



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Facultat d'Informàtica de Barcelona



# **Blockchain-based system for subscription payments in circular economy model**

*Adrian Manco Sanchez*

UNIVERSITAT POLITÈCNICA DE CATALUNYA  
(UPC) - BarcelonaTech  
FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

MASTER IN INNOVATION AND RESEARCH IN  
INFORMATICS

Computer Networks and Distributed Systems  
Master Thesis

**Directed by:**

Leandro Navarro Moldes  
Department of Computer Architecture

July 2, 2021

## Abstract

With the increased awareness on global warming, and the need for a change of paradigm in both production and consumption of electronic equipment, the need arises to find long-term sustainable ways to make a responsible use of the raw materials and other components used in IT systems. The amount of resources required to manufacture the devices we use on a daily basis, together with the high market demand for them result in an exploitation of the natural resources, that are becoming increasingly scarcer. The extraction of said materials from the planet threatens the environment, and raises the question of how long we can sustain the current paradigm of equipment production. Aside from the collection of raw materials, the actual manufacturing process results in the emission of not only greenhouse gases, but also dangerous substances that end up poured in rivers and the sea. Extending the lifespan of electronic devices through reuse helps to mitigate the impact of the manufacturing phase.

Beyond the ecology dimension, there is the social need to bridge the gap caused by purchasing power, and ease the access to IT equipment for everyone. The ability to acquire and use electronic devices has become almost compulsory for both individuals and societies, and not being able to access such equipment leaves people in all sorts of disadvantages. This phenomena can be seen in poor countries, where the purchasing power of the vast majority of people makes it unfeasible to invest in any sort of electronic equipment. Additionally, the lack of education on how to use these devices, or the misinformation as to how useful they are, are examples of situations that make the gap between users and non-users larger, even on rich countries.

As a last dimension, there is the economical need to make the electronic reuse market more transparent and accountable. Countries and organisations find that illegally dumping their electronic waste is more cost efficient than sending them through a regulated process. For the previous reasons, a regulated and transparent way of processing electronic waste would contribute to a more legal flow of money and interests, and may inspire organisations to engage on the business, and contribute to a more efficient and trustworthy way of recycling our devices.

This project aims to offer a model of infrastructure that allows a number of central organisations to enforce rules over a set of stakeholders, as well as the means to achieve a direct-debit-like decentralised payment architecture, that would enable individuals and organisations to be part of a electronic reuse system, and receive economic incentives and compensation for their participation. In such a system, a number of mediator entities would be in charge of linking providers of services to subscribers, that consume said services. Thanks to an automated payment system, subscribers could use a mediator entity as a bank, where they make a token deposit for the providers to collect. By using a private blockchain, the internal currency transfers are logged, resulting in an immutable register of activity.

To analyse and evaluate that such a system would be technologically feasible for the different potential use cases, and that the economical model could indeed help in the aforementioned context, a set of experiments and prototypes were conducted and developed, so the scalability and performance of the architecture could be assessed, and the correct integration of the different modules proved. The system was stressed at different levels, aiming to observe how the system performed under a growing demand.

The model proposed showed that the modules were able to coexist and interact successfully, with no major conflict points across the architecture, and was general enough so it could be adapted and extended further. Regarding the performance experiments, the results showed a remarkably good performance up to a certain load, with a non-negligible decreasing trend in performance as the stress levels increased. The whole proposal posed itself as a promising template to build systems featuring both a decentralised ledger, and a number of central authorities, although the blockchain module presented some problems regarding consistency, availability and performance in certain scenarios. When working at stress levels below the threshold observed in the experiments, the system performs correctly, but beyond that point, the uncertainty as to how the system will operate grows. Undesirable behaviours start appearing, and the overall performance starts to decrease, making the system unable of guaranteeing the availability of the service. Awareness of these limitations allows keeping the system under correct operational bounds.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	eReuse and NGIAtlantic . . . . .	5
1.2	The Need for a Circular Economy in the Electronics Market . . . . .	5
1.3	Objectives . . . . .	6
<b>2</b>	<b>Project Justification</b>	<b>8</b>
<b>3</b>	<b>Methodology</b>	<b>9</b>
3.1	Approach Taken . . . . .	9
3.2	Version Control . . . . .	9
3.3	Communications . . . . .	9
3.4	Tools and Environment . . . . .	10
<b>4</b>	<b>Architecture</b>	<b>11</b>
4.1	General Model . . . . .	11
<b>5</b>	<b>State of the Art</b>	<b>13</b>
5.1	Using a blockchain as intermediary to use Data as a Service . . . . .	13
5.2	Monitoring the status of subscriptions via a decentralised ledger . . . . .	16
<b>6</b>	<b>Implementation</b>	<b>20</b>
6.1	Detailed View of the Model . . . . .	20
6.2	Chronology of Tasks . . . . .	39
6.3	Security Considerations . . . . .	39
<b>7</b>	<b>Analysis</b>	<b>41</b>
7.1	Method used . . . . .	41
7.2	Results . . . . .	45
7.3	Discussion . . . . .	48
<b>8</b>	<b>Conclusions</b>	<b>50</b>
8.1	Future Work . . . . .	50

