

University of Zurich[UZH]

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

Design and Implementation of an IoT-based Hierarchical Payment System with BAZO Blockchain

Cepilov Ile
Cugnasco-Gerra, Ticino, Switzerland
Student ID: 13-929-740

Supervisor: Sina Rafati, Bruno Rodrigues
Date of Submission: April 11, 2019

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

ifi

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

Abstract

Bei der Blockchain Integration von IoT-Gerätedaten über zentrale Datenbanken besteht ein latentes Datenmanipulationsrisiko. Diese Arbeit befasst sich mit der Lösungssuche, wie die Daten bereits auf Stufe IoT signiert werden können und somit ohne Manipulationsrisiko indirekt in die Blockchain gelangen. Dafür wird in dieser Arbeit das LoRa Netzwerk Protokoll eingesetzt, da sich LoRa besonders für Low Power Consumption und Long Range auszeichnet. Da wir auch sehr auf Power Consumption achten, benutzen wir eine Blockchain die auf *Proof-of-Stake* basiert ist und im Vergleich zu *Proof-of-Work*-basierten Blockchains sowohl einen niedrigen Energieverbrauch als auch eine höhere Transaktionsgeschwindigkeit aufweist. Auf einer bereits bestehende Supply Chain Monitoring Platform sollen, dank signierenden IoT Geräten, Transparenz, Rückverfolgbarkeit und Authentizität in allen Schritten sichergestellt werden. Weil Transparenz ein Kernpunkt für uns ist, wenden wir keinen Verschlüsselungsalgorithmus an. Diese Arbeit zeigt auch auf, wie ein hierarchisches Zahlungssystem aufgebaut und eingeführt werden kann, um sicherzustellen, dass sämtliche Supply Chain Akteure mittels einer Blockchain fair entlöhnt werden können.

The blockchain integration of IoT data across central databases poses a latent data manipulation risk. This thesis deals with the solution finding, of how the data can be signed at the IoT level and thus indirectly reach the blockchain without any manipulation risk. For this purpose, LoRa Network Protocol is used in this work, as LoRa is especially distinguished for Low Power Consumption and Long Range. Also, because we pay close attention to power consumption, we use a blockchain based on *Proof-Of-Stake*, which has low power consumption and higher transaction speed compared to *Proof-Of-Work* based blockchains. On a already existing and implemented Supply Chain Monitoring Platform level, signing IoT devices ensure transparency, traceability and authenticity in all steps. Since transparency is a central point in this work, no encoding algorithms are applied. This thesis also shows how a hierarchical payment system can be set up and implemented to ensure that all the actors of a supply chain can be fairly compensated by means of a blockchain.

Acknowledgments

First of all, I would like to thank Sina Rafati for sharing his expertise with me and giving and motivating me throughout my work with his precious feedback.

Then, I am really greatful to Prof. Dr. Thomas Bocek for his competent and enthusiastic support in this thesis. His wide knowledge and passion in the topic immensely helped me.

I would like to thank Dr. Eryk Schiller for helping me in mastering and understanding the main concepts of the LoRa Protocol. He was even able to help me in running the LoRa Network for my thesis.

Lastly, I would like to express my gratitude to Prof. Dr. Burkhard Stiller for making possible to write my thesis with his group at the University of Zurich.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Description of Work 1

1.3 Thesis Outline 2

2 Related Work and Background **3**

2.1 Internet of Things, Supply Chain Monitoring and Blockchain 3

2.1.1 Blockchain-based IoT Systems 4

2.2 The Bazo Blockchain 5

2.3 Supply Chain Monitoring Platform 6

2.4 Background in Hardware Components 6

2.4.1 Arduino 6

2.4.2 Raspberry Pi 3 8

2.4.3 LoRa Shield 8

2.4.4 LoRa Gateway (TTN-GN-868) 9

3 Design and Implementation **11**

3.1 Design . . . 11

3.1.1 IoT Architecture . . . 11

3.1.2 IoT Transaction overview . . . 12

3.1.3 Supply Chain Platform Backend to Bazo . . . 13

3.1.4 Hierarchical Payment System . . . 14

3.2 Implementation . . . 19

3.2.1 Data Flow Overview . . . 19

3.2.2 Transmission Overview . . . 20

3.2.3 Send data from/to a Node . . . 22

3.2.4 Arduino Sketch for LoRa Nodes . . . 23

3.2.5 TTN-plugin . . . 26

3.2.6 Elliptic Curve Cryptography . . . 27

3.2.7 Bazo Blockchain . . . 29

3.2.8 IoT Transaction & Wallet instantiation . . . 31

3.2.9 IoT Transaction Implementation . . . 34

3.2.10 IoT Transaction Verification . . . 40

3.2.11 Supply Chain Back-end to Bazo . . . 44

3.2.12 Supply Chain Front-end adaptations . . . 48

4 Evaluation **49**

4.1 Transmission Approach 1 vs. Approach 2 . . . 49

4.2 Security Aspects . . . 50

4.2.1 IoT Wallet/Transaction . . . 50

4.2.2 TTN-Plugin & Acknowledgements . . . 51

4.2.3 Bazo Client REST API . . . 51

4.2.4 Cryptography with ED25519 and RSA . . . 52

4.2.5 Transaction Counter . . . 53

4.3 LoRa limiting factors 53
4.4 Tests & Comparisons 53
4.5 Real-world Test Scenario 55

5 Summary and Conclusions **57**

Bibliography **59**

Abbreviations **61**

Glossary **63**

List of Figures **63**

List of Tables **65**

Listings **67**

A Installation Guidelines **71**

A.1 Prerequisites 71
A.2 LoRa Node Application 71
A.3 TTN-plugin Application 72
A.4 Bazo Miner 73
A.5 Bazo Client 74
A.6 Supply Chain Monitoring Platform Back-end 75
A.7 Supply Chain Monitoring Platform Front-end 76

B Contents of the CD **79**

Chapter 1

Introduction

1.1 Motivation

Internet Of Things (IoT) devices continuously record data form the surrounding environment and send all their information to the Internet. A cloud provider is often used to store the huge amount of data and make it accessible, either publicly or privately. Since IoT devices are by definition not powerful at all, in terms of storage memory and processing power (a budget laptop from a couple of years ago is still more powerful than it), they are not able to run as full nodes.
Nowadays, there exists some valid alternatives such as *lightweight nodes*, but even those require too much processing power and because IoT devices can be even constrained in connection bandwidth, running them as lightweight nodes becomes almost impossible.

By being able to have an IoT-enabled blockchain, a very interesting use case can be the Supply Chain Monitoring. Introducing them in the latter, would mean that a new level of transparency, authenticity and even backtracking will be reached. Every step taken along the supply chain can be tracked and securely stored, making them even tamper-proof.

Once this two components are able to work together, a new Hierarchical Payment Systems can be designed and developed. This would mean that not only the producers or farmers will receive a higher portion of the product payment, instead of all the intermediaries, but even employed workers along the supply chain can be directly paid.

1.2 Description of Work

In this thesis, we will explain how IoT devices can be enabled so that they are able to sign and issue transactions to a blockchain (BC). This will mean that they are not going to be either full nodes or lightweight ones. But instead, they will run a simple hardware wallet and basically all the heavy and power consuming processing is outsourced to remote full nodes while the cryptographic keys never leave the devices and are securely stored in

them.
Having a central copy of the data sent by all the IoT devices would be much easier to access and manage. At the same time, a centralized entity will become an hot-point for attacks. It would expose the whole system to manipulation and thus, making less trustworthy all the applications and use cases in which IoT devices are being used.

Since we are going to use a BC which is not ready to accommodate the need of IoT devices and thus store their data as transactions, a big part of this thesis will be in adapting the Bazo BC to suit our use case. For this, we are going to first understand at most how we can exploit IoT devices based on Arduino which send data using the Long Range (LoRa) Wide Area Network (WAN). Once achieved this, we will have to create a sort of Lightweight Node which is in charge of letting communicating IoT devices with the BC and even manage them.

The next big step would be to adapt Bazo's cryptography methods to align with the same ones used on the IoT devices and then create a new type of transaction which perfectly adapts to the necessities of all different IoT devices.

Only at this point, we will start taking into consideration the already existing Supply Chain Monitoring Platform and try to integrate it with Bazo BC.
Here, we will have a look at the back-end and discover how to connect it to our BC, if that's possible. Moreover, we will gather the existing front-end of the platform and try to connect it as well to Bazo BC.
Since the current version of the platform runs on Ethereum and their Smart Contracts (SC), being able to connect it to a BC which does not support SC at the time of writing this thesis, might be very difficult or even impossible without trading some security aspects. Thus, we are going only to propose some design solutions of how the application should look like and express some thoughts of how the back-end can be changed to accommodate the new Hierarchical Payment System (HPS). Implementing the final version of the HPS will be part of another project once Bazo BC will officially support smart contracts.

1.3 Thesis Outline

In Chapter two an introduction to existing blockchain-based IoT system and similar approaches as the one in this thesis is given. The Supply Chain Platform and the Bazo BC are also going to be presented in this Chapter, as well as the hardware component utilized.
Chapter three contains two main parts, the first one shows the designs of all the main system components analyzed and developed during this thesis and the second one contains the most important details for the implementation of each system.
Chapter four compares the different approaches chosen and tries to evaluate them. Moreover, it explains why some design decisions had to be taken. Chapter five summarizes and concludes the report.

Chapter 2

Related Work and Background

This section gives an outlook of existing IoT approaches with the blockchain and use cases. Moreover, we will give a classification of existing hardware and communication system, which we can use for our work.

2.1 Internet of Things, Supply Chain Monitoring and Blockchain

A similar approach to this thesis for connecting IoT devices to a Blockchain has already been proposed in [1]. More precisely, the studied cases regarded Water and Air pollution. Since the communications method used in the transmission of collected data were not always trustworthy and data could easily be manipulated or corrupted, the concept of a peer to peer (P2P) network could be very useful in order to achieve the benefits of a decentralized network without the trust issues usually related to it[1]. IoT devices and systems in general, are quite efficient in gathering data and push it to the cloud for further access and usage. That approach is unluckily more centralized and thus difficult to scale and secure the data[1]. Therefore, the idea of pushing the data to a BC has been proposed with the benefits of high availability and trust.
The system works as follow: the data is gathered from a sensor and stored into a local database. In addition to it and in order to guarantee data integrity, the data is also stored onto a BC. Hence, the data is made public and secure through BC[1].

This approach works well but lacks of a fundamental thing, the trustworthiness of the oracle, which in our case is every sensor/node which gathers the data. In his thesis, Sanjiv does not take into consideration or he simply assumes that the sensor which takes the data and the whole process after it, until the data is actually stored on the blockchain, is never going to be manipulated or corrupted somehow.
Our goal is to improve this approach by signing each sampled and transmitted data with a private and public key. On the Blockchain, we store the data, the public key of the sensor which sent and collected it and the signature. In doing so, we are able to verify the integrity of the data at each step, not only after it has been put onto the Blockchain,

but even before during the whole process described later.
We achieve an additional layer of security which hasn't been taken into consideration yet. Moreover, we are not going to use a well known Blockchain such as Ethereum, like Sanjiv in his thesis did, but instead we are going to use Bazo, which is a Blockchain implemented recently in the programming language "GO" by the Communication System Group (CSG) at the University of Zurich and it is based on Proof-of-Stake.

2.1.1 Blockchain-based IoT Systems

In this section, we will provide some categories in which IoT applications are nowadays available. All of them are blockchain-based:

- **Supply Chain.** Some applications in this field aim at providing on-demand manufacturing [16], improve process efficiency, trace food, accurate and efficient record keeping and track sample between different providers[14]. Moreover, thanks not only to blockchains but even to smart contracts, it is possible to achieve a smart supply chain monitoring where every stage of goods can be tracked with their position[16].

- **Transportation.** In the last years a lot of IoT concepts have been proposed and designed for the next-generation of transportation systems, the most researched and interesting field for this category are smart vehicles[16].

- **Smart Energy.** One of the main proposed IoT system in this category aims at protecting users from malicious transaction where an user wants to sell or buy unreasonable amount of energy[16].

- **Smart Environment.** This category aims at ensuring availability and unrepudiability of sensed data collected in the wild, e.g. farms, fabrics, etc.[16]

- **Robotics.** In this field, a blockchain system should assure and support unmanned air vehicle (UAV)[16] so that they can coordinate between them their actions and movements.

Since scalability issue may become a problem when opening a blockchain to IoT devices, we will try to leverage them by relying on a central hub which handles a group of IoT devices, similarly like already done by O. Novo in [15]. This reduces the number of interactions between each single IoT devices and the blockchain, thus gaining potential to a scalable design. O. Novo managed to rely on a Smart Contract, what we actually cannot exploit yet on Bazo.

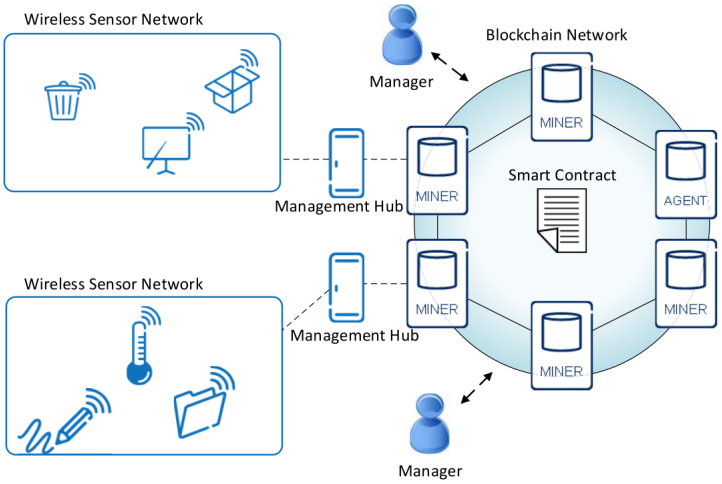


Figure 2.1: Decentralized Access Control System [15]

Our system completely differs in terms of communication layer from what done so far since we are going to use a network protocol like LoRa which implies even more challenges than Wifi. In order to communicate, our hub will need to do some more advanced operations like coordinating the nodes and aggregating their data until they are ready to issue a final transaction. It is not going to be just a simple forwarding to the clients.

2.2 The Bazo Blockchain

Bazo is a research project where its first goal was *"...to provide consumer bonus points based on blockchain technology for a financial service provider. The benefit of this solution in contrast to existing bonus systems would be that the financial service provider only took on the role as an exchange point of BAZO coins against a physical currency and not the role of the validator and book keeper of transactions themselves anymore. This would then allow businesses and customers to exchange goods more easily with these bonus coins than before because they could independently determine the terms of business with each other"*[9].
Over the time, Bazo blockchain has become a platform where different groups of students, coming from different universities, can work on it and experiment different approaches and architectures. At the time of writing this thesis, there are students trying to implement Smart Contracts, others are trying to improve the performance by applying some scaling designs, like sharding and other are trying to fix some small problems present in the current version.
One of the goal of this thesis is to make Bazo able to communicate and store data coming from IoT devices.
Important to mention is that Bazo was firstly designed to work with Proof-of-Work but it has been changed to meet the requirement for a *Proof-of-Stake* blockchain. This due to the significantly better power efficiency and security offered by this technology rather than

the dominant Proof-of-Work's Consensus Mechanism widely spread among well known blockchain, such as Bitcoin and Ethereum.

2.3 Supply Chain Monitoring Platform

While writing this thesis, other students are in charge of developing a Supply Chain Platform to allow transparency between the actors of the supply chain. The platform has multiple parts developed in different programming languages. The back-end is developed in Java using the Spring Boot framework. Then there are three front-ends, one is aimed to be used from Consumers of a product in the supply chain and it is actually a web application. The other two front-ends which are used by Producers of a product are written in Android and in React Native. Every action done to a product along the supply chain is saved into the ethereum blockchain. A smart contract is in charge of keeping track of it and update it. In doing so, every step that happens to a product is saved and can be retrieved.
As of now, the front-end lacks of a fundamental function. It cannot send any transaction to the ethereum Smart Contract, indeed all the transactions have to go through the back-end.
This means that the Food Chain platform is in charge of the costs for it and not the consumer. This is a design flaw that was done by one of the developer and hopefully it will be fixed and ready when we will have to connect it to Bazo.

Both front-ends applications are not ready and not every function is implemented and thus, implementing the Hierarchical Payment System can turn out to be more challenging than thought.

2.4 Background in Hardware Components

This section shows which hardware components are already existing on the market that we can utilize in order to run our system.
In this work, the main components we are going to use are Arduino and Raspberry Pi's. In order to be able to transmit data over the LoRa network, we need some LoRa Gateways (TTN-GN-868) and LoRa shields.

2.4.1 Arduino

Sanjiv's work was based on Arduino Uno but since we have to sign the data we need also to create and store in memory a private key, a public key and the signature. Unluckily the Arduino Uno does not have enough space to keep in memory all of them including the sketch, which is the code on which the arduino runs.

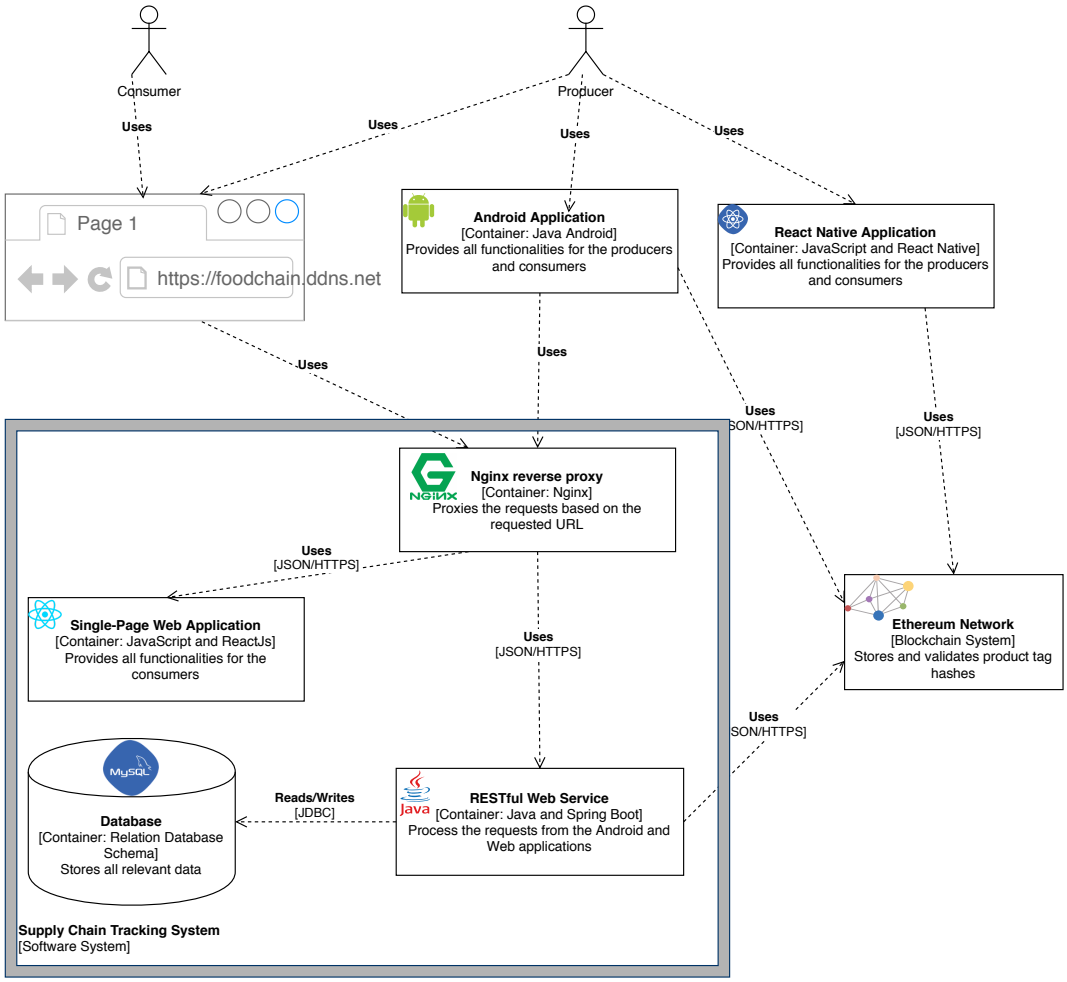


Figure 2.2: Container Diagram for Food Chain Tracking System from [11]

Therefore, we had to find a more suitable solution. We opted to use an Arduino Mega 2560 which has way more space to keep all of the variables we need in memory. The following table indicates quantitatively the difference between the two versions:

	Arduino Uno	Arduino Mega 256
Clock speed	16MHz	16MHz
Flash Memory (kB)	32	256
EEPROM (kB)	1	4
SRAM (kB)	2	8

Table 2.1: Arduino Uno vs Mega 2560

The Arduino Mega 2560 is a micro-controller board based on the ATmega2560[4]. It has several digital/analog input/output pins which can be used to connect sensor to the board (e.g. DHT11, DHT22) and it can be programmed in C++ using the Arduino IDE. It communicates thanks to an USB interface and it is compatible with most shields designed for Arduino Uno and other boards.

