Enhancing data security in the Internet of Things with blockchain

Anastasios Pateritsas

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
TECHNICAL UNIVERSITY OF CRETE

A thesis submitted in fulfillment of the requirements for the degree of Diploma in Electrical and Computer Engineering.

Supervisor: Euripides G.M. Petrakis, Professor
Samoladas Vasileios, Associate Professor
Ioannidis Sotirios, Associate Professor
October 2022, Chania

Abstract

The Web of Things (WoT) initiative aims at unifying the world of interconnected devices over the Internet. With the significant increase of IoT, there is a great need for easy, fast, and secure access to data that can be exploited and offer useful information to users with the responsibility or authority to handle this data. However, current blockchain-backed IoT systems use the blockchain to store access control policies for sensor data stored in a database. In this Thesis, we propose iBot, a blockchain architecture that validates and verify the identity of IoT devices, data, and applications. The difference with current blockchain-backed IoT systems is that the iBot creates a safe environment to develop applications based on data produced by IoT devices. We use blockchain technology to protect the data of the devices, users, and services from unauthorized access. We use Decentralized Identifiers (DID) and Verifiable Credentials (VC) that W3C proposes for User/IoT Authorization & Authentication to achieve this. The architecture encompasses different functionalities to serve a 3-tier architecture model, each serving different functionality for different types of users (Infrastructure Owners – System Administrators, Application Developers, Customers). It supports storage for subscriptions to sensors and their data for real-time updates, and with the help of the blockchain, there is a reliable recording of the use of the system by each user. iBot uses the Hyperledger Fabric framework as the blockchain back-end and also extends the idea that devices have OpenAPI descriptions, adding to the description and the smart contract definition to protect the device's data. The experiments show that the architecture can cope with a large workload in real time without losing information.

Contents

1 Introduction

1.1 Background and Motivation

The need to create "Internet of Things" (IoT) applications to control the operation of an ecosystem (such as a city, a factory, or a house) or to monitor geo-environmental phenomena is perceived in many areas of everyday life. In the health sector, where people with compromised health need to check parameters such as humidity or pollen level of an area before visiting it. In a smart city, the information on the levels of temperature, noise pollution, traffic on the streets, exhaust gases, etc., gives a helpful picture of the living conditions in the broader area. From the above assumptions and the significant increase of IoT, there is a need for easy, fast, and secure access to data that can be exploited and offer helpful information to users with the responsibility or authority to handle this data. There is a need to provide users with validated information securely.

Blockchain technology has shown to be a promising solution in many distributed applications where trust is a critical factor. As a result, the research community and the industry can research and implement solutions combining IoT and blockchain. Blockchain technology can provide a more advanced security environment [5], [6] than central database security. In blockchain technology, each block in the chain constantly tracks the list of records linked to previous blocks using the cryptographic hash function. It is also a distributed logbook that records transactions and can prevent distortions. The majority of these proposals [3],[8],[9],[10] follow a hybrid approach where a storage system (e.g., a cloud provider) hosts the data itself, and blockchain offers services to ensure, e.g., trust distribution and integrity. So these proposals use blockchain as a "middleware" that checks for unauthorized or malicious actions.

1.2 Problem Definition

There are many approaches to designing IoT systems either using blockchain or the traditional way. Approaches as the iXen[1] and iSWoT thesis, which support Semantic Web functionality in a Cloud Computing environment. iSWoT is an architecture for the automated connection of devices in the Web of Things[1] and data processing in a semantic framework. iXen is an approach to facilitate the creation of applications based on the devices and their data sent to the system. Accordingly, technologies such as Sash[3] or IoTeX[2] use blockchain technology to ensure the access and security of the data transmitted by Things. No architecture so far combines both elements. It is important that, the architecture should provide data security. Also, it should provide an approach to create applications based on the devices and their data easily. The blockchain technology, smart contracts, Verifiable Credentials[3], Decentralized Identifiers[4], and OpenAPI[5] specification are combined technologies that contribute to achieving one of the above goals. With blockchain and smart contracts, we achieve device data security and validity(data generated from reliable sources). At the same time, Verifiable Credentials and Decentralized Identifiers are used to offer protection of the identities and credentials of the system's actors so that proper authorization for users to view or handle IoT information is granted. Finally, using OpenAPI to describe the devices achieves a general way in which system users will interact with the devices.

1.3 Proposed Solution

iBot is an Internet of Things (IoT) platform in cloud computing and blockchain. The platform can manage an extensive collection of sensors of various models placed in different locations serving different applications or user needs. The set of sensors can produce a vast range of measurements (data) sent to the blockchain. Using the blockchain, we manage to safely store such a large volume of data without worrying about any losses, as long as the data is stored distributed without a single point of failure. Like other

1 https://www.w3.org/WoT/
2 https://iotex.io/
3 https://www.w3.org/TR/vc-data-model/
4 https://www.w3.org/TR/did-core/
5 https://swagger.io/specification/

systems [3],[10], smart contracts, as part of the blockchain, can offer a series of "rules" that must satisfy the data sent to the system. With this addition, our system becomes safe from incorrect measurements or malicious actions from the devices that are connected to the system.

Interestingly, while the system consists of many sensors of different specifications, our architecture treats all the devices as if they were services described by OpenAPI. Devices become part of the Web so that they can be discovered and reused by different types of users. OpenAPI provides a method for documenting RESTful services so that a user or another service can understand their purpose and reuse them in applications. Along with creating a proxy in the cloud, we can have a virtual image of each Thing as [2] proposed.

We use the DID to declare the existence of an entity(user/IoT/application) in the system for authentication and VC to declare the entity type (user, developer, IoT,..) for authorization. These Credentials are stored in the user locally and not in a database that has gathered all the user's data. Also, the credentials contain the DIDs authorized to access (update/delete) the credentials. The Decentralized Identifier and Verifiable Credentials are W3C proposal. Decentralized identifiers (DIDs) are new identifier that enables verifiable, decentralized digital identity. Verifiable credentials (VCs) are an open standard for digital credentials. They can represent information found in physical credentials, such as a passport or license, and new things with no physical equivalent, such as bank account ownership. A traditional identification method is not used in this work, where the users and their details are stored and exposed in a database. The blockchain allowed us to use the Decentralized Identifier(DID) and Verifiable Credentials(VC), with the result that the user is the owner of his credentials. Using these Credentials, he can only connect to the system and gain access to services he is authorized to do based on his role in the system.

Finally, it is worth noting that the system's actors are considered to be all the participants, such as ordinary users, device owners, programmers, and the devices themselves with the applications that use them. A business model can be based on this as long as the blockchain records all the use of transactions between users and devices or between applications and devices.

1.4 Contribution Work

iBot aims not to compete with any commercial application but to show that it is possible, using existing technological solutions and open-source tools, and build a user and data-safe environment (platform) that facilitates the creation of applications.

The main contributions of this work are the following:

- A key feature of the iBot is the Service Oriented Architecture (see section 2.1). Each system function is an independent service that communicates with the others through RESTful interfaces.
- The architecture design follows the "secure by design" tactic, ensuring the protection of system services in the computing cloud. This way, the REST interfaces of the services are only accessible by the system and authorized users. To achieve this, iBot proposes a decentralized approach for identification and authorization for system users, devices, or applications interacting with the system and sensors. The decentralized approach uses the Decentralized Identifiers and Verifiable Credentials proposed by the W3C.
- It uses an implementation of a Web Thing Proxy [2] service that exposes Things on the Web and implements all the model's operations on Things using HTTP[6].
- It proposes the usage of blockchain as a more secure data storage method of the characteristics that record all the operations that take place around the data and the users. Additionally, as a distributed system, blockchain provides high data availability. iBot stores almost all data (device data, user identifier, subscriptions, transactions, device descriptions) on the blockchain, so we have multiple instances of these data in different nodes (no single point of failure).
- With smart contracts, we automated operations such as the check for a device property's value whet it sends to blockchain.
- It demonstrates how OpenAPI can be applied to describe Things and their functionality without ambiguities. But it can also be extended and describe which smart contract is the one that controls the information sent by a Thing.
- The architecture supports subscriptions to sensor devices that produce data. By creating subscriptions to sensors, applications can be developed that use real-time data from the devices.

[6] https://developer.mozilla.org/en-US/docs/Web/HTTP

- It demonstrates how blockchain can easily integrate a business model by assigning different roles with rights to each user of the system according to their status. Along with recording each user's use of the system, there may be some charge or payment.

1.5 Thesis Outline

Chapter 2 presents the technologies used in the work, but also related work that unites Blockchain with IoT Devices. Section 3 analyzes the system design requirements and sets out the functional and non-functional specifications with UML diagrams. In addition, the system's architecture is presented, and its components are analyzed. Chapter 4 presents the technical solution of the architecture proposed in chapter 3. Explains how we manage blockchain technology for storing IoT devices and their information, but also how the OpenAPI descriptions of the devices helped us in this work. Chapter 5 presents experiments performed on the system based on realistic conditions. The experiments were performed to see how the system reacts in a realistic environment with many devices. Finally, in chapter 6, we discuss the conclusions and issues for future work.

2. Background and Related Work

2.1 Service-Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is a service-based architecture model where a service is a well-defined and self-contained functionality. A service does not need to know the technical details of another service it interacts with. This facilitation of communication is achieved through the implementation of a tightly defined interface, which can perform the necessary actions to allow data transmission between services. In Service Oriented Architecture, an application is built by assembling small, self-contained functional units. Therefore, developers can reuse services in multiple applications regardless of their interactions with other services. Since a service is an independent entity, it can easily be updated or maintained without considering other services. Finally, Multiple instances of a single service can run concurrently on different servers. As a result, it increases the scalability and availability of the service.

2.1.1 REST-Based services

The Representational State Transfer (REST) architectural standard, leveraging its significant properties (performance, scalability, and modifiability), offers Web Services the ability to function optimally on the World Wide Web. Data and functionality are considered resources based on the REST standard, and access to them is performed through a strictly defined familiar interface based on standard HTTP methods. In particular, each resource is accessible through a Uniform Resource Identifier (URI), i.e., a hyperlink on the Internet. Also, each resource consists of identity and a data type and supports a set of actions. The identity of the resource is the URI as mentioned above, and the standard HTTP methods implement the actions: retrieve/read (GET), create (POST), update (PUT), and delete (DELETE). The GET operation retrieves the current state of a resource (using some representation), the POST operation creates a new resource, and the PUT operation updates the current state of the resource. Finally, the DELETE function deletes a specific resource. To indicate the success or failure of the operation, the HTTP status code is used, where each code has its interpretation.

2.2 RabbitMQ

With tens of thousands of users, RabbitMQ[7] is one of the most popular open-source message brokers. From T-Mobile to Runtastic, RabbitMQ is used worldwide at small startups and large enterprises.

It originally implemented the Advanced Message Queuing Protocol (AMQP: an application layer protocol specification for asynchronous messaging) and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol (STOMP), MQ Telemetry Transport (MQTT), and other protocols.

[7] https://www.rabbitmq.com/

Written in Erlang, the RabbitMQ server is built on the Open Telecom Platform framework for clustering and failover. Client libraries to interface with the broker are available for all major programming languages. The source code is released under the Mozilla Public License.

Basic concepts of RabbitMQ:

- Producer: A producer is the one who sends messages to a queue based on the queue name.
- Queue: A Queue is a sequential data structure that is a medium through which messages are transferred and stored.
- Consumer: A consumer is the one who subscribes to and receives messages from the broker and uses that message for other defined operations.
- Exchange: An exchange is an entry point for the broker because it takes messages from a publisher and routes those messages to the appropriate queue.
- Broker: It is a message broker which provides storage for data produced. The data is meant to be consumed or received by another application that connects to the broker with the given parameters or connection strings.

iBot, as an IoT system, has a lot of data that should manage. That is why the RabbitMQ queues need to ensure that any information will be lost.
A RabbitMQ Messaging Queue has the below flow:

- The producer first publishes a message to an exchange with a specified type.
- Once the exchange receives the message, it is responsible for routing the message. The exchange routes the messages taking into consideration other parameters such as exchange type, routing key, etc.
- Now bindings are created from exchange to the RabbitMQ Queues. Each Queue has a name that helps to separate the two. Then the exchange routes the message into Queues based on the attributes of the message.
- Once the messages are enqueued in the Queue, it remains there until they are handled by a consumer.
- The consumer then handles the message from the RabbitMQ Queue.

2.3 Blockchain

Blockchain[8] technology is the concept or protocol behind the running of the blockchain. Blockchain technology makes cryptocurrencies (digital currencies secured by cryptography) like Bitcoin[9] work just like the internet makes email possible.

The blockchain is an immutable (unchangeable, meaning a transaction or file cannot be changed) distributed digital ledger (digital record of transactions or data stored in multiple places on a computer network) with many use cases beyond cryptocurrencies. Immutable and distributed are two fundamental blockchain properties. The immutability of the ledger means you can always trust it to be accurate. Being distributed protects the blockchain from network attacks. Blockchain as distributed digital ledger consists of multiple nodes (peer-to-peer network), each of which has a copy from the ledger. Each transaction or record on the ledger is stored in a "block." For example, blocks on the Bitcoin blockchain consist of an average of more than 500 Bitcoin transactions. The information contained in a block is dependent on and linked to the information in a previous block and, over time, forms a chain of transactions. Hence the word blockchain.

There are two types of blockchains. The first types are Public Blockchains like Bitcoin or Ethereum[10], which are open and accessible to anyone to request or validate a transaction (check for accuracy). Those (miners) who validate transactions receive rewards. The Private Blockchains, which are not open, have access restrictions. People who want to join require permission from the system administrator. For example, Hyperledger[11] is a private, permissioned blockchain.

To continue, it is important to mention the uses of Blockchains in this work. iBot uses the blockchain to store all the data that it needs. The advantages of storing the data in blockchain are:

- Protects the data from attacks because blockchain is distributed (all nodes have a copy of the blockchain).
- The immutability of the ledger adds validity to data.

[8] https://www.blockchain.com/
[9] https://bitcoin.org/el/
[10] https://ethereum.org/en/
[11] https://www.hyperledger.org/use/fabric

2.4 Smart Contract

Smart contracts[12] are programs stored on a blockchain that run when predetermined conditions are met. They are typically used to automate the execution of an agreement so that all participants can be immediately sure of the outcome without any intermediary's involvement or time loss. They can also automate a workflow, triggering the following action when conditions are met.

Smart contracts permit trusted transactions and agreements to be carried out among disparate, anonymous parties without the need for a central authority, legal system, or external enforcement mechanism.

Once a smart contract has been added to the blockchain, it generally can't be reversed or changed (although there are some exceptions).

Each node stores a copy of all existing smart contracts and their current state alongside the blockchain and transaction data.

Through smart contracts, we can set "rules" that implement the business logic and functional requirements of the system (if "x" occurs, then execute step "y"). For example, we can deploy a smart contract that has functions that implement steps when a device sends an update for its property. The smart contract should have functions that check if the property update is valid (the value is between a range of values). For a temperature sensor, the steps will be

step 1: if a temperature update.

step 2: then checks if the temperature is between -10 and 50.

step 3: if it is ok, then store it in the blockchain else, abort.

We can deploy smart contracts to securely update or retrieve any data stored in the blockchain. In public blockchains, anyone can deploy or execute a smart contract, but in a private blockchain administrator of the blockchain decides who can deploy or execute a smart contract.

2.5 HyperLedger Fabric

Hyperledger Fabric is an open-source, private blockchain framework started in 2015 by The Linux Foundation. Fabric blockchains are private, meaning all participating members' identities are known and authenticated from administrator. This benefit is functional in healthcare, supply chain, banking, and insurance industries, where blockchain cannot expose data to

[12] https://www.ibm.com/topics/smart-contracts

unknown entities. Also, this is a benefit for IoT architecture because this kind of architecture wants to share the devices' data with authorized users. The works [9], [3] have use the Hyperledger Fabric for them IoT architectures.
A Hyperledger Fabric has two types of nodes in the blockchain, the peers and the orderers:
Orderers

- A node cluster that sorts transactions within the block in a first-come, first-serve manner. The Orderer recieves transactions from peers, build the block and after that send the block to peers to update the blockchain.

Peers

- Are the nodes that has stored the blockchain
- Are responsible for executing the code(smart contracts) and its life cycle.

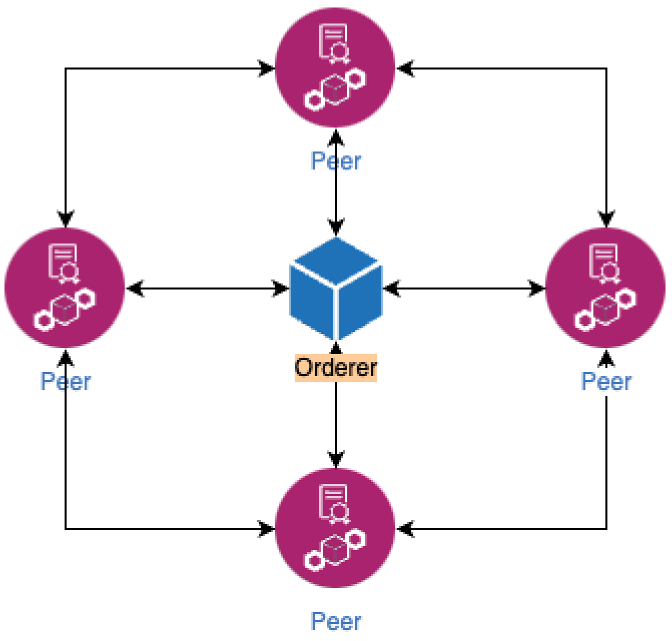
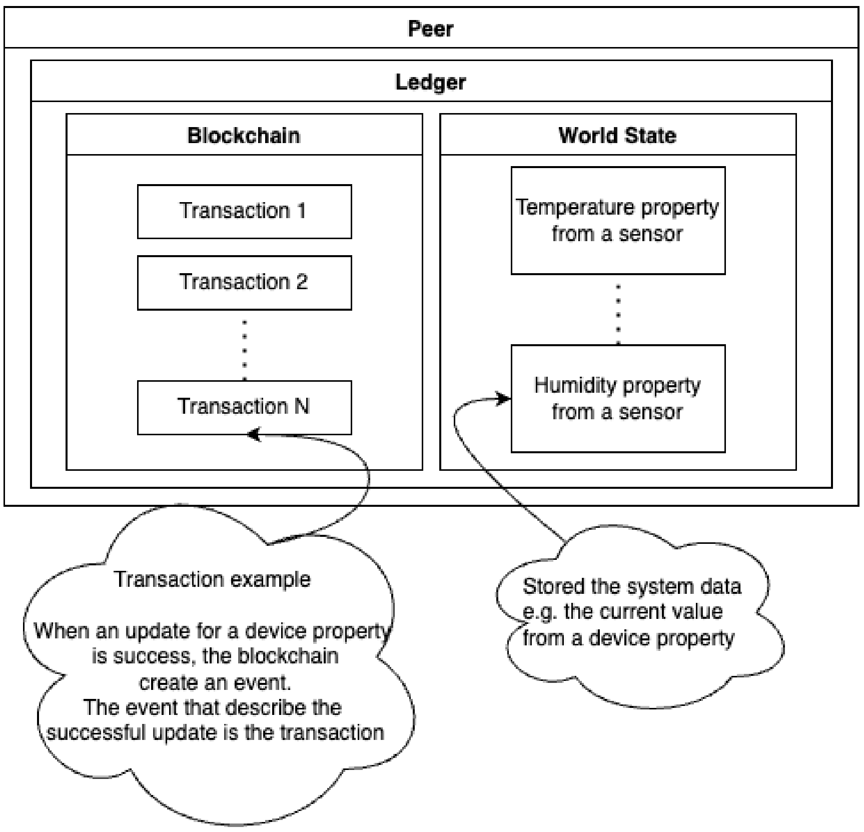


Figure 2.1: Example of a Hyperledger network with 4 peers and 1 orderer.

In Hyperledger Fabric, a ledger consists of two distinct, though related, parts a world state and a blockchain (Figure 2.1).
World State
The world state is a database that holds the current values of a set of ledger states (e.g. the current value from sensors or any other info that stores in blockchain). The stored data (value) and the information for identifying the data (key) are stored as a pair (key-value store or KVS). The state status is maintained only by peers, and smart contracts can only update the stored data.
Blockchain
Blockchain provides a verifiable history of all valid and invalid transactions and successful and unsuccessful status changes. Ordering service

manufactures blockchain as a perfectly ordered chain of blocks. Each peer is responsible for maintaining its ledger and allowing them to repeat the history of all transactions and reconstruct the current state of the blockchain.



Peer
Ledger
Blockchain
World State
Transaction 1
Transaction 2
Transaction N
Temperature property from a sensor
Humidity property from a sensor
Transaction example When an update for a device property is success, the blockchain create an event. The event that describe the successful update is the transaction
Stored the system data e.g. the current value from a device property

Figure 2.2: Example of a ledger that is stored on a peer.

Finally, the Fabric supports smart contracts that a developer can create using Go, Java, and Node.js.

2.6 Decentralized identifier

Decentralized identifier or DIDs are a type of identifier that enables a verifiable, decentralized digital identity. They are based on the self-sovereign identity[13] paradigm. A DID identifies any subject (e.g., a person, organization, thing, data model, abstract entity, etc.) that the controller of the DID decides that it identifies. DIDs are URIs that associate a DID subject with a DID document allowing trustable interactions associated with that subject.

A DID:

- Is a globally unique identifier made up of a string of letters and numbers

[13] https://en.wikipedia.org/wiki/Self-sovereign_identity

- Is created and owned by the user
- Comes with a private key and a public key that are also made up of a long string of letters and numbers

Each DID document can express cryptographic material, verification methods, or service endpoints, which provide a set of mechanisms enabling a DID controller to prove the ownership of the DID.

In iBot, a user, device, or application has a DID, a private, and a public key. The DID and public key are stored in the blockchain, an immutable database. Anyone who wants to verify or authenticate the DID can check the blockchain to see if the corresponding DID exist. Section 3.5.7.1 and Figures 3.10 and 3.11 explain how we use the DID, public, and private keys for user/device/application authentication.

2.7 Verifiable Credentials

Credentials are a part of our daily lives; driver's licenses are used to assert that we are capable of operating a motor vehicle, university degrees can be used to assert our level of education, and government-issued passports enable us to travel between countries. These credentials benefit us when used in the physical world, but their use on the Web remains elusive.
A verifiable credential can represent the same information that a physical credential represents. The addition of technologies, such as digital signatures, makes verifiable credentials more tamper-evident and trustworthy than their physical counterparts.
Holders of verifiable credentials can share them with verifiers to prove they possess verifiable credentials with specific characteristics.

A role is an abstraction that might be implemented in many different ways. The following roles are introduced in this specification:

- holder
    - A role an entity might perform by possessing one or more verifiable credentials and share them. Example holders include students, employees, and customers.
- issuer
    - A role an entity performs by asserting claims about one or more subjects, creating a verifiable credential from these claims, and transmitting the verifiable credential to a holder. Example issuers include corporations, non-profit organizations, trade associations, governments, and individuals.
- subject

    - A subject is an entity about which claims are made. Example subjects include human beings, animals, and things. In many cases, the holder of a verifiable credential is the subject, but in some instances, it is not. For example, a parent (the holder) might hold the verifiable credentials of a child (the subject), or a pet owner (the holder) might have the verifiable credentials of their pet (the subject).
- verifier
    - A role an entity performs by receiving one or more verifiable credentials, optionally inside a verifiable presentation, for processing. Example verifiers include employers, security personnel, and websites.

A verifiable credential is a set of tamper-evident claims and metadata that cryptographically prove who issued it and who is the holder. A credential is a set of one or more claims made by the same entity. Credentials might also include an identifier and metadata to describe properties of the credential, such as the issuer, the expiry date and time, a representative image, a public key to use for verification purposes, the revocation mechanism, and so on. The issuer might sign the metadata.

In the Verifiable Credentials ecosystem, the issuer and holder are required to use Decentralized Identifiers, or DIDs. The public key associated with the DID of the organization (which can also be a user or web service) that issued the credential is stored on the blockchain. So when someone wants to verify the authenticity of the credential, she/he can check the blockchain to see who issued it without having to contact the issuing party.

In our case, a holder is a user/device/application, and the issuer is the service that gives access to the user/device/application to log in to the system. In iBot, the verifiable credentials contain the role of a party of the system (any user, device, or application), so through VCs, the party can claim the role without storing it in any database. The Verifiable Credentials that contain personal details are securely stored on a decentralized digital wallet app(a personal place of storage).

2.8 Web Of Things Service

The WoT Service is an effort made within the framework of the laboratory. It is a proxy that exposes Things on the Web. This proxy deploys on a server (or a gateway) that keeps a virtual image of each Thing (e.g., a JSON representation). Also, it implements a directory (e.g. a database) with all Things descriptions. With this proxy, the Things become part of the Web, which can be discovered and reused. The REST API (i.e., endpoints, payloads, etc) is proposed in the Web Thing Model submission. It adopts the OpenAPI Specification as the main description language for devices and their exposed services. This is a novel implementation based on the WoT Architecture of W3C and builds on the REST API proposed by the Web Thing Model of W3C; it implements all the model's operations on Things using HTTP. The OpenAPI service description framework to Web objects (i.e., Things) uses a common description template. As a result, OpenAPI descriptions of Web Things provide complete documentation of the services exposed by Things and of their capabilities.

2.8.1 OpenAPI Generator

In creating the Web Of Things Service, a mechanism was also created that produces the OpenAPI descriptions. A user aware of the device characteristics can provide all the necessary information for the device and its functionality, including the endpoints, HTTP methods, and data schemas (e.g., request body and response body schemas) required for the service operations, etc. Therefore, all this information can be included in the OpenAPI definition of the Thing, enriched with semantic annotations that further describe concepts such as sensor, actuator, temperature, etc. Also, the mechanism is a RESTful API service that the user sends the input described above.

2.9 Related Works

2.9.1 Blockchain for Large-Scale Internet of Things Data Storage and Protection

The “Blockchain for Large-Scale Internet of Things Data Storage and Protection” [10] is a work that mitigates the problems that have traditional cloud-based IoT structures. These structures impose extremely high computation and storage demands on the cloud servers, so the strong dependencies on the centralized servers bring significant trust issues. The paper proposes a distributed data storage scheme employing blockchain and certificate-less cryptography. The scheme eliminates the centralized server and uses Distributed Hash Tables[11] to store the IoT data. The pointer to the DHT storage address can be stored in the blockchain. In the blockchain, a group of users, also known as miners, work cooperatively to create blocks as a public ledger that validate and record transactions. Also, when an external user/entity requests data from the DHT, the blockchain decides whether the access can be given or not, i.e., the authentication of the requester is handled by the blockchain instead of a trusted centralized server. Furthermore, the blockchain can record activities like accessing and modifying IoT data.

