded by a Kickstarter campaign in late 2015 to early 2016. The software and hardware are open-source. It connects with the Internet via Wi-Fi or Ethernet connection. The setup is web based and supposed to be very easy. It supports about 10.000 nodes. The first batch was to be shipped by the end of 2016, but had a few delays. [30]

As of November 2017, the first The Things Gateways were sent to Kickstarter backers, but it is not available at retail yet.

Estimate of costs: 300€

### **MultiTech Conduit**

The MultiTech Conduit is a cellular communication gateway for industrial IoT applications. It has two proprietary 'mCard' slots to expand the module with wired or wireless interfaces, for example a LoRaWAN module, which can support thousands of nodes. MultiTech is a contributor of the LoRa Alliance. [31]

Estimate of costs: 440€ (MultiTech Conduit) + 160€ (LoRa mCard)

### **Kerlink LoRa IoT Station**

The Kerlink LoRa IoT Station is an industrial LoRa Gateway intended for outdoor use. It can be linked with The Things Network, but is supposedly difficult to set up and maintain. [32]

Estimate of costs: ~1500€ (1200€ plus tax and tariff)

### **Lorrier LR2**

The Lorrier LR2 is a gateway for setting up professional IoT networks based on the LoRaWAN protocol. It is an outdoor device with an IP66 weatherproof casing, intended for establishing wide coverage networks. [33]

Estimate of costs: 755€

### **IMST iC880A-LoRaWAN Concentrator**

The IMST iC880A-LoRaWAN Concentrator is a LoRa connectivity board which is used with an embedded Linux Board, like a Raspberry Pi or a Beaglebone. There are a lot of instructions, tips and user stories to be found on The Things Network web page. It can communicate with "a huge amount of LoRa end-nodes" up to 15 km in line of sight [15]. Figure 32 shows the IMST iC880a board out of the box with pigtail and antenna attached.



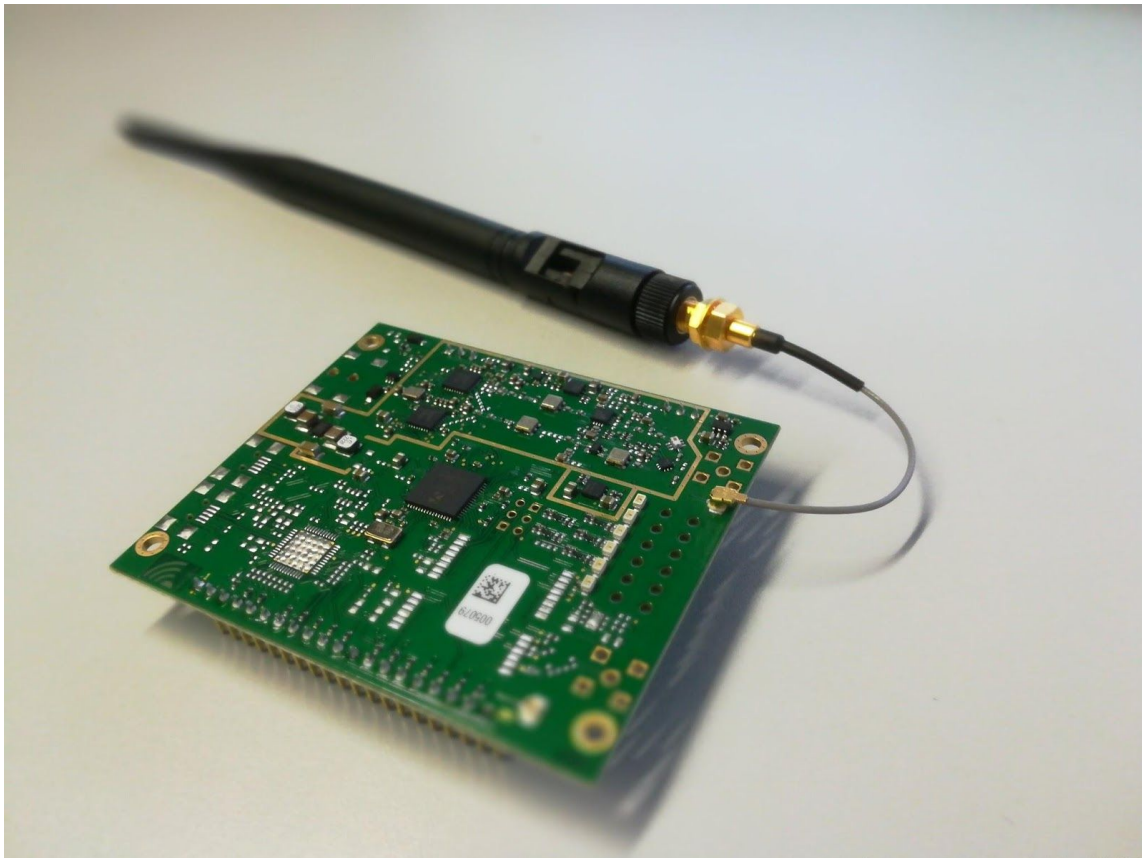


Figure 32: IMST iC880A-LoRaWAN Concentrator with pigtail and antenna

This board, in tandem with a Raspberry Pi 3, is used as the example gateway for this thesis.

Estimate of costs: 155€

### 3.2.2 Building The Gateway

Besides the IMST iC880a LoRa board with pigtail and antenna and the Raspberry Pi, a connector board or jumper cables are needed to connect all. As pictured in Figure 33, a connector board was built for the connections, as well as to secure the two boards inside a weatherproof case.

