



Dominik Delgado Steuter, Johannes Dorfschmidt, Jannik Lukas  
Hense, Darvin Schlüter and Alisa Stiballe

## Communication network for emergency situations using LoRa

System Design Group Project 22/23

21. April 2023

Please cite as:

Dominik Delgado Steuter, Johannes Dorfschmidt, Jannik Lukas Hense, Darvin Schlüter and Alisa Stiballe, "Communication network for emergency situations using LoRa," System Design Group Project 22/23, Department of Computer Science, Paderborn University, Germany, April 2023.

# **Communication network for emergency situations using LoRa**

System Design Group Project 22/23

authored by

**Dominik Delgado Steuter  
Johannes Dorfschmidt  
Jannik Lukas Hense  
Darvin Schlüter  
Alisa Stiballe**

**Computer Networks  
Department of Computer Science  
University Paderborn**

Supervisor: **Florian Klingler**  
Advisor: **Simon Welzel**

## Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.

I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Dominik Delgado Steuter)

(Johannes Dorfschmidt)

(Jannik Lukas Hense)

(Darvin Schlüter)

(Alisa Stiballe)

Paderborn, 21 April 2023

---

# Abstract

---

In view of today's circumstances, it is absolutely necessary to prepare for environmental disasters. Above all, a failure of the internet and mobile communication systems has serious consequences. In such cases, our project based on the LoRa frequency technology offers a backup network enabling emergency communication. Therefore, Raspberry Pi's with LoRa shields are used as mesh gateways, to which several end devices can connect via a WLAN. Messages can be written, sent, and received via a web-based chat client. All messages and routing information are stored in a database only accessible by the broker. To improve utility, a channel system is used to directly address the fire department, the ambulance or the police.

---

# Contents

---

<b>Abstract</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 The Issue . . . . .	1
1.2 The Solution . . . . .	1
<b>2 System Architecture</b>	<b>2</b>
2.1 Structural Overview . . . . .	2
2.2 Realization in hardware . . . . .	4
<b>3 LoRa Driver</b>	<b>5</b>
3.1 Utility . . . . .	5
3.2 LoRa Class . . . . .	5
3.3 Usage . . . . .	6
<b>4 Messages</b>	<b>8</b>
4.1 Protobuf . . . . .	8
4.2 Message Structure . . . . .	9
<b>5 Local chat website</b>	<b>12</b>
5.1 Introduction . . . . .	12
5.2 Features . . . . .	12
5.3 Structure . . . . .	13
5.4 Setting up the webserver . . . . .	14
5.5 User interface . . . . .	14
5.6 Sockets . . . . .	15
5.7 Conclusion . . . . .	16
<b>6 Broker Setup With Database</b>	<b>18</b>
6.1 General Architecture . . . . .	18
6.2 The Database . . . . .	19

---

<b>7</b>	<b>Routing</b>	<b>22</b>
7.1	Concept . . . . .	22
7.2	Dynamic path finding . . . . .	22
7.3	Fixed path messaging . . . . .	23
<b>8</b>	<b>Installation of Packets and default settings</b>	<b>25</b>
8.1	Installation from Scratch . . . . .	25
8.2	Raspberry Pi Settings . . . . .	25
8.3	Install Programs and Packets . . . . .	26
8.4	WAP is already active . . . . .	27
<b>9</b>	<b>Access Point</b>	<b>28</b>
<b>10</b>	<b>Starting up the System</b>	<b>33</b>

---

## Chapter 1

# Introduction

---

With the rapidly advancing digitalization around the globe also the dependency on internet and mobile communications rises in all areas of life. Whether at work or during leisure time, people are always networked via the internet. Therefore, one does not want to imagine a break-down of these communication systems. But considering the more frequent natural disasters due to global warming as well as the tense political situation, emergency services have to be developed and deployed now.

### 1.1 The Issue

Independent of the cause or the extent of the network outage, it is necessary to secure a basic possibility to communicate. Above all, it must stay possible for citizens to make emergency calls, especially in situations like a network outage, which is sure followed by more emergencies. Although, there are more facilities, we so far decided to model the police, the ambulance and the fire brigade.

### 1.2 The Solution

The network introduced in this work is made especially for emergency communication. Overall, it is base on the low range wide area (LoRa) technology, which is characterized by its wide range of coverage while keeping a low power consumption. The downside is, that the messages sent up to 2 kilometers in cities as well as up to 15 kilometers in open fields can only be transferred with a rather low data rate [6]. Furthermore, the network contains several *gateways*, which are accessible via a web-based chat client and a special node called *broker*, which is connected to a database.

---

## Chapter 2

# System Architecture

---

In this chapter, the network's structure is introduced. Therefore, the network is split up into several parts, which then are described in their function. Furthermore, a general overview is created of how communication is handled.

### 2.1 Structural Overview

There are many ways to structure a communication network and to enable efficient messaging. In our implementation, we decide to produce a rather straight forward network type without a wide variety of devices. This decision is based on our limited hardware, we were working with, and to allow focusing on other domains. To provide a clear structure in our communication network, we assigned only four different roles. The gateway to forward messages, the web server to allow a web

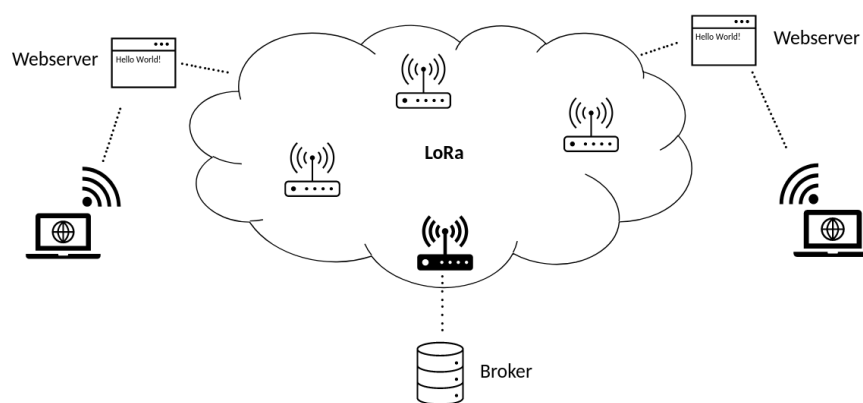


Figure 2.1 – Conceptual overview of the network



based chat, the broker to communicate outside LoRa, and the database to store relevant data. Each of those are further described in the following sections.

### 2.1.1 Gateways

The main components of the network are the gateways. They provide a service to forward messages via LoRa and therefore manage the whole information traffic of the network. Based on the used LoRa frequency technology [6], messages can be sent and received for up to 2 km distance in cities and up to 15 km on open fields. Depending on the desired area of network availability, a few or several hundred of those nodes can be set up. Each gateway can also be connected to a web server to allow access to the chat system. Necessary for our emergency use case is, that every emergency contact possesses its own gateway and chat, that is connected to the overall network.

