



**University of
Zurich** ^{UZH}

Performance Evaluation of LTE-M NBloT in IoT to Blockchain Transmissions

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

Supervisor: Sina Rafati, Eryk Schiller
Date of Submission: August 20, 2019

Abstract

Eine wachsende Anzahl von "Internet of Things" (IoT) - Geräten wird kompakter und energieeffizienter. In meiner Bachelor Arbeit habe ich aufgezeigt, was mit energieeffizienten Geräten möglich ist, die mit der Low Power Wide Area Network (LPWAN) und mit der Bazo Blockchain arbeiten. In dieser Arbeit fokussiere ich Geräte, die auf der populären "Long Term Evolution" (LTE) Technologie basieren, aber für IoT Anwendungen adaptiert sind, wie zum Beispiel LTE-m. In einem ersten Schritt gebe ich einen Überblick über die IoT-Blockchain Architektur. In einem zweiten Schritt zeige ich auf, wie IoT-Geräte in die bereits existierende Bazo Blockchain integriert werden können. Schliesslich vergleiche ich die LoRa mit der LTE-m Architektur, mit einem Fokus auf Energieeffizienz und Daten Overhead.

An increasing number of devices belonging to the domain of the "Internet of Things" (IoT) is becoming more compact and energy efficient. In my bachelor thesis, I have shown what is possible to do with energy efficient devices working with the Low Power Wide Area Network (LPWAN) and the Bazo blockchain. In this study, I am targeting devices that are based on the more popular technology "Long Term Evolution" (LTE) but adapted to IoT applications, i.e. LTE-m. In a first step I am giving an overview of the IoT-Blockchain architecture, in a second step I am showing the process of integrating those IoT devices in the already existing Bazo Blockchain. At the end, I compare the LoRa with the LTE-m architecture focussing on power efficiency and data overhead.

Contents

Abstract	i
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	1
1.3 Work Outline	2
2 Related Work and Background	3
2.1 Internet of Things and the Bazo Blockchain	3
2.1.1 Bazo Client	4
2.1.2 Arduino Cellular IoT Application Shield	4
3 Design and Implementation	5
3.1 Design	5
3.1.1 IoT Architecture	5
3.1.2 IoT Transaction overview	7
3.2 Implementation	7
3.2.1 Mandatory adaptations in the Sixfab CellularIoT Shield library . .	7
3.2.2 Arduino Sketch for LTE-m Nodes	11
3.2.3 UDP packet listener on Bazo Client	12

4	Evaluation	17
4.1	Security Aspects	17
4.2	Arduino LTE-m CellularIoT Shield	17
4.3	Energy efficiency of LTE-m	18
5	Summary and Conclusions	21
	Bibliography	23
	Abbreviations	25
	List of Figures	25
	List of Tables	27
	Listings	29
A	Installation Guidelines	33
A.1	Prerequisites	33
A.2	LTE-m Arduino sketch	33
A.3	Bazo Miner	34
A.4	Bazo Client	35

Chapter 1

Introduction

1.1 Motivation

Nowadays, many transmission standard are addressing the IoT market with different approaches. We have the Long Term Evolution (LTE) cat. M/M1, the enhanced Narrow Band IoT (NB-IoT) and the Extended Coverage GSM IoT (EC-GSM-IoT). Every standard has its advantages and disadvantages.

NB-IoT is an alternative that, thanks to reduced complexity, has a lower cost at the expense of decreasing data rate. Indeed, the bandwidth is up to 200 kbps in both directions. The involvement of the mobile operator in the process of developing private IoT networks is required when dealing with Cellular IoT. The bands on which those devices operate are actually licensed bands.

1.2 Description of Work

In this independent study, the first goal is to allow IoT devices, which use LTE-M, to send data to our Bazo blockchain. The version of the Bazo blockchain we are using for this study already supports IoT transactions and therefore it does not have to be adapted.

A network of simple topology is going to be implemented with only one LTE-M node provided with a SIM (Subscriber Identity Module) card.

This node is going to directly send data to the Bazo Client using either the Transmission Controll Protocoll (TCP) or the User Datagram Protocol (UDP).

On the Bazo client side, a listener will be implemented in order to allow the communication between the two points.

The second task of the study is going to be the energy efficiency evaluation of LTE-M with respect to the version implemented using LoRa nodes.

The analysis is going to be based on the signed packets sent to the Bazo blockchain. Moreover, an efficiency measure in terms of data overhead, energy consumption and transmission mechanism implementation is going to be provided.

Overall, the IoT adaptation architecture has to be appropriately modified to acknowledge a different protocol stack of the LTE-M network, i.e. IoT devices using the IP stack to transport information instead of the LoRaWAN-based TTN network.

1.3 Work Outline

In Chapter two an introduction to the previously implemented IoT architecture is given. Required hardware component utilized are also going to be part of that chapter.

Chapter three contains two main parts, the first one shows the designs of all the main system components analyzed and developed during this study and compared to the LoRa architecture. The second one contains the implementations that had to be done. Not only to create the whole system but also to improve the existing libraries provided by the producer of the Arduino CellularIoT Shield.

Chapter four contains the evaluation part of this study. The most important part of this chapter are going to be the energy efficiency compared to the LoRa architecture and the data overhead. Chapter five summarizes and concludes the report.

Chapter 2

Related Work and Background

This section is going to provide an overview of an existing IoT approach with a blockchain and this study is going to be based specifically on it. A classification of used and existing hardware, as well as communication systems useful for this study, is going to be provided

2.1 Internet of Things and the Bazo Blockchain

As a starting point, we are going to take the bachelor thesis based on LoRa nodes which can issue IoT transaction to the Bazo Blockchain [9].