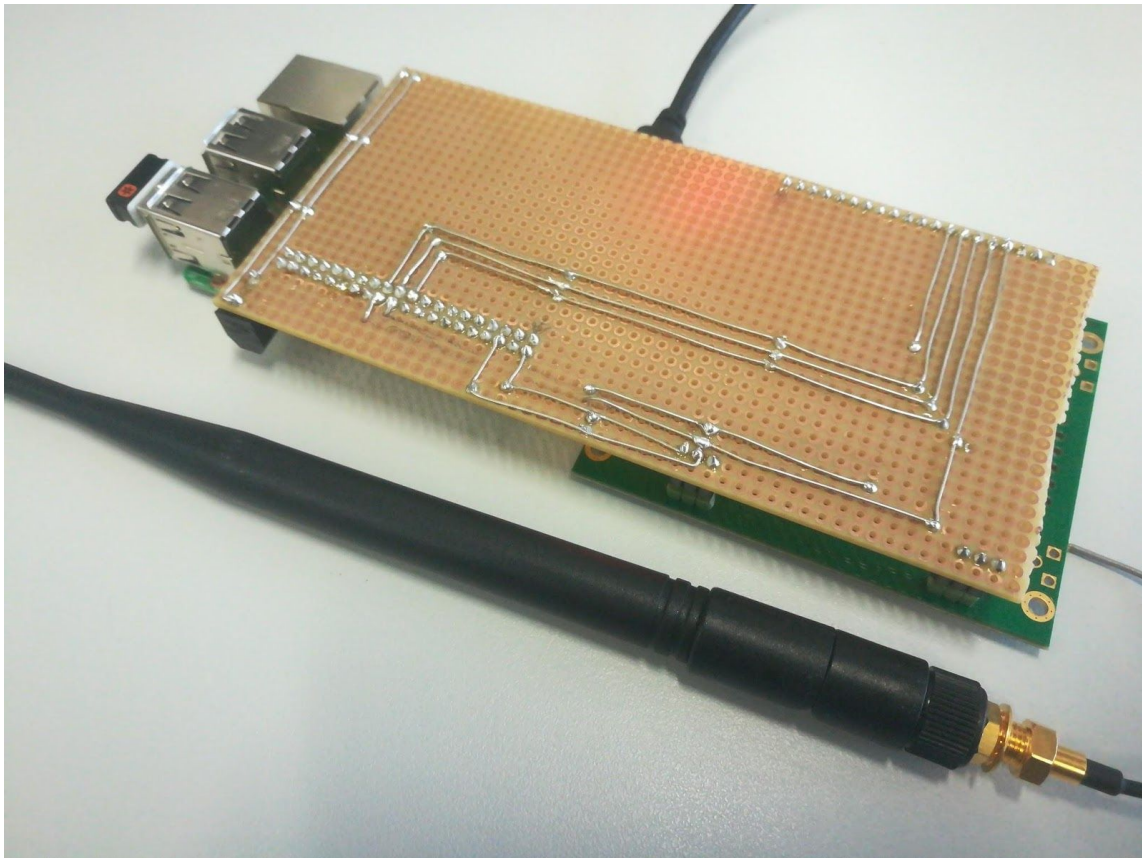


Figure 33: Raspberry Pi to IMST iC880A connector board

The cabling between the boards has to be as in table 2. They are connected over SPI.

Raspberry Pi pin	iC880a pin	description
2	21	Supply 5 V
6	22	GND
19	16	MOSI
21	15	MISO
22	13	Reset
23	14	SPI CLK
24	17	NSS

Table 2: Pin connection Raspberry Pi to iC880a

The connector board additionally has a ground, a UART TX, and a UART RX pin laid out. In that way the Raspberry Pi can be accessed via a serial link if it is not connected to the internet.

### 3.2.3 Configuration Of The Gateway

After the physical assembly, the gateway software must be loaded onto the Raspberry Pi. This gateway uses Raspbian Jessie, a Debian Linux distribution specifically made for the Raspberry Pi. The gateway software is open source and available online [34].

For this gateway, a new Linux user with the name `ttn` was created and added to the Linux sudoers list. As the software is on GitHub, it can just be cloned and installed onto the Raspberry Pi. When the installation is finished, the gateway is already up and running. The best way to check the gateway status is to register it to TTN over the TTN console, as described in [chapter 3.2.4](#).

### 3.2.4 Registering A Gateway To The Things Network

After installing the gateway software correctly, it is already connected to The Things Network. To be able to configure it over the TTN console and to make it appear on the network coverage map it must be registered. This can be done in the TTN console. The console can be accessed on the TTN homepage in the top right corner like in Figure 34.

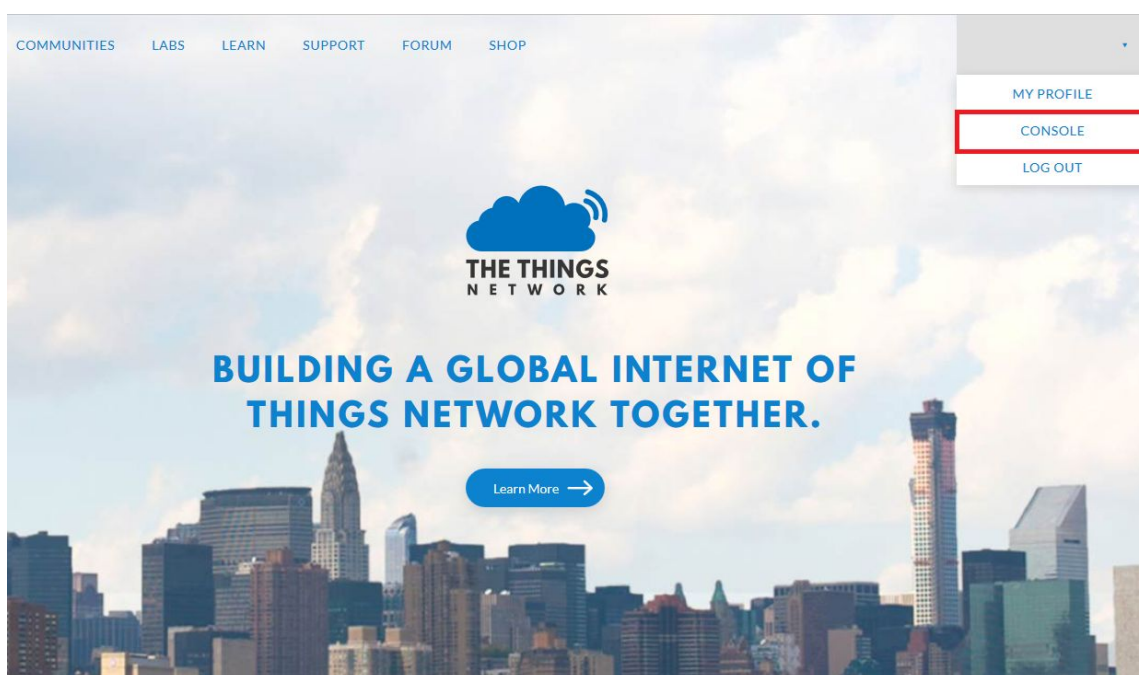


Figure 34: Step one, open the TTN console

The TTN console is a kind of a central hub, where users can create new applications or register new end devices or gateways, as well as view and change settings on applications, end devices, and gateways. The GATEWAYS tab, as in Figure 35, lets the user change settings on existing gateways or register new ones.