2.4.2 Raspberry Pi 3

A Raspberry Pi 3 is a low-cost basic computer which does not have any built-in memory but one can install a flash memory card up to 64GB and more. Positive fact is that one can install multiple system on different flash memory cards and switch between them just by replacing the SD card[1]. Moreover, it is a very light system and requires less power supply. In this project, it is going to be used as a client which constantly listens to packets sent from the nodes and sends back acknowledgements. In the meantime, it also pushes the data with the signature and the public key to the Bazo Blockchain.

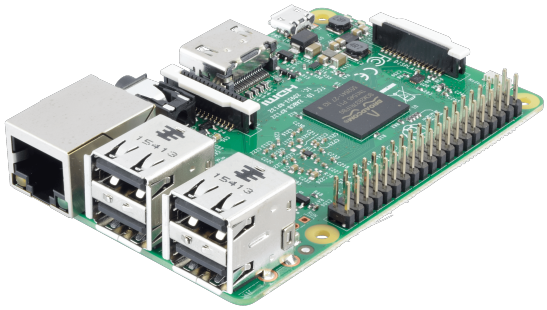


Figure 2.3: Raspberry Pi 3

2.4.3 LoRa Shield

LoRa Shield is a long range transceiver compatible with Arduino shield form factor and based on Open source library. The Shield allows the user to send data and reach extremely

long ranges at low data-rates. It provides ultra-long range spread spectrum communication and high interference immunity whilst minimising current consumption.

LoRa Shield is based on Semtech SX1276/SX1278 chip, it targets professional wireless sensor network applications such as irrigation systems, smart metering, smart cities, smartphone detection, building automation, and so on[5].

2.4.4 LoRa Gateway (TTN-GN-868)

In order to be able to utilize LoRa Nodes, there must be some LoRa gateways that are going to receive packets sent from nodes and in our case forward them to The Things Network (TTN). TTN allows the communication without owning a gateways and only on relying on other's one but since we want to make sure that our packets are going to be received, we utilize the official versions of the LoRa Gateway for Europe, i.e. TTN-GN-868.

Chapter 3

Design and Implementation

3.1 Design

This section explains different design decisions. Firstly, we will explain how IoT devices communicate to the blockchain and we will also show a specific overview of how an IoT transaction can be issued from a LoRa Node.
An explanation which shows how the Supply Chain Platform is connected to the Bazo Blockchain will also be included. Lastly, mock ups of the next version of the front-end are going to be shown as well.

3.1.1 IoT Architecture

Figure 3.1 shows an agnostic overview of the IoT Architecture. This system is designed to be compatible across most common system without requiring any special adaptations. Therefore, all the layers of communication and adaptations are not specified.
As we can see, we have an IoT Device used in a stage of the Supply Chain, that has a Blockchain Wallet which allows him to issue transactions to the blockchain. Each device is supposed to utilize a different layer of connection based on the use case. In some cases it might make sens to utilize LoRa as a transfer protocol, whereas in other contexts, where maybe some real time data is needed, it might be required to utilize something more powerful in terms of transfer rate and thus LTE-m or wifi might best suit it.
On the other end, before reaching the blockchain and thus the miners, there is an Edge Node.
An Edge Node acts as a hub and it is responsible for coordinating all IoT devices across the network before being able to forward their transactions.
They can be considered as lightweight nodes in our system, even though they do not store the blockchain information as miners nodes do, but they only verify them.
In some cases, e.g. when we use the LoRa Protocol, the Edge Node has to reconstruct the missing pieces of the transaction before sending it to a blockchain client, otherwise it won't be valid. This process will be explained later in this thesis.
The Edge Node can also act as a gateway, in our specific case as a LoRa gateway but it

can also be adapted to run as a wireless access point and thus listen to IoT devices that utilize wireless as a network protocol. Moreover, it can run even a blockchain client (Bazo client) and therefore avoid the necessity to utilize multiple different devices.
The client should own an account on the blockchain that can act as a sort of Smart Contract address, where all the nodes in the blockchain network need to know that address in order to be able to issue transactions on the network. Since Bazo does not support Smart Contracts yet, we had to find a valid work around.
In our case, everything (LoRa Gateway, Network Adapter, Bazo client) runs on a Raspberry Pi 3.
Once the transaction is ready to be sent it will be forwarded to the client which in turn will broadcast it to the miner of the blockchain.

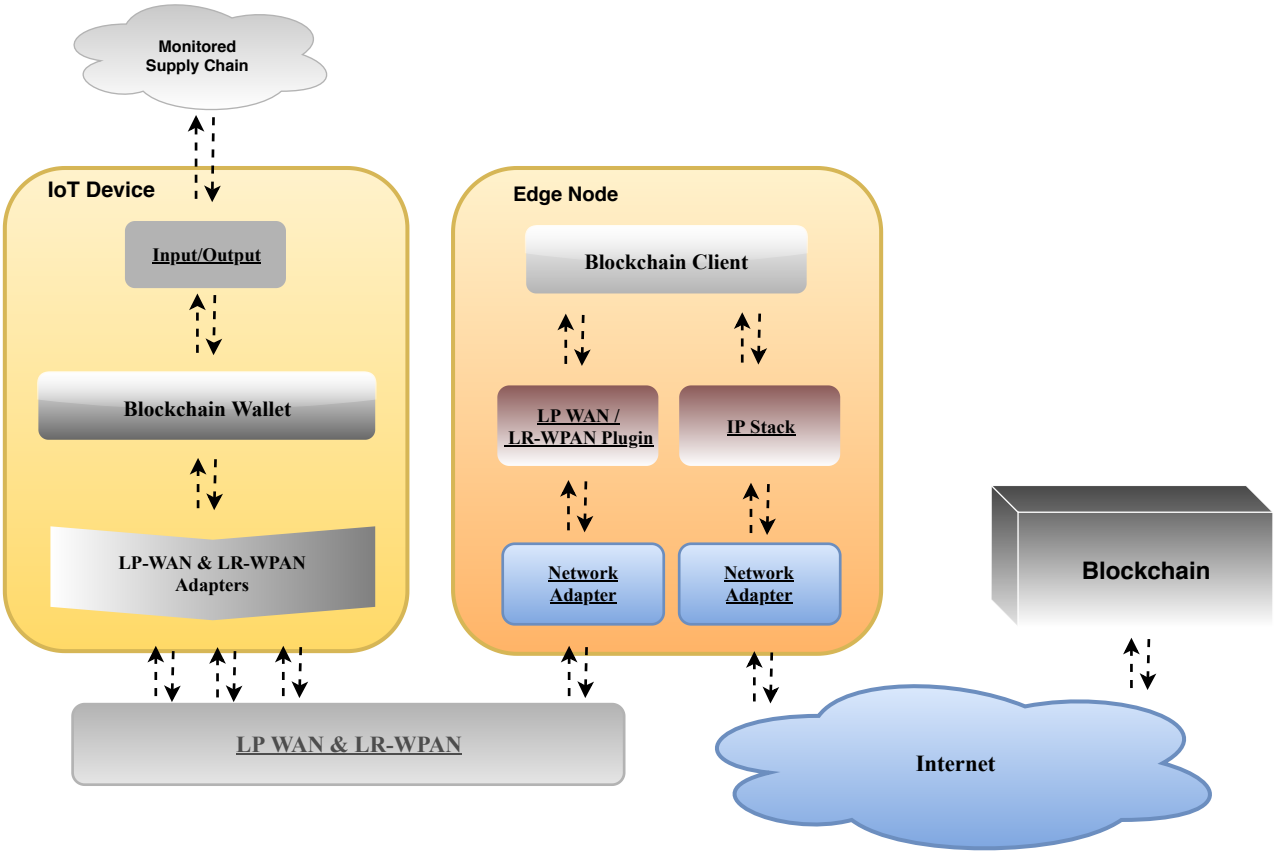


Figure 3.1: Agnostic IoT architecture

3.1.2 IoT Transaction overview

Now, we will have a look at which part of an IoT Transaction are sent from a device and how we recreate the whole transaction without having the need of sending every single piece.
On the Node we have hard-coded the receiver's account, since it is going to stay always the same. The issuers' address is the node's public key and the other fields, like the header, transaction fee and the counter are staying the same as well. Thus, only the data

(payload) and the signature are required to be sent. The public key is sent only once at the beginning and therefore we do not count it as something that has to be continuously transmitted. The Bazo light client (i.e. the TTN-plugin) is aware of the receiver's address and since the other parameters are static, it is capable of reconstructing the same structure for the transaction.
Figure 3.2 clearly shows which data is actually sent. As we can see, some fields are not leaving the LoRa Node and despite this, during the aggregation, the TTN-plugin can reassemble the IoT transaction and send it to the Bazo blockchain. The latter offers a REST API which we can use to forward our transactions.
The whole procedure is going to be deeper explained in the next chapters.

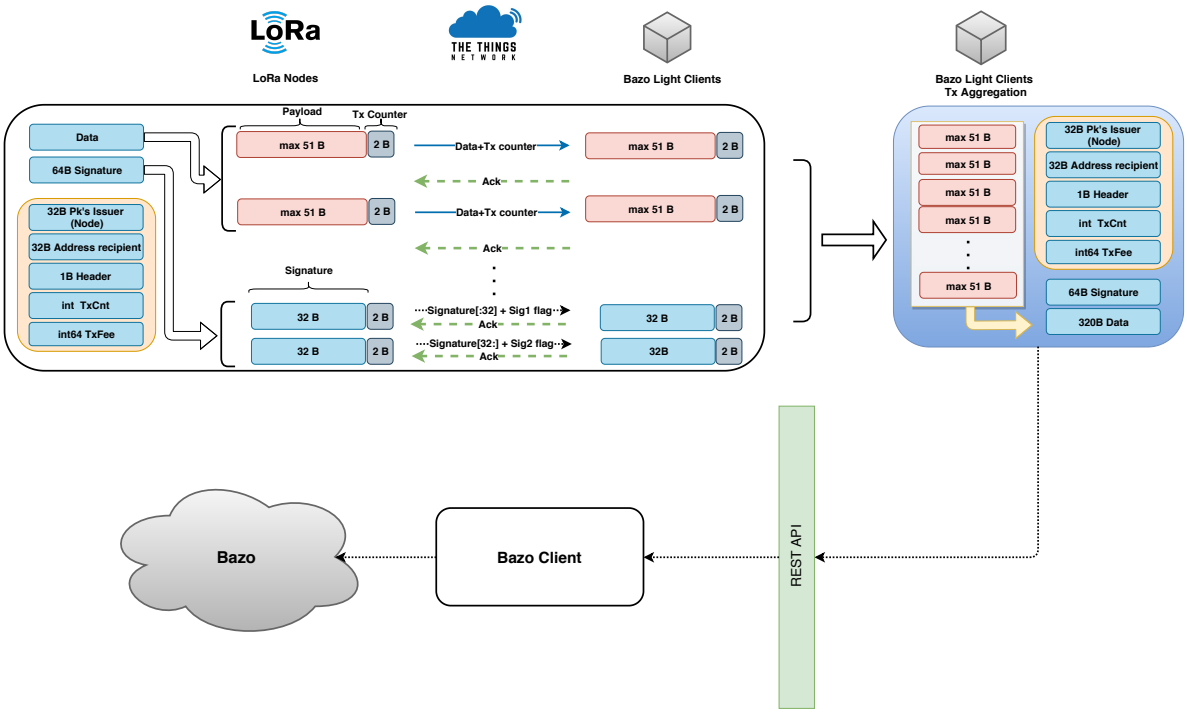


Figure 3.2: LoRa HW Wallet for a full IoT Transaction

3.1.3 Supply Chain Platform Backend to Bazo

In this section we will analyze how the whole Supply Chain Platform should look like once we connect it to Bazo.
First of all, the Android front-end gets out of the game since it is outdated. The React Native App is supposed to have all the functions as the android one but with the benefit of being compatible with iOS devices as well.
Since Bazo hasn't implemented Smart Contracts yet, we are only able to store the hash of the products but we are not able to retrieve them or check whether they have been already stored. Therefore, we rely on the database of the back-end at the moment. The front-end is so far not able to communicate directly to Bazo but it has to pass through the back-end. This implies that when a product tag has to be stored, it does not go directly

to the blockchain but it gets forwarded to the back-end where, once it is stored in the database, it gets pushed to the blockchain.

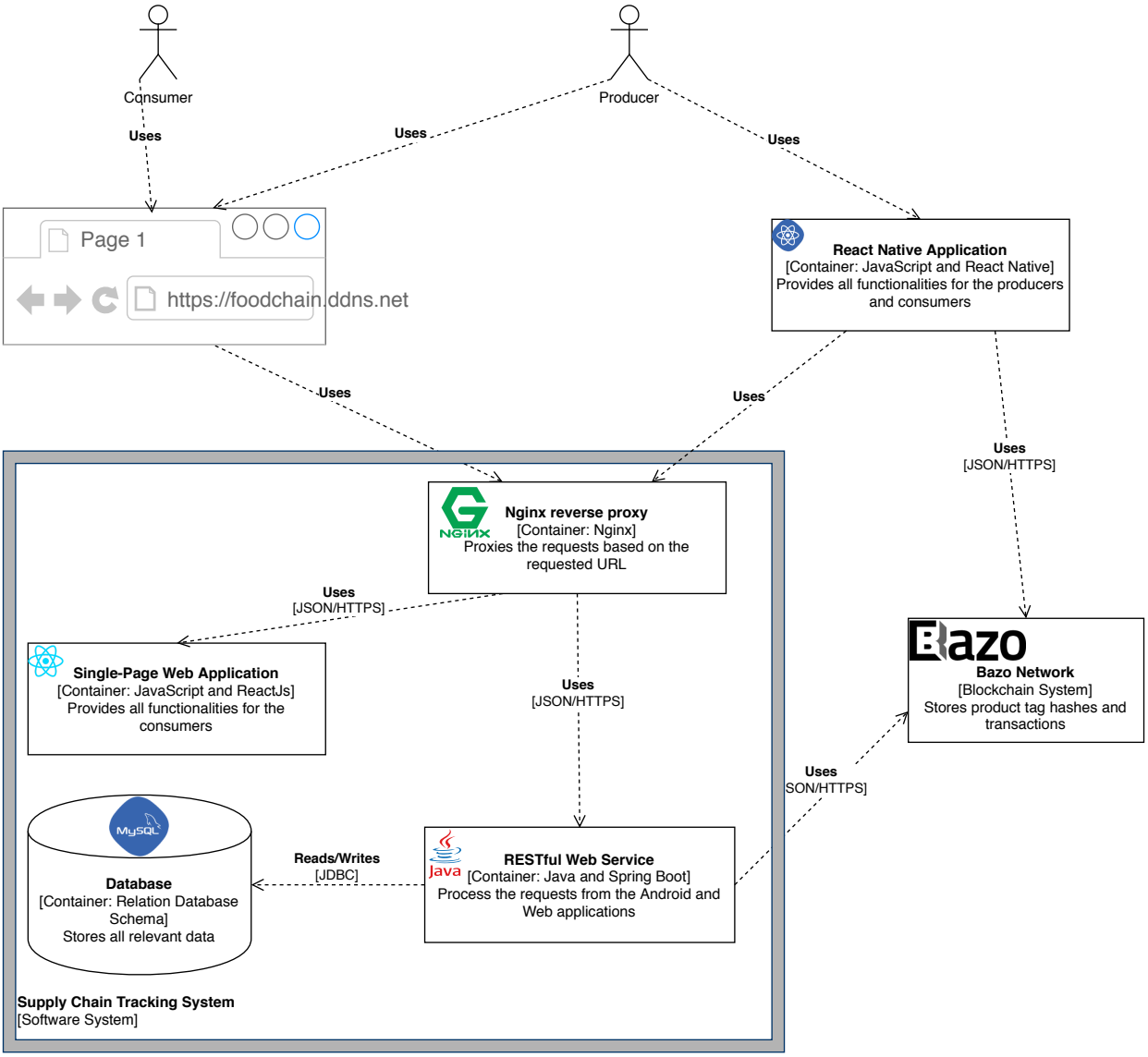


Figure 3.3: Adapted Container Diagram for Food Chain Tracking System connected to Bazo from [11]

3.1.4 Hierarchical Payment System

In this section, we are going to first introduce what a hierarchical payment system (HPS) is and then we are going to analyze which changes are required to the existing Supply Chain Monitoring Platform in order to be able to process hierarchical payments.

Definition of HPS

With the HPS, purchasing a product will initiate a set of transactions from a buyer to potentially all the actors in the production process of that particular product. HPS enables end-users to pay a specified amount to each actor (i.e., companies, farms, producers, factories). With HPS, not only the producers or farmer will receive the higher portion of the product fee in return of the product - instead of the intermediary dealers - but also, the workers of those farms and companies can be paid directly. HPS is a novel method to create transparency in payment systems by enabling direct payments between the end-users and actors in the supply chain systems

Supply Chain Back-end Adaptations

The existing version of the Supply Chain Tracking system is not designed to support any means of payments. Not only because it does not have any kind of price, but also because it does not have any kind of product saved as entity. Therefore, buying something as of now it's impossible.

In diagram 3.4 are represented in blue the existing entities in the database, in red the ones that should be added in order to allow payments.
First of all, the entity `Producer` has to switch it's account from an Ethereum address to a Bazo address. Moreover, since we are going to retribute workers, the latters have to be employed to a producer. Therefore we have an array of `workers_id` which contains all the worker's ids of a Producer. In order to assure that workers are really employed and work for the corresponding Producer, we can assume that we have a Know Your Customer (KYC) platform which manages all of this. An idea which might help as well, would be to implement a Smart Contract (SC) to keep track of it.

The new entity `Worker` has to be modelled as well. It should have a unique id (`worker_id`), a unique `producer_id`, a Bazo account (`bazo_account`) and the other attributes like a `Producer` has.
Since the purpose of the final system will be to sell products, we have to model them as well. As we can see, we have a new entity `Product`. This entity has a unique `product_id` which helps in identifying every single product, it has a `producer_id` which actually indicates who the producer is, it has a `price` and it has a `product_tag`. The product tag can be overwritten and should match the last created product tag for that product. Because the entity `Product_Tag` has a cyclic relationship with itself, the precedent tags can be retrieved as well. Thus, all the actions are going to be stored.
Each product can have a specific share (`product_share`), this indicates how much in percentage of the price the owner (producer) of the product will get. The remaining percentage, 1-`product_share`, is going to be divided by all the workers which performed an action in order to create that product and given to them as a retribution.

Since workers can also fulfill actions and not only producers, there is a table to match an action to a worker (`worker_action`). This should integrate pretty straightforward to the existing architecture.

One could think that we have to model customers as well, but since the BC is going to handle the actual payment, the back-end should not have to integrate anything. We could just simply store somewhere that a product has been purchased so that the back-end is aware of it.