Bazo is a blockchain being currently under develop at the University of Zurich and in collaboration with the "HSR Hochschule für Technik Rapperswil". It is written in the programming language "GO" and based on Proof-of-Stake [10]. It has been adapted to accomodate IoT transactions and in order to accomplish that, the cryptographic function used was changed in a more light-weight and IoT suitable Elliptic Curve Digital Signature Algorithm (ECDSA), namely Ed25519 [3]. This new cryptographic function is also compatible with the arduino cryptography library [2].

Because of the limiting factor of the LoRa technology explained here [9], the already implemented version needed a management HUB, called TTN-Plugin. The TTN-plugin is in charge of coordinating the incomplete transactions coming from each node and reconstruct them. Once reconstructed and validated, it forwards them to the Bazo Client which is then going to broadcast each transaction to the Miners of the network. In order to reduced the overhead of the signature and the actual data collected, the TTN-plugin can also aggregate multiple packets and thus optimize the whole architecture.

That approach works well for nodes based on the LoRa protocol, since it allows such a restricting technology to issue IoT transaction. In this study, we are going to issue a full IoT transaction without any intermediary, i.e. there will not be any management HUB to help us in achieving this and to do that, we are going to use a more powerful technology, namely LTE-m.

2.1.1 Bazo Client

A fundamental part of this study is going to be the Bazo Client. Since the Bazo Miner supports IoT transactions and communicates just between miners and Bazo clients, no change or implementation is going to be required. As for the Bazo Client, we will have to introduce the possibility to gather data from a LTE-m enabled node. We are not going to use the already implemented Representational State Transfer (REST) Application programming interface (API). Since we are going to implement everything over the UDP protocol, we will need to implement a UDP packet listener and integrate it on the client's side, so that it can simultaneously serve LoRa nodes as well as LTE-m nodes.

2.1.2 Arduino Cellular IoT Application Shield

Together with an Arduino Mega 2560 [5], we couple an LTE IoT Shield. Thanks to the expanded memory of the Arduino Mega 2560, we can manage and process all IoT Transactions without many hassles and thanks to the IoT Shield we are able to send data through the IP stack.

The Cellular IoT Shield [6] has combined LTE technology Cat.M1, Cat.NB1 (Nb-IoT) and eGPRS. This shield is based on the Quectel's BG96 Module [7], which offers a maximum data rate of 375Kbps downlink and 375Kbps uplink. Moreover, it features global frequency bands and ultra-low power consumption.

Some of the many key applications of this shield are in smart farming/cities sensor, Internet of Things sensors, agricultural and environmental monitoring and many others.

Chapter 3

Design and Implementation

3.1 Design

This section explains the design decision taken to enable the communication between LTE-m nodes and the Bazo client. We will explain how IoT devices communicate to the blockchain and we will also show a specific overview of how a full IoT transaction can be issued from a LTE Node.

3.1.1 IoT Architecture

Figure 3.1 is taken from our previous study [9] and adapted for LTE-m nodes. It actually shows the agnostic overview of the IoT Architecture mainly developed for LoRa nodes. The system is designed to be compatible across most common system without requiring any special adaptations. Indeed, if we compare both version of it, we will see that there is only one small difference.

As before, we have an IoT Device, in this case based on LTE-m, which has a Blockchain Wallet that enables IoT transactions to be issued to the Bazo blockchain. An edge node (Bazo client), placed on the other side as a receiver, gets the transactions before they can reach the Bazo blockchain and thus the miners.

Unlike in our previous version, we do not have a management hub which is responsible for coordinating all IoT devices across the network before being able to forward their transactions.

Indeed, our LTE-m nodes are able to issue full IoT transactions directly to a Bazo client and thus removing the necessity of any intermediary.