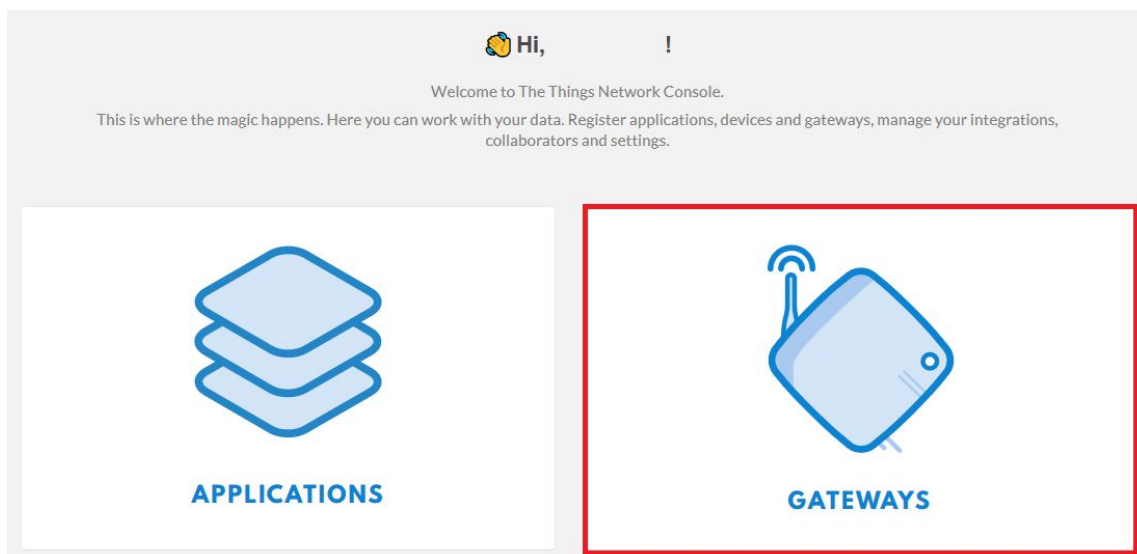


Figure 35: Step two, select gateways

The gateways tab shows an overview of all gateways registered for the currently logged in account. Left column shows the gateway ID, which identifies the gateway, followed by a nomination and the connection status. On the far right the frequency plan is shown. For this gateway it is the 868 MHz ISM band. New gateways can be registered in the 'register gateway' tab as in Figure 36.

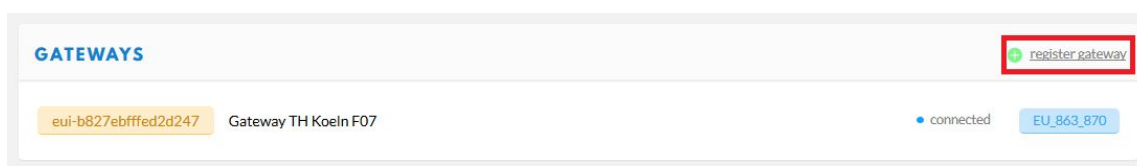


Figure 36: Step three, select register gateway

Figure 37 shows the REGISTER GATEWAY form. The first input in the form is the gateway ID which identifies the gateway in The Things Network. While in theory it may be any string larger than two characters, here it must be unique within TTN. If the gateway uses the legacy Semtech packet forwarder, as is the case with the Raspberry Pi and iC880a LoRa board, a gateway ID will be provided by the software during the installation process.

The description may be any string that helps the user to distinguish their gateways. The description will also be displayed on the gateway overview on the frontpage of TTN.

The next input asks for the used frequency plan. In Germany this is the 868 MHz ISM band, which is a European license free band.

Thereafter the user has to specify which router the gateway will be connected to. The router, as described in [chapter 2.8](#), manages gateway functions and schedules transmissions. For minimal latency it is best to choose the router that is physically the closest to the gateway, which will be the 'ttn-router-eu' for German gateways, since it is the only european router as of now.



On the bottom, the form asks for the placement of the gateway. The coordinates have to be given, so that it can be displayed correctly on the gateway overview.

Last question is whether the antenna is placed indoor or outdoor.

**REGISTER GATEWAY**

**Gateway ID**  
A unique, human-readable Identifier for your gateway. It can be anything so be creative!


☐ **I'm using the legacy packet forwarder**  
Select this if you are using the legacy [Semtech packet forwarder](#).

**Description**  
A human-readable description of the gateway

**Frequency Plan**  
The [frequency plan](#) this gateway will use

**Router**  
The router this gateway will connect to. To reduce latency, pick a router that is in a region which is close to the location of the router itself.

**Location**  
The exact location of your gateway. This will be used if your gateway cannot determine its location by itself. Set a location by clicking on the map.



**Antenna Placement**  
The placement of the gateway antenna

☐ Indoor ☐ outdoor

Figure 37: Step four: fill out form and submit

After the gateway is registered it appears in the GATEWAYS overview tab. From there all settings, with exception of the gateway IDs, may be changed belatedly.

### 3.2.5 Security

As the basis for the gateway is a Raspberry Pi, which is a Linux computer and thus prone to attacks from the internet. The gateway is connected to the university network

and has a static IP address. It is possible to connect to the gateway over a SSH connection protected with a secure password.

### 3.2.6 Protection Against Environmental Influences

For increased range of LoRa connections, the gateway is installed outdoors. This exposes it to environmental influences such as rain, snow, dust, wind, temperature, and lightning.

The gateway is enclosed in an acrylnitril-butadien-styrol (ABS) casing with an IP65 protection rating. This means the contents of that casing is completely protected against solid particles and water jets. The gateway needs two cable connections, one for power, the other one for internet. These cables are laid into the casing through bulkhead cable glands with an IP68 protection rating. This is more than sufficient against rain, snow, and dust.

The average temperature in Germany ranges from about  $-5^{\circ}\text{C}$  to  $+20^{\circ}\text{C}$ . This is good within the operational temperature of the Raspberry Pi and the iC880A [35], [15]. A bigger problem caused by the high temperature variation is the rising pressure inside the encasing which will lead to condensation inside the casing. To combat possible water damage a pressure compensation vent is installed at the bottom of the casing which lets humid air out of the casing while preventing water from the outside going in. This is most effective against condensation.

## 3.3 Network

The network used for this project is The Things Network, a free, open, and community driven LoRaWAN network.

### 3.3.1 Configuring The Things Network

After an application and an end device is registered in The Things Network, as described in [chapter 3.1.4 or 3.4.1](#), TTN generates all the EUI and keys necessary for end devices and applications to connect to the network. At this point the network is ready for use. A few configurations however can be made.

The Things Network has a few applications that can be integrated directly into the network. For example, a MySQL database, which stores the data for up to seven days. Also other services to visualize data or prototype applications with helpful features. For this thesis, none of them is used.

The second and most important configuration to make is the implementation of payload applications. Without these, all that TTN does is forward the information from the end

device to the application. Payload functions allow to manipulate the data in the network before it is forwarded.

### 3.3.2 Payload Formats

Applications and LoRa end devices use data differently. While applications can use Java Maps or JSON objects for enhanced readability, the LoRa end devices work mainly with byte arrays due to the method of transportation data to and from the cloud. With a standard DSL internet connection, it is easy to send and receive bigger, nicely formatted data. LoRas low bandwidth on the other hand makes it problematic to send big packets and even impossible to send packets bigger than 51 bytes [10]. The Things Network offers the possibility to implement four different payload functions that run on their network server to decode, convert, and validate messages and to encode messages.

The decoder decodes the messages coming from an end device and going in the application from a byte array to a JSON object, while the encoder encodes the messages going to the end device from a JSON object to a byte array, which the end device is able to understand. The converter function changes the JSON object into a data type for the application. Then the validator function checks, if the values are within an expected range and have the right type.

The functions are all written in JavaScript and can directly be edited in Payload Formats tab of the desired application in the TTN console under the application tab (see [chapter 3.4.1](#)). The functions are basically callback functions written by the user, called one after another after a message arrives in the TTN cloud.

