Blockchains Accesses for Low-Power Embedded Devices using LoRaWAN

Manon Arnaudo
LEAT, Université Côte d'Azur
Sophia-Antipolis, France
manon.arnaudo@etu.univ-cotedazur.fr

Luc Gerrits
LEAT, Université Côte d'Azur
Sophia-Antipolis, France
luc.gerrits@univ-cotedazur.fr

Ilya Grishkov
Delft University of Technology
Delft, The Netherlands
I.Grishkov-1@student.tudelft.nl

Roland Kromes
Delft University of Technology
Delft, The Netherlands
R.G.Kromes@tudelft.nl

François Verdier
LEAT, Université Côte d'Azur
Sophia-Antipolis, France
francois.verdier@univ-cotedazur.fr

Abstract

We present in this work a Lightweight Transaction Protocol (LTP) for enabling constrained embedded devices' interaction with multiple types of blockchains via the LoRaWAN communication protocol. The proposed protocol provides the benefit of interacting with multiple blockchains with only 13.5% decrease in battery life. The protocol also includes a gateway module API that connects to blockchains and generic smart contracts to authenticate end devices and store their data. The study points out that, in the worst case, end-device data can be stored in blockchains with a latency of 20.3 seconds for Substrate and 6.8 seconds for the Hyperledger Fabric blockchain.

Index Terms

IoT, Blockchain, LoRaWAN, Embedded Systems, Gateway, Hyperledger Fabric, Substrate blockchain framework

I. Introduction

One of the main objectives in the IoT (Internet of Things) is to provide operations or automated logic without any human intervention. These operations are generally executed in a centralized unit of the IoT network according to the input data recorded by the IoT devices. The number of devices in IoT networks can be typically significant, and their identification, authentication, and authorization are vital in IoT networks to achieve a highly secured environment. Since most IoT networks depend on a centralized unit (e.g., server, cloud), providing a secure environment is a real challenge because once the centralized unit is compromised with, the overall network structure can no longer be considered secure. From the point of view of users or IoT owners, the central entity of the IoT network cannot be considered a fully trusted party. It is because the operations that run on a centralized unit (which can be considered as a trusted party) are not auditable, thus can alter the provided results. Replacing the centralized entity of an IoT network with blockchain technology can be a sufficient solution to achieve a secure environment, as blockchain provides traceability, auditability and immutability of data. Moreover, with its ability to execute smart contracts, this technology can provide a trustful execution of any business logic.

A. Motivation

Nowadays, multiple blockchains exist, each of which has specific requirements, such as the digital signature, encoding format, and many others to interact with them. The constrained IoT devices (embedded systems) integration with blockchain technology is challenging because of the following main issues:

1) The device must be securely authenticated to the blockchain network. Therefore a hardware resource-intensive elliptic curve digital signature must be applied.
2) The hardware resources can be very limited in IoT devices such as their limited memory storage capacity, their computational power or their battery lifetime. Due to the memory constraint, a constrained device cannot contain a complicated API of significant size. These constraints make it more difficult for the IoT device to communicate with different types of blockchains.
3) In several cases, the transaction allowing the interaction with the blockchain must contain metadata about the actual ledger state. Thus, this metadata must be downloaded before the transaction creation. In the case of IoT, this procedure is critical because the number of communication (message sending) of IoT devices must be limited. Their energy consumption is highly increased due to the radio module activation. The blockchain's transaction size was not necessarily tailored for the use of constrained IoT communication protocols such as LoRaWAN or BLE Bluetooth. The "IoT-friendly" communication protocols typically apply a payload size of 256 bytes.
4) The choice of the IoT communication protocol significantly impacts the device's energy consumption, the topology, and the range of the IoT network.

In order to find an optimal solution to these issues, our principal objective is therefore to verify if some very Low-Power IoT devices could interact with multiple blockchains and smart contracts automatically and in an authenticated manner. This objective is separated into three parts. First the device must perform a digital signature for authenticating itself to the given blockchain. The IoT device also runs a highly optimized generic API that can access ***multiple types of blockchain*** without computing any specific blockchain-related requirements. The device sends information to the blockchain but does not download any information from it. Finally, the device uses LoRaWAN IoT-efficient communication protocol for sending its data.

B. Contribution

In order to achieve the goals mentioned above, a highly optimized API written in C language was developed to be used in constrained IoT devices. As the IoT device's application does not include the specific task for blockchain transaction creation, a specific gateway module API was also developed. The gateway module API based on Rust language includes all the necessary tools to handle the messages of the IoT devices, parses them to know the targeted blockchain, and finally, creates the transaction for the specified blockchain. It should be noted that the gateway module API-generated transaction includes the IoT device's payload. The current state of the gateway module API allows the interaction with Hyperledger Fabric [1] and a blockchain based on the Substrate framework [2], but it is not limited to targeting only these two blockchains. Generic smart contracts were also implemented to allow the secure authentication and data storage of the IoT devices.

The structure of the article is as follows: We give in Sect. II an evolving state of the art of blockchain technology and embedded device implementations, followed by the problem of creating blockchain transactions. Our proposed Lightweight Transaction Library and the gateway module API are detailed in Sect. III. The results of our implementations are presented in Sect. IV with both the additional energy consumption in the IoT devices due to the proposed library and the effects on the different blockchain used. Finally we conclude this article in Sect. V.

