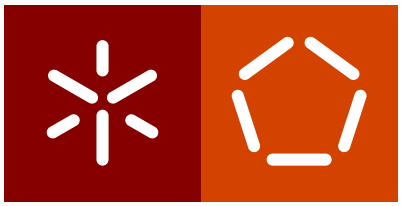
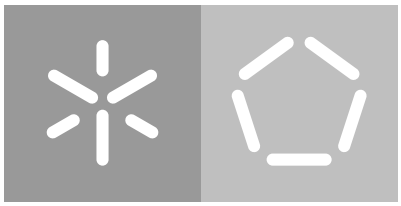


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno André Lopes Leite

A Secure IoT Communication System for Smart Contracts

February 2021



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno André Lopes Leite

A Secure IoT Communication System for Smart Contracts

Dissertation report
Integrated Master's in Informatics Engineering

Supervised by
Professor Doutor Alexandre Júlio Teixeira Santos
Professor Doutor Nuno Vasco Moreira Lopes

February 2021

ACKNOWLEDGEMENTS

To my parents, Carlos Leite and Ana Leite, for their unconditional love and support throughout my journey, ultimately leading me where I got today. To my brother, also Carlos Leite, for his precious advices, which directly impacted my academic path, placing me, today, in the area I love to work. Last but not least, to my fiancée and soon to be wife, Diana Tavares, for her love and extreme patience in the countless weekends of watching me typing away in a keyboard.

In addition, I would also like to deeply thank my supervisor, Professor Alexandre Santos for his direct and concise advices and supervision, providing valuable insights for the improvement of the work done in the dissertation and my co-supervisor and company supervisor Professor Nuno Lopes, for the opportunity to work in such an exciting project, which became my masters thesis.

I would also like to extend my grattitude to my colleagues at the *Digital Transformation CoLab* for the brainstorming sessions and the lessons learned from them.

Finally, a big thank you to my friends for providing me with the always necessary confidence, support and invaluable friendship that lifted my spirits.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

The need to ensure the confidentiality and integrity of data generated in industrial systems and applications has been increasingly highlighted over the years, due to the clear and urgent requirements of not disclosing sensible proprietary information and ensuring that data is kept immutable since it is generated until it is permanently stored.

It is from these two main ideas that this dissertation is created, framed in a project that is being developed at the *Digital Transformation CoLab* with *Bilanciai* and *Cachapuz*. These are the industrial partners and key stakeholders of this project, having identified the requirements for the weight measurement process that occurs in the weighing stations that are placed in their customers. This dissertation essentially consists on the definition of a secure *Internet of Things (IoT)* communication system between the devices that operate on the weighing stations of the customers and on top of that, develop a smart contract application using blockchain technology capable of: i) automating the process of verifying the correct application of weighing guidelines; and ii) registering and storing "receipts" of weighings that take place in the customers' weighing stations.

In this dissertation, a revision of the state of the art is made with the goal to perceive the most secure and current technologies capable of providing the required functionalities, which are the fuel for the identification of the problems and challenges that such a project might face, ultimately leading to the design of a solution that can both: i) mitigate the aforementioned problems and challenges; and ii) comply with the goals defined for the dissertation. Additionally, in this document, the development of such a solution is also explored by providing clear insights into the decisions that were made and the reasoning behind them and by implementing components that are able to provide registration and rich querying of weighing tickets (receipts), weighing ticket building and secure communication as well as the enforcing of a blockchain network structure that fosters data confidentiality.

Ultimately, results are shown, collected from a proof of concept, which essentially provide evidence on the functional correctness of the system that was built, i.e., its ability to grant the retainment of weighing ticket characteristics and the capabilities of the communication system, which demonstrates to be able to securely build and transmit weighing tickets, with fault tolerance.

The outcomes of this project can be integrated into existing systems of the industrial partners to increase efficiency, security and business innovation.

Keywords— blockchain, smart contracts, iot, security, compliance

RESUMO

A necessidade de assegurar a confidencialidade e a integridade dos dados gerados em sistemas e aplicações industriais tem sido cada vez mais destacada ao longo dos últimos anos, devido a claros e urgentes requisitos de não divulgar informação proprietária e de garantir que essa informação permanece imutável desde o momento em que é gerada até ao ponto em que é guardada permanentemente.

É a partir destas duas ideias principais que esta dissertação é criada, enquadrada num projeto que está a ser desenvolvido no *Digital Transformation CoLab* com a *Bilanciai* e a *Cachapuz*. Estes são os parceiros industriais e *stakeholders* do projeto, tendo identificado os requisitos para o processo contínuo de medição de pesagens que ocorre nas estações de pesagem dos seus clientes. Esta dissertação consiste, essencialmente, na definição de uma comunicação segura em IoT entre os dispositivos que operam nas estações de pesagem dos clientes e, complementarmente, desenvolver uma aplicação baseada em *smart contracts* utilizando tecnologia *Blockchain* com o intuito de: i) Automatizar o processo de verificação da aplicação correta de diretrizes de pesagem; e, ii) Registar e armazenar "recibos" de pesagem que são efetuadas nas estações de pesagem dos clientes.

Nesta dissertação, a revisão do estado da arte é feita com o objetivo de entender as tecnologias mais atuais e seguras capazes de providenciar as funcionalidades adjacentes aos requisitos, o que se torna na base para a identificação dos problemas e desafios que um projeto desta natureza pode enfrentar, resultando, em última instância, no desenho de uma solução que consiga: i) Mitigar os problemas e desafios anteriormente mencionados; E, ii) Cumprir com os objetivos definidos para esta dissertação. Adicionalmente, neste documento, o desenvolvimento da solução é explorado, ao fornecer informações claras sobre as decisões que foram tomadas e o raciocínio por trás das mesmas e ao implementar componentes capazes de fornecer o registo e consulta avançada de recibos de pesagem, construção e transmissão segura dos mesmos, como também a capacidade de estruturar e assegurar uma organização da rede blockchain que promove a confidencialidade de dados.

Finalmente, resultados são ilustrados, extraídos de uma prova de conceito, fornecendo provas da correcção funcional do sistema construído, isto é, a sua capacidade para garantir a manutenção das características dos recibos de pesagem e, além disso demonstra a capacidade do sistema de comunicação em transmitir, de forma segura, os recibos de pesagem, com tolerância a falhas.

Os resultados obtidos neste projeto têm a possibilidade de ser integrados em sistemas existentes dos parceiros industriais com o objetivo de aumentar a eficiência, segurança e inovação nos seus modelos de negócio.

Palavras chave— blockchain, contratos inteligentes, iot, segurança, compliance

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

A

API Application Programming Interface.

ARM Advanced RISC Machine.

C

CA Certificate Authority.

CAS Certificate Authorities.

CLI Command Line Interface.

COAP Constrained Application Protocol.

CPU Central Processing Unit.

D

DI Department of Informatics.

DTLS Datagram Transport Layer Security.

DTX Digital Transformation CoLab.

E

EVM Ethereum Virtual Machine.

H

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

I

IETF Internet Engineering Task Force.

IOT Internet of Things.

IP Internet Protocol.

J

JSON Javascript Object Notation.

JWT JSON Web Tokens.

K

KG Kilograms.

M

M2M Machine-to-Machine.

MB Megabytes.

MIEI Integrated Masters in Informatics Engineering.

MQTT Message Queuing Telemetry Transport.

MSP Membership Service Provider.

O

ODM Object-Document Mapping.

P

PBFT Practical Byzantine Fault Tolerance.

PL Programming Language.

POS Proof of Stake.

POW Proof of Work.

R

RAM Random Access Memory.

REST Representational State Transfer.

S

SSH Secure Socket Shell.

T

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TTL Time To Live.

U

UDP User Datagram Protocol.

UM University of Minho.

URL Uniform Resource Locator.

1

INTRODUCTION

This dissertation describes the investigation and development performed in the context of the *Integrated Masters in Informatics Engineering (MIEI)* at the *Department of Informatics (DI)* from *University of Minho (UM)* and a project developed by the *Digital Transformation CoLab (DTx)* along with *Bilanciai* and *Cachapuz*.

The aim of this chapter is to provide an initial context and understanding on what the project is and how it fits in the landscape of the companies' business, to declare the motivations behind the rise of such a project, its primary goals and, finally, to detail how this dissertation is structured in order to provide the best possible understanding on the work done in it.

1.1 CONTEXTUALIZATION

Bilanciai and *Cachapuz*'s main business is based on the supply of weighing solutions. Specifically, in the context of this dissertation, their industrial weighing solution is explored. In this solution, their clients are essentially companies with weighing stations that measure the weight of some loads, most of it are trucks. Each of these weighing stations may possess up to 4 weighbridges, which are the physical devices where the trucks stop to be measured. Each of these weighbridges may hold up to 8 load cells. A load cell is the device that actually measures the weight that is put upon it and, through the sum of the combined weights of all load cells, passing them through a proprietary compensation algorithm that increases the precision of the measure, the total weight of the load is obtained.

It is this use case that the project associated with this dissertation studies, by working on two main domains:

- The **Physical** domain by redesigning the company's load cells and upgrading them in terms of sensors and electronics;
- The **Cyber/Digital** domains by building a blockchain-based cloud platform to secure and analyze data sent from the stations and the load cells, as well as the definition of a secure IoT communication system.

This dissertation focuses on the **Cyber/Digital** domains, specifically, in a security point of view, by implementing systems, mechanisms and processes that allow that the weighings performed by the station's meet the requirements of periodic compliance processes that are put in place by standardization organizations, such as *WELMEC* [1].

1.2 MOTIVATION

The main motivation behind the development of this dissertation is, as said before, associated with the security of the information that is produced at the weighing stations, namely, the weighing tickets. A weighing ticket can be defined as a sort of receipt that a certain weighing took place at that exact time, with the weight that was measured, in the weighing bridge that measured it.

For the companies it is essential that they can ensure that those tickets remain immutable, i.e., that no external force has the ability to change certain properties of it. This is essential due to the aforementioned compliance processes, in which standardization companies verify the correct application of weighings (with well maintained and well functioning weighbridges) and, additionally, it serves as a connecting piece of information to other weighings when, for example, the weight of a truck has to be controlled along a certain route, in which that truck will be wheighed multiple times.

Essentially, the motivaton for this project arises from the lack of mechanisms and processes to control the emission and validity of those weighing tickets, which are a center-piece to the solution that they provide and in which this dissertation focuses.

1.3 GOALS

Recognizing the context in which this industrial solution is placed and the motivation for the development of a system capable of guaranteeing the immutability of the weighing tickets, the main goals of the dissertation can be described.

In overview, the goals of this dissertation lie in three areas:

- **Research**. To conduct research in the solutions and technologies that can best accomodate the solution to the aforementioned problem;
- **Development**. To implement a solution capable of handling the problem in focus;
- **Evaluation**. To obtain functional results capable of demonstrating that the solution is, in fact, capable of dealing with the problem in focus.

Acknowledging the importance of these three previously mentioned areas, some specific goals are set to better contextualize and evaluate the dissertation and the solution built in it:

1. Research blockchain & smart contract solutions that best suit this information immutability use case;
2. Develop a smart contract application able to immutably store weighing tickets and provide query mechanisms;
3. Develop a support platform that exposes *Application Programming Interface (API)*s for communication with the smart contract application and, additionaly, ensure its correct usage;
4. Establish a protocol definition for a secure communication between: i) the weighing stations and the cloud platform; And ii) the devices that operate on the weighing stations;
5. Provide a proof of concept demonstrating the usability of the platform and the communication system's proper functioning.

1.4 DOCUMENT STRUCTURE

Finally, to end this chapter, the structure of this document is presented and described in order to provide an insight on how the research & development process was conducted.

In addition to this introductory chapter, this dissertation is composed by five more chapters, structured as follows:

Chapter 2: State of the art explores the literature review covering a range of topics that are essential to understand and acquire knowledge in the necessary areas to develop the project, such as Blockchain, IoT and Blockchain & IoT Integration.

Chapter 3: Problem, challenges and proposed solution addresses the problems and challenges that this dissertation presents and proposes an architecture of a technological solution that is able to overcome said problems.

Chapter 4: Development addresses the decisions that were made as well as the specifics on how the solution was designed and built, focusing in a later instance on the outcomes of the developed solution.

Chapter 5: Proof of Concept explains how the solution that was built addresses the required problems by experimentation, i.e., an experiment scenario is created to provide the necessary assurance that the solution is able to fulfill all the proposed goals.

Chapter 6: Conclusions draws the main conclusions about the work done thus far, presenting some considerations on the prospect for future work to be done in each of the components.

2

STATE OF THE ART

In this chapter, the state of the art for the concepts and technologies approached in this dissertation is discussed. Several publications and solutions are analyzed in order to provide some clarification on the current technological landscape in each area. From the analysis of the goals of this dissertation, three main topics emerge as essential to serve as basis for the proposed solution: i) Section 2.1: IoT; ii) Section 2.2: Blockchain & Smart contracts ; And iii) Section 2.3: Integration of IoT and blockchain solutions .

2.1 INTERNET OF THINGS

IoT is one of the trending and growing topics of this millenium, due to the ever growing necessity of connecting *things* in order to improve information gathering as well as processes automation and it can be defined as an Internet-enabled architecture that facilitates the interconnection of devices with multiple technologies, fostered by *Machine-to-Machine (M2M)* communication protocols which ultimately result in better control of the goods and services that are exchanged or manipulated by industrial or comercial applications [2, 3].

The inherent capacity of IoT technology to *digitize* the physical world, i.e., communicating and generating the digital information that represents the physical state of *things* make it one of the clear enablers of Industry 4.0 [4].

Through this enormous capacity, the technology has the potential to be used in a huge number of applications such as:

- **Industry** - Predictive and prescriptive maintenance fostered by data collected from sensors in operating machines;
- **Agriculture** - Prescriptive actions on fields fostered by data collected from sensors in plantations;
- **Healthcare** - Remote patient monitoring; Faster item location in hospitals; Connected contact lenses;
- **Transportation** - Fleet management; Optimal Route; Public Transit Management.

Regardless of the application, IoT's structure is usually three-layered, composed by the Physical, Network and Application Layer [5]. The physical layer refers to the component of the structure responsible for device identification and discovery as the backbone for direct or indirect communication over the Internet between the devices. The network layer refers to the channels and interfaces responsible for the point-to-point transmission of information from an emitter to a receiver, aided by communication protocols designed for the effect, as explored in section 2.1.2. The application layer can be seen as the layer that makes sense of the data that is being collected by the devices, since it is responsible for its storage and service provisioning from the data.

Considering the purpose of this dissertation, it is more than obvious that IoT's application in this context refers to the industry, a sector that usually has some specific conditions and design goals for it [6, 7]:

- **Energy awareness** - The IoT devices should withstand long periods of operation with no need to recharge. The time frame depends on context but it may be up to years;
- **Low latency** - Fast and efficient data transmission and processing is necessary, for real-time operations;
- **High throughput** - Be aware of the maximum information that can be sent on the network at a given time;
- **Scalable** - Make it easy to increase devices in the network and application, improving and widening data collection;
- **Controlable resources** - Establish clear rules for the communication direction and communication partners, i.e., who communicates with whom;
- **Secure** - Ensure that the data transmission is safe, abiding to standardized properties of information security.

In the remainder of this section, the challenges that are presented when trying to comply with the aforementioned design goals are discussed as well as secure communication protocols that are capable of solving some of them.

2.1.1 *Challenges*

The very nature of IoT as a concept that fosters data exchange and is fostered by M2M communications, also makes it a challenging one to implement since those two concepts immediately raise many challenges [2, 4, 5, 7]:

- **Scalability** - The number of active IoT devices will continue to increase and current architectures may not be capable of withstanding or be prepared to deal with that change. New management protocols that can scale well are needed;
- **Heterogeneity** - Many devices from different manufacturers, using different protocols and different stacks make it difficult for seamless communication between them. More standardization efforts should be put in place;
- **Energy efficiency** - Devices have to be built so that they can last a long time energy-wise, which in turn usually constrains the device to really low resources;
- **Mobility management** - Mobile IoT devices can harm network protocols and mechanisms put in place to handle communications, since many of them are not suited for mobile IoT communication, due to severe energy and resource constraints;
- **Security and Privacy** - Probably the biggest challenge that IoT faces, the need to establish and implement secure protocols and mechanisms to provide a secure M2M communication, keeping in mind that most of the devices cannot run heavy protocols due to their restraints in terms of resources.

Section 2.1.2 dives deeper into existent communication protocols that are currently being used to mitigate some of the problems discussed, especially security, privacy and trust.

2.1.2 *Secure Communication Protocols*

A secure communication protocol is nothing more than a set of rules that define the way multiple devices can communicate in a safe way. A communication can be considered secure when, at the very least, it respects the following properties [8, 9]:

- **Confidentiality** - Property that states that data can only be seen by an authorized entity, i.e., an entity should posess an identity that allows it to see the data being communicated;
- **Authenticity** - An entity that communicates with another has to always be certain that the entity it is communicating is who it says it is;
- **Integrity** - Information cannot be manipulated during its lifecycle, i.e., data sent by an emitter has to be exactly the same data received by a receiver.

Over the last years, new protocols specifically designed to be scalable and light enough to be used in contrained environments have been developed. Some of them rely on already implemented infrastructure and protocols such as Ethernet or Wi-fi and others rely on new infrastructure protocols like *6LoWPAN* [10]. Protocols that are in use nowadays rely on

communication patterns of type publish-subscribe, response-request or both. Table 1 shows a comparison between some of the most currently used communication protocols in terms of their relevant communication atributes [3, 10]. The *Hypertext Transfer Protocol (HTTP)* [11], *Constrained Application Protocol (CoAP)* [12] and *Message Queuing Telemetry Transport (MQTT)* [13].

Attribute / Protocol	**CoAP**	**MQTT**	**HTTP**
Transport protocol	UDP	TCP	TCP
Communication type	asynchronous	asynchronous	synchronous
Communication pattern	publish-subscribe & request-response	publish-subscribe	request-response
Communication paradigm	polling	event-driven	long polling
Security	DTLS	TLS	TLS

Table 1: Comparison between diferent communication protocols

HTTP

Certainly, one of the most successful communication protocols in the application layer, due to its use in the World Wide Web. The protocol implements synchronous communication based on a one-to-one request-response model, i.e., a client requests data/services to a server, and the server responds with the data/services requested. The message content in HTTP communication is encoded in ASCII text. Despite all its prays, including the fact that it is *RESTful* which is a good feature when working with data, HTTP has a significant downside when used in contrained environments, the header size is around multiple KB, which is overwhelming for constrained devices, taking significant time and significant amount of energy to process [11].

CoAP

CoAP is also an application layer protocol, built specifically to be excellent in establishing communication in constrained environments with an IPv6-based infrastructure. CoAP can communicate both in a request-response model or in a publish-subscribe model. This protocol uses a short fixed size header, which joined with the utilization of the *User Datagram Protocol (UDP)* make it a very lightweight protocol in terms of message size [12].

MQTT

MQTT has a similar purpose to CoAP, it is targeted to be excellent in handling communication in constrained environments, but it has two significant differences:

- While MQTT uses a one-to-many communication type, CoAP uses a one-to-one communication type;
- While MQTT uses the *Transmission Control Protocol (TCP)* in the transport layer, CoAP uses UDP.

Aside from those two differences, both have low data overhead, which is essential when targeting contrained environments for the protocol's utilization.

From the standpoint of these protocols alone, secure communication is not guaranteed, since each one of them communicates clearly without applying mechanisms to ensure confidentiality and integrity of the data. In order to secure the transmission of the data using one of these protocols, there are really only two choices:

1. Implement a proprietary security protocol for the communication that extends the aformentioned application-layer protocols;
2. Use standard security protocols coupled with an application layer protocol.

While the first choice may offer more flexibility in the implementation and how the protocol itself works it has essentially two significant downsides:

- It is a complex task to implement a proprietary secure communication protocol, that works in distributed environments;
- The proprietary implementation may introduce severe security bugs which are harder to found since the protocol is proprietary and thus, not community-validated.

On the other hand, using a standard secure protocol removes the complexity of implementation of it, although it still has to be integrated in the communicating software and, since they are open-source, they are constantly being validated and corrected in order to prevent severe security bugs.

The most clear example of a secure, validated and standard secure communication protocol is without a doubt *Transport Layer Security (TLS)* [14], which is the underlying transport protocol used in *Hypertext Transfer Protocol Secure (HTTPS)*. It allows for the validation of only one of the entities or the validation of both and ensures that after the entities "handshake" the communication between them is secured. This security protocol is comprised by two stages: i) the handshake stage; And ii) the application data exchange stage. In the first stage client and server exchange the supported versions of the protocol as well as the session key that is going to be used in the second stage. Additionally, in this stage, the server certificate is validated and, if required, the client certificate is also validated. After this first stage, both entities can securely communicate the necessary application data, resting assured that

confidentiality and integrity of the data will be assured by using the session key previously established [14].

Although TLS is an extremely secure protocol, and that is also why it is widely used, statement that can be validated from the fact that it is the underlying secure communication protocol of HTTP, which is, probably, the most widely used application protocol, it is a computationally expensive protocol to use. Apart from all the cryptographic techniques that need to be applied, TLS uses TCP as the basis for its transport protocol which, due to the application of various control mechanisms to ensure transmission, is a heavy protocol in terms of header size and the rate of message exchange, thus making it a risky choice for use in restrained environments.

In order to counter some of these problems, the *Internet Engineering Task Force (IETF)* standardized the *Datagram Transport Layer Security (DTLS)* [15], which is a secure communication protocol in all things similar to TLS, except for the fact that it uses UDP as the basis for the transport protocol, thus making it less heavy, both in header size as well as in the rate of message exchanges which, ultimately saves computation energy and time.

Summarizing, the application protocols described in Table 1 guarantee uniformity and consistency on how the data is transmitted from one side to the other but, to ensure that the data that is transmitted is secured, a specific secure communication protocol has to be used so that the joint use of the application-layer and transport-layer protocols guarantee uniformity, consistency, efficiency and security.

2.2 Blockchain & Smart Contracts

With the introduction of Bitcoin in 2008, as proposed in [16], along came a more important concept, that of a distributed ledger that records every transaction ever made in the system where it is applied in an immutable manner, the Blockchain. With the years passing from its introduction, researchers and the industry started to realize the immense potential of the technology since it could be applied not only in electronic cash systems, as well as in many other situations where data immutability and traceability is of the utmost importance like, for example:

- **Law** - To provide transparency and data integrity on law related services;
- **Media and Entertainment** - To combat digital piracy by tracking content's lifecycle;
- **Supply Chain** - To provide data transparency, integrity and traceability to the currently confusing supply chains.

The functioning of the blockchain itself is completely reliable on cryptographic functionality such as public key cryptography, to provide ways for users to digitally sign and transact

in the system, and cryptographic hash functions that are used in many ways as it will be seen in 2.2.1. Before providing a more technical insight to what the blockchain really is and the operations it performs, it is important to keep in mind some of its core concepts [17] :

- **Pseudo-anonimity** - The accounts used to transact with the blockchain should, in most cases, anonimize the user that holds the account;
- **Traceability** - The data in the blockchain is traceable because it is never overwritten, it is just added to the chain;
- **Distributability** - The blockchain should be used in a distributed manner, to increase attack resiliency.

2.2.1 *Blockchain Fundamentals*

The purpose of this section is to provide insight on the blockchain's architecture, i.e., the components and processes that make it work and current alternatives to some of its components that are currently being researched and proposed. The structure of blockchain technology is commonly represented as a chain of blocks, in which a block holds multiple transactions.

The aforementioned structure is used in any type of blockchain. A blockchain can be of type:

- **Public** - Every entity can participate and each user is always anonimized;
- **Private** - Usually belonging to a single organization for a more permissionable control over the chain;
- **Consortium** - A set of nodes, maybe belonging to different organizations, can use the chain.

In Table 2 a comparison is made between the three types of blockchain, by using attributes deemed essential for the application of this type of technology. From the table, some important information can be retrieved.

- In terms of consensus, the public blockchain needs all miners' approval, while the private and consortium blockchains need a determined set of nodes (Centralized to one in the first case);
- The way the consensus is processed in each type leads to evident high efficiency values in the private and consortium blockchains, since they are very fast at verifying blocks, and low efficiency for public blockchains (long block generation and validation times);

- Tampering is nearly impossible in the public blockchain due to its high distributability, since an attacker would have to, in theory, subvert half of the network to tamper with the blockchain;
- The private and consortium blockchains' tampering proof is directly related to the size of the network that the companies implement as well as to the access control measures taken by them to prevent unwanted access.

Property	**Blockchain Type**		
	Public	**Private**	**Consortium**
Consensus	all miners	centralized	pre-determined nodes
Efficiency	low	high	high
Read permission	public	restricted	restricted
Write permission	public	restricted	restricted
Tampering	nearly impossible	possible (admin access)	hard
Distributed	yes	no	partially

Table 2: Comparison between types of blockchain

Despite the relevant differences between the existent types of blockchain, the underlying architecture of the technology is more or less the same, as shown in Figure 1 [18]:

- **Ledger** - The chain of blocks, which hold the records ever registered in the blockchain;
- **Consensus** - The algorithm by which the network abides in order to publish and validate new blocks to the blockchain;
- **Nodes** - The participating nodes of the blockchain. They must have, at the very least, read permission to be considered a node. Holds a copy of the blockchain and validates blocks that are going to be registered;
- **Miners** - Special kind of nodes that are responsible for introducing a new block to the blockchain, in accordance with the consensus algorithm, that has to then be validated.

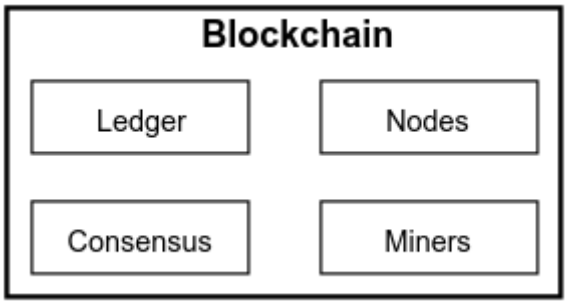


Figure 1: The components of a Blockchain network

Ledger

The ledger can be conceptually seen as a list of linked blocks, as we can see in Figure 2 for example.

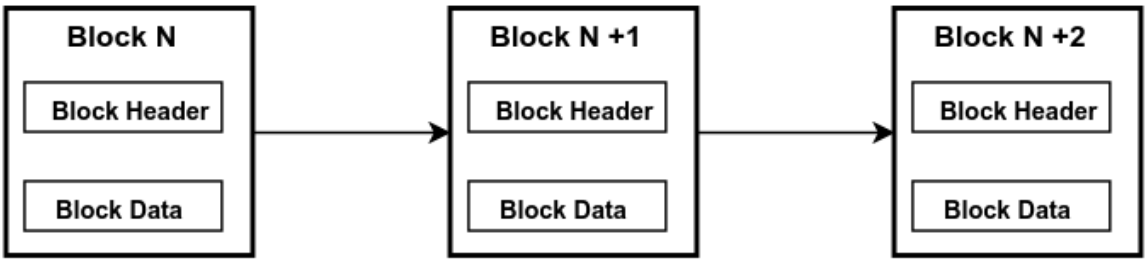


Figure 2: Visual representation of the ledger

Each block has its set of properties, that when combined in chain, make the data in the network immutable. The structure of a block is as follows in Figure 3 [19].

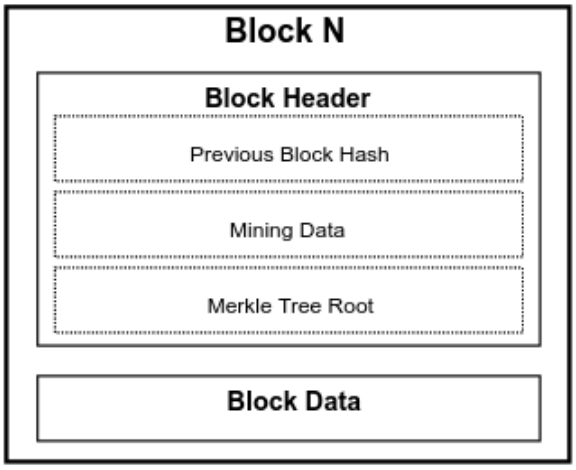


Figure 3: Components of a block in the blockchain

- **Block Header** - Comprised of metadata of the block:
  - **Previous Block Hash** - The cryptographic hash of the previous block's header in the chain (strong linkage between blocks);
  - **Mining Data** - Data needed for the mining process. The *nonce* which is a value used to introduce randomness to the block's data, the timestamp and the difficulty of mining the block;
  - **Merkle Tree Root** - A data structure that summarizes the transactions that exist in a block.
- **Block Data** - The list of transactions of this block.

The fact that each and every block, except the first (*genesis block*) holds the hash of the previous block, is what renders the blockchain invalid if any block is altered after its creation, since if block N is changed, then its hash will be certainly different, and block N+1 will no longer hold a valid hash. Summarizing, after a block has been published, it can never be removed or altered.

Due to the fact that the hash in each block's header is the hash of the header of the previous block, might make one think that the transactions are not being accounted for, since if the transactions aren't part of a block header, then changes to them will not affect the hash of the block header. And it is here where the *Merkle Tree* structure comes in.

The merkle tree is also known as a binary hash tree and the main ideia behind this tree authentication method is to recursively hash pairs of nodes (transactions) in order to produce a single root node that authenticates multiple hashes of nodes with a single hash [19, 20, 21].

Figure 4 shows an example of utilization of a merkle tree to authenticate 4 transactions, and as we can see this authentication method uses a kind of *divide-and-conquer* method by recursively hashing nodes in pairs until only one node is left, the root node. In the end of this process, the root node authenticates any of the transactions, and any change to a transaction will invalidate the merkle tree root calculated, which is why the existence of this field in the block header is essential. This technique also offers an interesting feature, since to prove that a transaction belongs to a block we only need $log2(N)$ hashes, where N is the total number of transactions, in order to build the path where that transaction is included.

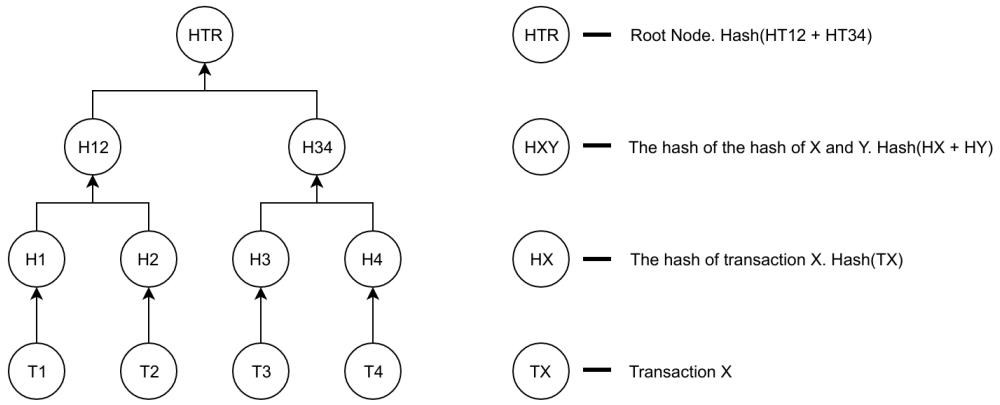


Figure 4: Example of utilization of a merkle tree

Consensus

When one truly thinks about the composition of the blockchain network, it can be imediately denoted that it is a network comprised by untrustworthy nodes, since most of them do not "know each other" and do not have any obligation to comply with the network, therefore it is critical to find some way for the network to reach a consensus on what to do in a distributed environment. This problem was raised and studied in [22] as the *Byzantine Generals Problem* which essentialy states, in summary, that a group of Byzantine generals hold a portion of the army each and some generals prefer to attack and the others prefer to retreat, but if some of them attack and the others retreat, the attack fails, so they have to reach a consensus on what to do [17]. As the blockchain network is a distributed environment, it also needs a consensus algorithm so that the ledger in all of the nodes is always the same and correct.

The choice of the consensus algorithm is an important one, since a weak consensus algorithm can hand huge advantages for adversaries to subvert the network and alter the ledger at their will. There are many propositions and implemented algorithms currently as seen in [17]. A consensus algorithm can usually be classified in two types:

- **Open** - All miner nodes are selectable by the consensus algorithm;
- **Permissioned** - The pool of available miner nodes is already less due to permissioned access.

It is safe to say that the most known open consensus algorithms are the *Proof of Work (PoW)* used by Bitcoin and Ethereum and the *Proof of Stake (PoS)* which will, possibly, start being used by Ethereum in the start of 2020 [23]. *Practical Byzantine Fault Tolerance (PBFT)* is probably the most popular permissioned consensus algorithm due to its utilization in Hyperledger Fabric [24].

The PoW consensus algorithm relies on the concept that the choice of the node that will mine the block cannot be random since it introduces vulnerabilities, instead the chosen node should prove he is not likely to attack the network by performing a lot of work. Generally, in order to publish a new block, each node of the network has to calculate different hash values of the block header by creating new nonces until the result hash respects the rule set by the consensus algorithm, e.g. the integer value of the hash (hexadecimal string) must be lesser or equal to a certain value. When a node successfully calculates a result, it must broadcast the block (with the used nonce) to all other nodes so they can confirm that the calculated hash is correct and, after that, add the block to the blockchain. If, by chance, two miners mine the blocks at the same time and create two branches of the ledger, the resolution of this conflict is made by keeping the longest branch, which will have all the blocks the other branch has, plus at least one.

The PoS consensus algorithm is a more efficient alternative to PoW, energy and time wise, since this algorithm relies on the concept that the miners have to prove ownership of a certain amount of cryptocurrency, and it is believed that the more currency one has the less likely he will want to attack the network. As the choice is solely based in the amount of coin, it obviously tends to be unfair and so beyond the stake held by the user, it is also usually added another type of ”randomness” so that the same user is not chosen everytime. It is obvious that this consensus algorithm is more prone to attacks unless it had already a solid base of users, in which case it is really solid and that is why most known blockchains do not start with a PoS type of algorithm. For example, Ethereum started with a PoW type of algorithm and will now change to PoS since it already has a good user base [17].

The PBFT consensus algorithm states the concept that a new block is mined in a round. In a round, a node is selected according to pre-defined rules (permissions) and this node will be responsible for ordering the transaction. The process is divided in three stages: i) *pre-prepared* ; ii) *prepared* ; And iii) *commit*. The node can only pass a stage if it receives over 2/3 of the votes from the nodes in the network, which means that if an attacker can control or subvert 1/3 of the nodes in the network it can compromise the process, although it is a nearly impossible task [25].

Nodes

A blockchain network is tipically distributed, which means that multiple nodes belong to it. Having that said, the nodes have to be uniquely identified and, usually, in most blockchain implementations a node has two components:

- **Key Pair** - The cryptographic public and private keys unique to this user;
- **Address** - A component that identifies and at the same time pseudo-anonimizes the node in the network, derived from the public key.

Whenever a transaction is sent to this node or for this node, it is sent to his **address** and whenever this node wants to send a transaction to another node he sends it to the node's address , signed digitally with his private key.

Miners

Miners in a blockchain network, besides having all the properties a regular node has, are also eligible for mining/publishing blocks, if they specifically state it so. They exist so that blocks may be published to the network according to the consensus algorithm, but it is obvious that a miner will have to spend huge amounts of computational resources to solve the problems posed by the consensus algorithm which translates to spending real money in hardware and energy. From this point of view, it is clear that there has to be some kind of incentive for miners to actually try to publish a block, and so in every blockchain implementation, a node that tries to publish a transaction has to pay a really small fee for the miner to mine his transaction.

2.2.2 *Smart Contracts*

The term *Smart Contract* was originally first proposed by Nick Szabo in the 1990s in [26] as a digital twin of the already known term of a contract between two entities, which enforces a set of rules and conditions for the utilization or transfer of a given asset. With blockchain technology, this concept can evolve to a point where the state it holds to fullfill the contract can be stated as immutable which is an essential attribute of a contract, once it is signed and verified it can never be changed unless both parties agree to it.

In its essence, smart contracts can be defined as a collection of functions and state deployed on the blockchain network using signed transactions. Smart contracts usually oblige to three properties:

- **Availability** - They execute on all nodes of the blockchain and its state has to be the same everywhere;
- **Deterministic** - Multiple executions of the same functionality has to yield exactly the same result;
- **Internal operation** - They cannot fetch data from external web services, they can only function in the blockchain network.

While there are many implementations of smart contracts over a blockchain network nowadays, the most famous ones can be divided into two categories: i) public blockchains, in which *Ethereum* stands out; And ii) private/consortium blockchains or enterprise-grade blockchains, in which three stand out, *Hyperledger Fabric*, *Corda* and *Quorum*. Besides the type

of chain there are also significant differences in the way they operate, as will be described next.

Ethereum

Smart contracts in *Ethereum* are written using the Solidity or Vyper *Programming Language (PL)*, both developed specifically for this purpose [27]. Solidity is an object oriented, high-level PL and Vyper is a contract oriented and security focused PL. The language code written in any of the PLs has to be afterwards compiled to low-level machine instructions called *opcodes* that can be executed by the *Ethereum Virtual Machine (EVM)*. The purpose of the EVM is to create an abstraction between the code that will execute and the machine that will execute it, fostering program portability [28].

The way *Ethereum* works is pretty much straightforward if one knows how a typical blockchain works, since Ethereum works the same way, the only difference is that the asset being tracked by the ledger's state can be anything, whether it be cryptocurrency or information. By tracking and registering every change made in the state of the ledger, *Ethereum* can grant data immutability and traceability.

Hyperledger Fabric

In *Hyperledger Fabric*, the approach taken consists of two separate terms, defined in [29]:

- **Smart contract** - Defines the transaction logic that controlls the objects' lifecycle;
- **Chaincode** - The packaged transaction logic in code that is deployed to the blockchain network.

The following paragraph defines the relation between a smart contract and the chaincode, as said in [29]: *"A smart contract is defined within a chaincode. Multiple smart contracts can be defined within the same chaincode. When a chaincode is deployed, all smart contracts within it are made available to applications."*

Smart contract implementation in Hyperledger Fabric can be done using the *Java* runtime or the *Node.js* runtime, which means that Java, Javascript, Typescript or other PL that uses one of those runtimes can be used to develop a Hyperledger Fabric smart contract.