In the following example the data of a two byte array is decoded into a JSON object, followed by the data types conversion, and the result validation

```

1      function Decoder(bytes, port) {
2          var decoded = {};
3
4          if(port === 3){
5              decoded.noxData = ((bytes[0] & 0xf0) << 4) + bytes[4];
6              decoded.coData = ((bytes[0] & 0x0f) << 8) + bytes[2];
7          }
8          return decoded;
9      }

```

The arguments of the decoder function in line 1 are the bytes of the raw payload field and the port address, to which the end device has send the packet. The raw data is decoded in line 5 and 6 into a JSON object which is returned and used by the converter function. End devices are allowed to send different kinds of packets. In this case the NOx sensor sends the NOx and CO data to port 3, so it could, for example, send its

temperature and humidity data to port 4 and the decoder function would handle it differently.

```

1 function Converter(decoded, port) {
2     var converted = decoded;
3
4     if (port === 3){
5         var boardVoltage    = 3.3;
6         var adcResolution   = 1024;
7         var r5               = 3900;
8         var r7               = 360000;
9
10        var voutNox = (converted.noxData / adcResolution) * boardVoltage;
11        converted.voutNox = voutNox;
12        var rNox = ((boardVoltage - voutNox) * r5)/voutNox;
13        converted.rNox = rNox;
14        var voutCo = (converted.coData / adcResolution) * boardVoltage;
15        converted.voutCo = voutCo;
16        var rCo = ((boardVoltage - voutCo) * r7)/voutCo;
17        converted.rCo = rCo;
18    }
19    return converted;
20 }

```

The converter function also has two arguments, the JSON object, returned by the decoder function and the port. The purpose of this function is to convert the data, send by the end device into a format the application is able to handle.

## 3.4 Application

An application in the LoRaWAN environment is a program on an arbitrary device interacting with the internet and collecting data from the end devices via the network. This may be a visual flow using Node-RED running in the cloud, a Python script running on a Raspberry Pi in a private environment, or a C program running on a weather station at a private home.

This chapter examines how an application is registered within The Things Network. Further, it is explained how the application for this thesis communicates with The Things Network and the SensorCloud.

### 3.4.1 Registering An Application Using The Things Network

Before sending and receiving data over TTN, an application must be created in the TTN environment. The settings of all TTN applications and end devices are done, like the



gateway settings in chapter 3.2.4, in the 'TTN console', but as marked in Figure 38 under the applications tab.

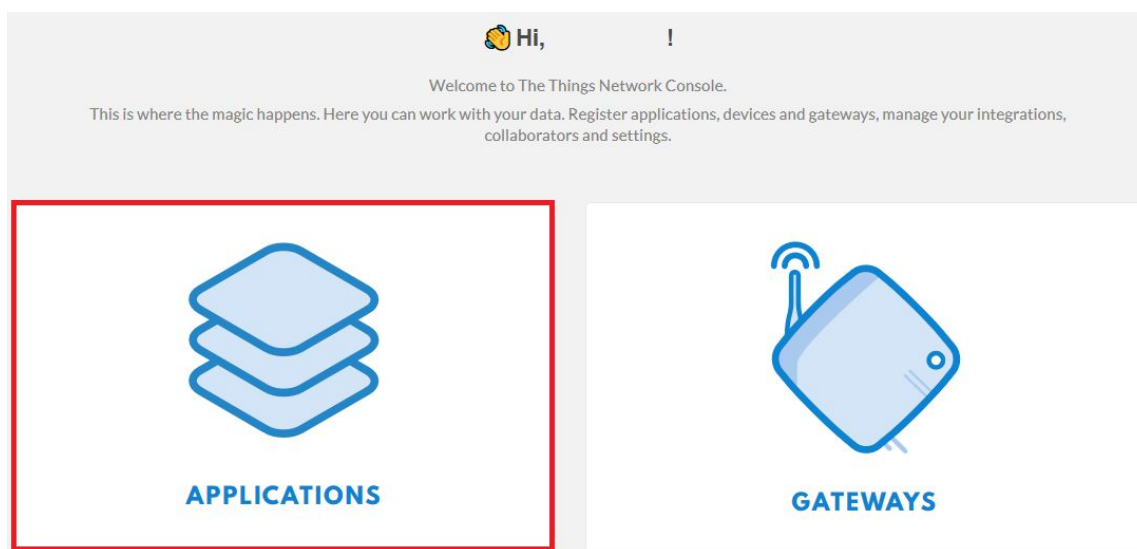


Figure 38: Step two: select applications'

After an application is created, end devices can be registered and attributed to that application.

Figure 39 shows the APPLICATIONS tab of TTN. Here the user can register new applications. It also shows a list of all applications already created. The list displays (from left to right) the Application ID, which uniquely identifies the application, followed by a description in human readable form, the handler identifier, and an 8 bytes long application EUI. As described in chapter 2.8, the handler is a network cluster responsible for message decryption, payload conversion, queueing and dispensing downlink messages, as well as hosting a MQTT broker. The naming convention for the TTN handlers is 'ttn-handler-'regional identifier'.

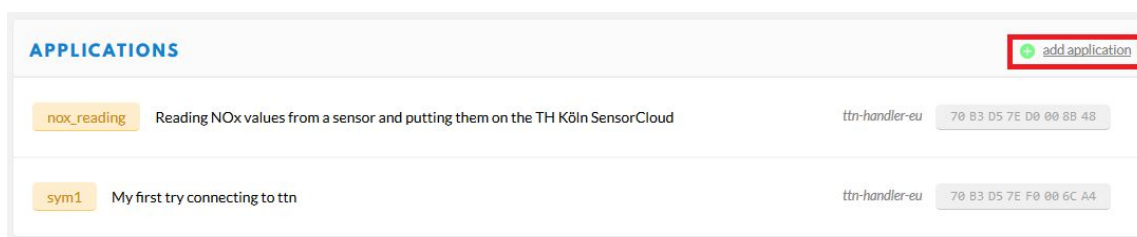


Figure 39: Step three: click on add application

The ADD APPLICATION screen, as in Figure 40, lets the user adjust some of the application settings. The Application ID is a network-wide unique ID. It is not a human readable form, but will be used by the applications, while it gets the data via the MQTT API. In the description field, a short text about the application may be entered optionally. The Application EUI is issued by TTN and is the identifier needed by the end devices to

connect to the network and the application itself. Also it is indispensable for, the applications to download the upload messages via the MQTT broker.

Afterwards, the application then can be registered to a handler. As of January 2018, TTN operates four handlers (ttn-handler-asia-se, ttn-handler-brazil, ttn-handler-eu, ttn-handler-us-west) on four different continents.

**ADD APPLICATION**

**Application ID**  
The unique identifier of your application on the network

**Description**  
A human readable description of your new app

Eg. My sensor network application

**Application EUI**  
An application EUI will be issued for The Things Network block for convenience, you can add your own in the application settings page.

EUI issued by The Things Network

**Handler registration**  
Select the handler you want to register this application to

ttn-handler-eu

Cancel Add application

Figure 40: Step four: fill out and submit form

After the application is added to The Things Network, the Application EUI and access keys are generated and can be used to extract data from the broker for the application.