II. State of the art

Blockchain is a distributed database that guarantees data immutability and the traceability of blockchain events. The blockchain is also a peer-to-peer network in which the blockchain members are identified and authenticated thanks to their public keys [3] [4]. Data can be set to the blockchain via the so-called blockchain transactions, signed by the transaction issuer's private key. Elliptic curve cryptography is applied to verify the signature over the transactions, which allows identifying if the issuer takes part in the blockchain network and if the transaction was not altered during its sending [3] [5]. The newly validated transactions are collected in a block added to the top of the chain of blocks by including the previous block's hash value. Since blockchain technology provides a secure environment, deploying digital codes on it, is advantageous because the execution logic and results cannot be altered. The previously mentioned business logic is also called smart contract [6].

A. *Blockchain and low power IoT*

Multiple research works on integrating the IoT devices with block-chain technology can be found in the literature. In this section, we discuss the related work which had the most significant impact on our implementation. In the works of Pincheira et al. [7] [8], the authors' overall goal was to establish direct communication between the IoT devices and the Ethereum blockchain [9]. The authors deployed an Arduino C library which allows the creation of Ethereum transactions locally in the IoT device. The valid Ethereum transaction creation requires the digital signature of the transaction, which must be done by a private key of a registered blockchain participant. One of the conclusions of this article is that the digital signature procedure should be accelerated because its creation can take a significant amount of time. The authors proposed a network architecture in which the transactions are forwarded to the blockchain via a gateway using the LoRaWAN protocol.

Other works [10] [11] propose an IoT API written in C++ to create valid blockchain transactions for the Hyperledger Sawtooth blockchain. In these works, the authors highlight that the Hyperledger Sawtooth blockchain requirements for valid transaction creation are different than in the case of the Ethereum blockchain. In the case of Hyperledger Sawtooth the transaction structure is different, and at least two signature is required (one on the transaction and one on the batch containing the transaction). In that study, the authors are convinced that IoT device's communication protocols like LoRaWAN are more efficient for sending transactions.

B. *Blockchain transactions*

As mentioned, each blockchain has different requirements to create valid blockchain transactions. In the case of the Ethereum blockchain, the Recursive Length Prefix (RLP) serialization method is used for encoding the transaction. Hyperledger Sawtooth uses protocol buffers and CBOR encoding for serialization. Moreover, this transaction is encapsulated in a batch that requires its signature in addition to the transaction's signature. With Substrate based blockchains, SCALE (Simple Concatenated Aggregate Little-endian Encoding) is used for all data sent and received in the network. Substrate has its own SCALE codec implementation and is optimized for WebAssembly execution.

The nonce and metadata problem can also be considered a typical challenge in IoT blockchain integration. For example, Ethereum is one of the most popular blockchain allowing the deployment of crypto-currency transactions and complex business logic via smart contracts. Substrate blockchain is a modular framework which allows the deployment of new complete blockchain network structures. In these two blockchains the nonce value must be incremented every time a new transaction is generated. However, the locally incremented nonce in the IoT and the nonce value in the blockchain may differ, which will cause the rejection of the transaction. The synchronization of nonce can avoid this problem, but in that case, the IoT device must perform additive communication. The same communication issue takes place, for example, in the case of EOS.IO [12] blockchain because it requires that the transaction issuer downloads some current metadata from the blockchain (otherwise, the transaction will be rejected).

Another issue is the signature requirements. For example, Hyperledger Sawtooth and Hyperledger Fabric use the SHA-2 cryptographic hash functions in the signature schemes. However, Ethereum uses Keccak. Requirements are also related to the elliptic curves used in the signature schemes. While Hyperledger Sawtooth, Ethe-reum, and Substrate apply the secp256k1 curve, for signing the Hyperledger Fabric transactions, the P-256 curve must be used.

Connecting an IoT device to multiple blockchain networks can be highly complicated due to the different requirements, not to mention that implementing multiple APIs requires more storage space. It must also be noted that in the related works, the transaction signature is verified by the blockchain protocol. This paper proposes a so-called Lightweight Transaction Protocol (LTP) in which the IoT device signs a lightweight transaction (generic structure) embedded in a real blockchain transaction by the gateway module API. The gateway creates the transaction according to the requirements of the given blockchain. The proposed generic smart contract then verifies the lightweight transaction signature (signed by the IoT device).

C. *TheThingsNetwork and other solutions*

TheThingsNetwork (TTN) [13] is an IoT ecosystem that creates networks, devices and solutions using LoRaWAN [14]. TTN provides full solutions to deploy/manage gateways, applications, and end-devices. TTN is the middleware between gateways and the application server.

All devices are secured with credentials (AES encryption on the payload and a network key) directly into the TTN web-interface. It is possible to secure the end-device payload using custom AES keys that are unknown by TTN, thus the application server decrypt the payload. In addition, the network layer of TTN is secured with a 4 byte signature (sliced AES-CMAC RFC4493) using the network key and the concatenation of multiple fields of a LoRaWAN network protocol. Similar ecosystems and companies use the same global architecture as TTN.