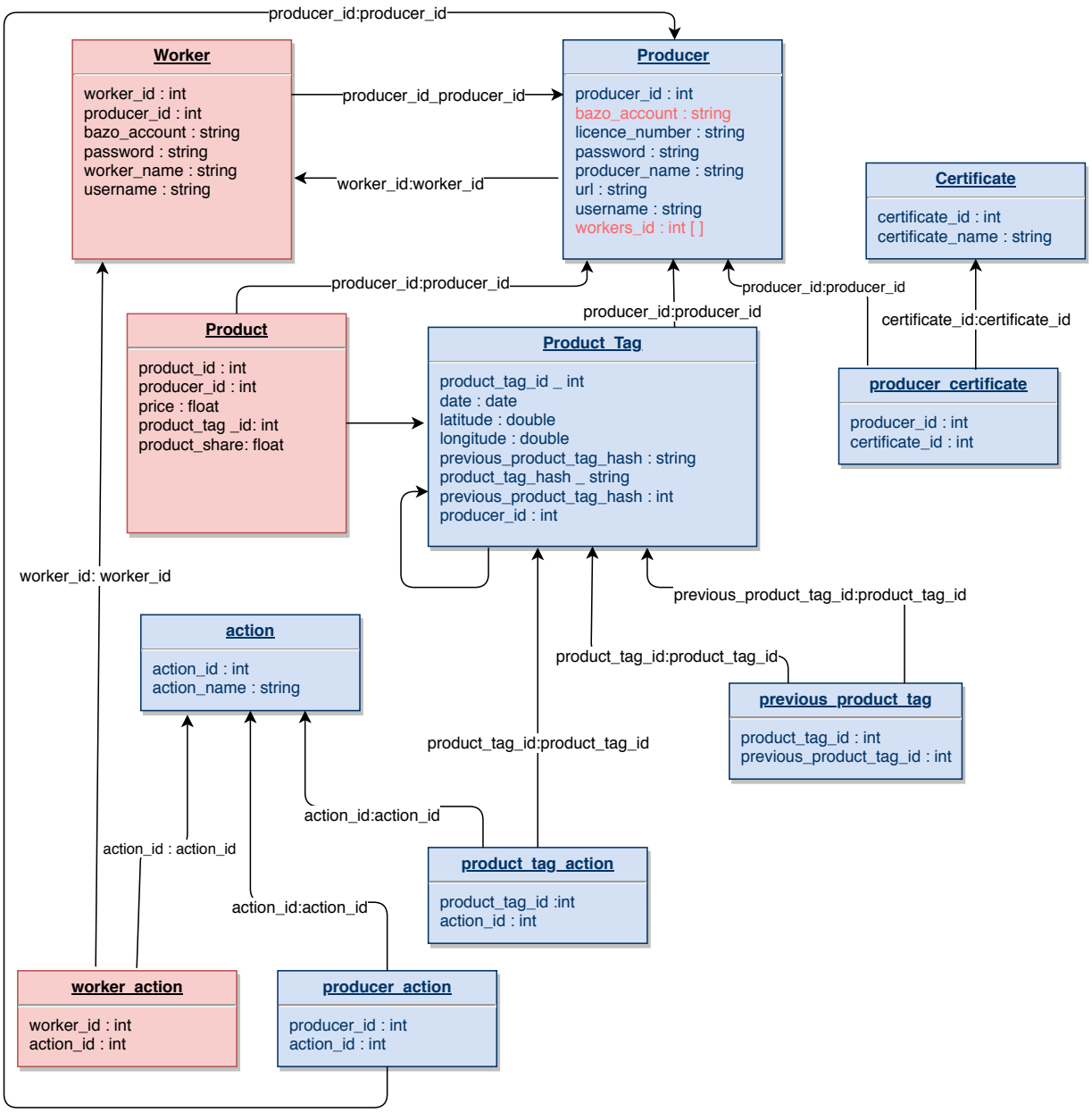


Figure 3.4: Adapted ER Diagram for Food Chain Tracking System from [11]

Supply Chain Front-end Adaptations

The front-end needs visually just a few adjustments. Once the back-end implements all the necessary entities with the right endpoints the results may be the following.

In figure 3.1.4 we see the homescreen that contains a new button, i.e. **Worker**. This should allow a worker to login and save his actions.
When a consumer uses the applications, he/she will get redirected to the **Scanned Products** tab in the `Consumer Mode`. Here, all the scanned products are available and if required, new products can be scanned thanks to the camera of the phone and the QR code.

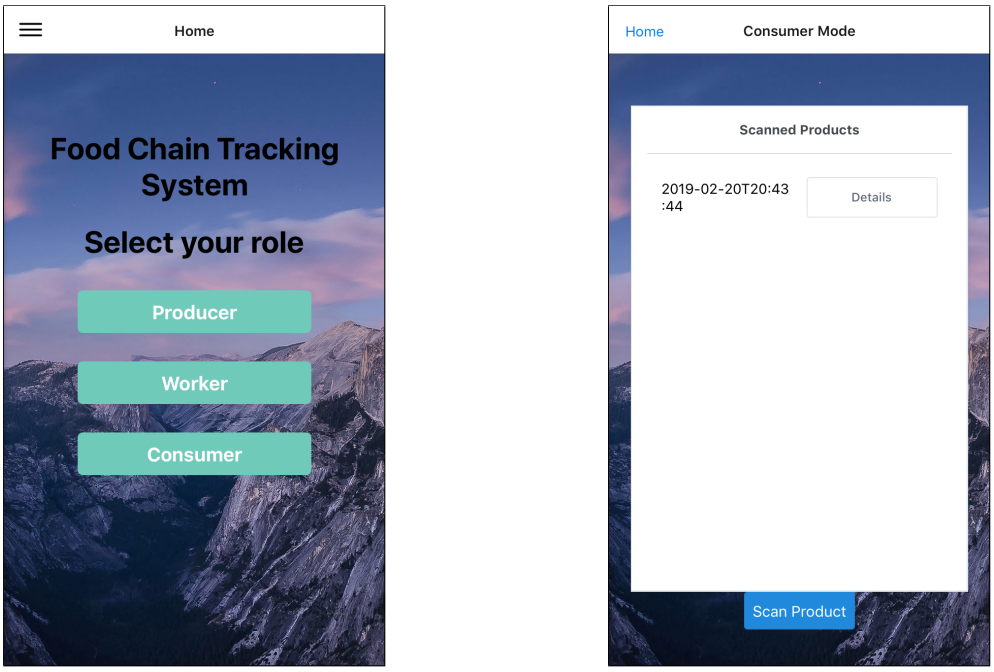


Figure 3.5: Home Screen & Scanned Products

When the button *Details* gets pressed, a map is displayed, on which the product is placed based on his coordinates. There, a dialog which shows all the details regarding the product is present.
The information about the price, the product share and how much the product owner gets (*Product Owner Share*) are being shown.
If we scroll down, we get the list of Workers. There we can see how many workers have taken part in it, what their current Bazo address is and how much of the remaining share they should receive.
At the end of the dialog a button to purchase the current product is placed (*Buy Product*).

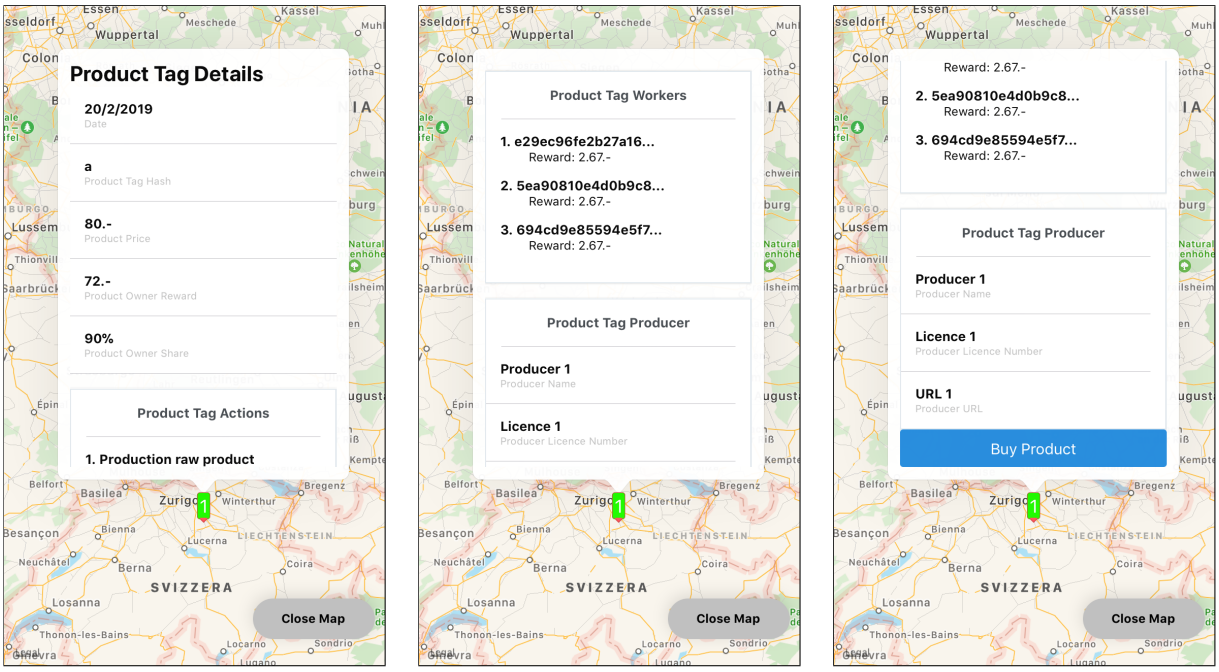


Figure 3.6: Map Overview & Product Details

When a customer wants to buy a product and confirms the payment, a popup gets displayed and the progress of the transactions is shown.

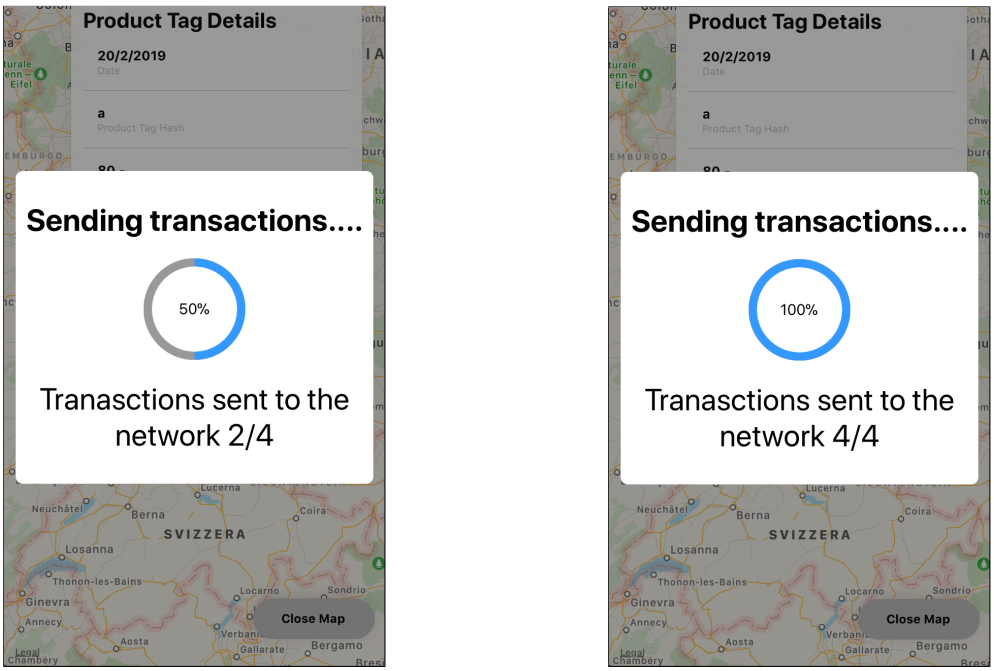


Figure 3.7: Buying a Product with HPS Transactions

3.2 Implementation

So far, we have only explained the design, either of the architecture or of the implementation.
In this section, every detail of the implementation for each component is going to be explained. We will start from the details regarding the LoRa nodes up to the changes in the blockchain. At the end, we will also discuss the required changes for the Supply Chain Monitoring Platform, in the back-end as well as in the front-end.

3.2.1 Data Flow Overview

The data flow is going to be presented and explained in every stage. This should help in better understanding each component of the system and their role in it since the platform used for this architecture is no longer agnostic but defined.

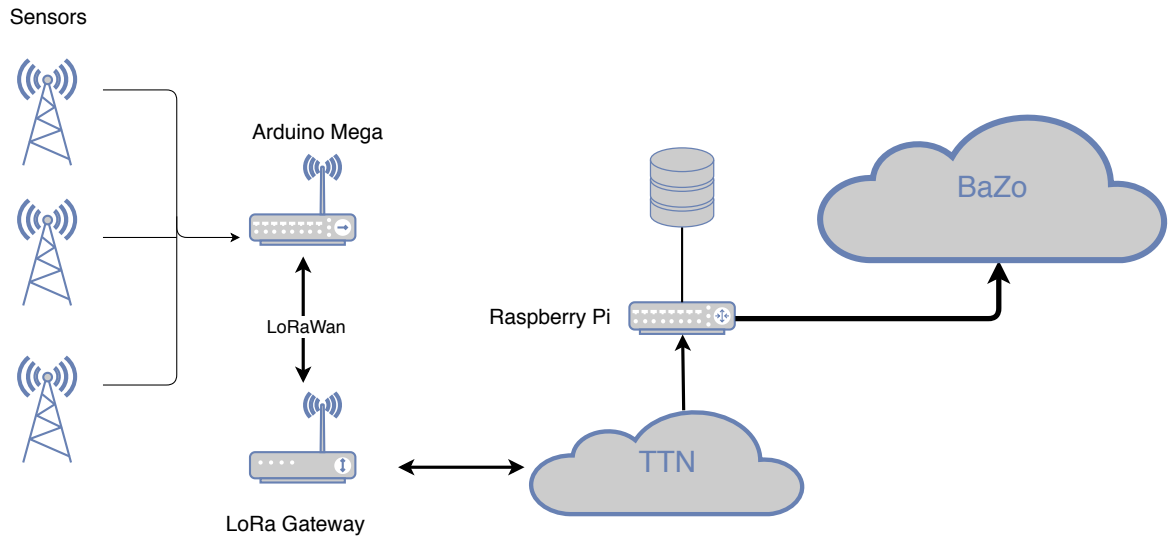


Figure 3.8: Data Flow Design with LoRa Nodes and Network

In order to collect data, IoT sensors (**Sensors**) are connected directly to the arduino (**Arduino Mega**). The connection between the two is wired and does not need any special protocol. The arduino retrieves directly data from the sensor and acts as a LoRa Node. Thus, each node is able to use the LoRaWAN protocol thanks to the LoRa Shield attached to the arduino.
The data is then signed with a private key along with other fields required for a full IoT Transaction and sent. Since we are using the LoRaWAN protocol, the data is also encrypted even though we do not really need this feature.
The nearest LoRa gateway receives the data and forwards it to a network service called The Things Network (TTN), which is responsible for decrypting the data and makes it available for downloading or forwarding. A RPi is constantly listening on the TTN

platform for messages sent from each node. It is responsible for collecting the data and communicating to nodes that their messages have been successfully received by sending an Acknowledgement. For security reasons, the RPi has to send the acknowledgement and not the gateways, this will be better described in chapter 4. When the RPi receives the whole data sent from a node, it verifies the integrity thanks to the signature and the public key of each node. Since it is aware of the whole IoT transaction without having the need to gather it from the node, it can reconstruct it and verify it. Once the data is verified, it can be forwarded to the Bazo Blockchain.
Moreover, the Raspberry Pi should save a touple of deviceId and public key (devId, pubKey) so that we know that our nodes are not beeing manipulated, since the public key of a node has to be always the same and we don't seek anonymity of the devices. The current status of a node should be saved as well so that when the system shuts down it knows the state of each node without having the need of asking the network about the current status. To do so, a local database could be implemented. This step is omitted since the architecture could change when Smart Contracts or other technologies such as WiFi/Bluetooth/LTE-m are going to be introduced.

3.2.2 Transmission Overview

Approach 1: ***data is signed once at the beginning and the signature is fragmented and sent with additional data***

The data transmission between a LoRa node and the listener is shown in Figure 3.9.
When a node wakes up, the first things it does is sending its public key. The size of the public key is 32 bytes and thus, it does not have to be fragmented. Once the TTN-plugin (on the RPi) gets it, it sends an acknowledgement to the node which will then proceed to the next step. Now, the collected data is signed using the private key of the node. Together with the data, some other parameters are sent, i.e. *counter* which indicates in how many parts the signature is going to be split and corresponds to the number of packets that have to be sent in order to get the whole signature on the TTN-client. *Signature = 0* indicates the signed data is sent. At this point, the TTN-listener is aware of the public key of the node and the signed data. The next step is to send the signature into more packets. Instead of wasting collection poinst while the signature has to be sent, not signed data with a piece of the signature is sent. The listener sends acknowledgement messages for each packet received so that the node knows which packet has to be sent and not a single one is missing. Important to keep in mind with this approach is that this data is not signed.
When the complete signature has been received, a last packet from the node is sent. This indicates to the listener that it should verify the signed data with the signature and the public key. Moreover, the listener knows as well that the whole process is going to start again (except for the public key which is sent only once).
This method has some problems and drawbacks that are going to be evaluated in the corresponding chapter.

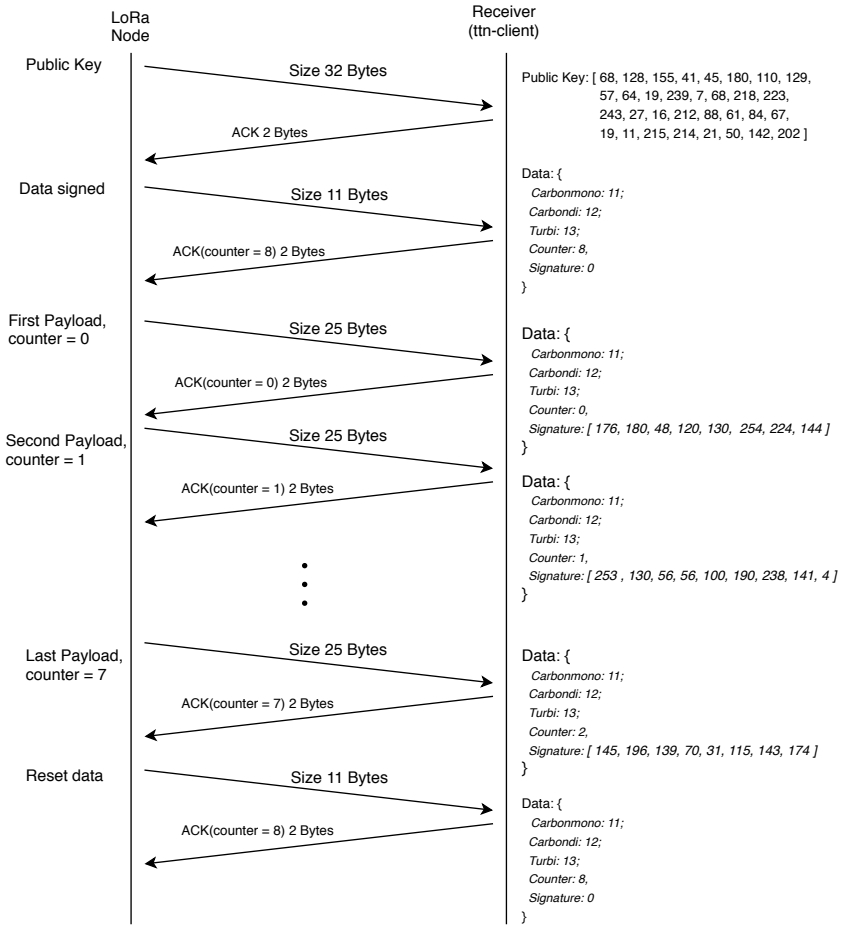


Figure 3.9: Data transfer diagram with overhead

Approach 2: data is sent first and signed at the end thanks to aggregation

The second approach is slightly different from the first one. In the same interval of time, the same amount of collected data can be sent but the whole process is different.
The public key is sent once at the beginning like before and after a successful transmission, data packets are sent. Important to point out is that those data are not signed yet. The client (receiver) sends as usual acknowledgements in order to keep the LoRa node informed on which packets are gathered so far. As of now, acknowledgements are counters that indicates which packets has arrived. However, a better approach has been suggested in order to keep the same state on both sides. It should consists of using an hash algorithm, like *sha1*, so that each time a node sends some data, the receiver sends a downlink message with the hash corresponding to the packet just received. In doing so, the node, which hashes the data as well, knows whether the data was correctly sent without any changes and it can send the next packet containing different data or resend the same packet if it was corrupted. This suggestion is with LoRaWAN not feasible, since we are not allowed to send many downlink messages and the size of them should be as small as possible.
Once enough data has been sent, the node can put together the whole data that it has been just sent and sign it along with the other variables which are present in an IoT

transaction. After that it can start sending the signature, either in multiple blocks or entirely. The receiver (TTN-plugin), which is aware of the packets, and thus the data, including the transaction variables, can put them together too and verify it with the signature and the public key from the node.