### 2.1.2 Web server

Web servers are used to visualize the chat and generate messages that shall be sent over the network. To accomplish this, they open up a local network to which end devices can connect via WLAN. Messages can then be read/written on the web-based chat client, from which they are sent back to the web server and subsequently over the gateways to the right addressee. The client gives an overview of all available chat partners and a history of recent messages like commonly used chat programs. For more information, see chapter 5.

### 2.1.3 Broker

In principle, the broker differs only slightly from a gateway in the way, that it's not only communicating with other gateways via LoRa, but also with the database. For more information, see chapter 6.

### 2.1.4 Database

In terms of security, the database is the most important unit. It stores every message sent over the network. Furthermore, metadata like the location of the nodes get stored as well. Important information for routing between each node is also stored in a separate table. For more information, see section 6.2.

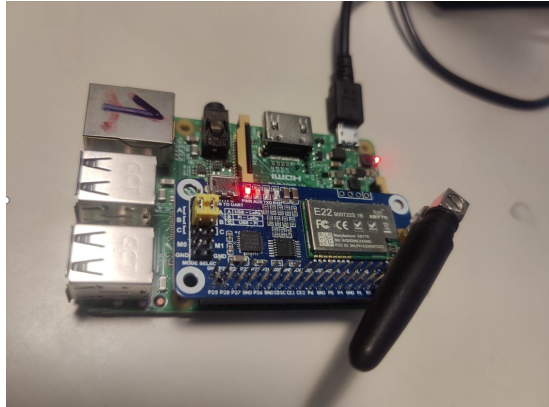


Figure 2.2 – Microcontroller with a LoRa shield

## 2.2 Realization in hardware

To build up our network, we used several Raspberry Pi's of the model 3B [1]. They are extended with SX1262 868M LoRa HAT's [11], which enable the technical fundamentals to use the frequency technology LoRa. In compliance with the legal requirements, we communicate over the freely usable 868Hz frequency band [2]. Due to the fact, that we only had limited hardware available, the previous declared roles are not implemented on one dedicated device per role. More precisely, in every case, the gateway and the web server are hosted on the same microcontroller. In case of the broker, the web server as well as the database runs on the same hardware. Although a clear structural division was not feasible to our network realization, it does not implicate a loss in functionality.

---

## Chapter 3

# LoRa Driver

---

### 3.1 Utility

The LoRa-Driver was build to get basic access to fundamental function such as sending and receiving with the Waveshare SX1262 868M LoRa HAT. It communicates with this LoRa-Module on hardware level and is currently hardcoded for the 3rd generation Raspberry Pi. It provides a new Class for LoRa usage that allows access to the previously mentioned functions. This driver is created from the official Waveshare SX1262 Driver, but that specific driver neither worked with their recommended hardware, nor was it written properly. No other driver was found for our specific hardware. So this one still has some aspects of the Waveshare driver, but it was specifically tailored for our case. Some of those aspects are remnants of the Waveshare driver that must be used for configuration purposes but are not specifically needed for this project (referring to the N.A.entries in the `cfg_reg` table).

### 3.2 LoRa Class

#### 3.2.1 Settings

The LoRa-Module provides multiple settings. To configure the Module one has to change the entries in the `self.cfg_reg`.

---

```
1      self.cfg_reg = [0xC0, 0x00, 0x09, 0x00, 0x00, ↵  
                      0x00, 0x62, 0x00, 0x17, 0x00, 0x00, 0x00]
```

---

This configuration must be sent over the standard serial port `'/dev/ttyS0'` to establish the connection between the Raspberry Pi and the LoRa-Module. A certain amount of time is required for the connection to be successfully created, so a sleep of around a second or higher is recommended. Important is that the 22nd Pin must be on low

and the 27th Pin must be high if you want to change the configuration. Both must be set to low if the send and receive functions are to be used. For more information on the possible settings, see the figure below 3.1.

We do not provide a dedicated function to change the settings, to change them one has to change them directly in the *cfg\_reg*.

### 3.2.2 Send & Receive: Basic Knowledge

Send and Receive are the main functions for the LoRa driver, they are controlled via a lock flag (written as an object variable: *self.locked*) so that they will not interfere in each others work while they are in the critical phase. Both functions are bound to the LoRa class. The send function will automatically frame the data that needs to be sent. The receive function will only accept messages with the correct framing and will also deframe the message to get the raw data.

**Important** to notice is, that the receive function will store the received data in an array. This is necessary to receive all incoming messages. Otherwise it can happen that multiple messages are seen as a single message, if there is a sudden burst in incoming messages.

The data that is transmitted is not automatically encrypted. If necessary, it has to be done beforehand.

### 3.2.3 Send & Receive: Specific Message

Send and Receive have no function to distinguish messages. Receive receives **all** messages, if only specific ones need to be seen it has to be filtered with information in the data that was transmitted. Thus the receiver must be written in the data that wants to be send and not in some kind in the LoRa Class. The Receiver must inspect the received data if the message was destined for itself.

## 3.3 Usage

The provided LoRa class can be used by simply importing the *driver.py* file. Create a new object that can use the previously described methods with a class parameter that will be the node address.

---

```
1 8 = my_node_address
2 lora = LoRa(my_node_address)
```

---

In this example, the initialised node will have the address 8.

The settings can be changed through the previously described *cfg\_reg*

cfg_reg index	usage	current default entry	alternative entries
0	Memory Address	0xC0 Don't save settings after node restart	0xC2 Save settings after node restarts
1	Unknown	N.A.	N.A.
2	Unknown	0x09	N.A.
3	High Address	N.A.	N.A.
4	Low Address	N.A.	N.A.
5	Temporary Node Address	N.A.	N.A.
6	Baudrate + Air Speed	0x62 Baudrate: 9600: 0x60 Air Speed: 2400bps: 0x02	Variable Baudrate and Air speed Corresponding values are defined as class variables
7	Buffer Size + Power + 0x20	0x20	N.A.
8	Frequency	0x12 Frequency is set to: initialised_frequency - 850	Can be set to anything reasonable: 850 - 888MHz But not recommended!
9	RSSI + 0x43	N.A.	N.A.
10	High Crypt	N.A.	N.A.
11	Low Crypt	N.A.	N.A.

Table 3.1 – CFG\_Reg Entrylist [8]

---

## Chapter 4

# Messages

---

Every data sent over the LoRa network is serialized into a clear structure. In that special use case, the mechanism *Protocol Buffers* (Protobuf) [3] from Google is combining all requirements into a single, easy to use format. Furthermore, we created three different types of messages Broadcast, Setup and Chat message, that indicate dedicated handling dependent on the type.