As it can be seen, the differences between *Hyperledger Fabric* and *Ethereum* start right away in smart contract programming, since in the first case they can be programmed with already existent and known languages and in the second case, we have a PL built specifically for that purpose. Those differences then continue to the actual way how both networks function, which seems reasonable since it is known that one is public and used by anyone and the other one is private and should only be used by the authorized entities. *Hyperledger Fabric* has an extremely modular architecture, which allows entities to replace certain components

if they already have one built, but more on this later. Generally speaking, *Hyperledger fabric's* architecture is comprised by the following essential components:

- ***Certificate Authorities (CAs)***, responsible for assigning key pairs to authorized peers, so that they issue transactions to the network;
- ***Membership Service Provider (MSP)***, responsible for knowing the permissioned identities. For an easy understanding, we can relate CAs to MSPs. While a *Certificate Authority (CA)* issues certificates for the permissioned identities, the MSP keeps the list of authorized identities;
- **Channel**. A channel can be also viewed as an actual singular network, since every information that is put in this channel can only be viewed in this channel and not by others;
- **Organization**. An entity that takes part of a business flow. An organization can correspond to an actual real entity or to a set of entities, it depends on the trust zones established by the business partners;
- **Peer**. An actual node that belongs to an organization;
- **Ordering service**. A special set of nodes that have the role of guaranteeing transaction uniqueness and validity. It can be seen as the service responsible for achieving consensus on the state of the ledger.

From the component definition above, it is clear that *Hyperledger Fabric's* architecture is really flexible and it can adapt to a whole plethora of use cases. Furthermore, if each organization wishes to provide a CA wich they already have they can do so, there is no need to use one provided by *Hyperledger*. In this architecture, smart contracts are executed in every peer and the state of the ledger is kept unique and valid across all organizations in a channel.

Corda

Corda [30] is the most recent framework to come out and the support for its development came mostly from the financial sector, which can clearly be noted due to its structure. *Corda* does not keep track of ledger across all the blockchain as *Ethereum* neither across an entire channel as *Hyperledger Fabric*, instead it implements the concept of direct transaction communication, i.e., when a transaction is issued, the information relevant to it is only kept safe and known by the entities that participated in it. As an example, lets think about three entities: A, B and C. Then two transactions are issued:

- A issues transaction with information X to C;

- B issues transaction with information Y to C.

After the transactions are issued, A will hold and know X, B will hold and know Y and, finally C will hold and know X and Y. Of course this could bring problems since many times an entity will have to know if a given organization can issue a certain transaction despite not having all information on all the transactions that the organization has ever issued. *Corda's* concept of a *Notary* fixes that problem as it will be explained next. *Corda's* architecture can generally be defined by four components:

- **State**. Refers to the actual state object, i.e., the state of the ledger. The view of the ledger may and should be different for each node;
- **Flow**. A set of protocols that define the way an entity directly communicates transaction information to another entity;
- **Node**. An actual node in the network;
- **Notary**. A special node, responsible for guaranteeing transaction validity and uniqueness.

It is clear that the architecture of *Corda* is specially tailored for environments where you may have multiple entities, but there is a need of maintaining secrecy between them. An example would be a financial use case, in which a bank could transact with many organizations whilst keeping private the information traded with each one.

Quorum

Quorum is an Ethereum-based solution that combines the innovation brought by Ethereum with special requirements that exist to cater enterprise needs [31]. The two most decisive requirements that Quorum fulfills that Ethereum doesn't, which makes it suitable for enterprise solutions are: i) Private transactions; And ii) permissioned access to the blockchain network. Quorum is essentially built over Ethereum, which means that a Quorum node is, in many ways, really similar to an Ethereum node, a decision made on purpose so that Quorum nodes could be compatible with future release versions of Ethereum. So, in order to fulfill those requirements Quorum introduces another entity, the *Privacy Manager*, comprised by the Transaction Manager and the Enclave [32].

The transaction manager is responsible for handling private transaction data, i.e., to control who can access a certain piece of data. The Enclave is used for cryptographic functionality, namely transaction authenticity, participant authentication and historical data preservation.

In order to use these functionalities as part of the transactions performed by the nodes in a network, Quorum introduces an attribute that can be attached to Ethereum transactions called *privateFor*, which allows the indication of a list of participants which are allowed to see the contents of that transaction. In the global ledger, Quorum substitutes the actual data

by its hash, which means that it can still grant the data's integrity while not allowing other nodes that have not participated in a transaction to see its contents. The contents can only be seen by parties that are authorized to it, whether it be the node that issued the transaction or the ones that were stated in the *privateFor* attribute.

Quorum's main selling point can be stated as the ability to provide the simplicity and immense tooling of working in the Ethereum ecosystem while enriching it with enhancements that are able to meet enterprise needs.

To summarize, a smart contract provides functionality and state that entities can use, resting assured that the set of rules of the contract are being applied. Every time the state of the smart contract changes (data insertion or modification for example) new transactions are registered in the blockchain. It is a vital point to not confuse transaction immutability in the blockchain with the data/state the smart contract holds, because this data can be altered if appropriate functions are made available by the smart contract, but the fact that that alteration was executed can never be changed.

2.3 Blockchain & IoT integration

Since its conception, IoT has had definitive limitations on some areas, especially when it comes to data reliability or processing. It is known that these IoT devices are ideal for digitizing the physical world, due to the fact that they're oriented to data transmissions even in hard to reach/see locations, for e.g. inside weighbridges. These inherent characteristics, despite making these devices ideal for sensing and capturing data, turn them into resource-restricted and power-restricted devices, that cannot process and analyze data to extract relevant information from it. That way, in the following years since it first appeared, disruptive technologies have been used to overcome its limitations, like for example using cloud computing as a technology that allows data storing and processing using big data techniques [33].

With the appearance of blockchain technology in [16], new ways of securely sharing and storing the information sensed by IoT devices have been studied, and the integration of these two technologies is being viewed as possibly a major step forward in mitigating security & privacy challenges in IoT applications. While the improvements of the integration of blockchain technology into IoT applications are mostly related with data security & privacy, the end result can also benefit application efficiency and scalability. Some of these improvements can be immediately stood out as described below [33, 34]:

- **Decentralization**. By applying blockchain technology to an IoT application, we are inherently decentralizing data, since blockchain applications are decentralized by convention (one of the reasons blockchain guarantees immutability);

- **Scalability**. The more devices are part of the network, the better for the blockchain, it makes it safer and, of course, additional devices provide more sensing and computer power, permitting a scale-up of the application;
- **Identity**. Device identification is inherent, since we have to map an account with a device;
- **Security**. The immutability of the data that is sensed can be granted.

The numerous advantages that can be obtained also depend on the way the integration is actually implemented, since not all implementations are suitable for all applications. In [33] the authors provide an interesting view on a three-way division of the types of implementations in terms of communication in the underlying IoT infrastructure:

- **IoT-IoT**. An approach where all the interactions between IoT devices are made only at infrastructure levels, through discovery and routing protocols. Only the data is sent for storage in the Blockchain;
- **IoT-Blockchain**. An approach where all interactions between devices and all data sensed go through the blockchain, creating an immutable record of all interactions and all data;
- **Hybrid**. An approach where part of the interactions and part of the data are stored in the Blockchain. This implementation is best used when leveraging multiple technologies like cloud or edge computing with IoT and Blockchain.

The type of implementation used in the IoT infrastructure depends on the characteristics of the application/platform being built. For example, one might disconsider an IoT-Blockchain approach if the application produces huge loads of traffic, which would mean recording a lot of interactions and a lot of data. For those cases possibly, the IoT-IoT or the Hybrid approach is usually a better solution. It all comes down to the use case that is being solved.

2.3.1 *Use Cases*

In [34], the authors explore solutions that leverage an IoT infrastructure with Blockchain technology to increase transparency in supply chain management. The role of IoT in the supply chain management is clear, to sense and deliver to the digital world the data that is meaningful to treat at that given stage. On this specific case, IoT can be used in three stages of the supply chain: i) warehouse; ii) production; And iii) transportation. In warehouses, sensing devices can be used to manage inventory, basically keeping track of what has already abandoned the warehouse or what is still there. In production, sensing devices can be used in many ways, one of the most important being, to gather intelligence data on the production

process to analyze if it is efficient, if the machines being used are in good conditions (e.g. predictive and prescriptive maintenance), among other parameters. In transportation, the use case for IoT is clear, to always know where a given product is at (real-time shipment tracking). Despite the clear improvements IoT can provide to warehouse, production and transportation operations, it can also cause harm if not properly controlled, due to the fact that security & privacy concerns are known and growing for this type of technology. The risk of an attacker accessing or corrupting the data that companies in the supply chain process gather is high, which can compromise the entities working with that data. Some of the most prominent concerns relate to access control and trust in those IoT environments. Blockchain technology can be introduced at this stage to provide trust in the data that is being generated by the devices, due to its inherent decentralized and linked nature which provides data immutability. Due to the probably high data flow in these types of processes, sometimes, solutions are implemented that use the concept of a sidechain , which is usually a private chain, in which data can be stored using smart contracts for example, and then add references to the main chain and, if that reference is a strong enough link to the data stored in the sidechain, the immutability property is applied as well. So, it is clear that Blockchain technology can provide supply chain processes with the trust and immutability it needs for the implementation of devices and applications that generally improve the management part of the supply chain.

Although supply chain management is one of the best known and most studied use cases for IoT and Blockchain integration, there are a lot more use cases like, for example, the application of Blockchain technology into smart manufacturing.

In [35], the authors present a solution for a platform that can adapt to different circumstances, including smart manufacturing, which is one of the most thoroughly researched areas nowadays, the need for better and more efficient manufacturing processes. They present enhancements to current cloud-only platforms by introducing decentralization and peer-to-peer networking concepts. The cloud bridge would still exist, but transactions and data would be kept, in a decentralized manner, in smart contracts, which could grant immutability to the data sensed in manufacturing processes.

2.4 SUMMARY

This chapter covered some essential and background topics of the subjects related to the problem that was proposed. From IoT, to blockchain, smart contracts and those technologies' integration, a wide range of concepts, as well as open research challenges and most used current implementations were mentioned and studied.

The chapter begun by exploring IoT technology, including: i) the problems and challenges that are currently being faced; ii) the applications and huge use-cases of it; iii) the specific

parameters and conditions of IoT in the industry; And, finally iv) the communication protocols that can be used in these kind of environments, with a big focus in the ones that provide secure and reliable transmission of the data.

Afterwards, blockchain technology was thoroughly described, explaining essentially its origins, its applications, the properties it upholds when correctly used in a system and, of course, how the technology itself works within. Additionally, comparisons were made between different types of blockchain systems in order to provide some background for when the choice of platform has to be made.

The chapter ended with a small topic on possible use cases of the integration between both technologies, as well as typical ways of integrating them, ranging from a complete interconnection between them to a complete separation of concerns of both technologies.

The set of concepts and technologies studied will serve as the foundation for the next section, where a solution is actually proposed that uses part of the technologies previously mentioned.

3

PROBLEM, CHALLENGES AND PROPOSED SOLUTION

This section opens with a discussion on the problems and challenges that are inherent in a solution to the problem in study. After this discussion, a solution is proposed by illustrating a system architecture, explaining its components and how that architecture can successfully solve the problem in study.

To solve the main goals regarding this problem, it is clear that the solution will, at the very least, comprise IoT and blockchain technologies, which by itself raises some concerns as studied in Chapter 2: State of the Art. Despite the evident need to implement IoT mechanisms, one must be aware of the problems and the challenges that the implementation itself faces, such as security & privacy and power concerns, which in this case, are the main possible sources of issues. Security & privacy concerns have been immensely discussed both in this dissertation, as well as in literature, due to the fact that the definition and implementation of secure communication in constrained devices poses itself as a difficult task due to two main factors:

- **Low resources**. Generic IoT devices are made to possess sensing capabilities only, so their resources are very limited, which hardens the task of applying techniques like digital signatures or message encryption;
- **Power consumption**. Even if cryptographic techniques are applied, it is more than likely that they will consume a lot of the device's power, shortening its lifespan.

In IoT environments, it is a common practice to have a more powerful device that can harvest and aggregate all the data from the sensors and send it to analysis and storing location. In that device, the application of current and standard techniques and protocols is pretty much straightforward, but the communication of the sensing devices to that aggregator still has to be taken care of.

The main challenge advent from the protocol definition that must be explicit is, in fact, to coordinate a secure and fault-tolerant communication between constrained devices to the aggregator.

Besides the IoT component of the project, it is also pretty clear that blockchain technology and smart contracts will also play an important role in the solution to the presented

problem, by allowing to implement a compliance process, in which metrological data sent by the stations have to comply with specified and standard rules and thresholds. Using current smart contract platforms facilitates the process of providing a solution, since they already have implemented tools and mechanisms to ease the process of making available an application in the format of a smart contract, but despite that big advantage, there are also some shortcomings. The utilization of a public network for an enterprise solution presents two main disadvantages:

1. **Payed transactions**. The fact that for each transaction a small sum has to be payed to miners to ensure that they are rewarded for mining the transaction and reaching consensus presents itself as an obstacle in corporate environments due to the possibly high throughput of transactions which would oblige the company to have their own miners to keep gaining coin to spend in the network;

2. **Performance**. The current mechanisms to reach consensus in public networks such as Ethereum make it a poor choice performance-wise, i.e., the throughput of transactions that it can withstand is really low (e.g. In Ethereum the limit is 15 transactions per second) which makes these networks not suitable, although new algorithms and protocols for reaching consensus and increasing the system's performance are continuously being addressed such as the PoS algorithm.

In another point of view, private or consortium blockchain networks, despite complying with most requirements such as privacy, access permissions and performance present a different set of obstacles, mainly related to implementation and maintenace since they comprise a lot more configuration and network maintenance (e.g. maintaining access permissions for organization) than their public counterparts.

3.1 Proposed Approach - Solution

The solution proposed here is a two folded approach since we have two different contexts of implementation:

- In the physical context (weighing stations), a protocol definition for secure communication between the devices operating in the stations and the cloud must be established;

- In the digital context, a cloud platform capable of receiving the data from the stations, and apply a compliance process using blockchain technology must be implemented.

3.1.1 *Communication System*

The protocol definition for secure communication between devices, at this point, is being addressed as a two-part communication:

1. From the load cells to the aggregator (Smart Box);
2. From the aggregator to the cloud system.

Figure 5 illustrates this two-part communication, where we can see an example of a weighing station, which holds two weighbridges and each weighbridge holds 8 load cells, designated by LCi where i goes from 1 to 8.

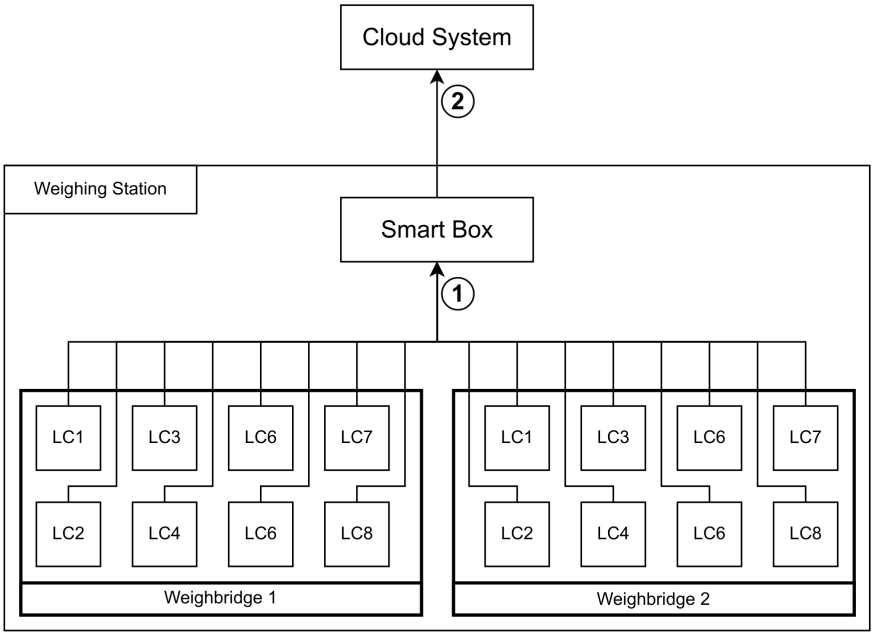


Figure 5: Proposed communication system

This separation of concerns between two possibly different communication environments, mainly in respect to the devices' communication capabilities, enables the establishment of a complete protocol definition that can rely on one or more different communication protocols, which can prove to be more advantageous, since a specific protocol can be used for a specific environment.

Despite the fact that some clear advantages can be seen in the communication diagram in figure 5, there is one clear security issue to denote in the Smart Box, which is the fact that it is subject to a man in the middle attack [36], since it is the bridge between the data generated in the load cells and their destination, the cloud system. It is not an easy security issue to explore due to the actual composition of the weighbridges in the stations, since the

Smart Box's internals are actually pretty isolated but still, this will obviously require some knowledge from the part of the cloud system as to what Smart Boxes are active and able to communicate with it.

In regards to the cloud system, it is clear that the application being built needs to be established over smart contracts, and that in some way, it has to expose APIs that allow a simple communication with the data and the operations over it. Of course that, when defining APIs, the protection of them also have to be taken into account, thus an authentication & authorization system should be built to ensure that the data API is used with the right permissions. Besides the evident use case of authentication & authorization, when a certain client communicates with the API to register data, it is more than normal that some extra information will have to be extracted to enable and simplify communication with the blockchain network.

Taking these factors into account, the next section proposes and explains a system architecture that can comply with the defined requirements for the problem.

3.1.2 *Cloud System Architecture*

Figure 6 illustrates the proposed system architecture for the cloud platform part of the problem being solved. It must be kept in mind that the full solution will ultimately comprise a protocol definition in the physical context.

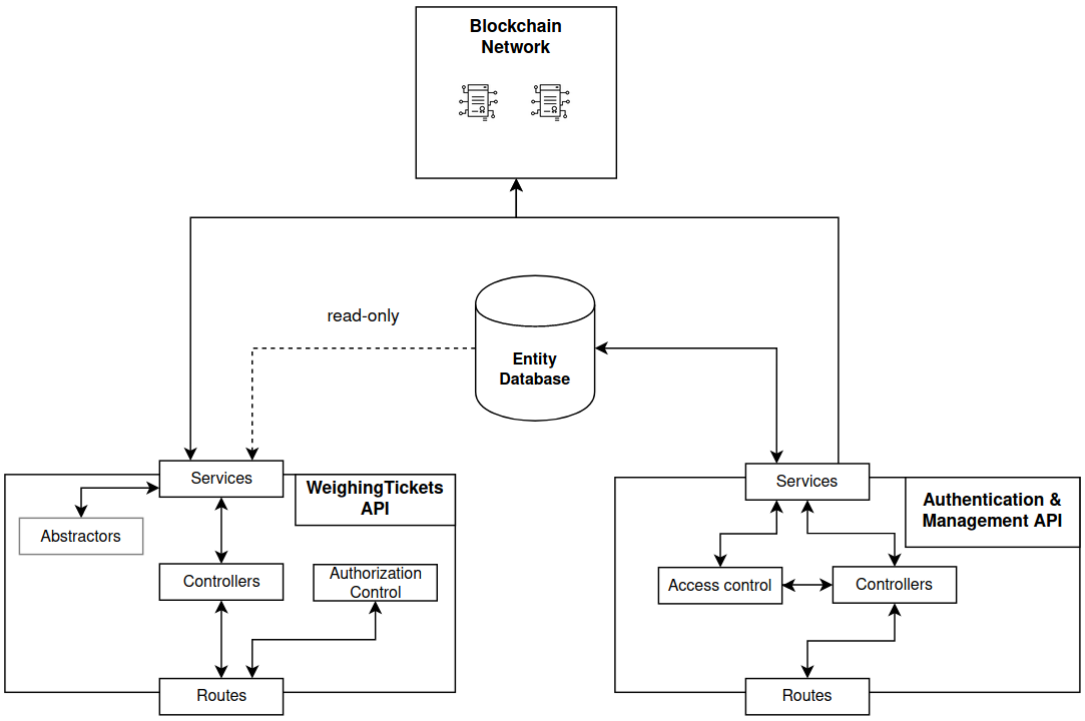


Figure 6: Proposed system architecture

In generic terms, the proposed architecture is comprised by the following components:

- **Blockchain network**. The blockchain network used to store the ledger associated with the operations performed over the smart contracts;
- **Smart contracts**. The digital contracts that define the behaviour of the entities relating to the consulting or manipulation of the weighing tickets;
- **WeighingTickets API**. An API that exposes a way of interacting with the resource being handled by the smart contract, the weighing tickets;
- **Authentication & Management API**. An API that exposes a way of creating, consulting and manipulating entities allowed in the system, as well as a way of requesting access to the most routes of either this API or the WeighingTickets API;
- **Entity Database** . A database responsible for storing entity records, including authentication information and the organization that it belongs to.

The technologies used in each of these components will be approached in section 3.1.3, but some more information can be already provided as part of the solution.

Blockchain network

The network will be comprised by organizations that are dependent and have interest in complying with data immutability and privacy in respect to the weighing tickets that are emitted by customers. Essentially, for now, this just states that it will be a multi-node network, which makes sense, since a single-node network cannot be considered a blockchain network per se. The actual organization of the network will be thoroughly studied and explained in section 4.1.

Smart contracts

The smart contract must clearly define the way that an entity can interact with a Weighing Ticket resource, i.e., defining the available methods that allow the execution of an operation against the ledger, whether it be consulting, creating or manipulating. Furthermore, it should have the ability of associating each sender with their weighing tickets.

WeighingTickets API

An API that exposes resource-related routes to communicate with the smart contract. This API exists to facilitate and abstract the communication with the smart contracts hosted in the blockchain network. This middle layer simplifies the logic and future proofs the application, since if applications are going to be built over this data in the future, it is much more easier to build them over a clearly defined API, than a smart contract, which has inherent concepts related to blockchain technology that may not be clear to everyone. To put it simply, this API has 5 main modules/components:

- **Routes**. Maps requests over the web to actual operations over resources, residing in the controllers;
- **Controllers**. Methods that implement route functionality, i.e., there should be a one to one unique association between routes and controllers;
- **Services**. Granular methods responsible for communicating with the smart contracts;
- **Abstractors**. Module that implements methods capable of abstracting certain blockchain concepts to easy-on-the-eye data, to release the user of that burden;
- **Authorization control**. Module that implements route authorization mechanisms, used by the **Routes** module.

Authentication & Management API

An API that also exposes resource-related routes to communicate with the entity database, as well as providing ways of requesting authorization to access the EntityManagement and WeighingTickets APIs. In a simple form, this API is comprised by 4 main modules/components:

- **Routes**. Maps requests over the web to actual operations over resources, residing in the controllers;
- **Controllers**. Method that implement route functionality, including entity management and access control. There has to be a one to one assocation between routes and controllers;
- **Services**. Granular methods responsible for communicating with the entity database;
- **Access control**. Granular method responsible for operations that have to do with authentication and authorization, such as generating authorization tokens or validating them.

Entity Database

This component represents a database that will simply hold entity information, including its authentication attributes for later comparison, its location, blockchain attributes when needed, cloud node being used, among others.

This definition is capable of handling the compliance process, while also providing methods and functionalities to be used by other possible modules to be integrated into this platform such as data analysis. This capability makes this initial architecture a good starting point for a solid platform in terms of integrating all the services being used by the companies of the group, since it already implements an authentication & authorization service, as well as metrological data storing and easy access to it.

3.1.3 *Technological Choice*

This section serves the purpose of clearly defining the technologies that will be used for the implementation of each component in the system architecture, as well as for the secure protocol definition used for communication, defined in 3.1.2. This way, five major technological decisions have to be made:

1. **Blockchain platform**. The platform to use for the implementation of the blockchain network and the smart contracts;
2. **WeighingTickets API**. The framework to use for the implementation of the WeighingTickets API;
3. **Authentication & Management API**. The framework to use for the implementation of the Authentication & Management API;
4. **Entity Database**. The database technology to use for storing entity information;

5. **Protocol Definition**. The protocol definition for the secure IoT communication.

For each component, the technology that was chosen is presented as well as the explanation to why that was the case.

Blockchain platform

The first step in deciding which platform to use had to obviously be whether it should be public or private and, taking into account, the enterprise-focused context of this use case it is clear that the platform to be used should provide both performance and privacy. To fulfill these requirements, a private or consortium blockchain platform has to be used, which means that the choice was now reduced to either *Hyperledger Fabric* [29], *Corda* [30] or *Quorum* [31].

In terms of the inherent features between the three aforementioned platforms, there is actually no big difference since all of them can provide: 1) performance; 2) privacy; 3) permissioned access; 4) scalability. Due to this fact, the choice relied mostly on the simplicity and the support that each platform could provide. In this particular topic *Quorum* shines, since it is mostly an enterprise-upgraded *Ethereum*, which means that all the immense tooling around the *Ethereum* ecosystem is also available for use in *Quorum*, simplifying most of the implementation from this part. Additionally, the process of establishing permissions and private transactions in *Quorum* is simpler, since it makes a clear separation on the components that govern permission in the network and the components that govern the data stored in the network's ledger itself, making the latter basically a problem to be solved just as it would with *Ethereum*.

WeighingTickets API

The development of a web API can be done in multiple frameworks, ranging from using Java with Springboot [37], Python with Flask [38], NodeJS [39], among others. The choice made for this component eyed mostly the maturity of the framework itself as well as the maturity of the integration tools with the *Ethereum*/*Quorum* ecosystem. NodeJS was chosen, since the biggest support for integration between *Quorum* and other languages/frameworks was definitely with NodeJS and Python, but while searching for support in either, using NodeJS with Quorum appears to be in fact the most common use-case. In fact, Quorum provides an extension for an essential library that is used to communicate with Ethereum/Quorum networks in a NodeJS application, that makes the integration and utilization of those additional features really simple.

Authentication & Management API

As in the previous API there are a lot of technologies to choose from, and even taking into account the purpose of this component, which is providing ways of manipulating entity resources as well as authentication and authorization mechanisms, all the previously mentioned frameworks can solve these problems in many ways. With that said, NodeJS was

chosen to promote consistency in code, since both APIs use the same language, as well as to exclude learning curves between APIs that could appear when using different frameworks for each component, which could harm the development of the solution.

Entity Database

Choosing the database technology to use has to do, mainly, with the type of data that is going to be stored and its characteristics, added of course with the performance of each technology, if applicable. In this use case and, for this database, the primary purpose is to be able to store entity objects from which we can later retrieve some information such as authentication or organization details. To do this, a relational or non-relational approach can be followed, i.e., a relational database technology can be considered that, in summary, stores information in tables and relates different tables by primary and foreign keys or a non-relational database technology can be considered which essentially stores objects in collections mostly, providing a schema-less way of interacting with the database. An example of a relational database technology is PostgreSQL and an example of a no-SQL (non-relational) database technology is MongoDB. Taking into account the simple nature of the data that is to be stored and the amazing integration that MongoDB has with the previously chosen framework (NodeJS), MongoDB was chosen as the technology to use to store entity-related data. Additionally, MongoDB also provides a rich query system that also enables us to easily query nested objects stored in its collections, which is essential and is something that is one of the best features of relational SQL databases.

Protocol Definition

As seen in section 3.1, the protocol definition for this use-case has to contemplate two distinct moments of communication: i) From the load cells in each weighbridge to the Smart Box; And ii) from the Smart Box to the cloud system. Taking into account the RESTful nature of the APIs that are going to be built, it is clear that the communication from the Smart Box to the cloud system would significantly benefit from a communication protocol that is RESTful by nature, since it would clearly simplify the establishment of the communication routes. Additionally, taking into account that the Smart Box is a device that does not fit into the constrained devices category, i.e., it is a more capable computation device that also possesses capabilities to display information on a display, the primary challenges of IoT communication like energy-awareness and low security due to constrained resources do not clearly apply here and, following that line of thought, HTTP can be perfectly applied in this communication since it is a more than mature protocol, that provides a RESTful communication and, beyond that, can be made secure simply by using it over TLS.

As for the communication between the load cells and the Smart Box, the scenario is clearly different since the load cells' electronics are really low-power, possessing only sensing and minimal computation capabilities. With that in mind, it is obvious that here a more lightweight protocol such as CoAP or MQTT would significantly improve the efficiency and

throughput of the communication of data from the load cells to the Smart Box. Additionally, according to the data provided by the companies, no more than 300 weighings are performed by day in each station, which means that performance won't be an issue, since either one of the protocols can perfectly deal with that kind of data throughput. With that in mind, the choice between the two protocols boils down to their own architecture and how they can simplify the communication system in terms of its understanding and implementation and in that area, CoAP is better suited for this use case due to the fact that it also has a RESTful architeture, which means that the communication of data from bottom (load cells) to top (cloud system) could all be done by RESTful interfaces, rendering the communication simple, understandable and transparent. Additionally, CoAP can use DTLS as the underlying transport-layer protocol to render the communication secure.

3.2 SUMMARY

This chapter covered the problems and challenges that might rise to be solved in the implementation of the proposed solution and the essential definitions of the solution to the problem presented in this dissertation, ranging from a high-level description of the architecture of the solution to be built to the actual technological choice in each of the components that comprise the solution.

Firstly, the main problems that can appear in the utilization of blockchain, IoT and its integration were presented, ranging from the properties of possible blockchain solutions to the characteristics of typical IoT environments.

Afterwards, the chapter explored the conceptual definition of a communication system taking into account the real world scenario of the problem presented in this dissertation. Immediately following that, the cloud system architecture, was explored, explaining each of its components and how they individually contribute so that together they can comply with the defined goals.

Finally, the technological choice for each of the components of the proposed solution, including the communication system and the cloud system was investigated, providing the options that were considered, the choice that was made and the reasoning behind that choice.

4

DEVELOPMENT

In this chapter, the goal is to thoroughly explain how the solution was designed and developed, starting with the main decisions that had to be done in each component of the solution as defined in Chapter 3, and ending with the specifics on how each component was implemented and how they abide by standard software development and security principles. The chapter ends with a summary of the development of this solution including: i) the software components that resulted from it; ii) what can be done with those software components when compared with the initial goals; And iii) explore and discuss some novelties or particularities that were considered and thought of in the development of those components.

As was seen before, the development of this solution can be divided into two main parts:

1. the cloud system, which includes the development of: i) the weighing tickets API; And ii) the authentication & management API;

2. the communication system, which includes the development of components able to communicate from the load cells to the smart box and from the smart box to the cloud system, in a secure manner.

The remainder of this chapter explores in Section 4.1, the decisions that were made in order to correctly develop the solution, exploring a wide range of topics such as the structure of the blockchain network, the authentication & authorization algorithms and the attempt to provide transparency and consistency when writing/reading data to/from the blockchain. In Section 4.2, the actual development of the components is discussed, exploring the existent heterogeneity of technologies and how each one suits best that particular use case, due to how the code structure can be implemented in such a way that standard software development and security principles are respected. Aside from that, the implementation also dives deeper in how some fault tolerance or configuration mechanisms are implemented and provide a safer, cleaner way of customizing each component. Finally, in this implementation section, the data models and its necessary attributes are also discussed in order to provide a complete understanding on the overall scope and purpose of the solution. Last but not least, Section 4.3 summarizes the work done in the implementation stage, while also referring the

main challenges that the implementation raised. The section ends with a discussion on why the outcomes of the development are capable of complying with the proposed goals.

4.1 DECISIONS

The development of a software solution has an inherent need of clearly deciding some particularities of it, both in conceptual and technical points of view. In this case, some conceptual decisions have to be solved first as they establish the basis of the solution and such decisions may be, for example, how the permissioned blockchain network is going to be structured in terms of the entities that take part in it. In a more technical point of view, there are some aspects of the solution that should be established prior starting the development stage, preventing the need to constantly change and refactor some components in order for them to work well with the others. Such technical decisions can be, for example: i) clearly defining the interface of the smart contract and what it will be able to do and what it delegates to the APIs; ii) clearly defining the authentication algorithm for both users and weighing stations; And iii) establish how the APIs will foster transparence and usability by abstracting complex blockchain-related particularities.

The remainder of this section approaches each critical decision that was made and explains the reasoning behind it. Essentially, the decisions that were made can be categorized into five main topics:

- The software components that need to be developed to comply with the proposed solution in Chapter 3;
- Structure of the blockchain network and how it is enforced;
- Smart contract requirements;
- How to increase transparency, usability and efficiency due to complexity brought by blockchain particularities (e.g. hexadecimal stored strings, no floating-point numbers can be stored, etc.);
- How to implement authentication & authorization with the APIs.

4.1.1 *Software Components*

In order to fulfill all the requirements raised by the proposed solution and the pre-defined objectives, there are a lot of software components to be built, and those components are essentially categorized in three areas:

- Communication system, in which software to securely communicate between the devices at the weighing stations and the cloud system have to be built;

- Cloud system, in which software to allow entity management, weighing tickets management and authentication in the APIs have to be built;
- Blockchain, where a smart contract has to be developed that manages the weighing tickets.

Communication System

The communication system defined in Chapter 3's figure 5 show us that there are two flows of communication: i) From the load cells to the smart box; And ii) from the smart box to the cloud system. In order to comply with both flows of communication, two software components must be developed, one for the load cells, that should be able to read the weight that it measured and securely transmit it to the smart box and another component for the smart box, that should be able to manage and coordinate the weighings received by the load cells and securely transmit a weighing ticket when the weighing is complete. Additionally, the implementation of these two components must comprise fault tolerance mechanisms as to guarantee that a weighing ticket is, at some point, delivered to the cloud system.

Cloud System

In order to implement the architecture defined in Subsection 3.1.2's figure 6, excluding the blockchain part which has its own specifics, the development should consider three main components:

- The weighing tickets API, which allows the management of weighing tickets and communicates directly with the blockchain network;
- The authentication & management API, which allows the management of entities that belong to the platform and, furthermore, provides the necessary mechanisms for users and stations to be able to authenticate themselves and, posteriorly, to authorize when calling an operation against any of the APIs;
- The database models that represent the entities that take part in the system, i.e., the database design.

Blockchain

For the blockchain part of the solution to work, a smart contract has to be developed, which is the software component that will actually control the management of weighing tickets and store them immutably, in the network.

4.1.2 *Blockchain Network Structure*

In order to establish the structure of the permissioned blockchain network, one has to clearly understand the entities that take part in this flow of information and what their roles are. Essentially, what can be retrieved from the information given by the companies is that:

- A **Customer** is an entity defined as a customer of any company of the Bilanciai group, which can hold multiple users and multiple stations;
- An **User** is defined as an entity that belongs to a customer and has access to that customer's data, especially, to query and manage its weighing tickets;
- A **Station** is defined as an entity that is owned by a customer and that registers the weighing tickets that are produced in it.

With the entities defined, it is clear that there is one main privacy rule that should be taken into consideration: A customer can only see its data.

Essentially, the structure of the permissioned blockchain network should be as figure 7 shows, each customer is its own organization inside the network, and each station is a node in the network, holding its own smart contract, agreed upon with the customer's administrator and the network administrator. A user simply points to the administrator node of that customer.

With the structuring of the network as shown in figure 7, two essential aspects can be granted:

- A customer's data can only be seen by the customer itself;
- All customers "work together" to grant the immutability and traceability of all the tickets in the system.

The two aforementioned points are always true, since Quorum will make sure, through its privacy manager, that only the nodes that belong to that organization can see the data, the other organizations will only see a hash of the message, which exposes no information, and since a hash is unique to a message, the integrity of the message can still be granted.

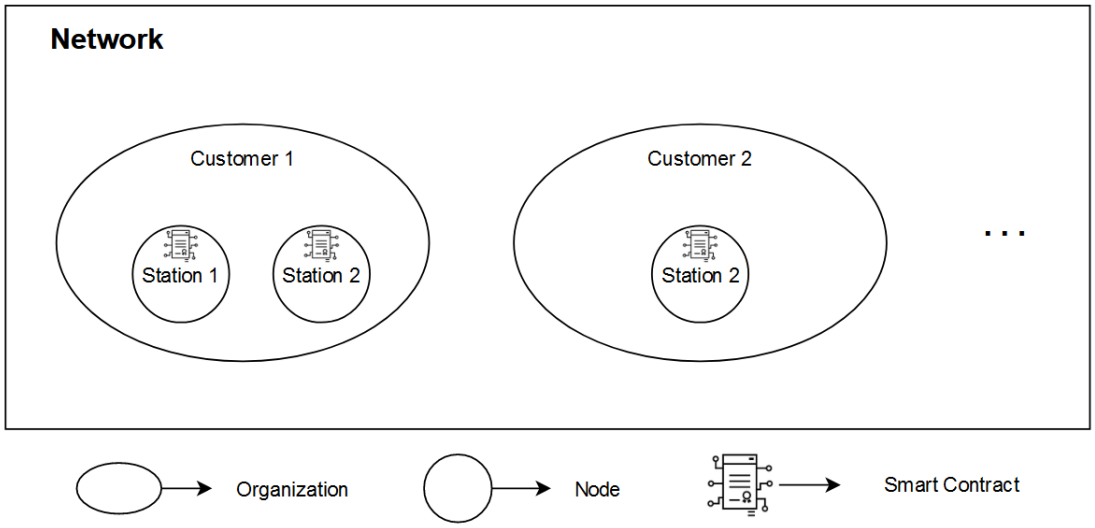


Figure 7: Structure of the permissioned blockchain network

In section Section 4.2 this topic is further explored, specifically on how the structure of the network and the properties it should uphold are enforced.

4.1.3 *Smart Contract Requirements*

Prior to the development of the smart contract and, taking into account that the technological choice for this process has already been made in Subsection 3.1.3, the requirements that it must comply must be clearly defined, including the definition of the state the smart contract will hold as well as the methods that can query or manipulate that state.

By observing the definition of the problem to be solved in this dissertation, it is implied that the state or resource that has to be stored immutably in the blockchain is a weighing ticket, which can be defined as a kind of receipt that a certain weighing took place with the exact attributes or parameters that are included in that receipt. Figure 8 shows the composition of a weighing ticket, explicitally providing the name of each parameter in the smart contract as well as its type.