TheThingsNetwork is a free Network server, open to all, pushing the idea of an open LoRaWAN network. The Things Industries is a paid solution offering a private network separated from the open community. ChirpStack [15] is an open source project, a manual solution for LoRaWAN deployment, development, and network management. In other words, it provides a solution to deploy a private gateway. Other organizations exists, mainly in the telecom operator sector, and multiple companies are actively deploying their LoRaWAN network creating their own private LoRaWAN stack solution (e.g. providing network subscriptions to specific LoRaWAN companies).

The common issue with these solutions is that they are all based on a centralized network management model. This centralization can be a significant issue when dealing with a high number of gateways and can create network organization monopoles. Niya et al. [16] have already associated LoRaWAN and blockchain to remove trusted third parties, in which they use TTN to receive the transmitted end-device payload. However, our approach removes the TTN layer and transmits the payload directly to a blockchain after the gateway receives it.

III. Architecture and proposed implementation

A. *Background*

Substrate blockchain [2] is an entirely modular blockchain framework that allows the modification of all of its components. The modification possibilities are vast, including the possible changes in signature schemes, consensus algorithms, and transaction structures. Its key feature is the connectivity with Polkadot, a relay-chain that provides shared security between multiple Substrate-based chains (also called *parachains*). This blockchain framework can probably allow a straightforward adaptation of blockchain to IoT technology.

Hyperledger Fabric [1] is one of the most popular private block-chains adapted to be used in complete ecosystems and enterprise use cases. The main particularities of this blockchain are that the smart contracts can be developed in different programming languages such as Golang, JavaScript, and Java. Another particularity is that the credentials used in this blockchain provide an easy way to deploy attribute-based access control.

B. Lightweight Transaction Protocol

We propose a novel protocol called *Lightweight Transaction Protocol* (LTP) that aims to enable blockchain transaction for very constrained embedded devices. The protocol is based on a specific data structure, asymmetric cryptography, requires blockchain smart contracts, and targets wireless wide area networks such as a LoRaWAN (Long-Range Wide Area Network). LTP has an associated library called *Lightweight Transaction Library* (LTL) written in C/C++ that is compatible with the Arduino IDE environment. The LTP is an additional layer of the LoRaWAN communication protocol, thus all LTP packets are encapsulated in LoRaWAN Uplink messages (more precisely in the LoRaWAN PHY payload), depicted in Fig. 1.

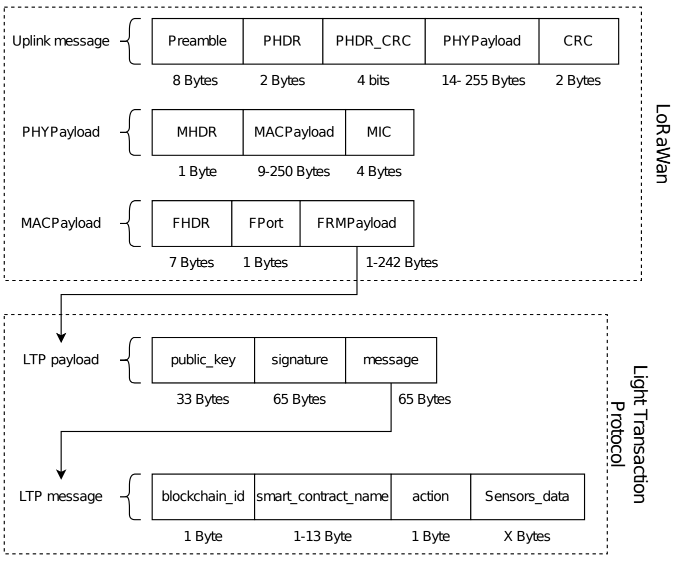

Fig. 1. LoRaWAN and LTP message structure

The LTP requires to implement a *blockchain module* on the LoRaWAN gateway which acts as a blockchain transaction demultiplexer. In Fig. 2 is the proposed Lightweight Transaction Protocol implementation using low power end-devices, a gateway, and two blockchains.

The LTP implementation uses i) Protobuf data format to serialize the structured LTP data, ii) the C library trezor-crypto for data hashing and signature generation and iii) LoRaWAN library called rfthings-stm32l4 which is derived from LacunaSpace code. The *signature* is based on ECDSA with the secp256k1 curve, thus using 32 bytes private key and a 33 bytes compressed *public_key*. The *blockchain_id* and the *action* are encoded on 1 byte and are represented in Protobuf by an enumerations type. The *smart_contract_name* is encoded on 1 to 12 bytes and represented in Protobuf by a string.

The smart contract will verify the LTP transaction using the LTP payload (public key, the signature, and the message). Thus, **LTP requires smart contracts** to operate. The smart contract i) manages the authorized end-devices to send data ii) verifies the signature and iii) stores the data on-chain. More details of the smart contract are in the Sect. III-C1 and Algorithm 1.