### 3.4.2 Communication To And From The End Device

To receive downlink messages from the end device or send uplink messages to the end device via The Things Network, The Things Network offers a Data API via MQTT. An AMQP Data API is in development, but not yet available as of date of this thesis. Furthermore, The Things Network offers SDKs for the Go and Java programming languages as well as the Node.js and Node-RED platform. These SDKs provide a layer of abstraction for an easier interaction with the APIs.

### 3.4.3 The Things Network MQTT API

MQTT (Message Queue Telemetry Transport) is a machine to machine messaging protocol utilizing the publish-subscribe message pattern. Consisting of multiple clients that are publishing messages to certain topics and a broker that dispatches the messages to those clients having subscribed to these topics.

The Things Network has a MQTT broker running on its servers. Messages published by the MQTT client on the TTN handler are distributed to the MQTT clients on the applications that are subscribed to these end devices. The server is accessible at '*{region}.thethings.network*', in which *{region}* is the handler for the application minus the 'ttn-handler-' part (e.g. if the handler is ttn-handler-eu, the MQTT Broker will be hosted at eu.thethings.network). The username is the application ID and the password is the application access key. This information is available in the TTN console under the application tab. Upon a message arriving from an end device, the TTN MQTT client publishes a message with the uplink packet sent, as well as information about the sending device and the used gateways, like the ID or position, and metadata about the packet, like timestamp or coding rate. Subjects of the MQTT messages will be '*{application ID}/devices/{device ID}/up*', where *{application ID}* and *{device ID}* are the ID of the application, to which the end device belongs, and the ID of that end device respectively. The MQTT message will be a JSON string which then can be converted and processed further. The message received by the client will look as follows:

```

1 {
2   "app_id": "sym1",
3   "counter": 1,
4   "dev_id": "sym1-2",
5   "hardware_serial": "000C97C70C6FEEC4",
6   "metadata": {
7     "coding_rate": "4/5",
8     "data_rate": "SF12BW125",
9     "frequency": 868.3,
10    "gateways": [
11      {
12        "channel": 1,
13        "gtw_id": "eui-00000000000001dee",
14        "latitude": 50.94862,
15        "longitude": 6.94265,
16        "rf_chain": 1,
17        "rssi": -109,
18        "snr": -10.5,
19        "time": "",
20        "timestamp": 950630412
21      },
22      {
23        "altitude": 40,
24        "channel": 1,
25        "gtw_id": "eui-b827ebfffe2f1313",
26        "latitude": 50.934147,
27        "location_source": "registry",
28        "longitude": 6.988626,
29        "rf_chain": 1,
30        "rssi": -91,

```

```

31             "snr": 7.2,
32             "time": "2017-11-21T08:57:25.454418Z",
33             "timestamp": 448653260
34         }
35     ],
36     "latitude": 50.93395,
37     "location_source": "registry",
38     "longitude": 6.987992,
39     "modulation": "LORA",
40     "time": "2017-11-21T08:57:25.360320439Z"
41 },
42 "payload_fields": {
43     "coData": 94,
44     "noxData": 94,
45     "ppmNox": 6.420886214166897,
46     "rNox": 38585.10638297872,
47     "ratioNox": 42.87234042553192,
48     "voutNox": 0.3029296875
49 },
50 "payload_raw": "AF4AXg==",
51 "port": 3
52 }

```

Line 2 and 4 show the application ID and the device ID respectively. Line 8 contains the spreading factor, which in this case is 12, and the bandwidth, which is 125kHz. Line 11 to 34 are the gateways that were able to receive this message. This packet was picked up by the two gateways with ID `eui-00000000000001dee` and `eui-b827ebfffe2f1313`. The payload is printed in line 42 to 49. These are the values calculated in the TTN cloud in the payload formats tab as described in [chapter 3.3.2](#). Line 50 has the raw payload data, which is encoded in Base64.

### 3.4.4 Uploading Data To The SensorCloud

The SensorCloud has its own MQTT client. When a message with a parsable JSON object arrives at that client, it will store it to the database. The application for the MQTT client needs to do two things: decode the JSON object published by the TTN handler into a format the SensorCloud can understand, and then publish it to the SensorCloud broker.

The application *mqttService.py* is a small python script with two MQTT client instances running. The first one, named *ttnClient* (The Things Network Client), is connected to The Things Network, subscribed to `+/devices/+/up`, so to all uplink messages by all end devices of this application. The second instance, *dfdsClient* (DatenFuerDieStadt Client), is connected to the SensorCloud. This client does not subscribe to anything but publishes the sensor data to the SensorCloud.

When an uplink message arrives at the application it converts it to a JSON object the SensorCloud. An example message to the SensorCloud looks as follows:

```

1 {
2     "typ": "1",
3     "gw": "bcff9a23-47f6-489f-a1ba-b6c26723cc70",
4     "bn": "e6bfb32e-9e0f-4b20-8664-f9ce5229204b",
5     "bt": "1516106096000",
6     "e": [
7         {
8             "n": "ec9f89f8-cf1c-42fb-88e1-6f4e688af898:R1",
9             "t": "-3500",
10            "sv": "2902300.884955752"
11        },
12        {
13            "n": "2df28f5a-dc99-4721-a91f-fcbbe62dac52:R2",
14            "t": "-3505",
15            "sv": "31441.592920353978"
16        },
17        ...
18    ]
19 }
```

Line 2 is the type of the message, which will always be 1 for sensor data. Line 3 is the ID of the gateway. Gateway in the context of SensorCloud is not the LoRa gateway that received the message, but the gateway that published the MQTT message. Both will be the same for each message. Line 4 is the sensor ID. SensorCloud uses different IDs for sensors than TTN, so the application has a dictionary object for returning the SensorCloud ID when given the TTN ID. “bt” in Line 5 is a Unix timestamp in milliseconds. In the next line “e” is the sensor data object. This object is an array of multiple sensor values. Each of those values has its own UID, which is the value for “n”. This UID identifies the single data point. The second part of the value, after the colon, is the measurand. In the example above the JSON object contains two sensor values describing two resistance values: ‘R1’ and ‘R2’. “t” is the difference between the timestamp “bt” and the time the sensor value was captured and finally “sv” is the sensor value itself. After the JSON object is converted, the *dfdsClient* publishes it to the SensorCloud broker.

The application is able to handle data from end devices that read different types of sensors. For example, the Nucleo-L073RZ end device only sends NOx data, while the OpenAir Node can additionally send CO, temperature, and humidity data. Plus it has the potential ability to be connected to a particulates sensor, which provide data for pm10 and pm25 value. The application checks which values the end devices had send.

The script is written in python, because python is natively able to handle JSON objects. So the conversion from the JSON object the TTN MQTT client publishes to the format the SensorCloud MQTT client is able to handle, is relatively easy. The MQTT clients are implemented with the Eclipse Paho library, which implements lightweight and easy to use MQTT clients. The script can run on a Raspberry Pi alongside with the TTN gateway software. So there is no need for additional hardware.