2.9.2 Towards Secure and Decentralized Sharing of IoT Data

The “Towards Secure and Decentralized Sharing of IoT Data” [3] is a work that proposes a framework named Sash. The Sash uses the FIWARE open source IoT platform and the Hyperledger Fabric framework as the blockchain back-end. The blockchain stores access control policies in Sash and takes access control decisions. The framework stores IoT data off-chain but offload the access control functionality to the blockchain—currently handled by a centralized entity such as the IoT broker. Data owners push their data to the off-chain storage and advertise it to the smart contract through an “offer,” so the Sash creates a data marketplace. The Sash stores in the blockchain the following data types:

- Users
    - Have two types (Data owners, Data Consumers)
- The offers from IoT Owners
- The Metadata
    - Each MetaData item has a unique fileID, owner information, storage location, and white-list type ACL(the list that has access to this certain data)
- Transactions

Furthermore, the framework named Sash has two data-sharing schemes: ACLs-based and prefix encryption-based.

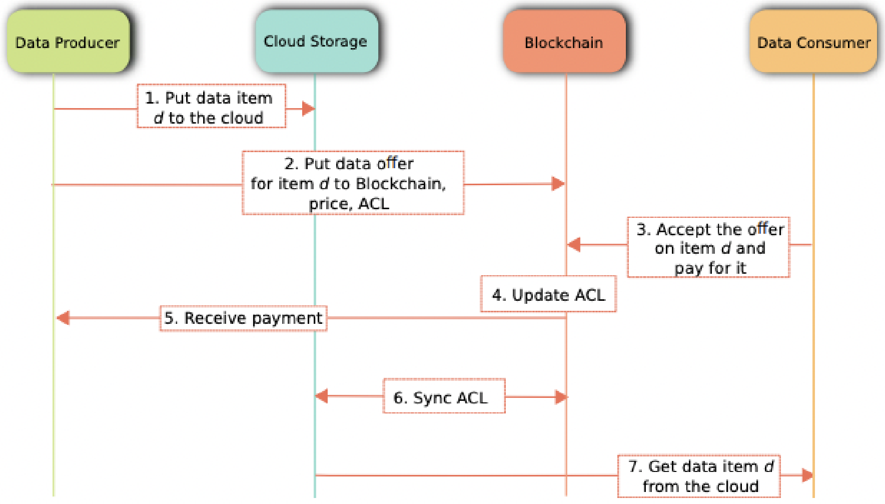

Data Producer
Cloud Storage
Blockchain
Data Consumer
1. Put data item d to the cloud
2. Put data offer for item d to Blockchain, price, ACL
3. Accept the offer on item d and pay for it
4. Update ACL
5. Receive payment
6. Sync ACL
7. Get data item d from the cloud

Figure 2.3: Data sharing scheme based on ACL(Access Control List).

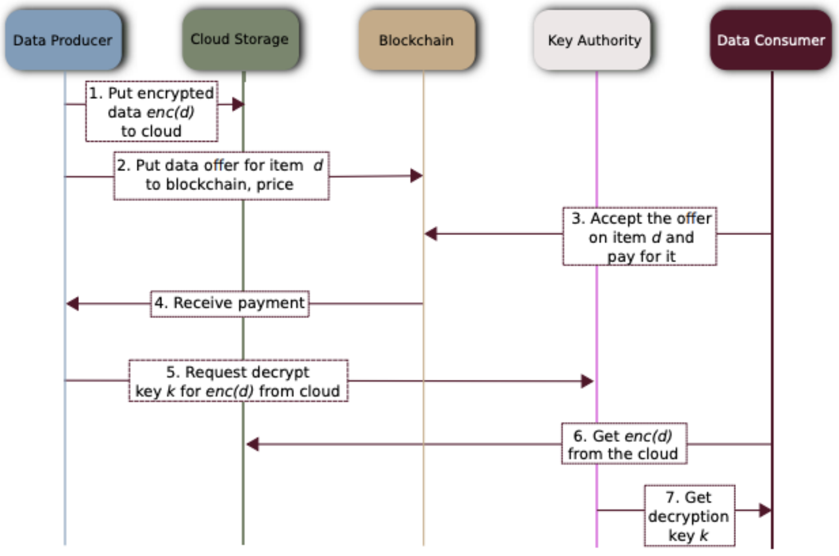

Data Producer
Cloud Storage
Blockchain
Key Authority
Data Consumer
1. Put encrypted data enc(d) to cloud
2. Put data offer for item d to blockchain, price
3. Accept the offer on item d and pay for it
4. Receive payment
5. Request decrypt key k for enc(d) from cloud
6. Get enc(d) from the cloud
7. Get decryption key k

Figure 2.4: Data sharing scheme based on prefix encryption-based.

2.9.3 A Permissioned Blockchain based Access Control System for IOT

The “A Permissioned Blockchain based Access Control System for IOT” [9] is a work a permissioned blockchain-based access control system for IoT were a different phase of access control like creating access policy and making the access control decision happens based on the consensus of all the stakeholders. Remarkably, the paper proposes a distributed access control for IoT that uses a permissioned blockchain called Hyperledger Fabric and leverages its smart-contract and distributed consensus. The proposal consists of two actors the resource provider/owner and the requester. The resource provider/owner is the owner of the IoT, and the requester is any party that accesses data generated by IoT devices is a requester. The system has two more components the blockchain and the local IoT network. The blockchain works as a policy enforcement point for any access request to a particular IoT resource. All attributes and Attribute Based Access Control (ABAC) policies are stored in the blockchain. Also, each local IoT network is composed of one or more IoT devices, a sink node, and a gateway. The sink node works like a network coordinator for all the IoT devices and is connected to the gateway. The gateway acts as an interface to the external world to access any resource within the local IoT network.

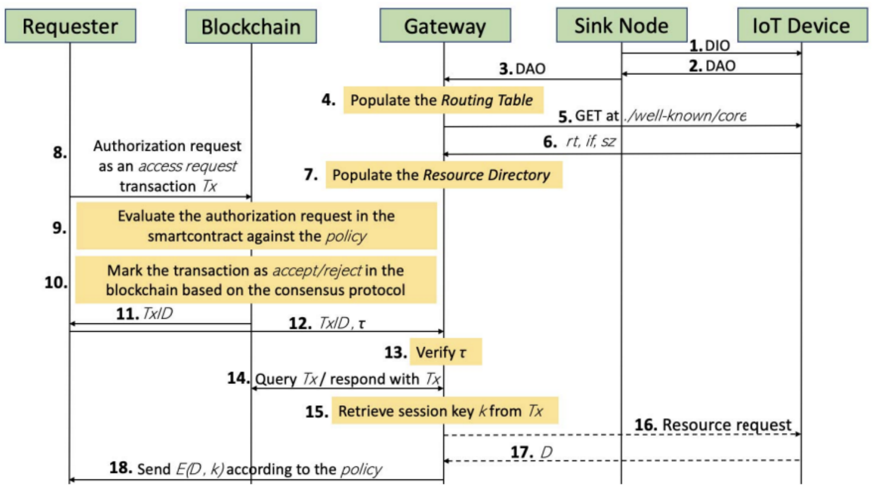

Requester
Blockchain
Gateway
Sink Node
IoT Device
1. DIO
2. DAO
3. DAO
4. Populate the Routing Table
5. GET at ./well-known/core
6. rt, if, sz
7. Populate the Resource Directory
8. Authorization request as an access request transaction Tx
9. Evaluate the authorization request in the smartcontract against the policy
10. Mark the transaction as accept/reject in the blockchain based on the consensus protocol
11. TxID
12. TxID, τ
13. Verify τ
14. Query Tx / respond with Tx
15. Retrieve session key k from Tx
16. Resource request
17. D
18. Send E(D, k) according to the policy

Figure 2.5: Resource Access.

2.9.4 IoTeX

IoTeX is more than just a blockchain, it is a full-stack platform to enable trusted data from trusted devices for use in trusted Apps. The platform use Decentralized Identities that enables users/devices to own their data, identity, and credentials.This blockchain is compatible with 2 IoT devices (Pebble Tracker, Ucam). Ucam replaces traditional password-based login with blockchain-based login and uses the user's private key to end-to-end encrypt all videos. Also, the Pebble tracker is equipped with a TEE and multiple sensors (GPS, climate, motion, light). Captures/signs data from the real world and converts it into verifiable, blockchain-ready data.

3. System Requirements and Design

3.1 Use cases

We have an IoT system consisting of 3 levels. These levels represent different types of users. The types of users are Infrastructure owners, application developers, and ordinary system users. By differentiating the functionality of the system by type of users, we achieve the following:

- Easier description of the system in subsections or subsystems.
- Ability to develop a flexible commercial use model underlying the system (Business Model).

By adopting a 3-tier architecture model, we create an environment that makes the best use of devices (by re-using them by multiple applications) and creates value from the use of the system by potential application users. Then a brief description of each level is given separately.

Layer 1 is the infrastructure level; the first user group (Infrastructure Owner) operates at this level. At this level, infrastructure owners have the right to register new devices in the system, which they can later manage. The purpose of the users is to offer devices that provide data (measurements) and will be able to use them to create applications.

In layer 2, users are active a) the system administrators and b) the application developers. The purpose of the administrators is the control (monitoring) of the platform at all levels and the management of users (creation, recovery, deletion of users) and devices. The second category of users (application developers) is to create applications using infrastructure devices after obtaining their usage rights. The latter is achieved by creating a subscription to the devices that they want to exploit.

In layer 3, the end-users can use the system to view and subscribe to existing applications that the developer has created.

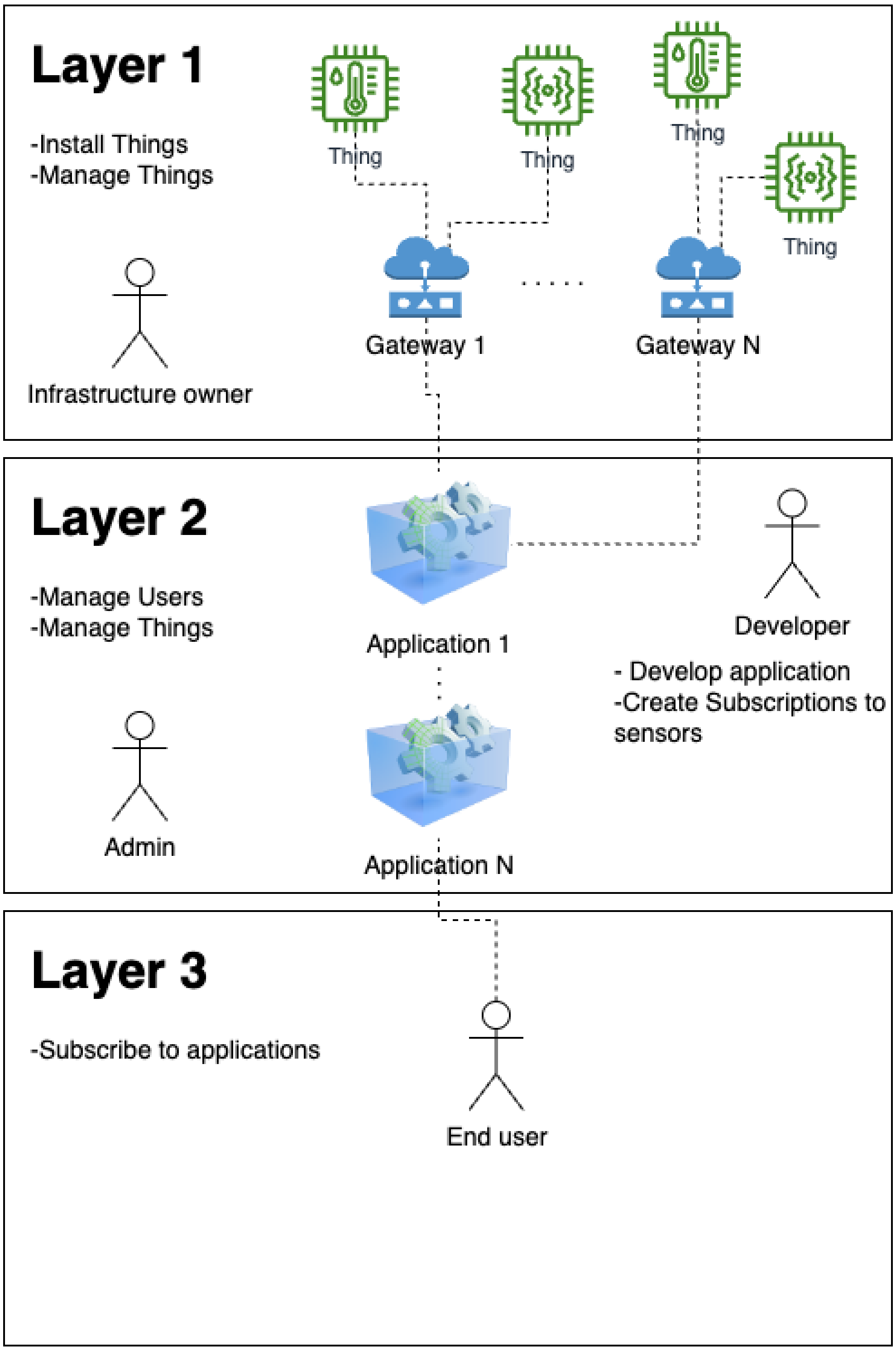


Figure 3.1: 3-tier architecture.

3.2 Functional and non-functional system requirements

The system requirements are then defined, separating functional and non-functional.

3.2.1 Functional system requirements

Functional requirements are defined as the procedures that must be performed to satisfy the system and are inextricably linked to its implementation. These requirements may be different for each user group; therefore, they should all be satisfied for it to be considered an entirely usable system. In the present work, we consider that the users perform the functions allowed by the user category to which they belong through graphics interfaces. A common element that all user groups have is that they have a Wallet. The wallet is an application that storages the VC, DID, public and private keys. The functions related to it are described below each user category.

Users

- Login system.
    - The user enters his login details on the login page to log in.
- Applies for VC and Decentralized Identifier.
    - The user fills out a form and submits it to the system. After that, the system issues the VC and DID. Users use VCs and DID to access the system and query or view applications that use data from the system sensors.
- Search for existing applications.
    - The user can search for applications, so the user has access to view the OpenAPI descriptions of the applications.
- Subscription to applications.
    - The user can create a subscription to one or more applications that he is interested in, to have the right to access them. (Otherwise, it cannot access any application). Respectively, in the same way, he can cancel an existing subscription to one application.
- See the cost of the system usage.
    - As long as the user can subscribe to applications, there is a corresponding charge for the use he makes of the system. The user is charged one transaction every time he subscribes or uses an application.

Developers

- Login system

    - The developer enters his login details on the login page to log in.
- Applies for VC and Decentralized Identifier.
    - Developer fills out a form and submits it to the system. After that, system issues the VC and DID. Developer uses VCs to access the system.
- Search for IoT and see the OpenAPI description.
    - The developer can search for sensors based on geographic location or through properties/actions. Also, the developer has the possibility to query the system to see only the descriptions of the temperature sensors (property: temperature). Through queries, he can search for sensors and see their descriptions.
- Retrieve Properties of devices
    - As long as the developer can see the OpenAPIs of the devices. Then it can retrieve properties by calling the corresponding endpoints. With the same logic, a developer can retrieve the corresponding actions of the device.
- Create VC and DID for apps to connect with the system.
    - Once they have been granted access (and have received VCs/DIDs) they can create VC/DIDs for the applications they create/posses. The developers who have made the applications have access to change the rights of the applications. The applications should also have those wallets with their own VC and DID as users or developers. In this way, the applications acquired developer rights.
- Subscription to IoT devices
    - The developers can, in turn, search the system for the various sensors with the criteria described above. Once they find the sensors they are interested in, they can subscribe to them and declare the callback URL that will receive notifications when the price of the sensor they have subscribed to changes. The subscriptions are stored in the blockchain. Also, the system only sends subscribers the price of a device that works properly.
- See the cost of the system usage.
    - As long as the developer can retrieve/subscribe to the properties of the devices, then there is a corresponding charge for the use he makes of the system. The developer is charged one transaction every time he retrieves/subscribes a property/action of the connected device having the corresponding cost.

Infrastructure Owners

- Login system
    - The developer enters his login details on the login page to log in.
- Applies for VC and Decentralized Identifier.
    - The user fills out a form and submits it to the system. After that, the system issues the VC and DID.
- Search for IoT and see the OpenAPI description.
    - The infrastructure owner can search for sensors based on geographic location or through properties/actions. That is, the infrastructure owner has the possibility to query the system to see only the descriptions of the temperature sensors (property: temperature). Through queries, he can search for sensors and see their descriptions. He can see only the descriptions of owned devices
- Create VC and Decentralized Identifier for devices
    - Infrastructure owners are those who own the devices. As the devices are part of the system's actors, they should have credentials (VC & DID). Infrastructure owners are responsible for creating the VC of the devices and the DID. As a result, the devices can connect to the system and send information.
- Update / Delete devices
    - The infrastructure owners can edit one already existing sensor entity. Also can delete the device from the system by deleting the DID & VC of the device. If it is deemed appropriate to delete a sensor entity that is used by applications and by extension, included in developer subscriptions, the system should update via Email to the subscribers affected by this change.
- Update Thing Description and OpenAPI for the devices
    - Also, infrastructure owners have the right to edit the OpenAPI of the devices. The device's OpenAPI is stored in the blockchain.
- See the income from his/her devices.
    - The infrastructure owner has the function of seeing his income from the sensors he has installed in the system.

Admin

- Manage IoT

    - The admin can change the credentials and decentralized identifiers of IoT devices. E.g., they can delete a DID from the system so that a device cannot connect to it if, e.g., considered a malicious device. When a device is deleted, the subscriptions related to the device are also deleted. The system should inform the subscribers affected by this change via Email.
- Manage Users
    - Admins have the right to edit user data such as credentials and decentralized identifiers. In this way, they can change a user's rights from user to Developer. They can also remove new DIDs from the system, causing users to be unauthenticated, so they cannot log into the system. The deletion of a user (deletion of their DID)should be simultaneously deleted from the system information entities related to him. Suppose the deleted user belongs to the user category developer. In that case, applications should be deleted equally: The subscription to sensors and the applications it has created (Deleting one application should trigger an informative Email to the final users who maintained a subscription to it). In the event of the deletion of an infrastructure owner, the devices belonging to the owner are also deleted.

- Install smart contracts
    - In addition, they can add Smart contracts to the system. This function is necessary because many devices may not be served by the system because there is no corresponding smart contracts related to the proper information acquired by devices and to prices. Still, there is a possibility in the future to be created later (the smart contract) and installed on the system. This functionality belongs only to the admin (as the authority that monitors the system) because is risky for the system. The system can be compromised if the smart contract is not well/properly defined.

IoT Devices

- The only thing a device has the right to do is to be able to send the information(property/action updates) it produces to the system. The system stores device’s infos in the blockchain.

3.2.2 Non-functional system requirements

Meeting these requirements is unnecessary to perform an application's basic functionality. However, the degree of their fulfillment affects the quality of the final product accordingly, especially if it is for commercial application. These include:

- Performance
  - Refers to the response speed of the system under high conditions load. In the present work, it is considered appropriate for all the functions defined in subsection 3.2.1 to be executed in real-time.
- Scalability
  - Refers to the ability of the system to improve and expand its functionality. In the work system, it is essential to new services, new smart contracts, new users, new sensors, and new can be added applications without disturbing the operation of the system.
- Security
  - It concerns the security of users, i.e., their secure access to the system and the protection of their identity and personal property data. All system actors (users, IoT, Services) have DID and VC, which are stored locally in a wallet for every actor separately. At the same time, it concerns the system's protection as a whole and prevents access to services and data over the network by non-authorized sources (services or users or IoT). Without DID and VC, the actor can not log in to the system and generate an access token that uses it in every request. Additionally protects the system from incorrect data or malicious actions that a device may send to the system. The system executes automated checks every time a device sends data to the system via different types of smart contracts(depending on the device type). The developed system infrastructure ensures, by the level of architecture, that all the requests between its services have the appropriate authorization, excluding non-authorized users and services to access the internet in system resources.
- Usability
  - Determines how easy the system is to use. This category includes features such as graphical interfaces and anything else

that aims to improve the application's user experience. It also determines how easy it is to connect a device to the system and that the information it sends is secure to the system.

- Availability time
  - As mentioned in the previous chapter, Service-Oriented Architecture significantly facilitates the maintenance and expansion of applications through independent services. The system must always be online and available for use in web applications. Therefore we want as little downtime as possible. In addition, the specific architecture allows sharing workload on different virtual machines, as the services are not necessarily hosted on the same virtual machine, increasing its system stability, as there is no single point of failure in the system. In all of the above, the blockchain adds more stability since it is a decentralized system, and it is almost impossible for all the system nodes to stop working.

3.3 Use Case Diagrams

Having described in Section 3.2.1 the functional requirements of users per user category, they are grouped below in UML (Use Case Diagrams) for the best capture.

More specifically, the functional requirements formulated in section 3.2.1 are shown in the figure diagram for the end user.

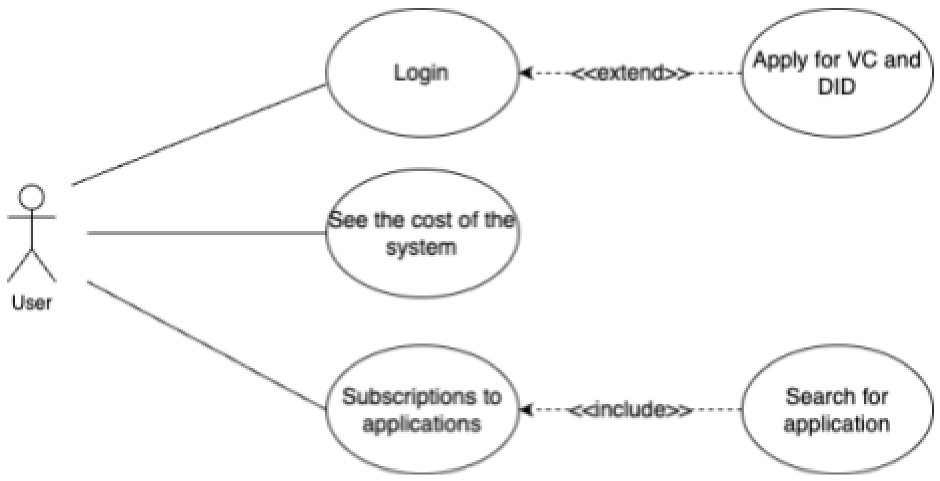


Figure 3.2: Use case Diagram for end user.

Respectively, for the application developer, the functional requirements, as described in section 3.2.1, are shown in Figure 3.3.1.

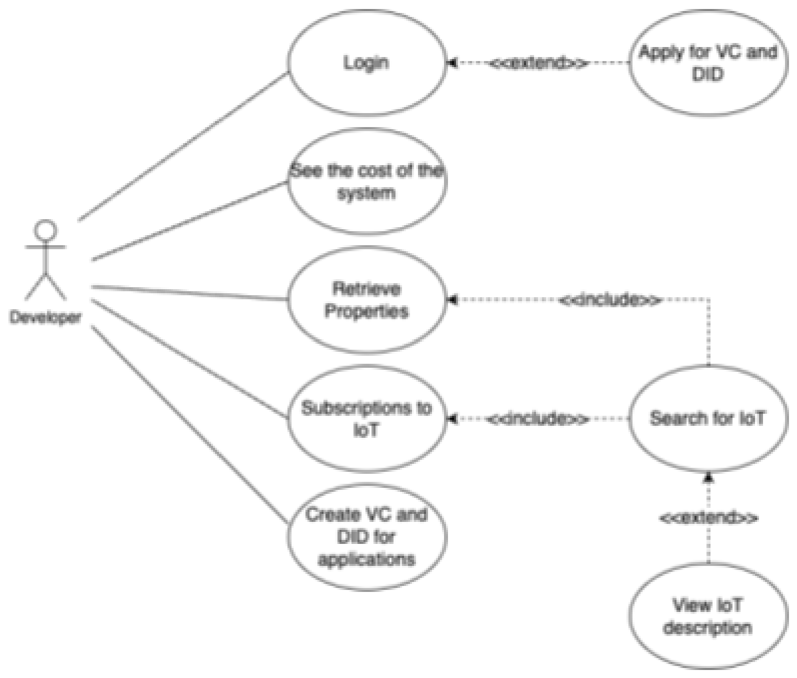


Figure 3.3.1: Use case Diagram for Developers

For the Infrastructure owner the functional requirements as described in section 3.2.1 are shown in Figure 3.3.2

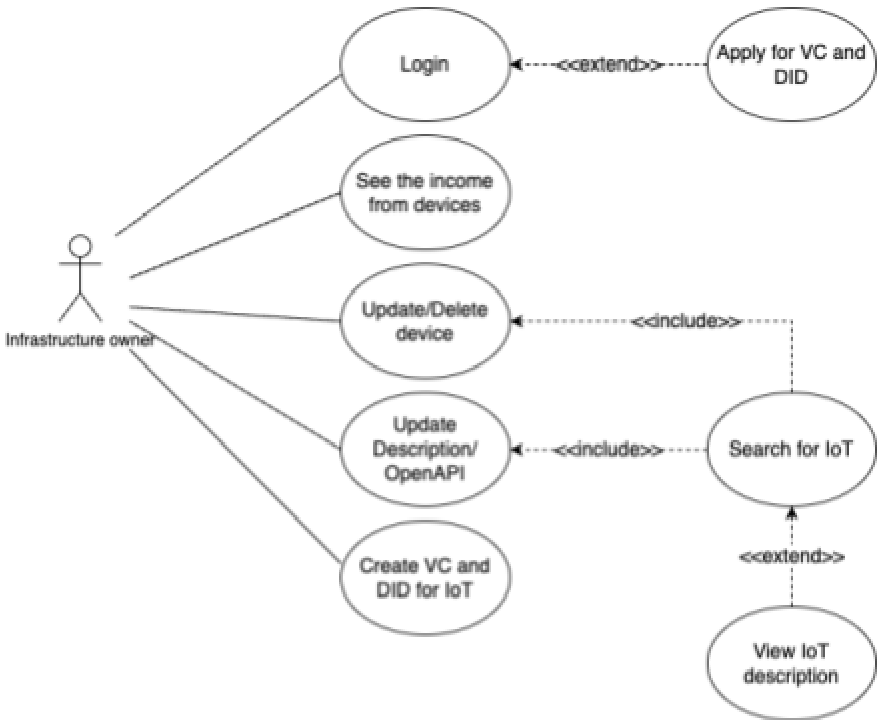


Figure 3.3.2: Use case Diagram for Infrastructure owner.