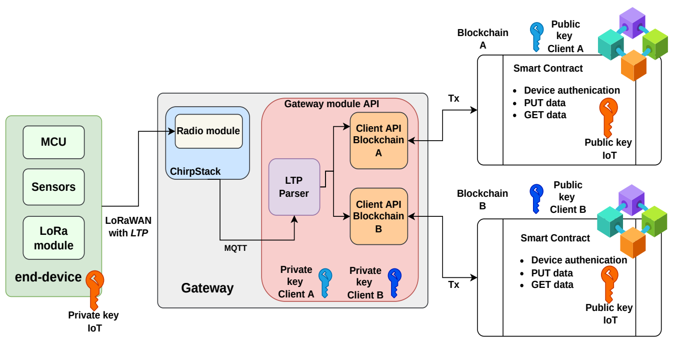

Fig. 2. The proposed LTP implementation

C. Gateway module API

Figure 2 represents a generic representation of the gateway containing the LoRaWAN radio module in a Chirpstack gateway module and the proposed Gateway module API. The radio module is associated with a Chirpstack gateway module which converts the LoRaWAN packages into UDP/MQTT packages [17]. MQTT is a lightweight protocol that makes easier the communication between client and broker (server) applications using publish/subscribe methods. Usually the UDP/MQTT packages are forwarded to a back-end (e.g., the Things Network server). In our implementation, the back-end is replaced by our gateway module API which creates blockchain transactions and forward them to multiple types of blockchain.

Thanks to an MQTT Client implemented in the gateway module API, the API can receive the MQTT packages forwarded by the ChirpStack module. The packages are set to a generic queue in order to avoid any loss. It is worth noting that the core implementation of the gateway module API is based on Rust programming language. The ThreadPool official Rust library was applied to achieve a multi-threaded environment and thus accelerate tasks execution.

In order to retrieve the lightweight transactions generated by the IoT devices, the thread pool instantiates as many threads as possible on the host architecture. After retrieving a lightweight transaction forwarded via an MQTT package, the LTP parser sub-module parses the lightweight transaction in order to determine which blockchain was targeted by the IoT device's (i.e., *blockchain_id*, see Fig. 1). This information is required because the gateway module API includes the given blockchain SDK or client application which must be executed for valid transaction creation. The LTP parser sub-module also provides the Lightweight Transaction Protocol payload (depicted in Fig. 1) to the blockchain client application.

The LTP payload contains valuable information for the valid blockchain transaction creation, such as the name or the address of the smart contract that must be called (i.e., *smart_contract_name*), the function/action to be executed in the smart contract (e.g., put or get data) and it also contains the data measured by the sensor (e.g., *Sensors_data*). One of the most important criterion of valid transaction creation is the digital signature of the transaction's payload, therefore the signature on the LTP payload. The digital signature is performed by a private key whose public pair is known by the blockchain.

In the current state of our gateway module API, the golang SDK of Hyperledger Fabric and the Rust client application of Substrate blockchain are available, and each API contains one private key.

Most SDKs are written in a specific programming language, which makes it difficult to integrate with the basic functionality of the gateway module API (written in Rust). In the case of Hyperldger Fabric, the golang SDK was first compiled into a C shared library which can be called by the Rust language using specific CString and pointer structures. Another challenging part of SDKs integration is related to the blockchain requirements (i.e., transaction structure and content). Substrate blockchain transactions

Algorithm 1: Generic Smart Contract required by LTP

```
1  PUT_DATA (Data, Signature, $PubKey_{IoT}$):
2  $exists$ = verify_ID_Storage($PubKey_{IoT}$)
3  if $exists$==$true$ then
4      $valid$ = verify_Signature($Signature$, $PubKey_{IoT}$, $Data$);
5      if $valid$ == $true$ then
6          $counter$ = get_Counter($PubKey_{IoT}$); store_Data($counter$,$PubKey_{IoT}$);
            increment_Counter($PubKey_{IoT}$);
7      else
8          return error;
9      end
10 else
11     return error;
12 end
13 GET_DATA ($PubKey_{IoT}$, $dataIndex$):
14 $exists$ = verify_ID_Storage($PubKey_{IoT}$);
15 if $exists$==$true$ then
16     $data$ = get_Data($PubKey_{IoT}$, $dataIndex$);
17     if $data$ != $nil$ then
18         return $data$
19     else
20         return error;
21     end
22 else
23     return error;
24 end
```

must contain a nonce value that specifies the number of transactions sent by a blockchain member. If the blockchain detects a difference of the nonce presented in the transaction and the one on the ledger state, than the transaction will be rejected. In a multi-threaded environment, increasing correctly the nonce value requires the application of mutexes, which also makes the API more complex.

1) Generic Smart Contract

In general, the given blockchain technology requires the use of a specific language to develop smart contracts (e.g., Substrate uses Rust, Ethereum Solidity). The LTP business logic has been implemented in Rust and Golang, which allows its application in Substrate and Hyperledger Fabric, respectively. Algorithm 1 presents the proposed business logic implemented in the previously mentioned smart contracts, which are called by the gateway module API. The business logic contains two main functionalities PUT_DATA and GET_DATA, for storing and recovering IoT device data. The PUT_DATA function (the last operation to be called after the IoT device sends data) first checks whether the IoT device is registered in the smart contract, and the device's public key is used as the device identifier. If the device exists in the database, the next step is the verification of the ECDSA signature [5]. The counter value associated with the IoT device identifier ($PubKey_{IoT}$) is recovered if the signature is valid. The counter value specifies the number of data sent by the device (incremented every time new data is added to the blockchain state). Finally, the data is stored according to the device's counter value and identifier. It must be noted that every time new data is added (i.e. the counter value is incremented), composite keys are generated to facilitate the storage in the blockchain state. External applications can recover data by calling the smart contract GET_DATA

function. The IoT device's public key and the number of the data ($dataIndex$) sent must be specified. If the device is registered, the data associated with the number of the data is provided to the caller.