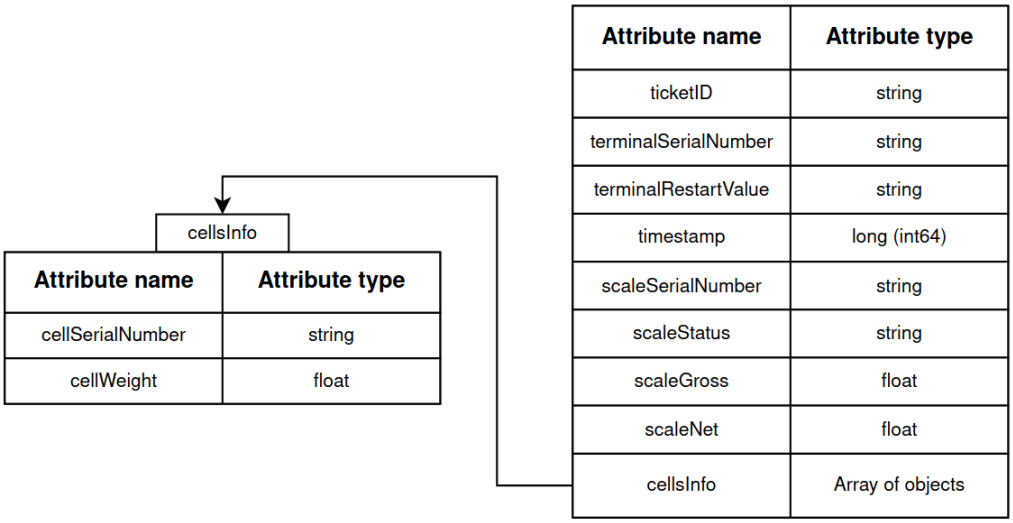


Figure 8: Composition of a weighing ticket

The weighing ticket parameters defined in figure 8 are explained as follows:

- **ticketID** A unique weighing ticket identifier, which is a string structured as *customerID_stationID_timestamp*, i.e., comprising the customer's system identifier, the station's identifier and the timestamp the ticket was issued;
- **terminalSerialNumber** The unique serial number of the terminal (smart box) that issued the ticket;
- **terminalrestartValue** Records the power state of the terminal (smart box);
- **timestamp** The UNIX timestamp at which the weighing was performed;
- **scaleSerialNumber** The unique serial number of the weighbridge that performed the weighing;
- **scaleStatus** The status of the weighbridge at the time the weighing was performed;
- **scaleGross** The gross weight value of the weighing;
- **scaleNet** The net weight value of the weighing;
- **cellsInfo** An array of objects, in which each object holds the information of a load cell, with up to a maximum of 8 objects. The information that each object comprises is as follows:
  - **cellSerialNumber** The unique serial number of the load cell;
  - **cellWeight** The amount of weight measured by this load cell.

With the parameters that are going to comprise the weighing ticket defined, which is the state that the blockchain network holds, now the methods that can manipulate or query that state also have to be defined.

Deciding the methods that the smart contract should expose requires an overview of what is the lifecycle of the weighing tickets. Essentially, the registration of new weighing tickets must be allowed in order to add a new ticket to the state of the network and, in addition to that, the query of those tickets also has to be permited in order to facilitate compliance processes that are periodically carried out. Diving deeper into the query system that the smart contract will implement and, taking into account that these tickets will be queried through the tickets API, it is clear that the continuous addition of new tickets to the network will result in increasingly larger responses from the network to the API. In order to tackle this issue, the smart contract will also provide some methods that allow immediate filtering, i.e., only returning tickets from the blockchain's state that match a certain criteria.

Finally, in this system it often happens that a certain ticket's need for storing expires, which essentially means that the aforementioned compliance processes, after a certain period of time, will no longer have the need to validate them. In this particular case, the network administrator may want to have the ability to free some state in the network and so, in order, to provide that possibility, the smart contract will also provide a removal operation which, of course, has to be protected so that it can only be accessible by a node with administrator permissions.

The weighing ticket's structure as well as the operations it has to provide in order to comply with its requirements, comprise all the necessary decisions that have to be clarified prior starting the implementation stage.

4.1.4 *Blockchain Complexity Abstraction*

The utilization of a smart contract and blockchain technology as an *immutable storage* that serves as the backbone for the system in study, has a side-effect in the way how the tickets' information is later represented. This is due to two main situations:

1. Solidity doesn't support the storage of floating point numbers, commonly known as *floats*, yet [40];

2. The libraries used in the API to interact with the smart contracts commonly extract string values as-is, which makes the outcome a hexadecimal string instead of a human-readable one.

Due to these two situations and to the fact that the structure of the weighing ticket defined in Subsection 4.1.3 holds both string and floating point properties, a decision on how to actually store the floats and represent strings in a human-readable format has to be made.

Taking this into account, the following approach can be used:

- For floating point numbers, they can be converted prior to their registration in their integer format and then converted back to float when queried. This can be done by configuring a conversion factor that is multiplied with a float with a fixed number of decimal places, turning them into integers and then, dividing the integer by the conversion factor to get the float back when the weighing ticket is queried;
- For strings, taking into account that this is a issue that is in the roadmap to be solved by the *Ethereum* community, the mechanism that handles this situation should first verify if the string is a hexadecimal string and, if it is convert it, otherwise do nothing.

The two aforementioned mechanisms, hidden in the weighing tickets API, are sufficient to abstract the complexity of what would be a situation in which the receiver had to deal with the conversion of the necessary properties.

As a final point, it is also worth noting that the development of the weighing tickets API is also an effort in terms of trying to remove some complexity in using a blockchain-based system, since it handles all operations needed to send transactions into the network and exposes the operations in a clean and simple *Representational State Transfer (REST)* API.

4.1.5 *Authentication & Authorization*

To finalize this section, a last, but an important decision has to be made, how the authentication of the entities that participate in the system will be conducted and how the requests made by those entities will be validated and authorized or not.

In terms of authentication, which is a process that requires an entity to proove to another entity its identity, this system needs to have the ability to authenticate both users and stations. Users need to authenticate themselves so that they can query for tickets and manage their customer profile and stations need to authenticate so that they can register the tickets that they emit.

While the authentication of users can be done with typical method such as a username-password mechanism, the process to authenticate a station has to be different, since it does not make sense to *hardcode* a two-word combination in the code that executes in the stations' terminal. With that said, in order to provide a dynamic and secure form of prooving the stations' identity, their authentication will be conducted using an algorithm based on public-key cryptography, which will be detailed in Subsection 4.2.1.

Assuming a user or station can authenticate themselves to the system, the authorization process also has to be thought of, i.e., the process that allows an entity to continuously issue requests and be authorized to do so. The first obvious but poor way of doing this is to authenticate in each and every request that is performed. This method is not adequate

because, despite the fact the communication will be protected by HTTPS, it is a security bad practice to constantly provide an entity's secret key. So, in order to counter this issue, the authorization process should be performed by providing a short, renewable proof of identity. This proof of identity could be used while it is still valid and then renewed by authenticating again when it is not valid. The validity of this proof of identity has to be though of in terms of security, since the longer the proof of identity is valid, the more it is prone to be caught but, on the other hand if it is too short the entity's username and secret will also be used multiple times. To address this issue, a technique that has been widely used in the last few years can be put in place called *JSON Web Tokens (JWT)* [41]. In Subsection 4.2.1 the authentication and authorization process is thoroughly described, specifically how the authentication and authorization processes combine to proove the identity of users and stations.

4.2 IMPLEMENTATION

In this section, the actual implementation of the necessary components is thoroughly explained, by specifically:

- Describing mechanisms and algorithms that are essential to the correct functioning of the system and the security of it, such as the algorithm used for authenticating and authorizing stations;
- Defining the structure of the software components and how they abide by standard good practices in software development such as modularity and separation of concerns;
- Defining how the software components can be dinamically configured, a necessary feature in such a heterogeneous system;
- Describing how each software component exposes, as a clear interface, the methods it implements.

The remainder of this section is divided in multiple sections, in which each of them describes the implementation process for one of the software components, algorithms or data structures designed and developed. More specifically, this whole section is divided as follows:

1. Subsection 4.2.1: Authentication & Authorization describes and explains the reasoning behind the authentication and authorization mechanisms and algorithms;
2. Subsection 4.2.2: Smart Contract defines the specific data structures that comprise the smart contract as well as the methods that manipulate those data structures;

3. Subsection 4.2.3: Data Models defines and describes the data models that serve as basis for the management of the entities that take part in this system;
4. Subsection 4.2.4: Cloud System APIs defines the code structure used for the APIs as well as how each one of them was implemented;
5. Subsection 4.2.5: Smart Box Communicator describes how the communicator program that operates in the smart box works, including its interactions with the load cells and the cloud system;
6. Subsection 4.2.6: Load Cell Communicator describes how the communicator program that operates in the load cells works, including its interaction with the smart box.

4.2.1 *Authentication & Authorization*

Taking into account the characteristics of this process, as defined in Subsection 4.1.5, it can be concluded that two entities take part in the authentication and authorization process, users and stations. Additionally, as stated before, at least the authentication process should be somehow different for each entity, since a user can insert its username and password at access time but, including a secret key hardcoded in the program that runs in the station's smart box is not a secure solution and thus another authentication mechanism has to be used, such as an algorithm based on public key cryptography as it was mentioned before.

The authentication process for users does not need further definition, since it is basically a typical username-password authentication mechanism, in which every user has to have an username and a password stored in the system's database and each time the authentication process is initiated, if the username and password that are given match the information in the database, the user is granted access to the system, only requiring to complete the authorization process each time the user makes a request. On the other hand, the authentication process performed by stations needs to be thoroughly described as it is not a standard nor typical mechanism for authentication. In its essence, the security of the station's authentication process is based on the properties provided by typical operations provided by public key cryptography schemes, namely message encryption and digital signatures. By using a public key cryptography scheme, it can be ensured that:

- If a message is encrypted with the public key of an entity, then that encrypted message can only be decrypted by the corresponding private key;
- If a message is digitally signed with an entity's private key, then it can always be verified if a certain message comes from that entity by verifying the signature with the corresponding public key.

With this knowledge as basis, the stations' authentication algorithm has to uphold two essential properties:

- The identifying information of the entity that is given in the authentication process has to be present in the system, i.e., the station's unique identifier has to be known by the system in order to authenticate said entity;
- The identifying information of the entity that is given in the authentication process has to correspond exactly to the entity that sends the authentication request, i.e., it has to be verifiable that the given unique station identifier in fact belongs to the entity that requests authentication.

So, in order to comply with the aforementioned security properties, an algorithm where two entities take part was developed. Essentially, a station's smart box has to perform one part of the algorithm to request authentication and the Authentication & Management API verifies that request by applying the second part of the algorithm. Algorithm 1 defines the algorithm the smart box implements in order to request for authorization tokens to the Authentication & Management API.

Algorithm 1: Construction of stations' authentication messages

Input: $numberBytes$, $apiPublicKey$ and $stationPrivateKey$
Output: {identifier; message; signature}

1 $rawID \leftarrow getStationIdentifier()$;
2 **identifier** $\leftarrow encrypt(rawID, apiPublicKey)$;
3 $randomBytes \leftarrow getRandomBytes(numberBytes)$;
4 **message** $\leftarrow encrypt(randomBytes, apiPublicKey)$;
5 **signature** $\leftarrow sign(message, stationPrivateKey)$;
6 **return** {*"identifier": identifier, "message": message, "signature": signature*} ;

As it can be inferred from the algorithm, it has essentially 6 critical steps:

1. Collect the station's system identifier from the configuration files;
2. Encrypt that identifier with the public key of the Authentication & Management API;
3. Generate *numberBytes* random bytes to be encrypted, as to provide some randomness to the authentication process;
4. Encrypt the random bytes generated with the public key of the Authentication & Management API and encode them in base64;
5. Sign the encrypted random bytes with the private key of the station and convert them into base 64;

6. Return a JSON object containing the encrypted identifier referred to as *identifier*, the base64 encrypted message, referred to as *message* and the base 64 signature, referred to as *signature*.

This algorithm is, of course, one part of the station's authentication process, specifically the part where the smart box that resides in the station builds the appropriate message to request authentication.

Algorithm 2 describes the algorithm that completes the second part of the station's authentication process, which is the verification part.

Algorithm 2: Verification of stations' authentication messages

Input: {identifier; message; signature}, apiPrivateKey
Output: {valid; decryptedIdentifier}

1 **stationID** $\leftarrow decrypt(identifier, apiPrivateKey)$;
2 $stationPublicKey \leftarrow getStationPublicKey(stationID)$;
3 **valid** $\leftarrow verify(message, signature, stationPublicKey)$;
4 **return** {*"valid": valid, "decryptedIdentifier": stationID*} ;

From the previous algorithm, it can be clearly seen that it has four critical steps:

1. Decrypt the received encrypted identifier with the Authentication & Management API's private key;
2. Collect the public key associated with the previous decrypted identifier;
3. Verify that the received signature is valid in comparison with the received message, by applying the station's public key;
4. Return a JSON object containing the validity of the operation, i.e., a simple True or False boolean, referred to as *valid* and the station's identifier, referred to as *decryptedIdentifier*.

This second algorithm completes the whole process of authenticating stations and verifying that they in fact belong to the system and are authorized to communicate with the APIs. Obviously, such an algorithm has to be clearly thought of in terms of how secure it is. As a first notice, this authentication process assumes that the provided keys are correctly generated and are protected by public key infrastructure mechanisms, this meaning that the public key must be certified by a trusted certification authority.

As a validation exercise, one can try to think like the mind of an attacker and try to "break" this authentication process. For example:

- The attacker might insert a random identifier, with a message and signature still valid since he can use the API's public key and, of course, his private key. This would not work since the verification process directly associates an identifier with the public key of that station and, if the provided identifier is not recognized, the authentication process fails;

- The attacker might get a hold of an actual station's identifier, but he would not possess the private key that is associated with the station that is recognized by that identifier and so, in the verification of the signature, the public key of that station would not match with the attacker's private key, used to sign the message, resulting in a failure of the authentication process.

So, as a conclusion to the security of the station's authentication process, in order to break it, the attacker would either have to somehow get a hold of a station's private key and its identifier, which are considered private data, or he would have to gain access to the system's database and insert a station with an identifier known to him and his public key. In the latter case, the fact is, due to the way how the weighing ticket's API and the blockchain network is strucured he would not be able to do much, since he would only be able to manipulate that fake station's contract and data. The attacker wouldn't even be able to see data from different station's of the customer it is associated.

The two previous mechanisms refer to the authentication process of an entity, but as was stated before, this authentication process should not be performed an immense number of times since it invariably requires the transport of secret credentials that belong to the entities. So, in order to continuously provide authorization to authenticated entities, a mechanism called JWT was used, which essentially allows the Authentication & Management API to, upon authentication, provide authorization tokens which contain the necessary information for the entity to use the APIs without providing secret credentials.

As a reinforcement measure, this authorization mechanism did, not only provide what is called as an *access token*, but also a *refresh token*. The *access token* is, as its name suggest, the token used by entities so that they can be authorized to perform each request. The *refresh token* is a token with a longer *Time To Live (TTL)* than the *access token*, which essentially, allows the renewal of the entities *access token* without having to perform the authentication process again.

Evidently, a mechanism, such as this one which allows continuous authorization to an entity, must be well secured to basically ensure two things: i) The authorization process is protected against stolen tokens; And ii) the tokens that are created **must** be issued by the Authentication & Management API.

The whole of the authorization process is essentially comprised by three algorithms:

- Creating the tokens to return to the entity, if the authentication process was successful;

- Verifying that a received token is valid;
- Creating a new access token, on a refresh request.

Algorithm 3 demonstrates the pseudo-code for the algorithm that actually creates and returns the authorization tokens, upon successful authentication. The first two lines of the algorithm just collect the options for the digital signature of each token, typically holding, at least, the algorithm to use and the TTL for that token. Currently, by default *access tokens* have a TTL of 15 minutes and *refresh tokens* have a TTL of 1 day. Both these values are configurable by the deployer of the Authentication & Management API. The third line of the algorithm creates the actual object that will be signed, holding the payload, i.e., the relevant information to hold about the authenticated entity so that it can later use the APIs, such as its ID which is relevant to identify when performing a request. In the creation of this payload, the algorithm shows a field called *userID*, that field can have that property or *stationID*, depending on the type of entity that has previously authenticated. The remaining of the algorithm simply calls the function given by the JWT library [42] used two times, one for the *access token* and another for the *refresh token*. This function creates the token structure and signs it with the Authentication & Management API's private key with the given options, returning a base-64 encoded token.

Algorithm 3: Creating the authorization tokens

Input: userID OR stationID, customerID, apiPrivateKey
Output: {accessToken; refreshToken }

1 $accessTokenOptions \leftarrow getAccessOptions()$;
2 $refreshTokenOptions \leftarrow getRefreshOptions()$;
3 $tokenPayload \leftarrow \{"userID" : userID, "customerID" : customerID\}$;
4 **accessToken** $\leftarrow jwtSign(tokenPayload, apiPrivateKey, accessTokenOptions)$;
5 **refreshToken** $\leftarrow jwtSign(tokenPayload, apiPrivateKey, refreshTokenOptions)$;
6 **return** $\{"accessToken": accessToken, "refreshToken": refreshToken\}$;

After the creation of the tokens, the entity that requested them must be allowed to use them by, fundamentally:

- Providing a way of verifying its *acess token* which comes embedded in the request header;
- Providing a way of refreshing its *access token*, by supplying its *refresh token*.

Algorithm 4 overviews the process of validating a token, whether that token is an *access token* or a *refresh token*. Quite simply, this algorithm simply calls the verification function of the JWT library [42] with the provided arguments and, then, if an error occurs then it returns an object stating that the token is not valid and the error that occurred, otherwise it returns an object stating the token is valid as well as the payload it holds.

Algorithm 4: Validating an access or refresh token

Input: token, apiPublicKey, tokenOptions
Output: {valid; payload}

```
1 error, payload ← jwtVerify(token, apiPublicKey, tokenOptions) ;
2 if error NOT NULL then
3 |   return {"valid": False, "payload": error.message} ;
4 else
5 |   return {"valid": True, "payload": payload } ;
6 end
```

Algorithm 5 declares the pseudo-code for the algorithm that is executed when a request to refresh a token is made. In the first two lines, the algorithm collects the options that are used for access tokens and refresh tokens. In the third line, it verifies the *refresh token* received, using Algorithm 4. Finally, if the property *valid* is *True*, then a new *access token* is created by calling the token signing function from the JWT library [42] and then it is returned. If the property *valid* is not *True*, then the algorithm returns an unauthorized response to the entity, providing the *refresh token* that was verified, as well as the error that occurred, which is present in *payload*.

Algorithm 5: Refreshing an access token

Input: refreshToken, apiPrivateKey, apiPublicKey
Output: {newAccessToken}

```
1 accessTokenOptions ← getAccessOptions() ;
2 refreshTokenOptions ← getRefreshOptions() ;
3 valid, payload ← verifyJWT(refreshToken, apiPublicKey, refreshTokenOptions) ;
4 if valid then
5 |   newAccessToken ← jwtSign(payload, apiPrivateKey, accessTokenOptions) ;
6 |   return {"token": newAccessToken} ;
7 else
8 |   provideUnauthorizedResponse(refreshToken, payload) ;
9 end
```

The description of the previous algorithms wrap up the explanation on the development of the authentication & authorization process, since all algorithms and processes that take part in it were explained. To summarize, a user authenticates with a username and password mechanism while a station authenticates with a public-key cryptography algorithm created for this effect. If the authentication is successful, they are provided with an *access token*, which they can use to authorize the requests made to the APIs, and a *refresh token*, which they can use to renew their expired *access token*. Both the tokens have a TTL and so, when

the *access token's* TTL expires, it must be renewed and, when the *refresh token's* TTL expires, the authentication process must be conducted again.

4.2.2 *Smart Contract*

In this section, the actual implementation of the smart contract is discussed, specifically, the data structures it manipulates in order to correctly hold the state of the application, as well as the methods that can manipulate it, including the reasoning for its implementation and the inputs and outputs of that method.

The first important point to describe is, without a doubt, the data structures that are present in the smart contract, due to the fact that those data structures comprise the *state* of the ledger, evolving over time and that are kept immutably. There are three essential data structures defined in the contract:

- The **Ticket** which holds all the properties of a weighing ticket as defined in Figure 8. It is also worth remembering that the smart contract does not work with floating point numbers and thus, all floats are represented as integers in it. This structure is essential for the registration of a new ticket as well as the query of existing tickets, because it allows storing tickets in a well-defined and always equal format;
- The **clientTickets** which is a mapping of string to an array of **Tickets**, where each string is a client's identifier. This structure allows a direct association between a client and the tickets that the client has emitted;
- The **idsPerClient** which is a mapping of string to an array of strings. The key is the client's identifier and the value is the array of ticket identifiers emitted by that client for a simpler access. This structure allows a direct association between a client and the ticket identifiers of the tickets the client has emitted, essential to provide a simple and low-processing way of obtaining all those identifiers.

To complement the definition of these 3 data structures, several methods were conceptualized and implemented. The development of these methods followed the requirements defined in Subsection 4.1.3. The actual definition of them is described as follows:

- **registerTicket**. A method that, as the name suggests, registers a new ticket in the ledger, based on a received set of parameters. The parameters received by this method are:
  - **clientID**. A string representing the identifier of the client that possesses the station that emitted this ticket;
  - **ticketID**. A string representing a unique ticket identifier, given by the Weighing Tickets API;

  - **terminalSerialNumber**. A string representing the smart box's serial number;
  - **terminalRestartValue**. A string representing the power state of the smart box that emitted the ticket;
  - **timestamp**. A long value representing the UNIX timestamp at which the ticket was emitted;
  - **scaleSerialNumber**. A string representing the serial number of the weighbridge where the weighing took place;
  - **scaleStatus**. A string representing the status of the scale at the time the weighing was taken;
  - **scaleGross**. A long value representing the gross weight value measured at the weighbridge;
  - **scaleNet**. A long value representing the net weight value measured at the weighbridge;
  - **cellsArray**. An array of 8 positions, where each position holds an object with two properties: i) **cellSerialNumber**. A string representing the serial number of that load cell; And ii) **cellWeight**. A long value representing the weight measured by that load cell.

  This method returns no value, it only receives these parameters, builds a **Ticket** structure, associates that **Ticket** with the identifier of the client and also the ticket's identifier with the identifier of the client, saving all the data that was built in the ledger.

- **getTicketIDS**. A method that, given a client identifier, extracts all the ticket identifiers related to that client. The parameter provided to this method is:
  - **clientID**. A string representing the identifier of the client that the requester wishes to extract the ticket identifiers from.

  This method returns an array of strings, where each string is a ticket identifier. Essentially, it just returns the value of the **idsPerClient** mapping where the key is the client's identifier.

- **getTicket**. A method that collects a specific ticket from the ledger. The parameters it receives are:
  - **clientID**. A string representing the identifier of the client that should hold the ticket the requester is looking for;
  - **ticketID**. A string representing the identifier of the ticket the requester is looking for.

  This method either returns a **Ticket** if both the identifier of the client and the ticket match or nothing otherwise.

- **getTickets**. A method that, given a client identifier, collects all the tickets associated with that client. The parameter this method receives is:
  - **clientID**. A string representing the identifier of the client from which the requester wishes to look for the tickets.

  This method returns an array of **Tickets**, which are associated with the given client.

- **getTicketsByString**. A method that filters a given client's tickets, based on an also given criteria, which has to be an equality for a string. The parameters it receives are:
  - **clientID**. A string representing the identifier of the client from which the requester wishes to filter tickets;
  - **variable**. A string representing the name of the variable to filter the tickets by. This value can be one of *terminalSerialNumber*, *terminalRestartValue*, *scaleSerialNumber* or *scaleStatus*;
  - **value**. A string representing the value of the variable by which the tickets should be filtered.

  This method returns an array of **Tickets**, that match the given string equality criteria.

- **getTicketsByInteger**. A method that filters a given client's tickets, based on an also given criteria, which has to be a comparison of integers. The parameters it receives are:
  - **clientID**. A string representing the identifier of the client from which the requester wishes to filter tickets;
  - **variable**. A string representing the name of the variable to filter the tickets by. This value can be one of *weight* or *timestamp*;
  - **mode**. A string representing the mode in which the comparison has to be made. Only accepts either *above* to search for tickets with a total weight equal or above the given weight, or *below* to search for tickets with a total weight below the given weight;
  - **integer**. A long value representing the value to compare the tickets' weight to.

  This method returns an array of **Tickets** that match the given integer comparison criteria.

- **getTicketsByInterval**. A method that filters a given client's tickets, based on a given criteria, which has to be an interval comparison of integers, i.e., finding integers between two numbers. The parameters it receives are:
  - **clientID**. A string representing the identifier of the client from which the requester wishes to filter tickets;

  - **variable**. A string representing the name of the variable to filter the tickets. This value can be one of *weight* or *timestamp*;
  - **lowerLimit**. A long value representing the lower limit in the interval search (inclusive);
  - **upperLimit**. A long value representing the upper limit in the interval search (exclusive).

  This method returns an array of **Tickets** that match the given integer interval search criteria.

- **deleteTicket**. A method that deletes a ticket from the ledger. This method is only available to the network administrator. The parameters this method receives are:
  - **clientID**. A string representing the identifier of the client where the requester wishes to look for the ticket to delete;
  - **ticketID**. A string representing the identifier of the ticket that should be deleted.

  This method returns no value. It simply removes the **Ticket** that maches the criteria from the ledger.

The previously described data structures and methods comprise all the necessary logic in the smart contract. While the implementation of those data structures is pretty straightforward since they only provide the basis for easily registering and querying tickets, the implementation of the different methods has some reasoning behind, especially the ones that implement filters. In total, the smart contract implements four filtering methods: i) by the ticket's identifier; ii) By a string value of a variable present in tickets; iii) By an integer value of a variable present in tickets; And iv) By an interval of integers of a variable present in tickets. The implementation of these four filters has one main goal, to provide a way of diminishing the traffic between the blockchain network and the weighing tickets API. This is a good measure considering that, this way, the extraction of tickets does not have to be complete, i.e., it does not have to extract all the tickets from a client in every request. In summary, at least this way, one filter can directly be applied in the smart contract to diminish the data volume that is returned from the network to the API and then, in the API, the remaining filters can be applied.

4.2.3 *Data Models*

With the definition of the smart contract in the previous section, the basis for the well functioning of the blockchain network is established and, in that note, the focus needs to turn to the implementation of the APIs that support and abstract the smart contract

application. The development of the APIs has to begin, obviously, by the foundations of what is going to be the system which, naturally, relies in the data models that support it.

So, this section aims to describe the fundamental data models that support the system of APIs. The data models that are going to exist have to, somehow, mimic the entities that are going to take part in the process the system intends to facilitate. Accordingly, prior to describing the implementation of the data models, it is a best practice to describe the process so that the entities that take part in it can be clearly highlighted.

Essentially, the process and entities can be described as follows:

- Customers buy load cells and weighing bridges from *Bilanciai* and *Cachapuz*, to operate in their stations;
- Those stations emit the weighing tickets that are the primary object of this dissertation;
- With the weighing tickets stored in the ledger, customers' have to be able to query those tickets via, for example, users that are associated with that customer.

With this description of the process, there are three entities that clearly standout; i) **Customer** which is the main entity in the system, that possesses stations and has users associated with it; ii) **Station** which is the entity that performs weighings and sends tickets. It is always associated with a customer; And iii) **User** which is an entity associated with a customer that has specific details, such as authentication credentials.

Taking into account that, as said in Subsection 3.1.3, the Weighing Tickets API and the Authentication & Management API are developed using *NodeJS* [39] and that the chosen technology for the database is *MongoDB* [43], the implementation of the data models will follow a ODM approach. An ODM is basically an abstraction that allows one to work with typical objects of the programming language, in this case of *Javascript* objects, while resting assured the changes made on that object are translated to the actual document stored in the database, which clearly facilitates and makes the development and maintenance process clear.

The definition of the ODM objects was done using a *Schema* object provided by *mongoose* [44], a library that provides an interface for working with *MongoDB* in *Javascript*. These *Schema* objects are essentially *Javascript* objects with special attributes that directly link the object's properties to the document in the database. This way, for example, when the object is saved, the associated document is also saved in the database.

Bearing this in mind and knowing, as said before, that three entities take part in the system, three ODMs were implemented, one for the customer, another for the station and finally one for the user.

Each and every one of these ODMs essentially holds three types of properties that define them:

- **Descriptive**. Properties that define the entity's real world characteristics;
- **Blockchain**. Properties that define the entity as a participant in the blockchain network;
- **System**. Properties that define the entity as a participant in the system being built and that optimize the management processes of the APIs.

Customer ODM

Table 3 illustrates and describes all the properties that define a customer in the context of this dissertation. Although most of the properties that are showed in the table are pretty much self-explanatory, there are three of them that require a bit more of insight, the blockchain-related properties.

These properties are what allow the association of what is a customer in the cloud system and what is a customer in the blockchain network. **customerAdminAddress** holds a unique network address that directly associates to that customer, which allows the association of weighing tickets to this specific entity. **customerAdminNode** identifies the node that is associated with the customer, i.e., which is configured with the customer's credentials. **customerAdminTesseraPublicKey** holds the public key of the privacy manager (Tessera) of this customer's node, which allows the establishment of private transactions to this node in the APIs.

The three aforementioned properties are what allows the APIs, after successful authentication, to perform requests to the smart contract in behalf of the customer.

	Name	Description	Data Type	Required
Descriptive	**name**	The name of the customer	String	✓
	location	The location of the customer	String	x
	description	An additional description of the customer	String	x
	companyID	A company identifier in the panorama of the Bilanciai group	String	✓
Blockchain	**customerAdmin-Address**	The customer's administrator address in the blockchain network	String	✓
	customerAdmin-Node	The identification of the customer's administrator node	String	✓
	customerAdmin-TesseraPublicKey	The public key of the customer administrator node's privacy manager	String	✓
System	**active**	flag indicating if the customer is blocked or not in the system	Boolean	✓
	deleted	A flag indicating if the customer is "deleted" from the system.	Boolean	✓
	createdAt	A timestamp indicating at which UNIX time the customer was created.	Long	✓
	updatedAt	The UNIX time of the last update on the information.	Long	✓

Table 3: Description of the customer's ODM properties

Station ODM

Table 4 describes all the properties that define a station in the context of this dissertation. As said before, most of the properties are self-explanatory and, by now, the **node**, **address** and **tesseraPublicKey** properties are also understood since they share the same purpose as they did in the customer's ODM, but this time for the stations. Despite this fact, this ODM introduces a new blockchain-related property, the **contractAddress**. This property holds a unique identifier pointing to the smart contract of this station, which holds all the state (of weighing tickets) related to it.

Additionally, there is also a new system-related property, **customer**, which essentially permits the direct association of a station and the customer it belongs to.

	Name	Description	Data Type	Required
Descriptive	**name**	The name of the station	String	✓
	latitude	The latitude of the station	String	x
	longitude	The longitude of the station	String	x
	description	An additional description about the station	String	x
Blockchain	**contractAddress**	The address of the station's contract	String	✓
	address	The station's unique blockchain address	String	✓
	node	The identification of the station's node	String	✓
	tesseraPublicKey	The public key of the station's node's privacy manager	String	✓
System	**customer**	The database identifier of the station's customer	String	✓
	active	A flag indicating if the station is blocked or not in the system	Boolean	✓
	deleted	A flag indicating if the station is "deleted" from the system.	Boolean	✓
	createdAt	A timestamp indicating at which UNIX time the station was created.	Long	✓
	updatedAt	The UNIX time of the last update on the information.	Long	✓

Table 4: Description of the station's ODM properties

User ODM

Table 5 describes the properties that identify an user in the context of this dissertation. The user's ODM is, in fact, the most dynamic ODM of the three listed because some properties only exist if some condition is met and other properties can be considered as belonging to two types.

As a first note, the blockchain-related properties of this ODM only exist if the **role** of the user is that of *admin*. If a user is a *customer*, then he uses the customer's blockchain properties and does not have ones of his own. On the other hand, if an user is an *admin*, the API enforces the existence of these properties for the administrator user to be able to connect to the blockchain network.

In a second note, although the **email** and **password** are listed as **descriptive** properties, which in fact they are, since they allow the user to uniquely identify itself, they can be also considered system-related properties, since these are the properties that allow the verification of the validity of an authentication process.

Finally, the system-related properties of an user contain three properties that are different from what has been shown so far:

- **role**. The role of this user in the cloud system. The user can be an administrator and, thus, not linked to any customer or a customer's user.
- **first_pass**. A boolean flag that indicates if it is the first password of this user. Since passwords are assigned by another user on registration, the newly created user cannot use the APIs until its password is changed.
- **password_changed**. A boolean flag that indicates if the user has changed its password since the last authentication process it performed. This allows the APIs to revoke the user's old tokens, forcing the user to authenticate again, enhancing security.

	Name	Description	Data Type	Required
Descriptive	**email**	The email that uniquely identifies the user	String	✓
	password	The hash of the user's secret password	String	✓
Blockchain	**address**	The administrator user's unique blockchain address	String	x
	node	The identification of the administrator user's node	String	x
	tesseraPublicKey	The public key of the administrator user's node's privacy manager	String	x
System	**customer**	The database identifier of the user's customer	String	✓
	role	The role of the user in the system. admin or customer	String	✓
	first_pass	A flag indicating if the user's password is still the first	Boolean	✓
	password_changed	A flag indicating if the user has changed its password since last authentication	Boolean	✓
	active	A flag indicating if the user is blocked or not in the system	Boolean	✓
	deleted	A flag indicating if the user is "deleted" from the system.	Boolean	✓
	createdAt	A timestamp indicating at which UNIX time the user was created.	Long	✓
	updatedAt	The UNIX time of the last update on the information.	Long	✓

Table 5: Description of the user's ODM properties

The three ODMs defined and described in Table 3, Table 4 and Table 5 were carefully implemented this way, since they are the fundamental basis of the well functioning of the cloud system. The way the ODMs are built, in a simple and concise manner, immensely facilitates integration with blockchain technology and the handling of entities in the cloud system.

4.2.4 *Cloud System APIs*

With the conclusion of the definition and implementation of the data models in the previous section, the basis is set for the development of the cloud system's support APIs, which handle everything from ticket querying and manipulation, to entity management and the authentication & authorization of said entities.

Although the weighing tickets API and the authentication & management API serve completely different purposes, their implementation, i.e., the code structure and the support mechanisms for the logic of each API are built with the same principles and objectives, which is why it is more advantageous to describe how the APIs are structured and designed, prior specifically describing their implementation and how they allow, through their interface, the correct application of the logic.

The development of the APIs took three essential principles into account:

1. **Separation of concerns**. To make sure that each part of the API is independent and designed such that, for example, the logic does not depend on the actual implementation of the data layer or vice-versa;

2. **High flexibility and configurability**. Ensure that the program is highly dynamic, i.e., where necessary, the behavior can be altered by simply introducing a new configuration set of parameters;

3. **Consistency**. The API design has to be consistent. Equivalent routes for different entities must perform the same functionality for a different entity.

Recognizing these principles as the basis for the implementation of the APIs, the first one to address is in fact, the **separation of concerns**, which can be granted immediately when designing the strategy for the implementation of the APIs. As said before, both of them will be implemented with roughly the same code structure, but with a different logic of course and, in order to comply with the first principle, the code structure of the APIs is overviewed in Table 6. As can be seen in this table, there are six main so-called modules in each of the APIs, where each one of them has its self-contained and independent purpose.

- **Routes** is the module that exposes the paths that can be called by external applications in a clear interface. Additionally, it is in this module, through the use of middleware

functions, that the verification of valid authorization of the requesting external application occurs, as well as the validation of that specific request's particularities, such as required parameters;

- **Controllers** is the module that implements the logic of each of the paths exposed by routes. There is an exact one-to-one mapping between **Routes** and **Controllers**, i.e., for every path defined in **Routes** there is only one method called by it in **Controllers** and, for every method in **Controllers**, there is always one and only path that calls it, defined in **Routes**;

- **Services** is the module that exposes methods to work over the data models. This module is implemented so that the **Controllers** can completely abstract the utilization of the underlying data models, without having to worry about its structure, since the **Services** will take care of that part for them. There is a one-to-one mapping between a **Service** and a **Model**, i.e., each **Model** is abstracted by a **Service** and a **Service** only abstracts a **Model**. Additionally, a single **Controller** may use multiple **Services**;

- **Models** is the module responsible for implementing the ODMs for each of the data models;

- **Helpers** is the module that provides numerous resources or utilitaries for the proper functioning of the API. Namely, it provides routing and fields validation, logging capabilities to the API, error handling at API level, utilitaries to connect the API to the database and to other applications, etc.;

- **Config** is a "providing" module, i.e., its single purpose is to allow a dynamic configuration of the application. It has the capability to use a default configuration that certainly works but, it can also read a completely new configuration object from the environment. The information provided by this module is used all around the API, with exception to the **Models** module. The semantics of the configuration object will be discussed in the appropriate section for each of the APIs.

Module	Purpose	Used By
Routes	Identification of the available operations over the API's resources	External use
Controllers	Implementation of the operations defined by routes	Routes
Services	Providing simpler and concise methods to access and manipulate data models	Controllers and Helpers
Models	Define and link API data objects to database documents	Services
Helpers	To provide route validation, logging, error handling and additional API management operations	Routes, Controllers and Services
Config	To allow a dynamic and flexible configuration for the API	Routes, Controllers, Services and Helpers

Table 6: Module structure for the implementation of the APIs.

While having thoroughly described each of the common modules that both of the APIs in the cloud system implement, it can also help to understand the relations between these modules by seeing a bigger picture of the communication between them.

Figure 9 illustrates how the different common modules of the APIs "talk to each other", where the pointing arrow means "uses". As it can be seen in this figure, the separation of concerns for the modules is always present. It is a fact that both **Config** and **Helpers** are widely used, which is expected, since they provide the necessary application-wide configurations and utilitaries & validation, respectively. Nonetheless, removing **Config** and **Helpers** from the equation, since they have such an application-wide context, it can be clearly seen that: i) **Routes** can only communicate with **Controllers**; ii) **Controllers** can only communicate with **Services** and receive requests from **Routes**; iii) **Services** can only communicate with **Models** and receive requests from **Controllers**; And iv) **Models** only receive requests from **Services**.

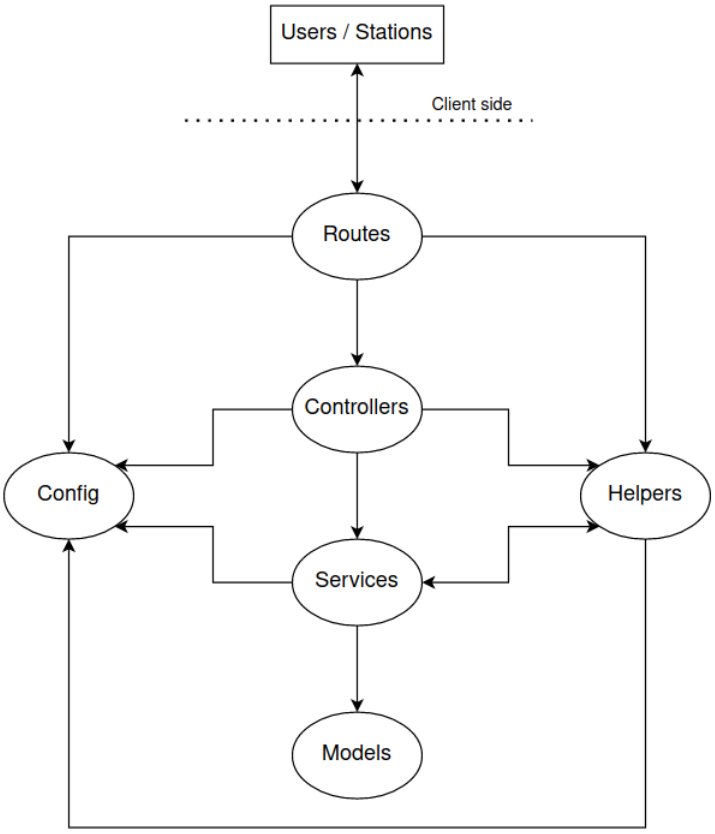


Figure 9: Relations between the APIs different modules

Essentially, the way how the APIs are implemented permits the assertion that each module is independent and is completely abstracted in terms of the data and logic layers, since: i) **Routes** never knows how data is accessed and it never needs to; ii) **Controllers** only knows that, by calling the **Services** module, it will get the necessary functionality over the data models; And iii) **Services** simply has to implement ways of handling the specific data models and provide an easy, structured way of accessing that functionality.