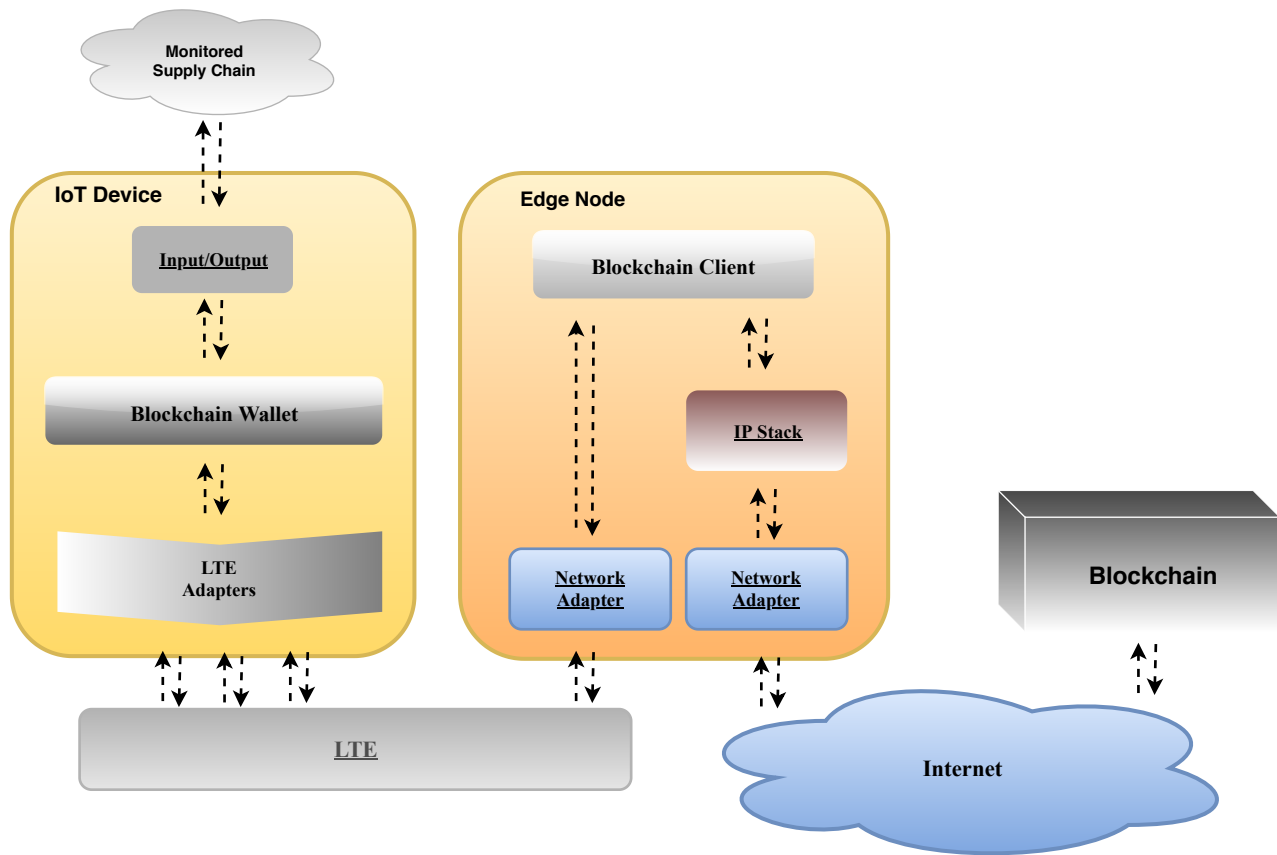


Figure 3.1: Agnostic IoT architecture adapted from [9]

3.1.2 IoT Transaction overview

In this brief section, we are going to have a look at what part are being transmitted when sending an IoT transaction.

Figure 3.2 shows which data is actually sent. If we compare this version with the architecture used with LoRa we can notice that the transmission pattern is completely different. With LoRa, we had to send only as few data as possible. This implied sending only a public key, split the signature of the data into two packets and split the data as well into multiple packets. Thanks to the much more powerful technology LTE-m, we are able to issue a single IoT transaction with all the required parameters. As opposed to the LoRa version where the transaction fee and counter had to be static, here we can actually change them and have them dynamic as well.

We are able to send all the required fields and thus even improve security aspects linked to the double spending attack [8], solved thanks to a transaction counter (**TxCnt**).

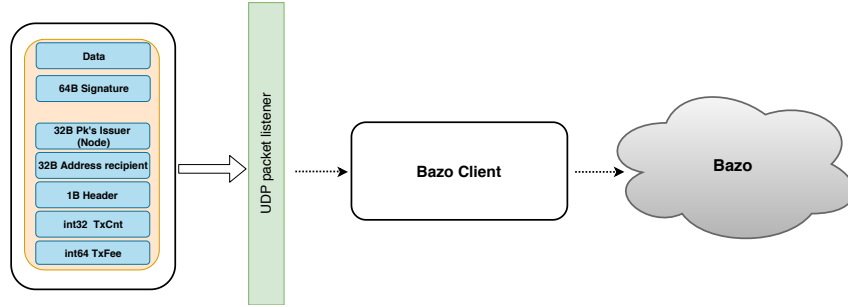


Figure 3.2: LTE-m HW Wallet for a full IoT Transaction

3.2 Implementation

This section shows the main implementation's steps taken to enable the communication between LTE-m nodes and the Bazo client. We will explain how IoT devices communicate to the blockchain through the protocol UDP and we will also show required changes in the official Sixfab Arduino CellularIoT library to allow a correct transmission of the data.

3.2.1 Mandatory adaptations in the Sixfab CellularIoT Shield library

Similar to the LMIC library [1] used for LoRa nodes, we have had encountered many problems when trying to send more data. We experienced two different behaviors; either the node will crash or freeze and a hard reboot was required or when sending data, not the whole data was actually sent. In this section, we are going to explain those issues and show how to fix them.

The first problem with the device freezing or rebooting was solved by modifying the official Sixfab library which can be found on github here [11].

First of all, we have to open the file named `Sixfab_CellularIoT.cpp`. At line 59 we can see that some constants are defined. These constants are actually too small when sending more than around 115 Bytes and the devices will crash or freeze. Therefore, we can modify them as follows:

```

1 // Constants
2 #define TIMEOUT 3000
3 #define IP_ADDRESS_LEN 30
4 #define DOMAIN_NAME_LEN 50
5 #define PORT_NUMBER_LEN 8
6 #define AT_COMM_LEN 2000
7 #define AT_RESPONSE_LEN 200
8 #define DATA_COMPOSE_LEN 200
9 #define DATA_LEN_LEN 200

```

Listing 3.1: Constant for Sixfab Cellular IoT library

Moreover, not only those have to be changed, but if we scroll down to line 640, we can see that the variables `compose` and `url` are also constrained.

`Compose` is used when wanting to send AT commands and `url` is used when wanting to send either a POST or GET request.

In our case, since we are using the protocol UDP, we need to change the variable `compose` so that we are able to send more data.

While working on solving this issues, we tried to execute HTTP requests, that's why we even modified the variable `url`.

```

1 private:
2     char compose[999];
3     char url[200];

```

Listing 3.2: Private value for Sixfab Cellular IoT library

After modifying the library and being able to send as much data as by us required, the node kept freezing. After some debugging, we discovered that the node was not actually freezing, it was just stuck in an infinite while loop.

Now, let's have a look at the functions we are using to try to explain the while loop.