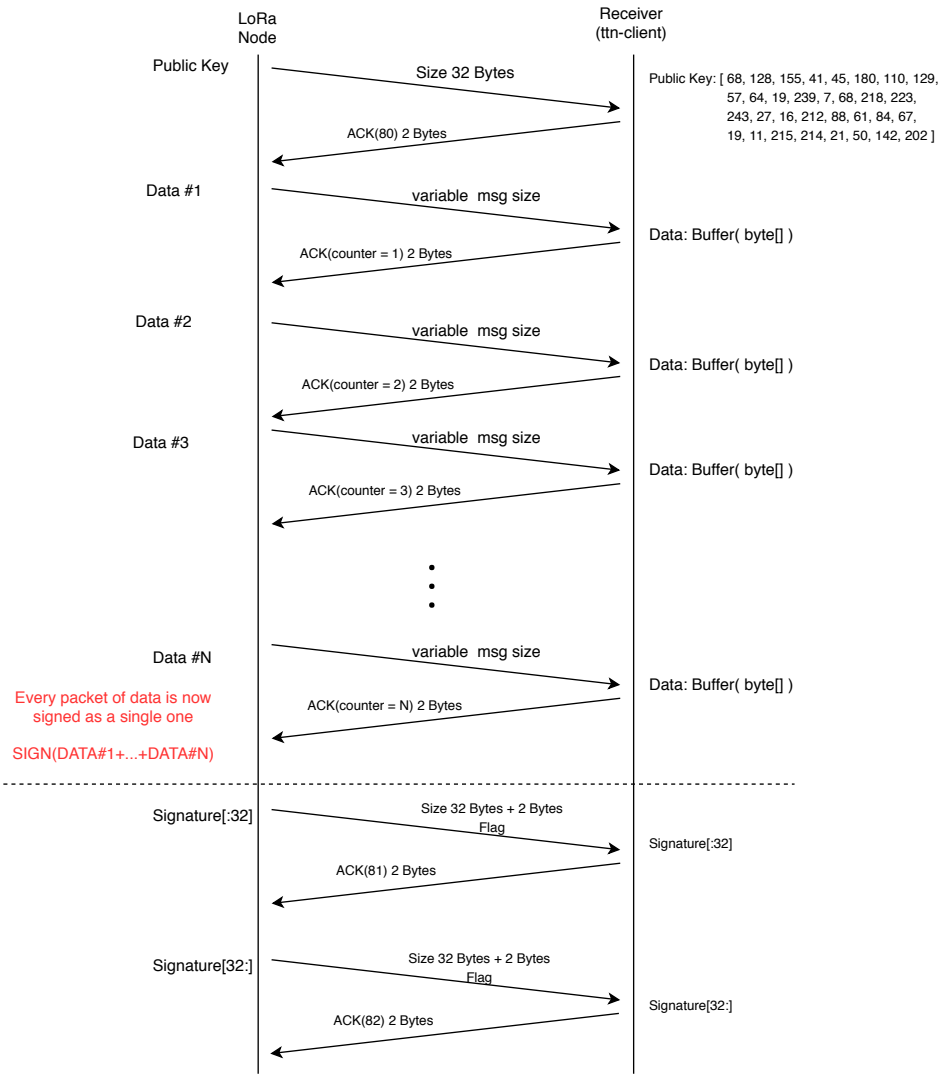


Figure 3.10: Data transfer diagram with aggregation

3.2.3 Send data from/to a Node

LoRaWAN is not really suitable for every use-cases and therefore it is important to really understand its limitations. Beside the limitations due to the duty cycle and therefore the response time from applications based on it, one of the main limit comes from the low bandwidth.
If we put together those two constraints, it means that each LoRa node should not send data for too much time and thus **payloads** sent must be shrunk as much as possible. As a consequence, LoRa nodes should never send JSON or plain (ASCII) text, but instead

encode the data as binary data. One way of achieving it is to utilize the *Cayenne Low Power Payload* format which is fully supported by TTN [7] but so far this process has been done manually.

The **interval** between messages should be kept in the range of several minutes in order to follow the policies, this implies that a node cannot send real time data and each sent packet should come from a smart design decision.

Since each LoRa node has to know in which state the client is, the latter sends downlink messages (acknowledgements) for each received packet. But when taking into consideration the limitations for downlink messages, the situation gets even worse.
A gateway is not able to receive any uplink packet from a Node while it is transmitting. Consequently, if a gateway is transmitting 10% of the time, it is not able to receive anything for that 10% of time. This would mean that devices which use that gateway to communicate are not able to send in that time interval and as a consequence their packets might get lost.
The best design decision would be to completely avoid **downlink** messages but since they are needed, their payload is kept as small as possible by sending only an integer converted into bytes.

3.2.4 Arduino Sketch for LoRa Nodes

In this section we will take a look at the most important parts of the code which runs on an arduino. The code allows not only to send an IoT transaction but also to transmit data using the LoRa Shield and thus through the LoRa network.
First of all, we will start with the configuration needed to utilize LoRa and TTN.

```
1 //Configurations for TTN
2 static const u1_t NWKSKEY[16] = { 0xB2, 0x7C, 0xD4, 0x1E, 0xE9, 0x40, 0x90, 0x0D, 0xF2, 0
      xC4, 0xAF, 0x47, 0x8D, 0xAB, 0x70, 0x73 };
3 static const u1_t APPSKEY[16] = { 0xC1, 0xEE, 0x14, 0x8C, 0xC8, 0x57, 0x17, 0xDF, 0xEA, 0
      xEC, 0x1F, 0xE9, 0xDA, 0x76, 0x5A, 0x1B };
4 static const u4_t DEVADDR = 0x26011382;
```

Listing 3.1: Configuration for a LoRa Node

This parameters corresponds to the one present on the TTN console when configuring a new application and adding a new device.

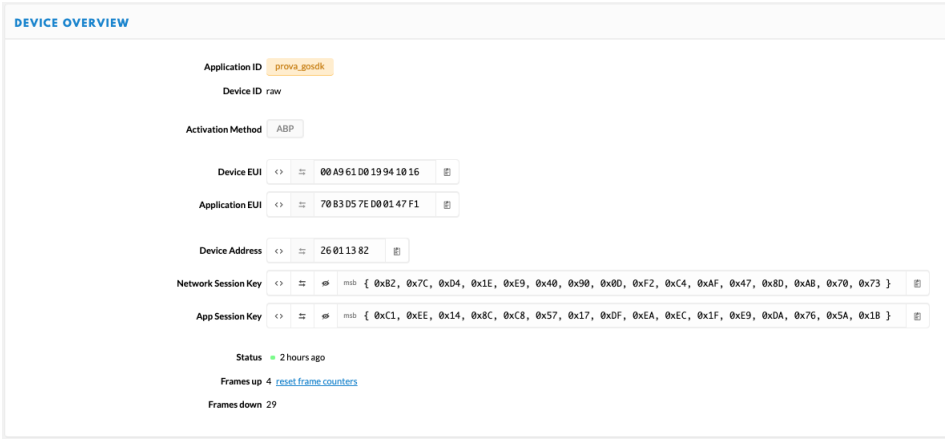


Figure 3.11: TTN Console

This values have to be directly copied from the TTN console once an application is created and a device is added. Important to mention is that those values have to be copied in the same exact format, which means, before copying them they have to be transformed in msb and not into lsb. The TTN's console provides already this option.

Now, we will explain the most important variables that are going to dictate how a LoRa node behaves in terms of collecting data and sending it.
`CounterValues` indicates how many collection data points are going to be signed, i.e are going to be included in an IoT transaction. `CounterValuesToSend` indicates how many collection data points are going to be transmitted each time and thus they will have to be aggregated on the other side from the TTN-plugin. `CounterPayloads` is just a variable used to keep track of the currently sent packet and therefore has to start at zero.
`Packets` represents how many packets (each containing the number of `counterValuesToSend`) are going to be sent in total for a complete IoT Transaction. `MaxByteToSend` indicates how big a payload should be in bytes. Since we are using the LMIC library[10] we are allowed to send at most around 51 Bytes.
We actually found a method to send more bytes than what the library allows so far and it consists in modifying the file *lmic.c* at line 217:

```
CONST_TABLE(u1_t, maxFrameLens) [] = { 256,256,256,256 };
```

important here is to chose a value which is a magnitude of 8 bits, like 64/128/256.
Moreover, the class `lorabase.h` also requires a small change at line 40 like this

```
enum { MAX_LEN_FRAME = 256 }
```

here the same rule as before applies and both values should match.
`Data[]` is an array which will contain all the collections data points that are going to be signed for a single IoT transaction and the array `toSend[]` will contain only the data sent each time and therefore will be overwritten after every successful transmission. Important to notice here is the `+2` in the size of the array, this is required because each time we send that array, we have to add the corresponding counter to keep track of the

status on the TTN-plugin. In order to be sure that we receive the right acknowledgement from the TTN-plugin, we keep track of what we should get on the node in the variable `expectedAck`.

```
//VARIABLES
uint16_t counterValues=0;               //Counter for total value that are going to
                                        //be signed
uint8_t counterValuesToSend=0;          //Counter for values that are going to be
                                        //sent each time
uint8_t counterPayloads = 0;

const uint8_t packets = 1;
const uint8_t maxByteToSend = 20;       //Max values that are going to be sent eacht time
const int sizeDataSigned = maxByteToSend*packets;   //At best it would be a multiple
                                                    // of 51, max size of data signed
byte data[sizeDataSigned];              //singed data
byte toSend[maxByteToSend+2];           //data sent each time

//Ackwnowledge payload to receive
uint8_t expectedAck = 0;

//FLAGS
const uint8_t PUB_KEY = 80;
const uint8_t SIGN_1 = 81;
const uint8_t SIGN_2 = 82;

//Schedule TX every this many seconds
//(might become longer due to duty
//cycle limitations).
const unsigned TX_INTERVAL = 120;
```

Listing 3.2: Configurations variables for a LoRa Node data collection and transmission

The `Flags` can be arbitrarily chosen and have to coincide with the ones in the TTN-plugin. In order to be able to recognize what we are sending on the other end, we have to append the corresponding flag to the `data` array.
`TX_INTERVAL` corresponds to the time interval in seconds that a LoRa Node will send data. For tweaking the transmission a little more and specifying more parameters, the official documentation available in [10] can be very helpful.

In order to make the sketch much more understandable at first sight, the whole code is based on states. E.g. when the node wakes up, it's initial state is to send the public key (`SEND_PUBKEY`), then once the Ack is received, it changes state to collect data (`COLLECT_DATA`) and so on, a list of all the state can be seen below.

```
//STATES
const int SEND_PUBKEY = 0;
const int SEND_DATA = 1;
const int SIGN_DATA = 2;
const int COLLECT_DATA = 3;
const int RESET_ALL_DATA = 4;
const int RESET_DATA_TO_SEND = 5;
const int SEND_SIGNATURE = 8;
const int SEND_SIGN_2 = 9;
```

Listing 3.3: State flags on a LoRa Node

The node has to keep always the same keypair and therefore we write them into the EEPROM, which is a non-volatile memory and thus it survives power outages. Hence, at boot the sketch in the function `setup()` checks whether there is a private key stored and if yes it reads it and derives the public key, otherwise generates a new one and writes it into the memory.

The function which is in charge of routing the whole system through the right state is `do_send(osjob_t* j)`. This function gets always called after something has completed, like collecting data, signing it or sending it. In doing so, we know that the Node will proceed to the next state. In order to decide how long we have to wait before executing that function again, we call a function and we pass our initial function:

```
os_setTimedCallback(&sendjob, os_getTime()+sec2osticks(TIME), do_send);
```

As we can see `TIME` refers to the interval that we are going to wait. If in the next state we have to send data through LoRa, then we can set it to `TX_INTERVAL` (see explanation in snippet 3.2), if in the next state we want to collect data from attached sensors, then we can even reduce it and have data points more frequently.
Lastly, the method that is responsible to return data from a sensor is `getDataFromSensor()`. As of now, it returns a random number but if adapted to a sensor it should interface to it and return the read values.

3.2.5 TTN-plugin

The TTN-plugin acts as a management HUB. In figure 3.2 it is represented as the Bazo Light Clients Tx Aggregation. It is responsible for multiple nodes and not only for one and as the name says, its main function is to aggregate transactions coming from different LoRa Nodes. As already mentioned, it is in charge of managing every node and it communicates with them through the TTN SDK written for **Node.js**.
The configuration is pretty straight forward and in order to connect to TTN only two parameters are required. This two parameters can be either hard-coded in the project or set as environment variables (`appId` and `accessKey`).

```
1 //TTN configuration
2 const appID = process.env.appID ? process.env.appID : "name_of_your_app_in_TTN";
3 const accessKey = process.env.accessKey ? process.env.accessKey : "ttn-account-example";
```

As soon as we get the payload from the node, we extract the counter and send it back as an Ack and we delete it from the buffer/data we have just received. Then, we proceed in analyzing what we have just received. Important to notice here is that when the second part of the signature is received, we load the wallet's key, verify the transaction and send it to the blockchain. In order to make it working, there must be a txt file named "WalletA.txt" which has to contain the same public key that the LoRa Node is aware of, otherwise the verification of the transaction and the transaction in general will fail.
When we receive data that is neither a public key nor a signature, we save it in our data structure. First of all, we check whether we have already gotten it by doing a check on the counter we have just received, if that's not the case we aggregate it.

```
const counter = HelperFunctions.DecoderCounter(payload_raw)
//Send the ACK as soon as possible to increase the change to transfer it on time
sendAck(client, devID, counter);


//Possible Packets = PublicKey | Signature1 | Signature2
if (payloadTmp.length === 32) {
    if (counter === PUBLICKEY && sensor.publicKey.length === 0) {
        savePublicKey(sensor, sensorTmp);
    }
    else if (counter === SIGNATURE_1 && sensor.signature.length === 0) {
        saveSignature(sensor, sensorTmp, SIGNATURE_1);
    }
    else if (counter === SIGNATURE_2 && sensor.signature.length === 32) {
        saveSignature(sensor, sensorTmp, SIGNATURE_2);
        bc.rootWallet.then((rootKey)=>{
            const rootPubKey = bc.getPubKeyFromPrivKey(rootKey);
            sensor.verifyDataAndSend(rootPubKey, bc)
        })

    }
}
//Every other packet
else {
    const {packetReceived} = sensor;
    //packet already received, send only ACK
    if (!!packetReceived[counter]) {
        sensor.log('info', `[${time}] ${devID} received existing data #${counter.toString
            ()}, datarate: ${data_rate}, airtime: ${airtime}, gateways: ${numberGateways
            }`)
    } else {
        sensor.packetReceived[counter] = true;
        sensor.counter = counter + 1;
        saveData(sensor, sensorTmp)
    }
}
```

Since the TTN-plugin has to connect to the Bazo Client, we created a class specific for this purpose. The class is located in the folder `api` and its name is `Blockchain.js`. It only requires adapting a mandatory parameter, i.e. the IP address of the Bazo client and the name of the Wallet file, which can be changed:

```
const BLOCKCHAIN_CLIENT_IP = 'http://localhost:8010';
const WALLET = 'WalletA.txt';
```

3.2.6 Elliptic Curve Cryptography

LoRa Node and TTN-listener

Since its invention, Elliptic Curve Crypthograhpy (ECC) has established itself as the default choice for classical public-key cryptography, in particular for constrained environment[17], especially for lightweight Internet of Things (IoT) applications. ECC allows an implementation of smaller memory footprint and low power/energy consumption.
Exactly because of its small memory footprint and low power consumption, we decided to use it on our Arduino's Nodes which should consume as low current as possible so that they can run over a longer period of time without having the need to either charge the batteries or substitute them.

Our LoRa nodes (Arduinos) and our TTN-plugin, which is supposed to run on a RPi, use Ed25519.

Ed25519 is an instance of Edwards-Curve Digital Signature Algorithm (EdDSA) which uses a Edwards Curve 25519 and SHA-512 for hashing. Its main goal was to solve an inconvenience of the more established ECDSA, namely, both schemes require the generation of a random value during the signature process and a critical part for security is the secrecy of that random value[17]. Since it is not specified how to generate the random value, guarantee the quality of it it's pretty difficult. EdDSA removes this dependence by deriving the secret deterministically from the message and a long term auxiliary key using a cryptographic hash function[17].

For the arduino implementation, a cryptography library [12] is used. This library creates digital signature based on the elliptic curve modulo $2^{255} - 19$. It generates a key pair (public and private key) of 32 Bytes each, which in our case fits into the memory of our arduino. Signing a message produces however a 64-byte signature that has to be split because of the limitation of the LoRaWAN.

The library can be used as follows:

```
#include <Ed25519.h>

uint8_t privateKey[32];
uint8_t publicKey[32];

//Generate the private and public key and saved^ it
Ed25519::generatePrivateKey(privateKey);
Ed25519::derivePublicKey(publicKey, privateKey);


//Sign a message and save the signature using the private and
//public key
byte message[N];
uint8_t signature[64];

Ed25519::sign(signature, privateKey, publicKey, message, N);

//Verify the signature for the message using the public key and
//the signature
if (!Ed25519::verify(signature, publicKey, message, N)) {
    // The signature is invalid.
    ...
}
```

Listing 3.4: Sign a message on the node

On the TTN-plugin, the ed25519 library can be found here [13] or since it is a node package manager (NPM) module it can be easily installed with the following command on the terminal "`npm install ed25519`" in the root directory of a project. First make sure to have the latest current available version of Node.js and NPM

The usage is pretty straight forward and it works as follows:

```
//Import statement in order to be able to use the library in this file
const ed25519 = require('ed25519');

//Signature and Public Key already generated elsewhere
//In our case they can be both from a LoRa node which signs the same
//message "Hello World!"
```

```
const signature = new Int8Array(32);
const publicKey = new Int8Array(64);

let message = "Hello World!";
valid = ed25519.Verify(new Buffer(message),
                    new Uint8Array(signature),
                    new Uint8Array(publicKey))

if (valid) {
        console.log('Signature valid');
        //...
    } else {
        console.log('Signature invalid');
        //...
    }
```

Listing 3.5: Verify a signature in Node.js

3.2.7 Bazo Blockchain

In this chapter we will discuss the adaptations needed on the Bazo Blockchain in order to be able to save data from IoT devices (LoRa nodes). Since the addresses, the public keys and the verification of all transactions had to change, the classes which needed some modifications are substantially many.
Making a summary of the affected ones would mean listing almost every class contained in the project. Therefore, I will point out only the most important changes and not the consequences that they have on smaller classes and methods that are straightforward to adapt.

Cryptography

Like already mentioned, Bazo was using the ECDSA cryptography and since our IoT devices use Ed25519 we had to change Bazo as well and thus, the way Public and Private keys are generated, the structure of Signatures and at the end the whole process of verifying transactions as well.
For the new implementation, we used an official *Go* package which can be found here [6] along with its official documentation for Ed25519.
The main difference between this package and the other packages used on the Arduino or on Nodejs (on other platform in general) is that the Private key is 64 Bytes long instead of 32 as usual.
By reading some documentation and by testing a little bit around, we found out that this is not a problem since the last 32 Bytes of the Private key contains exactly the Public key, *i.e PublicKey = PrivateKey[32:]*.
For sake of simplicity, during the implementation the Public Key is sometimes derived from the Private Key as shown above. The same approach can be found in the source code of the official package *ed25519/ed25519.go*

```
// GenerateKey generates a public/private key pair using
// entropy from rand.
// If rand is nil, crypto/rand.Reader will be used.
```

```
78  func GenerateKey(rand io.Reader) (PublicKey, PrivateKey, error) {
79    if rand == nil {
80      rand = cryptorand.Reader
81    }
82
83    seed := make([]byte, SeedSize)
84    if _, err := io.ReadFull(rand, seed); err != nil {
85      return nil, nil, err
86    }
87
88    privateKey := NewKeyFromSeed(seed)
89    publicKey := make([]byte, PublicKeySize)
90    copy(publicKey, privateKey[32:])
91    return publicKey, privateKey, nil
92  }
93
94  // NewKeyFromSeed calculates a private key from a seed. It will
95  // panic if len(seed) is not SeedSize. This function is provided
96  // for interoperability with RFC 8032. RFC 8032's private keys
97  // correspond to seeds in this package.
98  func NewKeyFromSeed(seed []byte) PrivateKey {
99    if l := len(seed); l != SeedSize {
100     panic("ed25519: bad seed length: " + strconv.Itoa(l))
101   }
102
103   digest := sha512.Sum512(seed)
104   digest[0] &= 248
105   digest[31] &= 127
106   digest[31] |= 64
107
108   var A edwards25519.ExtendedGroupElement
109   var hBytes [32]byte
110   copy(hBytes[:], digest[:])
111   edwards25519.GeScalarMultBase(&A, &hBytes)
112   var publicKeyBytes [32]byte
113   A.ToBytes(&publicKeyBytes)
114
115   privateKey := make([]byte, PrivateKeySize)
116   copy(privateKey, seed)
117   copy(privateKey[32:], publicKeyBytes[:])
118
119   return privateKey
120 }
```

Listing 3.6: Generate a Key-Pair with Ed25519

From the code snippet of the method NewKeyFromSeed(), we can clearly see that the public Key is on purpose put into the last 32 Byte of the private Key. When signing a message, only the first 32 Byte of the private Key are used. Thus, this evidently does not bring any difference with other Ed25519 implementations.
Verifying the data from the node using this package didn't require any particular change and the verification itself succeeded. Therefore, once we knew that it was compatible, we proceed in adapting the whole blockchain.