Finally, from Figure 9, it can also be seen that despite the fact that **Routes** and **Controllers** only "use" the **Helpers** module, the **Services** and **Helpers** module both use each other. This is mainly due to the fact that **Services** need to be able to raise application-specific errors when necessary, which is handled by the **Helpers** module and, on the other hand, the **Helpers** module needs to use the **Services** to validate the authorization of entities.

In a final note, on an aspect that also involves both the APIs, it must be mentioned that the data that is stored in the database for each of the entities that is related to blockchain properties as well as unique entity identifiers (e.g. the station's public key) is encrypted at rest, i.e., they are not stored in plaintext in the database. Taking into account the APIs' interaction with the database, i.e., the weighing tickets API only has read-only access and

the authentication & management API has read-write access, the application of this data encryption mechanism can be described as follows:

- Whenever the authentication & management API registers or updates blockchain-related or entity-identifying data in the database, the data is encrypted and authenticated prior insertion, i.e., the string inserted in the database holds both the encrypted data as well as an authentication code to verify the data was not changed at rest;
- Whenever either of the APIs need to query blockchain-related or entity-identifying data, they have to extract the string that is in the database, authenticate the encrypted data with the authentication code and only then decrypt the data and return it to whatever method needs to use it.

This mechanism was applied since it introduces some additional protection in entity properties that somehow define or identify them and, this way, only the APIs have the know-how on how to actually decrypt and see the contents of that data.

The two next subsections dive deeper into each of the APIs to explain some particularities of each one's implementation, specifically, how each of them can be dinamically configured and what functionality they expose through their routes. Additionally, due to the way how some information is presented, especially in terms of the responses given by the paths, a notation was used so that the information can be captured in a more simple and concise manner.

The notation to be used in those two subsections is defined in Table 7.

Notation	Description
K **and** Y	This means that an object is returned where the keys are K and Y, which usually also give an insight into what that value is
[K]	This means that the return value is an array of values of type K
–	This means that there is no return value in the message (although an HTTP code is always returned)

Table 7: Definition of the notation for the API responses

Weighing Tickets API

The weighing tickets API is, as mentioned before, the application responsible for managing the lifecycle of a weighing ticket, since its creation/registration to its querying. In order to be able to dynamically adjust the application by simply issuing new configuration parameters, instead of having to change them in the application code itself, the weighing tickets API possesses a mechanism that reads certain configuration parameters from the environment variables or assumes values by default. With the default parameters, the application is

always ready to run, but it is highly insecure to do so, especially due to cryptographic key pairs that should be generated pre-launch only instead of using the given ones.

The configuration of this application can be done in 6 different areas:

- **Server**. Settings and parameters that allow the user launching the application to configure aspects about the server hosting the API such as the address or the HTTPS keys;
- **JWT**. Settings related to the JWT mechanism, such as keys and the tokens' TTL;
- **Database**. Settings that permit connection to the database as a read-only user;
- **App**. Generic application settings;
- **Logging**. Parameters that tune and customize the way how the logging mechanism works in the API;
- **Smart Contract**. Generic information on the smart contract, required for when deploying contracts.

Table 8 describes each of the parameters available for each area, providing the parameter's actual name in the API, what it is and how it can be configured through environment variables.

	Name	Description	Environment Variable
Server	host	The address of the server that hosts the weighing tickets API	HOST
	port	The port where the server listens	PORT
	serverPublicKey	The public key certificate of the server for HTTPS use	IDENTITY_PUBLIC_KEY
	serverPrivateKey	The private key of the server for HTTPS use	IDENTITY_PRIVATE_KEY
	caCertificate	The public key certificate of the CA	CA_CERTIFICATE
JWT	algorithm	The algorithm to use for the JWT mechanism	JWT_ALGORITHM
	tokenTTL	The time to live of the access tokens	TOKEN_TTL
	refreshTokenTTL	The time to live of the refresh tokens	REFRESH_TOKEN_TTL
	authPublicKey	The public key to use for jwt verification	AUTH_PUBLIC_KEY
Database	name	The name of the database	DB_NAME
	host	The host where the database server is hosted	DB_HOST
	port	The port where the database server listens	DB_PORT
	username	The username for the read-only user of the database	DB_USER
	password	The password for the read-only user of the database	DB_PASS
	authenticationSource	The authentication database to use on connect	DB_AUTH_SOURCE
App	conversionFactor	The conversion factor of floats and integers to/from the blockchain network	CONVERSION_FACTOR
	validGroupKeys	The keys by which tickets can be grouped	VALID_GROUP_KEYS
	validDateFormats	The available date formats for conversion	VALID_DATE_FORMATS
	appPSK	The pre-shared key for database encryption	API_PSK
Logging	accessLogFilename	The filename to store API access logs	ACCESS_LOG_FILENAME
	accessLogPath	The directory to store access logs files	ACCESS_LOG_PATH
	accessLogFormat	The format to use when printing the logs	ACCESS_LOG_FORMAT
	apiLogFilename	The filename to store application-specific logs	API_LOG_FILENAME
	apiLogPath	The directory to store API logs	API_LOG_PATH
Smart Contract	name	The name of the smart contract	CONTRACT_NAME
	abiPath	The directory where the smart contract's interface is stored	ABI_PATH
	abiFilename	The name of the file that holds the interface of the smart contract	ABI_FILENAME

Table 8: Configurable parameters of the Weighing Tickets API

Prior talking about the REST interface that is exposed by this application, there is another subject that differs deeply between this API and the authentication & management API, which is the need to abstract certain data types when entering / leaving the blockchain network. As it was previously discussed in Subsection 4.1.4, until now, no efficient solution has been implemented in *Solidity* smart contracts and the EVM that allow the storing and manipulation of floats, which is why they have to be converted into an integer format prior being registrated in the ledger and then converted back into their float format when being queryied from the ledger. Additionally, there has to be a string abstraction mechanism also, although this particular problem is almost solved by the *Ethereum* community and so, soon, this won't be a problem anymore but, for now, it still is. Therefore, two simple algorithms are implemented to handle data abstraction, one to be run prior inserting data into the ledger and one to be run after querying data from the ledger.

Algorithm 6 illustrates the algorithm used to abstract a weighing ticket prior its registration in the ledger. While observing the algorithm, it can clearly be seen that it is a simple one: i) Extracts the conversion factor assigned in the app configurations; And ii) for each known float variable of the weighing ticket, convert it to an int.

Algorithm 6: Ticket's float to integer conversion prior ledger insertion

Input: JSONTicket, config
Output: ledgerTicket

$conversionFactor \leftarrow config.app.conversionFactor$;
$ledgerTicket \leftarrow JSONTicket$;
foreach $var \in Q \in ledgerTicket$ **do**
    $ledgerTicket.var \leftarrow int(round(ledgerTicket.var * conversionFactor)/conversionFactor)$;
end
return *ledgerTicket*;

Algorithm 7 illustrates the algorithm used to reconvert a weighing ticket back to *Javascript Object Notation (JSON)* format, after it has been collected from the ledger. This one is also a simple algorithm: i) Extract the conversion factor from the app configurations; ii) For each known float variable of the weighing ticket, convert it from int to its original float value; And, finally iii) for each string variable, if its format is that of an hex string, convert it to a regular (e.g. utf8) string, else do nothing.

Algorithm 7: Ticket's integer to float and hex to string conversion on ledger query

Input: ledgerTicket, config
Output: JSONticket

```
conversionFactor ← config.app.conversionFactor ;
JSONTicket ← ledgerTicket ;
foreach var ∈ Q ∈ JSONTicket do
    JSONTicket.var ← JSONTicket/conversionFactor ;
end
foreach stringVar ∈ JSONTicket do
    if isHexString(JSONTicket.stringVar) then
        JSONTicket.stringVar ← hexToRegularString(JSONTicket.stringVar) ;
    end
end
return JSONTicket
```

With the description of the tickets' abstraction process that runs on the API, there now remains only some definitions to be made about this application, namely about its interface, i.e., which paths it exposes to manage the tickets resource, what are each paths' inputs and outputs and, of course, the paths' functionality, which are all the necessary attributes to completely understand the implementation of each path.

Table 9 shows and describes all the paths that are implemented by specifying its method, how it can be accessed, what type of parameters it receives and what response body, if any, is given. The responses for each path are given in the notation defined in Table 7. In addition to the information present in Table 9, it is also worth noting that all the paths defined for this API are permissioned, i.e., only an entity that possesses a valid access token is allowed to successfully call one of the paths. Additionally, each path is only executed in the context of a customer (whether the authenticated entity is an user or a station) and so, a customer is not able, in any case, to query tickets from other customers.

The implemented mechanism of smart contract per station ensures that even at the blockchain network level, a customer cannot access data from another customer. In fact, not even stations from the same customer can do so, only the customer administrator node can see all its customer data.

Weighing Tickets API Overview							
- / -	Method	Path	Description	Parameters			Response
				Query	Path	Body	
Tickets	POST	*/tickets*	Register a new ticket in the ledger	x	x	✓	*ticketID* and *transactionHash*
	GET	*/tickets*	Query all the tickets from customer	✓	x	x	*[Ticket]*
	GET	*/tickets/:ticket_id*	Query for a specific ticket	x	✓	x	*Ticket*
	DELETE	*/tickets/:ticket_id*	Delete a ticket from the system	x	✓	x	–

Table 9: Overview of the functionality exposed by the Weighing Tickets API

In order to better describe each of the paths, specifically the parameters that they receive as input, Table 10 was built. This table shows, for each of the paths defined in Table 9, the parameters that that path receives as input including the name, the data type of the parameter and if it is required or not. It must also be kept in mind that the type of parameter can be looked up in Table 9 by simply looking up the specific method and checking which type of parameter it receives (query, *Uniform Resource Locator (URL)* path or body). As a way of providing some more insight into each of the possible requests, some additional information is presented as follows:

- *POST /tickets*. This path can only be called by an authenticated station. The parameters it receives are exactly the ones that can be found in the weighing ticket composition, with the same meaning and the same data type;
- *GET /tickets*. This path can only be called by an authenticated customer. It may receive a variable number of parameters from 0 to the maximum number since it basically implements filters over the tickets. Although many of these parameters are self-explainable since they are also associated with the parameter that resides in the weighing ticket composition, there are some of them which deserve further explanation:
  - **count** is a boolean. If true, then the API will return, alongside the array of weighing tickets, a count of how many tickets were returned;
  - **stations** is a coumpound string, i.e., a string that can hold multiple comma-separated values. Each value is a station identifier;
  - **group_by** holds the string that states by which variable the tickets shall be grouped. If it is not present, an array of weighing tickets is returned. If it is present, an object is returned, where the keys are the distinct values of the variable stated in **group_by**;

  - **date_type** is a string that allows the customization of how dates are returned in the tickets, either in integer format (UNIX timestamp) or full date formats.

- *GET /tickets/:ticket_id*. This path can only be called by an authenticated administrator or customer. It only needs the ticket_id parameter written in the URL path;

- *DELETE /tickets/:ticket_id*. This path can only be called by an authenticated administrator. It only needs the ticket_id parameter written in the URL path.

Weighing Tickets API Parameter Description					
- / -	Method	Path	Parameters		
			Name	Data Type	Required
Tickets	POST	*/tickets*	scaleSerialNumber	string	✓
			timestamp	long	x
			terminalSerialNumber	string	✓
			terminalRestartValue	string	✓
			scaleStatus	string	✓
			scaleGross	float	✓
			scaleNet	float	✓
			cells	array	✓
	GET	*/tickets*	count	boolean	x
			customer	string	x
			stations	string	x
			from_date	long OR date	x
			until_date	long OR date	x
			from_weight	float	x
			until_weight	float	x
			scale_serial_number	string	x
			terminal_serial_number	string	x
			scale_status	string	x
			terminal_restart_value	string	x
			group_by	string	x
			date_type	string	x
	GET	*/tickets/:ticket_id*	ticket_id	string	✓
	DELETE	*/tickets/:ticket_id*	ticket_id	string	✓

Table 10: Parameter description for the *tickets* route

The weighing tickets API overview and parameter description concludes the description and explanation of the implementation process of this API since the relevant points to understand its implementation have been covered: i) Its code structure; ii) The API's specific configuration mechanism; iii) The way how it abstracts some complexity in handling data

introduced by the blockchain platform; And finally, iv) Its interface, i.e., which paths it exposes for utilization, as well as the inputs those paths receive and the responses they give.

Authentication & Management API

The Authentication & Management API is the application responsible for managing the entities that participate in this system, as well as providing means of authentication and authorization so that those entities can, in fact, use the APIs.

Similarly to the weighing tickets API, this application can also be dynamically configured through environment variables. These configurations are mostly separated in five different areas:

- **Server**. Settings and parameters that allow the user launching the application to configure aspects about the server hosting the API such as the address or the HTTPS keys;
- **JWT**. Settings related to the JWT mechanism, such as the keys for access tokens, refresh tokens and their TTL;
- **Database**. Settings that permit connection to the database as a read-write user;
- **App**. Generic application settings;
- **Logging**. Parameters that tune and customize the way how the logging mechanism works in the API.

Table 11 describes each of the possible configurations for each area, providing the parameter's name in the API, its purpose and how it can be configured through environment variables.

	Name	Description	Environment Variable
Server	host	The address of the server that hosts the authentication & management API	HOST
	port	The port where the server listens	PORT
	serverPublicKey	The public key of the server for HTTPS use	IDENTITY_PUBLIC_KEY
	serverPrivateKey	The private key of the server for HTTPS use	IDENTITY_PRIVATE_KEY
	caCertificate	The public key certificate of the CA	CA_CERTIFICATE
JWT	algorithm	The algorithm to use for the JWT mechanism	JWT_ALGORITHM
	tokenTTL	The time to live of the access tokens	TOKEN_TTL
	refreshTokenTTL	The time to live of the refresh tokens	REFRESH_TOKEN_TTL
	authPublicKey	The public key to use for access token verification	AUTH_PUBLIC_KEY
	authPrivateKey	The private key to use for access token signature	AUTH_PRIVATE_KEY
	authPublicKeyRefresh	The public key to use for refresh token verification	AUTH_PUBLIC_KEY_REFRESH
	authPrivateKeyRefresh	The private key to use for refresh token signature	AUTH_PRIVATE_KEY
Database	name	The name of the database	DB_NAME
	host	The host where the database server is hosted	DB_HOST
	port	The port where the database server listens	DB_PORT
	username	The username for the user of the database	DB_USER
	password	The password for the user of the database	DB_PASS
	authenticationSource	The authentication database to use on connect	DB_AUTH_SOURCE
App	appPSK	The pre-shared key for database encryption	API_PSK
Logging	accessLogFilename	The filename to store API access logs	ACCESS_LOG_FILENAME
	accessLogPath	The directory to store access logs files	ACCESS_LOG_PATH
	accessLogFormat	The format to use when printing the logs	ACCESS_LOG_FORMAT
	apiLogFilename	The filename to store application-specific logs	API_LOG_FILENAME
	apiLogPath	The directory to store API logs	API_LOG_PATH

Table 11: Configurable parameters of the Authentication & Management API

Besides the configuration mechanism that the application possesses, there also two other significant aspects of the implementation process that are worth describing and explaining, namely:

- How this API enforces the network structure defined in Subsection 4.1.2;
- The interface exposed by this API which provides all desired functionality.

The process of enforcing the necessary blockchain network structure is deeply tied with the registration of new stations and the ability to establish private transactions. As was already mentioned this system implements a smart contract per station mechanism, which essentially means that each station establishes a contract between its associated customer's administrator and the network administrator.

Algorithm 8 illustrates the process of deploying a contract for a station that is being registered, inherently enforcing the network structure that is required. To further explain this process, the algorithm goes through the following steps:

1. Receives as input the identifier of the customer to whom this station belongs, which is known to be correct since the access token has already been verified and it receives as a route parameter the public key of the privacy manager (Tessera) of the station's node;
2. Queries for the public key of the privacy manager belonging to the network administrator;
3. Queries for the public key of the privacy manager belonging to the administrator of the customer that is associated with the station being registered;
4. Builds a list with the public keys of the privacy managers that are allowed to interact with the contract being deployed. The public key belonging to the privacy manager of the network administrator is not included, since this will be the key used for deploying the contract;
5. An instance of the smart contract is deployed by the network administrator, where only the customer's privacy manager, the station's privacy manager and the network administrator's privacy manager can see the data held by that instance of the smart contract;
6. After the contract is deployed the contract address is returned, so that that station and the users associated with that customer can later communicate with it.

Algorithm 8: Ensuring network structure on station registration

Input: customerID, stationTesseraPublicKey

Output: contractAddress

$networkAdminTesseraPublicKey \leftarrow getAdminTesseraPublicKey()$;
$customerAdminTesseraPublicKey \leftarrow getCustomerTesseraPublicKey(customerID)$;
$privateFor \leftarrow [customerAdminTesseraPublicKey, stationTesseraPublicKey]$;
$contractAddress \leftarrow deployContract(networkAdminTesseraPublicKey, privateFor)$;
return *contractAddress*

By applying this process each time a station registers, the confidentiality of the data is always maintained, since a station can only see its data and a customer can only see the data belonging to its stations.

With this type of mechanism, data privacy and confidentiality is ensured both at API level, since a station can only link or associate with its customer and a customer can only associate with its stations and users, and at the network level, since after contract deployment only the pre-determined public keys are able to access or call that contract. Furthermore, contract deployment is an operation with an immutable post-state, i.e., there is no possibility of adding additional keys to communicate with the contract later.

To finalize the description of the implementation process of the authentication & management API, its interface must be described and explained, as well as the parameters that each possible path receives and what permissions are established in order to comply with the necessary security requirements.

Table 12 presents an overview of the authentication & management API by showing the possible paths per resource that are able to manipulate or query them. The notation used in the **Response** column is the same notation presented before in Table 7. The resources this application handles are:

- **Customers**. An abstract entity that defines a customer and the users and stations that can act in its behalf;
- **Users**. Entities associated with customers or to administration that actually use the API;
- **Stations**. Entities associated with customers that use the weighing tickets API for ticket registration;
- **Blacklisted Tokens**. A management resource, implemented so that the possibility of revoking an access token or refresh token before its expiration date exists;
- **Authorization**. A set of processes to authenticate and authorize users and stations.

Authentication & Management API Overview							
- / -	Method	Path	Description	Parameters			Response
				Query	Path	Body	
Customers	POST	*/customers*	Register a new customer	x	x	✓	*customerID*
	GET	*/customers*	Query simple information for all customers	x	x	x	*[Customer]*
	GET	*/customers/:customer_id*	Query simple information on a specific customer	x	✓	x	*Customer*
	PUT	*/customers/:customer_id*	Update information on a customer	x	✓	✓	–
	DELETE	*/customers/:customer_id*	Delete a customer from the system	x	✓	x	–
Users	POST	*/users*	Register a new user	x	x	✓	*userID*
	GET	*/users*	Queries information on all users	x	x	x	*[User]*
	GET	*/users/:user_id*	Query information on a specific user	x	✓	x	*User*
	PUT	*/users/:user_id*	Update information on a specific user	x	✓	✓	–
	DELETE	*/users/:user_id*	Delete a user from the system	x	✓	x	–
Stations	POST	*/stations*	Register a new station	x	x	✓	*Station*
	GET	*/stations*	Query simple information for all stations	x	x	x	*[Station]*
	GET	*/stations/:station_id*	Query simple information for a specific station	x	✓	x	*Station*
	PUT	*/stations/:station_id*	Update information on a specific station	x	✓	✓	–
	DELETE	*/stations/:station_id*	Delete a station from the system	x	✓	x	–
Blacklisted Tokens	POST	*/blacklisted_tokens*	Add a new token to the blacklist	✓	x	✓	*tokenID*
	GET	*/blacklisted_tokens*	Query all blacklisted tokens	x	x	x	*[Token]*
	GET	*/blacklisted_tokens/:token_id*	Query for a specific blacklisted token	x	✓	x	*Token*
	DELETE	*/blacklisted_tokens*	Delete all blacklisted tokens that have expired	x	x	x	–
	DELETE	*/blacklisted_tokens/:token_id*	Delete a specific blacklisted token	x	✓	x	–
Autho-rization	POST	*/authorization*	Authenticates a user or a station	x	x	✓	*accessToken* and *refreshToken*
	PUT	*/authorization*	Refreshes an access token	x	x	✓	*accessToken*

Table 12: Overview of the functionality exposed by the Authentication & Management API

Table 12 allowed an overview of the resources the application handles, as well as the paths that can be called per each resource and what type of parameters are used in those paths, finalizing with the response that each path provides when called successfully.

In order to provide a better sense and to better explain how this API is built, the following couple of pages explore each resource individually, clarifying the actual parameters they receive, their data type and if they are required or not. As the type of parameter (query, URL path, body) for each path is already detailed in Table 12, the tables explaining each resource will not contain the type of the parameter, except when that path has more than one type of parameter (e.g. receives URL path and body parameters). In this latter case, the type of the parameter is indicated in the **Required** column after the checkmark or the parameter not required notation.

With this in mind, the detailed information for each resource can be given in the following manner:

- Table 13 describes each path available to handle the **Customers** resource.
  - *POST /customers*. Can only be called by a system administrator. All of its parameters are already defined in Table 3;
  - *GET /customers*. Can only be called by a system administrator. Needs no parameters;
  - *GET /customers/:customer_id*. Can only be called by a system administrator or a user associated with the customer identified by customer_id. Requires the customer_id parameter in the URL;
  - *PUT /customers/:customer_id*. Can only be called by a system administrator or a user associated with the customer identified by customer_id, depending on the attributes to update. Requires customer_id in the URL. Can receive multiple body parameters to update, all already defined in Table 3;
  - *DELETE /customers/:customer_id*. Can only be called by a system administrator. Requires the customer_id in the URL.

Authentication & Management API Parameter Description					
- / -	Method	Path	Parameters		
			Name	Data Type	Required
Customers	POST	*/customers*	name	string	✓
			location	string	x
			description	string	x
			address	string	✓
			node	string	✓
			tesseraPublicKey	string	✓
			companyID	string	x
			email	string	✓
			password	string	✓
	GET	*/customers*	x	x	x
	GET	*/customers/:customer_id*	customer_id	string	✓
	PUT	*/customers/:customer_id*	customer_id	string	✓(URL)
			name	string	x (BODY)
			location	string	x (BODY)
			companyID	string	x (BODY)
			description	string	x (BODY)
			block	boolean	x (BODY)
	DELETE	*/customers/:customer_id*	customer_id	string	✓

Table 13: Parameter description for the *customers* route

- Table 14 describes each path available to handle the **Users** resource.
  - *POST /users*. Can be called by any user. If the user has a role of administrator, then it can only create an administrator. If the user has a role of customer, it can only create users associated with that customer. The parameters it receives are the ones defined in Table 5 except for **adminLoginEmail** and **adminLoginPassword**. These parameters exist as an additional security measure, since for an administrator user to register another administrator user, the access token is not enough, he has to authenticate with its credentials;
  - *GET /users*. Can be called by any user. If the user has role of customer, only that customer's users are presented. If the user has role of administrator, only administrator users are presented. Does not require parameters;
  - *GET /users/:user_id*. Can be called by any user. Only successful if the calling user has the same identifier as user_id. Requires only user_id as URL parameter;
  - *PUT /users/:user_id*. Can be called by any user. Serves essentially has a way to udpate the user's password, or to block another user. The blocking functionality is only valid between users of the same customer or for system administrators;

– *DELETE /users/:user_id*. Can be called by any user. Administrator users can only be deleted by one of the kind. Customer users can be deleted by administrator users or by the other users associated with that customer. Requires only the user_id to delete as an URL parameter.

Authentication & Management API Parameter Description					
- / -	Method	Path	Parameters		
			Name	Data Type	Required
Users	POST	*/users*	email	string	✓
			password	string	✓
			address	string	x
			node	string	x
			tesseraPublicKey	string	x
			adminLoginEmail	string	x
			adminLoginPassword	string	x
	GET	*/users*	x	x	x
	GET	*/users/:user_id*	user_id	string	✓
	PUT	*/users/:user_id*	user_id	string	✓(URL)
			newPassword	string	x (BODY)
			oldPassword	string	x (BODY)
			block	boolean	x (BODY)
	DELETE	*/users/:user_id*	user_id	string	✓

Table 14: Parameter description for the *users* route

- Table 15 describes each path available to handle the **Stations** resource.
  - *POST /stations*. Can only be called by an user with a customer role. The parameters required to call this path are already described in Table 4;
  - *GET /stations*. Can only be called by an user with a customer role. No parameters are required;
  - *GET /stations/:station_id*. Can only be called by an user with a customer role, whose customer is the same customer as the station in the request. Requires station_id in the URL;
  - *PUT /stations/:station_id*. Can only be called by an user with a customer role, whose customer is the same customer as the station in the request. Any of the parameters that are required or optional have already been described in Table 4;
  - *DELETE /stations/:station_id*. Can only be called by an user with a customer role, whose customer is the same customer as the station in the request. Requires station_id in the URL.

Authentication & Management API Parameter Description					
- / -	Method	Path	Parameters		
			Name	Data Type	Required
Stations	POST	*/stations*	name	string	✓
			latitude	string	x
			longitude	string	x
			description	string	x
			address	string	✓
			node	string	✓
			tesseraPublicKey	string	✓
			publicKey	string	✓
	GET	*/stations*	x	x	x
	GET	*/stations/:station_id*	station_id	string	✓
	PUT	*/stations/:station_id*	station_id	string	✓(URL)
			name	string	x (BODY)
			latitude	string	x (BODY)
			longitude	string	x (BODY)
			description	string	x (BODY)
			address	string	x (BODY)
			node	string	x (BODY)
			block	boolean	x (BODY)
	DELETE	*/stations/:station_id*	station_id	string	✓

Table 15: Parameter description for the *stations* route

- Table 16 describes each path available to handle the **Blacklisted Tokens** resource.
  - *POST /blacklisted_tokens*. Can be called by any user. If the user has a customer role, then the user identifier or station identifier in the token has to match with the same customer. If the user has an administrator role, he can block any token. Receives only the token to blacklist in the body of the request;
  - *GET /blacklisted_tokens*. Can only be called by an administrator user. It may receive an user_id query parameter, to filter the search by user;
  - *GET /blacklisted_tokens/:token_id*. Can only be called by an administrator user. Receives the token_id as URL parameter;
  - *DELETE /blacklisted_tokens*. Can only be called by an administrator user. Clears all tokens that have already expired and thus no longer need to be blacklisted. Receives no parameters;
  - *DELETE /blacklisted_tokens/:token_id*. Can only be called by an administrator user. Receives token_id as URL parameter.

Authentication & Management API Parameter Description					
- / -	Method	Path	Parameters		
			Name	Data Type	Required
Tokens	POST	*/blacklisted_tokens*	token	string	✓
	GET	*/blacklisted_tokens*	user_id	string	x
	GET	*/blacklisted_tokens/:token_id*	token_id	string	✓
	DELETE	*/blacklisted_tokens*	x	x	x
	DELETE	*/blacklisted_tokens/:token_id*	token_id	string	✓

Table 16: Parameter description for the *blacklisted tokens* route

- Table 17 describes each path available to handle the **Authorization** resource.
  - *POST /authorization*. Can be called by anyone. If the authenticating entity is an user, email and password are required as body parameters. If the authenticating entity is a station identifier, message and signature are required as body parameters. One of these two combinations must exist in the body of the request;
  - *PUT /authorization*. Can be called by anyone. Has to receive the token as a body parameter, which is the refresh token.

Authentication & Management API Parameter Description					
- / -	Method	Path	Parameters		
			Name	Data Type	Required
Autho-rization	POST	*/authorization*	email	string	x
			password	string	x
			identifier	string	x
			message	string	x
			signature	string	x
	PUT	*/authorization*	token	string	✓

Table 17: Parameter description for the *authorization* route

As a final note to all the information presented before, it is also important to mention that the entity blocking and removal process implements a kind of cascade mechanism, i.e., if a customer is blocked or deleted, then any of its associated users and stations will also be blocked / deleted, but if a user or station individually is blocked or deleted, then the other entities associated with the customer will still be able to communicate.

This finalizes the process of describing how the authentication & management API is implemented since its common architecture had already been covered, which is similar to the weighing tickets API and, after that, in this subsection, the API's configuration mechanism, the way it enforces the network structure and, finally, the interface it exposes and what

each of the paths in that interface do were also covered, providing a simple and thorough overview of its functionalities.

4.2.5 *Smart Box Communicator*

The purpose of this section is to describe and explain how the smart box communicator is built. This program can be described as the application that runs on the smart boxes, the devices that operate on the customers' stations. Each station has only one smart box and this device controls all weighbridges present in that station.

Prior describing how the communicator is implemented it is, of course, necessary to define what will be used to implement it. For the implementation of this communicator and, taking into account that this is a device that does not fall into the restricted resources category, since its hardware is comprised by a Cortex dual-core *Central Processing Unit (CPU)* at 1GHz frequency and 512 *Megabytes (MB)* of *Random Access Memory (RAM)*, the language chosen was Golang [45]. This choice was made due to two reasons essentially:

- It is a high-performance programming language with excellent concurrency features;
- It has immense package support, including different communication modules that facilitate working with DTLS or TLS for example.

In Subsection 3.1.3: Technological Choice, the communication protocol was defined as being comprised by two different communication mechanisms: i) One that works between the load cells and the smart box; And ii) one that works between the smart box and the cloud system. Additionally, it was established that the first communication mechanism would be implemented by using CoAP as the application-layer protocol and DTLS as the transport-layer protocol and, the second communication mechanism would be implemented with HTTPS, which is essentialy HTTP as the application-layer protocol and TLS as the transport-layer protocol. Concluding this opening section, it is now clear that the smart box communicator will have to be able to communicate either via CoAP with the load cells or via HTTPS with the cloud system. Taking into account the language that is going to be used for the implementation (Golang), it is important to be aware, prior the choice, if it offers good capabilities in terms of dealing with these communication protocols. The utilization of HTTPS in Golang is quite simple since it only requires to make use of the Golang's standard packages: *net/http* [46] and *crypto/tls* [47]. For the utilization of DTLS communication in Golang there is also an excellent package that provides all necessary features, *Pion-DTLS* [48].

Figure 10 illustrates the conceptual architecture of the smart box communicator, including its distinct modules and how they communicate with each other. This architecture is comprised by six modules, each with its single purpose:

- **Configurations Manager** is the module responsible for handling and providing all necessary configurations for the other modules;
- **Ticket Builder** is the main entrance point in the program and is responsible for the ticket building process, i.e., orchestrating the other modules with the goal to successfully submit the ticket from the specified weighbridge;
- **Ticket Submitter** is the module responsible for handling submission of weighing tickets, including the application of the fault-tolerance mechanism;
- **Authentication Manager** is the module responsible for ensuring that the station has valid authorization tokens for ticket submission;
- **HTTPS Client** as the name suggests is a client for an HTTPS connection, used for communicating with both the weighing tickets and the authentication & management APIs;
- **CoAP + DTLS Client** is responsible for establishing a DTLS client capable of communicating with the load cells.

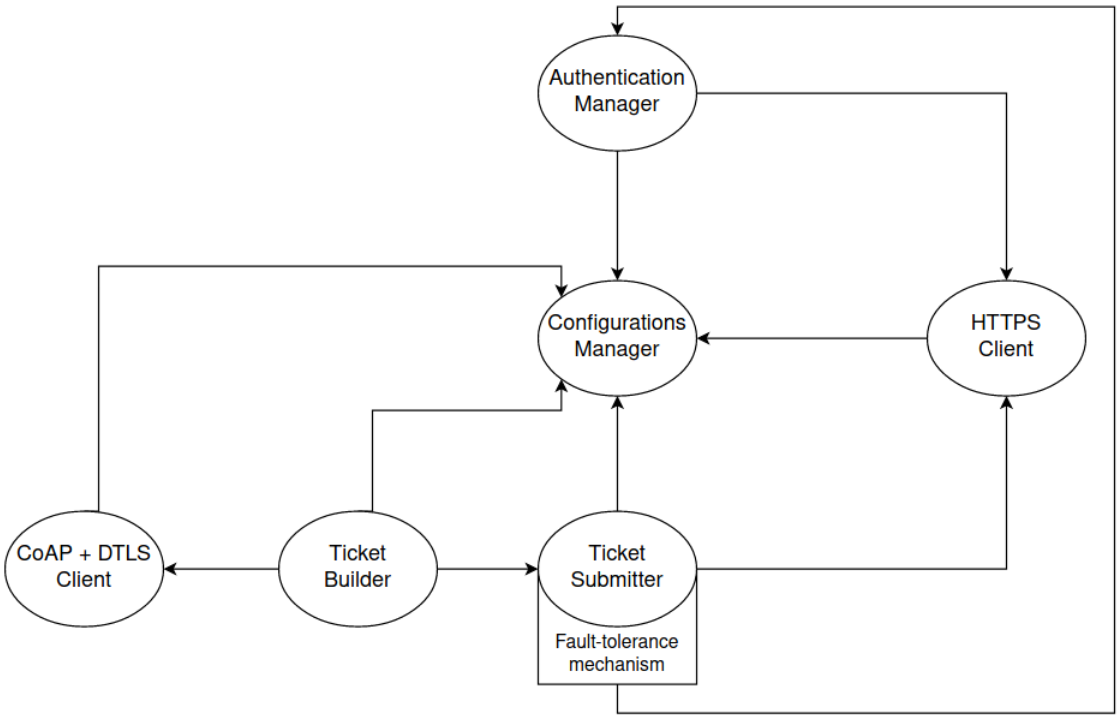


Figure 10: Smart box communicator architecture and intra-module communication

The configurations manager is a simple module that reads a configuration file, which has to be written in YAML and associates the configurations given in that file to a data

structure that is used in the program. It is through this configuration file that essential parameters to establish communication with the APIs and the load cells are provided, as well as parameters that allow the tuning of the fault-tolerance mechanism. In this file, the configurations are divided into six main sections:

- **Smart Box**. Section that holds parameters that tune where the communicator listens, like its address and port, as well as parameters that provide the required keys to communicate via TLS or DTLS;
- **CA**. Information on the root CA that is able to verify the APIs' certificates;
- **App**. Contains parameters that allow the fine tuning of the application's behaviour, including its fault-tolerance mechanism;
- **Auth_API**. Contains parameters that establish where the authentication & management API is listening, as well as the public key certificate to use in the stations authentication algorithm;
- **Weighing_Tickets_API**. Contains the required parameters for the communicator to establish communication with the weighing tickets API;
- **Station**. Configurations related to the station, including its identifier, the composition of the station, among others.

Table 18 overviews the available configurations for each aforementioned section. In each of these sections most of the parameters are pretty much self-explainatory or at the very least provide a good insight in what they are and what their purpose is, with one exception, the **weighbridges** configuration from the **Station** section, since it is a more complex structure. This parameter is, in fact, an object, which is structured as follows:

- The keys of the object are the identifiers from each of the weighbridges in that station;
- The values are an array of objects containing load cell information. These objects have the following structure:
  - **id**. The load cell's serial number;
  - **address**. The load cell's address;
  - **port**. The port where the load cell's communicator listens.

Additionaly, as it can be seen in Table 18 in the **App** section, the parameters responsible for tuning the fault-tolerance mechanism , such as the maximum pending submissions, or the maximum retries prior failing, can all be fine-tuned in order to make the mechanism more flexible or more rigid.

	Name	Description
Smart Box	host	The address of the server that hosts the smart box communicator
	port	The port where the communicator listens
	certpath	The path to the public key certificate for TLS and DTLS use
	keypath	The path to the private key for TLS and DTLS use
App	token_path	The path to store the authorization tokens
	ticket_info_path	The path to store information on submitted tickets
	pending_path	The path to store tickets pending submission
	failed_path	The path to store tickets that failed submission
	max_pending_submissions	The maximum number of submissions of pending tickets prior failing
	resubmission_interval	The number of seconds to wait before attempting resubmission
	max_retries	The maximum number of retries when a ticket fails submission
	pending_interval	The number of seconds to wait before attempting resubmission of pending tickets
Auth_API	host	The address of the server that houses the authentication & management API
	port	The port where the server listens
	pubkeypath	The path to the public key certificate for station authentication
CA	certpath	The path to the public key certificate of the CA
Tickets_API	host	The address of the server that houses the weighing tickets API
	port	The port where the API server listens for requests
Station	id	The station's database identifier
	key	The path to the private key for station authentication
	keypub	The path to the public key certificate for station authentication
	terminal	The terminal serial number of the station
	weighbridges	An object detailing the composition of the station

Table 18: Configurable parameters of the smart box communicator

Having described the configuration mechanism present in the smart box communicator, the goal is now to describe and explain how the communicator is built around that configuration

mechanism in order to be able to securely extract weights measured in the load cells and then, after building the respective ticket, emit it securely to the weighing tickets API.

The first step in deciding how the communicator is built is to think about how it is actually going to be executed in the context of this solution. By now, it is already known that weighing tickets are a kind of receipt of a weighing and so there are two properties to ensure here: i) For one weighing there is only one receipt and one receipt only matches with one weighing; And ii) the receipt must be taken when the total weight has stabilized to its actual value.

Taking into account these two properties that must be upheld, the solution that immediately comes to mind is to implement some kind of a trigger mechanism in the communicator. Essentially, when the operators or an automatic process detect that the weighing has stabilized, the smart box communicator is triggered to collect the weighing from the load cells of the respective weighbridge and then build and submit the ticket. Quite simply, each time there is a ticket to submit in the station, the operator or an automatic process launches a trigger to communicate, which is essentially the execution of the communicator program with a flag that indicates the weighbridge that is ready to submit a ticket.

As was previously mentioned, the **Ticket Builder** module is the point of entrance in the program, since prior submitting a weighing ticket, it must first be built accordingly.