## 4 Instructions For Use

This chapter briefly explains how to operate the implemented devices and software, and how to change them to use in other environments.

### 4.1 End Device

As soon as the OpenAir Node is powered up, it starts to measure and sends air quality data. At startup the green LED and the red LED will flicker very shortly. If the green LED stays lit, the `lmic.h` header has a problem initializing, most likely as the ESP32 has a problem communicating with the RFM95W. If the red LED stays lit for approximately one second, the end device did not receive a join-accept message during over-the-air activation. In operational state, the green LED will flicker every minute during CO reading. The red LED will stay lit for 30 seconds indicating the preheat phase of the MiCS-4514.

Before flashing the code to a new OpenAir Node, or any kind of ESP32 and RFM95W combination, a new device must be registered within The Things Network, as in [chapter 3.1.3](#), to get a new pair of DevEUI and AppKey. These have to be written inside the program code inside the `static const u1_t PROGMEM DEVEUI` array in little endian, so with the least significant first. By default, the TTN console displays the DevEUI in big endian, so the byte sequence must be reversed:

```
1 static const u1_t PROGMEM DEVEUI[8] = { 0xC1, 0x57, 0xD2, 0xE3, 0xCA,
    0xF1, 0x14, 0x00 };
```

The AppKey has to be written inside the `static const u1_t PROGMEM APPKEY` array in big endian, so it can be copied as is:

```
1 static const u1_t PROGMEM APPKEY[16] = { 0xE0, 0x3F, 0x4D, 0x39, 0x07,
    0x3B, 0xA5, 0xD0, 0x41, 0x5D, 0xAD, 0x0C, 0x59, 0xF7, 0x9E, 0x88 };
```

If the OpenAir Node is to be used with another application, in addition to changing the DevEUI and the AppKey, the AppEUI `static const u1_t PROGMEM APPEUI` has to be adjusted as well. Like the DevEUI, it has to be saved in little endian, so if the end device is within The Things Network, the last three bytes are `0xD5`, `0xB3`, `0x70` which are the network identifier for TTN:

```
1 static const u1_t PROGMEM APPEUI[8] = { 0x48, 0x8B, 0x00, 0xD0, 0x7E,
    0xD5, 0xB3, 0x70 };
```

## 4.2 Gateway

As soon as the Raspberry Pi is powered up, it connects to The Things Network and starts forwarding data to the network so there are no additional actions needed.

The Raspberry Pi that hosts the Gateway can be accessed via SSH or Serial connection. The connection parameters are as follows:

Hostname:     xxxxxxxxxx

User:           xxxxxxxxxx

Password:     xxxxxxxxxx

The board, that connects the Raspberry Pi with the IMST iC880A, also has Serial Rx, Tx, and ground pins laid out as seen in Figure 41. The connection is also labelled on the case.

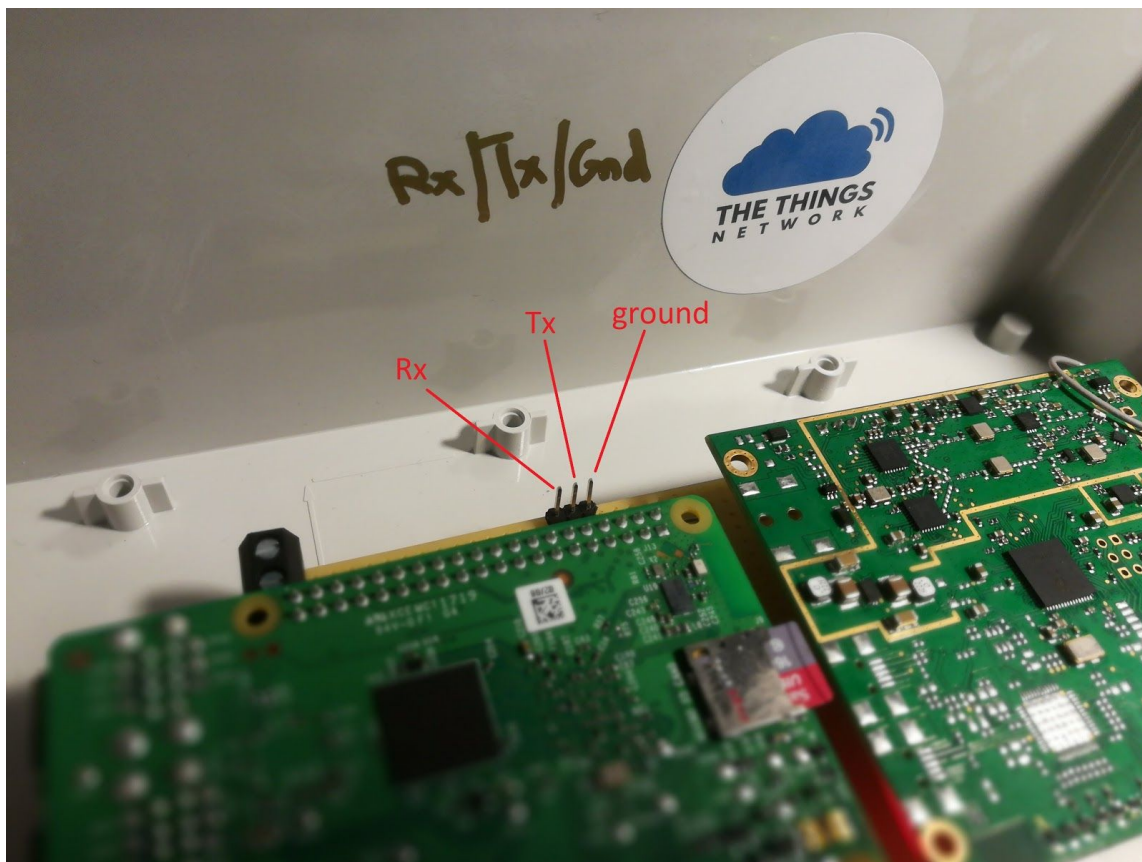


Figure 41: 'Raspberry Pi Serial connection pins

## 4.3 Application

The application runs on the same Raspberry Pi that hosts the gateway, so it can be accessed the same way. The Python script is located at `~/jsonConverter/mqttService.py`. After booting up, a cron job, which is a time based scheduler on Linux platforms, starts a shell script located at `~/jsonConverter/start.sh`, which on his part starts the `mqttService.py` as



sudoer after waiting for 20 seconds. All output is stored in the `~/logs/cronlog` log file. In case of an error, the log file can be checked for the type of error.