### 4.1 Protobuf

The main utility of Google's Protobuf format is to serialize data into a clear structure, while supporting important factors like efficiency and security. Therefore, it stands out as a good ready-to-use solution to convert data of several data types into binary code independent on the used programming language. To model a data structure, one simply declares all requested fields with a name and a type in a *.proto* file format. With the comprehensive python package *Protobuf* [4] one can then compile the *.proto* file format into python code, that handles the whole serialization into the desired structure.



**Figure 4.1** – Serialization of data using Protocol Buffers by Google

## 4.2 Message Structure

For this network implementation, we used a structure containing ten data fields as shown in 4.1, whereof two are optional and therefore not contained in every message. Starting with the first two attributes, *sender\_address* and *receiver\_address*, we can determine the temporary start and destination of a message. Temporary means, that those values change on every hop. The field *path* is successively generated, while the message propagates through the network. This is done by concatenating the address of the visited node for every hop. Therefore, one can comprehend the travelled way of the message through the network at any time. Although the field *send\_to\_path* sounds similar to the previous one, it has a significant difference in its purpose. While *path* is generated throughout the travel, *send\_to\_path* describes a fixed path, that the message should be sent over. It is set once, when initiating the message, and not changed throughout the forwarding. The next field *sender\_time* saves the UTC timestamp from the moment the message is sent and is therefore used as a message ID. Information about the geographical location of the sending node is stored in *location*. It is divided into two values, the north-south position latitude and the east-west position longitude. Also, the of the sender is added in field *name*. The last of the mandatory field is the *type* to indicate the type of the message depending on the use case. The optional field *end\_receiver\_address* stores the address of the end receiver, while the optional field *channel* names all currently available chat channels. The last optional field *text* contains the content of *chat messages* or other extensive data.

### 4.2.1 Type 1: Broadcast

A broadcast message is sent, when a new node is initialized. The new node is not known to any other node and vice versa. Therefore, no fixed routing can be used. Our solution is to simply send a *broadcast message* to every neighbor node, and they also send it to every of their neighbor nodes, until it reaches the broker. No Node will send the same broadcast twice. Thus, the *receiver\_address* always contains the broker address 0 and the *sender\_adress* always contain the node that started the broadcast. Another important field for broadcast messages is the *path*, because once the message reaches the broker, it contains the shortest and therefore fastest path to the new node. This path is then saved in the database and can be used for further communication, like the following *setup message*. The broker also updates its list of available chat channels with all nodes contained in the *path*. While the optional fields *channel* and *text* are not used and left empty, the remaining fields are all initialized as described in the previous section. Whereas the *type* is obviously set to 1, to signalize how to evaluate the message, once it is received by another

node. In general, every node has to initiate one *broadcast message* as a first step, but never again after receiving a *setup message*. This step will start immediately after executing the *run.py* script.

Number	Name	Type	Function	Example
1	sender_address	int32	Temporary sender address	4787
2	receiver_address	int32	Temporary receiver address	6887
3	path	string	Successively stores the node address from every hop	"4812,4824,4222": 4812 is the original sender 4824 is a stopovers 4222 current location of this message
4	send_to_path	string	The path that is used while forwarding the message	"4812,4824,4222,0"
5	sender_time	int64	The timestamp at which this message was sent in UTC format	1005311654014
6	location	string	GPS location of the node	"51.71905,8.75439"
7	name	string	The name of the sender	"Max Musterman"
8	type	int32	Defines the usecase of the message	1: Broadcast 2: Setup 3: Chat Message
9	end_receiver_address	int32	The last receiver of this message	4643
11	text	string	Content of chat messages	"Thank you for the fast support!"

**Table 4.1** – Overview of the message structure declared by the *.proto* file.

### 4.2.2 Type 2: Setup

To inform a new node, that the broadcast message reached the broker and the broadcast is therefore successfully done, a *setup message* with *type* 2, is sent back to the new node. Now, the stored path to the dedicated node can be used for an efficient forwarding. Special with *setup messages* is, that the field *path* has a second use by storing all available node addresses concatenated to the path separated with a ";". Therefore, the new node can update its list with all available nodes and the chat channels on the website. Furthermore, also the optional field *end\_receiver\_address*, containing the address of the new node, is used for the static routing. The optional field *text* is not used in *setup messages*, but all other fields contain data as describes beforehand. Once the setup message reached the new node, it stores the inverted content from *path* as the shortest route to the broker, which is hereafter used for every chat message. After the web server is then updated with the available channel, sending and receiving of *chat messages* is enabled.



### 4.2.3 Type 3: Chat message

When the installation of a node is successfully done, *chat messages* can be sent over the web client. Those messages of *type 3* are always first forwarded to the broker, thus the broker has access to the database, that stores paths to every node. So at first, the *send\_to\_path* contains the path to the broker, but is changed throughout the forwarding. The *end\_receiver\_address* is an important value for this type, because it determines the destination for the second half of the routing. So when a *chat message* reaches the broker, the *send\_to\_path* is overwritten by the path to the end receiver loaded from the database. The message content written by the user via the website is stored in the optional field *text*, that is not optional for *type 3 chat messages*. Also, the *path* is reset, once the message reaches the broker. When the message arrives at the end receiver, the content is uploaded to the web server and thereafter displayed in the web client. A response sent by the receiver would be evaluated as a *chat message* according to the just described process.

---

## Chapter 5

# Local chat website

---

### 5.1 Introduction

The chat website is an integral part of the chat webserver and is designed to provide users with a user-friendly interface for communication and collaboration during a disaster. The website can be accessed through a browser on modern devices and is designed to be accessible and easy to navigate, even in stressful situations. The website is hosted separately on each node.

### 5.2 Features

The chat website provides a variety of features for users, including a text messaging system, changing the style of the frontend (themes) and allowing to select multiple channels to send messages. When a user connects to the WLAN network provided by a node in the LoRa network, the chat website will automatically open, allowing them to quickly and easily communicate with others. This explained more in detail in chapter 9.

In addition, the chat website also provides a user-friendly interface for filling out online forms. This allows individuals affected by a disaster to provide critical information, such as location and type of emergency, to emergency stations. The form data is converted to a human readable string. The text is stored in a broker, a database that stores every message, and can be used by emergency responders to quickly and effectively allocate resources and provide assistance. There is a map to view and select

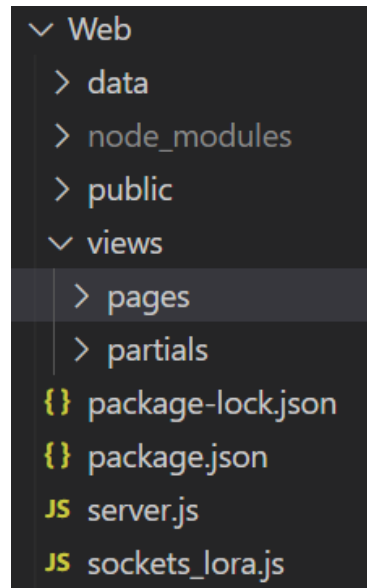


Figure 5.1 – File structure of local website

## 5.3 Structure

In the following is a briefly description of some important files / folders. The root folder of the Node project is called `/Web` and is located inside the main project.