Algorithm 9 defines a simplified version of the algorithm implemented to build tickets, which can be characterized in the following steps:

1. The algorithm receives as input the identifier of the weighbridge that is ready to submit a weighing ticket and the global configuration object;
2. A DTLS configuration object is built which essentially holds the public key certificate of the smart box as well as the certificate of the CA for verification purposes;
3. The information on the load cells that belong to the received weighbridge is collected;
4. A DTLS client is instanciated with the previously created **dtlsConfig**;
5. For each of the load cells that belong to the weighbridge, request the **coapDTLSClient** to query for the weighing of the load cell (A GET request to the path */weight*);
6. Adds a new weight to the ticket by providing the identifier of the load cell and the weight received;
7. Finally, it requests the **TicketSubmitter** to submit the new given ticket.

Aside from the previously described steps, there is also an important note to be made when the algorithm iterates over the load cells, performing for each one a weight request and an assignment of that weight to the ticket. This action is non-blocking, i.e., all requests are executed concurrently.

Algorithm 9: Simplified algorithm to build a weighing ticket

Input: weighbridgeID, config

```
1 dtlsConfig ← buildDTLSConfig(config) ;
2 loadCells ← getWeighbridgeLoadCells(weighbridgeID, config) ;
3 ticket ← generateNewTicket(weighbridgeID, config) ;
4 coapDTLSClient ← newDTLSClient(dtlsConfig) ;
5 foreach loadCell ∈ loadCells do
6 |   weight ← coapDTLSClient.getCellWeight(loadCell.host, loadCell.port) ;
7 |   ticket.addCellWeight(loadCell.id, weight) ;
8 end
9 TicketSubmitter.submit(ticket, config) ;
```

Before describing and explaining how the **Ticket Submitter** module works, it is important to completely define the fault-tolerance mechanism that is implemented in this communicator, since this module will implement a part of that mechanism.

The fault-tolerance mechanism implemented in this communicator is composed by two stages:

- **RETRY**, where the mechanism attempts to resubmit tickets that failed to be submitted until a maximum amount of attempts has been reached;
- **RECOVER**, where the mechanism saves tickets that were not submitted until the **RETRY** stage into a pending state. Additionally, the mechanism periodically attempts resubmission of pending tickets up until a maximum number of attempts where, finally, if not successful, the ticket is saved and marked as failed for later analysis.

The two aforementioned guidelines already provide a pretty good idea of how the fault-tolerance mechanism works, but it will be explored in much more detail in the modules where it is actually implemented, such as the **Ticket Submitter**.

The **Ticket Submitter** module's purpose is to, of course, submit tickets that are already built to the weighing tickets API. The first step in doing so is to setup all the necessary credentials to be able to perform the request. Algorithm 10 shows the process of setup that always happens prior submitting a ticket. The steps followed by this algorithm are:

- It receives as input the global configuration object;
- Build the TLS configuration object, which essentially holds the public key certificate of the CA for server verification;
- Generate a new instance of an HTTPS client with the previously built TLS configuration;

- If the authorization tokens already exist, then simply extract the access token, otherwise request the authentication manager to authenticate with the authentication & management API and then extract the access token;
- Finally, return the HTTPS client and the access token, which are the necessary credentials to perform the submission of the weighing ticket.

Algorithm 10: Setup algorithm prior submitting a weighing ticket

Input: config
Output: httpsClient, accessToken

```
1 tlsConfig ← buildTLSConfig(config) ;
2 httpsClient ← newHTTPSClient(tlsConfig) ;
3 if ∃config.Tokens then
4 |   accessToken ← config.Tokens.AccessToken ;
5 else
6 |   config.Tokens ← AuthenticationManager.authenticate(config) ;
7 |   accessToken ← config.Tokens.AccessToken ;
8 end
9 return httpsClient, accessToken
```

After this setup algorithm is executed, the **TicketSubmitter** is ready to submit a new ticket. Figure 11 shows a diagram that describes by illustration the algorithm followed on each ticket submission, with fault-tolerance included.

The point of entrance of the algorithm is always an attempt of submitting a ticket which will either return an error or a success message with the ticket information. If the response is a success message then the algorithm simply registers the ticket information (ticket identifier and transaction hash) in the configured path. In the eventuality of an error message, the algorithm first checks if this was an authorization problem, i.e., the access token was expired or invalid and, in this case, it simply calls a method to ensure a valid access token. This method first attempts to refresh the token, returning on success but, if refreshing the token also produces an authorization error, then the method simply authenticates again, receiving new tokens. After the access token is ensured, ticket submission is attempted again with a valid access token.

If the error message does not correspond to an authorization error and, instead, corresponds to any other error, the algorithm enters the **RETRY** stage of the fault-tolerance mechanism. In this stage the first thing it does is to check if the number of attempts to submit this ticket has already passed the configured threshold (max_retries in the **App** section of the configurations). Take into account that the actual number of attempts to check is always the number of attempts already made plus the current attempt. If it has passed, then the algorithm begins the **RECOVER** stage of the fault-tolerance mechanism by putting the ticket

in a pending state, awaiting for later re-submission by the remaining operations of the **RECOVER** stage. If the threshold has not been passed, then the fault-tolerance mechanism waits for the configured number of seconds (resubmission_interval in the **App** section in the configurations) before attempting re-submission. After that interval has gone through, the mechanism increments the number of submission attempts and finally, attempts to submit the ticket again, following the exact same process.

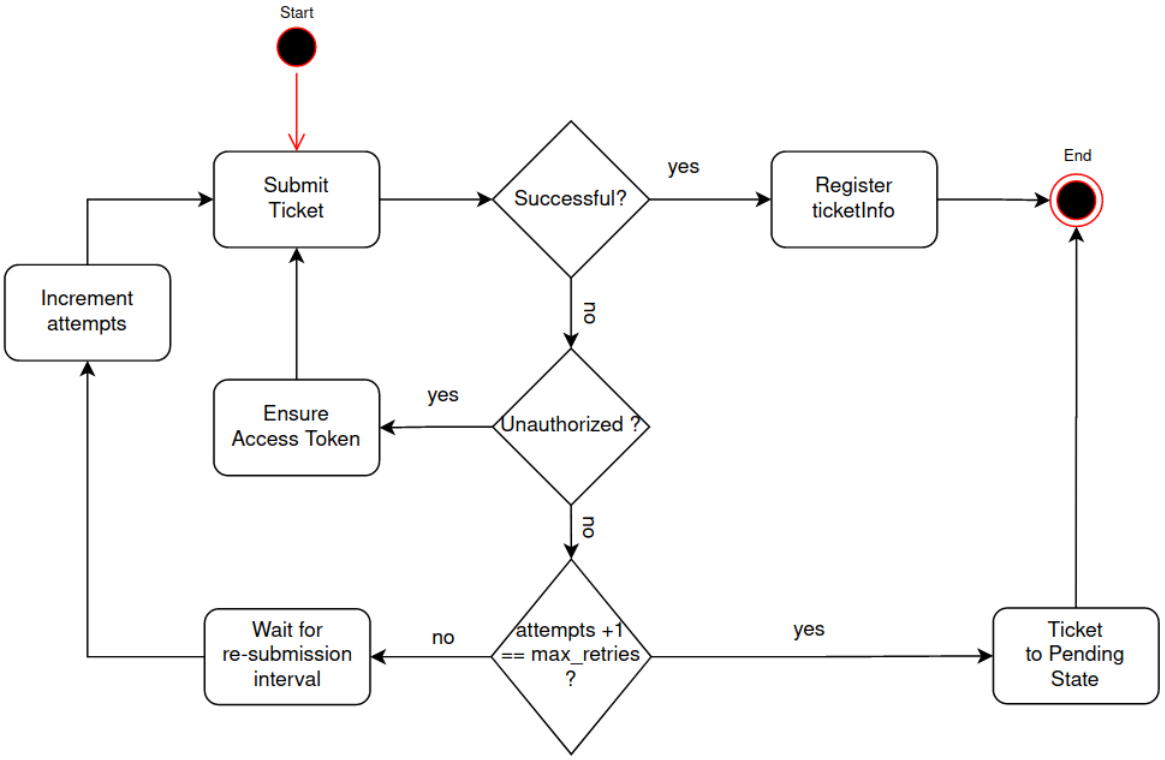


Figure 11: Illustration of the ticket submission algorithm with fault-tolerance

The definition of the algorithm shown in Figure 11 raises an important question: What happens to the tickets that are put in a pending state ?

The answer to this question is what remains of the **RECOVER** stage of the fault-tolerance mechanism which is done in a different mode of execution of the smart box communicator. Up until now, the only known mode of execution of the communicator is the finite execution, which starts with the construction of a weighing ticket and ends either with the successful submission of the ticket or with the ticket being put to the pending state but, in order to provide some way of handling pending weighing tickets, the communicator also provides an ”infinite” mode of execution, which is a mode that should be always running and periodically attempting to re-submit pending tickets. This mode of execution can be run by simply spawning an instance of the communicator with the flag *–submit_pending*. The spawned

instance will allways be active and, periodically, checking for pending weighing tickets to submit.

Figure 12 shows the diagram that describes the algorithm followed by this different mode of execution of the communicator, which allows it to recover pending weighing tickets.

The point of entrance to this algorithm is to submit all pending weighing tickets. The algorithm first checks if there is any pending ticket left to submit and, if there are none, then the algorithm ”sleeps” for a configured number of seconds (pending_interval in the **App** section of the configurations) before checking again. If there are weighing tickets to submit, then there is an attempt to submit the first on the list. If this attempt is successful then the algorithm simply registers the ticket information (ticket identifier and transaction hash) in the configured path and goes back to check if there are more pending weighing tickets to submit. If the attempt is unsuccessful, the algorithm first checks if the received error is an authorization error and, if so, it calls the method to ensure that a valid access token is returned, which was already described in the ticket submission algorithm and, after the access token is returned, an attempt to submit that ticket is performed again. If the error returned from the submission is not an authorization error, then the algorithm checks if the number of pending submissions for this ticket has already passed a certain threshold (max_pending_submissions in the **App** section of the configurations). If the number of attempts has not passed the threshold, then the number of attempts for this ticket is incremented and the ticket is saved again in the pending state for later re-submission and the algorithm goes back to check if there are more pending weighing tickets to submit. If the number of attempts has passed the configured threshold, then the ticket is put in a failed state and the algorithm returns to check if there are more pending weighing tickets to submit. The failed state is essentially a state in which the ticket will no longer be automatically submitted, since the number of attempts to submit it have already been too many. When a ticket is put in a failed state, a UNIX timestamp is taken as to know when that ticket failed for the last time, and it is associated with the ticket. This event is logged as an error so that the ones maintaining the communicator or the APIs may check what went wrong with this ticket’s submission.

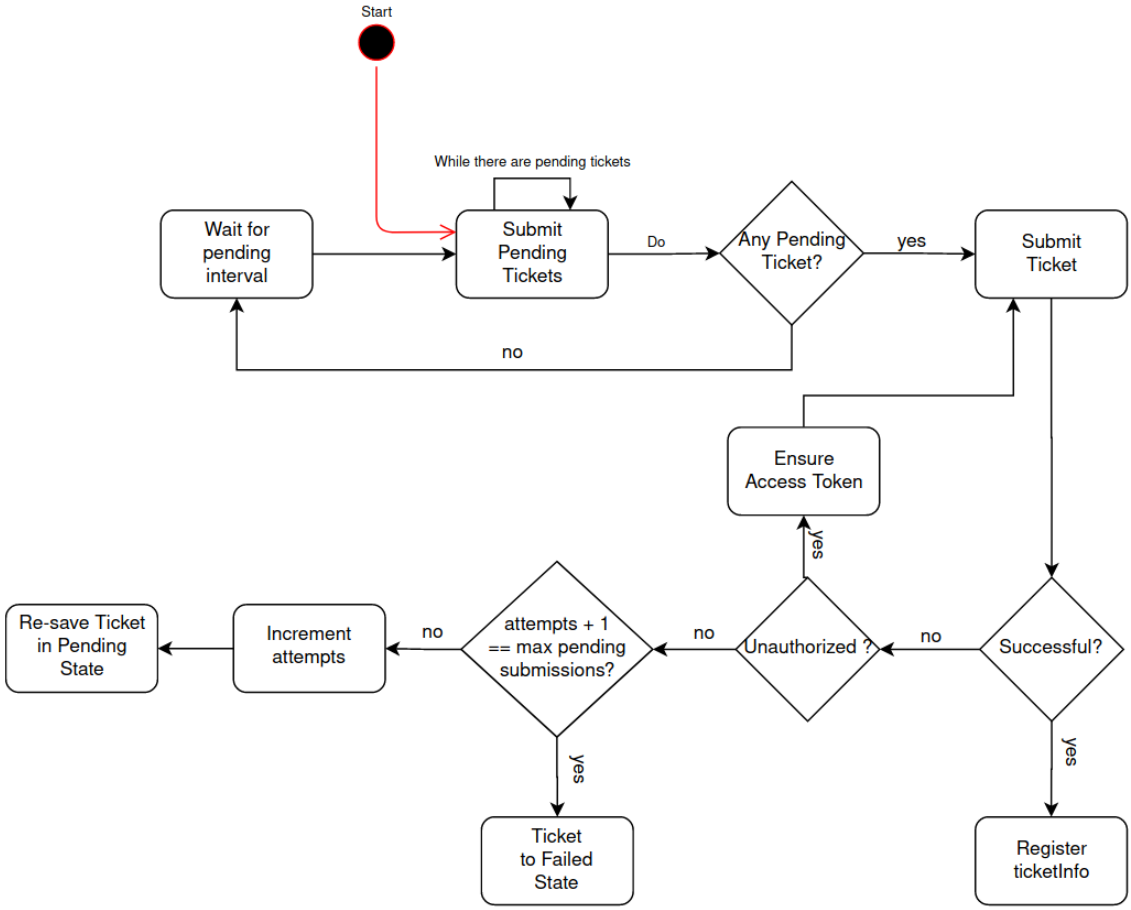


Figure 12: Illustration of the ”infinite” mode of execution of the smart box communicator

Finally, to wrap up the description of how the smart box communicator is implemented, two of the main algorithms of the **Authentication Manager** have to be explained, thus providing the full overview on the functionalities and the inner workings of the communicator.

Algorithm 11 describes the algorithm that ensures that an access token is returned when it is required, either by the algorithm in Figure 11 or in Figure 12. The steps followed by this algorithm are quite straightforward:

- It receives the global configuration object as input;
- Then it builds the TLS configuration object and instantiates the HTTPS client by passing it the just generated TLS configuration;
- After having the HTTPS client, the algorithm attempts to refresh its access token with the authentication & management API;
- If the previous request is unauthorized that means the refresh token is no longer valid and thus the program initiates a new authentication request and, then returns the necessary tokens;

- If all went well with the refresh request, then the algorithm simply returns the received tokens in the response.

Algorithm 11: Algorithm to ensure the existence of a valid access token

Input: config

Output: accessToken, refreshToken

```
1 tlsConfig ← buildTLSConfig(config) ;
2 httpsClient ← newHTTPSClient(tlsConfig) ;
3 response ← refreshToken(httpsClient, config.Tokens.refreshToken) ;
4 if response.Status ∈ Unauthorized then
5 |   config.Tokens ← authenticate(config) ;
6 |   return config.Tokens.accessToken, config.Tokens.refreshToken
7 else
8 |   return response.accessToken, config.Tokens.refreshToken
9 end
```

Algorithm 12 describes the authentication algorithm that requests for new authorization tokens to the authentication & management API. The logic of this algorithm can be described in the following way:

- The algorithm receives as input the global configuration object;
- Then it builds the TLS configuration object and instantiates the HTTPS client by passing it the just generated TLS config;
- After instantiating the HTTPS client, the algorithm collects both the public key of the authentication & management API (**apiPublicKey**) and the station's private key (**stationPrivateKey**);
- With those two parameters, the algorithm builds the authentication message for the station (Execution of Algorithm 1);
- Possessing the authentication message, the algorithm issues an authentication request to the authentication & management API;
- If the request, for some reason, does not succeed, this error is logged as a fatal error (which should be immediately checked);
- Otherwise the tokens are extracted from the response and returned accordingly.

The definition of these two last algorithms ends this subsection, where the implementation of the smart box communicator was explored. Essentially all the aspects of the implementation were covered, with highlight to how the communicator is able to submit tickets via a

Algorithm 12: Algorithm to perform the station's authentication process

Input: config
Output: accessToken, refreshToken

```
1  tlsConfig ← buildTLSConfig(config) ;
2  httpsClient ← newHTTPSClient(tlsConfig) ;
3  apiPublicKey getsreadFile(config.Auth_API.pubkeypath) ;
4  stationPrivateKey ← readFile(config.Station.key) ;
5  numberBytes ← getMessageByteCount() ;
6  authMessage ←
     buildStationAuthenticationMessage(numberBytes, apiPublicKey, stationPrivateKey) ;
7  response ← requestAuthentication(httpsClient, authMessage) ;
8  if response.Status ∈ Error then
9  |   logFatalError(response.Status, response.message) ;
10 else
11 |   return response.accessToken, response.refreshToken
12 end
```

trigger mechanism with fault-tolerance. This fault-tolerance mechanism is also, surely, one of the highlights of this implementation since it adds the necessary robustness and flexibility in operating this communicator. Finally, the main authentication and authorization handling algorithms were discussed in order to clarify how this program can securely authenticate and authorize each request it makes to both APIs.

4.2.6 *Load Cell Communicator*

With the definition of the implementation of the smart box communicator, the only point left to discuss in the implementation of the proposed solution is the communicator that runs in the load cells. As defined in Subsection 3.1.3: Technological Choice, this program only communicates using CoAP and DTLS with the smart box.

Before defining how this load cell communicator is implemented, the tools that are used to do it must be defined. Although the goal of the dissertation is to simply establish a definition and sample implementation of a secure IoT communication, which can be further optimized for a particular environment, the choice of implementation has to be careful in the sense that the tools used have to be able to be optimized or ported for restrained devices, which is the case of this load cell communicator. Nevertheless, after analysis the choice for the implementation of this program was Golang again and in this case, the reasons behind this decision are that: i) Golang has seen an incredible growth in the last couple years with the goal to make it a multi-environment PL, i.e., a PL that can perform well both in restrained and high-performance environments; And ii) as a follow-up to the first reason, this growth has lead to multiple libraries appearing that attempt to take Golang to the IoT world, such

as Gobot [49], EmbeddedGo [50] or TinyGO [51], especially the latter one which allows the compilation of Golang programs to multiple microcontroller architectures. Additionally, the familiarity with the PL also weighed in on the decision.

Having defined the PL to develop the program, the only remaining aspect that is left to define prior the actual implementation stage is the library that is going to be used for CoAP and DTLS communication which will be the same used in the smart box communicator, *PionDTLS* [48].

In order to describe the process of implementation of the load cell communicator, its necessary functionality has to be clearly defined first. By the knowledge acquired thus far in this document, it is clear that the load cell communicator only has to: i) Listen for requests made by a smart box communicator; And ii) respond with the weight it currently measures. So, basically, in this communicator, a server continuously listens for requests to a certain path and, each time it receives one, it handles the specific request.

Figure 13 illustrates the algorithm that runs in the load cell communicator and, as it can be seen, it is a simpe algorithm that goes through the following steps:

- The point of entrance to the algorithm is the function that instantiates a DTLS server and continuously listens for GET requests to the path */weight*;
- When a request is received, it is routed to the handler function which will deal with the response to that request;
- After routing the request to the handler, the communicator continues to listen for more requests.

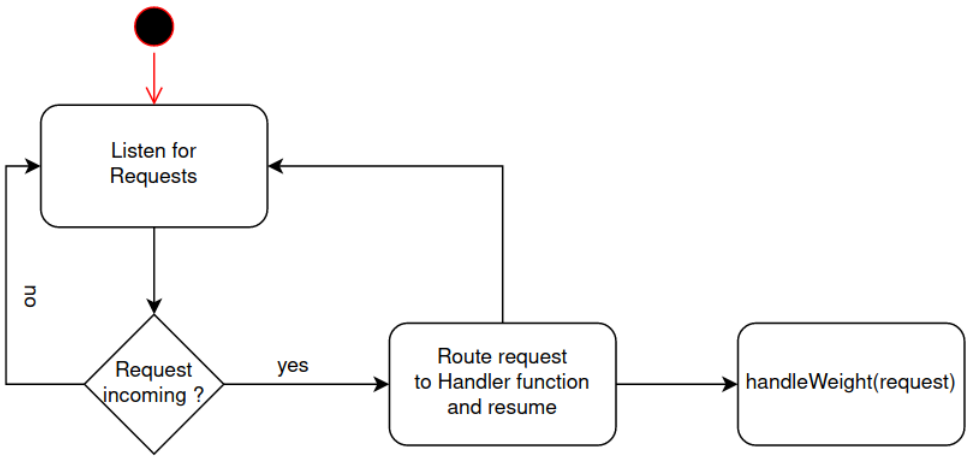


Figure 13: Algorithm running in the load cell communicators

While most of the steps in the algorithm seem pretty much straightforward, it is still worth describing how the **handleWeight** function actually works.

Algorithm 13 describes the workings of the **handleWeight** function, which is comprised by three simple steps:

1. Collect the weight currently measured by the load cell;
2. Convert that weight to bytes so that it can be sent across the network;
3. Respond to the request with the measured weight in bytes using the **responseWriter**.

Algorithm 13: Algorithm of the handleWeight function

Input: responseWriter

```
1 weight ← measureWeight() ;
2 weightBytes ← bytes(weight.toString()) ;
3 responseWriter.sendResponse(weightBytes) ;
```

The description of this algorithm completes the definition of the implementation process of the load cell communicator which clearly was an algorithm thought to be the simplest possible as to facilitate its probable optimization if, at a later stage, it is converted and compiled targeting microcontrollers.

4.3 SUMMARY

In this chapter the implementation process of the proposed solution was described, going through all the software components that are a direct result of it as well as the decisions that support the chosen path of implementation.

The chapter started with a plethora of decisions that had to be made prior beggining the actual implementation process such as the outcomes that had to be produced in order to theoretically comply will all the proposed goals, the structure of the blockchain network that best suited the use-case in study, as well as some more technical decisions such as how to mask the complexity of using a blockchain-based cloud system or even how to be able to implement a two-way authentication process with a unique authorization mechanism.

With the definition of all the required conceptual and technical decisions, the implementation process was then thoroughly explored. The description of this process was divided in six parts, where each part described an individual implementation process that was essential in order to provide the required functionality and component interoperability with the focus on complying with all the proposed goals. The implementation section first described the authentication & authorization process which, as said before, is a two-way authentication system with a unique authorization mechanism, since the authentication part has different algorithms for users and stations but the authorization mechanism is the same for both of them. Afterwards the smart contract was thoroughly described by defining the data

structures that it implemented as well as the methods that manipulated it in order to possess all the required functionality to manage the weighing tickets. Additionally, some different mechanisms were also discussed such as the application of filters directly in the smart contract to provide a way of diminishing the amount of weighing tickets in traffic from the blockchain network to the weighing tickets API.

From this point on, the implementation section starts deeply exploring the way how the cloud system is built and which components are part of it. Firstly, the data models that serve as the basis for the APIs were explored by clearly defining how many data models existed, to what entities they were related and how they were actually implemented, specifically defining the parameters that described each data model in the context of the cloud system. Following the definiton and description of the data models, the implementation of the APIs was explored and presented, starting by defining their common code structure and detailing how that structure abided by good software development principles with a clear separation of concerns. Then, the particularities of the weighing tickets API were described, including: i) how it implements a dynamic configuration mechanism and which type of parameters it allows to configure; ii) how it, in some way, tries to abstract the complexity of using smart contracts and blockchain technology; And, finally iii) how this API exposes its methods that provide a complete set of functions to manage and query weighing tickets. Having described the implementation of the weighing tickets API, there was one remaining major component of the cloud system to be explored, the authentication & management API. Accordingly, the implementation process of this API is then defined at a more deeper level, since its structure and architecture were already defined in the introduction to the description of the implementation of the cloud system APIs. In this part, the essential aspects of the authentication & management API were described, namely: i) Its configuration mechanism, which also allows a dynamic assignment of the configuration parameters via environment variables; ii) how the functionality implemented by it enforces the required network structure in the blockchain; And, finally iii) how it exposes its interface, providing a way of executing all the functionalities it implements.

Finally, the implementation section ends with the description of how the smart box and load cell communicators were built. In a first instance, the smart box's communicator is described, beggining with a choice of tools to implement it and then going through all the algorithms that are implemented in it in order to fullfil the goal of being able to build and submit weighing tickets to the API in a secure manner. In the definition of these algorithms, a big highlight of the implementation was clearly the fault-tolerance mechanism which ensures robustness and resiliency against flaws in the transmission of weighing tickets. In a second and last instance, the load cell communicator was described, starting by the definition of the tools used to implement it and the reason why those tools were chosen and ending with the description of the simple algorithm that runs in that communicator,

which essentially listens for requests and transmits weighings measured at that load cell. Ensuring that the implementation of this load cell communicator was as simple as it could was also an objective in order to foster and facilitate code optimizations that may be put in place at a latter stage of the project to enhance the communicator's usability in restrained environments.

This way, this chapter explored, defined and described all the components that comprise the proposed solution and that can, theoretically, comply with all the proposed goals.

5

PROOF OF CONCEPT

With the definition and description of the implementation process concluded, there is now a need to show how the solution that was built complies with the proposed goals and how it does it, by assembling a proof of concept that ensures the solution is actually capable of performing all the tasks it sets out to do.

In order to explain how the proof of concept was designed, set up and capable of providing the results that show and make evidence of the compliance with the objectives, this chapter is divided in four sections:

- Section 5.1: Experiment Setup describes the way how the proof of concept was designed with the goal to demonstrate the solution is capable of complying with the pre-defined objectives, as well as how it was actually set up for being carried out;
- Section 5.2: Results illustrates the results obtained with the execution of the proof of concept;
- Section 5.3: Discussion explores the results illustrated in the previous section, by comparing what was obtained to what should have been obtained, concluding on how each particular result is compliant to what should have happened;
- Section 5.4: Summary essentially summarizes the points previously discussed, by providing an overall conclusion to the execution of the proof of concept.

For the proof of concept to be considered successful, metrics have to be defined that clarify if the solution was able to deal with that particular aspect and, with that in mind, the essential aspects that have to be demonstrated are:

- That the communication both between the load cells and the smart box and between the smart box and any of the APIs is secured, providing confidentiality of the information that is transmitted;
- That the communicators in the station are capable of transmitting weighing tickets to the API and, consequently, to the blockchain ledger, receiving the transaction hash, i.e., the unique proof that the transaction was in fact registered in the ledger;

- That the communicators in the station are fault-tolerant, i.e., downtime in one of the APIs will not result in weighing tickets being lost;
- That the weighing tickets are correctly registered, i.e., the characteristics of a dataset of weighing tickets prior registration is mantained after registration;
- That weighing tickets of a specific customer cannot be accessed by another customer through the Weighing Tickets API.

Having defined the requirements to provide evidence that the solution can correspond with the expectation in terms of the objectives that it has to comply, the next section describes the setup of the experiment that was assembled in order to do that.

5.1 EXPERIMENT SETUP

With the goal to demonstrate the correctness of the solution, an experiment has to be setup that can execute the processes necessary to assess if the solution is able or not to comply with each of the previously defined requirements.

In order to build an experiment that is appropriated for the use case in study it can, at least, be perceived that it has to possess running instances of the components that are part of the cloud system: i) Blockchain network; ii) Weighing Tickets API; iii) Authentication & Management API; And iv) the entities database. Additionally, the experiment should demonstrate the usability of the communicators, both of the load cell and the smart box, by assuming the existence of, at least, two stations, in order to show the confidentiality of each one's data, and configuring the communicators as belonging to each of the stations.

To succeed in clearly describing the experience that was produced, this section is divided in three essential steps:

1. The design of the architecture of the experiment, i.e., how many components and how they will be executed in a way that attempts to best demonstrate the concept;
2. The definition, preparation and preprocessing of a dataset of weighings provided by Bilanciai, to serve as the source data of weighing tickets to be transmitted by the smart box communicators;
3. The execution of the experiment, where the configuration of each of the components and the clear definition of the beggining and end of execution is given.

5.1.1 *Experiment Architecture*

In this section, the experiment architecture is shown and discussed, explaining why it was built like so and how that architecure allows us to test all the requirements that were defined in the beggining of this chapter.

Figure 14 illustrates the architecture of the proof of concept. In a first glance, it can clearly be seen that this architecture has three major components: i) The cloud system; ii) Customer X's station; And iii) Customer Y's station. Additionaly, two logos appear constantly, one in most of the components, whether they belong to the cloud system or the stations and the other appears in each of the stations. The first logo represents *Docker* [52], a containerization system that allows to clearly define the dependencies and how an application must be executed, so that its execution is always ensured. The second logo is associated with *Raspberry Pi* [53], a small form-factor computer, which is present since each of the two stations present in the architecture will have the communicators run in their own *Raspberry Pi* computer. In summary, the cloud system will all be run in a laptop via *Docker* orchestration, customer X's station will be run in one *Raspberry Pi*, referred to from now on by *pi1* and customer Y's station will be run in another *Raspberry Pi*, referred to from now on by *pi2*.

Beyond the hardware aspects of this architecture, there are also some applicational aspects that have to be explained. First, the blockchain network appears in this architecture with 5 nodes and their associated privacy managers. This is due to the fact that, with this architecture, five entities will be created in the network:

1. The network's administrator node (node 1 & privacy manager 1), an entity that already exists on system boot-up;
2. Customer X's administrator node (node 2 & privacy manager 2), created by the network administrator in the Authentication & Management API;
3. Station X's node (node 3 & privacy manager 3), created by customer X in the Authentication & Management API;
4. Customer Y's administrator node (node 4 & privacy manager 4), created by the network administrator in the Authentication & Management API;
5. Station Y's node (node 5 & privacy manager 5), created by customer Y in the Authentication & Management API.

Second and last, the stations will be defined as possessing two weighbridges each, where each of the weighbridges hold four load cells. The load cell communicators will run in *Docker* containers, but the smart box communicator will run regularly as a process in the correspondent *Raspberry Pi*, *pi1* for Station X and *pi2* for Station Y.

In this last point, two stations, one from each customer, were defined in order to be able to demonstrate that the weighing tickets transmitted by one customer cannot be viewed by another customer, since they transmit them to different smart contracts

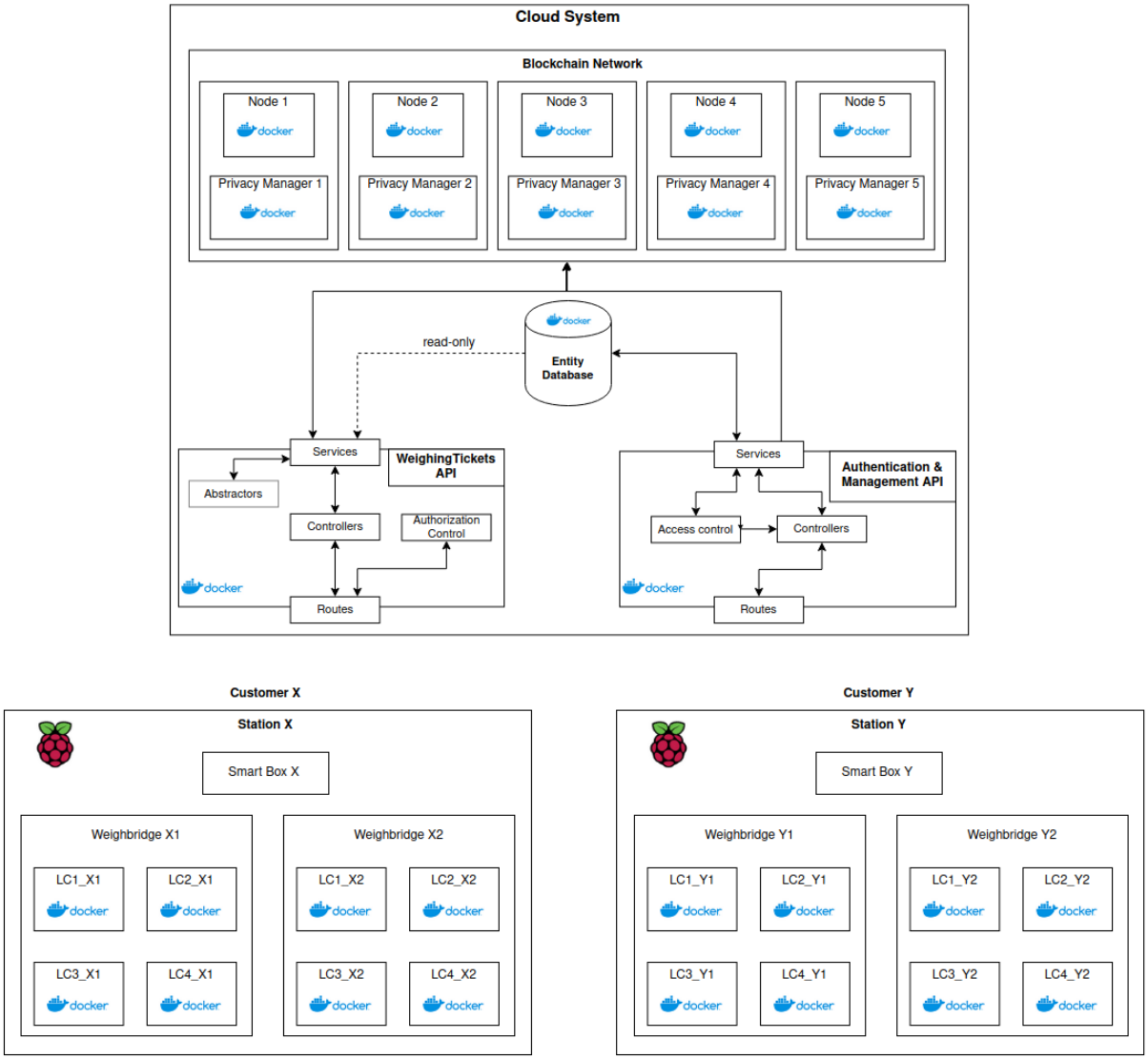


Figure 14: Architecture of the proof of concept

Additionally, it is worth mentioning that, although, the weighbridges are defined as X1, X2, Y1 and Y2, their actual serial numbers are different and these acronyms serve only as a simplification for representation. The actual serial numbers mapping is the following:

- Weighbridge X1 has a serial number of P191021852;
- Weighbridge X2 has a serial number of P220120900;
- Weighbridge Y1 has a serial number of P141140200;

- Weighbridge Y2 has a serial number of P300200111.

Having the setup been clearly defined, including the devices where each of the components will operate and why those components were chosen as so (e.g. blockchain network), the next step is to prepare a dataset of weighing tickets to serve as the basis of this proof of concept, since those are the tickets that are going to be emitted by the communicators to the Weighing Tickets API and, lastly, to the ledger. This proof of concept was done by using a real dataset of weighings, given by Bilanciai. Actual devices were not used for weight reading due to unavailability of real load cell and smart box devices and, of course, due to proprietary algorithms that are applied in the readings that could not have been included in a work of this public nature.

5.1.2 *Dataset Preparation and Metrics*

The purpose of this section is actually two-folded:

1. To clearly describe how, from an initial dataset provided by the company, the data was treated to serve as the base dataset for the proof of concept, i.e., the weighings that are going to be sent by the stations;
2. To unmistakably define the characteristics of the dataset that is going to be used as basis so that, after the execution of the proof of concept the weighing tickets transmitted from the dataset are exactly as they sould be in the ledger.

The first step in describing the preparation process of the dataset is to clearly define the starting point, i.e., the raw dataset that was delivered for this proof of concept.

This dataset comprises weighings emitted by one only test station from the companies, which holds eight load cells. With that said, the dataset possesses twenty columns:

- **terminalSerialNumber**. The serial number of the smart box that transmitted this weighing;
- **scaleSerialNumber**. The serial number of the weighbridge where the weighing was measured;
- **scaleGross**. The gross value of the weight measured (Equal to the sum of the load cells' weight);
- **scaleNet**. The net value of the weight measured (Equal to the sum of the load cells' weight);
- For each of the load cells, which are eight in total, there are two attributes:
  - **cellSerialNumber**. The serial number of the load cell;

– **cellWeight**. The weight measured by the load cell.

From this description of the dataset some aspects can be immediately recognized: i) Since there is no proprietary algorithm running in the measurement of weights, the total weight will be considered to be the sum of the loadcells' weight, both for **scaleGross** and **scaleNet**; ii) there is no **timestamp** because that will be taken when these weighings are transmitted by the smart box communicator; And iii) the dataset does not comprise two other attributes that should exist in the weighing ticket, **scaleStatus** and **terminalRestartValue**, so they will have to be manually added.

Having completely characterized the dataset, the next step that was performed was to remove rows of data that had missing values, i.e., cells that had no value despite being obligatory, so that only correct weighings are used for the transmision of weighing tickets. After this step, it can be ensured that the dataset holds all the required properties, with exception to **scaleStatus** and **terminalRestartValue** which will be later added.

The next decision to make is on the amount of samples to use for the transmission of weighing tickets and, recalling the fact stated in Subsection 3.1.3, that a station transmits, at maximum 300 weighings per station and, taking into account that this proof of concept contemplates two stations, each with two weighbridges, the number of samples to be transmitted will be 600 in total, 300 per station, 150 per weighbridge, which allows the simulation of a "day's work", albeit in a smaller transmission time frame. Since the dataset received only has one single **scaleSerialNumber** value, because they were taken from one weighbridge, that attribute was altered so that the final dataset could have 150 weighings per weighbridge and, thus, the distribution of the weighings was done as follows (zero-indexed):

- Station X's weighbridge 1 (serial number *P191021852*) is associated with weighings 0 until 149;
- Station X's weighbridge 2 (serial number *P220120900*) is associated with weighings 150 until 299;
- Station Y's weighbridge 1 (serial number *P141140200*) is associated with weighings 300 until 449;
- Station Y's weighbridge 2 (serial number *P300200111*) is associated with weighings 450 until 599.

With the structure and amount of data defined, the process goes on by including the two attributes that were missing and, although there was no clear definition for the possible values of **terminalRestartValue**, **scaleStatus** will be considered to have two possible values: i) *OK*; ii) *FAULTY*. For the **terminalRestartValue** attribute, all rows will be filled with a *CONNECTED* value, which only serves the purpose of filling that attribute, it has no semantic meaning. For the **scaleStatus** attribute and, taking into account that the verification of results

will also include using the filters provided by the Weighing Tickets API when querying tickets, both different values will be used, although of course the *FAULTY* value in a much smaller frequency.