Finally for the admin the functional requirements as described in section 3.2.1 are shown in Figure 3.3.3

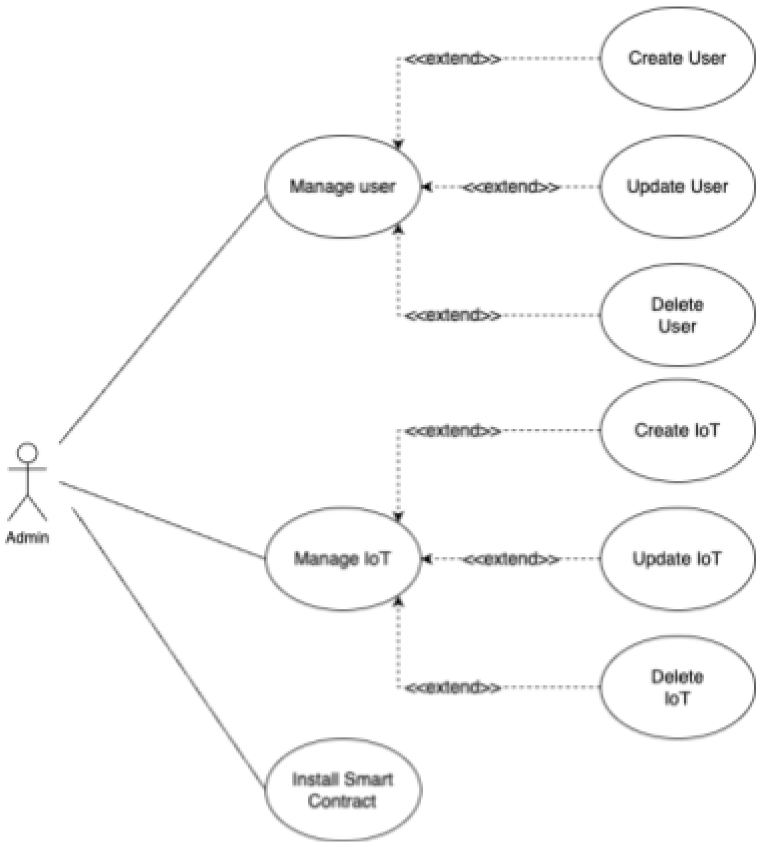


Figure 3.3.3: Use case Diagram for Admin

The IoT devices as an actor of the system has the following function requirements

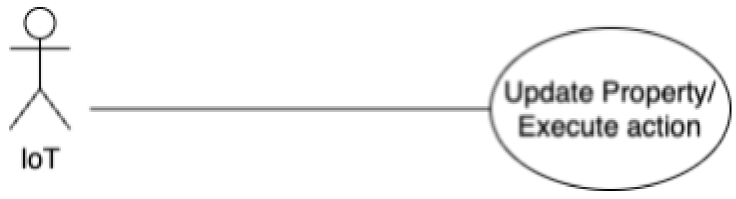


Figure 3.3.4: Use case Diagram for IoT devices

3.4 Activity Diagrams

Here are the activity charts for Developers, users, and devices. The aim is to present these categories' most important system usage scenarios. The diagrams for the functions of the system manager as a notification of them were not the subject of the work.

Figure 3.4 shows the general idea of sending information from sensors using the blockchain to ensure the information is valid. The devices data is stored in the blockchain, such as the OpenAPIs descriptions. However, device VC and DID that need to create access are stored in the wallet.

1. The Infrastructure owner should use the VC and DID, that is stored in the device's wallet, and send them to the system to log in the device. After that, the system(Authorization service) will generate an access token (more details in Section 3.5.7.1).
2. If the credentials are valid, information (the new value of a property/action) is sent to the blockchain.
3. The blockchain, through smart contracts, checks, if the value sent is valid.
4. If the value is valid, the system keeps two types of data, Aggregate and Raw, generated from this value. Furthermore, the system keeps the current value of a property/action.
5. When the current value is saved, we send it through the Pub/Sub mechanism that the value of the specific device has changed.
6. In turn, the Pub/Sub mechanism sends this information to the subscribers.

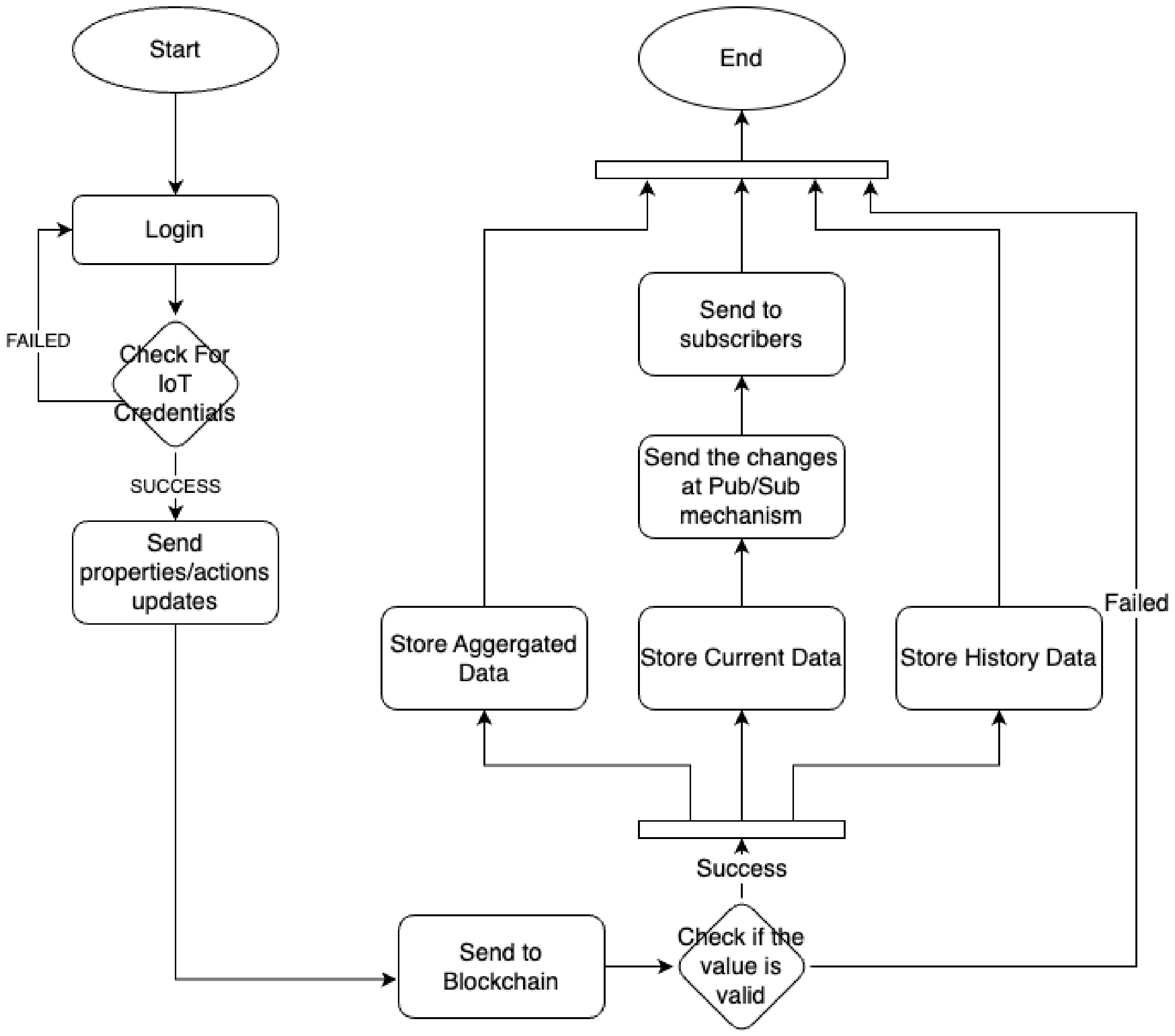


Figure 3.4: Activity Diagram for IoT PUT Properties (shows the flow when a device sends a value).

Next, the developers' usage scenario for creating subscriptions on the devices is presented in figure 3.5

1. As a first step, the developer must connect to the system by giving him the corresponding credentials.
2. If the credentials are valid, then log in to the system.
3. Then, through the graphical interface, he can make queries to see the IoT devices he is interested in based on these queries; some of these queries can be queried to find the sensors based on a specific location.
4. You can see each device's description if the query is correct and some devices are returned.
5. After seeing the description of each device. Then, it can subscribe to the specific device so that you return this change every time there is a new piece of information related to it.

6. Finally, the subscription is saved in the system (on the blockchain) after subscribing to a specific device.

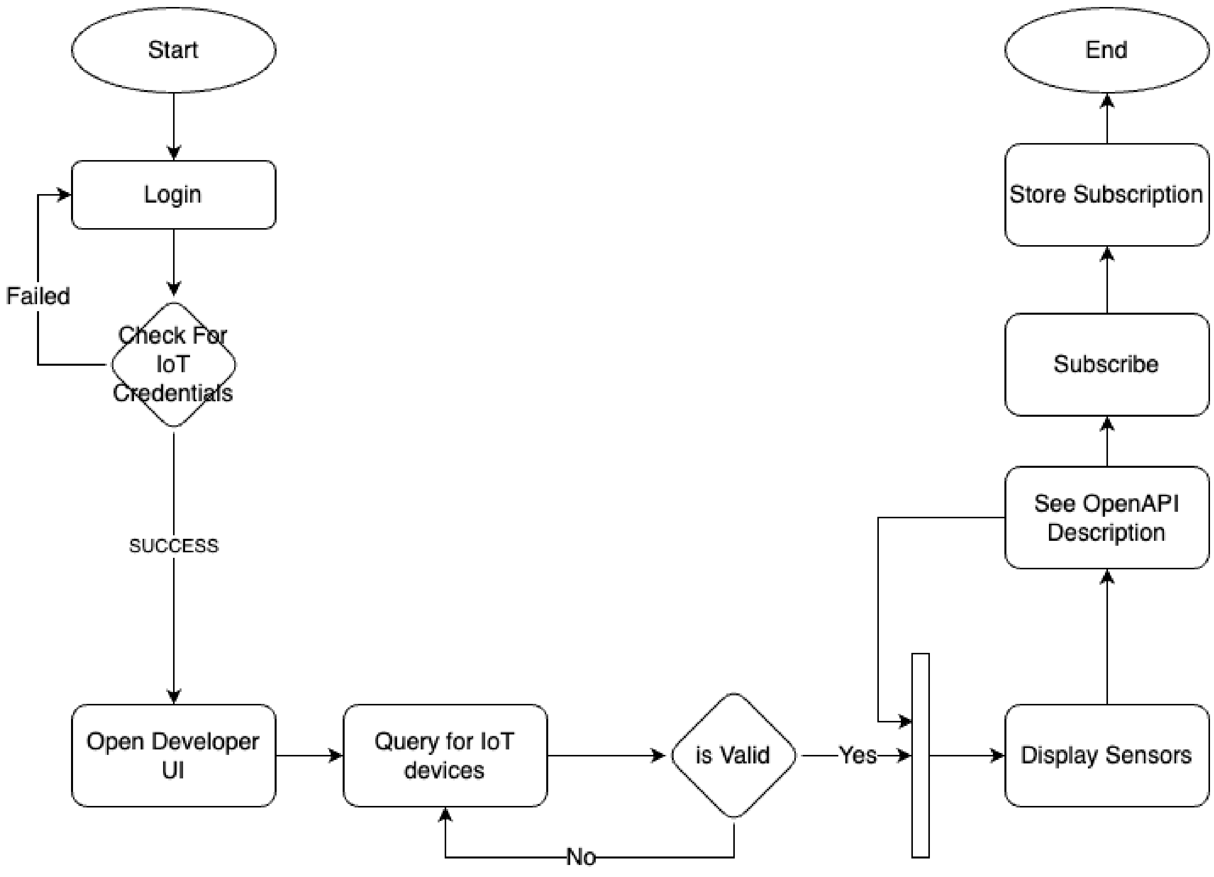


Figure 3.5 Activity Diagram for Developer subscription on IoT.

Finally, a scenario worth noting is the developers' creation of a wallet for Apps. As previously mentioned, developers can create applications with their Wallet to be able to interact with the system. This scenario is presented in figure 3.6.

1. As a first step, the developer must connect to the system by giving him the corresponding credentials.
2. If the credentials are valid, login to the system.
3. Then, the public and private key pairs are created.
4. Open the UI and select: create DID document for an Application.
5. Then, fill out the form with data: role, developer's DID, and app's public key.
6. The Authorization service creates the DID Document and saves it to the blockchain.

7. Auth service returns the DID to the developer.
8. Then, the developer creates the VC with the app's DID and issues the VC.
9. Finally, he/she saves the VC to the app's wallet.

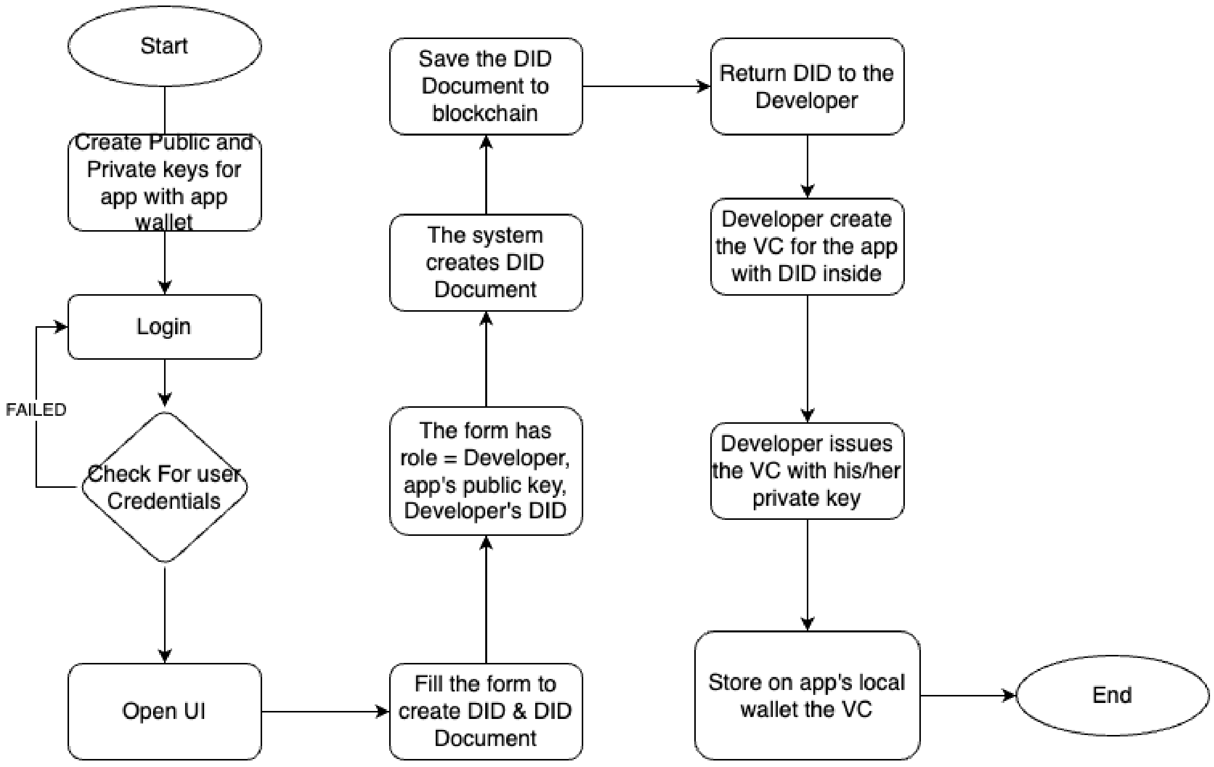

Start
Create Public and Private keys for app with app wallet
Login
FAILED
Check For user Credentials
Open UI
Fill the form to create DID & DID Document
The form has role = Developer, app's public key, Developer's DID
The system creates DID Document
Save the DID Document to blockchain
Return DID to the Developer
Developer create the VC for the app with DID inside
Developer issues the VC with his/her private key
Store on app's local wallet the VC
End

Figure 3.6 Activity Diagram Create VC for Apps.

3.5 Architecture Diagram

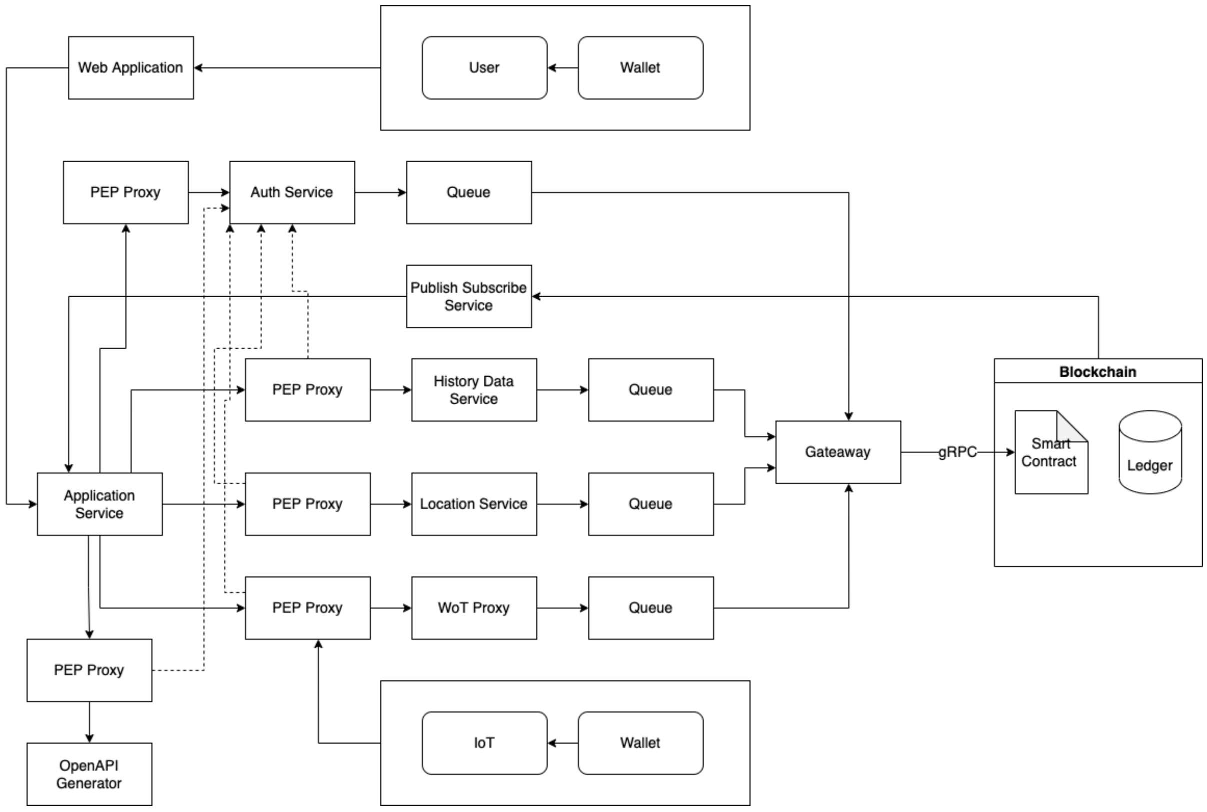


Figure 3.8: Architecture Diagram

The blockchain is used as a database where each Node of the blockchain has a non-relational basis. The blockchain contains raw and aggregated historical data of time series measurements for all sensors connected to the system. The smart contract undertakes to accept data that are sent to the blockchain and, in turn, store them either in raw or aggregated data to maintain the device's measurement history. The smart contract also undertakes the retrieval of the data from the blockchain.

In addition, the blockchain is used to store any information we want to store in the system, such as transactions, DID Documents, OpenAPI descriptions, and subscriptions.

DID Documents are used by the identification and authorization service to certify the existence of the user/device/service and the validity of the information sent or requested. Also, the specific service stores the DID Documents of the new users/devices/services that are created.

Subscriptions are stored on the blockchain because it is used by the event and subscription management service to store the information entities it manages.

In the present work, the idea is used that each device is treated as a Restful service. Each device has an OpenAPI description where the endpoints that each device can execute are described. The specific OpenAPIs are stored in the blockchain so authorized users can search, see the descriptions, and interact with the devices through the OpenAPI they provide.

Finally, there is a need to record the interaction of ordinary users/developers with the system. We store this record as a transaction of each user with the system. All transactions are also stored on the blockchain.

Below is a detailed description of the individual services of the system as well as the cooperation between them to implement the functions as mentioned above.

3.5.1 Blockchain and smart contracts

One of the system's essential components is the blockchain. The blockchain is the component that stores all the information we need, as mentioned above. By nature, it can store a transaction history without being able to change it. Using this feature, we store the value from devices while maintaining an unchanging time series history. In addition, we are interested in storing other information beyond the sensors' measurements, such as the grouped data. To be able to automatically create the grouped data every time a new measure enters the system, we use smart contracts. Smart contracts, as mentioned in the previous section, are programs installed on the blockchain nodes, which have methods and are called by services of the system. In the specific implementation, we have different smart contracts, each of which has methods for the various needs of the system. Every time an IoT wants to send information to the system, it calls the method of the smart contract where it stores this value in the blockchain, but at the same time, the corresponding method of the smart contract is called, which creates grouped data and stores it in the blockchain. The data received by blockchain is stored with two tactics:

- Raw: Data is stored raw as received in the relevant sensor measurement history.
- Aggregated: Aggregated data are statistical values resulting from the

combination of the newly arrived data with the existing historical data of the sensor. These prices concern:

  - Maximum value among all samples for a sensor measurement field in the last month/day/hour.
  - Minimum value among all samples for a sensor measurement field in the last month/day/hour.
  - The sum of all samples for a sensor measurement field in the last month/day/hour.

As can be seen, smart contracts have the role of services that can retrieve or store different types of information on the blockchain.
In addition, the blockchain is our single "database" as previously mentioned. For sharing these types of data, we also have different types of contracts that sometimes communicate between them. The types of smart contracts are as follows:

- Smart Contract IoT Proxy
  - The smart contract implements the methods that WoT Proxy calls to perform the operation of each device. The methods of the specific contract are responsible for retrieving or updating the various components of the devices, such as properties, actions, subscriptions, thing descriptions, and thing models.
- Smart Contract Security
  - The smart contract controls the values that the devices transfer to the blockchain. If the prices pass the check, then they will be stored in the system (More details below at section 3.5.7.3).
- Smart Contract Raw Data
  - As mentioned above, the smart contract stores the raw data as received in the measurement history of the relevant sensor.
- Smart Contract Aggregated Data
  - As mentioned above, the smart contract stores the aggregated data.
- Smart Contract Transaction
  - It is the smart contract that has the methods that show the use of a user in the system. With this smart contract, we can see how many times a user has used a sensor either by asking for a value or by making a subscription to it. Every time a user requests to retrieve a sensor value, a transaction is created that is stored in the user's history.
- Smart Contract DID
  - The smart contract is responsible for storing or retrieving the DID Document used by the identification service.

As previously mentioned, smart contracts work together to meet the needs of the system at the same time. The cooperation scenarios are as follows:

- When a measurement is sent to the blockchain, the corresponding method is called the Smart Contract IoT Proxy. Then we want to check if this value is valid, so we call the corresponding form from Smart Contract Security. In the end, we want to store the raw and aggregated data when the Smart Contract Raw Data and Smart Contract Aggregated Data are called.
- Another smart contract collaboration scenario is when a user(developer type) requests data from the system or subscribes to some device. When this is done, the Smart Contract IoT Proxy is called to retrieve the corresponding value or to make a subscription to the device, while the Smart Contract Transaction is then called to store in the history that a user requested each value from the specific sensor or made the corresponding subscription.
- A corresponding collaboration with the previous one is when the user(developer type) requests aggregated data. The first call is the Smart Contract Aggregated Data, then the Smart Contract Transaction to store this usage.

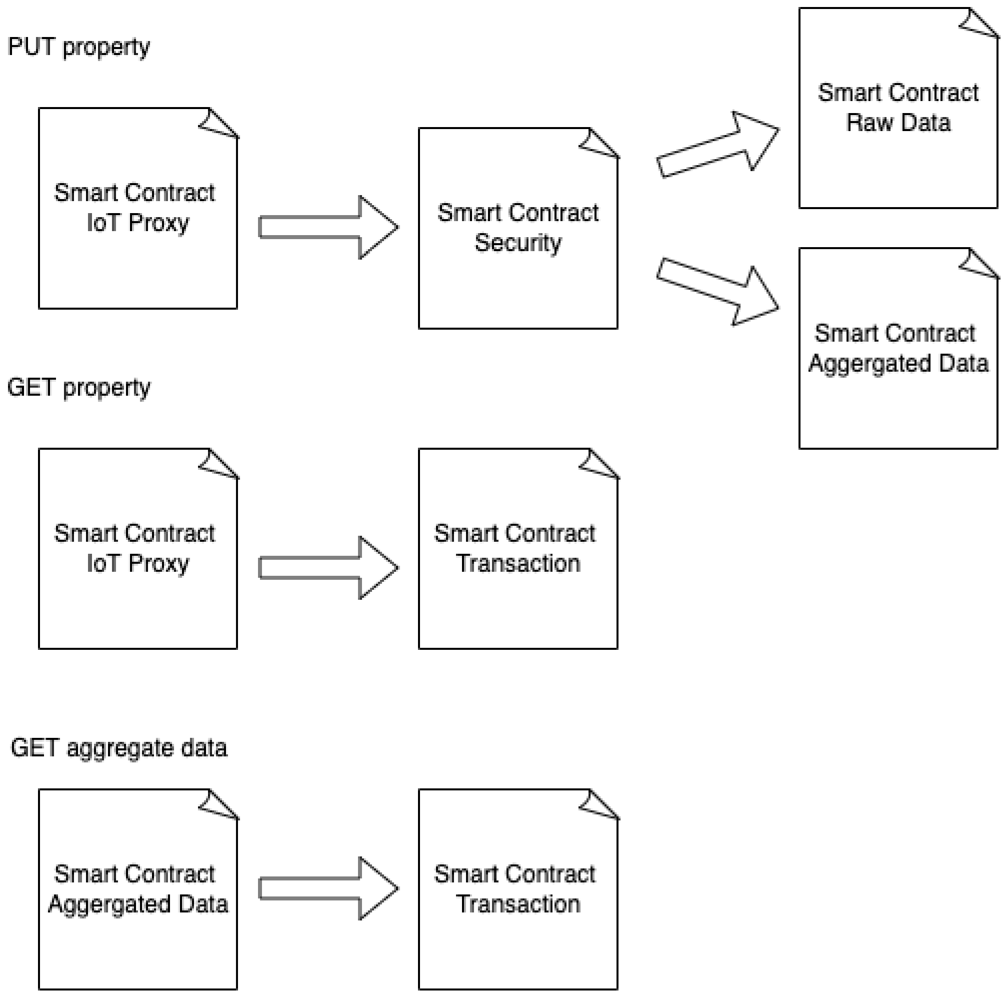


Figure 3.9: Smart contract flow examples

In addition, it is not necessary for any smart contract method to cooperate with anyone else. For example, retrieving the description of a device does not call any other smart contract but only finds the description stored in the blockchain and returns it.