### **data**

Includes a json file with all channels. Initial plan was to update this file by sending a request to the broker. Currently it's a static file we update manually.

### **public**

- **css, js** Css style files and javascript files
- **themes** Multiple theme folders. Each theme includes audio files, js, css and images.

### **views**

- **partials** Template file for header, footer, forms, ...
- **views** Template file the pages chat, settings, start

### **server.js**

Is the main file. Webserver can be started with `node server.js`

### **sockets\_lora.js**

Transfers data from website frontend (client) to the LoRa python program with sockets.

**package.json**

Contains information about the node project and the required dependencies.

**Other**

The folder *node\_modules* and file *package-lock.json* are created by node. These can be ignored.

## 5.4 Setting up the webserver

*server.js* is the main file and we used node v16. For We choosed port 8081, if you prefer to use port 80 (default http) or 443 (default https) then node must be run as *sudo*.

Additionally we created a *systemd* [9] service file for automatically (re)-starting the application.

## 5.5 User interface

We use Embedded JavaScript templating (EJS) as a template engine. This allows us to create html templates with embedded JavaScript for dynamically changing the content. The interface is divided in four pages (views):

### 5.5.1 Chat (.ejs)

Main page for listing all channels.

### 5.5.2 Settings (.ejs)

Provides access to various settings. New users (or those who have cleared their cookies) will be redirected to the settings page.

### 5.5.3 chat (.ejs)

This page displays the chat page and additionally a service form. The service form will be parsed to an string and handled as a normal text message.

### 5.5.4 map (.ejs)

Shows all nodes on a leaflet map. By now, there are only 3 layers of tile map images in the *public/map/tiles* folder. More tile images can be downloaded and moved to the tiles folder to show a more clear map. As the tile provider, OpenStreetMap was used. Its url to download the tiles is <https://a.tile.openstreetmap.org/z/x/y.png>, where z, x, z needs to replaced by the tile coordinates.

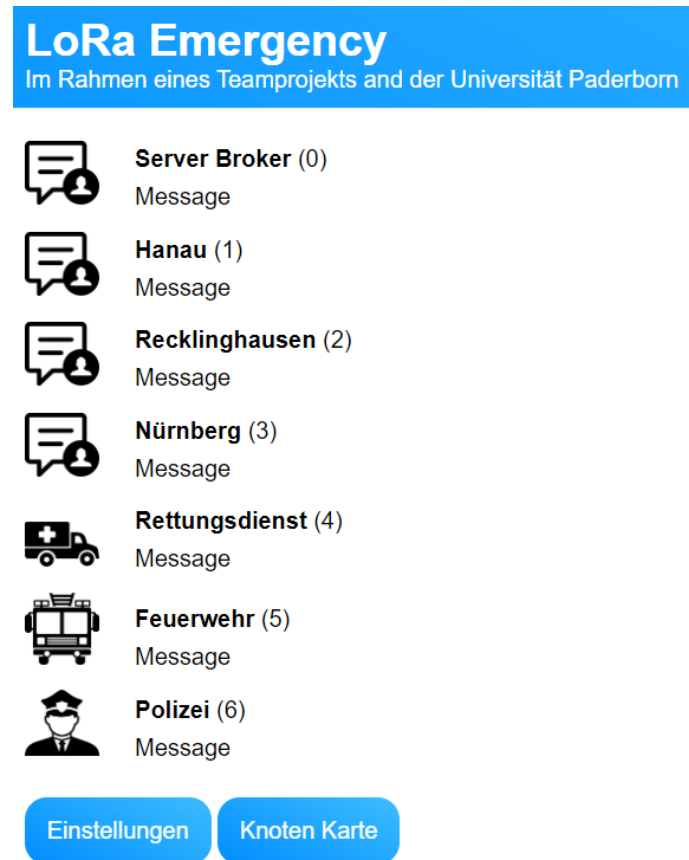


Figure 5.2 – Channel overview page

## 5.6 Sockets

Sockets are an important ingredient for our webserver. It allows to send and receive messages between programs in realtime.

The library `socket.io` [7] is used to connect the client with the nodejs backend. The backend then uses the `zeromq` socket library [10] to connect with the python LoRa program. ZeroMQ is using queues which is an important feature for a reliable communication, because users can send multiple messages in a short time to the backend and the backend then is holding these messages in a queue until the (slow) LoRa module is finished sending these. With `zeromq`, we use the Request/Reply patterns. This may allow the program to check if a message was received or send properly because each socket always requires an answer. We are using 2 `zeromq` sockets, one for sending messages and one for receiving messages. `Socket.io` allows bidirectional communication with only 1 socket.

## LoRa Emergency

Im Rahmen eines Teamprojekts and der Universität Paderborn

Bitte gib die folgenden Daten ein, um auf den Chat zugreifen zu können:

Name

Darvin

Theme

Default (Licence free)

☐ Darkmode?

☒ Disable sound?

Diese Option verringert die Übersichtlichkeit, bestenfalls nur aktivieren, wenn Sie Zugriff aus alle Knoten brauchen, z.B. Rettungskräfte:

☒ Nachrichten von allen Knoten erhalten

Speichern

Figure 5.3 – Settings page



## Rettungsdienst (4)

Im Rahmen eines Teamprojekts and der Universität Paderborn

Nicht jedes Feld muss ausgefüllt werden. Bitte so kurz wie möglich.

Hinweise zu Ihrem Standort. Wir haben bereits die Koordinaten (51.2;8.3) Ihres LoRa Nodes.

Straße, Hausnummer, Ortsbeschreibung, ...