Bazo-Miner

Adapting the Miner required much more effort than the client since the client usually imports whole packages from the miner. The biggest class which required a complete change can be found in the *crypto* package.
The *keypair.go* class is responsible for generating the private key and deriving the public key from it. It loads both of them from the wallet when we want to utilize an existing

account and it dumps them in it when we create a new account.
Moreover, it handles the transformation from a public key to an address and vice-versa. In the whole code, methods contained in this class are often used and thus, changing those methods, their input and output parameters, requires adapting all the affected classes and variables.

The two terms **Commitment Key** and **Commitment Proof** might be misleading since they are responsible for signing and verifying but as we will see, both of them are not used for transactions.
The **Commitment Proof** is *"a property which stores a signed message of the Height that a block was created. In particular, RSA(sk, SHA3-512(Height)) where sk represents the private key that corresponds to the public key pk that was set in the initial StakeTx of the node. Other validators can use pk in order to verify the proof"[3].*
A **Commitment Key** is a keypair generated when a node wants to join the set of validators. To keep in mind is that this keypair is different from the one in the user's wallet. Once a node submits it, it cannot change it in a later step but it is committed to it.
Very important to know, is that the generation of the latter pair, requires special properties for the generating function, therefore we cannot just assume that switching them to Ed25519 would work as good and no security leeches are introduced.
Because the keypair is only required by miners and not by users, we are still able to use the old version based on RSA and Ed25519 in parallel without any problem.
To sum up, when verifying and signing transaction, we use Ed25519, when miners which stake want to sign the height of the block, then they use RSA.

Bazo-Client

The client makes use of many Miner's packages and methods to run. This means that it didn't really need big adaptations in the base logic but only in the variables type and methods. For example, addresses were 64 Bytes long whereas now their are only 32 Bytes, methods that were using or returning objects of type *ECDSA* now have to use *ED25519*. Other major changes were not required so far for the adaptation to ED25519.

3.2.8 IoT Transaction & Wallet instantiation

This sub chapter is required in order to explain what happens before an IoT transaction from a LoRa Node can be actually issued on the blockchain. Since the whole procedure requires multiple steps, a sequence diagram, which should help in better grasping the concept, can be found at the end of this section.
Data types in order to be able to successfully make this steps and also create an IoT Transaction are going to be illustrated and explained.

The BaZo Client implements an already working REST API that we can utilize to send transactions from our TTN-plugin to the blockchain. Since IoT transactions are being introduced in this blockchain, we had to integrate some new endpoints in the REST API (class `REST/REST.go`), so that we could easily issue a new type of transaction.

For our TTN-plugin we had to write and utilize the following new endpoints:

```
38 router.HandleFunc("/sendTxIoT", SendIoTTxEndpoint).Methods("POST")
39 router.HandleFunc("/createAccTxIoT",CreateAccTxEndpointWithPubKeyIoT).Methods("POST")
40 router.HandleFunc("/createFundsTxIoT", CreateFundsTxIoT).Methods("POST")
```

Listing 3.7: New Endpoints Bazo Client REST API

When a node wants to send a transaction to the blockchain, the first thing the TTN-plugin will do, as soon as it receives the public key from the node, is to check whether the public key is already present in the blockchain. In doing so, we know whether the blockchain is aware of the node and a Wallet has been already created or not. This check is done by using an already implemented endpoint, i.e. an HTTP GET call to `/account/{id}`.
If the account is not present yet, it has to be instantiated before being able to successfully issue transactions to the blockchain. For this, we make an HTTP POST call to the endpoint: `/createAccTxIoT`. This call requires a body like the following one:

```
1 body = {
2     DevId: String,
3     PublicKey: Integer[],
4     Issuer: Integer[],
5     Fee: integer,
6     TxCnt: integer
7 }
```

Listing 3.8: New createAccTxIoT Body Endpoint

As a response we get a `txHash` that we have to sign using an existing wallet account. In our case we are using the root account. After signing the `txHash`, we send it together with the signature to the Client using another endpoint, i.e. `/sendAccTx/{txHash}/{signature}`. Only at this point an `accTx` transaction is issued to the miners of the network. Now, we have a wallet on the blockchain for the node but its account is empty, which means it does not have any funds. Therefore we have to send some amount by issuing a `fundsTx` to the miners. We need to do another HTTP POST call to the endpoint `/createFundsTxIoT` with the following body, where the amount signifies how many funds we want to transfer:

```
1 body = {
2     DevId: String,
3     ToPubKey: Integer[],
4     FromPubKey: Integer[],
5     Fee: integer,
6     TxCnt: integer,
7     Amount: integer
8 }
```

Listing 3.9: New createFundsTxIoT Body Endpoint

As before, we get a `txHash` which we have to sign again with a Wallet and send it to the Client using a different endpoint, i.e. `/sendFundsTx/{txHas}/{signature}`.
Only at this moment, the node is actually able to send transactions to the Blockchain,

since we know that it has a Wallet and some Funds.
At this point, the TTN-plugin makes a last HTTP POST call to a new endpoint `/sendTx-IoT`. In our case, this is the most important call because only now an IoT transaction starts actually getting into the network. The call looks as follow:

```
body = {
    PublicKey: Integer[],
    Data: Integer[],
    Signature: Integer[],
    TxCnt: integer,
    TxFee: integer,
    TxHash: Integer[],
    To: Integer[],
    DevId: String,
}
```

Listing 3.10: New sendTxIoT Body Endpoint

The parameter `PublicKey` corresponds to the one of the node that we get at the beginning when it starts sending data. It is individual and unique and it has to stay always the same for each node. Every node has to have an account instantiated on the blockchain and it has to contain some funds, those three properties are mandatory to be able to issue IoT transactions.
`Data` includes all the data taken from sensors connected to a node (for the purpose of this work, mock data has been used).
`Signature` is the generated signature on the node from signing the whole transaction with its private key, later on it will be better explained what data is signed on the node and how it is done.
`TxCnt` corresponds to the actual transaction counter on the Bazo blockchain, this parameter is required for funds transactions in order to avoid *double spending*.
To bypass some more synchronization between the node and the blockchain, we still accept IoT transactions on the blockchain even if the given counter does not correspond to the expected one on the blockchain. At the time of writing this work, a problem with the TxCnt is known and being tackled by another student, therefore we don't take it into consideration here.
`TxFee` corresponds to the fee that a node has to pay in order to be able to issue a transaction on the blockchain, it is always equals to 1 so far, which means that each transaction a node sends will cost him 1 token.
A better approach would be to dynamically estimate the cost based on a percentage of the size of the data sent (i.e: `Data`). In doing so, if a Node sends a lot of data, it will have to pay a higher fee which actually makes more sense since more data are going to be saved on the blockchain and thus it will require more storage on the miner's drive.
`DevId` is so far not used in the blockchain, therefore it can be either removed or stay optional, since it is provided by TTN we decided not to remove it. Lastly, the parameter `To` indicates the address of the receiver.

At the time of writing this work, Bazo does not support *Smart Contracts* and therefore we have to include in the transaction an address of a recipient and not of a contract. This approach still works because we can assume that an IoT device belongs to somebody and he/she is part of the blockchain as well, thus every node he/she owns, will send data to him/her. When an IoT device will utilize all its funds by sending data, the owner

can transfer some amount back to the correct wallet linked to the IoT device and thus re-enable the device to issue transactions to the network.

A sequence diagram is provided below to better understand the whole procedure of instantiating a new wallet from a LoRa node up to sending a transaction to the network.

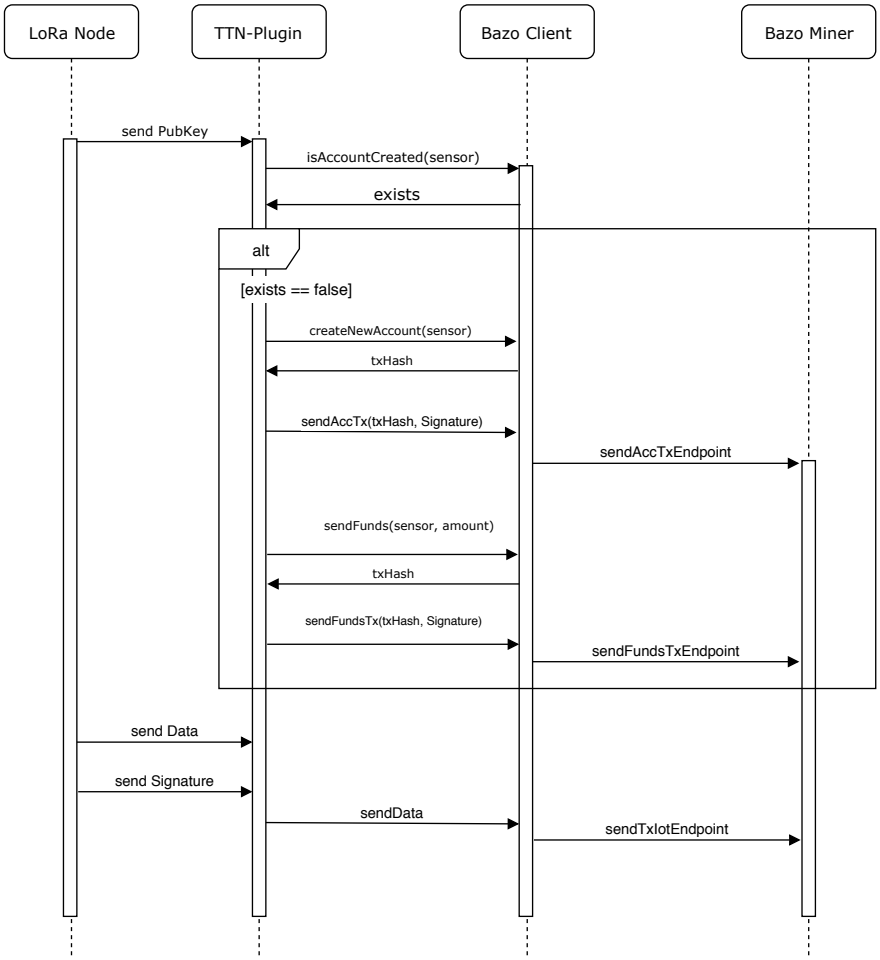


Figure 3.12: Data Flow and instantiation of an IoT Wallet on the blockchain.

3.2.9 IoT Transaction Implementation

In this sub chapter, we are going to explain what happens on the LoRa node so that a full IoT transaction can be issued starting from there. This means that not only the collected data from sensors is signed.
Right after it, we will have a look at how the TTN-plugin manages everything and issues a final IoT Transaction to the network. Lastly, we will analyze how Miners are going to validate the transaction since it is slightly different from any transaction issued on the network so far.

LoRa Node (Arduino)

In order to be able to have a completely signed transaction, every LoRa node should know the address of an Account (`walletPubKey`). This account will represent the address to whom transactions are going to be sent. Moreover, in order to have a complete transaction the LoRa node should be aware of the transaction fee (`txFee`), of the header (`Header`) and of the transaction counter (`txCnt`).

```
1 //Wallet account
2 uint8_t walletPubKey[32] = {103, 39, 245, 29, 180, 20, 70, 224, 64, 38, 31, 91, 8, 141,
      50, 148, 74, 224, 178, 225, 169, 244, 123, 236, 151, 62, 207, 117, 150, 236, 214,
      154};
3 uint32_t txCnt = 0;
4 uint64_t txFee = 1;
5 byte Header = 0;
```

Listing 3.11: Example of required IoT Wallet variables for Arduino devices

As we can see in the code snippet 3.11, an account (a root account in this case) is represented as an array of uint8_t of length 32, which corresponds to its public key. As already mentioned before, the `txCnt` is not used so far and therefore stays always equals to 0 until a final solution is found. The `txFee` is still static and equals to 1 and it can be easily adapted to change dynamically based on the size of the data the LoRa node is collecting and sending. The `Header` stays always equals to 0 as well.
Important to point out is that those three values (excluding the txCnt) are always known to the TTN-plugin and therefore, they don't have to be retransmitted each time the node wants to issue a transaction to the blockchain. Supposing that the `txFee` changes and gets dynamically adapted to the size of the data, calculating the actual fee won't require a new packet to be sent through the LoRaWAN to the TTN-plugin since the latter is also aware of the size of the data.

Although we can avoid transmitting those three values each time, if we decide to take into consideration the `txCnt`, which is used to avoid the double spending attack, then a new problem arises.
More precisely we can have two different scenarios:

- The transaction's counter (txCnt) is equal or greater than the actual txCnt on the blockchain (`tx.txCnt >= state.txCnt`).
- The transaction's counter has to be equal to the actual one on the blockchain (`tx.txCnt == state.txCnt`).

In the first case, we would have to send the actual txCnt from the LoRa Node to the TTN-plugin (uplink message) at some point before completing the IoT Transaction. We cannot assume that the actual txCnt is equals to the old txCnt +1 because we won't know if a complete transaction got lost during transmission and therefore all the next transaction will fail due to a wrong txCnt.

In the second case, where the tx.txCnt has to be equal to the state.txCnt on the blockchain,

we would have to send a downlink message from the TTN-plugin to the LoRa node. This is required because in this case, the node is not aware and cannot be sure that its transactions get always validated and successfully issued on the blockchain. Therefore, it has to know the current txCnt, so that it can correctly include it when signing the transaction and the generated signature will correspond to the right txCnt.
Unluckily, this approach implies another problem present on the current version of the Bazo Client, i.e: the REST interface on the Bazo client does not work as expect and when we want to gather informations about an account using the following endpoint: `/account/id`, we always get an account where the current state of all the variables is always equal to 0, like shown in the JSON response in 3.12. Moreover, since we are querying an IoT account, we can clearly see that the field `"isRoot"` is wrong as well; an IoT device cannot be the root account.

```
{
    "code": 200,
    "content": [
        {
            "name": "account",
            "detail": {
                "address": "76a1592044a6e4f511265bca73a604d90b0529d1df602be30a19a9257660d
                    1f5",
                "balance": 0,
                "txCnt": 0,
                "isCreated": true,
                "isRoot": true,
                "isStaking": false
            }
        }
    ]
}
```

Listing 3.12: JSON response from the Bazo client when querying an account

Compared to Figure 3.13, which is a screenshot showing the current state of the blockchain on a local machine, we can see that the account is not a root account (`Is root: false`), it has a balance which is not equals to 0 (but instead `Balance: 998`) and two transactions have been already issued to the blockchain from that account (`TxCnt: 2`).

```
Previous Hash: e6a91a9dc2e8a4d4     Previous Hash Without Tx: 24e532f097a22221
Aggregated: false
Nonce: 000000005c91272a
Timestamp: 1553016618
MerkleRoot: 0000000000000000
Beneficiary: 22863c37e05c4591
Amount of fundsTx: 0 --> []
Amount of accTx: 0 --> []
Amount of configTx: 0 --> []
Amount of stakeTx: 0 --> []
Amount of aggTx: 0 --> []
Amount of IoTTx: 0 --> []
Total Transactions in this block: 0
Height: 17
Commitment Proof: 54cbd6fb6041d918
Slashed Address:0000000000000000
Conflicted Block Hashes 1:0000000000000000  =  0000000000000000
Conflicted Block Hashes 2:0000000000000000  =  0000000000000000
State:
Number Accounts: 3
Is root: true, Hash: 22863c37e05c4591, Address: 6727f51db41446e0, Issuer: 0000000000000000, TxCnt: 2, Balance: 1804, IsStaking: true,
Is root: false, Hash: e2701b9f126cac26, Address: c58180c5e5fdaef5, Issuer: 6727f51db41446e0, TxCnt: 2, Balance: 998, IsStaking: false,
Is root: false, Hash: bb4a3b120a4e902d, Address: 76a1592044a6e4f5, Issuer: 6727f51db41446e0, TxCnt: 2, Balance: 998, IsStaking: false,
 -> With Balance: 3
```

Figure 3.13: State of the blockchain

To sum up, before taking any decision regarding the transaction counter, some other bugs present on the current version of the client have to be fixed.

"A structure type is a user-defined composite type. It is composed of fields or members that can have different types. In C++, a structure is the same as a class...[18]".
When we want to sign our transaction, we are not able to sign a structure which represents our IoT transaction, but instead we have to take every single field, convert it into a byte array and put it together with the other fields in a single byte array that is going to be hashed at the end. Only then, we are able to sign it. In this case, the order of the array is very important since the same order has to be kept in the TTN-plugin and in the miners when a transaction has to be verified. The chosen order is the following:

walletPubKeyHash | publicKeyHash | byteTxCnt | byteTxFee | Header | Data

Public-Keys have to be hashed as well before putting them into the data array that is going to be signed. This can be done using SHA3 and it produces an hash of length 32. The process is simple and displayed below in code snippet 3.13:

```
1 #include <SHA3.h>
2
3 uint8_t publicKey[32];
4 byte hash[32];
5
6 SHA3_256* sha = new SHA3_256();
7
8 sha->update(publicKey,sizeof(publicKey));
9 sha->finalize(hash,sizeof(hash);
```

Listing 3.13: Hashing a public key on arduino

TTN-plugin

The TTN-plugin is aware of the node's public key since it receives it when a nodes wakes up. Moreover, it gets the data that the node sends and aggregates it as well. In order to be able to recreate the IoT transaction and avoid that the node has to send the other account public key (of the receiver), it has to contain a wallet file in its root directory.
That file can be simply taken from the miners directory when an account is instantiated and copied into it. It has to be called `"WalletA.txt"`, just like in the miner. The `Header` is always equal to 0 and the txCnt stays always the same as well (see the two scenarios described above). Therefore, we don't need any other information from the node.
Because data can be either lost or corrupted during the transmission from the node to the TTN-plugin, once we receive the whole transaction, we verify that it matches the signature. In doing so, we not only assure that the transaction is valid and no packet is lost but we also avoid that the blockchain and thus miners get flooded with transaction that are already invalid. To keep in mind is that we only check whether the signature corresponds to the transaction, we do not check on the TTN-plugin if the node's account has enough balance to issue new transactions and so on. The procedure of verifying the

transaction on the TTN-plugin is simple and is done as follows:

```
const ed25519 = require('ed25519');
const Buffer = require('buffer').Buffer;
const sha = require('js-sha3');
const HelperFunctions = require("./HelperFunctions");

let rootPubKey = [1,2,3...];
let publicKey = [4,5,6...];
let signature =[2,5,7...];
let txCnt = 0;
let txFee = 1;
let data = Buffer.from([0,0,1,1,...]);
let header = new Buffer.alloc(1);
header.writeInt8(0, 0);

//The order is very important, they have to be concatenated
//and thus signed in the same order as in the arduino and in the
//blockchain
let txToSign = Buffer.concat([
    Buffer.from(sha.sha3_256(Buffer.from(rootPubKey)), 'HEX'),
    Buffer.from(sha.sha3_256(publicKey), 'HEX'),
    Buffer.from(HelperFunctions.getInt32Bytes(txCnt)),
    Buffer.from(HelperFunctions.getInt64Bytes(txFee)),
    header,
    data]);

const txHash = Buffer.from(sha.sha3_256(txToSign), 'HEX');

isValid = ed25519.Verify(txHash, signature, publicKey);

if (valid) {
        console.log('Signature valid');
        //...
    } else {
        console.log('Signature invalid');
        //...
    }
```

Listing 3.14: Hashing and verifying an IoT transaction in the TTN-plugin

At this point, if the validation does not fail, the IoT transaction gets sent to the Bazo-Client as explained in chapter 3.2.8.