IV. Results

A. Results on the end-device

1) Context and development board

The implementation of LTP, using LTL, was done on a development board (called *UCA board*) realized in the LEAT Laboratory. It is a board used for educational purposes, but can also be used in more advanced projects. The board is equipped with an Ultra-low-power Arm Cortex M4 32 bit STM32L476RG processor, 245 KB flash memory, running at 80 MHz, and uses LoRaWAN communication protocol thanks to the SX1262 Semtech module. The sent data is generated by two sensors: the HP203B precision barometer and altimeter sensor and the ICM20948 accelerometer. The HP203B retrieve temperature, altitude and pressure data. The ICM20948 retrieve x, y, and z axis accelerations. The board is programmed using Arduino IDE v1.8.19.

We used the OTII Arc device to measure the energy consumption. The OTII interface allows serial link communication with the development board, which is used to synchronize the software and energy consumption. Measurement (time, current, energy, and serial output) are exported in CSV files with a 4KHz sampling frequency and an accuracy of $\pm(0.1\% + 150\mu A)$. Software is synchronized (for each analyzed functions) with two or three bytes which are sent on the OTII serial link at a bit rate of 115200 bauds, which adds a delay of 0.19ms to 0.29ms on the measurements. We configure the LoRaWAN transmission of the end device with a spreading factor of 7, an end-point output power (*TXPower*) of 16 dBm, a custom preamble length of 984 symbol, disabled reception and a transmission frequency of 865MHz. The energy engaged by the LoRaWAN module amplifier can fluctuate according to the measurement environment, thus the experiments are conducted in a fixed environment, avoiding external disturbances.

The LoRaWAN protocol allows us to send payloads of 242 bytes if we use a spreading factor of 7. The use of the LTL requires minimum 124 bytes because of the blockchain transaction requirements. Protobuf encoding consist of key-value pairs, with the key representing two information. Increasing the number of fields in the Protobuf message causes the encoding of the key to vary (i.e. it may be using more than one byte). The remaining 118 free bytes can be used for other purposes. If only floating values are sent, the payload will contain 22 fields, assuming 5 bytes per field (1 for the key + 4 bytes for float representation).

2) Detailed analysis of LTL

The use of the two sensors consumes power but allows the reporting of a practical example. In the first case, the LTL library is not used, the 24 bytes are sent directly by LoRaWAN protocol (Fig. 3). An average current of 10.91mA can be observed during the sensing period and an average current of 91.45mA when the board is sending the data (LoRaWAN).

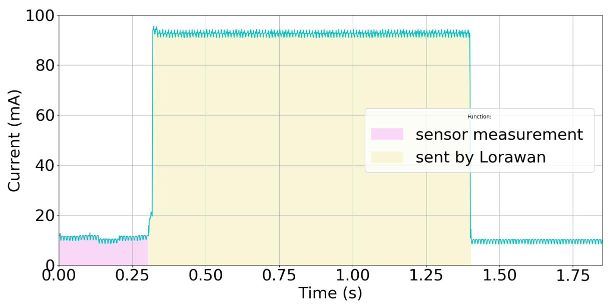

Fig. 3. Overall current consumption to transmit 24 bytes of sensor data without LTL

In the second case, the LTL library is used, thus adds 130 bytes to the payload which is a total of 154 bytes sent by LoRaWAN (Fig. 4). We are sending six floating values which consist of 30 bytes in total because of the Protobuf encoding. The end-device thus sends a lightweight transaction containing six data fields retrieved by the sensors. An average current of 10.85mA can be observed during the sensing period, an average current of 19.62mA when building the lightweight transaction by LTL, and an average current of 91.24mA when the board is sending the data (LoRaWAN).

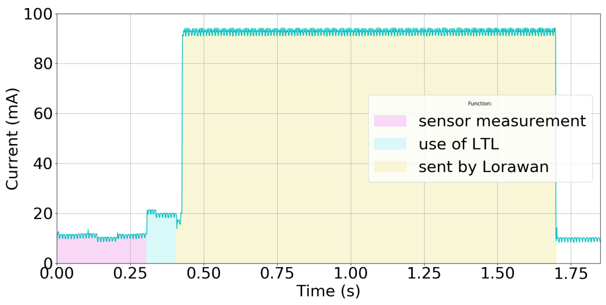

Fig. 4. Overall current consumption to transmit 24 bytes of sensor data with LTL

A benchmark of measurements is described in Table I. To observe the stability and variance of the results we performed 10 consecutive measurements. ΔT is the elapsed time and ΔE is the energy consumed between the wake-up phase of the micro-controller and the end of sending phase.