Beschreiben Sie den Notfall

Was ist passiert? Notfallbeschreibung

Typ

Bitte wählen

Anzahl betroffener Personen

Wenn unbekannt: Schätzen

JETZT ABSCHICKEN

Verbunden / Online

Verbunden mit Kanal **Rettungsdienst** (ID: 4) 11:25

Figure 5.4 – Service form and chat page

## 5.7 Conclusion

The chat website is designed to provide a reliable and efficient communication tool for emergency responders and affected individuals during a disaster. Its user-friendly

## LoRa Emergency

Im Rahmen eines Teamprojekts and der Universität Paderborn

Standorte der Knoten. Änderungen der Standorte werden eventuell in einer Krisensituation nicht übernommen.  
(Simulierte Standorte für Demo)

Zurück

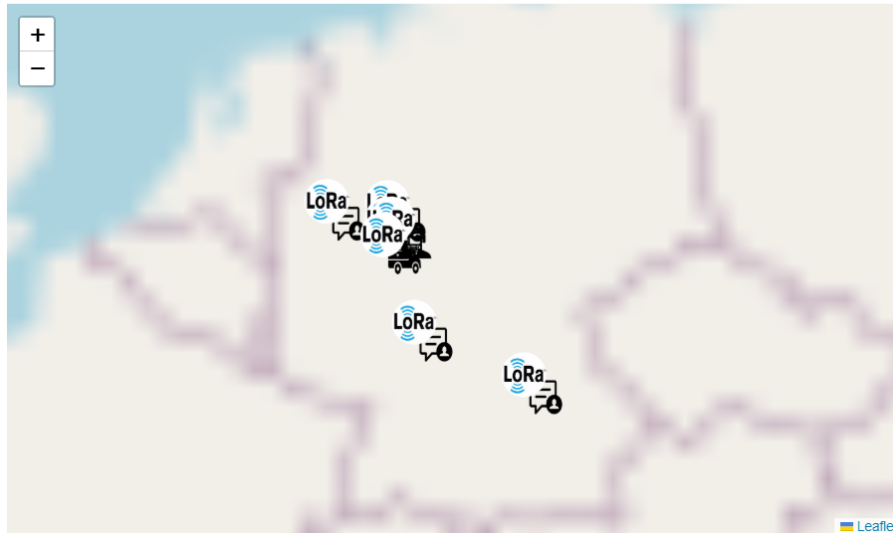


Figure 5.5 – Leaflet map page

interface and versatile features make it a valuable tool for a variety of communication and collaboration needs. The chat website's ability to function without internet connectivity makes it an ideal solution for emergency communication during a disaster.

---

## Chapter 6

# Broker Setup With Database

---

The broker is the main central node in this LoRa network. It hosts the database and manages the communication between all other nodes.

### 6.1 General Architecture

The core component of the broker is a Node.js file. This file handles the database (creates the tables, inserts and queries from it) and provides an interface between LoRa and the database. LoRa and Node.js communicate over a ZeroMQ-Server hosted by Node.js.

There is also a website which displays the database data for better supervising.

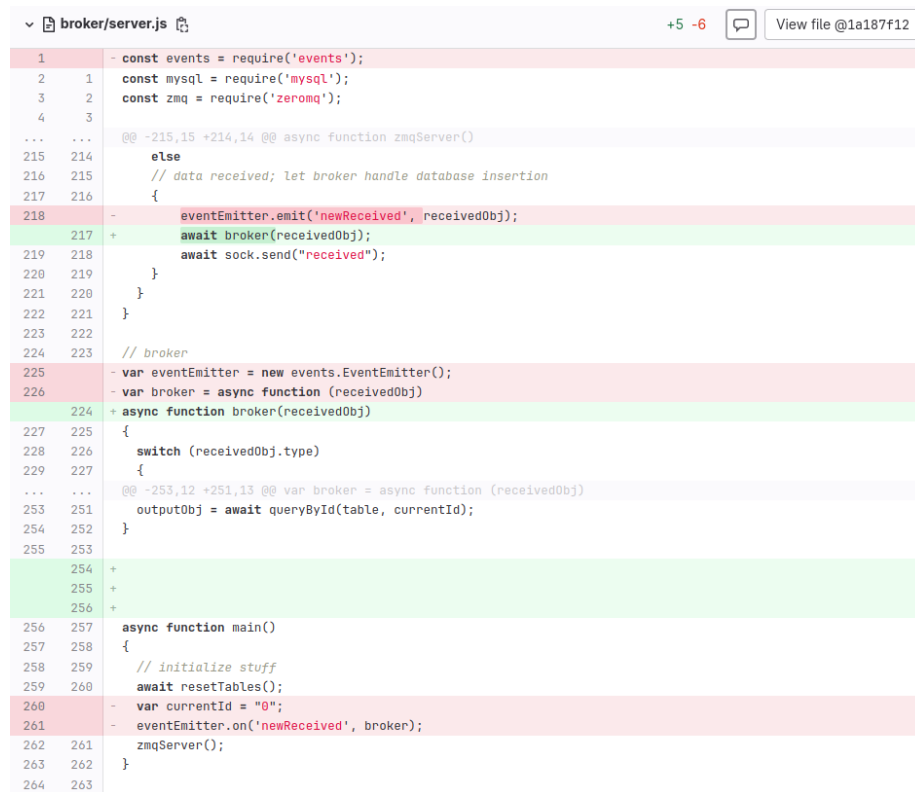
#### 6.1.1 The Node.js File

We had no prior experience with Node.js. The software seemed fitting for our use case as it enables asynchronous message handling and made the impression that it is easy to get into. There are, however, many caveats with using JavaScript. Especially understanding the promise/resolve-patterns with await statements was a struggle. At one point we also used an event system which was later scrapped because it introduced more complication and was not really necessary. See Figure 6.1.

We also ran into multiple dependency issues. It was not always easy to install the correct version of Node.js and especially of ZeroMQ.

The source code is well documented with comments. The main()-function initializes (resets) the database tables and starts the ZeroMQ-server. When the ZeroMQ-server receives a request for the path, the server answers with the corresponding path. If the server receives data, it makes a JavaScript object out of the received JSON-string and passes this object to the special "broker"-function which handles the proper database insertion.





```

1  - const events = require('events');
2  1  const mysql = require('mysql');
3  2  const zmq = require('zeromq');
4  3
...  ...
215 214   @@ -215,15 +214,14 @@ async function zmqServer()
216 215       else
217 216       // data received; let broker handle database insertion
218 217       {
219 218           - eventEmitter.emit('newReceived', receivedObj);
220 219           + await broker(receivedObj);
221 220           await sock.send("received");
222 221       }
223 222   }
224 223   // broker
225 224   - var eventEmitter = new events.EventEmitter();
226 225   - var broker = async function (receivedObj)
227 226   + async function broker(receivedObj)
228 227   {
229 228       switch (receivedObj.type)
230 229       {
231 230           @@ -253,12 +251,13 @@ var broker = async function (receivedObj)
232 231           outputObj = await queryById(table, currentId);
233 232       }
234 233   }
235 234   +
236 235   +
237 236   +
238 237   async function main()
239 238   {
240 239       // initialize stuff
241 240       await resetTables();
242 241       - var currentId = "0";
243 242       - eventEmitter.on('newReceived', broker);
244 243       zmqServer();
245 244   }
246 245   }

```

Figure 6.1 – Git diff of the removed event system

### 6.1.2 Displaying the Database

The broker also hosts an Apache web-server which displays the database tables. The site is visible at the local IP address of the broker node. The database gets queried with PHP, the rest of the site is just HTML and CSS.