3.5.2 Application Logic service

The Application Logic is the heart of the system as it includes the code to orchestrate the individual services so that the system implements the functionality defined in the specifications. The user interface system (Web Application) is considered part of the logical application as it includes the necessary code to implement the system's graphical interfaces (for all different types of users). The requests of the system users arise from the user interface system and are forwarded to the application logic for their appropriate routing. Here is a relevant example of their operations:

- A user through the user interface system requests to enter the system. The request is routed to the Application Logic service, which in turn routes it to the Auth service. The Application Logic service then creates a login session for the user, including their credentials and corresponding access token. Once the details are verified, the user will be logged into the system.

Following this logic, all requests arising from the GUIs of the user interface system are routed through the "application logic" service to their intended services.

3.5.3 WoT proxy & OpenAPI Generator

3.5.3.1 WoT proxy

To implement the system requirements, we need a Web Thing Proxy that allows the interaction of clients (i.e., users or devices) with Things on the Web and proposes a particular Web Service (i.e., a REST API) that implements particular operations. Each time a Thing registers to Web Thing Proxy, a new entity is created in the blockchain. A Thing is identified by its resources: a Web Thing Resource, a Model Resource, a Properties Resource, an Actions Resource (as long as the Thing supports actions), a Things Resource, and a Subscriptions Resource. The blockchain stores all data about Things (i.e. descriptions, properties, measurements, actions, action executions, and subscriptions to Things). The Web Thing Model service can retrieve, update, or delete these data.

3.5.3.2 OpenAPI Generator

As previously mentioned, Things are described by specific resources stored in the blockchain. According to this logic, we use the OpenAPI Web Thing template that employs a JSON (or YAML) description format, which is common to all Things. It is a valid OpenAPI document that can be handled by all known OpenAPI tools (e.g., Swagger editor, code generator, etc.). To help users create these descriptions, a mechanism that produces OpenaAPI thing Descriptions has been created. The mechanism generates the OpenAPI description of a Thing from user input. The input comprises a) the standard OpenAPI Thing Description template that applies to all Things and b) a payload in JSON with the user settings (e.g., security settings) and the Thing characteristics that will be instantiated to the template. The user specifies the necessary information that characterizes the device and the functionality it supports (e.g., the properties it provides, the actions it performs, etc.). The output of this mechanism is the OpenAPI description of the Thing (in YAML or JSON format). The mechanism is a RESTful service.

3.5.4 Publish Subscribe Service

The Publish-Subscribe service is a mediator for sensor subscriber entities. These entities are stored in JSON format in the blockchain. The JSON representation of an instance of the "Sensor Subscription" entity refers to a specific application developer. It is a "list" of the sensors that the developer has added to his/her subscriptions for all his/her applications. It includes unique event ID and unique Sensor IDs (Sensor’s DID) and the date of sensor subscriptions. The service checks every moment if changes are made to the sensor prices on the blockchain. When any change is made to the current sensor price, the smart contract creates an event with the value change, the DID of the device whose value changed, and the subscriptions that exist for the specific device. The Publish-Subscribe service "listens" to these events and has the JSON files for the records that contain the callback URLs. The service sends the changed data to these URLs. In addition, the service has a non-relational database as cache memory to store the most recent changes.

3.5.5 Aggregate Data Service

The Aggregate Data Service is the RESTful interface of historical data stored on the blockchain containing raw & aggregated data. It is connected to the blockchain and provides REST methods for retrieving raw and aggregated historical time series data about the evolution of system sensor measurements. For example, in the scenario where we need to retrieve the maximum temperature measured by the sensor {wt} for each hour of the last 24 hours, we will use the following REST method:

```
GET localhost/aggregate-data/{sensor_id}/{property}
{
“method”: [“max”],
“period”:”hours”,
“dateFrom”: “2022-03-12T00:00:00”,
“dateTo”:”2022-03-12T23:00:00”
}
```

3.5.6 Location Service

The Location Service is the RESTful location data interface for sensors connected to the blockchain. It is connected to the blockchain and provides REST methods for retrieving the location of a sensor or for retrieving sensors that open in a geographic zone defined by the user by giving some limits for latitude and longitude. For example, a user can provide Crete's geographical length and width and return all the sensors within these limits.

3.5.7 Security Services

The security of the system, as mentioned in section 3.2.2, results from its architecture, prohibiting the use of its functions by users and services that do not belong to the system or do not have access authorization. The three components responsible for the system's security are the Auth service, Policy Enforcement Points, and Smart Contracts. The system is now not only protected from unauthorized users/applications/devices but also from the malicious information/value a device can send.

Before we explain the usage and functionality of the Authorization service, we need to clarify the wallet's usage and concept. The wallet is an application on the user's local computer/mobile/device that keeps the information the user/IoT/application needs to connect to the system. The wallet is an application that we developed that stores the VC, public and private keys, and DID. Also, the wallet has a mechanism that generates a public and a private key, so the user can, through this mechanism, generate his/her public and private keys. Another useful mechanism is that a user can issues a VC with his/her private key (Admin, Developer and Infrastructure can use this mechanism).

3.5.7.1 Authorization service

We developed this service, which is the system's starting point as it is responsible for registering and connecting users. When a User wants to register, he creates a pair of public & private keys. Then, during registration, the user defines the characteristics that make up his profile, such as role, public key, and passphrase. These data are sent to the system's router's Application Logic service. After that, it sends the data to the Authorization (Auth) service through its RESTful interface, which creates the unique DID and the DID Document.

The DID Document is a JSON which has the format described in section 2. It consists of the following elements:

- id : Is the DID unique for each document and is also the identifier for the owner of the DID document. We can retrieve the DID Document from the blockchain through this specific identifier.
- verificationMethod: A DID document can express verification methods, such as cryptographic public keys that can be used to verify data that is signed by the corresponding private key. The verification method MUST include the id, type, controller, and public key fields.
    - id: It is the same DID as the previous id, the only difference being that #keys-1 has been added at the end.
    - type: The value of the type property must be a string that references exactly one verification method type. In this case, we use the verification method RsaVerificationKey2018[14].
    - controller: the DID of the person who has the right to change the DID Document/VC elements. The users in the controller field have the same DID as theirs, while a device has the DID of its owner because only he can make changes to the device.
    - publicKeyMultibase: It is the Public Key that the user has created with the wallet as described in the previous section. It is the Key that verifies information signed with the private key of each user.

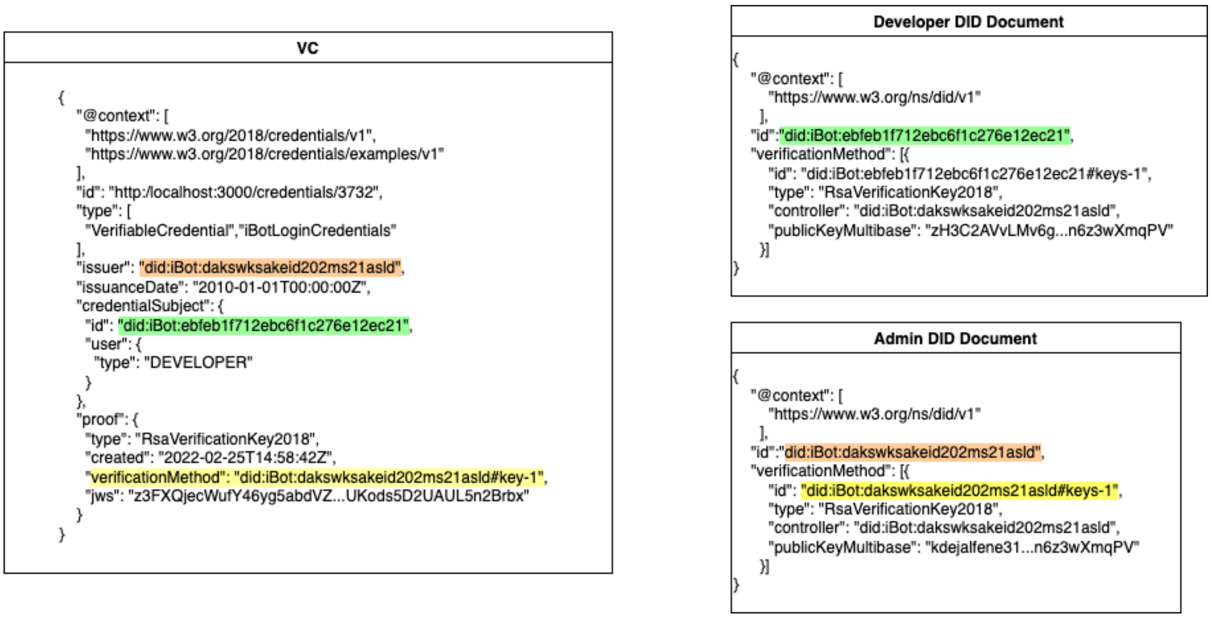
After the DID Document is created, the DID Document is sent to the blockchain for storage. Since the DID Document has been saved, the Auth service should create the VC of the user. The VC is also a JSON file with the format described in section 2. The VC has the following fields:

- id: It is the identifier generated by the service (Auth service) and is unique.
- type: It is the type of Verifiable credential. In this particular case, the type is iBotLoginCredentials, which describes that it is a credential for connecting to the system.
- issuer: Enter the identifier of the service that signed and created the VC in this field. Because the Auth service creates the VCs of the system, then the link of the service is entered in the specific field.
- issuanceDate: Is the date the VC was created.

[14] https://w3c-ccg.github.io/ld-cryptosuite-registry/#rsasignature2018

- credentialSubject: This is one of the most important fields in VC. It consists of the fields id and user. The id field stores the DID of the actor related to the VC, while the user field stores the type of user that can be developer, user, infrastructure owner, IoT, application, and Admin.
- proof: It is the service's signature that the record is valid. The signature is a JSON Web Signature[15](JWS) generated by the service and contains the user's role, DID, and passphrase. By sending this information to the service, the service can understand if the file is valid.

In Figure 3.10, we have an example describing the relation between a developer's VC and the DID of the admin that issued VC. In the field proof, the VC has JWS issued from the Authentication service with the admin private key and has the verification method from Admin DID Document as a verification method (for the purpose of Thesis this is an automated mechanism). The verification method from Admin DID Document has the public key that verifies the VC's signature. Also, the field issuer has the DID from Admin, while the field credentialSubject has the DID and the Developer's role.


VC

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "http:/localhost:3000/credentials/3732",
  "type": [
    "VerifiableCredential","iBotLoginCredentials"
  ],
  "issuer": "did:iBot:dakswksakeid202ms21asld",
  "issuanceDate": "2010-01-01T00:00:00Z",
  "credentialSubject": {
    "id": "did:iBot:ebfeb1f712ebc6f1c276e12ec21",
    "user": {
      "type": "DEVELOPER"
    }
  },
  "proof": {
    "type": "RsaVerificationKey2018",
    "created": "2022-02-25T14:58:42Z",
    "verificationMethod": "did:iBot:dakswksakeid202ms21asld#key-1",
    "jws": "z3FXQjecWufY46yg5abdVZ...UKods5D2UAUL5n2Brbx"
  }
}
```

Developer DID Document

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1"
  ],
  "id":"did:iBot:ebfeb1f712ebc6f1c276e12ec21",
  "verificationMethod": [{
    "id": "did:iBot:ebfeb1f712ebc6f1c276e12ec21#keys-1",
    "type": "RsaVerificationKey2018",
    "controller": "did:iBot:dakswksakeid202ms21asld",
    "publicKeyMultibase": "zH3C2AVvLMv6g...n6z3wXmqPV"
  }]
}
```