As was already mentioned, each weighbridge in each station will be associated with 150 weighings and, taking that value into account, the distribution of the **scaleStatus** attribute was done as follows:

- Station X's weighbridge 1: 147 weighings with **scaleStatus** *OK*; 3 weighings with **scaleStatus** *FAULTY*;
- Station X's weighbridge 2: 148 weighings with **scaleStatus** *OK*; 2 weighings with **scaleStatus** *FAULTY*;
- Station Y's weighbridge 1: 150 weighings with **scaleStatus** *OK*; 0 weighings with **scaleStatus** *FAULTY*;
- Station Y's weighbridge 2: 146 weighings with **scaleStatus** *OK*; 4 weighings with **scaleStatus** *FAULTY*.

The definition and insertion of these attributes into the dataset results in a correctly deifned dataset, suited to be the basis of the proof of concept. In order to further treat the dataset, the process on how that data will be read by the communicators to form the weighing ticket has to be explained, since by the reasons already mentioned, the communicators will not read weights from an actual device. With that in mind, the process of building a weighing ticket is essentially a three-step process:

1. The smart box communicator reads the generic weighing ticket attributes, such as the **scaleSerialNumber**, **terminalSerialNumber**, **scaleStatus** and **terminalRestartValue** and then requests the weights of that weighing to the load cells associated with that weighbridge;
2. Each of the load cells responds to the smart box communicator with its corresponding weight;
3. The smart box communicator receives the weights of all load cells, calculates the **scaleNet** and **scaleGross** attributes (sum of all load cell weights) and this terminates the process of building the weighing ticket, which is now ready for submission.

By overviewing the aforementioned ticket building process, it can be stated that: i) each weighbridge has to have access to the generic attributes for each weighing; ii) Each load cell has to have access to its weight for each weighing. So, the ultimate goal in this data preparation process is to create twenty JSON files, which will then be placed at the appropriate location:

- One file per weighbridge, in a total of four files, containing an array of objects, in which each object represents the generic attributes of a weighing. **scaleSerialNumber** and **terminalSerialNumber** are not included in these objects, since those values are already configured in the smart box communicator and so, there is no need to duplicate information;
- One file per load cell, in a total of sixteen files, containing an array of floats, where each float represents the weight measured by the load cell in that weighing.

Prior applying the transformation process that converts the dataset in possession to the twenty aforementioned JSON files, there is one last item to take into account, which is the serial number of the load cells. Since the original weighings were only measured in one weighbridge, there are also only eight different load cell serial numbers. Additionally, for this proof of concept only four of them are required. So, to tackle this issue, only four of the original load cell serial numbers were used: *7450332.0*, *7450333.0*, *7450339.0* and *7450340.0*. These four load cell serial numbers were assigned to Customer X's weighbridge 1. The remaining assignment of load cell serial numbers per weighbridge was conducted as follows:

- Customer X's weighbridge 2 was defined as having almost the same serial numbers, but instead of starting by *745*, they start by *755*;
- Customer Y's weighbridge 1 was defined as having almost the same serial numbers, but instead of starting by *745* or *755*, they start by *765*;
- Customer Y's weighbridge 2 was defined as having almost the same serial numbers, but instead of starting by *745* or *755* or *765*, they start by *775*.

At this point, the data is split in four datasets, one per weighbridge and now, the main goal is to export that data into JSON files with the already aforementioned format. This exporting process followed, at a high level, Algorithm 14, with the final goal to export all the data to the necessary files.

The process described in Algorithm 14 can be explained as follows:

- The first step is to initialize the array that will hold the generic attributes for each of the weighings, one per index and initialize the object that will map a load cell serial number to the weights measured by it in each weighing, one weight per index;
- After initializing the required variables, the algorithm iterates through each of the rows existent in **weighbridge_data** and for each one:
  - For each of the load cells existent in that weighing, extract its weight and append it to the array associated with that load cell serial number;
  - Append the generic attributes required (**scaleStatus**, **terminalRestartValue**) to the array of generic attributes for the weighbridge.

- Write the array of generic attributes collected to a file, whose name will have the format *weighbridge_serial_number.json*, where **weighbridge_serial_number** is the value received as input;
- For each of the load cells existent in this weighbridge, collect its weighings data from **loadcell_data** and write that array into a file, whose name will have the format *loadcell_serial_number.json*, where **loadcell_serial_number** is the current value in the loop.

Algorithm 14: Export process of the data to the correct files

Input: weighbridge_data, weighbridge_serial_number, weighbridge_loadcell_serial_numbers

```
1  weighbridge_generic_attributes ← [] ;
2  loadcell_data ← {} ;
3  foreach loadcell_serial_number ∈ weighbridge_loadcell_serial_numbers do
4  |  loadcell_data[loadcell_serial_number] ← [] ;
5  end
6  foreach row ∈ weighbridge_data do
7  |  foreach loadcell_serial_number ∈ weighbridge_loadcell_serial_numbers do
8  |  |  cellWeight ← row["loadcells"][loadcell_serial_number]["cellWeight"] ;
9  |  |  loadcell_data[loadcell_serial_number].append(cellWeight) ;
10 |  end
11 |  generic_attributes ← {"scaleStatus" : row["scaleStatus"], "terminalRestartValue" :
   |   row["terminalRestartValue"]} ;
12 |  weighbridge_generic_attributes.append(generic_attributes) ;
13 end
14 writeToFile(weighbridge_serial_number, weighbridge_generic_attributes) ;
15 foreach loadcell_serial_number ∈ weighbridge_loadcell_serial_numbers do
16 |  writeToFile(loadcell_serial_number, loadcell_data[loadcell_serial_number]) ;
17 end
```

At the end of the execution of this algorithm, all files that are required, i.e. four associated with the four existent weighbridges and sixteen associated with the sixteen existent load cells, are created and ready for use, which concludes the process of preparing and transforming the original dataset to serve as the basis for the proof of concept.

To conclude this section, what is left is to characterize this dataset, so that values obtained after the execution of the proof of concept can be compared with these values and ensure that all weighing tickets were correctly transmitted.

Throughout this section, some of the metrics were already established, such as the fact that there have to be 600 weighings in total, 300 per each station and 150 per each weighbridge. Additionally, the number of weighing tickets with **scaleStatus** *OK* and *FAULTY* were also already defined for each weighbridge. In order to complete this characterization and to provide clearer evidence that the weighing tickets are correctly transmitted and stored, the dataset will also be characterized in terms of the weight distribution for each weighbridge. As it was referred in subsubsection 4.2.4, the Weighing Tickets API has capabilities to filter weighing tickets by single weight value comparison and by weight interval comparison and, with that in mind, the last metric to be used for the characterization of the dataset is the distribution of the weight in the weighings in each weighbridge below an X value (exclusive), between an X value (inclusive) and a Y value (exclusive) and, finally, above an Y value (inclusive). Since any values that are chosen will correctly define the distribution of weighings, for this exercise, X value was chosen to be 50 KG and the Y value was chosen to be 1000 KG. So, in summary, when the weighing tickets are all submitted, its weighing distribution has to comply with the following properties, for each weighbridge:

- There are N weighings with a total weight until 50 KG;
- There are M weighings with a total weight from 50 KG until 1000 KG;
- There are K weighings with a total weight from 1000 KG.

With this last characterization aspect of the dataset clearly defined, what remains is to actually calculate the values of N, M and K for each weighbridge for later comparison. The algorithm followed for this calculation is really simple:

- First it calculates the total weight for all the weighings of each weighbridge;
- After that, an object is declared which holds the number of weighings that fit in the interval of that key. The possible keys are **until50**, **from50_until1000** and **from1000**, which are self-explanatory in what they represent;
- Then, the array of total weights is iterated for each weighbridge and, if the value of total weight is lesser than 50 KG (exclusive), **until50** is incremented; If the value of total weight is greater than or equal to 50 KG, but lesser than 1000 KG (exclusive), **from50_until1000** is incremented; And, finally, if none of the previous conditions match, **from1000** is incremented.

Figure 15 illustrates the results obtained, showing the values for each weighbridge, from where we can conclude that:

- Customer X's weighbridge 1 has 28 weighings until 50 KG; 7 weighings from 50 KG to 1000 KG; And 115 weighings from 1000 KG;

- Customer X's weighbridge 2 has 50 weighings until 50 KG; 16 weighings from 50 KG to 1000 KG; And 84 weighings from 1000 KG;
- Customer Y's weighbridge 1 has 15 weighings until 50 KG; 1 weighings from 50 KG to 1000 KG; And 134 weighings from 1000 KG;
- Customer Y's weighbridge 2 has 37 weighings until 50 KG; 8 weighings from 50 KG to 1000 KG; And 105 weighings from 1000 KG.

```
1 weight_distribution_customerx_weighbridge1
{'until50': 28, 'from50_until1000': 7, 'from1000': 115}

1 weight_distribution_customerx_weighbridge2
{'until50': 50, 'from50_until1000': 16, 'from1000': 84}

1 weight_distribution_customery_weighbridge1
{'until50': 15, 'from50_until1000': 1, 'from1000': 134}

1 weight_distribution_customery_weighbridge2
{'until50': 37, 'from50_until1000': 8, 'from1000': 105}
```

Figure 15: Weight distribution over intervals for each weighbridge

The description of the dataset's weight distribution concludes this section and now the chapter focuses on the description of the process that was conducted to execute the proof of concept, in the next section.

5.1.3 *Experiment Execution*

With both the experiment architecture, and the definition of the dataset preparation process as well as the metrics that should be observed to perceive if the proof of concept was successful or not, what remains to explain in the setup of the experiment is the actual way how it was conducted, i.e., how each of the components were configured, how they were ran and what additional actions where put in place in order to obtain the necessary results to evaluate the correctness of the solution.

The first step that had to be done was to be aware of the architecture of the devices that will run each of the software components. While the APIs will run in a laptop with a x64 CPU architecture and thus, no change has to be made to their compilation process,

both the smart box communicator and the load cell communicator, will run in devices with an *Advanced RISC Machine (ARM)* CPU architecture and so, they have to be instructed to be compiled for those environments. This essentially means that the *Docker* containers running the load cell communicators will have an ARM based image and that the smart box communicator executable will be built/compiled targeting the ARM CPU architecture.

After correctly targeting the compilation step for the devices that existed, the next step is to extract the *Internet Protocol (IP)* addresses of each of the devices, to be able to copy the required files for execution and, additionally, to be able to configure the smart box communicator with the capability to discover the host where the APIs are running. So, by simply running $ifconfig$ in each of the device's terminals, their IP address of the wireless interface could be extracted. The results obtained were:

- The host where the APIs and the blockchain network run has an IP address of *192.168.1.231*;
- *pi1*, the raspberry device where station X runs, has an IP address of *192.168.1.203*;
- *pi2*, the raspberry device where station Y runs, has an IP address of *192.168.1.202*.

With the IP information extracted, the next and final step before being able to "boot up" the cloud system is to generate the required public key certificates for use in the communication and, since this step is being executed, certificates for all the other devices are also going to be generated.

Figure 16 illustrates the hierarchy of public key certificates implemented. Essentially, there is one root CA, which digitally signs the public key certificates for each of the required entites: i) The Weighing Tickets API; ii) the Authentication & Management API; iii) both the smart box communicators; And iv) all the load cells from each of the stations.

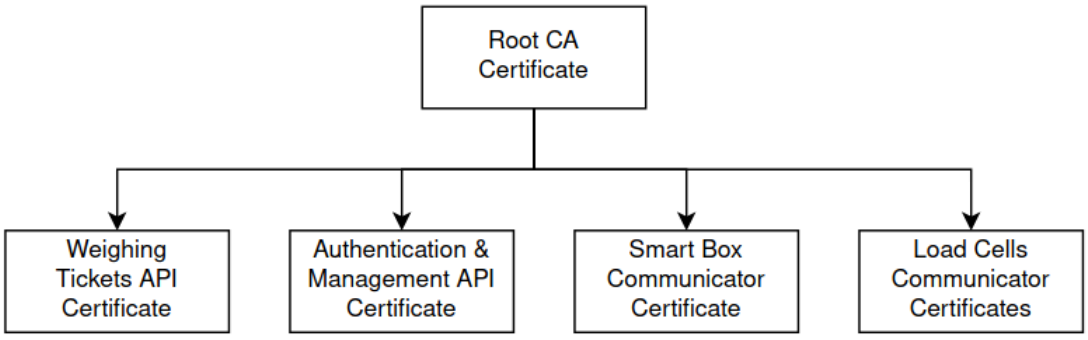


Figure 16: Hierarchy of the public key certificates

The generation process of these certificates was conducted using the *OpenSSL*'s *Command Line Interface (CLI)* [54]. The key pair generation algorithm used for every entity was an

Elliptic Curve algorithm, which essentially provides the same security with a much smaller key size.

The certificate generation can be viewed as a two-step process, since the second step of the process is repetitive for each of the entities. The first step of the process was conducted in the following way:

1. Generate an elliptic curve cryptographic key pair for the root CA;
2. Create a self-signed certificate for the root CA.

After generating the required artifacts for the root CA, the next step was to generate certificates for the remaining entities. For each of the entities, the steps followed in the certificate generation process were:

1. Generate an elliptic curve cryptographic key pair for the entity;
2. Generate a certificate signing request with the information of the entity;
3. Sign the certificate signing request of the entity with the root CAs private key, producing the entity's certificate.

With the conclusion of the certificates' generation process, the cloud system was ready to be "booted up".

Figure 17 shows information on the running cloud system after a status check request, which demonstrates that all of the required components are running, namely:

- The Authentication & Management API (*poc_auth_api_1*);
- The Weighing Tickets API (*poc_weighingtickets_api_1*);
- The Entities Database (*mongo_database*);
- The Blockchain Network with five nodes:
  - Node 1 (*poc_node1_1*) and its Privacy Manager (*poc_txmanager1_1*);
  - Node 2 (*poc_node2_1*) and its Privacy Manager (*poc_txmanager2_1*);
  - Node 3 (*poc_node3_1*) and its Privacy Manager (*poc_txmanager3_1*);
  - Node 4 (*poc_node4_1*) and its Privacy Manager (*poc_txmanager4_1*);
  - Node 5 (*poc_node5_1*) and its Privacy Manager (*poc_txmanager5_1*).

```
CONTAINER ID        NAMES                       STATUS
34f2546ca1bf        poc_node3_1                 Up 39 seconds (healthy)
d4066a78480d        poc_node2_1                 Up 39 seconds (healthy)
66977f719745        poc_weighingtickets_api_1   Up 39 seconds
6c8daab8c08b        poc_auth_api_1              Up 39 seconds
fbce7b5a58bf        poc_node1_1                 Up 39 seconds (healthy)
abe4546f3890        poc_node5_1                 Up 40 seconds (healthy)
b5fa91a8e165        poc_node4_1                 Up 39 seconds (healthy)
865bc7d23105        poc_txmanager4_1            Up 42 seconds (healthy)
4337f1267dfe        poc_txmanager1_1            Up 41 seconds (healthy)
89417dc1abf1        poc_txmanager5_1            Up 42 seconds (healthy)
fd4db2df3dec        mongo_database              Up 41 seconds
eda0d12eeecb        poc_txmanager2_1            Up 41 seconds (healthy)
3db30be161ff        poc_txmanager3_1            Up 41 seconds (healthy)
```

Figure 17: Status information of the cloud system

With the cloud system running, the next step that had to be made is the registration of the customers and their correspondent stations, so that the cloud system is aware of their existence and allows the registration and subsequent query of weighing tickets. To be able to perform the registration of customer X and Y, it has to be remembered that the user that makes these requests has to be an administrator, which is already registered at cloud system "boot up". So, three sequential steps were put in place to register the customers, remembering that the IP address of the cloud system is the *192.168.1.231* and that the Authentication & Management API listens in port 3001:

1. Perform a POST request to *https://192.168.1.231:3001/authorization* with the administrator's credentials in order to obtain access and refresh tokens;

2. Including the access token in the headers, perform a POST request to *https://192.168.1.231:3001/customers* with customer X's information, namely its name, the email of its administrator and chosen password, the address of the blockchain node for this customer, which is node 2, the url for accessing the node and the public key of its privacy manager. These three last blockchain-related informations are all provided on node creation;

3. Including the access token in the headers, perform a POST request to *https://192.168.1.231:3001/customers* with customer Y's information, namely its name, the email of its administrator and chosen password, the address of the blockchain node for this customer, which is node 4, the url for accessing the node and the public key of its privacy manager. These three last blockchain-related informations are all provided on node creation.

With both customers registered in the system, the goal now is to register their stations as to make possible the communication from the stations' smart boxes. As a side note, it is worth mentioning that the first action of any user or customer in the system has to be to update its password, which is included as an additional security measure due to the fact that the

password is chosen by the system administrator. After changing both customers' passwords, the execution of this experiment goes on to the registration of the stations and, assuming that an access token for each customer already exists, since they are already authenticated, the two following steps were:

1. Including the customer X's administrator access token, perform a POST request to *https://192.168.1.231:3001/stations* with the information on its station, namely the station's descriptive name, the station's public key for authentication purposes, the station's blockchain node's address, which is node 3, the url for accessing the node and the public key of its privacy manager. These three last blockchain-related informations are all provided on node creation;

2. Including the customer Y's administrator access token, perform a POST request to *https://192.168.1.231:3001/stations* with the information on its station, namely the station's descriptive name, the station's public key for authentication purposes, the station's blockchain node's address, which is node 5, the url for accessing the node and the public key of its privacy manager. These three last blockchain-related informations are all provided on node creation.

With these last two steps, the registration of entities in the cloud system is concluded and, as Figure 18 shows, there are two stations known to the cloud system in the database, station X and station Y, which have their unique system identifiers.

```
> db.stations.find({},{_id: true, name: true})
{ "_id" : ObjectId("5fad74ac75c459001915bb8c"), "name" : "Y" }
{ "_id" : ObjectId("5fad74f075c459001915bb8d"), "name" : "X" }
```

Figure 18: Identifiers of the stations, X and Y

After concluding the entity registration step, the process continued with the configuration of the smart box communicator for each of the stations following the configuration syntax defined in Table 18: SmartBox Configurations.

Table 19 describes the values assigned to each of the configurable attributes in both the smart box of station X and Y. Obviously, the certificate files that identify the smart box have different contents from station X and Y, but the path to that certificate is exactly the same in both the smart boxes. Additionaly, the **station** configuration parameters are not present in this table since they almost completely differ from one station to the other and so, those parameters will be presented separately. Despite that, aside from the parameters that identify the certificate and key files, as well as the hosts and ports where the components listen, which are self-explanatory, there are some parameters worth further explaining such as the ones that comprise the fault-tolerance mechanism.

As it can be seen in Table 19, the communicator attempts to submit weighing tickets at most three times, with a 5 second interval between those submissions and, if it fails all those submissions, the ticket is put into a pending state. Regarding the weighing tickets that are in a pending state, the communicator attempts to resubmit them at most 5 times, with a 10 second interval between submission and, if the submission fails those 5 times, the weighing ticket is then put in a failed state, where it has to manually be checked for further resubmission.

	Name	Value
Smart Box	host	127.0.0.1
	port	8000
	certpath	Read from the *credentials* folder in a file named *smartbox.crt.pem*
	keypath	Read from the *credentials* folder in a file named *smartbox.key.pem*
App	token_path	Located in the *config* folder in the file *tokens.json*
	ticket_info_path	Located in the *app_data* folder in the file *ticket_info.json*
	pending_path	Located in the *app_data* folder in the file *pending.json*
	failed_path	Located in the *app_data* folder in the file *failed.json*
	max_pending_submissions	5 submissions
	resubmission_interval	5 seconds
	max_retries	3 retries
	pending_interval	10 seconds
Auth_API	host	192.168.1.231
	port	3001
	pubkeypath	Read from the *credentials* folder from file *jwt.pub.key*
CA	certpath	Read from the *credentials* folder from file *ca.crt.pem*
Tickets_API	host	192.168.1.231
	port	3000

Table 19: Common configuration parameters of the smart boxes from station X and Y

Aside from the common configuration parameters assigned to either station X's smart box communicator and station Y's smart box communicator, there is a major difference in the **station** section of the configurations.

Table 20 shows the configuration parameters that were passed to the smart box communicator associated with station X in the **station** section of the parameters, i.e, the one running in *pi1*. As it can be seen in the table, the **id** parameter is equal to the station X identifier provided in Figure 18. Additionally, the station's private and public keys for authentication purposes are read from the files indicated in the table and the smart box's serial number is 160690.

	Name	Value
Station	id	5fad74f075c459001915bb8d
	key	Read from the *credentials* folder from file *station.jwt.key.pem*
	keypub	Read from the *credentials* folder from file *station.jwt.key.pub.pem*
	terminal	160690
	weighbridges	* (described below)

Table 20: Configuration of the **station** section of station X's smart box communicator

The **weighbridges** parameter is hard to describe in a table format since it is an object, thus it is provided in the configuration file as follows:

```
weighbridges:
    P191021852:
        - id: "7450332.0"
        address: 127.0.0.1
        port: 8888
        - id: "7450333.0"
        address: 127.0.0.1
        port: 8889
        - id: "7450339.0"
        address: 127.0.0.1
        port: 8890
        - id: "7450340.0"
        address: 127.0.0.1
        port: 8891
    P220120900:
        - id: "7550332.0"
```

```
17              address: 127.0.0.1
18              port: 8892
19          - id: "7550333.0"
20              address: 127.0.0.1
21              port: 8893
22          - id: "7550339.0"
23              address: 127.0.0.1
24              port: 8894
25          - id: "7550340.0"
26              address: 127.0.0.1
27              port: 8895
```

The above configuration essentialy tells the smart box communicator that it has the capability to monitor two weighbridges, with serial numbers P191021852 and P220120900, which both have four load cells each. For each of the four load cells in each weighbridge, their serial number (**id**), the address and the port where they listen are provided. The address in this case is the local host, since the loadcells, running in *Docker*, also transmit their traffic to the local host's port specified in this configuration file

Table 20 shows the configuration parameters that were passed to the smart box communicator associated with station Y in the **station** section of the parameters, i.e, the one running in *pi2*. As it can be seen in the table, the **id** parameter is equal to the station Y identifier provided in Figure 18. Additionally, the station's private and public keys for authentication purposes are read from the files indicated in the table and the smart box's serial number is 270100.

	Name	Value
Station	id	5fad74ac75c459001915bb8c
	key	Read from the *credentials* folder from file *station.jwt.key.pem*
	keypub	Read from the *credentials* folder from file *station.jwt.key.pub.pem*
	terminal	270100
	weighbridges	* (described below)

Table 21: Configuration of the **station** section of station Y's smart box communicator

Again, the **weighbridges** parameter is hard to describe in a table format, thus it is described in the following excerpt:

```
weighbridges:
    P141140200:
        - id: "7650332.0"
        address: 127.0.0.1
        port: 8888
        - id: "7650333.0"
        address: 127.0.0.1
        port: 8889
        - id: "7650339.0"
        address: 127.0.0.1
        port: 8890
        - id: "7650340.0"
        address: 127.0.0.1
        port: 8891
    P300200111:
        - id: "7750332.0"
        address: 127.0.0.1
        port: 8892
        - id: "7750332.0"
        address: 127.0.0.1
        port: 8893
        - id: "7750332.0"
        address: 127.0.0.1
        port: 8894
        - id: "7750332.0"
        address: 127.0.0.1
        port: 8895
```

The above configuration essentialy tells the smart box communicator that it has the capability to monitor two weighbridges, with serial numbers P141140200 and P300200111, which both have four load cells each. For each of the four load cells in each weighbridge, their serial number (**id**), the address and the port where they listen are provided. The address in this case is the local host, since the loadcells, running in *Docker*, also transmit their traffic to the local host's port specified in this configuration file.

After correctly configuring each of the stations with the adequate parameters, the only remaining task to perform is to copy the dataset generated in Subsection 5.1.2 to the appropriate directories, where the data will be read by the smart box and load cell communicators in order to build and transmit the weighing tickets.

As it may be recalled, in Subsection 5.1.2, 20 files were generated, one per weighbridge, totalling 4 and one per load cell, totalling 16 files. So, in this step, by using the tool *scp (Secure Copy)*, which essentially copies files from one host to the other through the *Secure Socket Shell (SSH)* protocol, the files were adequately put in their corresponding directories, which means that:

- The files from Station X's weighbridges, called *P191021852.json* and *P220120900.json*, were copied to the smart box communicator workspace, in *pi1*;
- The files from Station X's load cells, called *7450332.0.json*, *7450333.0.json*, *7450339.0.json*, *7450340.0.json*, *7550332.0.json*, *7550333.0.json*, *7550339.0.json*, *7550340.0.json*, were copied to the load cells communicators workspace, in *pi1*;
- The files from Station Y's weighbridges, called *P141140200.json* and *P300200111.json*, were copied to the smart box communicator workspace, in *pi2*;
- The files from Station Y's load cells, called *7650332.0.json*, *7650333.0.json*, *7650339.0.json*, *7650340.0.json*, *7750332.0.json*, *7750333.0.json*, *7750339.0.json*, *7750340.0.json*, were copied to the load cells communicators workspace, in *pi2*.

At this point, the communicators in both of the stations were executed, since they already had all the configurations and data required to transmit the weighing tickets. The "boot up" of the load cell communicators just required an action to start all the *Docker* containers. The load cells from station X (*pi1*) can be seen running in Figure 19 and the load cells from station Y (*pi2*) can be seen running in Figure 20.

```
nuno@nuno-pi1:~/Documents/poc$ docker-compose logs
Attaching to poc_w2_lc1_1, poc_w1_lc2_1, poc_w2_lc3_1, poc_w1_lc3_1, poc
w1_lc2_1  | 2020/11/12 17:47:22 Serving DTLS communication at [::]:8888
w1_lc3_1  | 2020/11/12 17:47:23 Serving DTLS communication at [::]:8888
w1_lc1_1  | 2020/11/12 17:47:22 Serving DTLS communication at [::]:8888
w2_lc1_1  | 2020/11/12 17:47:22 Serving DTLS communication at [::]:8888
w1_lc4_1  | 2020/11/12 17:47:23 Serving DTLS communication at [::]:8888
w2_lc2_1  | 2020/11/12 17:47:21 Serving DTLS communication at [::]:8888
w2_lc4_1  | 2020/11/12 17:47:22 Serving DTLS communication at [::]:8888
w2_lc3_1  | 2020/11/12 17:47:23 Serving DTLS communication at [::]:8888
```

Figure 19: Load cells from station X active

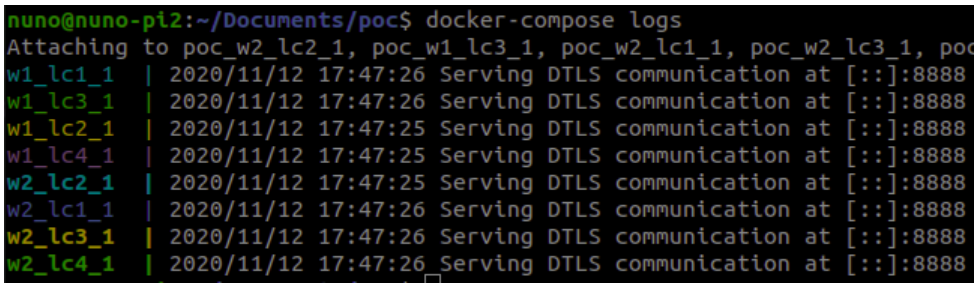


Figure 20: Load cells from station Y active

After the load cells are running in each station, the smart box communicators can also be executed in each of the stations. The execution of each of these communicators is the same in both the stations, since essentially, a script is run that:

1. Launches an instance of the communicator with the *–submit_pending* flag, which is the instance that will continuously look for pending tickets to transmit;

2. Continuously, launches an instance of the communicator that emits a weighing ticket from the first weighbridge, waits for three seconds, launches an instance of the communicator that emits a weighing ticket from the second weighbridge and waits for three seconds.

Since all the required software components are already running, the next step is to introduce some mechanism to demonstrate that the communication between them is secure and, in this case, a packet sniffer which can intercept packets that are transmitted in the network was used, namely *Wireshark* [55].

An instance of *Wireshark* was ran in each of the devices running the smart box communicators, *pi1* and *pi2*, listening in the loopback network, which means that it will listen in the localhost, which is where the smart box communicator communicates with the load cell communicators and another instance of *Wireshark* was ran in the laptop running the cloud system, listening in the wireless network *192.168.1.0/24*, where the smart box communicators communicate with the cloud system.

Finally, in order to be able to demonstrate if the fault-tolerance mechanism was working as requested, the Weighing Tickets API was stopped for approximately 30 seconds, in order to ensure that the smart box communicators could not transmit weighing tickets in that period of time, putting the fault-tolerance mechanism to test. After that period of time of approximately 30 seconds, the Weighing Tickets API was started again and the execution resumed until there were no more weighing tickets to transmit from the data, at which point, the communicators in *pi1* and *pi2* were stopped, leaving only the cloud system running in order to be able to perform queries to the Weighing Tickets API.

The result of the queries that were made, as well as significant aspects retrieved from the captures made by *Wireshark* and screenshots from the logs produced by the smart box

communicators when they were in fault-tolerance mode, are presented in the next section as the results from this proof of concept, being later discussed in terms of their correctness and validity.

5.2 RESULTS

The purpose of this section is to illustrate and provide the results obtained in this proof of concept both during and after its execution. During its execution, logs produced by the smart box communicators as well as by the weighing tickets API and screenshots from the packet captures performed in both networks are shown to facilitate the evaluation of the solution during communication. After the execution, queries to the weighing tickets API are made and their results are presented as to facilitate the evaluation of the data that the blockchain ledger holds, i.e., if the data it holds has the same characteristics of the original dataset, or not.

Following the previously defined order, the results are presented first during execution and then after the execution. In the first case, the results shown will be related with secure communication, evidence of ticket registration and finally the application of the fault-tolerance mechanism by the smart box comunicators.

5.2.1 *Weighing Ticket Building and Registration*

The first result obtained at the beggining of the weighing tickets communication process is, of course, the information that demonstrates the registration of weighing tickets, i.e, the logs indicating weighing ticket transmission and reception by the smart box communicators and the weighing tickets API, respectively. Figure 21 and Figure 22 show the logs produced on ticket transmission and recognition of registration in station X and Y, respectively. Figure 23 shows information produced on the weighing tickets API upon ticket reception and registration in the ledger.

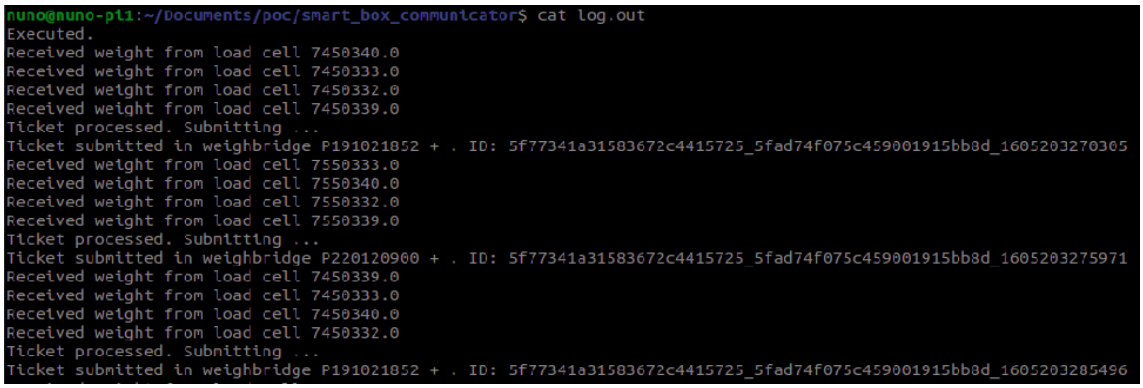


Figure 21: Logs indicating weighing ticket transmission and registration at Station X

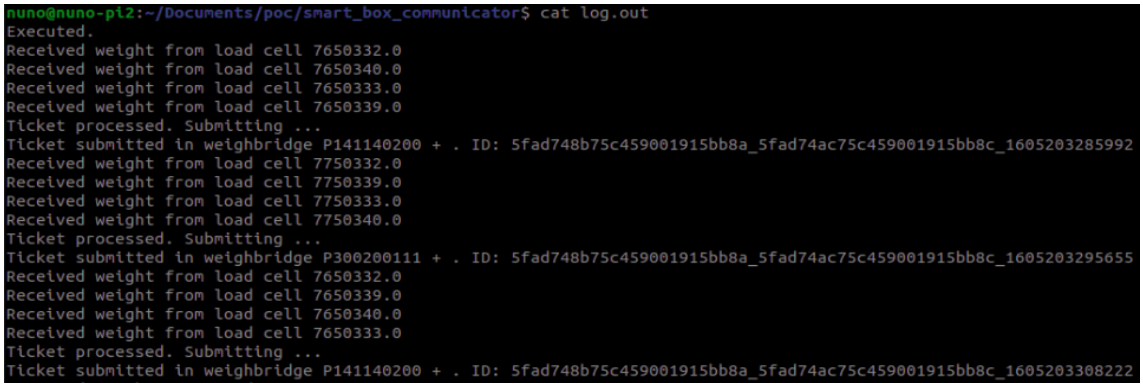


Figure 22: Logs indicating weighing ticket transmission and registration at Station Y

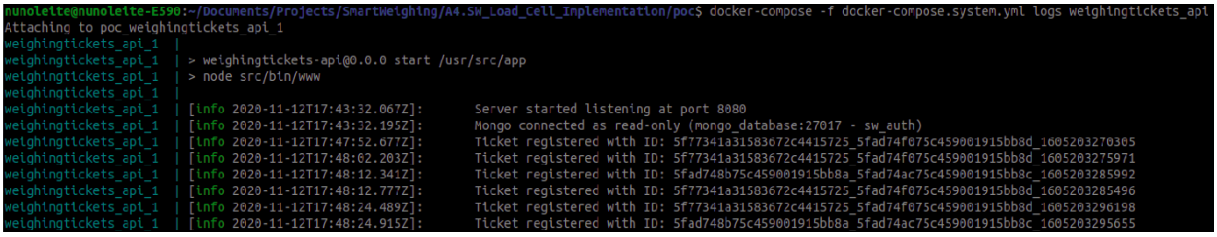


Figure 23: Logs indicating weighing ticket reception and registration at the Weighing Tickets API

Additionally, in the previous figures, it could be seen the statement that the smart box communicators are receiving weights from the load cells. Figure 24 illustrates the logs produced on the load cell communicators when receiving a weight request.

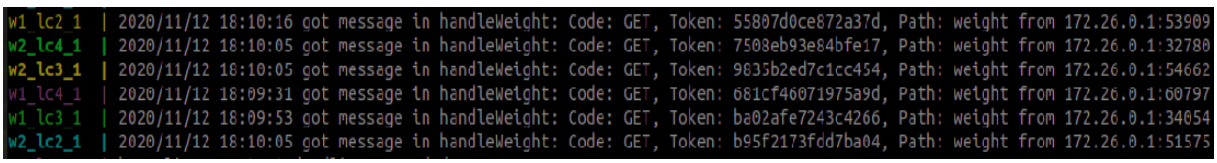


Figure 24: Logs indicating the reception of weight requests in the load cells

5.2.2 *Fault Tolerant Communication*

After providing the results produced on weighing ticket submission and, continuing in that line of thought, the results obtained when the smart box communicators entered the fault-tolerance mode are now presented. Figure 25 shows the logs produced when station X entered the fault-tolerance mode and as it can be seen, three attemps are made to submit a weighing ticket (value given in the smart box's communicator configurations) and, when those attempts are all spent, the ticket is put into a pending state.

Figure 25: Logs showing station X in fault-tolerance mode

Figure 26 shows the logs produced when station Y is in fault-tolerance mode, and despite having similar behavior to what was seen in Figure 25, it can also be seen that an attempt to submit an already pending ticket was performed, but it could not be transmitted.

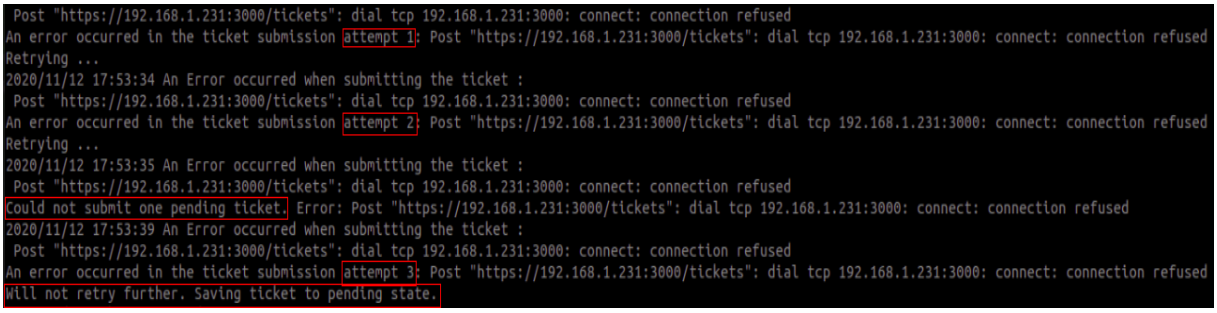


Figure 26: Logs showing station Y in fault-tolerance mode

After some time, the weighing tickets API is back up again and it is expected that the communicators are able to exit fault-tolerance mode and continue submitting weighing tickets, including the ones that are pending. Figure 27 and Figure 28 show the logs produced when the communicators resume communication in station X and Y, respectively, after being in fault-tolerance mode and, essentially, what can be perceived is that normal weighing ticket submission is happening again, as well as the submission of tickets that were in a pending state and now have to be transmitted.