To unify the broker in the future the database serving could also be handled by Node.js instead of Apache and PHP. For this the Node.js framework Express could be used.

## 6.2 The Database

For the database MariaDB is used. There are two tables: The routing table and the message table. In the past, there was also an emergency table. That one was meant to log specific emergency information, but got unified into the message table later on.

### 6.2.1 Routing Table

The routing table handles the routing between the nodes. It consists of (server\_address, path)-pairs. Whenever a node sends a broadcast message, the broker saves the shortest path to the corresponding node.

When node A wants to send a message to node C, LoRa asks the broker for the path to node C. The broker queries the database and if there is a valid path to node C, it replies with this path.

### 6.2.2 Message Table

This table saves all of the regular chat messages and associated metadata like the username and used communication channel of the sender. Every entry in this table has a unique ID. This conveniently saved storage of data can be used for a lot of operations. E.g. the nodes can fetch this data from the broker in order to get and display the chat history in case the node temporarily broke down.

### 6.2.3 Extending the Database with Tables

Figure 6.2 shows how data gets inserted into the message table. For inserting data into a new, custom table, this figure can be a reference.

```
// OTHER TABLES
// insert into other tables.
// written as a switch-statement, so this can be easily extended with new additional tables
switch (table)
{
  case "message":
    var insertQuery = "INSERT INTO message (sender_address, receiver_address, path, send_to_path, "
      + "sender_time, location, name, end_receiver_address, text) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)";
    var insertArray = [receivedObj.sender_address, receivedObj.receiver_address, receivedObj.path,
      receivedObj.send_to_path, receivedObj.sender_time, receivedObj.location, receivedObj.name,
      receivedObj.end_receiver_address, receivedObj.text];
    break;
}

con.query(insertQuery, insertArray, function (err, result)
{
  if (err) throw err;
  currentId = result.insertId;
  resolve(currentId);
});
```

Figure 6.2 – Inserting data into the message table

The new table also has to be declared in the resetTables()-function. A block similar to the other database blocks should be added. See Figure 6.3.

```
// delete tables (if they exist) and then initialize them
function resetTables()
{
  return new Promise((resolve) => {
    con.connect(function(err)
    {
      if (err) throw err;
      // now connected to database

      // rebuild routing table
      var sql = "DROP TABLE IF EXISTS routing";
      con.query(sql, function (err, result) {
        if (err) throw err;
      });
      var sql = "CREATE TABLE routing (sender_address INT(255), path VARCHAR(255))";
      con.query(sql, function (err, result) {
        if (err) throw err;
        console.log("Table 'routing' resetted");
      });
    });
  });
}
```

Figure 6.3 – Message table is declared

---

## Chapter 7

# Routing

---

### 7.1 Concept

There is a wide variety of routing protocols, that can be used for network communication. Due to the fact, that the LoRa frequency technology does not enable high transmission speeds as WLAN for instance [quote], our communication protocol does not have to be maximised in that property either. Instead, we focus on reliability to ensure that each of the emergency messages arrives at its destination. The design of the used routing algorithm is also based on some assumptions about data rates, size of the network and locations of the nodes. In general, we implemented two different approaches for different situations. One approach is building up shortest paths from node to broker and the second one is about using those paths as described in the dedicated sections below.

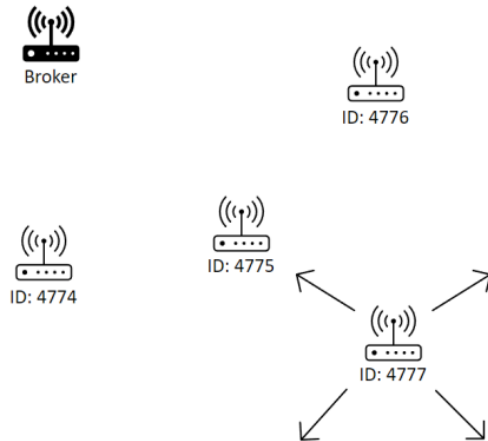
### 7.2 Dynamic path finding

Once a new node is about to be set up, it has to notify the broker and therefore the data server. This first *setup message* is forwarded through the network via *broadcast*. That is a simple non-address-based algorithm to ensure a message is spread over the complete network. In order to achieve that, the *setup message* is sent to every neighbour node. Every node receiving a broadcast message also sends it on to all its neighbour nodes. To not create an infinite cycle, a received broadcast message is only redirected once. Because the end receiver of our *setup message* is the broker, the message is not forwarded once it reaches it. Therefore, duplicates of the same *setup message* will be discarded by the broker. The previously mentioned neighbour node is defined by being installed in reach to receive the frequencies correctly and parse the message. For the realisation the *setup message* has to carry some information

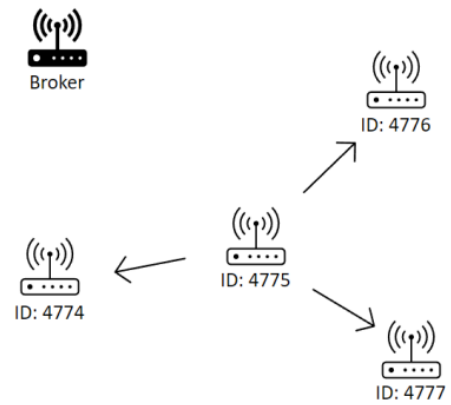
about the path as described in the section 4.2.2. The first *setup message* that reaches the broker is evaluated. The travelled path is stored in the database associated with the new node's address and the list of available nodes is updated. This method also ensures, that only the shortest and therefore the fastest path is used for future messages to achieve desired efficiency. In further communication, the broker can now use the stored path to send a message to a dedicated node. Also, the new node is thereupon informed about the successful setup and can use its stored path to the broker as described in the following section.