	Without LTL	With LTL
Memory Footprint (kBytes)	57.1	115.1
Data sent by LoRaWAN (Bytes)	24	154
Avg ΔT (s)	1.40	1.70
Avg ΔE (mJ)	311.31	370.65
STD ΔE (mJ)	0.22	0.42

TABLE I
ENERGY, MEMORY FOOTPRINT, AND TIME IMPACT OF LTL (ON 10 MEASUREMENTS)

The impact of LTL on the end-device, for the same sensor data, costs an average of 21.4% more time, 19.1% more energy and an additional 130 Bytes has be be sent. An increase of 101.6% can also be observed in the memory footprint when using LTL.

Note: LTL is compiled with secp256k1 pre-computed elliptic curve (EC) points provided in the trezor-crypto library. Disabling pre-computed EC points will increase computation time by an order of 4 and reduce memory by 14%. The optimization of Arduino IDE is configured to generate smallest code.

Energy consumption depends on the length of the message, which varies when using or not LTL. To isolate the LTL consumption independently of the sensors, we disconnected them and we propose a comparison giving its results in Fig. 5. The comparison is obtained by averaging 10 measurements and increasing the data size of the payload. The data consists only of N floating values. As explained in Sec. IV-A1, using Protobuf in LTL, one floating value is encoded into 5 bytes. However, when not using LTL it is possible to send directly the 4 bytes representing the float value. To summarize, the data size is a function defined as:

- Without LTL: $f(N) = N * 4$
- With LTL: $f(N) = N * 4 + 130 + ProtoFieldKey$
  N = the number of floating values
  ProtoFieldKey = Protobuf encoding field key
  $$ProtoFieldKey = \begin{cases} 1, & \text{if N} < 15 \\ 2, & \text{if N} \geq 15 \end{cases}$$

The curve without LTL shows a linear energy consumption starting at 282.78 mJ and going up to 337.43 mJ. With LTL, the energy consumption is between 337.43 mJ and 380.04 mJ. Globally, using the LTL library represents an increase in energy from 54.65 mJ to 66.5 mJ (for 4 and 88 bytes respectively).

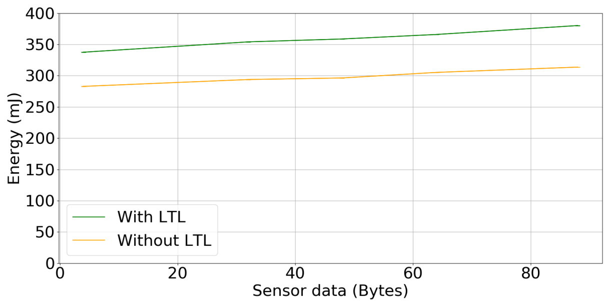

Fig. 5. Energy per transaction as a function of the number of bytes sent

We compare the energy consumption of the runtime using the minimal (4 bytes) and the maximal (88 bytes) payload. The results given in Table II shows that the sign message function (i.e. sign the LTP transaction) has the biggest impact of the LTL energy consumption. The use of the library is always about 4 mJ and does not depend on the data size sent. However, LoRaWAN communication takes 333.17 mJ to send 4 bytes and 375.75 mJ to send 88 bytes. Thus the overall energy consumption depends on this LoRaWAN communication step and not on LTL. Therefore, increasing data size will primarily increase the energy consumption of the LoRaWAN communication.

		Sensors data size	
		4 bytes	88 bytes
LTL (mJ)	Encoding message	<1	<1
	Hash message	<1	<1
	Sign message	4.19	4.20
	Encoding payload	<1	<1
	Total LTL	4.23	4.24
LoRaWAN communication (mJ)		333.17	375.75

TABLE II
RUNTIME ENERGY CONSUMPTION OF LTL FUNCTIONS

3) *Autonomy estimation*

Table III shows the estimation of the end-device's battery life using previous results (given in Table I). The scenario is the following: first, the end-device is sensing and sending a lightweight transaction containing 24 bytes of sensor data (i.e. dynamic energy). Then, the microcontroller is in sleep mode for 120 seconds (i.e. static energy). In sleep mode, the end device's current consumption is about 100 µA (ultra-low power). Therefore, the total energy consumption is the sum of static and dynamic energy. There are two 2.4 Ah batteries that are considered perfect, delivering a voltage of 3V until complete discharge.

	Without LTL	With LTL
Data sent by LoRaWAN	24	154
ΔE static (mJ)	51.89	51.89
ΔE dynamic (mJ)	311.31	370.65
ΔE total (mJ)	363.20	422.54
Estimated battery life (days)	200	173

TABLE III
ESTIMATED DAYS OF AUTONOMY SENDING 24 BYTES OF SENSOR DATA (WITH AND WITHOUT LTL)

Number of messages	Substrate Broadcast	Substrate Finalized	Fabric Commited
1	389.80	16476.71	3511.50
16	1200.24	18175.43	3444.26
32	2114.49	18520.19	4479.25
48	2493.45	19636.03	5472.69
64	4018.58	20374.68	6825.06

TABLE IV
AVERAGE LATENCIES OF THE GATEWAY MODULE API IN MILLISECONDS

With:

$$\begin{aligned} Estimation(hours) &= \frac{E_{battery}(Wh)}{E_{total}(W)} \\ E_{total}(W) &= \frac{E_{total}(J)}{Period(s)} \end{aligned} \tag{1}$$

The end-device battery life drops thus by 13.5% with the use of LTL.