Admin DID Document

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1"
  ],
  "id":"did:iBot:dakswksakeid202ms21asld",
  "verificationMethod": [{
    "id": "did:iBot:dakswksakeid202ms21asld#keys-1",
    "type": "RsaVerificationKey2018",
    "controller": "did:iBot:dakswksakeid202ms21asld",
    "publicKeyMultibase": "kdejalfene31...n6z3wXmqPV"
  }]
}
```

Figure 3.10: Example of VC and DID from a developer and the admin DID.

Finally, the created verifiable credential is returned to the user to store it in his wallet and used later as a valid user credential (the process is also described in Figure 3.11). With this specific implementation, the data is not exposed to the database, while each user has control over when he wants to

[15] https://en.wikipedia.org/wiki/JSON_Web_Signature

provide his data. The data, such as the role of the user and the private key are only in the personal wallet of the user. At the same time, in the blockchain (that is used as a database), there is only the DID Document that declares the existence of each user and does not contain any other element.

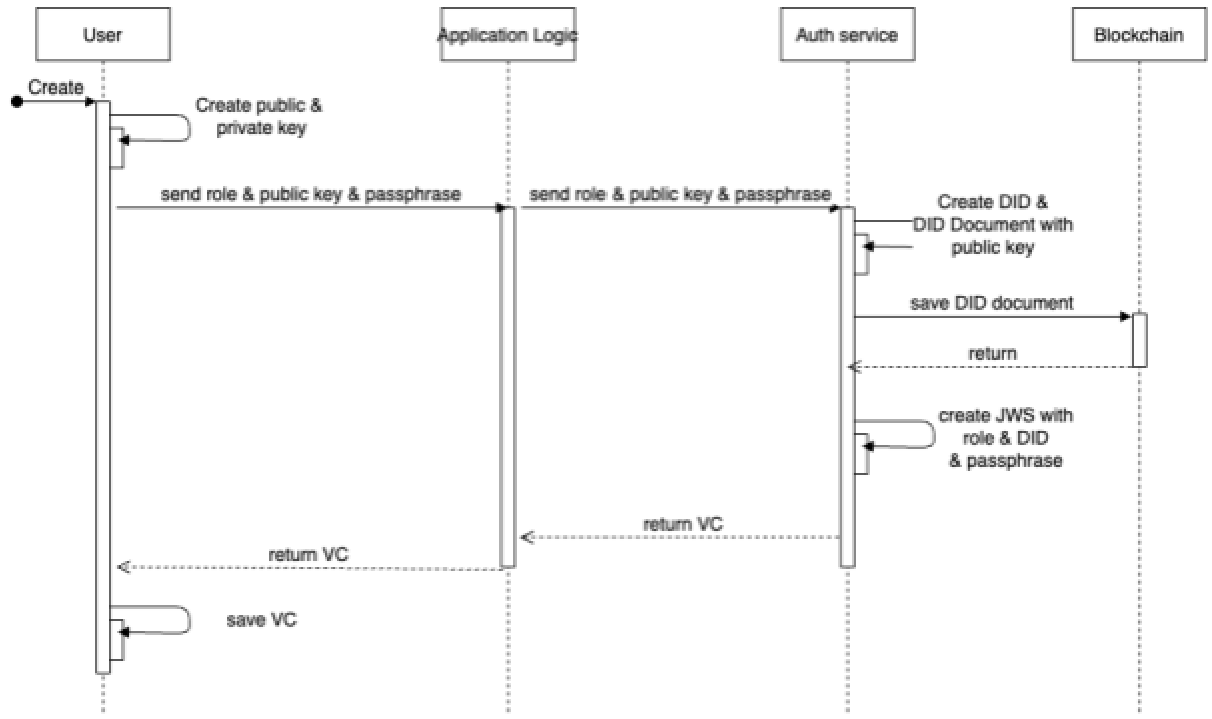

User
Application Logic
Auth service
Blockchain
Create
Create public & private key
send role & public key & passphrase
send role & public key & passphrase
Create DID & DID Document with public key
save DID document
return
create JWS with role & DID & passphrase
return VC
return VC
save VC

Figure 3.11: Create VC & DID Document for user

After that, the applications and the devices are registered in the system in a similar way process. The difference is the VCs for applications and devices issued by developers and Infrastructure owners, respectively.

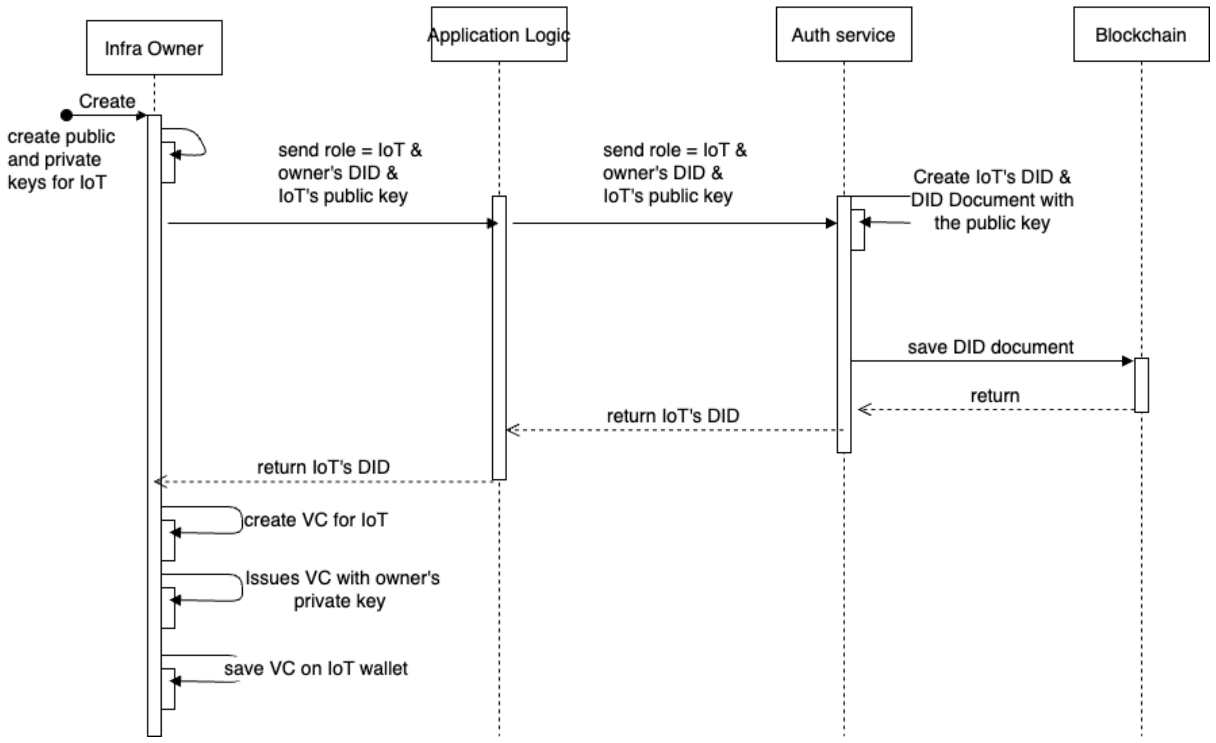

Infra Owner
Application Logic
Auth service
Blockchain
Create
create public and private keys for IoT
send role = IoT & owner's DID & IoT's public key
send role = IoT & owner's DID & IoT's public key
Create IoT's DID & DID Document with the public key
save DID document
return
return IoT's DID
return IoT's DID
create VC for IoT
Issues VC with owner's private key
save VC on IoT wallet

Figure 3.12: Infrastructure owner creates and issues VC and DID for an IoT.

The Auth Service, through the Restful interface, can identify registered users/applications/devices and grant them access to the system (Figure 3.13). First, the user should have the following Infos:

- a passphrase encrypted with user's private key.
- The user's DID.
- The user's role.
- The field proof from the user's VC.

Then the Auth service searches the blockchain for the DID Document and the user's public key. With the user's public key, Auth service can decrypt the encrypted passphrase. Then the service with admin public key (that issues the JWS) and JWS can verify that the data (user's role, DID, and passphrase) are valid.

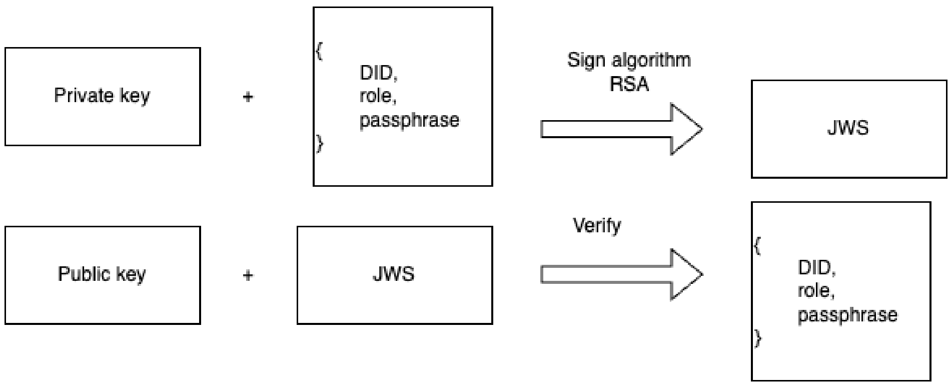


Figure 3.13: Flow for JWS creation and verification.

After data validation, the service creates an access token (JSON Web Token[16]) that includes the user's role and DID. The access token is a unique-dynamic token with limited time validity, which acts as the user's identity within the system.

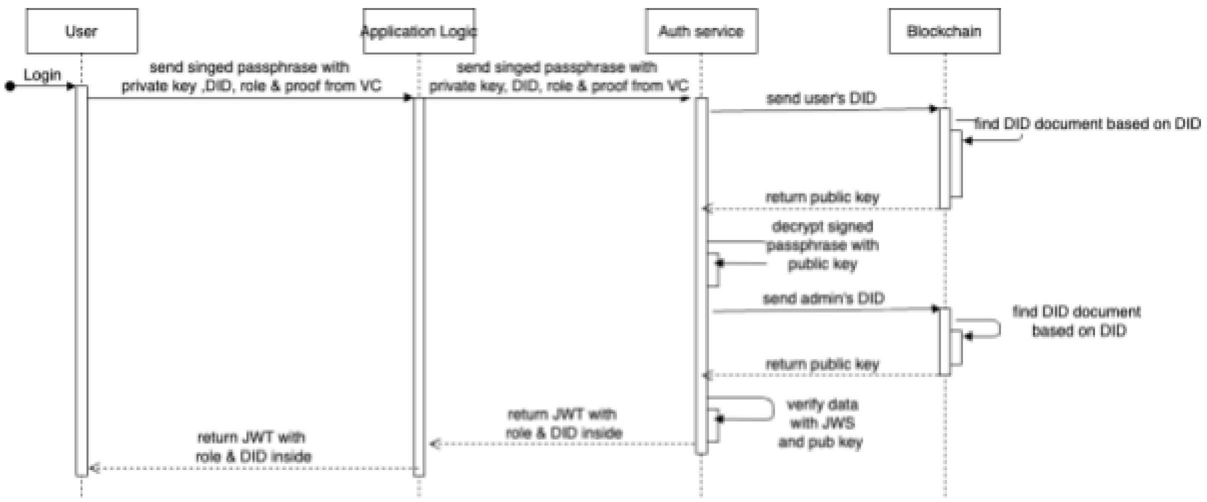

User
Application Logic
Auth service
Blockchain
Login
send singed passphrase with private key ,DID, role & proof from VC
send singed passphrase with private key, DID, role & proof from VC
send user's DID
find DID document based on DID
return public key
decrypt signed passphrase with public key
send admin's DID
find DID document based on DID
return public key
verify data with JWS and pub key
return JWT with role & DID inside
return JWT with role & DID inside

Figure 3.13: Login

3.5.7.2 Policy Enforcement Point (PEP)

A PEP is a proxy server that acts as an intermediary for requests from users seeking resources from other services. PEP proxy is developed by us using the Guard[17] class from NestJS[18] framework. A user connects to the proxy server, requesting some service, such as a file, connection, web page, or another resource available from a different server. The proxy server undertakes to forward the request to the server with the specific service and, after receiving the query response, delivers it to the requester.

[16] https://jwt.io/
[17] https://docs.nestjs.com/guards
[18] https://docs.nestjs.com/

Services in the architecture that have resources not legitimately accessible by unauthorized services or users do not publicly expose their REST interface. Therefore, to be able to serve external (Relative to the virtual machine on which it "runs") requests, each service has a local proxy server that undertakes to accept at its public endpoint the requests intended for it and forwards them to it.
The Policy Enforcement Proxy is a proxy that requires in the header of the HTTP requests it receives one of the following two tokens; otherwise, the requests are ignored:

- Access token: A valid access token corresponds to a user and has been created by the Auth service upon entering the system.
- Master Key: This secret code is specified during the initialization of the Policy Enforcement Proxy Server. Each different Policy Enforcement Proxy in the architecture has its own - unique - Master key.

If the request it accepts carries the secret code "Master Key" and the Master key code is correct, the PEP Server forwards it to the service it mediates and returns its response to the requester (Figure 3.14).

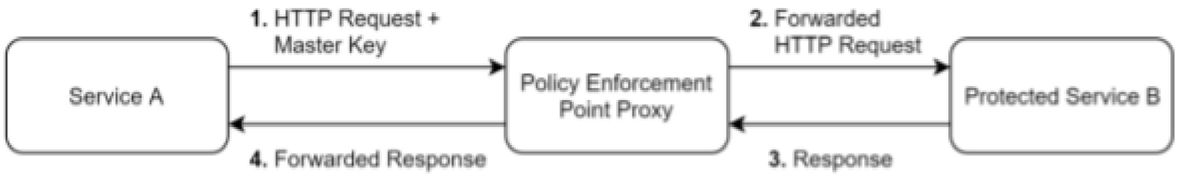

1. HTTP Request + Master Key
2. Forwarded HTTP Request
Service A
Policy Enforcement Point Proxy
Protected Service B
4. Forwarded Response
3. Response

Figure 3.14: Service authentication flow via PEP proxy

In the following case of requests using an access token, the process is illustrated in Figure 3.15, and below, the actions performed are analyzed in the order in which they occur.

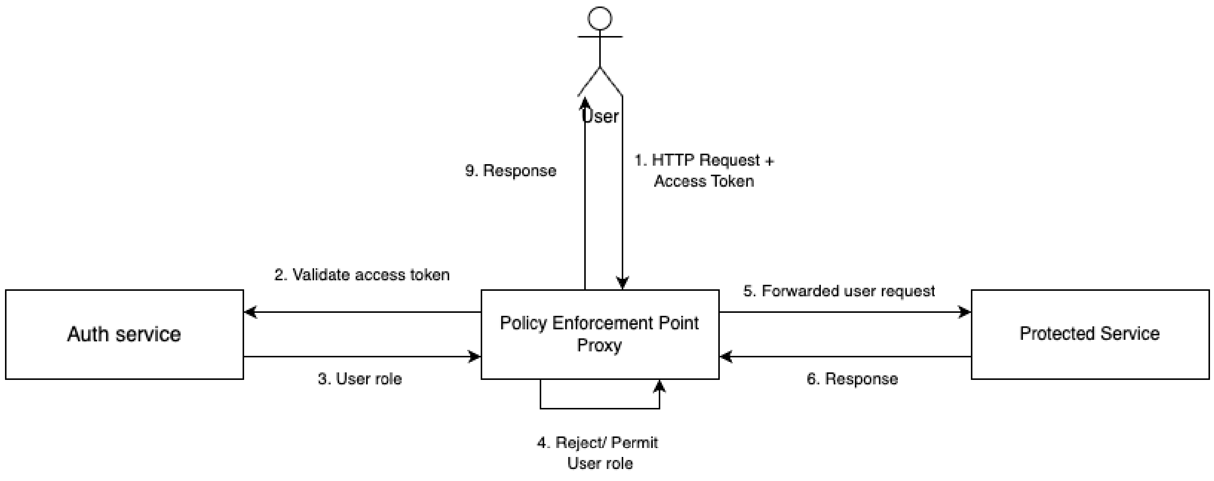

User
9. Response
1. HTTP Request + Access Token
2. Validate access token
Auth service
Policy Enforcement Point Proxy
5. Forwarded user request
Protected Service
3. User role
6. Response
4. Reject/ Permit User role

Figure 3.15: User authentication flow via PEP proxy

1. The PEP Server accepts an HTTP request that includes an access token.
2. Checks if the token that carries in its header the request it received is a valid access token. This check happens via REST communication with the Auth service.
3. Once the validity of the access token is confirmed, the Auth service returns to the PEP Server the information related to the identity and roles of the user related to the access token, such as the public key and the role. In case the access token is not valid, the process stops here.
4. It is now time to evaluate if the request made by the user in step 1) is approved based on the roles he has.
5. If the request is approved, the PEP Server forwards the request to the protected service it is mediating.
6. The service returns the request response to the PEP Server.
7. The PEP Server forwards the service response to the user.

As shown in the system architecture diagram (Figure 3.8), every service not legitimate to offer its REST interface cooperates publicly with a PEP.

In Figure 3.15, we notice that Auth service returns the user's role. We have implemented a Role Base Access Control [19](RBAC) mechanism based on a different level for each user/device/application according to the category to which it belongs. The PEPs have mapped each user type to a service operation. We created this mapping when we developed the PEPs. If the user type and request's operation do not match, the request is rejected.

[19] https://en.wikipedia.org/wiki/Role-based_access_control

In the remaining functions, where communication occurs between system services (and not between a system service and a request received from a user or a service outside the system such as an external IP), there is no need to identify or configure different levels of authorization. Therefore, in this scenario, a system service, when it requests another service of the system, includes in the header of its request the appropriate Master key code of the responsible PEP. The PEPs, in this case, work according to the process depicted in the diagram in figure 3.14.

In this way, the system infrastructure ensures that a service that receives the above "security" is not available to requests outside the system.

3.5.7.3 Smart Contract Security

With the above two sections, we have covered the security cases in which any system actor cannot communicate with the system services without the corresponding permission. In addition, another point that leaves the system exposed to threats is the devices that are connected to the system. A device can send malicious or wrong information to the system. This can happen for various reasons, such as the device being damaged or its hardware being changed by human intervention. To deal with such weaknesses, we used the advantage that the blockchain gives us, the smart contracts, and the WoT Proxy described above.
First, by using smart contracts, we can perform some actions the moment we send information to the blockchain and check if the values are correct before we store them in the blockchain. With this logic, we can write smart contracts that execute some procedures before storing the value sent to them. An example is when a temperature sensor sends its value to a smart contract to write it on the blockchain, the smart contract should check if this value is less than 50 and if it is, then writes it on the blockchain. We focused on this concept and created different types of smart contracts so that we could control different scenarios. Three scenarios are:

- The value range control.
- The amount of information sent by a sensor.
- The frequency it sends.

Next, we assume that every time a device sends information to the system, it declares which method from and which smart contract it will use to write the information to the system. This can be done because we assume each device is a restful service that can send information.

Algorithm:

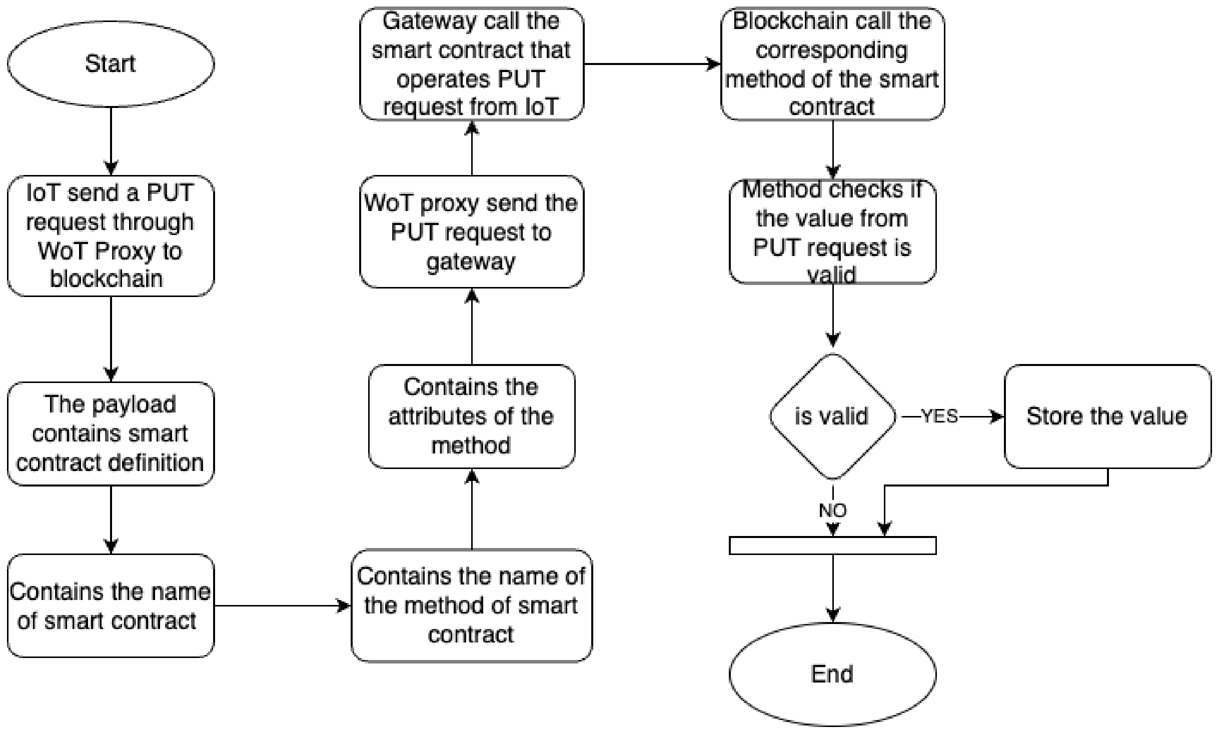

Start
IoT send a PUT request through WoT Proxy to blockchain
The payload contains smart contract definition
Contains the name of smart contract
Contains the name of the method of smart contract
Contains the attributes of the method
WoT proxy send the PUT request to gateway
Gateway call the smart contract that operates PUT request from IoT
Blockchain call the corresponding method of the smart contract
Method checks if the value from PUT request is valid
is valid
YES
Store the value
NO
End

Figure 3.16: The flow that automatically checks a device's value through the smart contract.

Done in this way, every information a device sends to the system is checked by the system itself; if it is valid, it will be written in it.

3.5.8 Queues

The work describes a system of IoT devices. The big problem in an IoT system with many devices is that it can manage all the information it receives continuously without losing measurements or crashing the system. For this reason, we use the blockchain, which by its very nature is secure and quite difficult to fail because of the many nodes it has. Additionally, we use queues to ensure that the requests sent to the blockchain are not lost. We use queues between the services that send requests to the blockchain, with the result that we use four queues. The queues are as follows:

- A queue is between the Auth service and the blockchain.
- The second queue is between the Aggregate Data service and the blockchain.
- The third queue is between the WoT Proxy and the blockchain.
- The fourth queue is between the Location service and the blockchain.

With the queues, we temporarily store the continuous messages from the sensors/users until the blockchain consumes them.

4. System Implementation

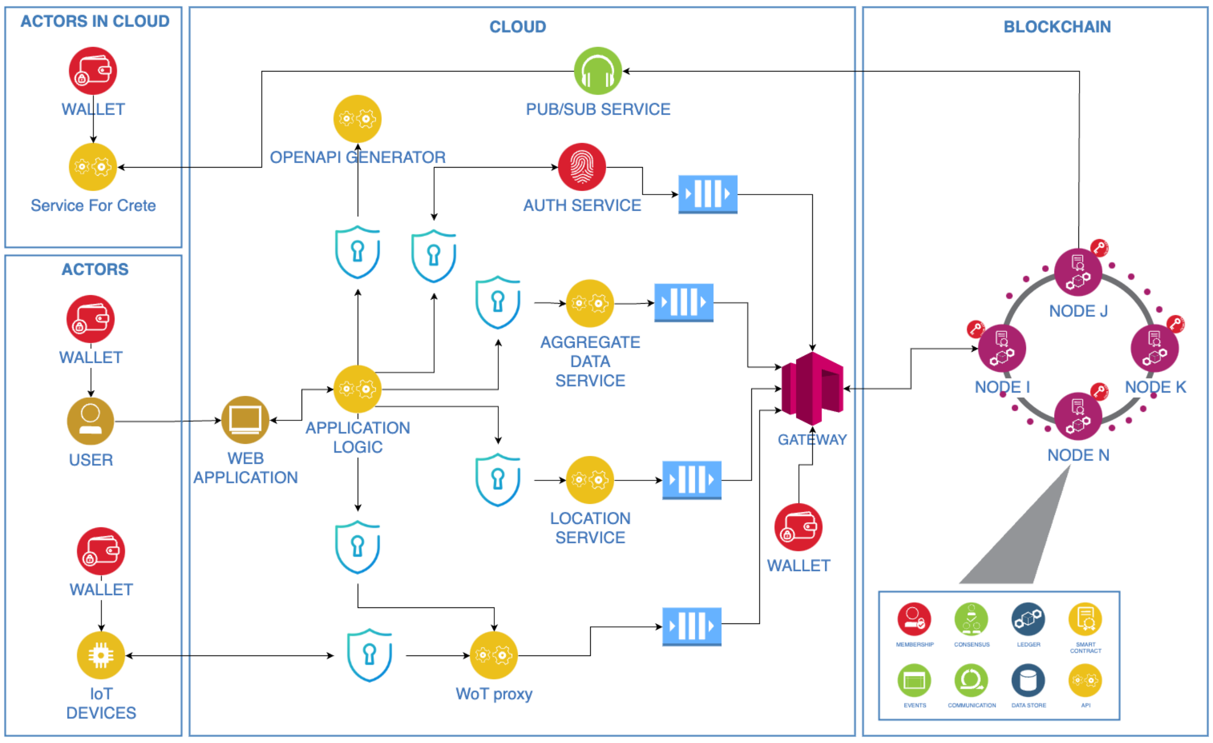


Figure 4.1: System architecture implementation diagram

Based on the specifications set in chapter 3, System Requirements and Design, we proceed to describe the development of the system in cloud computing. Figure 4.1 shows the system architecture diagram as it results from its implementation. In section 4.1, "Implementation of Services in the Computing Cloud", the function and use of the services that structure the implementation system are analyzed, as shown in the diagram. Then section 4.2, "Implementation of Blockchain," analyzes how we created the blockchain, while section 4.2 comes and completes section 4.3, which analyzes the smart contracts of the architecture and how they contribute to the implementation of the system.

4.1 Implementation of Services in the Cloud Computing

The term Back-End describes the set of services that "run" in the computing cloud to process and respond to requests coming from end-user applications (Front-End). The system used the following technologies for the planning of cloud services:

- Node.js[20]: Node.js is a software (mainly server) development platform built on a Javascript environment. Node's goal is to provide an easy way to build scalable web applications. Unlike most modern network application development environments, a node process does not rely on multithreading but an asynchronous I/O communication model. Node.js offers many extensions through the well-known npm package management tool. The advantage it offers is that through the collection, it is easy to find a package with ready-made methods that implement the functionality we are looking for. An additional advantage is that extension packages are included in the code in a simple and fast way.
- Nest[21] is a framework for building efficient, scalable Node.js server-side applications. It uses modern JavaScript[22], is built with TypeScript[23] (preserves compatibility with pure JavaScript), and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming). Under the hood, Nest uses Express but also provides compatibility with a wide range of other libraries, like, e.g., Fastify, allowing for easy use of the numerous third-party plugins available. All services(Application logic, Auth, Aggregate data, Location, PEPs, WoT proxy) from architecture developed with NestJs.
- Google Cloud Platform[24]: is a suite of cloud computing services that runs on the same infrastructure that Google uses internally for its end-user products, such as Google Search, Gmail, Google Drive, and YouTube. We used the GCP to deploy and host the services on the cloud. We also host the blockchain infrastructure on the cloud.

4.1.1 Wallet

The wallet, as we mentioned in chapter 3, is a NestJS application developed by us. We need the functions that have the wallet, so a user/application/device be able to connect to the system. The wallet perform the following functions:

[20] https://nodejs.org/en/
[21] https://nestjs.com/
[22] https://www.javascript.com/
[23] https://www.typescriptlang.org/
[24] https://cloud.google.com/

- createPublicAndPrivateKeys(passphrase): Create the public & private key pair using a passphrase.
- login(passphrase): Connect to the wallet using a passphrase. The user cannot implement the following procedures without connecting to the wallet.
- issuesData(data): Sign information based on the private key with RSA. As an example, the data can be VC from an IoT, and the infrastructure owner needs to issue the VC with his/her private key and creates JWS..
- exportDID(): With this function can export the DID from that is stored in VC at field credentialSubject. To be able to export the DID document and VC. This function exists so that the user can send the necessary information to the network to identify the user's existence and validity.
- exportVC(): With this function can export VC.
- storeVC(VC): With this function the user can store the VC that he/her has received from Auth service. We developed the wallet to store more than one VC.

4.1.2 Authorization Service

Chapter 3 analyses the functionality and the operations that implement the Authorization(Auth) service. This service aims to identify and authorize the system's actors (users, IoT, applications) to access the rest of the services. The service is created with Nest JS. This chapter presents the API implementation necessary to enforce the service's functionality.

The REST API of the service is described below:

- Method : POST
  URL: localhost/register
  Payload:
  {
  “role”: “DEVELOPER”,
  “public_key”:”xsadfe93n3sfdmao3sj9j5”
  “passphrase”: “1234”,
  }
  Description: Request for registration in the system. The request returns the VC that gerated from Auth service.
- Method : POST
  URL: localhost/login
  Payload:
  {

```
“passphrase” : “123”,
“DID” : “did:iBot:213m31”,
"proof": {
    "type": "RsaVerificationKey2018",
    "created": "2022-02-25T14:58:42Z",
   "verificationMethod":"did:iBot:dak...id202ms21asld#key-1",
    "jws": "z3FXQjecWufY46yg5ab... L5n2Brbx"
  }

}
```
Description: Request for login to the system. The request return the JWT that generated from Auth service and the user needs it to request in any other service.

- Method : POST
  URL: localhost/generate-did
  Payload:
  ```
  {
  “public_key”: “asdee2wa..kljul”,
  “DID” : “did:iBot:213m31”
  }
  ```
  Description: Request to create a DID and a DID document. For example when an infrastructure owner want to create a DID for an IoT. The request return the DID.
- Method : GET
  URL: localhost/retrieve-did
  Description: Request to retrieve a DID document.
- Method : PUT
  URL: localhost/{DID}
  Payload:
  ```
  {
  “document”: {...}
  }
  ```
  Description: Update specific field from a DID Document.
- Method : DELETE
  URL: localhost/{DID}
  Description: Request to delete a DID document.

4.1.3 Guard NestJS

As the name suggests, it guards something against being accessible without permissions. Guards[25] are a common concept in most backend frameworks, whether provided by the underlying framework or coded by the developer. Nestjs simplifies protecting and safeguarding APIs from unauthorized or unauthenticated users. Every guard you use must implement the CanActivate interface. The CanActivate interface properties make it easy for developers to custom code their guard logic. On the other hand, a guard has access to the ExecutionContext instance and thus knows what is to be executed exactly after it. They are much like filters and pipes and can interpose the correct logic at the correct time in a Request-Response cycle. The Guard operates according to the Policy Enforcement Point Proxy specification in section 3.6. The implementation of the system includes two different types of guards. Guards that mediate between services and requests from system users require the user's JSON Web token in the request header. In this way, by communicating with the Auth service, the Guard confirms the identity and roles of the user to whom the token belongs.

In the communication scenario between system services, Guards require in the header of the requests they receive the secret Master key. As long as a request includes the correct Master key, the Guard forwards the communication to the service it mediates.

4.1.4 Application Logic

The application logic service is the central part of our system as it contains the code we developed to orchestrate the services and implement the functional requirements we have captured. We developed the service with NestJS to execute and route REST requests.

The user interface system forwards user requests using system functions through graphical interfaces to the application logic. The application logic receives the requests and routes them to the system services they intend to execute.

[25] https://docs.nestjs.com/guards

4.1.5 Aggregated data service

It is a service created in NestJS. It is connected to the blockchain. Through its RESTful interface, it serves the retrieval of Raw and Aggregated historical information stored in the blockchain. This information has been created and stored in the blockchain through the corresponding smart contracts every time a new value (properties/action updates) enters the system, as we saw in chapter 3. It will be further analyzed in the next section. Also, a user can retrieve information about the usage of the system (the number of requests to devices) or a device owner can retrieve the transactions of a device(the number of users that requested the device). The REST API of the service is described in the table below:

Table 4.1: HTTP endpoints for Aggregate data service.

GET	localhost/aggregate-data/{sensor_id}/{property}	{ “method”: [“max”], “period”:”hours” , “dateFrom”: “2022-03-12T22:00:00”, “dateTo”:”2022-03-12T23:00:00” }	We are retrieving grouped data of a sensor based on the time and the method we want. We can do more than one method.
GET	localhost/raw/{sensor_id}/{property}	{ “lastN” : 1 }	Retrieve the last N data for the sensors.
GET	localhost/raw/{sensor_id}/{property}/query	{dateFrom”: “2022-03-12T22:00:00”, “dateTo”:”2022-03-12T23:00:00” }	Retrieval of a sensor's data based on time.
GET	localhost/transaction/user/{did}		Retrieval of transactions made by a user with all the system's sensors.
GET	localhost/transaction/iot/{did}		Retrieval of the users' transactions with the specific

			sensor.

4.1.6 Location Service

It is a service created in NestJS. It is connected to the blockchain. Through its RESTful interface, it serves the retrieval of a device's location based on the device's DID or the retrieval of multiple sensors based on a geographic area given as input.

Table 4.2: HTTP endpoints for Location service.

GET	localhost/location/{sensor_id}/		Retrieve the location for a specific sensor.
GET	localhost/location/	{ “min_x” : 34.91, “max_x” : 35.67, “min_y” : 26.33, “max_y” : 23.51, }	Retrieve sensors located in the area is defined by the geographic coordinates we have given as limits. Return an array with DIDs

4.1.7 Queues

As mentioned in chapter 3, we need queues to keep the system safe from information loss, whether from the sensors or user requests that remain unanswered. For this reason, we use RabbitMQ between the service and the blockchain. In this way, all the queries from the services to the blockchain are stored in the queue and consumed by the blockchain at the rate it can process them without missing queries. A system that uses a queue consists of 3 elements:

- Producer: A producer is the one who sends (publishes) messages to a broker.
- Consumer: On the other hand, a consumer is the one who is listening and will receive the messages from the broker so it can handle the tasks in the background.
- Message Broker: Message broker that acts like queuing storage for API.

First, we create a RabbitMQ instance through the cloud that has a queue name and an AMQP URL that we will be using to send/receive messages from the queue.

Implementing producer
Producers are all the services that communicate with the blockchain through the gateway (which will be analyzed in the next section). Producers send/publish to the queue with a pattern that is the one that the consumer app will have to listen to and data that is the one that the consumer app will receive when it receives the message from the queue. We have four patterns depending on which queue we send to (IoT-proxy, registry-id, aggregated-data, location-data). Through the data, the services send the REST requests to the gateway, having sent the following information:

- The smart contract name
- The method
- and the attribute values of the methods

For example if we have a REST request from the WoT Proxy service of the form:
GET {wt}/properties/temperature
then to the queue with pattern iot-proxy will be sent:

- smart contract name : iot-proxy which is the name of the smart contract
- function name : retrieveProperty where is the name of the method that searches the last value for the corresponding property
- attr[] : {{wt}, temperature, {user DID}} where is a table with the attributes of the above method. {wt} is the id of the device that will be searched, while the temperature is the name of the property that wants to retrieve the value.

Implementing consumer

Consumer is the gateway that communicates with the blockchain, which listens to the above patterns and the messages that the above 4 queues have. In this way, no query from the services to the blockchain is lost.

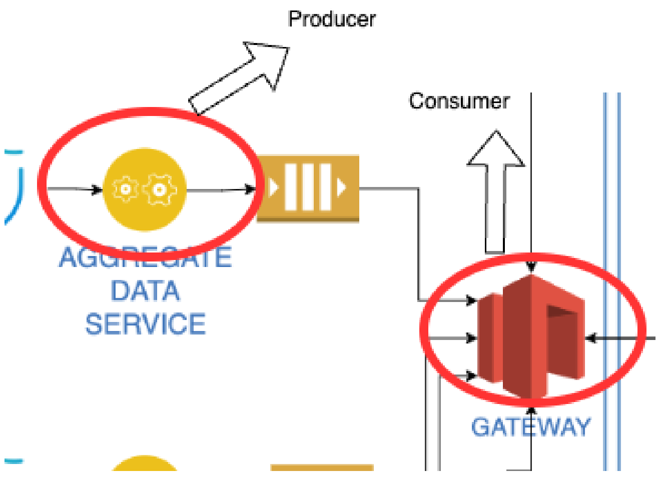
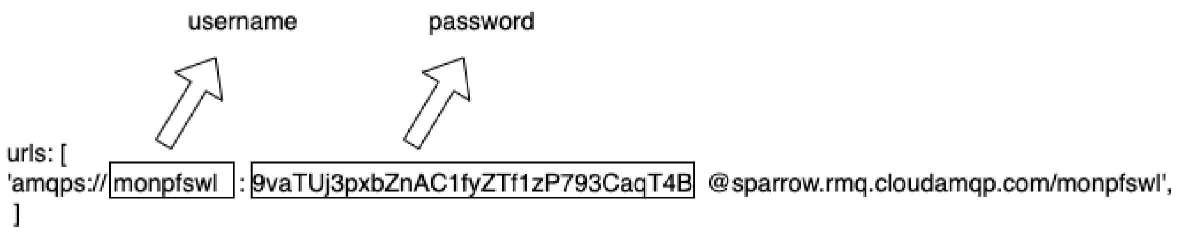


Figure 4.2: A producer and a consumer example on architecture.

Additionally, for security issues, each queue has a username and password so only services that know these details can send information to the queue.

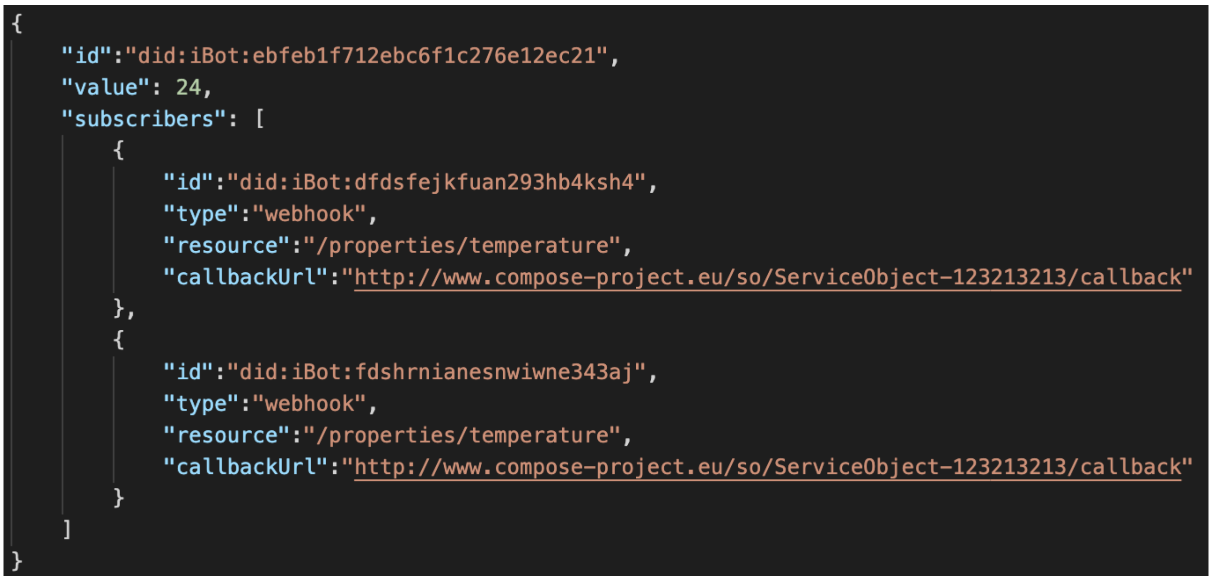


```
urls: [
'amqps://monpfswl:9vaTUj3pxbZnAC1fyZTf1zP793CaqT4B@sparrow.rmq.cloudamqp.com/monpfswl',
]
```

Figure 4.3: Example of the RabbitMQ link

4.1.8 Publish Subscribe Service

As mentioned in chapter 3, the service manages the changes in the devices and promotes the changes in the values of the devices to those who have registered. For this reason, we use the rabbit mq, which consists of the producer-consumer and the message broker, as mentioned in the previous section. The difference is that the producer listens to the changes made in the blockchain and sends the information to the queue based on these changes. Then the consumer consumes this information from the queue and must pass this information to the subscribers. In more detail, users subscribe to devices, and subscriptions are stored on the blockchain, as mentioned in a previous chapter. Then every time a value (property/action) of a device change, the blockchain creates an event with three values: the DID of the device, the value that changed, and the subscribers of the device.