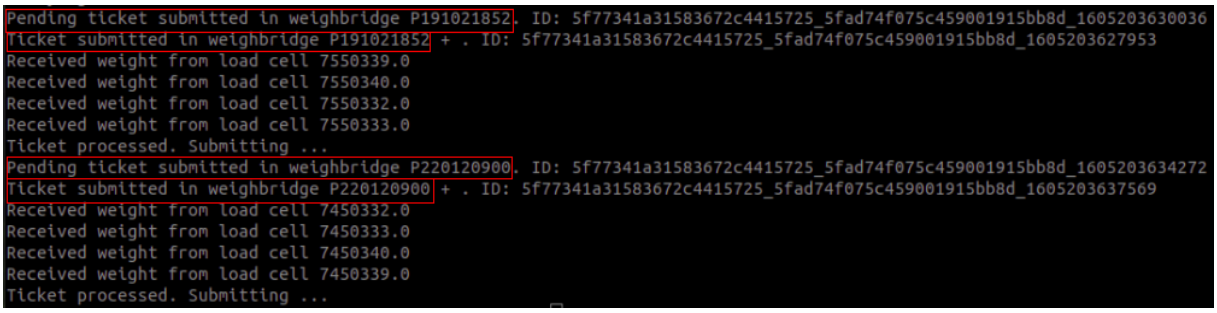


Figure 27: Resuming normal execution in station X with pending tickets submission

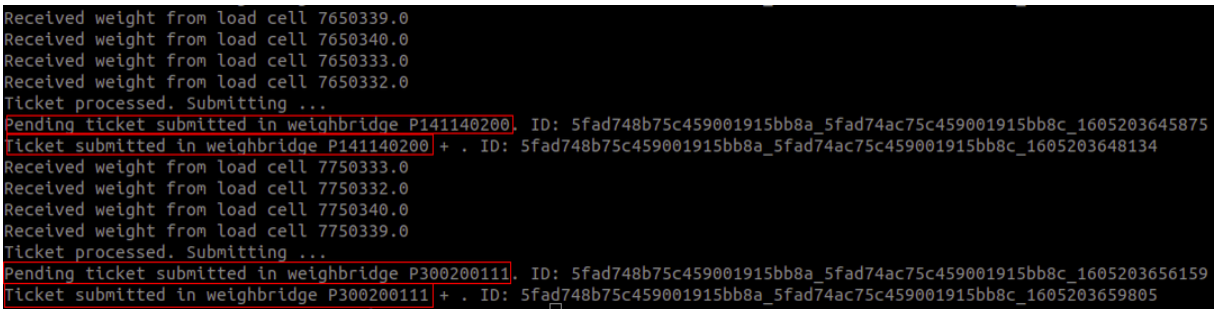


Figure 28: Resuming normal execution in station Y with pending tickets submission

Additionally, in order to provide clearer results of the fault-tolerance mechanism working, Figure 29 and Figure 30, show the contents of the JSON file which holds pending tickets prior entering fault-tolerance mode, when the communicators are in fault-tolerance mode and finally, when the communicators resume normal communication, with the goal to demonstrate that in the first stage there are no pending weighing tickets to submit, then in the second stage there is one pending ticket to submit in each station and, finally, after pending tickets are submitted in each station none is left.

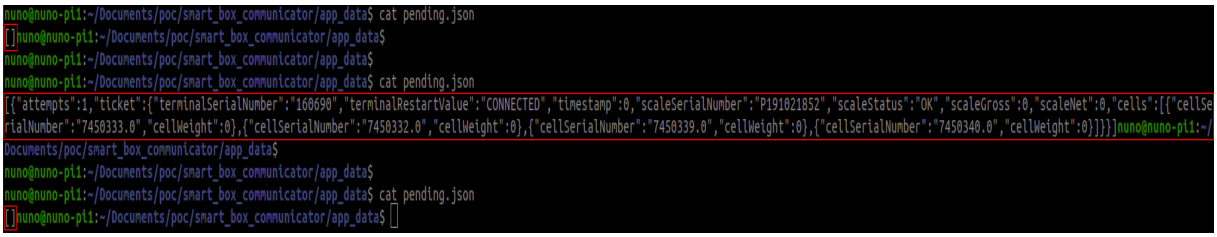


Figure 29: Evolution of pending tickets in station X

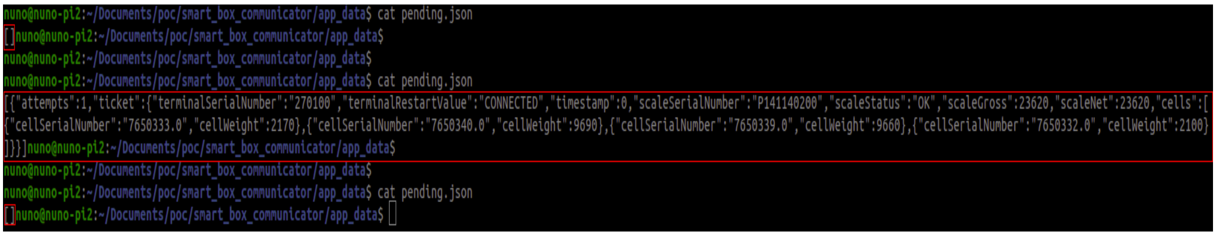
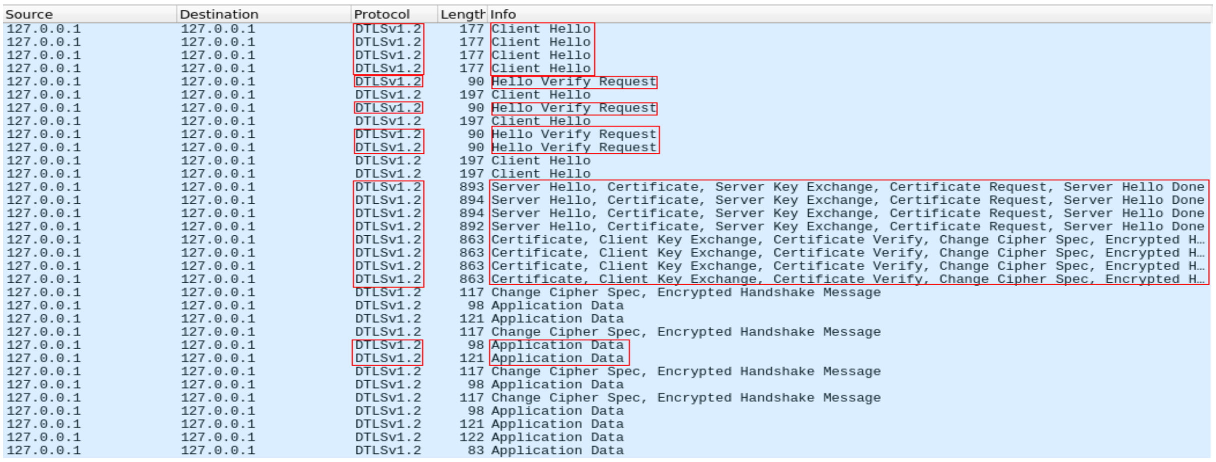


Figure 30: Evolution of pending tickets in station Y

5.2.3 *Secure Communication*

Having presented the results that demonstrate the registration of weighing tickets from either station, as well as the application of the fault-tolerance mechanism, there is one significant aspect of the communication that is left to cover, which is the secure communication, both between the smart box and load cell communicators, as well as between the smart box communicators and the cloud system APIs.

Figure 31 and Figure 32 illustrate a communication excerpt between the smart box communicator and the load cells in station X and station Y, respectively. In those figures, DTLS exchanges are highlighted, since the client sends its hello message, the certificates are verified, until the server terminates its hello, terminating with the exchange of application data. Other client hellos are also present since it has to be remembered that the smart box communicator communicates with up to eight load cells, four per weighbridge, in each station.



Source	Destination	Protocol	Length	Info
127.0.0.1	127.0.0.1	DTLSv1.2	177	Client Hello
127.0.0.1	127.0.0.1	DTLSv1.2	177	Client Hello
127.0.0.1	127.0.0.1	DTLSv1.2	177	Client Hello
127.0.0.1	127.0.0.1	DTLSv1.2	177	Client Hello
127.0.0.1	127.0.0.1	DTLSv1.2	90	Hello Verify Request
127.0.0.1	127.0.0.1	DTLSv1.2	197	Client Hello
127.0.0.1	127.0.0.1	DTLSv1.2	90	Hello Verify Request
127.0.0.1	127.0.0.1	DTLSv1.2	197	Client Hello
127.0.0.1	127.0.0.1	DTLSv1.2	90	Hello Verify Request
127.0.0.1	127.0.0.1	DTLSv1.2	90	Hello Verify Request
127.0.0.1	127.0.0.1	DTLSv1.2	197	Client Hello
127.0.0.1	127.0.0.1	DTLSv1.2	197	Client Hello
127.0.0.1	127.0.0.1	DTLSv1.2	893	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
127.0.0.1	127.0.0.1	DTLSv1.2	894	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
127.0.0.1	127.0.0.1	DTLSv1.2	894	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
127.0.0.1	127.0.0.1	DTLSv1.2	892	Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done
127.0.0.1	127.0.0.1	DTLSv1.2	863	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted H…
127.0.0.1	127.0.0.1	DTLSv1.2	863	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted H…
127.0.0.1	127.0.0.1	DTLSv1.2	863	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted H…
127.0.0.1	127.0.0.1	DTLSv1.2	863	Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted H…
127.0.0.1	127.0.0.1	DTLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
127.0.0.1	127.0.0.1	DTLSv1.2	98	Application Data
127.0.0.1	127.0.0.1	DTLSv1.2	121	Application Data
127.0.0.1	127.0.0.1	DTLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
127.0.0.1	127.0.0.1	DTLSv1.2	98	Application Data
127.0.0.1	127.0.0.1	DTLSv1.2	121	Application Data
127.0.0.1	127.0.0.1	DTLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
127.0.0.1	127.0.0.1	DTLSv1.2	98	Application Data
127.0.0.1	127.0.0.1	DTLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
127.0.0.1	127.0.0.1	DTLSv1.2	98	Application Data
127.0.0.1	127.0.0.1	DTLSv1.2	121	Application Data
127.0.0.1	127.0.0.1	DTLSv1.2	122	Application Data
127.0.0.1	127.0.0.1	DTLSv1.2	83	Application Data

Figure 31: Excerpt of communication in the loopback network of station X's *pi1*

Figure 32: Excerpt of communication in the loopback network of station Y's *pi2*

Although the two previous figures illustrate a DTLS data exchange process, some additional aspects can be further highlighted and thus, Figure 33 and Figure 34 show the actual contents of an application data packet, where the first is a packet being sent from the smart box communicator to a load cell in station X and the second is a packet being sent from a load cell to the smart box communicator in station Y. Additionally, the figures clearly highlight the fact that data exchange is encrypted, illustrated in the parameter *Encrypted Application Data* inside the *Datagram Transport Layer Security* section of the packet.

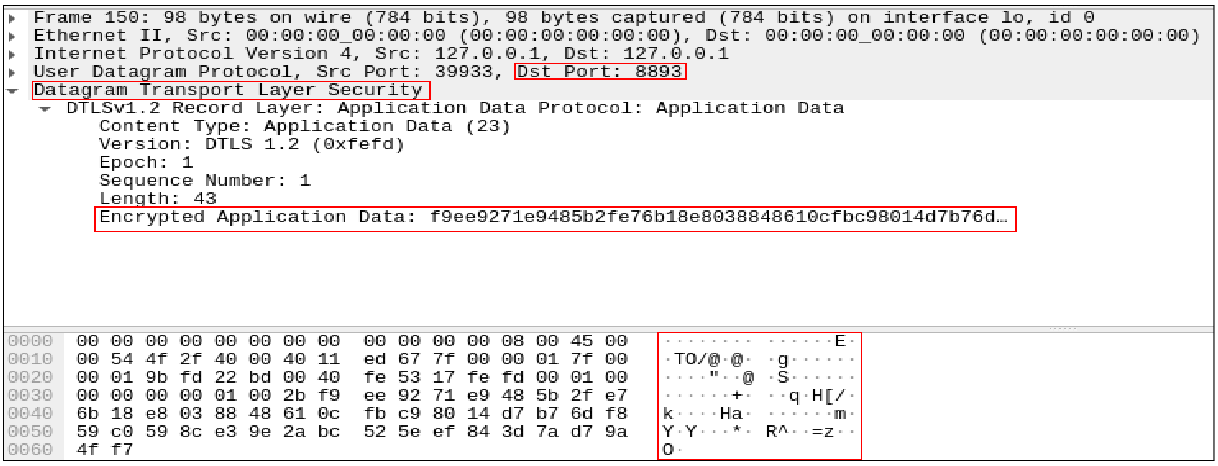


Figure 33: Sample packet captured in Station X's *pi1* loopback network

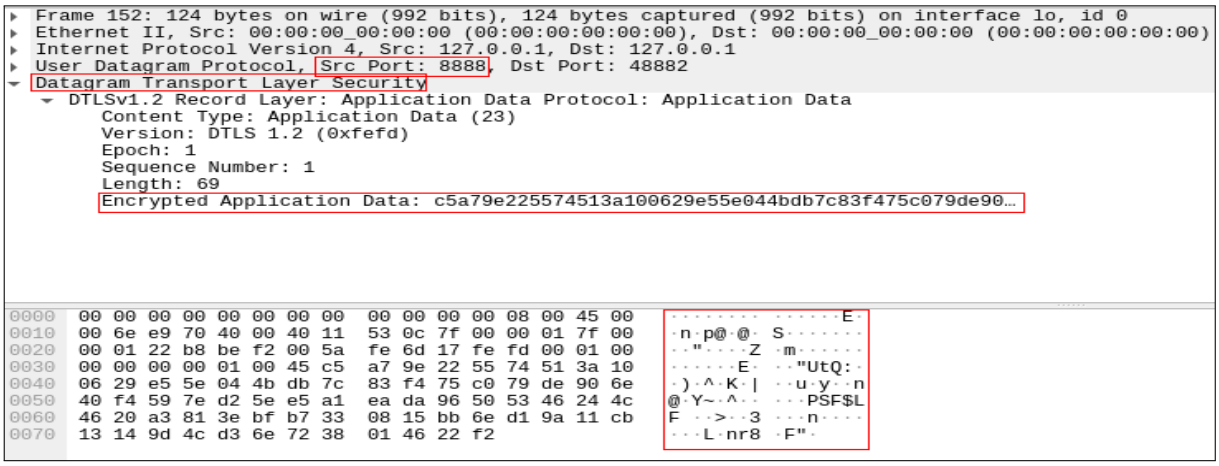


Figure 34: Sample packet captured in Station Y's *pi2* loopback network

Finally, to end the presentation of results for secure communication, Figure 35 illustrates TLS communication between station Y's *pi2* and the cloud system APIs in the first exchange, as well as between station X's *pi1* and the cloud system APIs in the second exchange. Additionally, Figure 36 illustrates an actual sample of application data packet, where *pi1* transmits a packet to the weighing tickets API, whose data is encrypted as demonstrated by the parameter *Encrypted Application Data* in the *Transport Layer Security* section.

Source	Destination	Protocol	Length	Info
192.168.1.202	192.168.1.231	TLSv1.3	309	Client Hello
192.168.1.231	192.168.1.202	TCP	66	3000 → 47662 [ACK] Seq=1 Ack=244 Win=65024 Len=0 TSval=634232001 TSecr=1176428418
192.168.1.231	192.168.1.202	TLSv1.3	1562	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Dat…
192.168.1.202	192.168.1.231	TCP	66	47662 → 3000 [ACK] Seq=244 Ack=1449 Win=64128 Len=0 TSval=1176428466 TSecr=634232013
192.168.1.202	192.168.1.231	TCP	66	47662 → 3000 [ACK] Seq=244 Ack=1497 Win=64128 Len=0 TSval=1176428466 TSecr=634232013
192.168.1.202	192.168.1.231	TLSv1.3	146	Change Cipher Spec, Application Data
192.168.1.202	192.168.1.231	TLSv1.3	1274	Application Data
192.168.1.202	192.168.1.231	TLSv1.3	379	Application Data
192.168.1.231	192.168.1.202	TCP	66	3000 → 47662 [ACK] Seq=1497 Ack=324 Win=65024 Len=0 TSval=634232080 TSecr=1176428515
192.168.1.231	192.168.1.202	TCP	66	3000 → 47662 [ACK] Seq=1497 Ack=1532 Win=63872 Len=0 TSval=634232080 TSecr=1176428517
192.168.1.231	192.168.1.202	TCP	66	3000 → 47662 [ACK] Seq=1497 Ack=1845 Win=63616 Len=0 TSval=634232080 TSecr=1176428517
192.168.1.231	192.168.1.202	TLSv1.3	608	Application Data, Application Data
192.168.1.202	192.168.1.231	TCP	66	47662 → 3000 [ACK] Seq=1845 Ack=2039 Win=64128 Len=0 TSval=1176428538 TSecr=634232088
192.168.1.202	192.168.1.231	SSH	110	Server: Encrypted packet (len=44)
192.168.1.231	192.168.1.202	TCP	66	33490 → 22 [ACK] Seq=1 Ack=617 Win=501 Len=0 TSval=1307161917 TSecr=1176428912
192.168.1.203	192.168.1.231	TCP	74	33742 → 3000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=3455386435 TSecr=…
192.168.1.231	192.168.1.203	TCP	74	3000 → 33742 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=158751…
192.168.1.203	192.168.1.231	TCP	66	33742 → 3000 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3455386453 TSecr=1587514030
192.168.1.203	192.168.1.231	TLSv1.3	309	Client Hello
192.168.1.231	192.168.1.203	TCP	66	3000 → 33742 [ACK] Seq=1 Ack=244 Win=65024 Len=0 TSval=1587514039 TSecr=3455386455
192.168.1.231	192.168.1.203	TLSv1.3	1561	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Dat…
192.168.1.203	192.168.1.231	TCP	66	33742 → 3000 [ACK] Seq=244 Ack=1449 Win=64128 Len=0 TSval=3455386469 TSecr=1587514048
192.168.1.203	192.168.1.231	TCP	66	33742 → 3000 [ACK] Seq=244 Ack=1496 Win=64128 Len=0 TSval=3455386469 TSecr=1587514048
192.168.1.203	192.168.1.231	TLSv1.3	146	Change Cipher Spec, Application Data
192.168.1.203	192.168.1.231	TLSv1.3	1274	Application Data
192.168.1.231	192.168.1.203	TCP	66	3000 → 33742 [ACK] Seq=1496 Ack=324 Win=65024 Len=0 TSval=1587514107 TSecr=3455386515
192.168.1.231	192.168.1.203	TCP	66	3000 → 33742 [ACK] Seq=1496 Ack=1532 Win=63872 Len=0 TSval=1587514107 TSecr=3455386516
192.168.1.203	192.168.1.231	TLSv1.3	375	Application Data
192.168.1.231	192.168.1.203	TCP	66	3000 → 33742 [ACK] Seq=1496 Ack=1841 Win=63616 Len=0 TSval=1587514107 TSecr=3455386516
192.168.1.231	192.168.1.203	TLSv1.3	608	Application Data, Application Data

Figure 35: Excerpt of communication in the wireless network 192.168.1.0/24

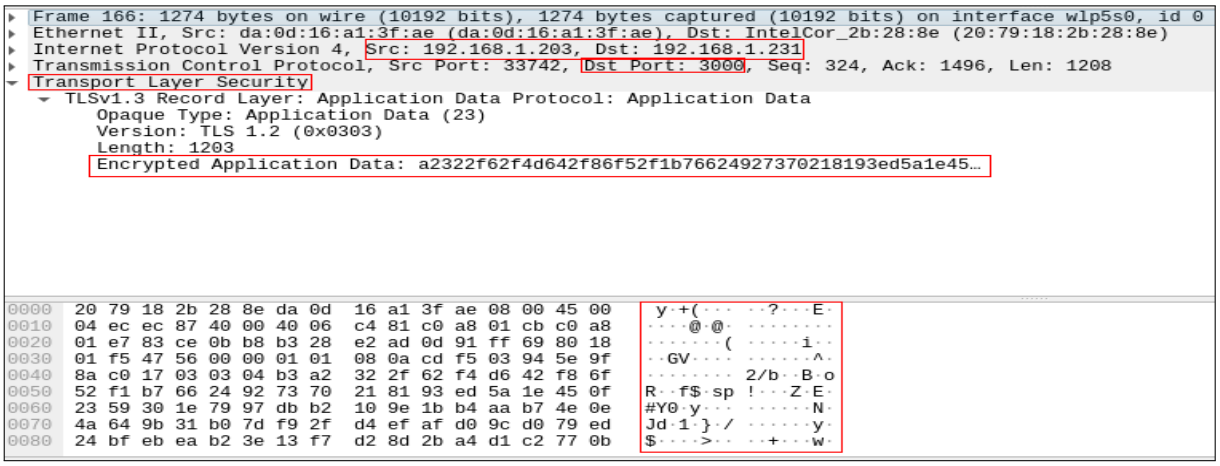


Figure 36: Sample packet captured in wireless network 192.168.1.0/24

5.2.4 *Querying and Applicational Logic Validation*

With the results from the secure communication point of view presented, the demonstration that belongs to the more communicational aspect of this proof of concept is concluded and what remains now is to present the results obtained after the execution, done by querying and understanding the characteristics of the data that is stored in the blockchain ledger as to perceive if there were no communication and applicational logic problems in the execution of the proof of concept and if the weighing tickets were all stored correctly, since it is already known that they were, at the very least, securely transmitted.

In summary, the results that are left to show should provide the capability to verify that all the weighing tickets from the original data were transmitted (600), that each of the stations transmitted 300 weighing tickets, and that there are 150 tickets per weighbridge. Additionally, the remaining results should also provide the ability to verify intrinsic characteristics of the data as established in Subsection 5.1.2, such as how many weighing tickets had a **scaleStatus** of *OK*, or how many tickets had a total weight between 50 KG and 1000 KG for example.

Prior starting the illustration of query results, it is worth mentioning that all requests were made using a tool called *Postman* [56], which allows the issuing of requests to REST APIs, receiving the responses in clear user interfaces, visually ideal to demonstrate results.

The first result to be presented is the amount of weighing tickets associated with each station since that can clarify if, at least, all weighing tickets from the different stations were correctly transmitted. Figure 37 and Figure 38 show the result of issuing two requests to the weighing tickets API, the first one to collect and count all weighing tickets belonging to station X and the second one to collect and count all weighing tickets belonging to station Y. Both requests were made to the URL *https://192.168.1.231:3000/tickets?count=true* with the sole difference that, in the first case an authorization token from customer X's administrator

was used and, in the second case an authorization token from customer Y's administrator was used. As it can be seen in the figures, both stations have 300 weighing tickets associated to them.

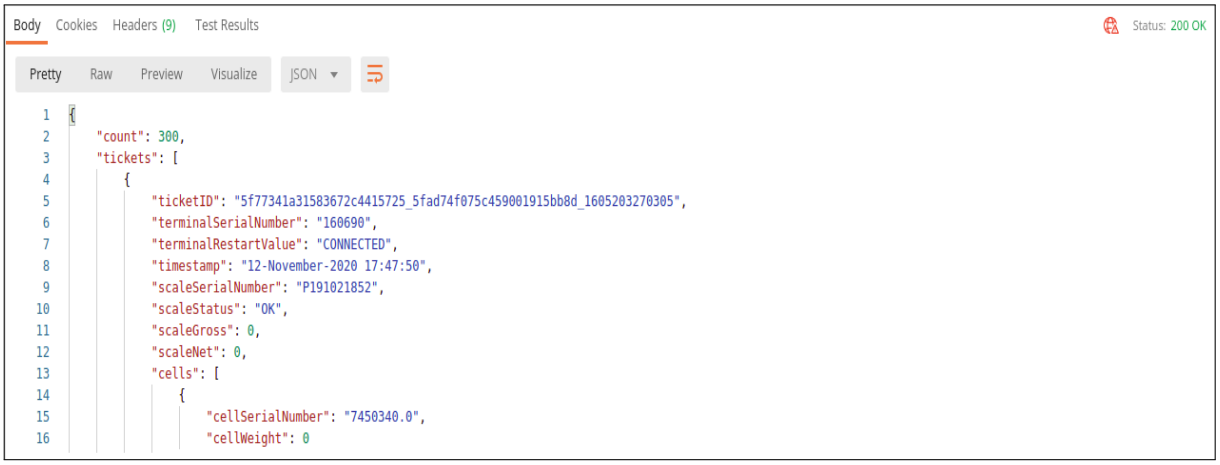


Figure 37: Total of weighing tickets associated with station X

Figure 38: Total of weighing tickets associated with station Y

The next step is to ensure that the weighing tickets per station are correctly distributed, i.e., that each weighbridge from each station has 150 weighing tickets associated with them.

Figure 39 illustrates the result obtained when issuing a request to the URL *https://192.168.1.231:3000/tickets?count=true&scale_serial_number=P191021852* with customer X's authorization token, which requests the API to collect and count all weighing tickets that are associated with weighbridge P191021852. As it can be seen this weighbridge has 150 weighing tickets associated to it.

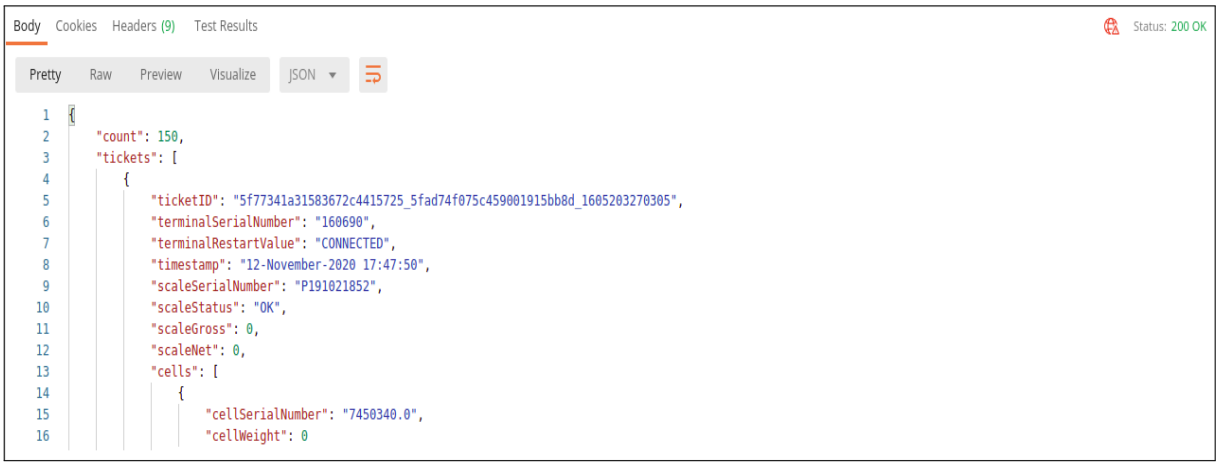


Figure 39: Total of weighing tickets associated with station X and its weighbridge P191021852

Since the remaining tests to perform in the 3 weighbridges that are left, are really similar to the one presented in figure 39, the remaining tests and respective results are provided in Appendix A, in Section A.1 (see figures 46, 47 and 48), which provide evidence that the global results show the correct amount of weighing tickets per weighbridge.

Having provided the results that show the correct amount of weighing tickets per station and per weighbridge, the next step is to demonstrate that the intrinsic characteristics of the original data are retained.

Beggining with the **scaleStatus** attribute, Figure 40 shows the response to a request to URL *https://192.168.1.231:3000?count=true&scale_serial_number=P191021852&scale_status=OK*, which collects and counts all weighing tickets associated with station X's weighbridge P191021852 which have a **scaleStatus** of OK.

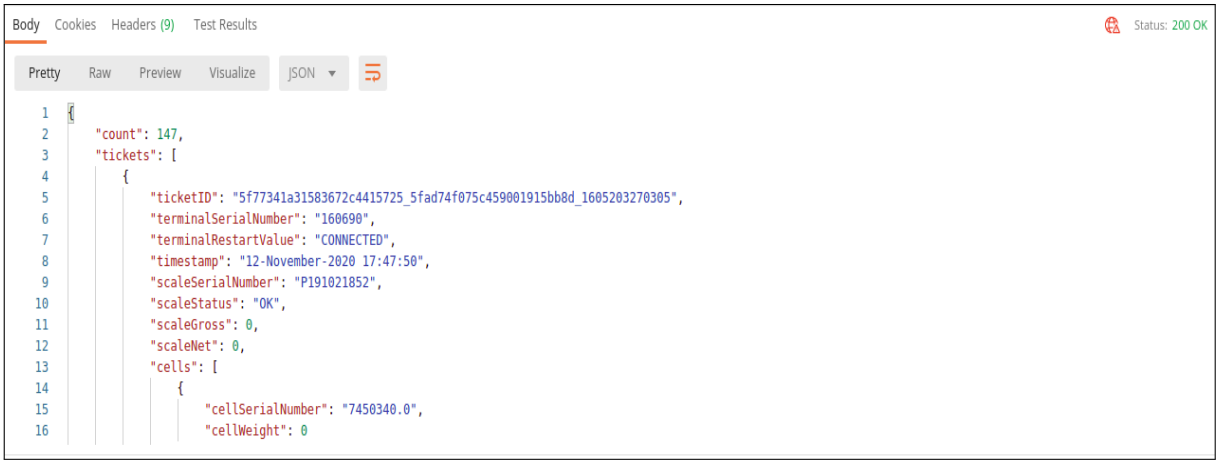


Figure 40: Total weighing tickets with status *OK* associated with station X and weighbridge P191021852

In Section A.2, located in Appendix A, an equal test was performed for the remaining 3 weighbridges as the one described for figure 40 (see figures 49, 50 and 51) and these global results show that the initial characteristics in terms of the weighing tickets' **scaleStatus** attribute remain unchanged.

The characteristic that is left to demonstrate is the total weight of the weighing tickets, i.e., showing the results that allow to ensure that the distribution of the total weight in the weighing tickets remain unchanged for values under 50 KG, from 50 KG until 1000 KG and from 1000 KG, for each weighbridge.

Figure 41 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P191021852&until_weight=50*, which collects and counts all weighing tickets associated with station X's weighbridge P191021852 which have a total weight under 50 KG.

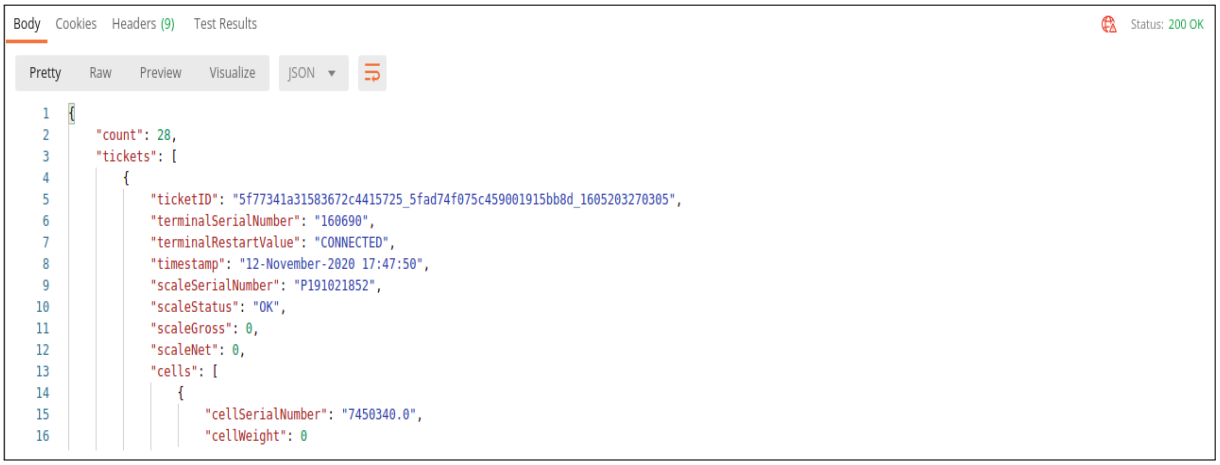


Figure 41: Total weighing tickets with a total weight until 50 KG (exclusive) associated with station X and weighbridge P191021852

Figure 42 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P191021852&from_weight=50 &until_weight=1000*, which collects and counts all weighing tickets associated with station X's weighbridge P191021852 which have a total weight equal or above 50 KG and under 1000 KG.

Figure 42: Total weighing tickets with a total weight between 50 KG (inclusive) and 1000 KG (exclusive) associated with station X and weighbridge P191021852

Figure 43 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P191021852&from_weight=1000*, which collects and counts all weighing tickets associated with station X's weighbridge P191021852 which have a total weight equal or above 1000 KG.

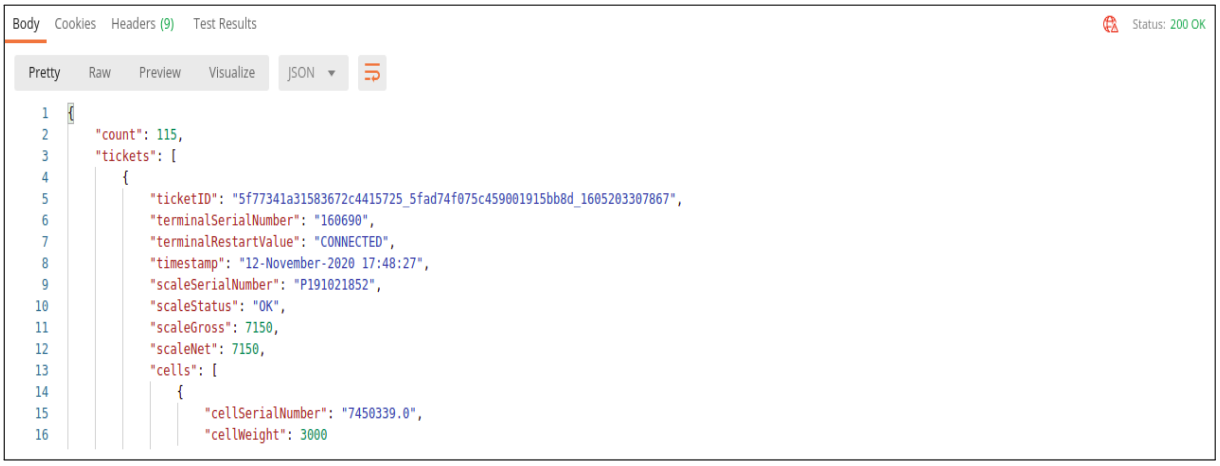


Figure 43: Total weighing tickets with a total weight from 1000 KG (inclusive) associated with station X and weighbridge P191021852

Similar tests to the ones presented in figures 41, 42 and 43 were performed for all remaining 3 weighbridges recognized in the system, which are shown in Appendix A, in Section A.3 (see figures 52 to 60). These set of tests and results allow to make evidence that the weight distribution per weighbridge in the aforementioned intervals is retained.

To conclude this section of illustration and demonstration of the results obtained in the execution of the proof of concept, an additional aspect is presented next, more related to the

guarantee of confidentiality between customers, i.e., the fact that a certain customer cannot access another customer's data.

Figure 44 illustrates the result obtained when an user possessing an access token associated with customer X, attempts to collect the tickets issued by station Y's weighbridge P141140200, by issuing a request to URL *https://192.168.1.231:3000/tickets?scale_serial_number=P141140200*. As it can be seen in the figure, nothing is returned, since the weighbridge is not associated with the authenticated user.

Figure 44: Result obtained when consulting customer Y's weighing tickets from an user belonging to customer X

Figure 45 illustrates the result obtained when an user possessing an access token associated with customer Y, attempts to collect the tickets issued by station X's weighbridge P220120900, by issuing a request to URL *https://192.168.1.231:3000/tickets?scale_serial_number=P220120900*. As it can be seen in the figure, nothing is returned, since the weighbridge is not associated with the authenticated user.

Figure 45: Result obtained when consulting customer X's weighing tickets from an user belonging to customer Y

These two last figures conclude the results demonstration process, since it has gone through the communication aspects of the solution, such as the ticket transmission and registration, the fault-tolerance mechanism and the security applied in the communication as well as the application logic correctness of the solution since it correctly stores all the weighing tickets transmitted and allows easy querying of them, while not permitting the access of one customer to another customer's weighing tickets.

In the next section, the results presented in this section are thoroughly discussed and compared to what should be obtained in order to evaluate the correctness and validity of the results that were shown.

5.3 Discussion

The purpose of this discussion is to evaluate the results presented in the previous section, comparing them with the objectives for this solution in order to clearly state whether the solution complies with the goals or not. Some goals are more related to the broad aspects of the solution, such as:

- The ability to establish a secure communication between devices for the buiding process and transmission of weighing tickets;
- The capability of the communication system to be fault tolerant, i.e., ensure one delivery per weighing ticket;
- The development of a cloud system capable of correctly receiving, storing and querying weighing tickets in the ledger.

Additionally, there are other goals that must be upheld, which relate to more specific aspects of the solution, such as:

- The retainment of all the characteristics of the data that is registered in the ledger, i.e., correctly registering the weighing tickets as they were created, without change to the quantity or the actual content of them;
- The security provided by the network structure, which must ensure that each customer only has access to its tickets.

In order to verify that the solution retains the characteristics of the weighing tickets that are sent, it is worth remembering the structure of the experiment given in Subsection 5.1.1 as well as the description provided on the dataset that is used for the proof of concept in Subsection 5.1.2 in order to compare it with the results obtained. Essentially, the experiment and the dataset can together be described as follows:

- Two customers exist. Customer X and Customer Y. Each of these customers possess one station, station X and station Y, respectively;
- Station X holds two weighbridges with four load cells each. The serial numbers of station X's weighbridges are P191021852 and P220120900;
- Station Y holds two weighbridges with four load cells each. The serial numbers of station Y's weighbridges are P141140200 and P300200111;
- The dataset used for the experiment contains 600 weighing tickets. 300 associated with weighbridges belonging to station X and 300 associated with weighbridges belonging to station Y;
- Each of the weighbridges is associated with exactly 150 weighing tickets;
- Station X's weighbridge P191021852 has 147 weighing tickets with a **scaleStatus** of *OK*, 3 with a **scaleStatus** of *FAULTY*;
- Station X's weighbridge P220120900 has 148 weighing tickets with a **scaleStatus** of *OK*, 2 with a **scaleStatus** of *FAULTY*;
- Station Y's weighbridge P141140200 has 150 weighing tickets with a **scaleStatus** of *OK*, 0 with a **scaleStatus** of *FAULTY*;
- Station Y's weighbridge P300200111 has 146 weighing tickets with a **scaleStatus** of *OK*, 4 with a *scaleStatus* of *FAULTY*;
- Station X's weighbridge P191021852 has 28 weighing tickets with a total weight until 50 KG (exclusive), 7 with a total weight from 50 KG (inclusive) to 1000 KG (exclusive) and 115 with a total weight from 1000 KG (inclusive);
- Station X's weighbridge P220120900 has 50 weighing tickets with a total weight until 50 KG (exclusive), 16 with a total weight from 50 KG (inclusive) to 1000 KG (exclusive) and 84 with a total weight from 1000 KG (inclusive);
- Station Y's weighbridge P141140200 has 15 weighing tickets with a total weight until 50 KG (exclusive), 1 with a total weight from 50 KG (inclusive) to 1000 KG (exclusive) and 134 with a total weight from 1000 KG (inclusive);
- Station Y's weighbridge P300200111 has 37 weighing tickets with a total weight until 50 KG (exclusive), 8 with a total weight from 50 KG (inclusive) to 1000 KG (exclusive) and 105 with a total weight from 1000 KG (inclusive).

Having clearly defined the goals and the characteristics to check for in the results, the discussion on the outcomes obtained by comparison with what should be the outcome can be initiated.