Bazo-client

The client isn't required to do much, it will only gather the IoT transactions from the TTN-plugin, convert it into the right format for the blockchain and then broadcast it to the miner to whom it is connected. The latter will then verify the transaction and do the rest of the job. The client acts simply as a forwarder since we cannot have a light client on the Arduino because of the limitation imposed by the LoRaWAN.
In the following code snippet, we can see the structure of an IoT Transaction on the blockchain.
Important to mention is that this transaction does not have a field `amount`, since it does not transfer any funds and the field `data` has not a specific length because it can be dynamic and we do not know how much data an IoT device will send to the blockchain. Therefore, we can even see that the function `func (tx *IotTx) Size()`, which is used

to get the size of the transaction, cannot be static like we did for the other types of transaction, but it has to be dynamic and based on the size of the data transmitted as well.

```
type IotTx struct {
  Header byte
  TxCnt  uint32
  From   [32]byte
  To     [32]byte
  Sig    [64]byte
  Data   []byte
  Fee    uint64
}

func (tx *IotTx) Size() uint64  {
  size := int(unsafe.Sizeof(*tx)) + len(tx.Data)
  return uint64(size)
}
```

Listing 3.15: IotTx Data Struct in miner/protocol/iottx.go

Bazo-miner

The first thing a miner does, as soon as it receives the IoT transaction, is to process the transaction and broadcast it to other miners. When processing it, it first checks whether the transaction was already present in the open or in the closed transactions. If not, it writes it in the openTx on the storage and broadcasts it. The code snippet from the class `process.go` below explains better how this works.

```
1  func processIotTxBrdcst(p *peer, payload []byte, brdcstType uint8) {
2    var tx protocol.Iot;
3
4      //****
5      //omitted code
6      //***
7
8    if storage.ReadOpenTx(tx.Hash()) != nil {
9      logger.Printf("Received  IoT transaction (%x) already in the mempool.\n", tx.Hash())
10     return
11   }
12   if storage.ReadClosedTx(tx.Hash()) != nil {
13     logger.Printf("Received  IoT transaction (%x) already validated.\n", tx.Hash())
14     return
15   }
16
17   //Write to mempool and rebroadcast
18   storage.WriteOpenTx(tx)
19
20   toBrdcst := BuildPacket(brdcstType, payload)
21   minerBrdcstMsg <- toBrdcst
22 }
```

Listing 3.16: Process and Broadcast of an IoT Transaction

This process is required to avoid double transactions.
At this point, the IoT transaction is written in the mempool and is ready to be checked and validated when the next block is going to be mined.

3.2.10 IoT Transaction Verification

In this section, we will explain which requisite an IoT Transaction has to satisfy in order to be saved on the blockchain, Moreover, the whole process of verifying IoT transactions will be thoroughly explained since it differs quite a bit from all the other types of transactions present on Bazo.

Since the incoming IoT transaction has been saved in the mempool, the process of mining a block can now start from the function `mining` in the file `blockchain.go`;

```
func mining(initialBlock *protocol.Block) {
    currentBlock = ...
    ...
    ...
    prepareBlock(currentBlock)
    blockValidation.Unlock()
}
```

As we cann see, the function `prepareBlock()` is called; starting from this function, all the validations required for the transaction are going to be executed for all the transaction open in the storage.

```
func prepareBlock(block *protocol.Block) {
  //Fetch all txs from mempool (opentxs).
  opentxs := storage.ReadAllOpenTxs()
  ...
  ...
  err := addTx(block, tx)
  if err != nil {
    //If the tx is invalid, we remove it completely, prevents starvation in the mempool.
    storage.WriteINVALIDOpenTx(tx)
    storage.DeleteOpenTx(tx)
}
```

Every transaction in the block has to go through the function `verify(tx)` and when the block includes an IoT Transaction, then the function `verifyIotTx(tx *protocol.IotTx)` gets executed. This function is very important since it contains most of the validations required.

```
1  func addTx(b *protocol.Block, tx protocol.Transaction) error {
2      ...
3      ...
4      if !verify(tx) {
5      return errors.New("Transaction could not be verified.")
6      }
7      ....
8      ....
9      case *protocol.IotTx:
10         err := addIoTTx(b, tx.(*protocol.IotTx))
11       if err != nil {
12         return err
13     }
14 }
```

```
func verify(tx protocol.Transaction) bool {
    var verified bool

  switch tx.(type) {
  ...
  case *protocol.IotTx:
    verified = verifyIotTx(tx.(*protocol.IotTx))
}
```

Initially, we gather both the Sender's and Recipient's addresses and if one of them does not exit, which means that it is not stored in the storage's state, then the transaction won't be valid. That's why in the TTN-plugin, the first thing we have to check is whether the LoRa nodes's wallet with its public key is already instantiated in the blockchain. If that weren't the case, then all the IoT transaction coming from that specific node would fail.

```
1  func verifyIotTx(tx *protocol.IotTx) bool {
2      if tx == nil {
3      return false
4    }
5
6    //Check if accounts are present in the actual state
7    accFrom := storage.State[tx.From]
8    accTo := storage.State[tx.To]
9
10   //Accounts non existent
11   if accTo == nil || accFrom == nil {
12     logger.Printf("Account non existent. From: %v\nTo: %v\n", accFrom, accTo)
13     return false
14   }
15   //IoT devices can hash the address differently. Usually we transform
16   //both addresses (From & To) into a string using sprintf
17   //and then we hash it, in arduino this does not give the same hash
18   //and therefore the only way is to
19   //hash it as a byte[] directly without converting it first into
20   //a string.
21   //Only so, the txHash will correspond to the right one
22   accFromHash := protocol.SerializeHashContentIoT(accFrom.Address)
23   accToHash := protocol.SerializeHashContentIoT(accTo.Address)
24   copy(tx.From[:], accFromHash[:]);
25   copy(tx.To[:], accToHash[:]);
26
27   txHash := tx.Hash()
28   pubKey := accFrom.Address[:]
29   if ed25519.Verify(pubKey, txHash[:], tx.Sig[:]) && tx.From != tx.To {
30     tx.From = protocol.SerializeHashContent(accFrom.Address);
31     tx.To = protocol.SerializeHashContent(accTo.Address);
32     return true
33   } else {
34     return false
35   }
36 }
```

The second important thing now is that we cannot just take the address' hash like we usually would do with other transactions. Normal transactions utilize the function `SerializeHashContent(data interface{})` to hash addresss or objects in general.
As we can see in the function, the input parameter gets first converted into a string by the function `fmt.Sprintf("%v",data)` and then hashed. Unluckily, this same process is not possible on the Arduino sketch, since converting the address into a string and then

hash it, gives a completely different hash than the one obtained on the blokchain. Because of this, the method `SerializeHashContentIoT(data [32]byte)` has to be used. This function should be used only to hash addresses or public keys, that's why it accepts only byte arrays of length 32. In doing so, we can guarantee that the hashes generated starting from the same address or public key will be the same.

```
//Serializes the input and returns the sha3 hash function applied on the
//input
func SerializeHashContent(data interface{}) (hash [32]byte) {
  return sha3.Sum256([]byte(fmt.Sprintf("%v", data)))
}
//Hash the data as it is done in the Arduino sketch so that the
//two hashes are the same
func SerializeHashContentIoT(data [32]byte) (hash [32]byte) {
  return sha3.Sum256(data[:])
}
```

Now, since on Arduino we cannot just hash the transaction as if it were a structure/class, we have to follow a precise order. Namely, we have to convert again every field of the transaction into a byte array and then concatenate them in the same way we did it on the node (see Chapter 3.2.9). Only then we can actually hash the transaction and return the corresponding one which will match the one on the node.

```
1  func (tx *IotTx) Hash() (hash [32]byte) {
2    if tx == nil {
3      //is returning nil better?
4      return [32]byte{}
5    }
6    //Order -> To From  txCnt txFee Header  data
7    buf := new(bytes.Buffer)
8    binary.Write(buf, binary.BigEndian, tx.To);
9    binary.Write(buf, binary.BigEndian, tx.From);
10   binary.Write(buf, binary.BigEndian, tx.TxCnt);
11   binary.Write(buf, binary.BigEndian, tx.TxFee());
12   binary.Write(buf, binary.BigEndian, tx.Header);
13   binary.Write(buf, binary.BigEndian, tx.Data);
14
15   return sha3.Sum256(buf.Bytes())
16 }
```

As a comparison, we can see how all other transactions are usually hashed. This method is pretty simpler than what we have to use but unluckily we cannot apply it.

```
1  func (tx *ConfigTx) Hash() (hash [32]byte) {
2
3    if tx == nil {
4      return [32]byte{}
5    }
6
7    txHash := struct {
8      Header  byte
9      Id      uint8
10     Payload uint64
11     Fee     uint64
12     TxCnt   uint8
13   }{
14     tx.Header,
15     tx.Id,
16     tx.Payload,
17     tx.Fee,
```

```
18      tx.TxCnt,
19    }
20    return SerializeHashContent(txHash)
21  }
```

Only at this point, we are able to verify our IoT Transaction (*func verifyIotTx(), line 29*) and if the signature corresponds to the transaction, we `return true` but before that, we have to revert back our addresses as they are usually hashed on other transactions, otherwise we will get the error that those addresses do not exist in the storage's state even though they do. This is caused because of the problem explained above.
After returning `true` and thus not throwing any error, the code will enter a `switch statement` and when an IoT transaction is being processed, then the function `addIoTTx()` will be called.
This last method is in charge of checking the last state of the transaction and of the account before being permanently stored.
First, the code makes sure that the account which sends the transaction has enough funds, if it is not a root account. This check is done in the same ways as it is done for funds transactions. After that, the check for the tranasction counter (txCnt) is done but as we can see from the code, we do not return any error at the moment. It's there as a place holder.
Next, the corresponding issuer's account is retrieved from the store so that the transaction counter can be updated and the transaction fee detracted.

```
1  func addIoTTx(b *protocol.Block, tx *protocol.IotTx) error {
2    if !storage.IsRootKey(tx.From) {
3      if (tx.Fee) > b.StateCopy[tx.From].Balance {
4        acc:= b.StateCopy[tx.From]
5        return errors.New("Not enough funds to complete the IoT transaction!")
6      }
7    }
8    if b.StateCopy[tx.From].TxCnt != tx.TxCnt {
9      //err := fmt.Sprintf("Sender txCnt IoT does not match: %v (tx.txCnt)
10     //vs. %v (state txCnt)", tx.TxCnt, b.StateCopy[tx.From].TxCnt)
11     //return errors.New(err)
12   }
13   accSender := b.StateCopy[tx.From]
14   accSender.TxCnt += 1
15   accSender.Balance -= tx.Fee
16   b.IoTTxData = append(b.IoTTxData, tx.Hash())
17   return nil
18   }
```

The IoT transaction is saved on the blockchain as soon as the mining process for that block is done.
Figure 3.14 shows a block that has just been mined containing the hash of a valid IoT transaction.

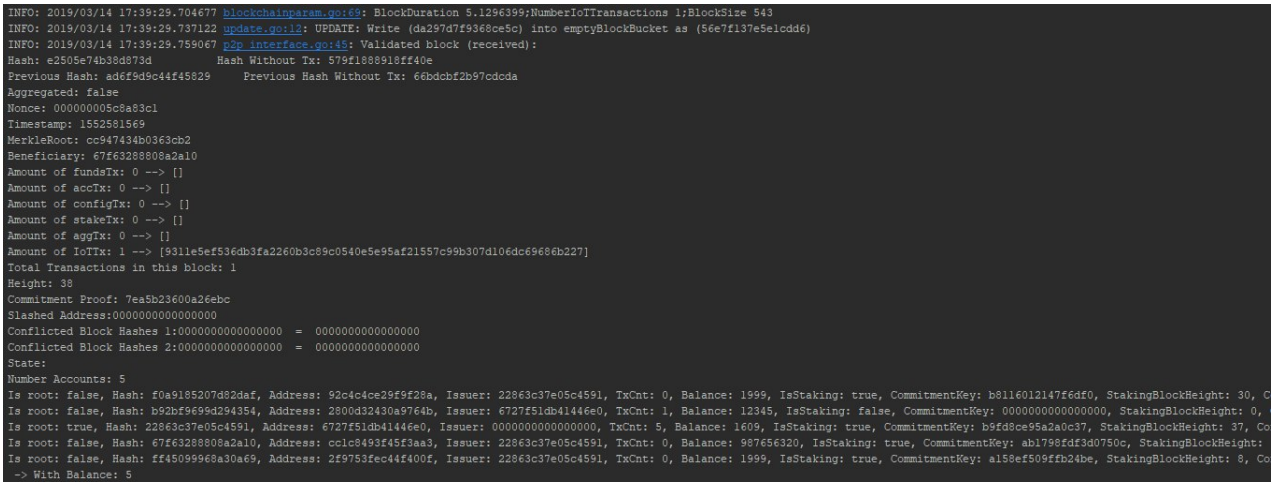


Figure 3.14: Mined block containing an IoT Transaction.

3.2.11 Supply Chain Back-end to Bazo

First of all, we will explain how the configuration for Bazo on the Supply Chain platform is done. Thanks to this explanations the back-end can be then changed to work under different conditions and properties.
The new properties are done similarly like done for Ethereum, for better explanation take a look at the official report of the supply chain monitoring platform.

```
1  package com.food.chain.server.fcserver.bazo;
2  import org.springframework.context.annotation.Configuration;
3
4
5  @Configuration
6  class BZPropertiesBean {
7
8      private String rootAddress =
9          "6727f51db41446e040261f5b088d32944
10         ae0b2e1a9f47bec973ecf7596ecd69a";
11
12     private String accountSeed =
13         "9d61b19deffd5a60ba844af492ec2cc44
14         449c5697b326919703bac031cae7f60d75
15         a980182b10ab7d54bfed3c964073a0ee17
16         2f3daa62325af021a68f707511a";
17
18     private String bazoHttpServiceUrl = "http://localhost:8010";
19
20     public String getRootAddress() {
21         return rootAddress;
22     }
23
24     public void setRootAddress(String rootAddress) {
25         this.rootAddress = rootAddress;
26     }
27
28     public String getAccountSeed() {
29         return accountSeed;
30     }
31
32     public void setAccountSeed(String accountSeed) {
33         this.accountSeed = accountSeed;
34     }
```

```
35
36      public String getBazoHttpServiceUrl() {
37          return bazoHttpServiceUrl;
38      }
39
40      public void setBazoHttpServiceUrl(String bazoHttpServiceUrl) {
41          this.bazoHttpServiceUrl = bazoHttpServiceUrl;
42      }
43  }
```

Listing 3.17: Bazo Properties in the Food Chain Backend

Because of different configurations with the environments, we were not able to externalize the variables and therefore they are hard-coded into the Class BZPropertiesBean.
The `rootAddress` is the public key of the root account, or any account already present in the blockchain, and it has the same role like in the TTN-plugin and on the LoRa Node, i.e. it acts like a Smart Contract address.
In the current version of the Supply Chain Backend it corresponds to the variable `contractAddress`.
The `accountSeed` corresponds to the `accountPrivateKey` in the original version. In our case we use a seed from which we will derive the private and public key. We wrote a small script in javascript to recreate a wallet file compatible with Bazo starting from a seed. The script can be found in the github repository and it's called `ConvertSeedToWallet.js`. Make sure to have a file named `Seed.txt` in the main directory where the script's file is, containing your seed in a single line and the output will be then written in a file named `Wallet.txt`. In order to use it just rename it and copy it in the bazo client or miner.
The variable `accountSeed` should not even exist in the backend but because of a design flaw it is necessary to run the current version as it is. This means that the Food Chain has to have a valid account on the blokchain as well and all the transactions done are charged on this account.
The `bazoHttpServiceUrl` simply represents the URL of the the Bazo Client to which the Supply Chain Backend will connect.

The next important class is `BAZOClient.java`. Like already done with the `ETHClient.java`, this new class acts as a proxy for the whole backend. When a service wants to send data to Bazo, then it has to inject this class and utilize it.
It wraps the more important methods like `sign()` and `verify()` so that the client of this class does not have to do multiple steps.

```
1   /** IMPORTS & GETTERS/SETTERS OMITTED **/
2   package com.food.chain.server.fcserver.bazo;
3
4   @Slf4j
5   @Service
6   public class BAZOClient {
7
8       private final BZPropertiesBean bazoProperties;
9       private final EdDSAParameterSpec spec;
10      private final EdDSAPrivateKeySpec privKey;
11      private final EdDSAPublicKeySpec pubKey;
12      private final byte[] seed;
13      private final byte[] rootAccount;
14
15      public BAZOClient(BZPropertiesBean bazoProperties) {
16          this.bazoProperties = bazoProperties;
17          this.seed = Utils.hexToBytes(bazoProperties.getAccountSeed().substring(0, 64));
18          this.spec = EdDSANamedCurveTable.getByName(EdDSANamedCurveTable.ED_25519);
```

```
19          this.privKey = new EdDSAPrivateKeySpec(this.seed, this.spec);
20          this.pubKey = new EdDSAPublicKeySpec(this.privKey.getA(), this.spec);
21          this.rootAccount = Utils.hexToBytes(bazoProperties.getRootAddress());
22      }
23
24      public PublicKey getPubKey() {
25          return new EdDSAPublicKey(pubKey);
26      }
27
28      public PrivateKey getPrivKey() {
29          return new EdDSAPrivateKey(privKey);
30      }
31
32      public byte[] sign(byte[] msg, PrivateKey sKey) throws Error {
33          try {
34              Signature sgr = new EdDSAEngine(MessageDigest.getInstance(this.spec.
                    getHashAlgorithm()));
35              sgr.initSign(sKey);
36              sgr.setParameter(EdDSAEngine.ONE_SHOT_MODE);
37              sgr.update(msg);
38              return sgr.sign();
39          } catch (Exception e) {
40              System.out.println(e);
41              throw new Error(e);
42          }
43      }
44
45      public boolean verify(byte[] msg, byte[] signature, PublicKey vKey) {
46          try {
47              Signature sgr = new EdDSAEngine(MessageDigest.getInstance(this.spec.
                    getHashAlgorithm()));
48              sgr.initVerify(vKey);
49              sgr.setParameter(EdDSAEngine.ONE_SHOT_MODE);
50              sgr.update(msg);
51              return sgr.verify(signature);
52          } catch (Exception e) {
53              throw new Error(e);
54          }
55      }
56
57      public String sendTx(byte header, int amount, int fee, int txCnt, byte[] fromPubB,
            byte[] toPubB, byte[] data) {
58          byte[] arrayHeader = {header};
59          String headerTmp = Utils.bytesToHex(arrayHeader);
60          String amountTmp = Integer.toString(amount);
61          String feeTmp = Integer.toString(fee);
62          String txCntTmp = Integer.toString(txCnt);
63          String fromPub = Utils.bytesToHex(fromPubB);
64          String toPub = Utils.bytesToHex(toPubB);
65          String dataTmp = Utils.bytesToHex(data);
66          String params = headerTmp + "/" + amountTmp + "/" + feeTmp + "/" + txCntTmp + "/"
                + dataTmp + "/" + fromPub + "/" + toPub;
67          try {
68              StringBuffer response = this.getResponse("/createFundsTxWithData/" + params);
69              JSONObject jo = new JSONObject( response.toString());
70              JSONArray contentResponse = (JSONArray)jo.get("content");
71
72              return contentResponse.getJSONObject(0).get("detail").toString();
73          } catch (Exception e) {
74              throw new Error(e);
75          }
76      }
77
78      public StringBuffer getResponse(String endpoint) throws Exception {
79          try {
80              URL url = new URL(bazoProperties.getBazoHttpServiceUrl() + endpoint);
81              System.out.println("\nSending 'GET' request to URL : " + url);
82              HttpURLConnection con = (HttpURLConnection) url.openConnection();
83              con.setRequestMethod("POST");
84              int responseCode = con.getResponseCode();
85
```

```
86              BufferedReader in = new BufferedReader(
87                      new InputStreamReader(con.getInputStream()));
88              String inputLine;
89              StringBuffer response = new StringBuffer();
90
91              while ((inputLine = in.readLine()) != null) {
92                  response.append(inputLine);
93              }
94              in.close();
95              return response;
96          } catch (Exception e) {
97              throw new Error(e);
98          }
99      }
100
101     public void sendTxSigned(String txHash, PrivateKey privateKey) {
102         byte[] hash = Utils.hexToBytes(txHash);
103         byte[] signature = this.sign(hash, privateKey);
104         String signatureTmp = Utils.bytesToHex(signature);
105
106         String params = txHash + "/" + signatureTmp;
107         try {
108             StringBuffer response = this.getResponse("/sendFundsTx/" + params);
109             JSONObject jo = new JSONObject(response.toString());
110         } catch (Exception e) {}
111     }
112
113     public void sendProductHash(String productHash, EdDSAPrivateKeySpec privateKey, byte
        [] contractAddress){
114         EdDSAPublicKeySpec pubKey = new EdDSAPublicKeySpec(privateKey.getA(),
            EdDSANamedCurveTable.getByName(EdDSANamedCurveTable.ED_25519));
115
116         String txHash = this.sendTx((byte)0,0,1,0,pubKey.getA().toByteArray(),
            contractAddress,Utils.hexToBytes(productHash));
117
118         this.sendTxSigned(txHash,new EdDSAPrivateKey(privateKey));
119     }
120 }
```