If another end device is to be added to the system, the `get_senID` function has to be changed. This function converts the DevEUIs used by The Things Network to the device IDs used by the SensorCloud. If another end device is added to the system, it is necessary to add DevEUI: device ID just below line 4:

```
1     def get_senID(devEUI):
2         return {
3             "00AA76B1E9DEA616": "e6bfb32e-9e0f-4b20-8664-f9ce5229204b",
4             "0014F1CAE3D257C1": "f61ae9b4-0d74-4d0b-8a67-602290d61425"
5         }.get(devEUI, get_senID(devEUI)):
```

The rest of the application works, regardless of how many devices are in the system. However, if there is another measurand to be added, the `on_message` function must be extended to check for that data in the payload like shown underneath:

```
1     if "temp" in parsedPayload["payload_fields"]:
2         tempData = {}
3         tempData["n"] = str(uuid.uuid4()) + ":temp"
4         tempData["t"] = -3000
5         tempData["sv"] = parsedPayload["payload_fields"]["temp"]
6         dfdsMessage["e"].append(dict(tempData))
```

This snippet checks for temperature. To add the other measurand, all that needs to be done is to copy that block and change “temp” in line 1, 3, and 5 to the required unit.

## Abstract And Outlook

The goal of this bachelor thesis was to evaluate the LoRa technology and to create a LoRaWAN network with an example sensor. In the course of this thesis two end devices with different sensor capabilities were implemented and a gateway was built. An application forwarding the sensor data to the University of Applied Science Cologne SensorCloud was developed. The network services are supplied by The Things Network community.

LoRaWAN is a technology with many strengths but also big limitations. So it is important to exactly determine the requirements of the system, before using it as a means of transmitting data. While the range is very high, even in cities, and the energy consumption is relatively small compared to other communication methods such as Wireless LAN. The transmitted data rate is extremely small, and if the end device has to be able to receive data at any time it omits the low energy consumption. But LoRaWAN is an optimal choice for reading air quality data with many sensors in a large area. Also everywhere else where the sensors that far apart and where is no need to send data more often than every five minutes.

One of the two sensors is a Nucleo-L073RZ by STMicroelectronics with a SX1272MB2DAS LoRa RF expansion board. This is able to read NO<sub>x</sub> data and send them via LoRaWAN. The second one is the OpenAir Node, a sensor board developed by the OpenAir Cologne community. This end device is able to read CO data, temperature, and humidity in addition to NO<sub>x</sub> data. The OpenAir Node originally transmitted the sensor data over Wi-Fi, but was equipped with a Hoperf RFM95W LoRa chip and reprogrammed to send this data over LoRaWAN instead.

The gateway is a Raspberry Pi plus an IMST iC880A-LoRaWAN concentrator. It is connected to The Things Network. The gateway is encased in a weatherproof box so it can be installed outdoors. It was planned to install the gateway on the eighth floor of the IWZ of the University of Applied Science Cologne during the course of this bachelor thesis. Sadly, shortly before the installation was due, the permission to install the gateway was rejected. The installation of this gateway outside in great height would cover the entire city of Cologne with LoRaWAN and The Things Network functionality.

The Raspberry Pi used for the Gateway is also the host of an application that is used to pass the sensor data from The Things Network to the SensorCloud. The application is a python script with two MQTT clients. One of them is connected to The Things Network and subscribed to the data of the two sensors, the other one is connected to the SensorCloud and publishes the data after converting it to the SensorCloud JSON format.

With the addition of LoRaWAN capabilities to the OpenAir Node, these can now be deployed in places that do not have a Wireless LAN hotspot within its range. To extend the development, the OpenAir Nodes could be equipped with a battery or a means of energy production, like solar cells or a dynamo on a bicycle, to make them truly independent of any cable connections. This will require some optimization of the end device however. Further With the addition of a deep sleep mode during down time, energy consumption can be impressively reduced.

# Appendix

## Acronyms And Abbreviations

Term	Definition
ABP	Activation by personalization
ADC	Analog-to-digital converter
AMQP	Advanced Message Queuing Protocol
API	Application programming interface
AppEUI	Application identifier
AppKey	Application key
AppSKey	Application session key
CO	Carbon monoxide
CSS	Chirp spread spectrum
DevEUI	Device identifier
DMA	Direct memory access
LoRa	Long range radio technology
LoRaWAN	LoRa wide-area network
LPWAN	Low Power wide-area network
MIC	Message integrity code
MQTT	Message Query Telemetry Transport
NOx	Nitrogen oxide
NwkSKey	Network session key
OTA	Over-the-air activation
RFTDMA	Random Frequency and Time Division Multiple Access
SDK	Software development kit
SDR	Software defined radio
SF	Spreading factor

SoC	System on Chip
TTN	The Things Network

Table 3: List of acronyms and abbreviations

## CD Contents

Path	Content	Remark
\Application	Source code for the mqttService.py application	
\Application\paho	Eclipse Paho MQTT Python Client	
\Bibliography	Bibliographic Sources	Sorted by their reference numbers, excluding published books
\End Devices\ OpenAir Node	Source code for the OpenAir Node	
\End Devices\ OpenAir Node\libraries\ IBM_LMIC_framework	Library for the Hoperf RFM95W	The Folder can be copied as is in the Arduino library folder
\End Devices\ OpenAir Node\hardware\espressif\ esp32	Arduino core for ESP32 WiFi chip	Includes the Preferences.h. In .rar. Has to be extracted before copying to the Arduino hardware folder
\End Devices\Nucleo	Source code for the STM32 Nucleo end device	Contains a SW4STM32 project file

Table 4: CD Contents

# List Of Figures

Figure 1: OpenAir Node V1.1 by Press Every Key	6
Figure 2: LoRaWAN network architecture	8
Figure 3: Receive windows in different device classes	11
Figure 4: LoRaWAN device address	11
Figure 5: Join-request payload	12
Figure 6: Join-accept payload	13
Figure 7: PHY layer uplink structure	14
Figure 8: PHYPayload structure	14
Figure 9: MAC header structure	14
Figure 10: MACPayload structure	15
Figure 11: FHDR structure	15
Figure 12: PHY layer downlink structure	15
Figure 16: LoRaWAN security	16
Figure 17: estimated LoRa service range in Cologne	19
Figure 18: Thirty minute measurement of energy consumption	20
Figure 19: Zoom on heating period	21
Figure 20: Elements of a LoRa packet	22
Figure 21: LoRaWAN packet captured with SDR	24
Figure 22: LoRaWAN packet captured by a gateway	25
Figure 23: Time on air of LoRaWAN packets	25
Figure 24: TTN packets per day under the fair access policy	26
Figure 25: The Things Network Backend	28
Figure 26: Sequence diagram of an uplink	30
Figure 27: Hoperf RFM95W mounted on the backside of the OpenAir Node V1.1	32
Figure 28: STM32 Nucleo-L073RZ with a SX1272MB2xAS shield	33
Figure 29: State machine end device	36

Figure 30: Step one: Click on register device	
38	
Figure 31: Step two: fill out form and submit	39
Figure 32: IMST iC880a LoRa board with pigtail and antenna	41
Figure 33: Raspberry Pi to IMST iC880A connector board	42
Figure 34: Step one, open the TTN console	43
Figure 35: Step two, select gateways	44
Figure 36: Step three, select register gateway	44
Figure 37: Step four: fill out form and submit	45
Figure 38: Step two, select applications	49
Figure 39: Step three, click on add application	50
Figure 40: Step four: fill out and submit form	50
Figure 41: 'Raspberry Pi Serial connection pins	56