B. Results on the gateway module API and blockchain

In our work, a gateway module API was deployed to facilitate the IoT devices' interaction with multiple types of blockchain smart contracts. The API has multiple tasks to execute, such as the parsing of the LTP protocol messages and the creation of valid transactions. In the following experiment we measured the average latency of each LTP transactions from its arrival at the gateway module API (delivered by the LoRaWAN radio module) until the blockchain transaction (created from the LTP) is committed (i.e. finalized) on the blockchain.

The maximum number of LoRaWAN packets that can be received in parallel by the LoRaWAN radio module is 64, which also means that the maximum number of MQTT packets (parsed LoRaWAN packets) present in parallel at the gateway module API cannot exceed this number. It should be noted that the API's latency is highly dependent on the number of simultaneous packets received and the throughput of the blockchain (the time taken to consider a transaction finalized in the blockchain state). Table IV shows for the two blockchains the average latencies when the MQTT packages arrive in parallel, and the gateway module API instantiates 64 threads to handle the LTP package parsing, transaction creation, and their forwarding.

The benchmark can also be considered as the simulation of the LoRaWAN radio module packets forwarded via the Chirpstack bridge (depicted in Fig. 2), which forwards MQTT packages simultaneously to the gateway module API. The generation of MQTT packages is performed by a PC that forwards the generated packages to the gateway using TCP IP transmission. For the package content generation, 64 different end-device public keys were used, and the public key for content creation was chosen randomly.

1) Benchmark and settings

The gateway module API is executed on a Raspberry Pi model 3B+ (Quad-core Cortex-A53 @1.2GHz) as this hardware architecture is one of the most popular and cheapest solutions for LoRaWAN gateway deployment.

Substrate blockchain network contained five nodes, and they were implemented in a cluster environment with a total capacity of 282 GB of RAM and 192 CPU cores. The Aura consensus algorithm is applied to create blocks and a second consensus algorithm called GRANDPA to finalize the blocks and transactions.

The deployed Hyperledger Fabric network consists of two peers and one orderer node (official test network) and was executed on a 16-core 11th Gen Intel(R) Core(TM) i7-11850H @ 2.50GHz architecture. The peers and orderer use the Practical Byzantine Fault Tolerance (PBFT) consensus algorithm.

In order to make our experiments more realistic, the implementation locations are different, the gateway and the Substrate blockchain are located in France, and the Hyperledger Fabric implementation is located in the Netherlands.

2) Latency results

The previously explained setup provides us with the results depicted in Fig. 6. The simulation sends messages to the module similar to an end-device payload. We can notice that the measurements where made with 5 different configurations.

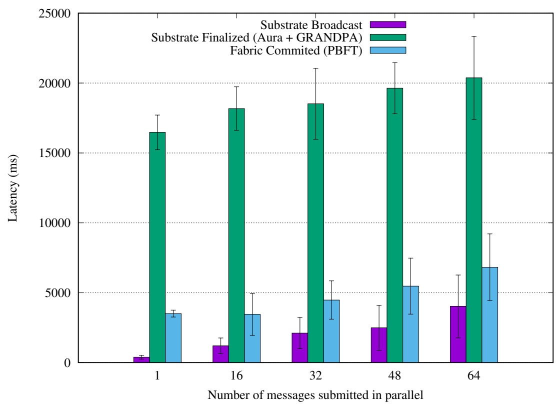

Fig. 6. Gateway Blockchain Module Latencies +/- standard deviation over 5 measurements

The first observation is that the latency of the module increases when submitting more messages to the gateway, independently of the blockchain used. The latency depends on the message targeted blockchain (Substrate based blockchain or Hyperledger Fabric) and depends on the blockchain transaction finalization we choose. As explained previously the module will always wait for the HTTP/ws status after sending a transaction.

In the case of a Substrate based blockchain, a transaction status can either be *broadcasted*, *inblock*, or *finalized*. When only *broadcasting* the transaction, the module does not wait if it has reached blockchain consensus. When waiting the *finalized* status, the transaction has reached the consensus process and thus stored the transaction in the blockchain ledger (i.e. using a block time of 6 seconds, the consensus GRANDPA is reached after 3 block time, thus 18 seconds).

In the case of Fabric, a transaction status is known directly after sending the transaction (*pending*, *failed*, or *committed*), thus when successful, a transaction is *committed* and has reached the consensus (PBFT) process. The following module latencies are obtain when sending only one message: 1) 389.80 ms for a Substrate broadcast transaction, 2) 16476.71 ms for a Substrate finalized transaction, 3) 3511.50 ms for a Fabric committed transaction.

The latency of the Substrate broadcast mode is five times lower than the latency in finalized mode. In the latter, the transaction has to wait three block-time before being finalized. However, in broadcast mode, Substrate only validates and queues transactions. Hyperledger Fabric latency is close to Substrate broadcast mode, which is due to PBFT consensus algorithm.The finalized latency in Substrate is higher than in Fabric due to performing two consensus algorithms (Aura and GRANDPA).

If the use case requires ensuring that the transaction has been stored on the chain, the gateway blockchain module has to be configured to wait the finalized and committed status, thus increasing latency. This will allow also to implement a downlink message in the case of a transaction fail. However, in some application, it is allowed to use the broadcast status, resulting in a much lower latency compared to using finalized or committed statuses.