First of all, we have to open the file named `Sixfab_CellularIoT.h`. There, we can find the function named `sendDataUDP(const char *data)`.

```

1 // fuction for sending data via udp.
2 void SixfabCellularIoT::sendDataUDP(const char *data)
3 {
4     clear_compose();
5     char data_len[10];
6     sprintf(data_len, "%d", strlen(data));
7
8     strcpy(compose, "AT+QISEND=1,");

```

```

9   strcat(compose, data_len);
10  strcat(compose, ",\\");
11  strcat(compose, ip_address);
12  strcat(compose, "\\");
13  strcat(compose, port_number);
14
15  sendATComm(compose, ">");
16  sendDataComm(data, "OK\\r\\n");
17
18  clear_compose();
19 }

```

Listing 3.3: Send data UDP function in the Sixfab CellularIoT library

In this function, we have done a small modification:

Instead of calling twice `sendAtComm`, we call `sendDataComm` when we do not want to send a command but we want to actually send data.

Now, let's move to the `sendDataComm` function and see where the real problem is.

```

1  // function for sending data to BG96_AT.
2  const char* SixfabCellularIoT::sendDataComm(const char *command, const char *
      desired_reponse)
3  {
4      uint32_t timer;
5      char response[AT_RESPONSE_LEN]; // module response for AT commands.
6      memset(response, 0 , AT_RESPONSE_LEN);
7      BG96_AT.flush();
8
9      BG96_AT.print(command);
10
11     timer = millis();
12     while(true){
13         if(millis()-timer > 20000){
14             BG96_AT.print(command);
15             timer = millis();
16         }
17         char c;
18         int i = 0;
19
20         while(BG96_AT.available()){
21             c = BG96_AT.read();
22             DEBUG.write(c);
23             response[i++] = c;
24             delay(5);
25         }
26         if(strstr(response, desired_reponse)){
27             return response;
28             memset(response, 0 , strlen(response));
29             break;
30         }
31     }
32 }

```

Listing 3.4: Send data command in Sixfab CellularIoT library

As we can see, in this code snippet above at line 12 there is a `while(true)` loop that should actually never be used. The only way to break this loop is to get into the `if`

condition at **line 16**. The problem arises when sending too much data: The response is actually bugged and we will never be able to enter into it and thus break the loop.

The solution is quite simple: we just change the condition in the **while** loop and put a timeout as follows:

```

1 while(millis()-timer < 20000){
2     ...
3     ...
4 }
```

Listing 3.5: Avoid an infinite while loop in the Sixfab CellularIoT library

Only at this point, we are able to send much more data through the CellularIoT Shield from Sixfab.

The problem regarding the loss of data while sending it, is still related to the library but we were not able to change it. Instead, we found a workaround which works perfectly fine (without any loss of data) and has very little overhead.

The strange behavior appears when sending raw bytes and more precisely, when the data we send contains a byte equals to 0. Instead of sending the whole buffer of bytes, the data is split right where the 0 is and the rest does not get sent.

The 0 gets interpreted as an escape sequence, i.e. as the terminating character. Moreover, some weird behavior start happening with the Shield, i.e. some random data is sent right after it.

A solution is to save how many 0's our data contains, save their position in the data somewhere else and replace them with their index in the array. When sending the data, append the number of 0's, their positions and the modified data. Then, on the receiver side, just replace everything with the following algorithm implemented in javascript:

```

1 let nrOfZeros = rawData[0];
2 let zerosPosition = rawData.slice(1, nrOfZeros);
3
4 let overflow = 0;
5 let previousByte = zerosPosition[0];
6 for(let i = 0; i < zerosPosition.length; i++){
7     let tmpByte = zerosPosition[i];
8     if(previousByte > tmpByte){
9         overflow++;
10    }
11    previousByte = tmpByte;
12    const zeroIndex = tmpByte + (255 * overflow)
13    rawData[zeroIndex] = 0;
14 }
```

Listing 3.6: Reconstruct the received data with 0's

We take the number of zeros we know we will have, we extract the array based on that number and then we simply loop into it and we overwrite the value with **rawData[zeroIndex] = 0**.

The **overflow** variable is required when we send from the node data which is longer than 256 Bytes. This is necessary when there is a 0 value after the 255th position, since the node will not send any number bigger than 255 but it will just start again from 0. Therefore, we simply check the array and if we notice that we went back from 255, we increase

our overflow and so we are able to extract the correct index greater than 255.

This method works, but it's actually really cumbersome to manage when sending different variables and the intention is to extract each of them.

Therefore, we came up with another method easier to implement and less error prone. We are going to explain it in the next section.

3.2.2 Arduino Sketch for LTE-m Nodes

In this section we will take a look at the code running on our Arduino Mega equipped with an LTE-m IoT Shield. The code allows not only to send an IoT transaction but also to transmit data using the Shield and thus through the IP Stack.

The behaviour of the node, i.e. how the device will collect data and send it, is almost identical to the LoRa version. The only thing that changes is that we do not have any aggregation and thus all that part is no longer required. With LoRa and the TTN network, we were able to send acknowledgements to the node. In doing so, the node was aware whether the data it was actually being sending, reached the TTN-plugin.

With the Cellular IoT Shield from Sixfab, we are not able to have the same behaviour. Sending acknowledgement or data to the node is officially not supported, i.e. not implemented by the producer. We tried to implement it but it did not work as expected and thus we had to leave it for the moment.

The first idea was to send only bytes in order to avoid any overhead and thus optimize at most our architecture. When sending raw bytes like we used to for the LoRa architecture, some data was lost as explained in the previous section. The first workaround we found was not really stable and the Bazo miner couldn't properly reconstruct the IoT transaction and verify it.