Listing 3.18: Bazo client in the Food Chain Backend

The method that is going to be needed for sending the products hashes to Bazo blockchain is `sendProductHash(productHash, privateKey, contractAddress)`. It takes as input parameters the hash of the product, the private key of the Food Chain and the address of the smart contract which in our case is simply another account. As we can see in the class `ProductTagService.java`, this methods gets called inside the main method `addProductTag(CreateProductTagWsDTO productTag)`.

```
1   public ResponseEntity<?> addProductTag(CreateProductTagWsDTO productTag) {
2
3           Map<String, String> errors = productTagValidator.validateProductTag(productTag,
                productTagRepository, ethClient);
4
5           if (errors.size() == 0) {
6
7               ProductTag productTagModel = createProductTagWsDTOToProductTagModelConverter.
                    convert(productTag);
8
9
10              ProductTagWsDTO savedProductTagWsDTO = modelToProductTagWsDTOConverter.
                    convert(productTagRepository.save(productTagModel));
11
12              try {
13                  //just for testing since the client hasn't implemented an ethereum
14                  //client to send the data
15                  //Replace bazoClient.getPrivKeySpec() and getRootAccount() with
```

```
16                  //FoodChain.getPrivKeySpec and getRootAccount()
17                  //bazoClient.getPrivKeySpec() is an account for the foodchain whereas the
18                  //getRootAccount() acts as a Smart Contract even
19                  //though it is not. We send all the productHashes to that
20                  //account in form of data
21                  bazoClient.sendProductHash(savedProductTagWsDTO.getProductTagHash(),
                        bazoClient.getPrivKeySpec(), bazoClient.getRootAccount());
22                  ethClient.getFoodChainContract().addProductTagHash(savedProductTagWsDTO.
                        getProductTagHash()).sendAsync();
23              } catch (Exception e) {
24                  e.printStackTrace();
25              }
26
27              return new ResponseEntity<>(savedProductTagWsDTO, HttpStatus.CREATED);
28          } else {
29              return ResponseBodyGenerator.generateErrorResponseBody(FCConst.PPRODUCT_TAG,
                    Long.MIN_VALUE,
30                      OperationType.CREATE, HttpStatus.BAD_REQUEST, errors);
31          }
32      }
```

Listing 3.19: Changes to addProductTag method in the Food Chain Backend

The call simply copies what has already been done with the Ethereum Smart Contract but instead of sending it to that network we forward it to our Bazo Client.

3.2.12 Supply Chain Front-end adaptations

In this section we are going to explore how some parts, implemented for the mock ups design, can be useful for implementing the final version of the front-end.

Since we are using Ed25519 as a cryptography method, we have to find the right version compatible with the front-end, i.e. React-native, as well.
A library which perfectly suits our needs is named `react-native-tweetnacl` and can be found here [26]. The usage of the library is simple, similar to the one used in the ed25519 library for the TTN-plugin.

```
1  const nacl = require('tweetnacl-rn');
2
3  const keypair = {
4          publicKey: // Uint8Array with 32-byte public key
5          secretKey: // Uint8Array with 64-byte secret key
6      };
7
8  //Signing
9  const signature = nacl.sign(txHash, keypair.secretKey);
```

As we can see, it does not require any special pre-/post-processing. Important to mention is that the consumer who is going to buy a product, has to have a Bazo account as well. Since it has to sign transactions, the consumer has to have the public key as well as the private key stored on the phone.
We also need a way to know whether the transaction is valid or not. This means that we need to know whether the consumer has enough funds in its wallet. For this purpose, a similar approach taken in [2] can be taken. It consisted of a *Multisignature* (Multisig) server, which is in charge of verifying and signing transactions that are only signed in a light client, i.e. not by the network yet. This approach could lead to malicious behavior. For more information consult [2].

Chapter 4

Evaluation

In this chapter, we will try to evaluate the work done in this thesis. We will first have a look at both approaches taken with the transmission from LoRa node and then we will switch to the Bazo blockchain. There, we will discuss about the state of the Art before we changed it and the situation afterwards. More important, we will also take into consideration some security aspects of it.
At the end, we will have some comparisons and tests with other transaction types, in order to show whether the blockchain slows down with IoT transactions.

4.1 Transmission Approach 1 vs. Approach 2

The first approach consisted in sending signed data at the beginning and then split up the signature in multiple packets and transmit them over a longer period of time. Moreover, since we could actually send more data, we included non signed data with the split parts of the signature. This approach was actually used to check whether we could aggregate somewhere else what we have sent and reconstruct the same data structure on another device which runs on a different technology (Arduino vs. Raspberry Pi).
Using this approach confirmed that what we wanted to achieve was possible. We almost immediately realized that it was the wrong approach to implement in a blockchain. Not only the whole data sent was not signed and thus we would have a security problem that we couldn't afford but the overhead of the signature made the whole approach even less attractive. We couldn't implement a system that can only sign data smaller than the actual signature and also have data not signed at all. It wouldn't have been efficient in any sense.
Therefore, we came up with the idea used in the second approach, which provided much more advantages.
We send the public key only once and thus we spare our air time limited imposed by the LoRa newtork. Moreover, since all the data we send is signed, we do not waste it as well. We can have multiple sample/collections in the same period of time as the first approach with the benefits that everything is going to be verifiable.
We sign only at the end and not every single packet, therefore we even have a lower power

consumption. One of the biggest problem using the first approach, i.e the overhead, can by dynamically adjusted. If we wish to have less overhead, then we just send more data before signing it and our only constrain is dictated by the memory of the IoT devices, which in our case is an Arduino. Still, even an Arduino provides enough memory to have a more than acceptable overhead.

The only disadvantage we have so far using this approach is related to the time. Since we have to send more packets over a period of time and then sign them, we have always to wait for the completed aggregation on the TTN-plugin before being able to verify a transaction and issue it to the blockchain.
This problem arises mainly from all the limiting factors related to the LoRa Network.

4.2 Security Aspects

In this section, we will briefly discuss about some security aspects concerning the architecture and the single responsible parts. Because of some aspects, we had to take some design decision that could have been much different and maybe better if we wouldn't have taken into consideration the security of the whole system

4.2.1 IoT Wallet/Transaction

A private key is assigned to a LoRa Node and since the public key generated from it as to be known to the blockchain, we can know if it has been altered or not. Moreover, the account has to be created from another wallet and thus, the link to the real issuer would be saved as well. In order to avoid any manipulation from third party actors, some security methods might be implemented, such as burning the memory when a manipulation as been detected. We didn't focused on avoiding it since it is a out of scope topic for this thesis, but it is still worth mentioning it.

The whole IoT transaction is signed and not only the collected data. This means, that from a security perspective, everything that leaves our IoT devices and gets written into the blockchain can be verified. We managed to make small devices, such as Arduinos, able to issue transaction using a limiting protocol such as LoRa.

The only part of the whole system and architecture that we still cannot guarantee is the manipulation of the sensors connected to the device. Take as an example an IoT device which is used to collect sample of the temperature every 15 minutes. The interval of 15 minutes can be programmed into the devices and thus not altered, but the sensor in charge of taking the temperature can be still manipulated. The IoT device can only read what the sensor attached to it says and if someone manages to alter it, then the device will read other values that are different from the real ones.

4.2.2 TTN-Plugin & Acknowledgements

When a LoRa node sends a packet, it is received in the TTN-plugin. In order to be able to go to the next step, the TTN-plugin has to send an acknowledgement (Ack) to the node so that it knows that its message has been received. The LoRa node can only listen to those Ack's for a restricted and small amount of time when it sends something. As a consequence, sometimes the TTN-plugin recieves the correct packet, sends the right and corresponding Ack as well but the node does not receives it in that opened time window and therefore it resends the packet as if it were lost.
An idea would be to send the Ack directly from the gateway avoiding the required forwarding to TTN's server before sending it. This would have some problem in general and also some security problems.
First of all, a gateway that sends an Ack to the node might lose connection to the TTN server and therefore we will never receive that packet and the transaction would be invalid.
If we would do the aggregation on gateways, then it might be that another gateway receives the packet and sends the Ack to the node but the data, which has to be aggregated, is spread around multiple gateways. This would lead to a major problem of aggregating everything together between all the present gateways.

Lastly, if we rely on Acks from gateways, someone might hijack one of them and always send either fake Acks or send Acks without forwarding the packets. If other gateways haven't received the packet, we wouldn't be able to aggregate and issue the IoT transaction. The whole system might then not work anymore.
Because of this, we have to keep sending Acks from the TTN-plugin and accept that sometimes they are not correctly received. Implementing a method for being able to receive downlinks on a node without sending data would drastically improve the process but we don't know if it is possible using the LoRa protocol.

4.2.3 Bazo Client REST API

In this section, we are going to anayze two possible scenario that a malicious user can exploit to manipulate our transaction and even steal our funds when using the REST API. This two problems come from the *Multisig* server which has been discontinued in favour of a merged implementation with the client.
In order to issue an IoT Transaction the multisig server is not needed anyway since we do not need transactions to be signed by the receiver as well (the client).

Malicious Bazo Client

While using the REST API available on the Bazo Client, we found out that some security problems may arise.
When we want to send a `fund` transaction, what we actually do is sending the parameters that are going to be part of the transaction to the client and we are getting as a response

the hash (`txHash`) of the transaction that we have to sign and send back together with the newly generated signature. Let's assume that we are using a REST API of a client we do not know, this one could simply change every transaction we do by modifying the parameters we send and give us a hash to sign that corresponds to the malicious one. Whitout knowing that it does not correspond to our parameters, we will sign it and send it back. In doing so, a malicious client can steal all our funds or part of them. Therefore, we cannot assume that we can simply connect to a Bazo client to make our HTTP calls but we have to connect to our own client.

Man in the middle attack

Unluckily, this way of sending a transaction, getting a hash to sign and sending back the signature is prone to the **Man in the middle attack**[19]. Let's assume someone listens to our traffic, i.e to the transactions we send, gets them, manipulates them so that the receiver address of the transaction corresponds to its account and then forwards them to the client. The client as usual is going to give back the `hash`, the sender is going to sign it and send it to the client which is going to broadcast it to the miner. In doing so, the sender will unwittingly send funds to the attacker.
This attack can be avoided by signing the transaction directly and then send everything to the client. It might require a lot of work since the hash of the transaction can be different because the transaction is represented as an object and thus different compilers will return different hashes.

If we assume that we have something similar to a **Know Your Customer**[20] platform for our LoRa nodes, then the new implemented IoT transaction is not prone to any of those problems. Every transaction is signed on the node and modifying it at a later step would lead to an invalid signature and thus it won't be stored on the blockchain.
Thanks to the KYC platform, we could avoid the steps where we have to initialize the LoRa Node wallet and send some funds to their wallet because this can be done already in an earlier stage and not when the node sends its public key for the first time to the network.

4.2.4 Cryptography with ED25519 and RSA

In order to be able to process IoT transactions, we had to switch to Ed25519. The old ECDSA family of signature schemes is on a technical level a slower design than the EdDSA [21]. As state from Pradana in [22], Ed25519 offers better security and faster performance compared to ECDSA.
Unluckily, in our tests we didn't find any performance inprovement by switching to Ed25519. Probably, the old ECDSA was not a bottleneck for the BC and therefore didn't have any effect. We couldn't switch the commitment key to Ed25519 as well, but instead we hat to let it run based on RSA. Luckily, they can work together and we don't have any problem with the system. An exhaustive explanation to why the commitment key should stay on RSA can be found here[23].

4.2.5 Transaction Counter

A problem already presented in this thesis is the classical double spending attack, that has happened to several cryptocurrencies due to the 51% attack[24]. Double-spending implies a security flow in which it allows the same digital token to be spent more than once. With physical cash this is not possible, since it cannot be physically duplicated, but with digital tokens this is possible. It can even lead to inflation since it creates a new amount of token that was previously not present in the circulation[25].
In order to avoid this attack, the Bazo blockchain introduced a transaction counter (`txCnt` for funds transaction.

While implementing the new IoT transaction, we had a lot of thoughts regarding whether we need a counter as well or we could just leave it and ignore it. Introducing it would lead to many complications already explained in chapter 3.2.9 and since IoT transaction do not transfer any funds but they just have a transaction fee equals to one, we decided not to include the transaction counter check as a requirement for our new transaction.

4.3 LoRa limiting factors

One of the biggest challenge of this work was to exploit at most the LoRa Network. Every design choice had to be analyzed and made sure that it could work using LoRa.
Since we were able to connect LoRa nodes to a blockchain, we know now that using an advanced technology, such as LTE-m, is possible and probably would require much less hassle than what we have done so far. The whole TTN-plugin is designed to enable them to store data into the blockchain. Using a better technology would not have required such an aggregation and transactions could have been issued directly from a non-LoRa node.

We even had the possibility to test our scenario in a real environment and we successfully concluded that our design indeed works.

4.4 Tests & Comparisons

In this section we will have a look at how bazo performs with different loads of IoT transaction and we can compare it to traditional funds transaction as well.

In table 4.1 we show a test scenario we did by simulating different IoT transactions issued my multiple LoRa nodes. We started from simulating 400 up to 1600 nodes sending a single transaction where each node started issuing a transaction in a randomly distributed time between 0s and 120s. We run each test four times and we averaged the results from each one.
As we can see, the blockchain didn't have any issues processing that amount of IoT transaction sent over 2 minutes of time, i.e. the max Transaction Validation Duration is approximately at most the range of 120sec + 15sec. 15 seconds come from the block

validation time of a block of 15 sec. It can be higher because it can happen that the first transaction to arrive gets immediately validated alone in a block and thus adds 15 seconds.

Testrun	Transactions Sent	Avg #Blocks	Transaction Validation Duration	Avg Block Duration	Avg #Transactions in Block	TPS (Tx/sec)
1	400	12	137.98	11.5	33.33	2.9
2	800	7.5	95.14	12.69	106.67	8.41
3	1200	9	103.54	11.5	133.33	11.59
4	1600	7	107.71	15.39	228.57	14.86

Table 4.1: Testing IoT transaction in different scenarios

We also did a stress test which consisted in sending simultaneously as many transactions as possible to find out how many transaction per second the blockchain could validate. From table 4.2 we can see that we can approximately validate 50 transactions per second which corresponds pretty good with our funds transaction validation speed done (TPS) by K. Aydinly[8].

#Testrun	Transactions Sent	Total Blocks Mined	Transaction Validation Duration	Avg Block Duration	Avg #Transactions in Block	TPS (Tx/sec)
1	4058.25	6.75	49.84	12.06	601.22	49.84

Table 4.2: Stress test with IoT Transactions

To sum up, we can conclude that the new IoT transaction did not affect the performance of the Bazo blockchain.

4.5 Real-world Test Scenario

During this work, we had the possibility to do a real-world test with our LoRa nodes and gateways. The main goal was to check whether LoRa nodes placed inside were able to reach other TTN-gateways.
First, we had a setup like shown in picture 4.1. We were in a farm and we placed one gateway inside it, one outside and two LoRa nodes outside. They were successfully able to communicate, send data and their corresponding signatures to our gateways. They were all powered by power banks and none of them had problem running. The gateways were running on a RPi.

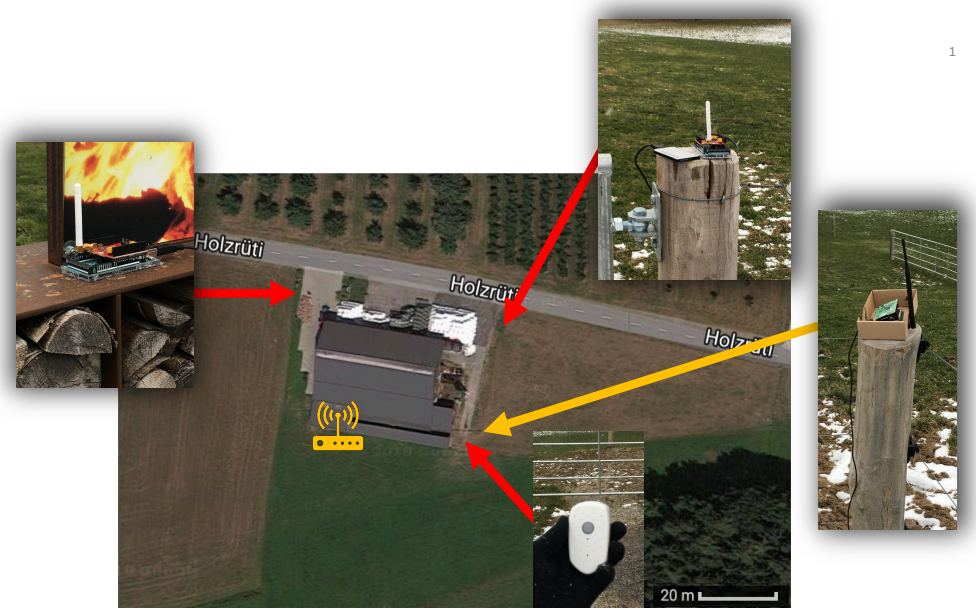


Figure 4.1: Real-world test with LoRa Nodes

We then checked whether they could reach other gateways beside ours. Unluckily, our nodes, even thought they were placed outside, weren't able to reach anything. We even checked on the TTN map were the next gateways were supposed to be placed. As we can see in figure 4.2, they were not even placed that far away, just some kilometers as the crow flies. TTN even claimed that some of them were placed outside and thus increasing the chance to reach them. The landscape of the region did not present any mountains but only some pretty small hills, which might have nevertheless influenced our results.

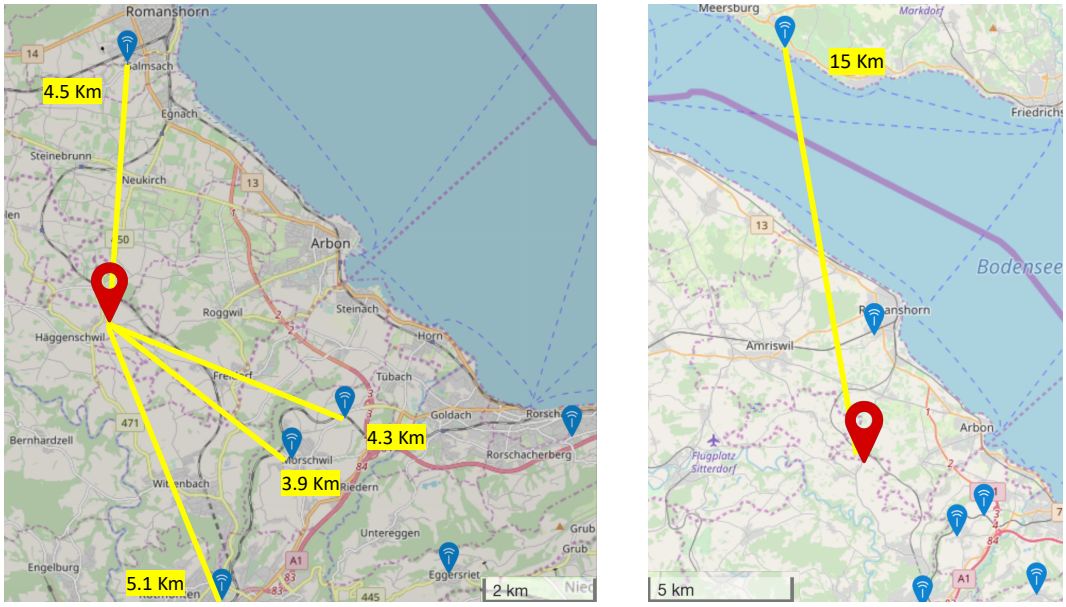


Figure 4.2: Location of LoRa Gateways in the TTN map

We had only one exception. At some point, we were able to reach a gateway quite far away from us. The gateway resulted to be placed in Germany, on the other side of the Lake Constance at circa 15 km of distance, which means with no big obstacles in between. We were actually able to send a packet to it but then we did not receive any Ack from our TTN-plugin. Therefore, we could not utilize it for our purpose.