```
{
    "id":"did:iBot:ebfeb1f712ebc6f1c276e12ec21",
    "value": 24,
    "subscribers": [
        {
            "id":"did:iBot:dfdsfejkfuan293hb4ksh4",
            "type":"webhook",
            "resource":"/properties/temperature",
            "callbackUrl":"http://www.compose-project.eu/so/ServiceObject-123213213/callback"
        },
        {
            "id":"did:iBot:fdshrnianesnwiwne343aj",
            "type":"webhook",
            "resource":"/properties/temperature",
            "callbackUrl":"http://www.compose-project.eu/so/ServiceObject-123213213/callback"
        }
    ]
}
```

Figure 4.5: The information that the blockchain sends to Pub/Sub service

The producer then connects to the blockchain with a gRPC connection. Then it registers the events with the pattern "events" created by the smart contract every time a new value is stored in the system in the format we mentioned above. The producer sends this event to the RabbitMQ queue we created with the "events" pattern. The specific queue has its AMQP URL and a separate username and password so only the specific producer can send data.

In turn, the consumer listens to the pattern "events" from the queue and reads all new incoming events. As mentioned earlier in the event's body, all system subscribers have the callback URL. Using these URLs, the consumer makes POST requests to these URLs with the new value of the sensor and the DID of the sensor.

The whole logic is presented in the image below.

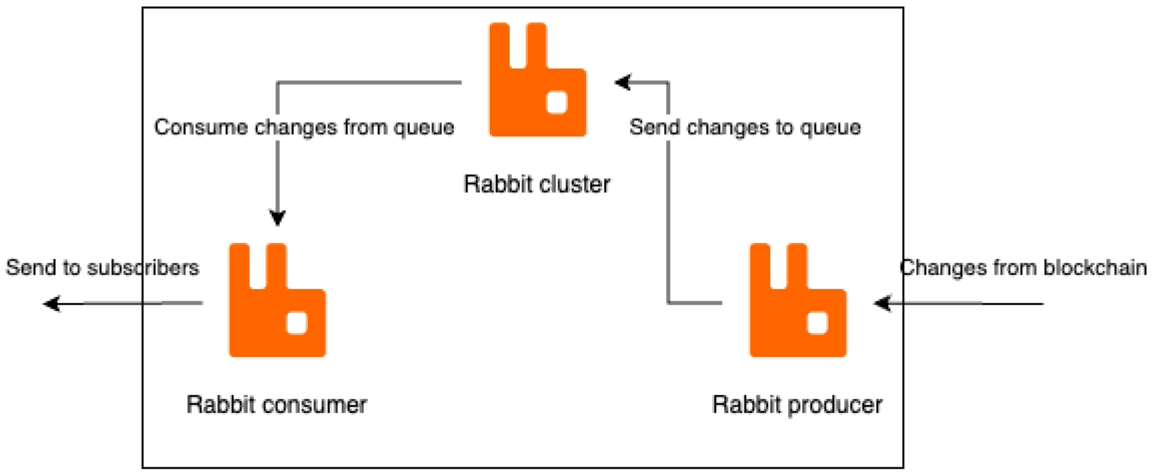


Figure 4.6: Architecture behind Pub/Sub mechanism with RabbitMQ

4.1.9 Gateway

It is a service that connects other services with the blockchain and is made with NestJS. Through this service, the various smart contracts are called so that the functions of the system and its services can be performed. In other words, this service is the translator of the REST requests received from the services into smart contract method calls so that the system services can retrieve or store information on the blockchain. Initially, this service has a wallet with public & private keys so that it can be connected to the blockchain with a gRPC[26] connection. Since the connection has been made with the blockchain, the service has the right to interact with the smart contracts and call its methods. As mentioned in 4.1.7, the service is also the consumer of the queues that the services send their requests. Since it reads the queries from the queues based on the pattern declared, the service decides which smart contract will call and which method, with the corresponding attributes completed as they have been sent to the queue. After a smart contract method call, the service forwards the response from the blockchain.

4.1.10 WoT Proxy

As mentioned in chapter 3 is an autonomous RESTful service in Python Flask[27] and implements some of the WoT Architecture operations on Things using HTTP. The WoT Proxy is designed to support Thing operations and uses the same JSON payloads, API endpoints, and response codes as described in the W3C Web Thing Model submission of W3C. Specifically,

[26] https://grpc.io/
[27] https://www.fullstackpython.com/flask.html

WoT Proxy supports all operations for retrieving and updating Thing descriptions and their properties and all Thing model operations. It implements functions that send a command to a Thing (i.e., an actuator) to execute or retrieve actions and action executions, as well as functions that create, retrieve, and delete subscriptions on Web Thing resources. Most of these operations (i.e., 14 out of 18) are used in the OpenAPI Thing template proposed in Section 3. More detailed, the specific operations are:
Web Thing Resource

- Retrieve a Web Thing (GET rootUrl/{wt})
- Update a Web Thing (PUT rootUrl/{wt})

Model Resource

- Retrieve the model of a Thing
- Update the model of a Thing

Properties Resource

- Retrieve a list of properties (GET rootUrl/{wt}/properties)
- Retrieve the value of a property (GET rootUrl/{wt}/properties/{property_name})
- Update a specific property (PUT rootUrl/{wt}/properties/{property_name})
- Update multiple properties at once (PUT rootUrl/{wt}/properties)

Actions Resource

- Retrieve a list of actions (GET rootUrl/{wt}/actions)
- Retrieve recent executions of an action (GET rootUrl/{wt}/properties/{action_name})
- Execute an action (POST {wt}/actions/{action_name})
- Retrieve the status of an action (GET {wt}/actions/{action_name}/{id})

Things Resource

- Retrieve a list of Web Things (GET {wt}/things)
- Add a Web Thing to a gateway (POST {wt}/things)

Subscriptions Resource

- Create a subscription (POST /subscriptions)
- Retrieve a list of subscriptions (GET /subscriptions)
- Retrieve information about a specific subscription (GET /subscriptions/{id})
- Delete a subscription (DELETE /subscriptions/{id})

4.1.11 OpenAPI Generator

In chapter 3 we mentioned the mechanism for generating OpenAPI description for Things. The diagram that describes the operation of the mechanism is given below:

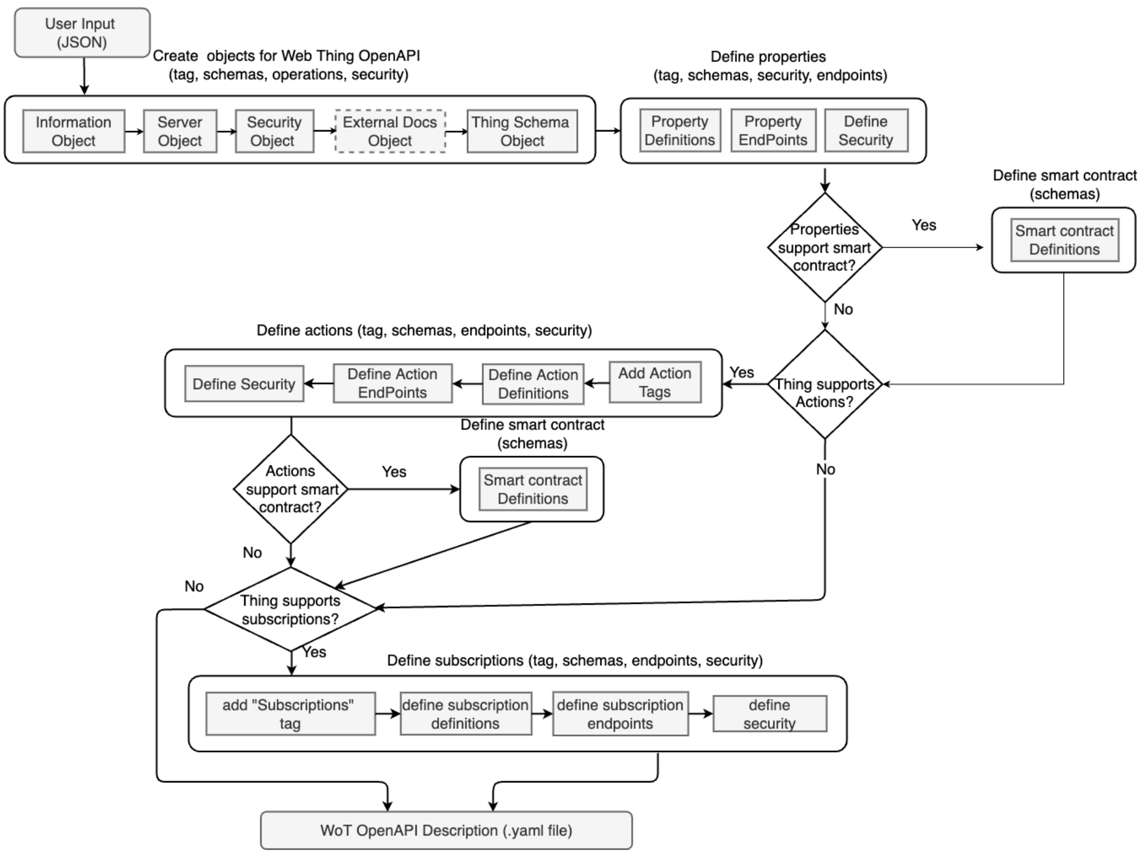
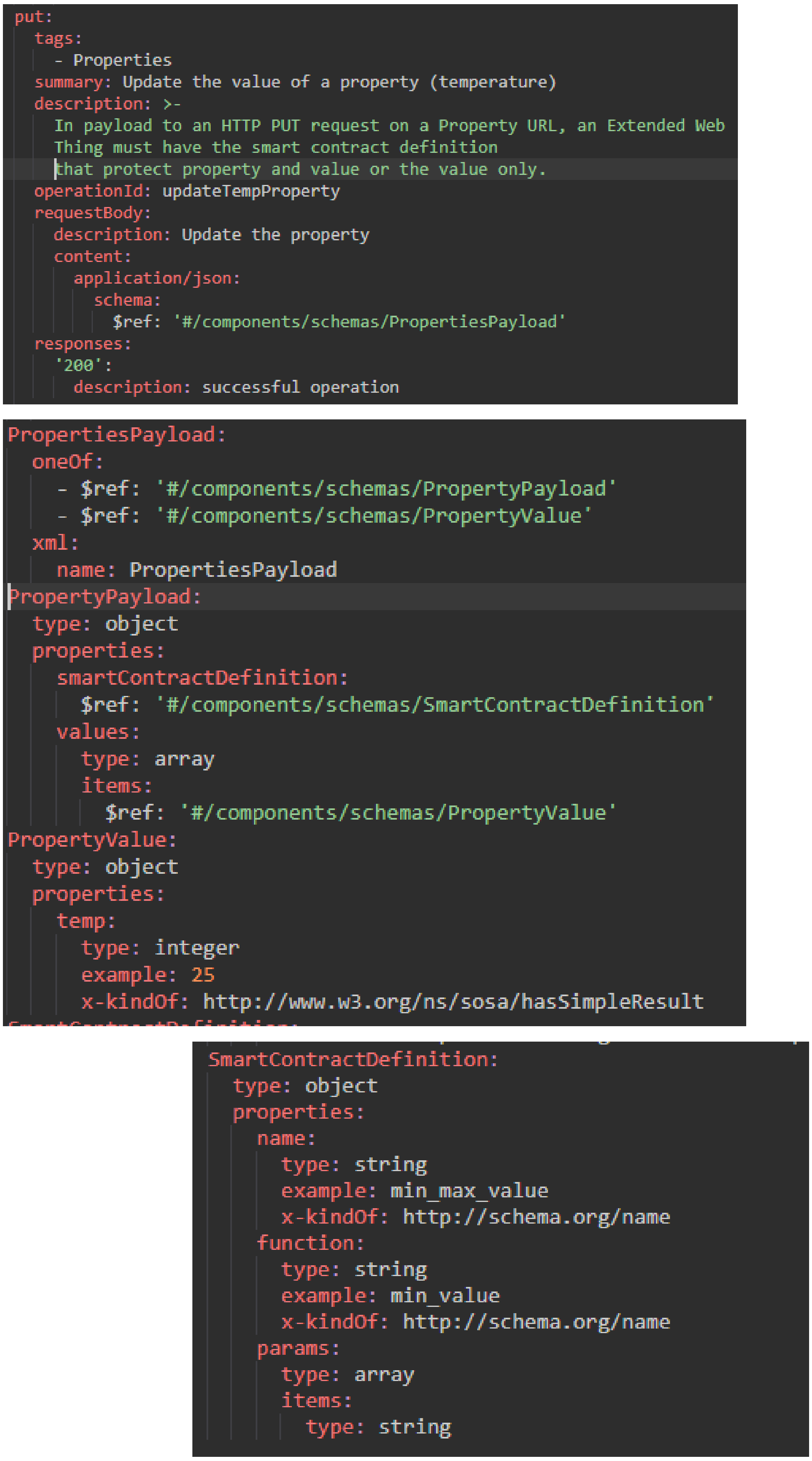

User Input (JSON)
Create objects for Web Thing OpenAPI (tag, schemas, operations, security)
Information Object
Server Object
Security Object
External Docs Object
Thing Schema Object
Define properties (tag, schemas, security, endpoints)
Property Definitions
Property EndPoints
Define Security
Properties support smart contract?
Yes
Define smart contract (schemas)
Smart contract Definitions
No
Thing supports Actions?
Yes
Define actions (tag, schemas, endpoints, security)
Add Action Tags
Define Action Definitions
Define Action EndPoints
Define Security
Actions support smart contract?
Yes
Define smart contract (schemas)
Smart contract Definitions
No
No
Thing supports subscriptions?
No
Yes
Define subscriptions (tag, schemas, endpoints, security)
add "Subscriptions" tag
define subscription definitions
define subscription endpoints
define security
WoT OpenAPI Description (.yaml file)

Figure 4.7: Generating an OpenAPI Thing Description

However, in this Thesis, we also present the security of the information of the devices within the smart contracts. The proposal is for the devices to declare to the endpoints that update the properties and actions of the smart contracts that control this information. According to this proposal, the description of the properties and actions should include the object containing the properties or actions definitions and the smart contract that can checks this information. For example, for a temperature sensor such as the DHT22, PUT/properties/temperature in the device description is done like this:



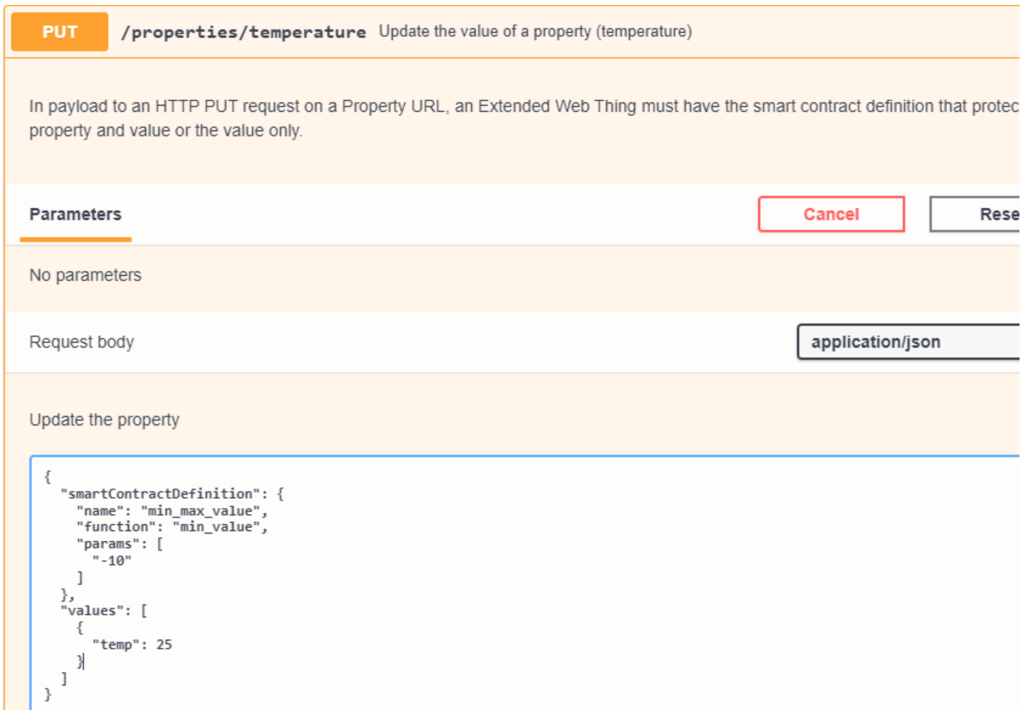
```
put:
  tags:
    - Properties
  summary: Update the value of a property (temperature)
  description: >-
    In payload to an HTTP PUT request on a Property URL, an Extended Web
    Thing must have the smart contract definition
    that protect property and value or the value only.
  operationId: updateTempProperty
  requestBody:
    description: Update the property
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/PropertiesPayload'
  responses:
    '200':
      description: successful operation
```

```
PropertiesPayload:
  oneOf:
    - $ref: '#/components/schemas/PropertyPayload'
    - $ref: '#/components/schemas/PropertyValue'
  xml:
    name: PropertiesPayload
PropertyPayload:
  type: object
  properties:
    smartContractDefinition:
      $ref: '#/components/schemas/SmartContractDefinition'
    values:
      type: array
      items:
        $ref: '#/components/schemas/PropertyValue'
PropertyValue:
  type: object
  properties:
    temp:
      type: integer
      example: 25
      x-kindOf: http://www.w3.org/ns/sosa/hasSimpleResult
```

```
SmartContractDefinition:
  type: object
  properties:
    name:
      type: string
      example: min_max_value
      x-kindOf: http://schema.org/name
    function:
      type: string
      example: min_value
      x-kindOf: http://schema.org/name
    params:
      type: array
      items:
        type: string
```

Figure 4.8: Example of an OpenAPI description that include smart contract definition.

In more detail, according to the description, the PUT/properties/temperature endpoint has also added the smartContractDefinition object to its payload. The smartContractDefinition contains the following fields name, function, and params. The name is the name of the smart contract that contains the function named function. Finally, the params field is an array of strings that declare the parameters that must be sent to the "function". The result is that the new endpoint is as follows:



In payload to an HTTP PUT request on a Property URL, an Extended Web Thing must have the smart contract definition that protec
property and value or the value only.

No parameters

Request body

application/json

Update the property

```
{
  "smartContractDefinition": {
    "name": "min_max_value",
    "function": "min_value",
    "params": [
      "-10"
    ]
  },
  "values": [
    {
      "temp": 25
    }
  ]
}
```

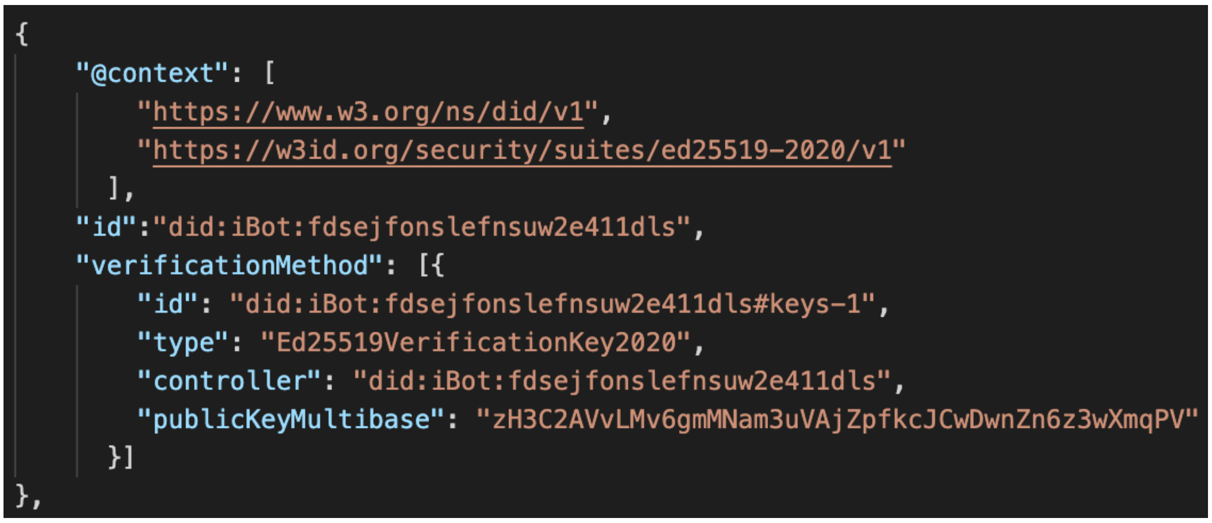
Figure 4.9: Example of the endpoint with smart contract definition for security

For whether a device description has the declaration of smart contracts in the properties or the actions, it is declared by the user in the input given as a JSON file.

4.2 Implementation of Blockchain

As mentioned in Chapter 3, the blockchain is one of the system's essential components because it stores all the important information the system needs to fulfill all the functional requirements. To implement the system, we used Hyperledger Fabric, an open-source blockchain platform managed by the Linux Foundation, as described in section 2. The Fabric network that makes up the blockchain deploys the smart contracts (which will be explained in the next chapter) using Typescript as programming language and Kafka[28] as a consensus mechanism. Previously we mentioned that the Fabric consists of the Ledger, which stores the transactions (each change or read in data) that take place in the Fabric, while it also consists of the World State – a database that holds current values of a set of ledger states. Ledger states are, by default, expressed as key-value pairs and frequently change, as states can be created, updated, and deleted. In this case, we use CouchDB[29] as the basis for the World State, which is also the default option from Fabric. In this section, we will analyze the way we store the data and the structure they have inside the World State to serve the functional requirements. Before analyzing the structure of the data we store, we should mention that each type of data has a unique id because the World State consists of a key-value pair. In contrast, the management and creation of the structures is done by the corresponding smart contracts that will be analyzed in another section. Below is the structure of all the data we use:

- DID Documents: as mentioned in the previous section, it consists of the elements id, controller, and public key.



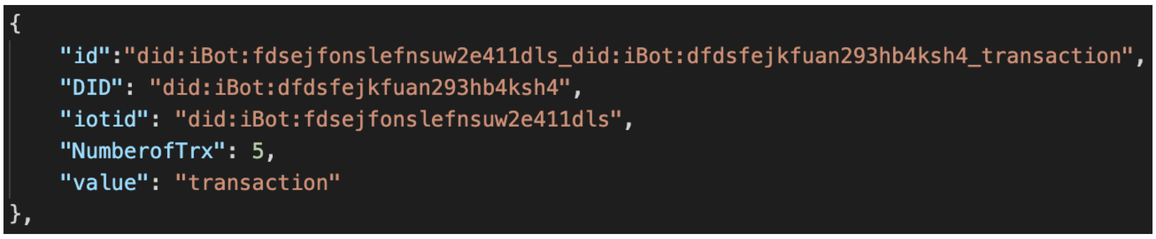
```
{
    "@context": [
        "https://www.w3.org/ns/did/v1",
        "https://w3id.org/security/suites/ed25519-2020/v1"
      ],
    "id":"did:iBot:fdsejfonslefnsuw2e411dls",
    "verificationMethod": [{
        "id": "did:iBot:fdsejfonslefnsuw2e411dls#keys-1",
        "type": "Ed25519VerificationKey2020",
        "controller": "did:iBot:fdsejfonslefnsuw2e411dls",
        "publicKeyMultibase": "zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
      }]
},
```

Figure 4.10: DID Example

- OpenAPI Descriptions: According to the concept of the system, each device has its OpenAPI description. In addition, the specific structure has the field description, which stores the full description of the device, and the field model, which stores the model of the device. The JSON that describes this structure consists of the id field, a string of the form {DID}_openapi. In this way, we create a unique id reason for the uniqueness of the DID, which is easy to retrieve every time someone requests the description of the device.

[28] https://kafka.apache.org/
[29] https://couchdb.apache.org/

- Transactions: In this structure, the user's billing is stored based on his use of a specific device. It consists of the following fields:
    - id: It is of the form {user DID}_{IoT DID}_transaction. With this form, a pair of the user and the device from which he requests information is created and is the unique reason for the uniqueness of the 2 DIDs. It is also easy to search once we know the 2 DIDs.
    - DID: is the DID of the user who created the request to retrieve a value from the specific sensor.
    - iotId: It is the device's DID that gave its data to the user.
    - NumberofTrx: It is the number that shows how many times the user used the sensors. For example, if a user calls the REST endpoint GET property/temperature of a temperature sensor, then the NumberofTrx number will increase by one. It is worth noting that if someone subscribes to a device, this number increases by one.
    - Value: It is the value that defines the type of transaction that took place. The two types are [subscription, transaction]