## List Of Tables

Table 1: LoRa range depending on the mobile station antenna height	19
Table 2: Pin connection Raspberry Pi to iC880a	42
Table 3: List of acronyms and abbreviations	60
Table 4: CD Contents	61

# Bibliography

- [1] K. Bigge and D. Pöhler, “Stickstoffdioxid-Messung in 12 Städten”, Dept. of Environmental Physics, Heidelberg Univ., Germany, 2016.
- [2] OKLab Cologne, “OpenAir Cologne - Daten für die Stadt”, n.d. [Online]. Available: <http://openair.codingcologne.de/> [Accessed: January 1, 2018].
- [3] J. Thompson et al., “akamai’s [state of the internet] Q1 2017 report”, Akamai, Cambridge, MA, 2017.
- [4] LoRa Alliance, “LoRaWAN What is it?”, LoRa Alliance, Beaverton, OR, 2015.
- [5] Semtech, “LoRa™ Modulation Basics”, Semtech Corporation, Camarillo, CA, May 2, 2017.
- [6] The Things Network, “The Things Network”, n.d. [Online]. Available: <https://www.thethingsnetwork.org/> [Accessed: January 1, 2018].
- [7] A. Minaburo, A. Pelov, and L. Toutain, “LP-WAN Gap Analysis”, February 17, 2016 [Online]. Available: <https://tools.ietf.org/html/draft-minaburo-lp-wan-gap-analysis-0> [Accessed: January 1, 2018].
- [8] Technische Hochschule Köln, “Forschungsprojekt SensorCloud”, n.d. [Online]. Available: [https://www.th-koeln.de/informations-medien-und-elektrotechnik/forschungsprojekt-sensorcloud\\_32813.php](https://www.th-koeln.de/informations-medien-und-elektrotechnik/forschungsprojekt-sensorcloud_32813.php) [Accessed: January 1, 2018].
- [9] LoRa Alliance, “LoRa Alliance Technology”, n.d. [Online]. Available: <https://www.lora-alliance.org/technology> [Accessed: January 1, 2018].
- [10] LoRa Alliance Technical Staff, *LoRaWAN™ Specification Version 1.0.2*, LoRa Alliance, San Ramon, CA, 2016.
- [11] The Things Network, “Address Space in LoRaWAN”, December 14, 2017 [Online]. Available: <https://www.thethingsnetwork.org/wiki/LoRaWAN/Address-Space> [Accessed: January 1, 2018].
- [12] LoRa Alliance Technical Staff, *LoRaWAN™ 1.0.2 Regional Parameters Rev. B*, LoRa Alliance, San Ramon, CA, 2016.
- [13] B. Rembold, “Modellierung von Funkkanälen” in *Wellenausbreitung: Grundlagen - Modelle - Messtechnik - Verfahren*. Wiesbaden, Germany: Springer Fachmedien, 2015.
- [14] R. Holst and K. H. Holst “Bauhöhe, Konstruktionshöhe und lichte Höhe” in *Brücken aus Stahlbeton und Spannbeton: Entwurf, Konstruktion und Berechnung* 6th ed. Bergisch Gladbach, Germany: Ernst & Sohn, 2014.



- [15] IMST Technical Staff, *WiMOD iC880A Datasheet Version 0.50*, IMST GmbH, Kamp-Lintfort, Germany.
- [16] Semtech Technical Staff, *SX1272/73 Datasheet Revision 3.1*, Semtech Corporation, Camarillo, CA, 2017.
- [17] Hoperf Technical Staff, *RFM95/96/97/98(W) - Low Power Long Range Transceiver Module Version 1.0*, Hoperf ELECTRONIC, Shenzhen, China.
- [18] "Electromagnetic compatibility and Radio spectrum Matters (ERM); Short Range Devices (SRD); Radio equipment to be used in the 25 MHz to 1 000 MHz frequency range with power levels ranging up to 500 mW", ETSI EN 300 220-1, January, 2012.
- [19] Semtech Technical Staff, *SX1272/3/6/7/8: LoRa Modem Designer's Guide AN 1200.13 Revision 1*, Semtech Corporation, Camarillo, CA, 2017.
- [20] W. Giezman and T. Telkamp, Lecture, Topic: "The Things Network 2016 Update", Amsterdam, The Netherlands, 2016.
- [21] The Things Network, "The Things Network Backend", May 18, 2017 [Online]. Available: <https://www.thethingsnetwork.org/wiki/Backend/Home> [Accessed: January 1, 2018].
- [22] The Things Network, "Network", n.d. [Online]. Available: <https://www.thethingsnetwork.org/docs/network/> [Accessed: January 1, 2018]
- [23] The Things Industries, "The Things Uno", n.d. [Online]. Available: <https://www.thethingsindustries.com/marketplace/product/the-things-uno> [Accessed: January 1, 2018].
- [24] Pycom Technical Staff, *LoPy 1.0 Datasheet*, Pycom Ltd., Guildford, United Kingdom, 2017.
- [25] Espressif Technical Staff, *ESP32 Datasheet Version 2.0*, Espressif Systems, Shanghai, China, December 1, 2017.
- [26] STMicroelectronics Technical Staff, *UM1724 User manual Revision 12*, STMicroelectronics, Geneva, Switzerland, 2017, December.
- [27] SGX Sensortech Technical Staff, *MiCS-4514 Data Sheet Revision 16*, SGX Sensortech, Corcelles-Cormondreche, Switzerland.
- [28] SGX Sensortech, "SGX Metal Oxide Gas Sensors (How to Use Them and How They Perform)" SGX Sensortech, Corcelles-Cormondreche, Switzerland, July 14, 2014.
- [29] Press Every Key, "opennode\_hw", October 29, 2017 [Online]. Available: <https://github.com/everykey/openairnode-hw> [Accessed: January 1, 2018].

- [30] The Things Network, “The Things Gateway”, n.d. [Online]. Available: <https://shop.thethingsnetwork.com/index.php/product/the-things-gateway/> [Accessed: January 1, 2018].
- [31] The Things Network, “MultiConnect Conduit”, n.d. [Online]. Available: <https://www.thethingsnetwork.org/docs/gateways/multitech/> [Accessed: 2017, November 27].
- [32] Kerlink, “Wirnet Station”, n.d. [Online]. Available: <https://www.kerlink.com/product/wirnet-station/> [Accessed: January 1, 2018].
- [33] Lorrier “Introducing LR2”, n.d. [Online]. Available: <https://lorrier.com/#introducing-lr2> [Accessed: January 1, 2018].
- [34] TTN Zürich, “The Things Network iC880a-based gateway”, May 11, 2017 [Online]. Available: <https://github.com/ttn-zh/ic880a-gateway/tree/spi> [Accessed: January 1, 2018].
- [35] Raspberry Pi Technical Staff, *Raspberry Pi Compute Module Datasheet Version 1.0*, Raspberry Pi (Trading) Ltd., Cambridge, United Kingdom, October 13, 2016.