### 7.3 Fixed path messaging

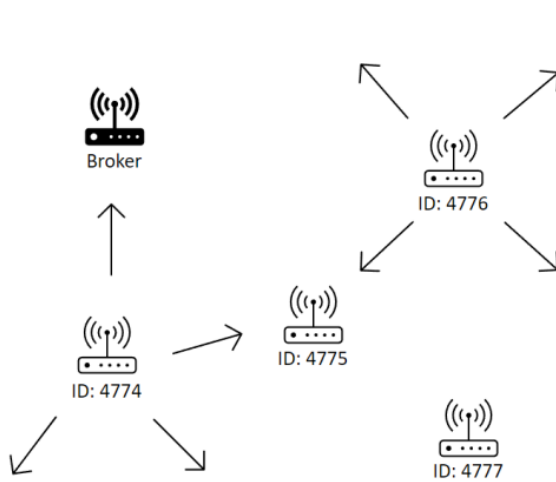
After a node stored the fastest path to the broker, it can now send every message over that path. The broker can then read the end receiver out of the message body and redirect it over another stored path to the dedicated receiving node. Those paths are stored in a simple and efficient way. Each address of a node on the way is concatenated as text, separated by a comma. Thus, once a message reached a node, it just has to be evaluated, if the reached node is part of the path. In case it reached a legit stopover, the new temporary receiver is the following node address in the path and the message is sent on. If it reached the end receiver, there is no following node in the path and the message can be processed by the receiver. The receiver is the first listed in this path and the sender is the last one that is listed. This is due to the structure of creating this shortest path in the broadcast. The last case is, that a message is received by a node, that is not mentioned in the path. Therefore, the message will be discarded. By using the broker as a central distributor, not every node has to store a full table of all nodes and paths. It also allows saving every sent message in the database. A possible downside is, that the broker becomes a significant bottleneck throughout the network. According to our considerations, this should not lead to impairments or data loss, due to moderate message frequency. However, higher message rates might not be supported by our solution. Moreover, this method ensures secure messaging only as long as nodes are not changing its positions or fail to communicate.



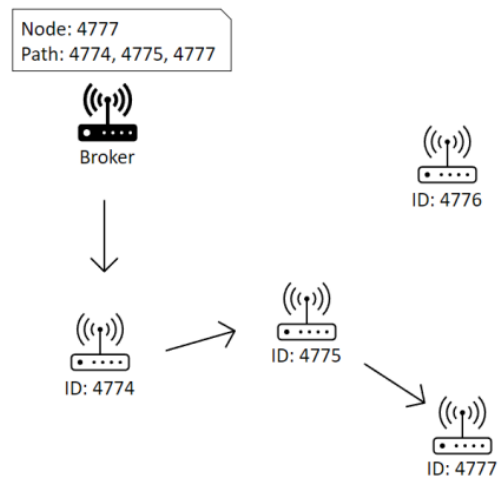
**Figure 8.1** – Node 4777 starts the broadcast.



**Figure 8.2** – The broadcast is proceeded by neighbour nodes.



**Figure 8.3** – The broker received the *setup* message



**Figure 8.4** – A message is sent over a fixed path.

---

## Chapter 8

# Installation of Packets and default settings

---

### 8.1 Installation from Scratch

To use the WaveShare LoRa-Module and all files we provide some packets have to be installed and some settings on the Pi itself have to be changed. This is a brief guide for all necessary packets and how to install them. After carrying out all steps in the shown order the processes in the *Access Point 9* and *Starting up the System 10* Chapters can be executed without any issues.

Notice that the Access Point9 is necessary for hosting the own network, after activating the Access Point you might not be able to download packets and thus the Access Point must be disabled first and a connection to the internet must be established. This is described in *WAP is already active 8.4*

### 8.2 Raspberry Pi Settings

After connecting with the Raspberry Pi via SSH or another method of your choice, you have to first activate the serial Port to enable communication between Raspberry Pi and the LoRa-Module. Use

---

```
1 sudo raspi-config
```

---

to get access to the Raspberry Pi Settings. Go to Interfaces and deactivate SPI, I2C. After that choose the Serial Interface and deactivate the first Window that is shown, activate the second.

## 8.3 Install Programs and Packages

Update the packet manager.

```
1 sudo apt update
2 sudo apt upgrade
```

Install git. Follow all shown steps.

```
1 sudo apt install git-all
```

Get our files out of git. Move into the correct folder.

```
1 sudo git init
2 sudo git clone ↘
    https://git.uni-paderborn.de/klingler/set2022w.git
3 cd set2022w
```

Install python packet manager and necessary python packets.

```
1 sudo apt install pip
2 pip install pyserial
3 pip install protobuf==3.20.3
```

Install ZeroMQ

```
1 apt-get install libzmq3-dev
```

Install Node and Nodejs

```
1 sudo apt install -y curl
2 curl -sL https://deb.nodesource.com/setup_16.x | ↘
    sudo bash -
3 sudo apt install -y nodejs
```

Install NPM.

```
1 sudo apt install npm
```

Move into the Web folder to get all necessary packets for the Website. This takes a long time!

```
1 cd Web
2 npm install
```

Now all necessary programs and packets are installed.



---

## 8.4 WAP is already active

To Pull into your Git folder or install packages while the WAP is active, deactivate the DNS-Mask.

---

```
1 systemctl stop dnsmasq
```

---

Then connect the Pi with the internet and follow the steps shown in Install Programms and Packets 8.3. After all steps there are completed, reactivate the DNS-Mask.

---

```
1 systemctl start dnsmasq
```

---

---

## Chapter 9

# Access Point

---

Disclaimer:

The changes made in this guide will change the wlan0 interface, if you are using ssh on this interface you will disconnect during the process.

To not run into that problem, connect with ssh on the eth0 interface or execute the commands locally.

(X (not x) is a substitute for the PI number change before copying!)

This chapter is a step-by-step guide to configure a Raspberry Pi as an access point for the LoRa Network

In order to work as an access point, the Raspberry Pi will need to have access point software installed, along with DHCP server software to provide connecting devices with a network address.

To create an access point, we'll need DNSMasq and HostAPD. Install all the required software in one go with this command:

---

```
1 sudo apt install dnsmasq hostapd
```

---

Since the configuration files are not ready yet, turn the new software off as follows:

---

```
1 sudo systemctl stop dnsmasq
2 sudo systemctl stop hostapd
```

---

Configuring a static IP

We are configuring a standalone network to act as a server, so the Raspberry Pi needs to have a static IP address assigned to the wireless port. This documentation assumes that we are using the standard 192.168.x.x IP addresses for our wireless network, so we will assign the server the IP address 192.168.X.1. It is also assumed

that the wireless device being used is wlan0.

To configure the static IP address, edit the dhcpd configuration file with:

---

```
1 sudo nano /etc/dhcpd.conf
```

---

Go to the end of the file and edit it so that it looks like the following:

---

```
1 interface wlan0
2 static ip_address=192.168.X.1/24
3 nohook wpa_supplicant
```

---

Now restart the dhcpd daemon and set up the new wlan0 configuration:

---

```
1 sudo service dhcpd restart
```

---

Configuring the DHCP server (dnsmasq)

The DHCP service is provided by dnsmasq. By default, the configuration file contains a lot of information that is not needed, and it is easier to start from scratch. Rename this configuration file, and edit a new one:

---

```
1 sudo mv /etc/dnsmasq.conf /etc/dnsmasq.conf.orig
2 sudo nano /etc/dnsmasq.conf
```

---

Type or copy the following information into the dnsmasq configuration file and save it (X is a substitute for the PI number change before copying!):

---

```
1 interface=wlan0          # Use the require wireless ↘
    interface - usually wlan0
2 dhcp-range=192.168.X.2,192.168.X.255,255.255.255.0,15m
3 address=/#/192.168.X.1 # Redirect all domains ↘
    (the #) to the address 192.168.X.1 (the server ↘
    on the (Pi)
```

---

So for wlan0, we are going to provide IP addresses between 192.168.X.2 and 192.168.X.20, with a lease time of 15 minutes. If you are providing DHCP services for other network devices (e.g. eth0), you could add more sections with the appropriate interface header, with the range of addresses you intend to provide to that interface. We are also redirecting all domains to the ip address 192.168.X.1. Later on we're going to redirect all traffic destined for 192.168.X.1 to the IP and port where we're hosting the server - 192.168.X.1:8081.