Therefore, the only solution that actually worked is to predefine some values as replacement for 0's. This method consists in substituting 0's with a predefined flag right before sending the transaction.

```
1 to_send = send_convert(sigTx, sizeof(sigTx));
2 node.sendDataUDP(to_send);
```

Listing 3.7: Transaction conversion

In order to achieve this, we keep characters starting from 0x03 to 0xFF as usual, i.e. we do not modify them, and we use 2 byte as gray code for the remaining characters: 0x00, 0x01, 0x02.

We do the following conversion when encountering those characters:

- 0x00 is converted to 0x01 0x1
- 0x01 is converted to 0x01 0x2

- 0x02 is converted to 0x02 0x1

We then do the opposite when receiving the data in order to get back to the original sequence.

The following code snippet shows the implementation of the method for converting a transaction before leaving the LTE-m device.

```

1 uint8_t* send_convert(uint8_t* str, int len) {
2     // max size of the returned string is 2 * len + 1
3     int final_len = 2 * len + 1;
4     int i = 0, j = 0;
5
6     uint8_t* vec = (uint8_t*) malloc(final_len);
7     memset(vec, 0, final_len);
8
9     for(i = 0; i < len; i++) {
10         if(str[i] == 0) {
11             vec[j] = 0x1;
12             vec[j+1] = 0x1;
13             j += 2;
14         }
15         else if(str[i] == 1) {
16             vec[j] = 0x1;
17             vec[j+1] = 0x2;
18             j += 2;
19         }
20         else if(str[i] == 2) {
21             vec[j] = 0x2;
22             vec[j+1] = 0x2;
23             j += 2;
24         }
25         else {
26             vec[j] = str[i];
27             j++;
28         }
29     }
30
31     vec[j] = '\0';
32
33     return vec;
34 }
```

Listing 3.8: Gray coding forbidden characters

3.2.3 UDP packet listener on Bazo Client

In order to be able to get IoT Transactions from LTE-m nodes, the Bazo Client has to listen to incoming UDP packets. To simplify the process and enable a single Bazo Client to listen not only to LTE-m nodes but even to LoRa nodes, we implemented everything together. We created a new package called UDP into the root folder of the Bazo Client. This package includes a file `UDP.go` which manages the incoming UDP packets.

It has a single function called `Init()` which starts the whole process. At line 4 we start

our UDP listener by invoking the method `net.ListenUDP(...)`. It is configured to listen to incoming packets on the port 5000. This can be arbitrarily changed but has to match the same port configured on the Arduino Sketch.

```

1 func Init() {
2     logger = util.InitLogger()
3     logger.Printf("%v\n\n", "Listening to incoming UDP packets...")
4     ServerConn, _ := net.ListenUDP("udp", &net.UDPAddr{IP: []byte{0, 0, 0, 0}, Port: 5000,
5         Zone: ""})
6
7     defer ServerConn.Close()
8     buf := make([]byte, 1024*2)
9     for {
10         n, addr, _ := ServerConn.ReadFromUDP(buf)
11
12         logger.Println("Received transaction: from ", n)
13         length, tx := convert_tx(buf, n)
14
15         var signature [64]byte;
16         copy(signature[:], tx[:64]);
17         tx = tx[64:]
18
19         var To [32]byte;
20         copy(To[:], tx[:32]);
21         tx = tx[32:]
22
23         var From [32]byte;
24         copy(From[:], tx[:32]);
25         tx = tx[32:]
26
27         TxCnt := [4]byte{}
28         for index := range tx[:4] {
29             TxCnt[4-index-1] = byte(tx[index])
30         }
31         TxCnt32, _ := binary.Uvarint(TxCnt[:]);
32         tx = tx[4:];
33
34         TxFee := [8]byte{}
35         for index := range tx[:8] {
36             TxFee[8-index-1] = byte(tx[index])
37         }
38         TxFee64, _ := binary.Uvarint(TxFee[:]);
39         tx = tx[8:];
40
41         Header := tx[:1];
42         tx = tx[1:];
43
44         data := tx[:];
45
46         IotTx := protocol.IotTx{
47             Header: Header[0],
48             TxCnt:    uint32(TxCnt32),
49             From:     protocol.SerializeHashContent(From),
50             To:       protocol.SerializeHashContent(To),
51             Sig:      signature,
52             Data:     data,
53             Fee:      TxFee64,
54         }
55         network.SendIotTx(util.Config.BootstrapIpport, &IotTx, p2p.IOTTX_BRDCST)
56     }
57 }

```

Listing 3.9: Implementation of the UDP listener client/UDP/UDP.go

The logic behind the transformation of the data stays the same as in the previous section. Here, we will only show the code to reconstruct the original data after the conversion has been applied to successfully send a transaction through the LTE-m shield.

The transaction is composed as follows:

Signature | To | From | TxCnt | TxFee | Header | Data

where:

- **Signature** is of length 64 Bytes.
- **To** represents the address of the recipient, it is 32 Bytes.
- **From** is the address of the sender (the node), it is 32 bytes as well.
- **TxCnt** is the transaction counter which in Bazo is a uint32, which means a byte array of 4 Bytes.
- **TxFee** is the transaction fee which in Bazo is a uint64, which means a byte array of 8 Bytes.
- **Header** is the transaction header of exactly 1 Byte
- **Data** is the transaction data of variable size

In code snippet 3.9, the variables are extracted in the same order and by following the same rules for the data sizes.