Chapter 5

Summary and Conclusions

The main task of this thesis was to develop and implement an Hierarchical Payment System to bring fairness and ethics into a reality that sometimes lacks in both of them as well as in transparency.
Since Bazo BC is not specifically designed with the goal of addressing the requirements for a hierarchical payment system, the main focus of the thesis shifted in designing and developing additional components to Bazo, so that the basic requirements of the supply chain could be gained in Bazo as well. This pattern can be clearly seen in the thesis. So many additional components and adaptations have been proposed in order to accommodate the Supply Chain with IoT devices.
Throughout the work, many different technologies and programming languages have been used. This reflects the idea that heterogeneous systems, such as a blockchain and IoT devices, can work together, no matter which technology they use or which programming language they are written in.

Correctly connecting the Supply Chain Platform was not only almost impossible due to lack of Smart Contracts in Bazo, but also the back-end and front-end of the platform were not ready and fully implemented. Some important design modifications had to be taken and implementing them while the base structure was not ready would have brought more troubles than solutions. Therefore, we decided to approach the task in a more heuristic way by providing details on how the implementation should look like and by giving designs of the front-end once the right endpoints and logic in the back-end has been implemented.

To conclude, we can say that the overall architecture for IoT devices works very good, as seen in the experiments as well, even when using limiting technologies. The new design of the Supply Chain front-end is a good mock up of how the platform could look like in a not so far future.

Bibliography

[1] Sanjiv Subodhnarayan Jha, "Design and Implementation of an Integrated Water Quality Monitoring System and Blockchains", January 25th, 2018.

[2] Marc-Alain Chételat, "Design and Prototypical Implementation of a Mobile Light Client for the Bazo Blockchain", March 8th, 2018.

[3] Blum Roman, "Cryptographic Sortition for Proof of Stake in Bazo", April 27th, 2018.

[4] Arduino Store, `https://store.arduino.cc/arduino-mega-2560-rev3`, accessed February 2019

[5] LoRa Shield for Arduino, `http://www.dragino.com/products/module/item/102-lora-shield.html`, accessed February 2019

[6] Ed25519 Go package, `https://godoc.org/golang.org/x/crypto/ed25519`, accessed March 2019

[7] CayenneLPP support for TTN, `https://www.thethingsnetwork.org/docs/devices/arduino/api/cayennelpp.html`, accessed March 2019

[8] Aydinly Kürsat, "Design and Implementation of a scalable IoT-based Blockchain", March, 2019.

[9] Bazo Blockchain Wiki, `https://github.com/bazo-blockchain/bazo-smartcontract/wiki/Introductionl`, accessed March 2019

[10] Arduino-Lmic library, `https://github.com/matthijskooijman/arduino-lmic`, accessed January 2019

[11] A.G. Nabi, D. Dordevic, T. Mann, "Blockchain-based Supply Chain Tracking Platform, Zurich, Switzerland", February 15th, 2019.

[12] Ed25519 Library for Arduino, `https://rweather.github.io/arduinolibs/classEd25519.htm`, accessed February 2019

[13] Ed25519 Library for Node.js, `https://github.com/dazoe/ed25519`, accessed April 2019

[14] Blockchain technology for supply chains – A must or a maybe?, `https://www.mckinsey.com/business-functions/operations/our-insights/blockchain-technology-for-supply-chainsa-must-or-a-maybe`,accessed March 2019

[15] Novo, O. (2018). Blockchain Meets IoT: An Architecture for Scalable Access Management in IoT. IEEE Internet of Things Journal, 5, 1184-1195, https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8306880, accessed March 2019

[16] F. Restuccia, S. D'oro et all, IEEE INTERNET OF THINS JOURNAL, VOL.1, NO.1, JANUARY 2018, Blockchain for the Internet of Things: Present and future, http://www.ece.neu.edu/wineslab/papers/Restuccia2018IoTJ.pdf, accessed March 2019

[17] Samwel, Batina, Bertoni, Breaking Ed25519 in WolfSSL, https://eprint.iacr.org/2017/985.pdf, accessed March 2019

[18] Definition of a structure in Arduino, http://playground.arduino.cc/code/struct, accessed March 2019

[19] Man in the middle Attack, https://en.wikipedia.org/wiki/Man-in-the-middle_attack, accessed March 2019

[20] Definition of a Know Your Customer platform, https://en.wikipedia.org/wiki/Know_your_customer, accessed March 2019

[21] ECDSA, EdDSA and Ed25519 relationship/compatibility, https://crypto.stackexchange.com/questions/58380/ecdsa-eddsa-and-ed25519-relationship-compatibility, accessed March 2019

[22] Risan Bagja Pradana, Upgrade Your SSH Key to Ed25519, https://medium.com/risan/upgrade-your-ssh-key-to-ed25519-c6e8d60d3c54, accessed March 2019

[23] I. Damgard and J. B. Nielse: Commitment SChemes and Zero-Knowledge Protocols (2011), Aarhus University, last accessed: 24.03.2019, https://medium.com/risan/upgrade-your-ssh-key-to-ed25519-c6e8d60d3c54, accessed March 2019

[24] Definition of 51% attack, https://www.investopedia.com/terms/1/51-attack.asp, accessed March 2019

[25] Explanation of double-spending attack, https://en.wikipedia.org/wiki/Double-spending, accessed March 2019

[26] TweetNaCl library for React-native, https://www.npmjs.com/package/react-native-tweetnacl

Abbreviations

Ack	Acknowledgement
API	Application Programming Interface
BC	Blockchain
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
Ed25519	Edwards-Curve Digital Signature Algorithm
HPS	Hierarchical Payment System
IoT	Internet of Things
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
KYC	Know Your Customers
LoRa	Long Range
LoRaWAN	Long Range Wide Area Network
Multisig	Multisignature
NPM	Node Package Manager
P2P	Peer to peer
PoS	Proof-of-stake
PoW	Proof-of-Work
REST	Representational State Transfer
SC	Smart Contract
SDK	Software Development Kit
TTN	TheThingsNetwork
tx	Transaction
txCnt	Transaction Counter
txFee	Transaction Fee

Glossary

Seed A seed is a number or a vector used to initialize a pseudorandom number generator. When a pseudorandom number generator is reinitialized with the same seed, it will produce the same sequence of numbers and thus, the same keypair of public and private key.

Multisignature Multisignature might require transactions to be signed and thus authorized from multiple private keys. In doing so, responsibility for a transaction is divided between more parties.

List of Figures

2.1 Decentralized Access Control System [15] 5

2.2 Container Diagram for Food Chain Tracking System from [11] 7

2.3 Raspberry Pi 3 8

3.1 Agnostic IoT architecture 12

3.2 LoRa HW Wallet for a full IoT Transaction 13

3.3 Adapted Container Diagram for Food Chain Tracking System connected to Bazo from [11] 14

3.4 Adapted ER Diagram for Food Chain Tracking System from [11] 16

3.5 Home Screen & Scanned Products 17

3.6 Map Overview & Product Details 18

3.7 Buying a Product with HPS Transactions 18

3.8 Data Flow Design with LoRa Nodes and Network 19

3.9 Data transfer diagram with overhead 21

3.10 Data transfer diagram with aggregation 22

3.11 TTN Console 24

3.12 Data Flow and instantiation of an IoT Wallet on the blockchain. 34

3.13 State of the blockchain 36

3.14 Mined block containing an IoT Transaction. 44

4.1 Real-world test with LoRa Nodes 55

4.2 Location of LoRa Gateways in the TTN map 56

List of Tables

2.1 Arduino Uno vs Mega 2560 8

4.1 Testing IoT transaction in different scenarios 54

4.2 Stress test with IoT Transactions 54

Listings

3.1 Configuration for a LoRa Node 23
3.2 Configurations variables for a LoRa Node data collection and transmission 25
3.3 State flags on a LoRa Node 25
3.4 Sign a message on the node 28
3.5 Verify a signature in Node.js 28
3.6 Generate a Key-Pair with Ed25519 29
3.7 New Endpoints Bazo Client REST API 32
3.8 New createAccTxIoT Body Endpoint 32
3.9 New createFundsTxIoT Body Endpoint 32
3.10 New sendTxIoT Body Endpoint 33
3.11 Example of required IoT Wallet variables for Arduino devices 35
3.12 JSON response from the Bazo client when querying an account 36
3.13 Hashing a public key on arduino 37
3.14 Hashing and verifying an IoT transaction in the TTN-plugin 38
3.15 IotTx Data Struct in miner/protocol/iottx.go 39
3.16 Process and Broadcast of an IoT Transaction 39
3.17 Bazo Properties in the Food Chain Backend 44
3.18 Bazo client in the Food Chain Backend 45
3.19 Changes to addProductTag method in the Food Chain Backend 47

Appendix A

Installation Guidelines

A.1 Prerequisites

In order to be able to follow this guidelines, some prerequisites are necessary. First of all, if a Raspberry Pi 3 has to be used as a LoRa gateway, this has to be already installed and running, the installation procedure is not covered here. Moreover, installing an operating system on it is not going to be explained as well. If the gateway does not have to run on the RPi, then an external one connected to TTN will be enough. Git has to be installed as well in order to be able to download the following projects.
The RPi was running on Raspbian GNU/Linux 4.9.59-v7+

A.2 LoRa Node Application

In order to run the sketch on an arduino, a LoRa Shield has to be connected to the board. The Arduino application on the computer is required as well to upload the sketch to the board. The sketch is contained in the same git repository as for the TTN-plugin, in order to get it either download the the repo as a zip file or open a terminal and write:

```
git clone https://github.com/ilecipi/TTN-plugin.git
```

Next, navigate into the root folder of the project, then navigate in the folder `arduino` and there you will find the file `data_aggregation.ino`.
Open the file using the official Arduino Ide and insert your TTN credentials like explained in section 3.2.4. Make sure to install all the required libraries which can be found in the CD as well. At this point you are ready to upload the sketch using an USB cable connected to the arduino. Once uploaded, there is no required step for the LoRa Node. Just make sure that the node can actually communicate with a gateway and thus to TTN.

A.3 TTN-plugin Application

Since you have already downloaded the project for the Arduino sketch, you can navigate in the root folder and install the required dependencies.
First, make sure to have the latest version of Node compatible with your raspberry pi, you can download it here:

```
https://nodejs.org/en/download/
```

Then, you have to install the node package manager (NPM) as well. A guide can be found here:

```
https://www.npmjs.com/package/raspberry
```

At this point, you are ready to install the required dependencies for the project, to do it run in the terminal the following command in the root directory:

```
npm install
```

It might be that some problem arises, a common solution is to run:

```
npm rebuild
```

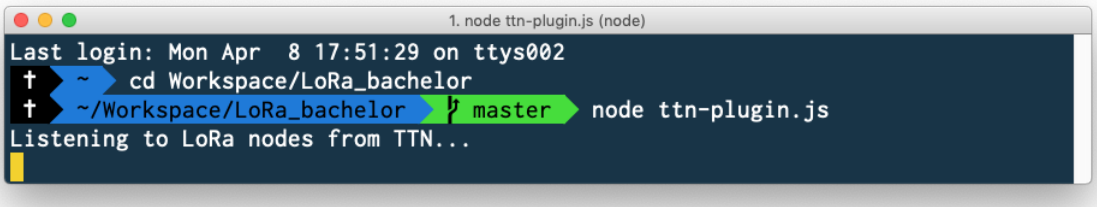
and then re run:

```
npm install
```

After having done the changes to the configurations like explained in section 3.2.5, you can now start the application by writing:

```
node ttn-plugin.js
```

If everything went has expected, i.e. the dependencies are correctly installed and the TTN configurations are the right one as well, you should see something like this on the console:



When a node sends data to TTN, you will see the information on the terminal as well. A logger is implemented as well, in the root folder you can find a directory called `log` which is created after starting the project for the first time. There you can find different logs. One called `ALL.log` which contains the logs from every device and for each single device there is a separate file, named `<device_name>.log`.

A.4 Bazo Miner

Getting the correct version of the miner is a little bit tricky since it hasn't been merged into the official repository.
First, make sure to install the latest version of *GO* in the RPi. Then you can download the Bazo Miner by typing in the terminal:

```
go get github.com/ilecipi/bazo-miner
```

The Miner is still connected to the official repository, you have to change the remote link of the git repository. Type in the terminal:

```
git remote set-url origin https://github.com/ilecipi/bazo-miner
```

Now you can download the required version by typing:

```
git pull
```

Switch to the current and most updated branch by typing:

```
git checkout "ed25519"
```

Now, build the project and let the miner run with:

```
./bazo-miner start –database StoreA.db –address localhost:8000 –bootstrap localhost:8000 –wallet WalletA.txt –commitment CommitmentA.txt –multisig WalletA.txt –rootwallet WalletA.txt –rootcommitment CommitmentA.txt
```

At this point, remember to copy the file `"WalletA.txt"` and insert it into the root folder of the TTN-plugin.

A.5 Bazo Client

In order to run the client, the same steps of the miner have to be followed. Start by downloading the project:

```
go get github.com/ilecipi/bazo-client
```

Move inside the project's folder and change the url of the git repo by typing in the terminal:

```
git remote set-url origin https://github.com/ilecipi/bazo-client
```

Download the right version:

```
git pull
```

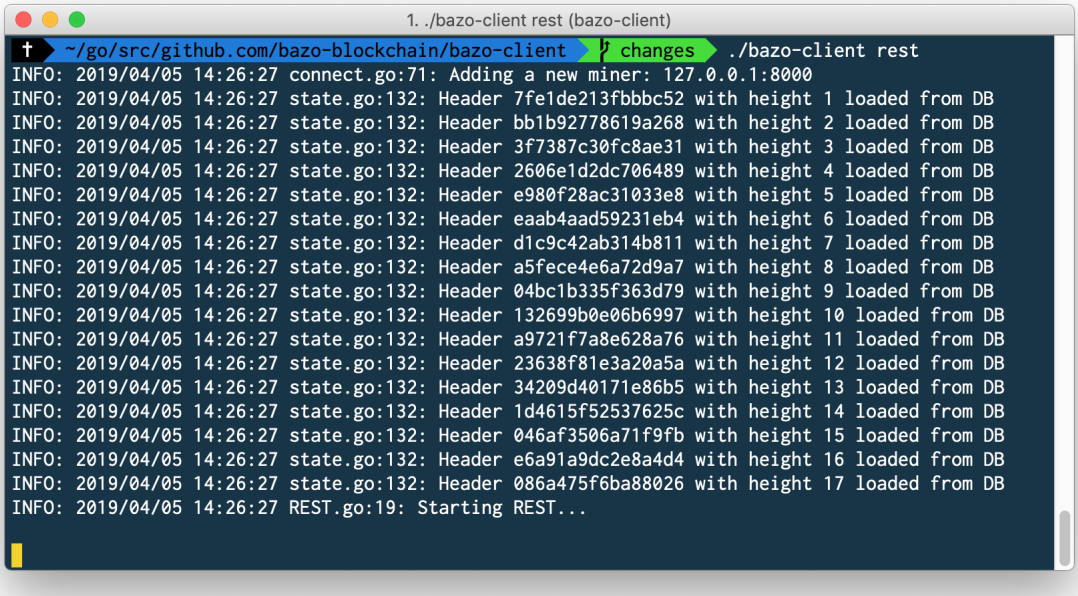
Change the branch:

```
git checkout "ed25519"
```

Build the project and start the REST interface, first make sure to have a local bazo-miner running:

```
./bazo-client rest
```

The result should be the following:



A.6 Supply Chain Monitoring Platform Back-end

The repository for the supply chain platform is private, therefore before being able to download it you have to get the permissions, once you have them download it as follows:

```
git clone https://github.com/ilecipi/food-chain-platform.git
```

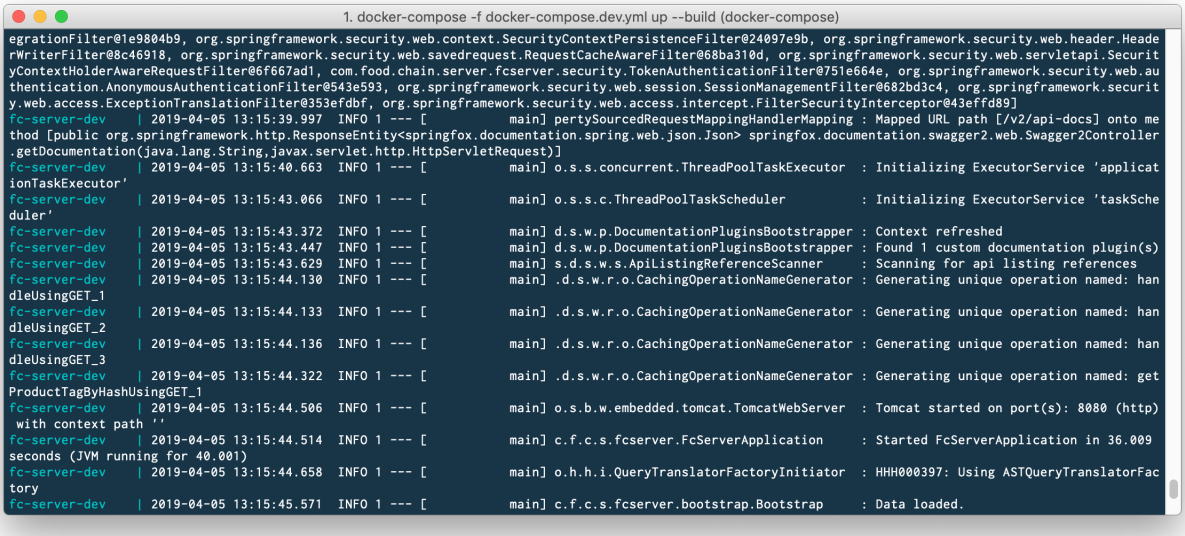
Change the current branch to the update one:

```
git checkout "backend_bazo"
```

Change the configuration of the backend, i.e. write the correct IP of the Bazo miner like explained in section 3.2.11.
Now, be sure to have Docker installed and let it run in the background. Follow the instructions contained in [11] for a better explanation. Once everything is installed, let everything run with the following comand:

```
docker-compose -f docker-compose.dev.yml up –build
```

If the projects correctly starts, the result on the terminal should be the following:



Now, you can use the Supply Chain Platform as usual and when new products are added, the hash of the product will be stored on Bazo.

A.7 Supply Chain Monitoring Platform Front-end

The new design of how the Supply Chain Monitoring Platform should look like, can be found in git as well, but it also need permission in order to be accessible:

```
git clone https://github.com/ilecipi/food-chain-native-client.git
```

Change the branch to the updated one:

```
git checkout ”HPS”
```

Make sure to have React-native installed and Expo as well. To install them first you should install NPM, like explained above for the TTN-plugin and then run:

```
npm install react-native-cli
npm install expo-cli global
npm install
```

In doing so, all the dependencies should be installed if no errors arise, then simply type:

```
expo start
```

Now you can access the app with your mobile phone thanks to the Expo App.

Appendix B

Contents of the CD

Source Code for the Arduino Sketch of a LoRa node

Arduino libraries

Source Code of the TTN-plugin for LoRa nodes

Bazo Miner IoT enabled Source Code

Bazo Client IoT enabled Source Code

Supply Chain Monitoring Back-end with connection to Bazo Source Code (Docker)

Source Code for the Supply Chain Monitoring Front-end with mock ups

LaTex Source Code

Final Thesis (PDF)

Intermediate Presentation