```
{
    "id":"did:iBot:fdsejfonslefnsuw2e411dls_did:iBot:dfdsfejkfuan293hb4ksh4_transaction",
    "DID": "did:iBot:dfdsfejkfuan293hb4ksh4",
    "iotid": "did:iBot:fdsejfonslefnsuw2e411dls",
    "NumberofTrx": 5,
    "value": "transaction"
},
```

Figure 4.11: Example of a Transaction object.

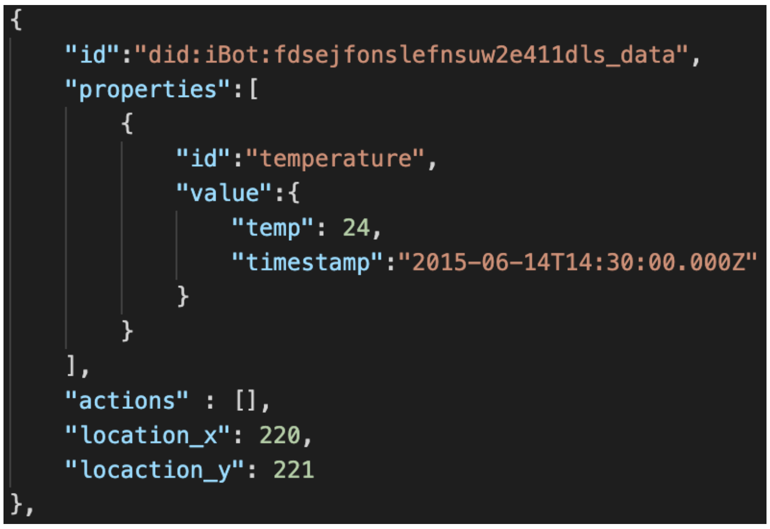
- Current Data: It is the data that shows the real-time value that the sensor has sent to the system, whether it is a change in a property or an action. The location of the sensor is also stored in this structure. In more detail, the structure consists of the following fields:
    - id: It is the unique identifier of the structure and has the format {DID}_data. {DID} is the DID of the device.
    - location_x: It is the global coordinate x for the position of the sensor.
    - location_y: It is the global coordinate y for the position of the sensor.
    - properties: It is the list of properties of a device (a device doesn't need to have only one property). The properties consist of two basic characteristics from the id, e.g., temperature, and the values , which consist of the current values of the sensor

and the timestamp of the moment they were last updated. An example of a temperature property is:

```
{
  "id":"temperature",
  "values":{
   "temp":22,
   "timestamp":"2015-06-14T14:30:00.000Z"
  }
 }
```

- actions: The specific field follows the same philosophy since it stores a device's actions. The difference in actions is that in addition to the id, values, and timestamp, they also consist of the status field that shows the status of the action. Example of an action:

```
{
 "id":"233",
 "value":{
  "delay":50,
  "mode":"debug"
 },
 "status":"executing",
 "timestamp":"2015-06-14T14:30:00.000Z"
}
```

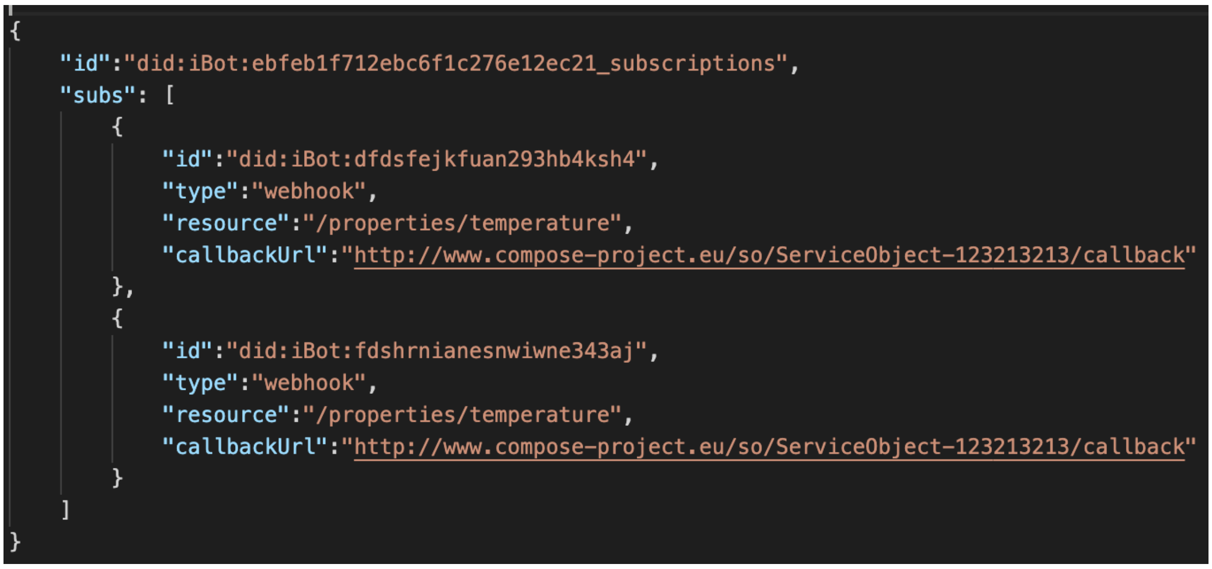


```
{
    "id":"did:iBot:fdsejfonslefnsuw2e411dls_data",
    "properties":[
        {
            "id":"temperature",
            "value":{
                "temp": 24,
                "timestamp":"2015-06-14T14:30:00.000Z"
            }
        }
    ],
    "actions" : [],
    "location_x": 220,
    "locaction_y": 221
},
```

Figure 4.12: Example of a Current Data object.

- Aggregate Data: As mentioned in chapter 3, statistical data shows the maximum/minimum/sum of a property of a sensor concerning different types of periods such as a month, a day, and an hour. In more detail, the structure consists of the following fields:
  - id : It is the unique identifier of the structure created to describe the aggregated data for the specific type of time duration. The format describes the id
  - {IoT DID}_{property name}_{aggregated period}_{timestamp}
  - Where {IoT DID} is the DID of the device, {property name} is the property we want to request, such as temperature, {aggregated period} describes the period we want to report that the specific structure shows (hours/day/month). In contrast, {timestamp} shows the time when the specific structure was created. {timestamp} depends on {aggregated period} because if {aggregated period} has the value hours then {timestamp} has the value 2022-03-12T22 if it is day it has the value 2022-03-12 while if it has the value month then {timestamp} is 2022-03.
  - max_properties: As its name states, it keeps the maximum value of the sensor throughout its operation. For example, if the temperature sensor has sent the values 40, 35, and 22, it will keep the value 40.
  - min_properties: Keeps the minimum value of the sensor throughout its operation.
  - sum_properties: Holds the sum of all values sent by the sensor.
  - dateFrom: The value indicates the beginning of the time field to which the structure belongs. For example, if the value was entered into the system at 2022-03-12T22:30:00 and the {aggregated period} is hours, the specific field will show 2022-03-12T22.
  - dateTo: Correspondingly, as above, it shows the end of the field to which the structure belongs. For example, if the value was entered into the system at 2022-03-12T22:30:00 and the {aggregated period} is hours, then the specific field will show 2022-03-12T23.
- Subscriptions: This structure stores developer subscriptions made to sensors. It consists of the following fields:

- id : This field follows the id logic of the OpenAPI description. That is, its form is of the form {DID}_subscriptions with {DID} denoting the identifier of the device.
- subs: Which field is a table consisting of other JSON of the format id (the DID of the developer), type (subscription type, e.g., webhook), resource (in which property the user subscribed), e.g.,/properties/temperature), callback URL (which is the URL that the developer has declared to receive feedback from each change made to the sensor).



```
{
    "id":"did:iBot:ebfeb1f712ebc6f1c276e12ec21_subscriptions",
    "subs": [
        {
            "id":"did:iBot:dfdsfejkfuan293hb4ksh4",
            "type":"webhook",
            "resource":"/properties/temperature",
            "callbackUrl":"http://www.compose-project.eu/so/ServiceObject-123213213/callback"
        },
        {
            "id":"did:iBot:fdshrnianesnwiwne343aj",
            "type":"webhook",
            "resource":"/properties/temperature",
            "callbackUrl":"http://www.compose-project.eu/so/ServiceObject-123213213/callback"
        }
    ]
}
```

Figure 4.13: Example of a Subscription object.

4.3 Implementation Of Smart Contracts

From a developer's perspective, a smart contract and the ledger form the heart of a Hyperledger Fabric blockchain system. Whereas a ledger holds facts about the current and historical state of a set of business objects, a smart contract defines the executable logic that generates new facts that are added to the ledger. To create smart contracts, we must define a common set of contracts covering common terms, data, rules, concept definitions, and processes. Taken together, these contracts lay out the business model that governs all of the interactions between transacting parties. We can turn these contracts into executable programs – known in the industry as smart contracts – to open up a wide variety of new possibilities. We have separated the smart contracts we created into three categories according to the business rules. The three categories are as follows:

- Smart contracts deal with the retrieval/updating of system information such as properties, action, location, subscriptions, OpenAPI descriptions, transactions, and aggregated data of applications.
- The smart contracts that deal with the control of the information(properties & actions updates) sent by the devices to the system must be checked if they are correct according to the specifications of each device.
- The smart contracts deal with the retrieval and creation of Decentralized Identifiers.

4.3.1 Smart Contracts for IoT security

An essential part of the implementation we present is the automated security we offer when a device sends its information to the system. As it has been emphasized, to write information on the blockchain, it must be done through a smart contract. Taking advantage of this logic, we can define different types of smart contracts with sensors' values as an argument and check if these values are correct according to the "terms" of the contract. Three smart contracts are currently installed on the blockchain and implement methods that control device prices and are responsible for the system's security from malicious devices. In more detail, the three smart contracts are as follows:

- min_max_value: The specific smart contract consists of 3 methods the min_value, the max_value, and the min_max_value.
    - min_value: The specific method has as a parameter the value of the sensor that it sends to the system, while it also accepts a limit as a second parameter. Then the method checks if the value is smaller than the value of the limit given to it as input; if it is, then it returns true; else, it returns false. We use this method's answer to decide whether the information should be written to the blockchain or not. When the answer is true, then the information is invalid.
    - max_value: The specific method is similar to the one above, with the only difference that it checks if the value given by the sensor is greater than the limit also given as a parameter. If the value is greater, it will return true; if it is not, it will return false. We use this method's answer to decide whether the information should be written to the blockchain or not. When the answer is true, then the information is invalid.

    - min_max_value: This method uses the previous two methods. It has three parameters, one is the value of the sensor, and the other 2 are the min/max limits. Then it calls the other two functions with the min and max limits, respectively. If one of the 2 returns true, it will also return true; otherwise, it will return false. When the answer is true, the information is invalid as it is easy to understand according to the two functions above.
- payload_frequency: This smart contract aims to check if a device sends too frequent requests (which is sometimes a malicious action by the device). The contract has a method:
    - check_freq: The method has three parameters; the first 2 are timestamps, the first being the timestamp when the device sent the new value, while the 2nd was the timestamp of the most recent value sent by the device. The third parameter is time in milliseconds. If the subtraction of the two timestamps gives a number smaller than that of the 3rd parameter, then the function returns true, and the information is considered malicious.
- payload_size: With this smart contract, we check if the size of the information sent by the device is larger than allowed. The contract has a method:
    - check_size: The method has two parameters, one is the size (in bytes) of the value sent by the device, and the other is the limit. If the size is greater than the limit, then it returns true. The information is invalid and will not be saved when the answer is true.

Finally, it is worth noting that in the blockchain, we can add other smart contracts that will check other types of properties/actions according to the requirements of the sensors that will be added to the system. Smart contracts can currently only be added by the blockchain admin.

4.3.2 Smart Contracts for Data storage

As mentioned above, the specific smart contracts deal with the retrieval/updating of system information. Specifically, there are three smart contracts, each of which processes one or more data types. In more detail, each smart contract will be analyzed below:

- aggregated-data-contract: This smart contract implements the methods we need to create the aggregated data or retrieve it according to the endpoints of the aggregate data service. More specifically, the methods implemented by the specific contract are:

  - createAggregateData: The parameters we give to this function are the DID of the sensor, the value of the property of the sensor, and the property's name. Then the 3 different ids are created based on the format {IoT DID}_{property name}_{aggregated period}_{timestamp} for each different aggregated period (hours/day/month). If the id exists, it updates the fields max_properties, min_properties, sum_properties, avg_properties with the property's new value. If the id does not exist, then it creates a new structure with this id and initializes the max_properties, min_properties, sum_properties, avg_properties fields, and additionally initializes the dateFrom and dateTo fields.
  - readAggregateData: The parameters we give to this function are the DID of the sensor, property name, aggregated period (hours/day/month), aggregate method, dateFrom, dateTo. Based on the parameters, we create one or more IDs and look them up in the blockchain. For example if we have given aggregated period=hours and dateFrom=2022-03-12T21:00:00 and dateTo=2022-03-12T23:00:00, then the generated ids are 2 {IoT DID}_{property name} _hours_2022-03-12T21 and {IoT DID}_{property name}_hours_2022-03-12T23. Finally, the value of the requested aggregate method is returned.
- transaction-contract: The specific smart contract implements the methods we need to create transactions every time someone asks for some system information, as mentioned in the previous section. More specifically, the methods implemented by the specific contract are:
  - update_transaction: The parameters we give to the specific function are the DID of the sensor, DID of the user, and the value. Then it creates the id in the form {user DID}_{IoT DID}_transaction. Suppose the structure with the specified id does not exist. In that case, it creates this structure by initializing the iotid and DID fields with the corresponding data from the parameters and sets NumberofTrx = 1. End sets value = transaction | subscription (the function parameter). On the other hand, if the type structure exists, then the NumberofTrx increases by one.
  - retrieve_transaction: The parameters of this one are a DID and a boolean variable that indicates whether the DID is an IoT

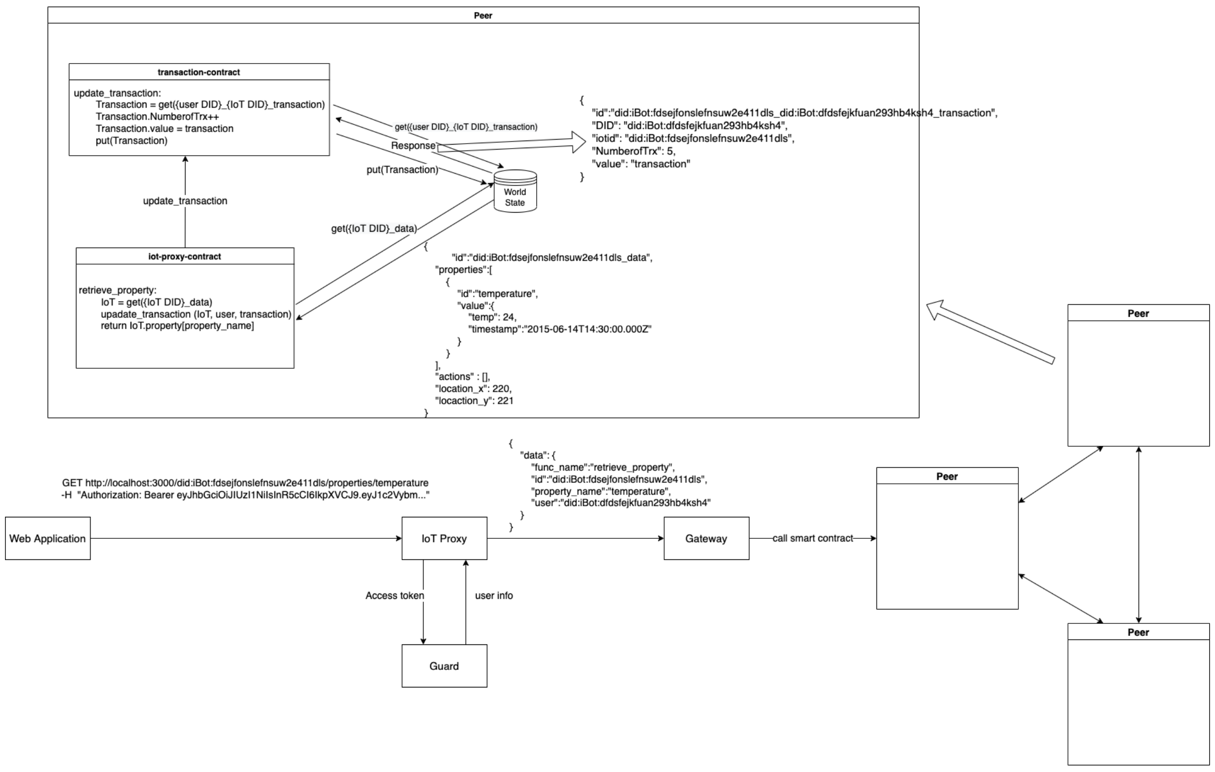
device or not. If it is, it returns all the structures with the specific identifier as iotId, with the result that a list of the transactions of all users with this device is returned. On the other hand, if the parameter DID is a developer DID, it will return a list of the transactions concerning the user and all the devices he has used.

- IoT-proxy-contract: This smart contract implements the functions that serve the needs of the WoT proxy and its endpoints. As a result, the specific smart contract has access to the current data, OpenAPI devices, and Subscriptions. As mentioned earlier, the functions in question serve the WoT proxy endpoints, for example, there are the following functions:
    - retrieve_properties: In the specific function, we give the DID of the sensor. Then it finds the structure with the id with the key value {DID}_data and returns the list of properties with the values.
    - retrieve_property: With this method we give the DID of the sensor we want to retrieve and the property's name (e.g., temperature). It then finds the structure with the id with the key value {DID}_data and returns from the property list the property with id temperature. It then calls update_transaction from the transaction-contract smart contract to update the amount of use the user made to the system.
    - update_properties: With the specific method, we give the DID and the JSON with the changes to the properties we want to change. We also provide the details (smart contract name, method name, and prices) for the smart contract that checks if the prices are correct so they can be stored in the system.
    - update_property: With the specific method, we give the DID and the JSON with the change to the property we want to change. We also provide the details (smart contract name, method name, and prices) for the smart contract that checks if the price is correct so the contract can store the value in the system. Once the smart contract that checks the value we want is called, we store the value, while in parallel, we call the smart contract that stores the aggregate data.
    - retrieve_actions: It follows the same logic as retrieve_properties where we give the DID of the sensor from which we want to retrieve the list of actions. Then it finds the structure that has the id with the value {DID}_data and returns

the list of actions. It then calls update_transaction from the transaction-contract smart contract to update the amount of use the user made to the system.

- retrieve_action: It follows the same logic as retrieve_property; we give the DID of the sensor from which we want to retrieve and the name of the action (e.g., upgrade firmware). Then it finds the structure with the id with the value {DID}_data and returns from the action with the id upgrade firmware with the most recent actions from the list of actions. It then calls update_transaction from the transaction-contract smart contract to update the amount of use the user made to the system.
- retrieve_action_status: In the specific function, we give the DID of the sensor from which we want to retrieve the name of the action (eg, upgrade firmware) but also the id (e.g., 233) of the action to be able to retrieve the status of the specific action. Then it finds the structure that has the id with the value {DID}_data and returns from the list of actions the action with id upgradefirmware and in the list of the most recent actions, it is selected with the id we gave it to return the status.
- update_action: In the specific function, we give the DID of the sensor and the name of the action (eg upgradefirmware) from which we want to send a new action. Also, along with the previous elements, we take the data we want to store for the specific action in the function.
- retrieve_model: As parameters to the function, we give the DID of the sensor, and then it finds the structure with the id {DID}_openapi and returns the model, which is a JSON that describes the properties and actions that can be performed on Things.
- update_model: As in retrieve_model we give the DID of the sensor. Then it finds the structure with the id {DID}_openapi and updates the model field with the model object given as a parameter to the function.
- retrieve_thing: The parameters of the function are the DID of the sensor. If the structure with the id {DID}_openapi is found, it returns the thing description.
- update_thing: The parameters of the function are the DID of the sensor and the thing object. Once the structure with the id {DID}_openapi is found, it updates the thing description field.

- create_sub: The function parameters are the DID of the sensor and the subscription object that has the subscription elements. Then it retrieves the structure with id {DID}_subscriptions and completes the elements mentioned in the previous section based on the subscription object given as a parameter and stores them in the list of subscriptions that the specific structure has. It then calls update_transaction from the transaction-contract smart contract to update the amount of use the user made to the system.
- retrieve_sub: The parameters of the function are the DID of the sensor. Then it retrieves the structure with id {DID}_subscriptions and returns the list of subscriptions.
- delete_sub: The function's parameters are the DID of the sensor and the DID of the user who has registered to the specific sensor. Then retrieve the structure with id {DID}_subscriptions, and from the list of subscriptions, it is deleted with id (DID) that the user given as a parameter.
- init_device: The parameters given to this function are the sensor's DID and the device's coordinates. Once the DID is given, it creates and initializes all the related structures such as {DID}_data, {DID}_subscriptions, {DID}_openapi, and also stores the location of the device in the {DID}_data structure.
- retrieve_location: The parameters given to this function are the DID of the sensor, and if the {DID}_data structure is found, it returns the device's location.



```
update_transaction:
    Transaction = get({user DID}_{IoT DID}_transaction)
    Transaction.NumberofTrx++
    Transaction.value = transaction
    put(Transaction)
```

```
{
  "id":"did:iBot:fdsejfonslefnsuw2e411dls_did:iBot:dfdsfejkfuan293hb4ksh4_transaction",
  "DID": "did:iBot:dfdsfejkfuan293hb4ksh4",
  "iotid": "did:iBot:fdsejfonslefnsuw2e411dls",
  "NumberofTrx": 5,
  "value": "transaction"
}
```

```
retrieve_property:
    IoT = get({IoT DID}_data)
    upadate_transaction (IoT, user, transaction)
    return IoT.property[property_name]
```

```
{
    "id":"did:iBot:fdsejfonslefnsuw2e411dls_data",
  "properties":[
    {
      "id":"temperature",
      "value":{
        "temp": 24,
        "timestamp":"2015-06-14T14:30:00.000Z"
      }
    }
  ],
  "actions" : [],
  "location_x": 220,
  "locaction_y": 221
}
```

```
GET http://localhost:3000/did:iBot:fdsejfonslefnsuw2e411dls/properties/temperature
-H  "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm..."
```

```
{
  "data": {
    "func_name":"retrieve_property",
    "id":"did:iBot:fdsejfonslefnsuw2e411dls",
    "property_name":"temperature",
    "user":"did:iBot:dfdsfejkfuan293hb4ksh4"
  }
}
```

Figure 4.14: Flow when a developer requests for a temperature property.

4.3.3 Smart Contract for Decentralized Identifiers

As mentioned in previous sections, each user (user/developer/infra owner/IoT device/external application) has their wallet with its elements such as PKI and VC. In addition, the user also has a DID Document, which is stored in the blockchain, and its purpose is to keep the user's public key, thus also showing its validity (if it exists in the blockchain, it cannot be disputed). To be able to share the information of the DID Document and to be able to satisfy the needs of the registry id service, we have implemented the corresponding smart contract. In more detail, the smart contract is called did-document-contract and consists of the following functions:

- createDid: It is the function that initializes and creates the DID Document in the blockchain. The function accepts as a parameter a DID Document object which has all the elements for the initialization of the DID Document such as the id (DID), the controller (same DID as id or another) and publicKey. If the function checks that there is no other structure with the specific DID in the blockchain, it stores the structure in the system and returns the DID Document.
- readDid: The function accepts a DID as an argument. It then searches the blockchain for the DID, and if it finds it, then returns the DID Document that had the DID as an id.

- updateDid: The function accepts a DID and a DID Document object as an argument. Then it uses the id from the DID Document object to search to find if the DID Document exists by calling readDid with the id parameter. After readDid returns a DID Document, it checks if the controller is the same as the DID given as a parameter to the function. If this restriction is also satisfied, we replace the DID Document with the new DID Document object given as input.
- deleteDid: The function accepts a DID as input. Then it searches the blockchain for the DID; if it finds it, it deletes it from the blockchain.
- retrievePyblicKey: The function accepts a DID as an argument. Then it calls readDid with the DID parameter and returns the public key from the DID Document returned by readDid.

5. Backend Analysis & Comparisons with related works

5.1 Backed Analysis

The infrastructure of the implemented system is in the
google cloud, a google service that allows developers to host their services. The machine that housed the infrastructure consists of the following characteristics:

CPU	x86_64@3.8Ghz
Memory	8GB
HDD	256GB
OS	ubuntu 18.04

To determine the system's performance in real conditions, the Jmeter tool was used, which can create several simultaneous requests. Through Jmeter[30], it is possible to create them load conditions in each service of the system separately, defining it number of requests it will be asked to serve as well as how many of them will be executed simultaneously. Each

[30] https://jmeter.apache.org/

experiment follows the same pattern by including 2000 requests to the considered function of each system service, with the requests repeated each time with a different concurrency (Number of Concurrent Requests). The metrics refer to the average service time per request and are divided into categories according to the concurrency at which the requests were sent. These categories are per-one, per-hundred, per-two-hundred, per-five-hundred (For example with one-by-one synchronization the requests are sent consecutively, with fifty-one synchronization they are sent consecutively in groups of fifty requests).
It is also worth noting that before starting the experiments, we initialized the blockchain, having connected to the system 10 thousand sensors with random locations. One thousand users are connected and saved together.

5.1 Experiment 1.

Scenario: A developer searches for sensors based on a geographic area where he wants to see which sensors belong to that area.
Services: The request is routed from the Web Application to the application logic. The application logic routes the request to the Location Service. The request is translated into query syntax in a smart contract that the Gateway will call once it gets the query from the queue. Then the query goes to the corresponding smart contract, which selects the sensors (from the structures with the id being of the form {DID}_data) where the location_x and location_y values are within the geographic area given as input. The smart contract then returns a list of sensors that meet the specified criteria.
Details: The request considered in the experiment is about finding sensors that belong to a specific range of geographic coordinates. We requested over 10 thousand sensors, and 3000 of them meet the criteria for the input we gave.