Before being able to do this, we have to convert it to the original data with the following method, called in snippet 3.9 at line 12:

```

1 func convert_tx(str []byte, length int) (int, []byte){
2     tx_raw := [2000]byte{};
3     i:=0;
4     j:=0;
5
6     for i < length {
7         if(str[i] > 2) {
8             tx_raw[j] = str[i];
9             i++;
10            j++;
11        }
12        if(str[i] == 0x1) {
13            if(i+1 < length) {
14                if(str[i+1] == 0x1) {
15                    tx_raw[j] = 0x0;
16                    i += 2;
17                    j++;
18                } else if(str[i+1] == 0x2) {
19                    tx_raw[j] = 0x1;
20                    i += 2;
21                    j++;
22                } else {
23                    // error
24                    i++;
25                    j++;
26                }
27            }
28        }
29    }
30    if(str[i] == 0x2) {
31        if(i+1 < length) {

```

```
32         if(str[i+1] == 0x2) {  
33             tx_raw[j] = 0x2;  
34             i += 2;  
35             j++;  
36         } else {  
37             // error  
38             i++;  
39             j++;  
40         }  
41     }  
42 }  
43 }  
44 return j, tx_raw[:j]  
45 }
```

Listing 3.10: Conversion method for reconstructing the original IoT transaction

Important to mention is that at **line 17** we create a buffer of bytes of length 1024×2 . If the incoming message is going to be bigger than that buffer, we will not receive the whole incoming data. Therefore, it has to be manually changed if required.

At the end, we simply construct our IoT transaction (**IoTTx**) as required for Bazo and we broadcast it to the network.

We did not implement the TCP protocol, since the node would get stuck if the Bazo client is for some reason unreachable. The only way we found to unfreeze it, it is a manual reboot of the devices which does not seem like a reasonable solution for us, since every device should be able to run autonomously.

Chapter 4

Evaluation

In this chapter, we will analyze the work done in this study and compare some part of it with the version running with LoRa nodes. Firstly, we will see which security aspects we could improve and which ones we are still missing. Then, we will discuss their limitations and we will have a look at their differences, including advantages and disadvantages. At the end, we will have some comparisons and tests.

We relied on the Sixfab CellularIoT library and therefore we had to neglect some important aspects that could have been improved a lot.

4.1 Security Aspects

In this section, we will quickly discuss some security aspects about the implemented system.

An important aspect for security reasons regards receiving acknowledgements from the Bazo Client to the LTE-m node. By being able to send them, we would not only be sure that the data is received, but with an hash as an acknowledgement, we would be able to ensure that the data is even correctly transmitted.

With LoRa and its duty cycle limits, we weren't able to send more than a flag as downlink packets. This time, because of the missing implementation in the official library, we were not able again to correctly apply this aspect as well. Therefore, the node cannot be sure that its transactions are being sent. Hopefully, the official library will get soon updated with the fixes we explained in the previous chapter and maybe receiving downlink message will also be supported out of the box, moving the effort and focus from the device and its shield to the actual use case implementation.

4.2 Arduino LTE-m CellularIoT Shield

One of the biggest challenge of this work was to actually be able to properly send data through the LTE Shield and not the connection to the Bazo Blockchain itself. Since Bazo

was already adapted to accomodate IoT Transaction, we had to put almost zero effort in it.

In this study, we explained the workaround we had to do in order to be able to send our data through the Shield. As almost every workaround that manages to achieve a specific goal, our also has some downsides and some compromises had to be taken.

In this section we will have a look at the optimized version with small overhead and show how well it performs.

Transaction Length	Converted Transaction Length	Data Collected	Overhead in %
151	165	10	9.27
181	195	40	7.73
241	256	100	6.22
291	306	150	5.15
341	356	200	4.40
381	399	240	4.72
641	660	500	2.96
1141	1165	1000	2.10

Table 4.1: Comparisong of converted data with the Arduino LTE-m CellularIoT Shield

In table 4.1, the overhead refers to the additional bytes that have to be sent when converting the transaction. In percentage the overhead starts from less than **10%** and gets lower and lower as the size of the transaction increases. This is due to the transaction counter (TxCnt) and the transaction fee (TxFee), which converted from a uint64, respectively from a uint32 to a byte array, contain many zeros that are going to cause some overhead due to the gray coding explained in the previous chapter. Increasing the size makes their impact much less significant and therefore the overall overhead gets reduced.

The second last column shows the size of the raw data collected we are sending. This can actually be compared to the one aggregated for the LoRa nodes. If we subtract the size of the collected data from the size of the transaction before it leaves the nodes and thus, before converting it, we can see that we have an overhead of always **141 Bytes**. Those bytes come from all the attributes of the IoT transaction and they are the trade-off between an aggregation (with LoRa) or having a full IoT transaction without any intermediary between the node and the Bazo Client.

Even if the Sixfab library would properly work, this bytes are always going to be present as long as something like a management HUB is not implemented.

4.3 Energy efficiency of LTE-m

In this section we are going to analyze the energy efficiency of the architecture running on LTE-m and where possible, we are going to compare it to the one for LoRa nodes.

Since both architecture use the same device (Arduino Mega 2560) and also share a lot of

the Arduino Sketch, most important the same cryptographic function such as SHA-25 and the EllipticCurve Digital Signature Algorithm Ed25519 , we are not going to compare the power consumption while deriving the public key, hashing the transaction and signing the hash. For those reasons, we are going to analyze the power consumption while the device is doing nothing and when the device continuously sends data.

Table 4.2 shows the power consumption of both devices under two different conditions: While in stand-by, both devices consume much less power than when sending data. LoRa is the technology which consume at least power. Indeed, LTE-m when in stand-by, consume around 35% more current. If we compare both technologies with a bare arduino (no shield attached), we clearly see that the LoRa shield consumes only about 15% more power, which is great considering the whole LoRa stack attached to it.