There are many more options for dnsmasq; see the dnsmasq documentation for more details.

---

Start and reload dnsmasq to use the updated configuration:

```
1 sudo systemctl start dnsmasq
2 sudo systemctl reload dnsmasq
```

---

Configuring the access point host software (hostapd)

You need to edit the hostapd configuration file, located at `/etc/hostapd/hostapd.conf`, to add the various parameters for your wireless network. After initial install, this will be a new/empty file.

---

```
1 sudo nano /etc/hostapd/hostapd.conf
```

---

Add the information below to the configuration file. This configuration assumes we are using channel 7, with a network name of Pi WiFi, and no password. This means anyone can connect to the network. The original Raspberry Pi guide has configuration options for a passworded WiFi.

---

```
1 interface=wlan0
2 driver=nl80211
3 ssid=Pi WiFi X
4 channel=7
5 hw_mode=g
```

---

We now need to tell the system where to find this configuration file.

---

```
1 sudo nano /etc/default/hostapd
```

---

Find the line with `#DAEMON_CONF`, and replace it with this:

---

```
1 DAEMON_CONF="/etc/hostapd/hostapd.conf"
```

---

Start it up

Now enable and start hostapd:

---

```
1 sudo systemctl unmask hostapd
2 sudo systemctl enable hostapd
3 sudo systemctl start hostapd
```

---

Do a quick check of their status to ensure they are active and running:

---

```
1 sudo systemctl status hostapd
2 sudo systemctl status dnsmasq
```

---

Add routing and masquerade

Edit /etc/sysctl.conf

```
1 sudo nano /etc/sysctl.conf
```

and uncomment this line:

```
1 net.ipv4.ip_forward=1
```

Add a masquerade for outbound traffic on eth0:

```
1 sudo iptables -t nat -A POSTROUTING -o eth0 -j ↘  
MASQUERADE
```

(more information about masquerade in iptables <https://askubuntu.com/a/466451>)

Add redirect for all inbound http traffic for 192.168.X.1 (which we defined earlier in dnsmasq.conf) to our Node.js server on port 8081 (192.168.X.1:8081) (X substitute for PI number change before copying!).

```
1 sudo iptables -t nat -I PREROUTING -d 192.168.X.1 ↘  
-p tcp --dport 80 -j DNAT --to-destination ↘  
192.168.X.1:8081
```

Save the iptables rules.

```
1 sudo sh -c "iptables-save > /etc/iptables.ipv4.nat"
```

Edit /etc/rc.local

```
1 sudo nano /etc/rc.local
```

and add this just above "exit 0" to install the iptables rules on boot.

```
1 iptables-restore < /etc/iptables.ipv4.nat
```

Reboot and ensure it still functions.

Using a wireless device, search for networks. The network SSID you specified in the hostapd configuration should now be present, and it should be accessible (with the specified password).

If SSH is enabled on the Raspberry Pi access point, it should be possible to connect to it from another Linux box (or a system with SSH connectivity present) as follows, assuming the pi account is present:

---

```
1 ssh pi@192.168.X.1
```

---

By this point, the Raspberry Pi is acting as an access point, and other devices can associate with it. Associated devices can access the Raspberry Pi access point via its IP address for operations such as rsync, scp, or ssh.

[5]

---

## Chapter 10

# Starting up the System

---

This whole is a guide showing the steps that need to be taken to start up the System. For this guide to work the Installation of all required programs on the Raspi-nodes (chapter 8) must be completed.

start run.py (X is substitute for Pi ID)  
for the access nodes

---

```
1 python3 run.py -mgw X
```

---

for the broker node

---

```
1 python3 run.py -bro
```

---

As the access point should already be running, Users can now connect to the Wi-Fi of one of the access nodes and will automatically be directed to the chat website.

---

## Bibliography

---

- [1] Raspberry Pi Foundation, ed. *Raspberry Pi 3 Model B*. 2023. URL: <https://www.raspberrypi.com/products/raspberry-pi-3-model-b/> (visited on 03/18/2023).
- [2] IMST GmbH. *Channel Access Rules for SRDs*. 2012. URL: [https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Telekommunikation/Unternehmen\\_Institutionen/Koexistenzstudie\\_EN.pdf](https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/Telekommunikation/Unternehmen_Institutionen/Koexistenzstudie_EN.pdf) (visited on 03/18/2023).
- [3] Google, ed. *Google's Protocol Buffers Documentation*. 2023. URL: <https://protobuf.dev> (visited on 03/18/2023).
- [4] Google, ed. *Python module for Google's Protocol Buffers*. 2023. URL: <https://github.com/protocolbuffers/protobuf> (visited on 03/18/2023).
- [5] Tom Humphries. *Access Point Configuration*. 2019. URL: <https://github.com/TomHumphries/RaspberryPiHotspot> (visited on 03/21/2023).
- [6] SEMTECH. *What are LoRa® and LoRaWAN®?* URL: <https://lora-developers.semtech.com/documentation/tech-papers-and-guides/lora-and-lorawan/> (visited on 03/22/2023).
- [7] *Socket.IO - Bidirectional and low-latency communication for every platform*. 2023. URL: <https://github.com/socketio/socket.io> (visited on 03/18/2023).
- [8] *SX1262 868MHz LoRa HAT*. 2023. URL: [https://www.waveshare.com/wiki/SX1262\\_868M\\_LoRa\\_HAT](https://www.waveshare.com/wiki/SX1262_868M_LoRa_HAT) (visited on 03/15/2023).
- [9] *systemd - System and Service Manager*. 2023. URL: <https://github.com/systemd/systemd> (visited on 03/18/2023).
- [10] *The ZeroMQ project*. 2023. URL: <https://github.com/zeromq> (visited on 03/18/2023).
- [11] Waveshare, ed. *SX1262 868M LoRa HAT*. 2023. URL: [https://www.waveshare.com/wiki/SX1262\\_868M\\_LoRa\\_HAT](https://www.waveshare.com/wiki/SX1262_868M_LoRa_HAT) (visited on 03/18/2023).