In a first instance, the solution had to be capable of building and trasmitting weighing tickets, securely, i.e., granting the confidentiality and authenticity of the weighing tickets it sends and, in the results provided there are figures that can actually demonstrate ticket building and secure transmission, namely figures 21 to 24 and figures 31 to 36, which show the transmission of weights from the load cells to the smart box communicators, the transmission of the weighing ticket from the smart box communicator and reception by the weighing tickets API and, finally, that the packets that comprise those transmissions all have their application data encrypted.

While it can be seen that the communicators that operate in the stations are able to correctly and securely transmit the weighing tickets, it has to be taken into account that it cannot be expected a 100% uptime of the weighing tickets API and, thus, a fault-tolerance mechanism had to be put in place and it has to be verified. Figures 25 to 30 demonstrate the correctness of the fault-tolerance mechanism, since in the first two figures, it can be seen that, in the moment the weighing tickets API does not respond, the fault-tolerance mechanism immediately kicks off, attempting to resubmit tickets and, when it cannot be done after a configurable number of retries, putting them into a pending state. Additionally, Figures 27 and 28 show the submission of weighing tickets that were in a pending state, which is the required and expected behaviour of the fault tolerance mechanism. Finally, figures 29 and 30 provide evidence of this mechanism's validity, since they show that, at first neither of the stations had pending weighing tickets to submit, then they both have at least one and, finally they have none again, since the pending tickets were already submitted, which definitively demonstrates that both the **RETRY** and **RECOVER** stages of the fault tolerance mechanism are working.

With the demonstration of the correctness of the communication system, what is left to demonstrate is the capability of the weighing tickets API to provide rich querying on the weighing tickets that are stored in the ledger and, of course, provide the evidence required to ensure that the application logic of both the communicators and the weighing tickets API are well developed and maintain the exact characteristics of the weighing tickets that pass through them. This demonstration of retainment of charachteristics can be done by essentially providing evidence that the weighing tickets stored in the ledger comply with the data characteristics defined above and in Subsection 5.1.2.

The first characteristic to prove is the actual amount of weighing tickets that were stored in the ledger, since it provides immediate clarity in the capability of the communicators to transmit all weighing tickets. Figures 37 and 38 demonstrate that the total count of weighing tickets for station X and Y, respectively is 300 for both, which totals 600. This demonstrates

that 600 weighing tickets were transmitted, and that of those 600, 300 were transmitted from each of the stations existent in the system. Additionally, figure 39 and figures 46 to 48 (Section A.1 in Appendix A) illustrate and demonstrate that each of the four weighbridges existent in the system (two from station X and two from station Y) are associated with 150 weighing tickets.

After demonstrating that the correct amount of weighing tickets per station and per weighbridge were transmitted, it is now required to ensure that the characteristics of the data were maintained, namely that the distribution of weighing tickets with a **scaleStatus** of *OK* and that the distribution of weighing tickets with a total weight under 50 KG, from 50 KG to 1000 KG and above or equal to 1000 KG, remain unchanged.

Figure 40 and figures 49 to 51 (Section A.2 in Appendix A) illustrate the amount of weighing tickets in each of the weighbridges that have a **scaleStatus** attribute value of *OK*. For this characteristic, it was expected that: i) station X's weighbridge P191021852 had 147 weighing tickets with a **scaleStatus** of *OK*, which is demonstrated in Figure 40; ii) station X's weighbridge P220120900 had 148 weighing tickets with a **scaleStatus** of *OK*, which is demonstrated in Figure 49; station Y's weighbridge P141140200 had 150 weighing tickets with a **scaleStatus** of *OK*, which is demonstrated in Figure 50; And, finally iv) that station Y's weighbridge P300200111 had 146 weighing tickets with a **scaleStatus** of *OK*, which is demonstrated in Figure 51.

In terms of dataset characteristics, only the distribution of total weighing in the weighing tickets is left to discuss. For this specific characteristic, it was expected that: i) station X's weighbridge P191021852 had 28 weighing tickets with a total weight under 50 KG (exclusive), 7 weighing tickets with a total weight from 50 KG (inclusive) to 1000 KG (exclusive) and 115 weighing tickets with a total weight from 1000 KG (inclusive), which can be seen demonstrated in figures 41, 42 and 43, respectively; ii) station X's weighbridge P220120900 had 50 weighing tickets with a total weight under 50 KG (exclusive), 16 weighing tickets with a total weight from 50 KG (inclusive) to 1000 KG (exclusive) and 84 weighing tickets with a total weight from 1000 KG (inclusive), which can be seen demonstrated in figures 52, 53 and 54 (Section A.3 in Appendix A), respectively; iii) station Y's weighbridge P141140200 had 15 weighing tickets with a total weight under 50 KG (exclusive), 1 weighing ticket with a total weight from 50 KG (inclusive) to 1000 KG (exclusive) and 134 weighing tickets with a total weight from 1000 KG (inclusive), which can be seen demonstrated in figures 55, 56 and 57 (Section A.3 in Appendix A), respectively; And, finally iv) station Y's weighbridge P300200111 had 37 weighing tickets with a total weight under 50 KG (exclusive), 8 weighing tickets with a total weight from 50 KG (inclusive) to 1000 KG (exclusive) and 105 weighing tickets with a total weight from 1000 KG (inclusive), which can be seen demonstrated in figures 58, 59 and 60 (Section A.3 in Appendix A), respectively.

In terms of the goals that were initially established, there is only one aspect left to demonstrate, which is the capability of the cloud system to correctly separate data of one customer from the other, i.e., ensuring weighing tickets' confidentiality between different customers. Figures 44 and 45 demonstrate this exact notion since, in the first figure, a request is made to the weighing tickets API with a customer X's authorization token, in which an attempt to consult weighing tickets from weighbridge P141140200, which belongs to customer Y, is made. What can be seen in that figure is that the amount of weighing tickets returned is 0, whilst it has already been seen that weighbridge P141140200 holds, in fact, 150 weighing tickets. The second figure, demonstrates the exact opposite, an attempt from an authorized customer Y user to consult weighing tickets from weighbridge P220120900, which belongs to costumer X, whose response is also 0 and no weighing tickets are returned. The weighing tickets API returns an empty array instead of, for example, an unauthorized error code because it is completely abstracted from weighbridge serial numbers. For example, in the first case, while the user associated with customer X placed a serial number of a customer Y's weighbridge, what happens is that customer X's stations do not possess weighing tickets associated with that weighbridge in the smart contract and, thus no tickets are returned.

This last demonstration concludes the discussion on the results obtained in this proof of concept, since all the results have been studied and discussed, which comply with the goals that were defined.

5.4 SUMMARY

This chapter covered the preparation, execution and discussion of results of the proof of concept that had the intent to demonstrate the capability of the proposed solution to comply with the goals defined in the beggining of this dissertation.

The initial section of this chapter showed and explained the experiment architecture, i.e., the way how the proof of concept was assembled, specifically, which devices would be used and which components would be used in each of the devices. Additionally, the structure of the blockchain network used was also clarified, indicating the number of nodes that were launched and to what entity they were associated with. After discussing the experiment architecture, the preparation of the dataset that was used in the proof of concept was showed, by describing the original set of weighings received, how they were transformed to serve as basis for the experiment and how they were structured. Finally, the section concludes with a clear description on the characteristics of the data after being completely transformed, in order to facilitate later evaluation of the results obtained. With the dataset prepared for execution, the chapter continues with the actual execution of the proof of concept, by showing the characteristics of the devices, such as their IP address, by demonstrating how the components were configured to run , both in the cloud system as well as in the

communication system and, finally, by showing the step-by-step execution of the proof of concept, with an explanation per step on why it was included in the process.

In the second section of the chapter, all the results obtained from the proof of concept were showed, in a complete illustration on all the functionalities provided by the solution, such as the correct building and secure transmission of the weighing tickets, the application of the fault tolerance mechanism, the adequate storage of weighing tickets in the blockchain ledger and the rich querying system provided by the weighing tickets API.

Finally, the chapter ends (excluding this section) with a discussion on the correctness and validity of the results obtained in the previous section. Essentially, a comparison is made between the results that were expected and the ones that were obtained in order to assess if the solution and the experiment were successful.

6

CONCLUSION

This chapter concludes this dissertation, by overviewing the work done in a small summary of the document and, finally, by indicating the prospect for possible future work that might arise.

6.1 SUMMARY

The work performed along the writing of this dissertation can be essentially categorized in three main areas, as previously stated in Chapter 1: Introduction:

1. **Research**, by having reviewed literature and the state of the art for the technologies and concepts associated with the problem in study;
2. **Development**, by having proposed a clearly defined solution that might mitigate problems & challenges offered either by the technologies or the problem themselves. Additionally, it also includes the development stage, where the proposed solution is actually implemented into tangible software components;
3. **Evaluation**, by having built an experiment capable of evaluating the solution that was built in terms of its compliability with the initially proposed goals.

The first point presented, **Research**, is comprised by Chapter 1: Introduction and Chapter 2: State of the Art. In the first chapter, problem assessment and definition is made by providing the context and motivation for the creation of the project and by defining the objectives for the project, which can be used as a measure of success of this dissertation. The second chapter, with the context provided by the first one, explores the technologies that are relevant to the creation of a solution capable of complying with the proposed goals, researching blockchain and IoT technologies and their integration to a reasonable extent. First the chapter explores IoT technologies, by reviewing and describing some of its most common applications as well as the challenges that might arise from the development of solutions using such technologies. After this initial contextualization on IoT, state of the art communication protocols and how

they can be secured were explored, namely by overviewing their characteristics in terms of its communication type, pattern and paradigm as well as the underlying transport protocol.

Having discussed and explored IoT technologies, the chapter proceeds with the study of blockchain technology, by presenting essential constructs and concepts of blockchain, in order to provide insights on why that technology is suitable for the use case in study, namely how it can ensure the weighing tickets' immutability after they are put in the ledger by the smart contract. Additionally, existent blockchain platforms were presented and summarily described, such as *Ethereum* [23], *Quorum* [31], *Hyperledger Fabric* [29] and *Corda* [30], allowing the establishment and definition of each platform's unique characteristics and how they approach blockchain-based applications.

Finally, the chapter ends with a small study in the integration of both tecnologies (blockchain & IoT). First, it presents some possible improvements that the utilization of blockchain brings to IoT applications, depending on the way how the technologies are integrated, ranging from all-inclusive approaches where all data generated is sent through the blockchain application, to less inclusive applications where only a select set of data is actually put in the ledger. In the end, some real use cases of the integration of these technologies are also presented with the goal to highlight the growing utilization of them.

The second point described, **Development** is essentially comprised by Chapter 3: Problem, Challenges and Proposed Solution and Chapter 4: Development. The third chapter leans more to the design and architecture part of the solution, since it explores the possible problems and challenges that might arise and, finally, presents a solution capable of, in some way, mitigating those challenges while, of course, complying with the established goals. Furthermore, in this chapter, the technological choice for the components that are part of the cloud system was also presented, indicating: i) the blockchain platform that was going to be used, which was *Quorum* and why; ii) the database engine that was used; iii) the programming language and framework used to develop the APIs; And, finally, iv) the protocol definition for the secure communication between the devices that operate in the weighing stations and the cloud system.

The fourth chapter, as its name suggests, explores the development of the solution that was proposed, by exploring prior-implementation decisions that had to be made and the actual implementation of each of the software components defined in the solution. The first task was definitely to decide a number of aspects that directly impacted the development of the solution, such as:

- The software components to be built, by analyzing the proposed solution and clearly defining what components had to be built in order to comply with that solution;
- The structure of the blockchain network, i.e., how all its members were going to be organized in order to promote data privacy;

- The requirements of the smart contract, which essentially defined what the smart contract application had to be able to do to fully provide all the required functionality;
- How the complexity introduced by a blockchain-based application could be abstracted from the end user;
- How the entities that participate in the system would authenticate themselves and, furthermore, how they would authorize the requests that they made to the cloud system APIs.

After clearly describing the decisions made for each of the aforementioned aspects, the chapter goes on to describe the implementation process of each of the software components that were built, namely:

- The authentication & authorization process, which explains how entities can authenticate themselves in the cloud system and how they can be authorized when issuing further requests;
- The *Smart Contract*, which possesses all the required functionality to manage weighing ticket assets in the blockchain;
- The *Data Models*, which model the data relevant to the cloud system such as the entities that participate in it and their attributes;
- The *Cloud System APIs*, where the implementation of the *Authentication & Management API* and the *Weighing Tickets API* is shown, keeping in mind that while the first API serves the purpose of enabling simple entity management and authentication, the second API simplifies the management and manipulation of weighing tickets for each entity;
- The *Smart Box Communicator*, which implements mechanisms to securely communicate weighing tickets from the weighing stations to the blockchain, in a fault-tolerant manner;
- The *Load Cell Communicator*, which implements a simple process to securely transmit the weight it measures to the requesting *Smart Box Communicator*.

The presentation and description of the implementation process of each of the software components ends the chapter, providing a clear insight on how the solution was built.

The third point presented, **Evaluation** can be directly associated to Chapter 5: Proof of Concept, where an experiment is assembled to demonstrate the ability of the solution of complying with the proposed objectives, beggining with an overview of the experiment, such as its architecture, the dataset that was used and how the actual experiment was executed,

then the results extracted from that execution are shown and, finally, a discussion on the validity and correctness of those results is done. It is worth mentioning that the goal of this proof of concept was to demonstrate, functionality-wise, the capabilities of the developed solution, such as its ability to securely transmit weighing tickets or the ability to facilitate the querying of weighing tickets already registered, through the query system of the *Weighing Tickets* API.

The main takeaway that can be noted with the conclusion of the proof of concept's chapter is that the solution that was built was capable of complying with all the proposed functional goals for this dissertation.

6.2 FUTURE WORK

With the dissertation concluded, an analysis has to be made on the prospect for future work that will be associated with this project. In functional and conceptual terms, the solution presented here complies with all the requirements laid out in the beggining of the project, however there will certainly be aspects more related to integration and optimization that will still occur.

These aspects can be simply described in four essential points:

- The integration of the cloud system in a solution capable of fitting current systems owned by the company, such as for example, replacing the authentication component with a possible system the company already uses;
- The integration of the station communicators in real smart boxes and load cells, with the capability of reading the weight measured by the sensors and building tickets from that live data;
- The optimization of the load cell communicator, since this program will be run in highly restrained devices, which might oblige a review of the structure and technology used in the implementation of the program, but not its functionality and applicational logic;
- Finally, the testing of the solution in a real-world scenario, instead of a simulated one.

These four last points define and summarize the tasks that this project may face from here on out, with the assurance that, functionaly, it has already reached a good maturity level.

BIBLIOGRAPHY

[1] WELMEC. European Cooperation in Legal Metrology. https://www.welmec.org, 2001. [Online; Accessed 17-October-2020].

[2] Rolf H. Weber. Internet of Things - New security and privacy challenges. *Computer Law and Security Review*, 26(1):23–30, 2010. ISSN 02673649. doi: 10.1016/j.clsr.2009.11.008. URL http://dx.doi.org/10.1016/j.clsr.2009.11.008.

[3] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys and Tutorials*, 17(4):2347–2376, 2015. ISSN 1553877X. doi: 10.1109/COMST.2015.2444095.

[4] Debasis Bandyopadhyay and Jaydip Sen. Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications*, 58(1):49–69, 2011. ISSN 09296212. doi: 10.1007/s11277-011-0288-5.

[5] Ioannis Andrea, Chrysostomos Chrysostomou, and George Hadjichristofi. Internet of Things: Security vulnerabilities and challenges. *Proceedings - IEEE Symposium on Computers and Communications*, 2016-Febru(July):180–187, 2016. ISSN 15301346. doi: 10.1109/ISCC.2015.7405513.

[6] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on Industrial Informatics*, 10(4):2233–2243, 2014. ISSN 15513203. doi: 10.1109/TII.2014.2300753.

[7] Ibrar Yaqoob, Ejaz Ahmed, Ibrahim Abaker Targio Hashem, Abdelmuttlib Ibrahim Abdalla Ahmed, Abdullah Gani, Muhammad Imran, and Mohsen Guizani. Internet of Things Architecture: Recent Advances, Taxonomy, Requirements, and Open Challenges. *IEEE Wireless Communications*, 24(3):10–16, 2017. ISSN 15361284. doi: 10.1109/MWC.2017.1600421.

[8] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini. Security, privacy and trust in Internet of things: The road ahead. *Computer Networks*, 76:146–164, 2015. ISSN 13891286. doi: 10.1016/j.comnet.2014.11.008. URL http://dx.doi.org/10.1016/j.comnet.2014.11.008.

[9] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010. ISSN 13891286. doi: 10.1016/j.comnet.2010.05.010. URL http://dx.doi.org/10.1016/j.comnet.2010.05.010.

[10] Samer Jaloudi. Communication protocols of an industrial internet of things environment: A comparative study. *Future Internet*, 11(3), 2019. ISSN 19995903. doi: 10.3390/fi11030066.

[11] Internet Engineering Task Force. Hypertext Transfer Protocol version 2. https://tools.ietf.org/html/rfc7540, 2015. [Online; Accessed 5-January-2015].

[12] Internet Engineering Task Force. The Constrained Application Protocol. https://tools.ietf.org/html/rfc7252, 2019. [Online; Accessed 5-January-2019].

[13] OASIS. MQTT - Message Queuing Telemetry Transport. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, 2019. [Online; Accessed 5-January-2019].

[14] Internet Engineering Task Force. Tls - transport layer security. https://tools.ietf.org/html/rfc8446, 2018. [Online; Accessed 21-October-2020].

[15] Internet Engineering Task Force. Dtls - datagram transport layer security. https://tools.ietf.org/html/rfc6347, 2012. [Online; Accessed 21-October-2020].

[16] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. *Journal for General Philosophy of Science*, 39(1), 2008. ISSN 09254560. doi: 10.1007/s10838-008-9062-0.

[17] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. *Proceedings - 2017 IEEE 6th International Congress on Big Data, BigData Congress 2017*, pages 557–564, 2017. doi: 10.1109/BigDataCongress.2017.85.

[18] MLS Dev. Blockchain architecture basics: Components, structure, benefits & creation. https://medium.com/@MLSDevCom/blockchain-architecture-basics-components-structure-benefits-creation-beace17c8e77, 2019. [Online; Accessed 27-December-2019].

[19] Damien Cosset. What is in a block ? https://dev.to/damcosset/blockchain-what-is-in-a-block-48jo, 2017. [Online; Accessed 27-December-2019].

[20] Ralph C. Merkle. Protocols for public key cryptosystems. *Proceedings - IEEE Symposium on Security and Privacy*, (April 1980):122–134, 2012. ISSN 10816011. doi: 10.1109/SP.1980.10006.

[21] Ralph C Merkle. Comments in 2012 about the 1979 paper: A Certified Digital Signature A CERTIFIED DIGITAL SIGNATURE. 2012. URL http://www.merkle.com/papers/Certified1979.pdf.

[22] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. ISSN 15584593. doi: 10.1145/357172.357176.

[23] Ethereum. Proof of Stake - Ethereum. https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake/, 2019. [Online; Accessed 28-December-2019].

[24] Elli Androulaki, Artem Barger, Vita Bortnikov, Srinivasan Muralidharan, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Chet Murthy, Christopher Ferris, Gennady Laventman, Yacov Manevich, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, 2018-Janua, 2018. doi: 10.1145/3190508.3190538.

[25] The Linux Foundation. The hyperledger project. https://www.hyperledger.org/. [Online; Accessed 28-December-2019].

[26] Nick Szabo. The Idea of Smart Contracts. https://nakamotoinstitute.org/the-idea-of-smart-contracts/, 1990. [Online; Accessed 28-December-2019].

[27] Ethereum. Developer Resources. https://ethereum.org/developers/#getting-started, 2019. [Online; Accessed 3-January-2019].

[28] Hollander, Luit. The Ethereum Virtual Machine - How does it work? https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e, 2019. [Online; Accessed 3-January-2019].

[29] Hyperledger Fabric. Smart contracts and chaincode. https://hyperledger-fabric.readthedocs.io/en/release-1.4/smartcontract/smartcontract.html, 2019. [Online; Accessed 3-January-2019].

[30] R3 Foundation. Corda Technical White Paper. https://www.r3.com/reports/corda-technical-whitepaper/, 2019. [Online; Accessed 27-June-2020].

[31] Quorum. Quorum - The proven blockchain for business. https://www.goquorum.com/, 2015. [Online; Accessed 8-June-2020].

[32] Quorum. Quorum - Documentation. http://docs.goquorum.com/en/latest/, 2019. [Online; Accessed 8-June-2020].

[33] Ana Reyna, Cristian Martín, Jaime Chen, Enrique Soler, and Manuel Díaz. On blockchain and its integration with IoT. Challenges and opportunities. *Future Generation Computer Systems*, 88(2018):173–190, 2018. ISSN 0167739X. doi: 10.1016/j.future.2018.05.046.

[34] Abderahman Rejeb, John G. Keogh, and Horst Treiblmaier. Leveraging the Internet of Things and blockchain technology in Supply Chain Management. *Future Internet*, 11(7): 1–22, 2019. ISSN 19995903. doi: 10.3390/fi11070161.

[35] Arshdeep Bahga and Vijay K. Madisetti. Blockchain Platform for Industrial Internet of Things. *Journal of Software Engineering and Applications*, 09(10):533–546, 2016. ISSN 1945-3116. doi: 10.4236/jsea.2016.910036.

[36] F. Callegati, W. Cerroni, and M. Ramilli. Man-in-the-middle attack to the https protocol. *IEEE Security Privacy*, 7(1):78–81, 2009.

[37] Spring. Production-grade Spring applications. https://spring.io/projects/spring-boot, 2014. [Online; Accessed 27-June-2020].

[38] Pallets. Flask - Web development one drop at a time. https://flask.palletsprojects.com/en/1.1.x/, 2010. [Online; Accessed 27-June-2020].

[39] Open JS Foundation. Node.JS - a javascript runtime. https://nodejs.org/en/, 2012. [Online; Accessed 27-June-2020].

[40] Ethereum Foundation. Solidity Types. https://solidity.readthedocs.io/en/v0.5.3/types.html, 2020. [Online; Accessed 30-September-2020].

[41] IETF - Internet Engineering Task Force. JSON Web Token (JWT). https://tools.ietf.org/html/rfc7519, 2015. [Online; Accessed 30-September-2020].

[42] autho. jsonwebtoken - npm. https://www.npmjs.com/package/jsonwebtoken, 2018. [Online; Accessed 13-October-2020. Version 8.5.1].

[43] Mongo DB, Inc. MongoDB - The database for modern applications. https://www.mongodb.com/, 2009. [Online; Accessed 27-June-2020].

[44] Automattic. Mongoose - mongodb object modeling for node.js. https://github.com/Automattic/mongoose, 2018. [Online; Accessed 14-October-2020]. Version 5.10.9.

[45] Google. Golang. https://golang.org/, 2020. [Online; Accessed 21-October-2020. Version 1.15].

[46] Google. Golang- net http package. https://golang.org/pkg/net/http/, 2020. [Online; Accessed 21-October-2020].

[47] Google. Golang- crypto tls package. https://golang.org/pkg/crypto/tls/, 2020. [Online; Accessed 21-October-2020].

[48] Pion. Pion dtls - a go implementation of dtls. https://github.com/pion/dtls, 2020. [Online; Accessed 21-October-2020].

[49] The Hybrid Group. Gobot - Golang powered robotics. https://gobot.io/, 2019. [Online; Accessed 21-October-2020.].

[50] embeddedgo. Embedded GO - GO for microcontrollers. https://github.com/embeddedgo, 2020. [Online; Accessed 21-October-2020.].

[51] TinyGO Org. TinyGO - A GO compiler for small places. https://tinygo.org/, 2020. [Online; Accessed 21-October-2020.].

[52] Docker Inc. Docker. https://www.docker.com/, 2020. [Online; Accessed 13-November-2020].

[53] Raspberry PI Foundation. Raspberry pi. https://www.raspberrypi.org/, 2020. [Online; Accessed 13-November-2020].

[54] OpenSSL Software Foundation. Openssl - cryptography and ssl/tls toolkit. https://www.openssl.org/, 1999. [Online; Accessed 17-November-2020].

[55] The Wireshark Foundation. Wireshark. https://www.wireshark.org/, 2020. [Online; Accessed 18-November-2020].

[56] Inc. Postman. Postman - the collaboration platform for api development. https://www.postman.com/, 2020. [Online; Accessed 18-November-2020].

A

QUERYING RESULTS

A.1 COUNT PER WEIGHBRIDGE

Figure 46 illustrates the result provided when issuing a request to the URL *https://192.168.1.231:3000/tickets?count=true&scale_serial_number=P220120900* with customer X's authorization token, which requests the API to collect and count all weighing tickets that are associated with weighbridge P220120900. As it can be seen this weighbridge has 150 weighing tickets associated to it.

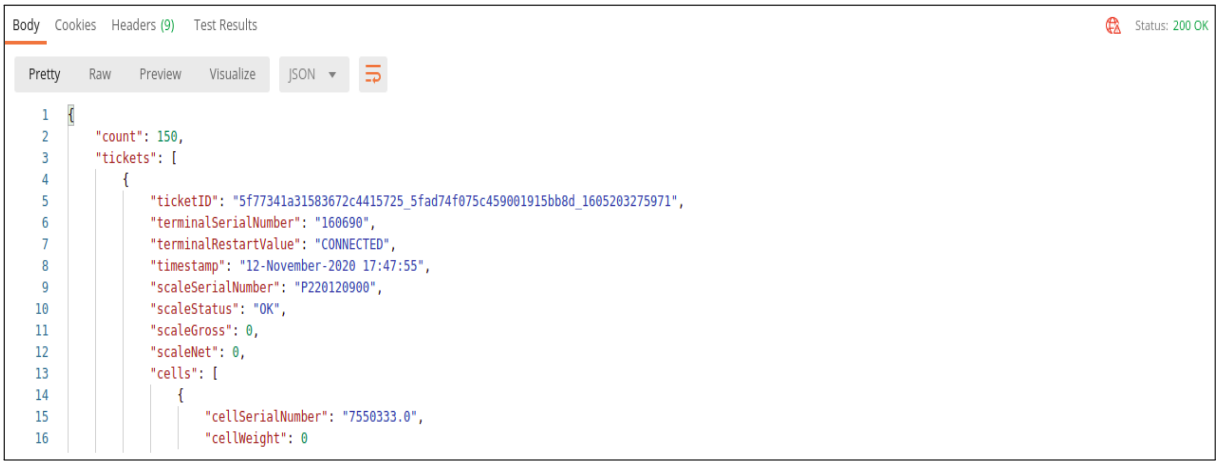


Figure 46: Total of weighing tickets associated with station X and its weighbridge P220120900

Figure 47 illustrates the result provided when issuing a request to the URL *https://192.168.1.231:3000/tickets?count=true&scale_serial_number=P141140200* with customer Y's authorization token, which requests the API to collect and count all weighing tickets that are associated with weighbridge P141140200. As it can be seen this weighbridge has 150 weighing tickets associated to it.

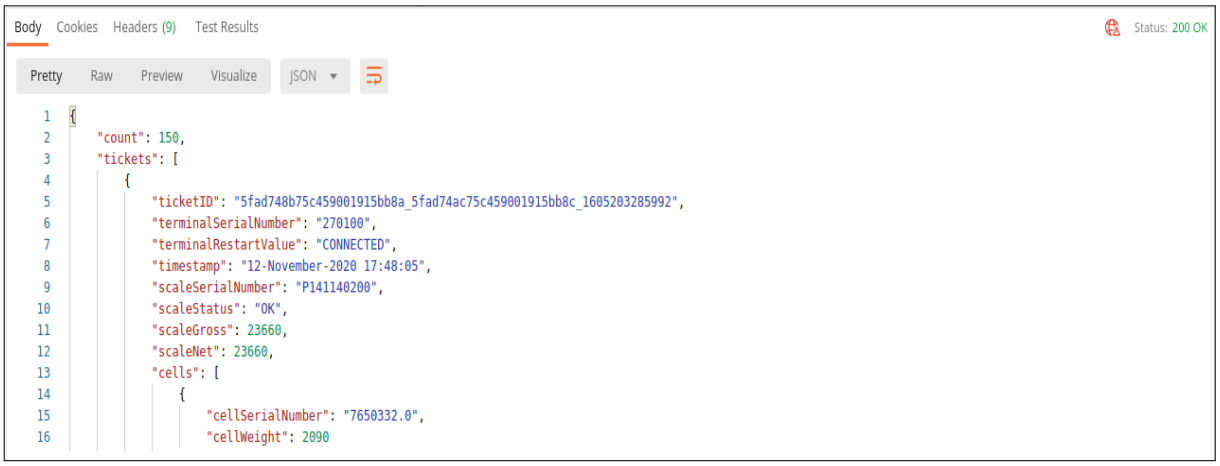


Figure 47: Total of weighing tickets associated with station Y and its weighbridge P141140200

Figure 48 illustrates the result provided when issuing a request to the URL *https://192.168.1.231:3000/tickets?count=true&scale_serial_number=P300200111* with customer Y's authorization token, which requests the API to collect and count all weighing tickets that are associated with weighbridge P300200111. As it can be seen this weighbridge has 150 weighing tickets associated to it.

Figure 48: Total of weighing tickets associated with station Y and its weighbridge P300200111

A.2 SCALE STATUS PER WEIGHBRIDGE

Figure 49 shows the response to a request to URL *https://192.168.1.231:3000?count=true&scale_serial_number=P220120900&scale_status=OK*, which collects and counts all weighing tickets associated with station X's weighbridge P220120900 which have a **scaleStatus** of OK.

Figure 49: Total weighing tickets with status *OK* associated with station X and weighbridge P220120900

Figure 50 shows the response to a request to URL *https://192.168.1.231:3000?count=true&scale_serial_number=P141140200&scale_status=OK*, which collects and counts all weighing tickets associated with station Y's weighbridge P141140200 which have a **scaleStatus** of OK.

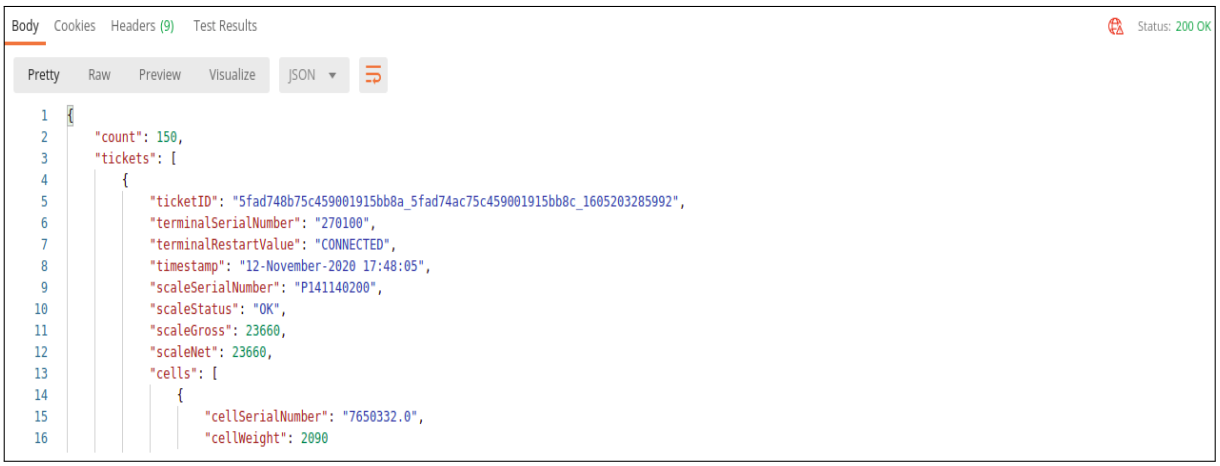


Figure 50: Total weighing tickets with status *OK* associated with station Y and weighbridge P141140200

Figure 51 shows the response to a request to URL *https://192.168.1.231:3000?count=true&scale_serial_number=P300200111&scale_status=OK*, which collects and counts all weighing tickets associated with station Y's weighbridge P300200111 which have a **scaleStatus** of OK.

Figure 51: Total weighing tickets with status *OK* associated with station Y and weighbridge P300200111

A.3 WEIGHT DISTRIBUTION PER WEIGHBRIDGE

Figure 52 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P220120900&until_weight=50*, which collects and counts all weighing tickets associated with station X's weighbridge P220120900 which have a total weight under 50 KG.

Figure 52: Total weighing tickets with a total weight until 50 KG (exclusive) associated with station X and weighbridge P220120900

Figure 53 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P220120900&from_weight=50&until_weight=1000*, which collects and counts all weighing tickets associated with station

X's weighbridge P220120900 which have a total weight equal or above 50 KG and under 1000 KG.

Figure 53: Total weighing tickets with a total weight between 50 KG (inclusive) and 1000 KG (exclusive) associated with station X and weighbridge P220120900

Figure 54 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P220120900&from_weight=1000*, which collects and counts all weighing tickets associated with station X's weighbridge P220120900 which have a total weight equal or above 1000 KG.

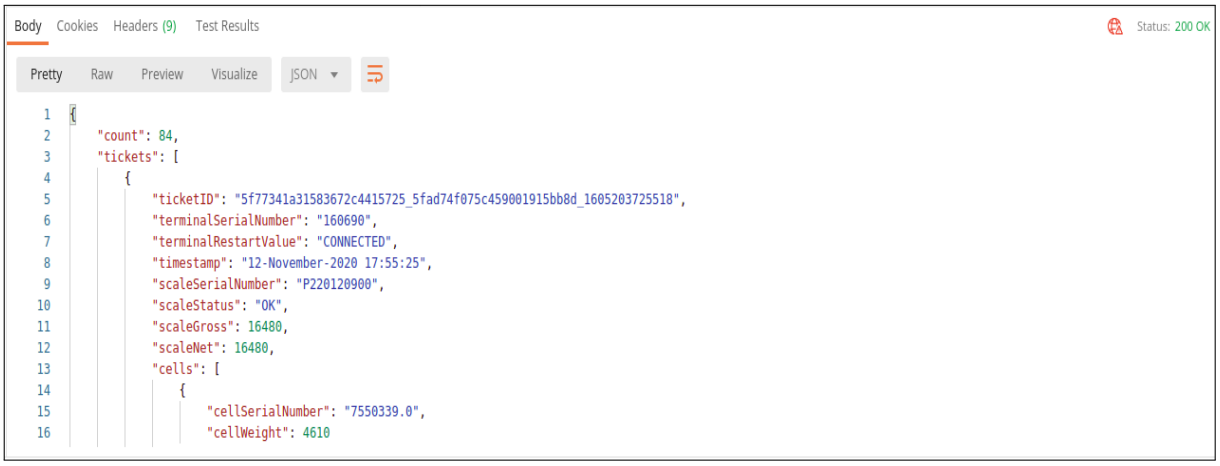


Figure 54: Total weighing tickets with a total weight from 1000 KG (inclusive) ssociated with station X and weighbridge P220120900

Figure 55 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P141140200&until_weight=50*, which collects and counts all weighing tickets associated with station Y's weighbridge P141140200 which have a total weight under 50 KG.

Figure 55: Total weighing tickets with a total weight until 50 KG (exclusive) associated with station Y and weighbridge P141140200

Figure 56 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P141140200&from_weight=50 &until_weight=1000*, which collects and counts all weighing tickets associated with station Y's weighbridge P141140200 which have a total weight equal or above 50 KG and under 1000 KG.

Figure 56: Total weighing tickets with a total weight between 50 KG (inclusive) and 1000 KG (exclusive) associated with station Y and weighbridge P141140200

Figure 57 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P141140200&from_weight=1000*, which collects and counts all weighing tickets associated with station Y's weighbridge P141140200 which have a total weight equal or above 1000 KG.

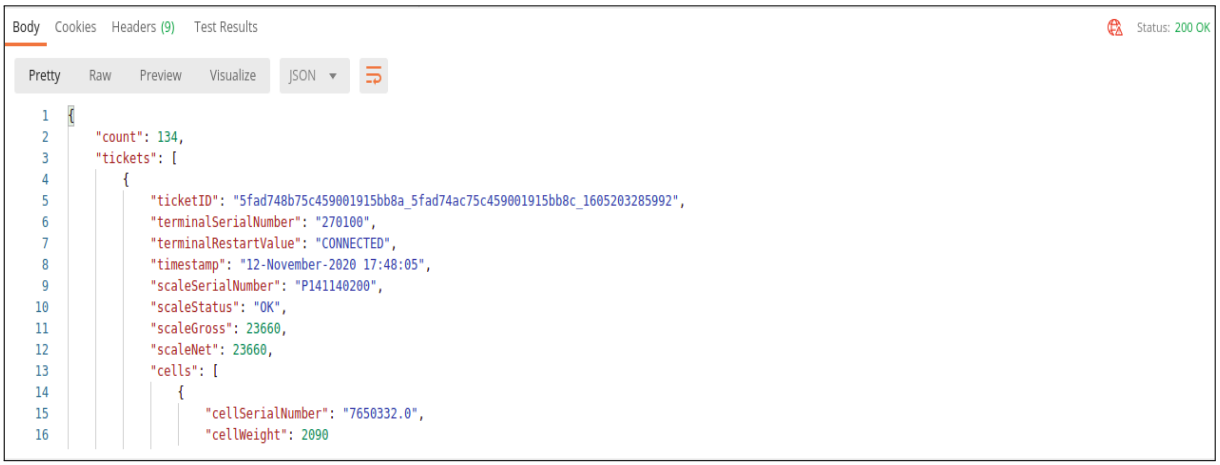


Figure 57: Total weighing tickets with a total weight from 1000 KG (inclusive) associated with station Y and weighbridge P141140200

Figure 58 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P300200111&until_weight=50*, which collects and counts all weighing tickets associated with station Y's weighbridge P300200111 which have a total weight under 50 KG.

Figure 58: Total weighing tickets with a total weight until 50 KG (exclusive) associated with station Y and weighbridge P300200111

Figure 59 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P300200111&from_weight=50 &until_weight=1000*, which collects and counts all weighing tickets associated with station Y's weighbridge P300200111 which have a total weight equal or above 50 KG and under 1000 KG.

Figure 59: Total weighing tickets with a total weight between 50 KG (inclusive) and 1000 KG (exclusive) associated with station Y and weighbridge P300200111

Figure 60 shows the result of performing a request to the URL *https://192.168.1.231:3000?count=true&scale_serial_number=P300200111&from_weight=1000*, which collects and counts all weighing tickets associated with station Y's weighbridge P300200111 which have a total weight equal or above 1000 KG.

Figure 60: Total weighing tickets with a total weight from 1000 KG (inclusive) associated with station Y and weighbridge P300200111