Both technologies consume definitely more when continuously sending data. In this case, the LTE-m stack consumes only about 5% more than LoRa but both of them consume around 58-103% more than in stand-by. A bare arduino is not able to send data without any shield attached and therefore its value in the last column is missing

	Stand-by (Ampere)	Sending (Ampere)
LTE-m	0.1104A	0.1744A
LoRa	0.0813A	0.1655A
Bare Arduino	0.0720A	-

Table 4.2: Power consumption of LTE-m, LoRa and a bare Arduino device

An interesting case to investigate would have been to compare both devices when in sleep mode. Unluckily, for the LoRa node we didn't find a way to put it into sleep mode. For the LTE-m node using the CellularIoT shield there is the following command which is supposed to put the shield into sleep:

```
sendATComm("AT+CSCLK=1", "OK\r\n");
```

The response we get is correct but the power consumption does not change at all, therefore we do not know whether the device went really into sleep mode or not.

Another very important point to mention is the time the device needs to get its configuration done and be ready to send data.

The LoRa stack doesn't almost need any configuration since it can just wake up and start sending data immediately. The LTE-m stack however, needs to get connected to the network provider and only then can start sending data.

While a LoRa node needs around 29 milliseconds to send data, a LTE-m nodes needs around 37 seconds to achieve this state. Moreover, this can depend on the signal strength of the network service. When testing this devices, we sometimes experienced major issues where we had to wait more than 230 seconds for the device to establish a connection with the provider and thus being able to proceed and send data.

	Average elapsed time
LTE-m	37584ms
LoRa	29ms

Table 4.3: Elapsed time before being able to send data

Chapter 5

Summary and Conclusions

The main objective of this study was to compare the overhead and energy consumption of LTE-m IoT nodes with respect to LoRa nodes. We were not able to compare the devices when in sleep mode but we were able to get some results when in stand-by and when transmitting data over to the blockchain.

We can actually conclude that the LoRa protocol works best in terms of energy consumption. Since the nodes are not supposed to constantly transmit data and thus their main state will be the stand-by mode, the LoRa node considerably consumes less power. As already stated, both devices consume the same amount while signing transactions and since the field/data to be signed can be the same, no difference is present there.

The transmission pattern looks quite different and so does the data overhead. With LTE-m nodes we always send the whole IoT transaction and thus the overhead is already present there. Moreover, since the official library for the CellularIoT shield is not working as expected, we have to introduce even more overhead as explained in a previous chapter.

Connecting the devices to the Bazo Blockchain was quite simple once we managed to correctly send data out of the LTE-m node. The changes we had to do for the Bazo Client are minor changes compared to the rest of the study and compared to the work done in [9].

To conclude, we can state that the overall architecture for both families of IoT devices (LoRa and LTE-m) works quite good together. LTE-m nodes can be used in scenarios where more data is needed in the same time window since we do not have any duty cycle or air time limits. LoRa nodes can be employed where data frequency is quite low and energy efficiency plays a significant role.

Bibliography

- [1] Arduino-Lmic library, <https://github.com/matthijskooijman/arduino-lmic>, accessed January 2019
- [2] Ed25519 Library for Arduino, <https://rweather.github.io/arduino-libs/classEd25519.htm>, accessed February 2019
- [3] Samwel, Batina, Bertoni, Breaking Ed25519 in WolfSSL, <https://eprint.iacr.org/2017/985.pdf>, accessed March 2019
- [4] Definition of a structure in Arduino, <http://playground.arduino.cc/code/struct>, accessed March 2019
- [5] Arduino Mega 2560, <https://store.arduino.cc/mega-2560-r3>, accessed August 2019
- [6] Sixfab Arduino CellularIoT Shield, <https://sixfab.com/product/arduino-cellular-iot-application-shield/>, accessed August 2019
- [7] Quectel BG96, <https://www.quectel.com/product/bg96.htm>, accessed August 2019
- [8] Explanation of double-spending attack, <https://en.wikipedia.org/wiki/Double-spending>, accessed August 2019
- [9] Cepilov Ile, "Design and Implementation of an IoT-based Hierarchical Payment System with BAZO Blockchain", April 11th, 2019.
- [10] Bazo Blockchain github repository, <https://github.com/bazo-blockchain>, accessed August 2019
- [11] Sixfab Arduino Cellular IoT library, https://github.com/sixfab/Sixfab_Arduino_CellularIoT_App_Shield, accessed August 2019

Abbreviations

Ack	Acknowledgement
API	Application Programming Interface
BC	Blockchain
ECDSA	Elliptic Curve Digital Signature Algorithm
Ed25519	Edwards-Curve Digital Signature Algorithm
IoT	Internet of Things
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
LoRa	Long Range
LoRaWAN	Long Range Wide Area Network
NPM	Node Package Manager
PoS	Proof-of-stake
REST	Representational State Transfer
TTN	TheThingsNetwork
tx	Transaction
txCnt	Transaction Counter
txFee	Transaction Fee

List of Figures

3.1	Agnostic IoT architecture adapted from [9]	6
3.2	LTE-m HW Wallet for a full IoT Transaction	7

List of Tables

4.1	Comparisong of converted data with the Arduino LTE-m CellularIoT Shield	18
4.2	Power consumption of LTE-m, LoRa and a bare Arduino device	19
4.3	Elapsed time before being able to send data	20

Listings

3.1	Constant for Sixfab Cellular IoT library	8
3.2	Private value for Sixfab Cellular IoT library	8
3.3	Send data UDP function in the Sixfab CellularIoT library	8
3.4	Send data command in Sixfab CellularIoT library	9
3.5	Avoid an infinite while loop in the Sixfab CellularIoT library	10
3.6	Reconstruct the received data with 0's	10
3.7	Transaction conversion	11
3.8	Gray coding forbidden characters	12
3.9	Implementation of the UDP listener client/UDP/UDP.go	13
3.10	Conversion method for reconstructing the original IoT transaction	14

Appendix A

Installation Guidelines

A.1 Prerequisites

In order to follow this guidelines, some prerequisites are necessary. First of all, an account with enough funds has to be already created in the Bazo blockchain. The private and public key of this account have to be saved so that they can be manually inserted into the Arduino sketch. Moreover, since we are going to send data through the UDP protocol, the corresponding port, in our case the port 5000, has to be opened on the router in case that the Bazo client does not have a public IP address. For more information on how to open the port on your router please visit: <https://portforward.com/> Lastly, a subscription to a mobile internet provider and an active Subscriber Identification Module (SIM) card is required in order to be able to send data from the LTE-m node.