Furthermore, the gateway can process multiple messages at the same time (batches of 16, 32, 48 and 64 messages). The increase in latency shows the hardware limits of the gateway that cannot allow the module to increase indefinitely the number of threads. One batch of 64 transaction has an average latency of 6825.06ms (using Fabric). Meaning each message will create a new thread and takes on average 6825.06ms to be committed to the ledger. Sending too many messages will overflow the number of threads and in the same way increase latency drastically.

The results in Table IV are a key element to understand the maximum LoRaWAN uplink duty cycle of the end-device. If we have a high probability to send multiple messages at the same time, the end-device uplink should at least have a duty cycle greater or equal to the corresponding latency of the number of messages submitted in parallel (e.g., high probability of 16 LoRaWAN messages in parallel requires the end-device uplink duty cycle to be at least 3444.26ms when using Fabric).

V. Conclusion

The principal objective of our research is thus completely reached. We have shown that even in Low-Power IoT devices we are able to interact with two (or more) blockchains and smart contracts. Moreover the devices are authenticated by the smart contracts, respecting the ECDSA signature verification.

To the best of our knowledge, our proposed work is the first implementation of a lightweight protocol integrating the LoRaWAN communication protocol for allowing constrained devices to interact with multiple blockchains. Our proposed protocol allows sending data to the Substrate and Hyperledger Fabric blockchains with a relatively low overall energy consumption of 422 mJ for sending 24 bytes of data. The experiments also highlight that the end-device battery life drops by 13.5% when using LTL, which can be considered acceptable as it also offers the possibility of interaction with multiple blockchains. Furthermore, the proposed solution contains a gateway module API which includes the SDKs of the blockchains mentioned above, handling the valid blockchain transaction creation and sending. The results show that 64 simultaneously forwarded LoRaWAN packages can be converted into transactions and committed (stored in the blockchain state) with a latency of 20.3s for Substrate and 6.8s for Hyperledger Fabric blockchain. This result indirectly means that if a new flow of 64 packets arrived at the gateway within the mentioned latency, the gateway module API and the blockchain will not be able to handle all of the transactions, and some of the transactions will be rejected. Thus the duty cycle and the number of devices per gateway must be chosen according to these latencies. Details about the LTL library and the LTP protocol, along with the way of sending the MQTT messages to the blockchains are detailed in a github repository[1].

VI. Acknowledge

This work has been supported by the French government, through the ANR-19-CE25-0008 and by the European Research Council (ERC) Starting Grant ResolutioNet (ERC-StG-679158). We want to thank Georgios Smaragdakis for his constructive comments and feedback.

References

[1] E. Androulaki *et al.*, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538

[2] "Substrate Developer Hub," https://substrate.io, 2022.

[3] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with IoT. Challenges and opportunities," *Future Generation Computer Systems*, vol. 88, pp. 173–190, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17329205

[4] Z. Zibin, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 4, pp. 352–375, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17329205

[5] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, Aug 2001. [Online]. Available: https://doi.org/10.1007/s102070100002

[1] Available on: https://github.com/KRolander/blockchain-access-LTP-LoRaWAN

[6] N. Szabo, “Formalizing and Securing Relationships on Public Networks,” *First Monday*, vol. 2, no. 9, Sep. 1997. [Online]. Available: https://firstmonday.org/ojs/index.php/fm/article/view/548

[7] M. Pincheira and M. Vecchio, “Towards Trusted Data on Decentralized IoT Applications: Integrating Blockchain in Constrained Devices,” in *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2020, pp. 1–6.

[8] M. Pincheira, M. Vecchio, R. Giaffreda, and S. S. Kanhere, “Cost-effective IoT devices as trustworthy data sources for a blockchain-based water management system in precision agriculture,” *Computers and Electronics in Agriculture*, vol. 180, p. 105889, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168169920330945

[9] G. Wood *et al.*, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, 2014.

[10] R. Kromes, L. Gerrits, and F. Verdier, “Adaptation of an embedded architecture to run Hyperledger Sawtooth Application,” in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2019, pp. 0409–0415.

[11] L. Gerrits, R. Kromes, and F. Verdier, “A True Decentralized Implementation Based on IoT and Blockchain: a Vehicle Accident Use Case,” in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–6.

[12] Y. Huang *et al.*, “Characterizing EOSIO Blockchain,” *arXiv e-prints*, p. arXiv:2002.05369, Feb. 2020.

[13] “The Things Network,” https://www.thethingsnetwork.org, 2022.

[14] J. Haxhibeqiri, E. De Poorter, I. Moerman, and J. Hoebeke, “A survey of lorawan for iot: From technology to application,” *Sensors*, vol. 18, no. 11, 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/11/3995

[15] “ChirpStack,” https://www.chirpstack.io, 2022.

[16] S. R. Niya, S. S. Jha, T. Bocek, and B. Stiller, “Design and implementation of an automated and decentralized pollution monitoring system with blockchains, smart contracts, and lorawan,” in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–4.

[17] A. Banks and R. Gupta, “MQTT Version 3.1.1. OASIS Standard.” *http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html*, 2014.