```
REST: GET /location/
{
“min_x” : 34.91,
“max_x” : 35.67,
“min_y” : 26.33,
“max_y” : 23.51,
}
```

5.2 Experiment 2.

Scenario: A developer searches for aggregate temperature sensor data for a 24-hour period, wanting to know the maximum temperature every hour within this interval.
Services: The request is routed from the Web Application to the application logic. The application logic routes the request to the Aggregate Data Service. The request is translated into query syntax in a smart contract that the Gateway will call once it gets the query from the queue. Then the query goes to the corresponding smart contract which selects the structures with the form {IoT DID}_{property name}_{aggregated period}_{timestamp} where {property name}= temperature ,{aggregated period}=hours and {timestamp} takes the values from 2022-03-12T00 ... to 2022-03-12T23 . Then the smart contract returns a list of the maximum temperature values stored in the max_value field.
Details: The request considered in the experiment concerns finding maximum temperature values within 24 hours. We requested data collected by a sensor for two days.
REST: GET /aggregate-data/{sensor_id}/temperature

```
{ “method”: [“max”],
“period”:”hours”,
“dateFrom”: “2022-03-12T00:00:00”,
“dateTo”:”2022-03-12T23:00:00”
}
```

5.3 Experiment 3.

Scenario: A developer searches for the latest price of a temperature sensor.
Services: The request is routed from the Web Application to the application logic. The application logic routes the request to the WoT proxy. The request is translated into query syntax in a smart contract that the Gateway will call once it gets the query from the queue. Then the query goes to the corresponding smart contract (iot-proxy-contract), which selects the structure with the format {IoT DID}_data. Then the smart contract returns the value of the property requested by the user (in this case, temperatures).

Details: The request considered in the experiment is to find the most recent temperature value of the requested sensor. We requested over 10 thousand sensors that are connected to the system.
REST: GET /{sensor_id}/properties/temperature

5.4 Experiment 4.

Scenario: A user (developer) subscribes to a sensor to be able to receive information on every change in the sensor's values.
Services: The request is routed from the Web Application to the application logic. The application logic routes the request to the WoT proxy. The request is translated into query syntax in a smart contract that the Gateway will call once it gets the query from the queue. Then the query goes to the corresponding smart contract (iot-proxy-contract), which selects the structure with the format {IoT DID}_subscriptions. Then the smart contract adds to the subs field (it is a list of records) the new record sent by the user.
Details: The request considered in the experiment is about creating a developer account on a specific device. We requested over 10 thousand sensors that are connected to the system.
REST: POST /{sensor_id}/subscriptions

```
{
        "id":"did:iBot:dfdsfejkfuan293hb4ksh4",
        "type":"webhook",
        "resource":"/properties/temperature",
"callbackUrl":"http://www.compose-project.eu/so/ServiceObject-12321321
3/callback"
 }
```

5.5 Experiment 5.

Scenario: A sensor wants to send information to the system to update its value.
Services: The request is routed by the WoT proxy. The request is translated into query syntax in a smart contract that the Gateway will call once it gets

the query from the queue. Then the query goes to the corresponding smart contract (iot-proxy-contract), which sends the information to the smart contract (min_max_value), which checks the value of the sensor if it is valid. Then if the value is valid, the smart contract (iot-proxy-contract) finds the structure with the form {IoT DID}_data and updates the value of the properties field while simultaneously sending the value to the smart contract (aggregated-data-contract) to update the aggregated data.
Details: The request considered in the experiment concerns updating the value of recent properties of a device. We requested over 10 thousand sensors that are connected to the system.
REST: PUT /{sensor_id}/properties/temperature

```
{
  "smartContractDefinition":{
    "name":"min_max_value",
    "function": "min_value",
    "params":[
      "-10"
    ]
  },
  "values":[
    {
      "temp": 25
    }
  ]
}
```

5.6 Experiment 6.

Scenario: A user wants to register in the system and create the necessary elements to connect to the system in the future.
Services: The request is routed from the Web Application to the application logic. The application logic routes the request to the Registry Id service. The request is translated into query syntax in a smart contract that the Gateway will call once it gets the query from the queue. Then the query goes to the corresponding smart contract (did-document-contract). Then it creates the corresponding structure and stores it in the system.
Details: The request considered in the experiment concerns the creation of a DID Document for a builder. We requested over 1000 users who are connected to the system.
REST: POST /register

```
{
“role”: “DEVELOPER”,
“public_key”:”xsadfe93n3sfdmao3sj9j5”
“passphrase”: “1234”,
}
```

Before reporting the results of the experiments, it should be emphasized that the experiments can be divided into three main categories. These categories are as follows:

- The category [Write] is where the experiments write some information to the blockchain, such as experiments 4,5, and 6. The grouping is done as long as we have the same number of data in all three experiments and the information is written in the same database (blockchain), then the results are almost the same.

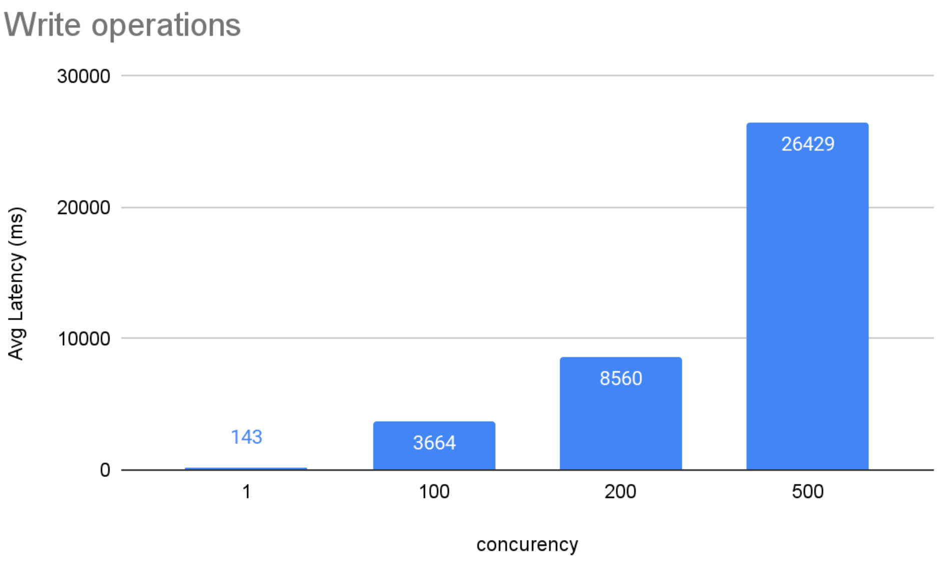


Figure 5.1: Time diagram for write operations.

- The category [Read] is where the experiments read some information from the blockchain, such as experiment 3.

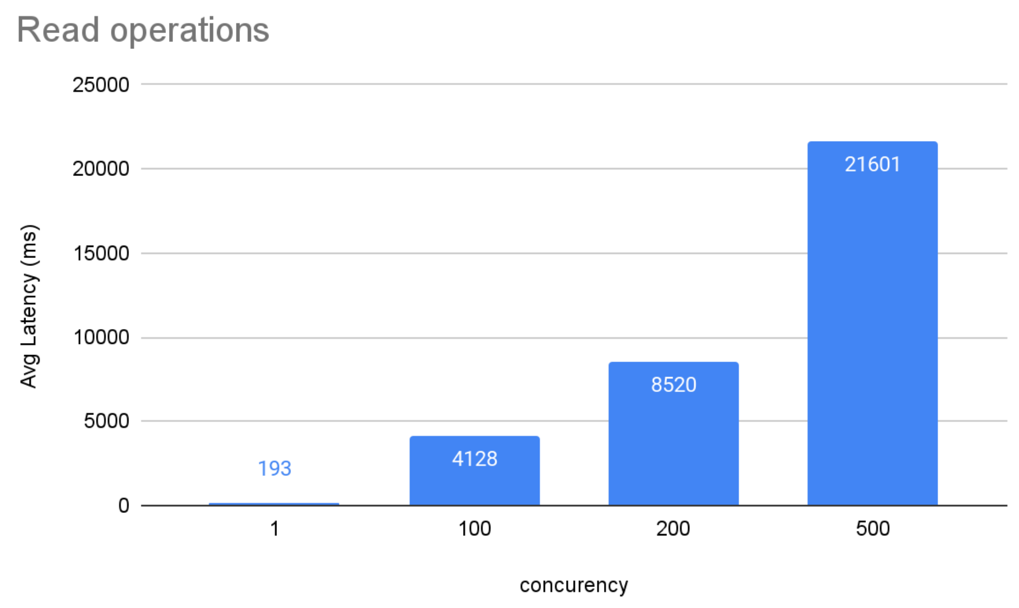


Figure 5.2: Time diagram for read operations.

- The category [Range Query] is the category that the experiments that read and return a volume of data that meets some criteria, such as experiments 1 and 2. Also, the grouping is done as long as we have the same number of data in both cases.

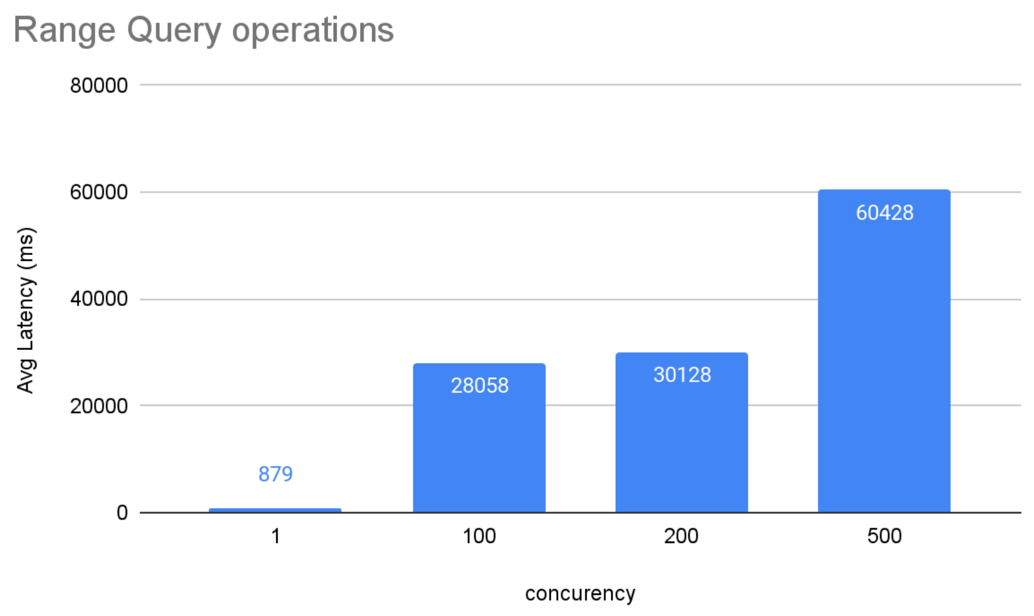


Figure 5.3: Time diagram for range query operations.

5.2 Comparisons with related works

The current implementation, as mentioned earlier, is an attempt to create a platform that tries to preserve and protect the data of IoT devices and the users who preserve the platform. To achieve something like this, we used blockchain technology and already existing technologies used in IoT frameworks & platforms. In this section, we compare the system presented in chapters 3 & 4 with related works starting with the use of blockchain in conjunction with IoT presented in section 2.9. This particular comparison is divided into six parts. It is worth emphasizing that the system that we present was created based on the functional and non-functional requirements presented in section 3.2. None of the related works supports all these requirements together.

User / IoT protection
As previously mentioned, iBot system uses DID and VC to secure users/IoT and their data/property. As we can see in IoTex, the users/devices store their connection data in DIDs, so no centralized base has this data. The paper [10] mentions that the users have a public and private key where the blockchain can recognize if the user's request is valid based on the specific public key. The devices interacting with the system create a public and private key pair with the corresponding logic. The other two works [3] and [9] do not mention how they manage the information of their users/devices.

User Storage
As is logical, a complementary role in protecting users/devices is also where the respective information is stored. In our proposal, we use DID and VC, as previously mentioned. Based on this, VCs are stored locally in a wallet, while DIDs are stored on the blockchain. With this logic, the DIDs used in IoTeX are also stored in the blockchain. Then in the following works [3],[9],[10] that use only one wallet with a public and private key, the private key is stored locally in a wallet while the public key is known throughout the blockchain.

Data protection
One of the most basic criteria for comparing architectures is the protection that each implementation provides for the data of the devices connected to

it. In our proposal, the data is protected within the smart contracts since access to the data is made only through the calls of the smart contracts. There is also the possibility of checking the value of a device first written in the system. In contrast, IoTeX provides a kind of encryption to protect the data. On the other hand, the works [10], [9], [3] provide security to the data of the devices through smart contracts, so the smart contracts are responsible for deciding whether the requester can have access to the data they are requesting. In work [3], data encryption is supported as an additional security layer.

Data storage

Equally important in the comparison is where the data produced by the devices is stored. In the present implementation, the data is stored in the blockchain, while data stored in IoTeX have a corresponding treatment. The work [3] has stored the devices' data in the cloud using the blockchain to access the data and not as a data storage. Correspondingly, the work [9] has the data stored in a local network and, like [3], uses the blockchain to access the data. A special case is a work [10] that stores the data in a Distributed Hash Table, thus keeping the decentralized character.

Indexing

Based on the previous part (Data storage), an essential element of comparison is whether the storage part can provide a function for indexing the data. In iBot we use Hyperledger Fabric which uses CouchDB to store system data. Since CouchDB supports data indexing, we can create indexes for the data produced in iBot. On the part of IoTeX, it can support data indexing through The Graph[31] protocol. Then the work [10] uses the Distributed Hash Table to store the data where it is not mentioned in the work if indexing is supported. The work [3] stores the data in the cloud at the cloud-based platform FIWARE[32] (consists of 2 main parts IoT Edge and IoT backend) so it can use data indexing. Finally, in work [9], the local network that stores the data is an SQLite[33] database that also supports data indexing.

Usability

In this part, we compare the ease of connecting devices to each implementation. In our implementation, OpenAPI is used to describe the

[31] https://thegraph.com/en/
[32] https://www.fiware.org/
[33] https://www.sqlite.org/index.html

devices. In this way, all devices are treated as services, having their API, so it becomes very easy for a developer to develop applications using the devices as services in his application. The rest of the implementations do not support such functionality. IoTeX supports connection with only two devices, such as Pebble & Ucam. At the same time, works [10], [9], and [3] support the registration of devices in each system to be able to send information to the system. A general way of managing the devices' information in each system is not supported so that a developer can use the information in a common way.

Subscriptions
As in the previous comparison, IoTeX and the rest of the works [10], [9], [3] do not support the possibility of subscription to any device, with the result that no user/developer can be informed in real-time about the changes and the data sent by the devices. In contrast to all these, our implementation has this functionality.

Table 5.1: Comparison system/features.

	User / IoT protection	Data Protection	Usability	Data Storage	Indexing	User Storage	Subscriptions
iBoT	DID & VC	via smart contract. Also checks the value of the property.	Uses OpenAPI to behave on devices as an API service.	Blockchain	Yes	Hybrid: Local (VC) & Blockchain (DID Documents)	Yes
IoTeX	DID	Only encrypted data without extra check	Only 2 IoT devices can connect to the network	Blockchain	Yes, through The Graph protocol	Blockchain	No
[10]	From blockchain	via smart contract check if the data from a device is accessible from requester	There is no common way to gain access to any device.	Distributed Hash Table	N/A	Blockchain	No
Sash [3]	N/A	via smart contract check if the data from a device is accessible from the	There is no common way to gain access to any	Cloud (FIWARE cloud-based platform)	Yes	Blockchain	No

		requester, encrypted data	device.				
[9]	N/A	via smart contract check if the data from a device is accessible from the requester	There is no common way to gain access to any device.	At Local Network (SQLite)	Yes	Blockchain	No

Table 5.2: Comparisons between systems/works.

6. Conclusion and Future Work

6.1 Conclusion

The use of Service-Oriented Architecture and specifically RESTful services greatly facilitated communication between services and, thus, the planning of the services orchestration structure (App logic). A great advantage offered by the architecture was flexibility in using different programming languages for each system's function and ease of intervention modifications of individual services without affecting the overall system.
In addition, the development of the architecture with the help of Decentralized technologies such as blockchain, Decentralized Identifiers, and Verifiable Credentials helped us make our architecture more secure for data protection cases, users, and external applications created by developers. The blockchain as a technology has no single-point failure, and the system always remains available even if a peer is down.

Evaluating the architecture, we could list the following positives offered by the combination of the above technologies:

1. Users' identities cannot be exposed en masse since the information about their identity is stored only in their wallets and not in a database. The architecture has 2 factors for authentication and authorization, one factor is the wallet and the other one is the passphrase. A user needs wallet and passphrase to login at system.
2. In the blockchain, no one else can connect and send information other than the gateway that signs that it sends information with its wallet.
3. Blockchain also offers us the security that every transaction is recorded and cannot be changed. It helps our implementation by offering validity, that every use of the system by a user is recorded and cannot be disputed.
4. A very powerful feature is storing OpenAPI descriptions for devices. This feature allows developers to read sensor descriptions and

interact with the sensors as web services by connecting the WoT proxy.

5. Through the various smart contracts, we can simultaneously check if the value that sends(updates a property/action) a device is correct or not. In other circumstances, if we did not use the blockchain, we would have to create a microservice for each type of check, which would act as an intermediary between the corresponding API and the database. This procedure would be time-consuming, and every time we add a new check (a new microservice or update the existing one), we would have to update the existing services and connect to the new one. In contrast to the current implementation, we must create smart contracts and install them on the blockchain. Any device that wants to use a smart contract to check its properties/actions declares it in its description and in the payload it provides.

As is logical, there is no perfect architecture without negatives/disadvantages. For this reason, it is worth referring to these weaknesses in more detail. Below are the following negatives:

1. It was mentioned above about the positives offered by each user having their wallet so that the user's information is not exposed on any database or service. Beyond the positives it offers, this functionality for the user to be the owner of his credentials also contains certain weaknesses. The most significant disadvantage is that the credentials depend on how secure the user's wallet is. However, many implementations of different wallets, either separate hardware or web browser extensions, keep the information encrypted and difficult to recover from a third party.
2. Another weakness of the system is that there are too many types of devices, each sending its own type of information. Based on this diversity, it isn't easy to cover all scenarios for the security of all these types of information with smart contracts.
3. As a new technology, blockchain has a lot of room for improvement. One of them is the processing times of transactions (importing or retrieving information from the system) which are currently slower than current databases (such as Redis, MongoDB, etc.). Also, the implementation method of smart contracts plays an important role, as happens with the code of any architecture. The downside of slow transactions is trying to be countered by too many blockchain implementations as more and more are created or improved over time.

6.2 Future Work

Implementing the system as designed can finally satisfy the specifications set (Functional - non-functional system requirements, Chapter 3), achieving the development of a fully scalable IOT platform for device management and application creation following the architecture model's three levels. However, the work as it was prepared within a certain time frame is inevitable to have some weaknesses, which will be described above, accompanied by possible suggestions for their solution.

An improvement that the system accepts is searching for a better blockchain infrastructure or creating a better infrastructure using the Hyperledger Fabric. Some studies show that with a proper setup of the infrastructure of a Fabric system, 20000 transactions per second can be supported. At the same time, with the creation of better smart contracts, reducing the complexity, we can achieve very good results of transactions per seconds measurements. While another study[7] proposes the PBFT-DPOC Consensus Algorithm and achieves 10000 transactions per second(tps).

In addition, after the analysis of some papers mentioned in chapter 2 but also in conjunction with some components of the FIWARE[34] suite, such as the Orion[35] context broker, Cygnus[36] and STH COMET[37] we can replace some objects that are stored in the blockchain but store them off-chain in some database that will be connected to these services. With this logic, each infrastructure owner will have a node consisting of one WoT Proxy instance, one Orion context broker instance, Cygnus, and STH COMET. In each instance, there will be information about the infrastructure owner's devices, such as the subscriptions, current data & OpenAPI description, and the aggregated and raw historical data. Using this architecture, the blockchain will have the information of the users (DID Document) as analyzed in this work. Still, it will also have smart contracts for checking the information of the devices before they are written to the node described previously. In addition, users will send queries back to the blockchain, which will be selected from which node will request the information to answer the query. We will achieve that the huge amount of information that

[34] https://www.fiware.org/
[35] https://fiware-orion.readthedocs.io/en/master/
[36] https://fiware-cygnus.readthedocs.io/en/latest/
[37] https://fiware-sth-comet.readthedocs.io/en/latest/

the devices continuously produce is not stored in the blockchain but is stored and distributed. The blockchain will act as an intermediary that will select the correct node where the information is located each time. The architecture is presented in more detail in the image:

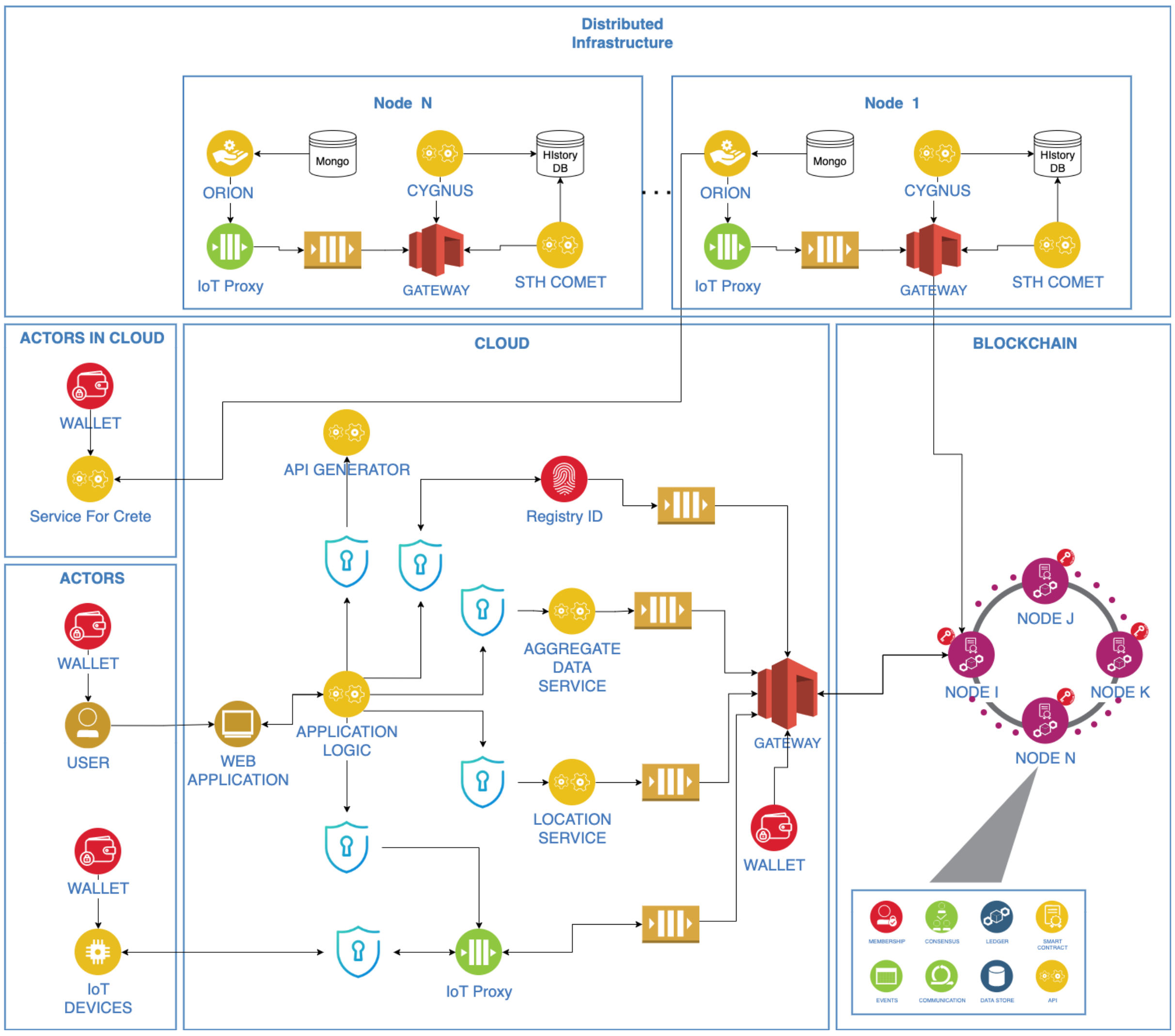


Figure 6.1: Future Architecture.

Finally, all requests between system services are implemented with the use of the HTTP protocol. Significant improvement in its security area the system is the communication protocol to change to HTTPS[38] as it is - by its architecture- a more secure protocol for transmission "sensitive" information (Authorization tokens, Master keys, etc.).

[38] https://el.wikipedia.org/wiki/HTTPS

7. Bibliography

[1] Xenofon Koundourakis, Euripides G.M.Petrakis, ed. n.d. "iXen: context-driven service oriented architecture for the internet of things in the cloud." https://www.sciencedirect.com/science/article/pii/S1877050920304488.

[2] Aimilios Tzavaras; Nikolaos Mainas; Fotios Bouraimis; Euripides G.M. Petrakis, ed. n.d. "OpenAPI Thing Descriptions for the Web of Things." https://ieeexplore.ieee.org/document/9643304.

[3] Hien Thi Thu Truong, Miguel Almeida, Ghassan Karame, Claudio Soriente, ed. n.d. "Towards Secure and Decentralized Sharing of IoT Data." https://ieeexplore.ieee.org/document/8946129.

[4] "Building the Connected World." IoTeX. https://iotex.io/research.

[5] Islam M. Momtaz A. Sadek; Muhammed Ilyas. n.d. "Securing IoT Devices using Blockchain Concept." https://ieeexplore.ieee.org/document/9659792.

[6] K. M. Giannoutakis, G. Spathoulas, C. K. Filelis-Papadopoulos, A. Collen, M. Anagnostopoulos, K. Votis, N. A. Nijdam. n.d. "A Blockchain Solution for Enhancing Cybersecurity Defence of IoT." https://ieeexplore.ieee.org/document/9284690.

[7] Shitang Yu; Kun Lv; Zhou Shao; Yingcheng Guo; Jun Zou; Bo Zhang. n.d. "A High Performance Blockchain Platform for Intelligent Devices." https://ieeexplore.ieee.org/document/8606017.

[8] Vagif A. Gasimov; Shahla Kh. Aliyeva. n.d. "Using blockchain technology to ensure security in the cloud and IoT environment." https://ieeexplore.ieee.org/document/9461397.

[9] MD Azharul Islam; Sanjay K. Madria. n.d. “A Permissioned Blockchain based Access Control System for IOT.” https://ieeexplore.ieee.org/document/8946172.

[10] Ruinian Li , Tianyi Song, Bo Mei, Hong Li , Xiuzhen Cheng. n.d. “Blockchain for Large-Scale Internet of Things Data Storage and Protection.” https://ieeexplore.ieee.org/document/8404099.

[11] M. F. Kaashoek and D. R. Karger. n.d. “Koorde: A simple degree optimal distributed hash table.” https://www.researchgate.net/publication/2942766_Koorde_A_Simple_Degree-Optimal_Distributed_Hash_Table.