A.2 LTE-m Arduino sketch

In order to run the sketch on an arduino, an Arduino CellularIoT 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 used for the LoRa project [9], in order to get it either download 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, Switch to the current and most updated branch by typing:

```
git checkout "udp"
```

Then navigate in the folder `arduino` and there you will find the file `lte-shield.ino`. Open the file using the official Arduino Ide and insert your previously saved public and private key. In order to convert it to the appropriate format, you can navigate to the following link and convert them:

```
https://cryptii.com/pipes/base64-to-hex
```

Make sure to install all the required libraries, including the Sixfab Arduino CellularIoT Shield library, which can be found here [6]. At this point you are ready to upload the sketch using a USB cable connected to the arduino. Once uploaded, there is no required step for the LTE-m Node. Just make sure that the node has internet connectivity.

A.3 Bazo Miner

The installation of the Bazo Miner stays the same like in the work done in [9]:

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/ilecippi/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/ilecippi/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.4 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
```

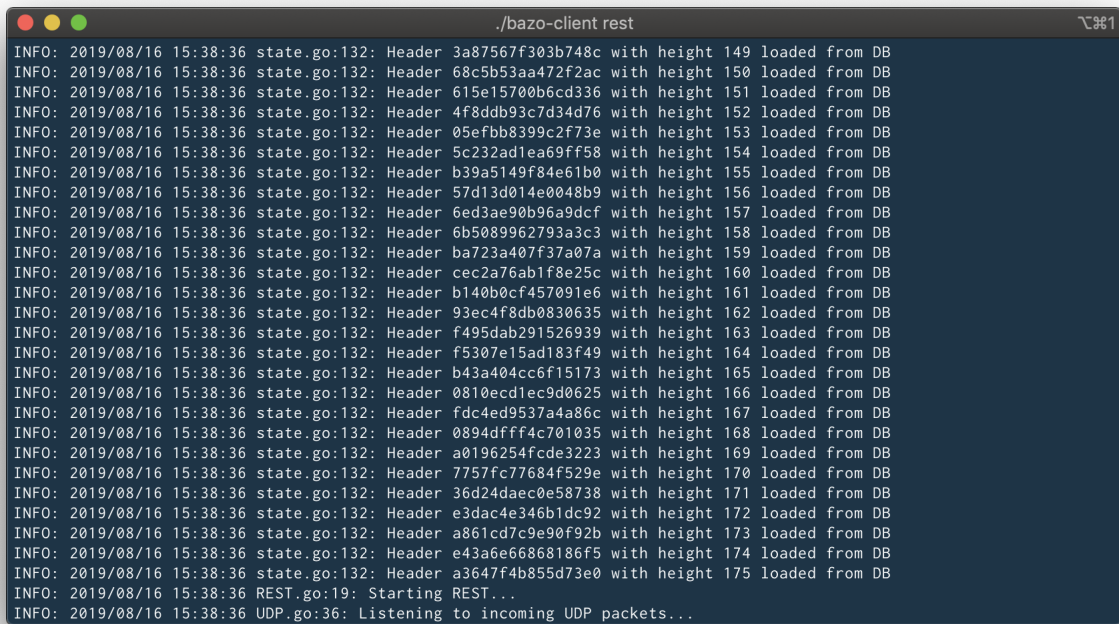
Change the branch:

```
git checkout "lte_final"
```

Build the project and start the REST interface, which will also start the UDP listener. First make sure to have a local bazo-miner running and most important make sure to have correctly done the port-forwarding so that your Bazo client is reachable from outside, i.e. from your LTE-m node:

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

The result should be the following:



```
./bazo-client rest
INFO: 2019/08/16 15:38:36 state.go:132: Header 3a87567f303b748c with height 149 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 68c5b53aa472f2ac with height 150 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 615e15700b6cd336 with height 151 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 4f8ddb93c7d34d76 with height 152 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 05efbb8399c2f73e with height 153 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 5c232ad1ea69ff58 with height 154 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header b39a5149f84e61b0 with height 155 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 57d13d014e0048b9 with height 156 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 6ed3ae90b96a9dcf with height 157 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 6b5089962793a3c3 with height 158 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header ba723a407f37a07a with height 159 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header cec2a76ab1f8e25c with height 160 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header b140b0cf457091e6 with height 161 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 93ec4f8db0830635 with height 162 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header f495dab291526939 with height 163 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header f5307e15ad183f49 with height 164 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header b43a404cc6f15173 with height 165 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 0810ecd1ec9d0625 with height 166 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header fdc4ed9537a4a86c with height 167 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 0894dfff4c701035 with height 168 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header a0196254fcde3223 with height 169 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 7757fc77684f529e with height 170 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header 36d24daec0e58738 with height 171 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header e3dac4e346b1dc92 with height 172 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header a861cd7c9e90f92b with height 173 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header e43a6e66868186f5 with height 174 loaded from DB
INFO: 2019/08/16 15:38:36 state.go:132: Header a3647f4b855d73e0 with height 175 loaded from DB
INFO: 2019/08/16 15:38:36 REST.go:19: Starting REST...
INFO: 2019/08/16 15:38:36 UDP.go:36: Listening to incoming UDP packets...
```