## List of Figures

1	Relationship between stakeholders . . . . .	11
2	Model of the implementation . . . . .	12
3	Stakeholders and Relationships . . . . .	14
4	Access to the blockchain . . . . .	15
5	Registration . . . . .	17
6	Requesting Access Token . . . . .	17
7	Using Access Token . . . . .	18
8	Subscription Expiration . . . . .	18
9	Interfacing . . . . .	21
10	Mediator Contract . . . . .	22
11	DDToken Contract . . . . .	23
12	Migration of DDToken . . . . .	25
13	Migration of Mediator . . . . .	26
14	Ganache JSON Configuration . . . . .	27
15	Connecting to the local Ganache Blockchain . . . . .	28
16	Fetching the deployed Smart Contracts . . . . .	28
17	Application Programmer Interface . . . . .	31
18	Example of a route . . . . .	32
19	Initialising the API . . . . .	33
20	Cron rule syntax . . . . .	34
21	Postman request . . . . .	35
22	docker-compose file . . . . .	36
23	Primary Keys and Tables . . . . .	37
24	pgAdmin . . . . .	38
25	Chronography of the Project . . . . .	39
26	Transaction object . . . . .	41
27	Intercepting the events of a transaction . . . . .	41
28	Results of the first experiments . . . . .	42
29	Configuration of a chart . . . . .	43
30	Preparing the input data . . . . .	44
31	Time required for each batch . . . . .	45
32	Rate of instructions per second . . . . .	46
33	Resource usage in the first experiment . . . . .	46
34	Resource usage in the second experiment (HTTP) . . . . .	47
35	Resource usage in the second experiment (WebSocket) . . . . .	47
36	Grafana failure . . . . .	49

# 1 Introduction

*This section explains the context in which this project lives, and how the goals for this project evolved as it was implemented. Furthermore, details are given about relevant organisations for the project, and why circular economy [5] –a paramount concept in this thesis– is needed in the current and future electronics market.*

## 1.1 eReuse and NGIAtlantic

eReuse [3] is a federation of activists, universities, NGOs and companies dedicated to collect and refurbish used ICT equipment. Its goal is to promote a collaborative and circular electronics market. Since 1995 the Universitat Politècnica de Catalunya has been involved in the refurbishing, channelling and tracking of thousands of electronic devices, either out-of-use from UPC or donated by other organisations, most of which were completely reusable. Educational activities, service learning, as well as research has contributed to these activities. Since 2013 the eReuse initiative started with a focus on developing software tools to help refurbishment and facilitate accountability of the circular lifespan of digital devices (computers, mobiles, tablets) [7]. The core idea of the entity, as stated in its web site, is "what if we opt for reusing rather than prematurely recycling"?

eReuse, together with the Distributed Systems Research group at UPC [8], has been involved in the development of a distributed, blockchain-based private ledger, that aims to serve as a verifiable log database for device transfer operations and proofs. Thanks to the inherent properties of the blockchain, this system poses itself as a transparent, immutable, verifiable and consensus driven way of storing business-relevant and circular economy related data.

Thanks to the work in this private blockchain system, we were able to contribute to an international project under NGIAtlantic [11] -a USA-Europe joint research effort on the Next Generation Internet, in collaboration with OBADA, a USA-based organisation dedicated to develop a blockchain-based protocol to track and document IT assets [9]. The system implemented by eReuse aims to be the blockchain part of the OBADA system, that is intended to become a standard for this kind of business needs.

## 1.2 The Need for a Circular Economy in the Electronics Market

Although this project has a highly technical side, it is also only understood when seen in context. This does not only include the technologies it discusses, or the analysis made, but also a number of social, economical, and environmental factors.

The resources needed to manufacture the electronic devices we use in our daily life are scarce, and its extraction is usually associated with some kind of exploitation of both the work force, and the environment [2]. Because of this, the reuse of the devices would lead to a decrease in the extraction of primary raw materials, and also in the interests that surround it. It is clear then, that the problem of electronic waste is not just a matter of efficiency. It is a complex matter that involves many agents, each of them having different motivations, and looking for some kind of profit.

Because it is a potential source of economical revenue, the electronic waste market is prone to feature activities of dubious legality. The voids in the legislation of some countries make the inadequate treatment of waste possible, and leaves room for a profitable market. It is because of such behaviours that systems such as the one discussed in this project would help to make the whole market more transparent, and thus make the appliance of laws easier.

Leaving the obscurity of this market aside, there is the understandable need of entities in the reuse chain to be compensated -that is, of making their work worth the effort. It is common to see incomplete recycling processes, due to the fact that the own process yields an amount of expense that exceeds the possible benefit. It is critical then, to be pragmatical in this, and find a way to compensate –or incentivize– the entities that decide to contribute to the circular model.

It is here that eReuse proposes a way to make the circular economy realistic and verifiable, by compensating the agents involved in the model [6]. By attaching an identifier in the registered devices, when a device cannot be reused anymore, and arrives to a recycling plant, it will be possible to see how much the raw materials in the device are worth, and thus they will be able to assess beforehand whether or not the device is worth processing. Additionally, by contributing to the final step of a device, the plants will receive an incentive, a reward in the form of a token, with an economical value attached to it. Last, but not least, there is the social factor. By introducing devices in the circular model, the organisations involved can refurbish them, and offer them for a lower price or even donate them, making them more accessible to people with less resources.

This project deals specifically with a subscription payment system, which is a part of the global system proposed. It is relevant to consider all the factors that influence this project, as well of its area of impact.

### 1.3 Objectives

The original purpose of this project was to propose a system to enable payments in a subscription-based fashion, following the example of a direct debit traditional bank system. The blockchain was the main technology to use, specifically a permissioned Ethereum blockchain, with smart contracts written in Solidity. Besides the implementation of such a model, and the discussion of its details, this project intended to include an analysis of the performance and scalability of the module, by deploying it in a testbed and conducting different experiments. By the time of the conception of this project, neither the testbed, nor the metrics or tests needed were clearly defined.

As the development of the project continued, the objectives remained the same, with no major variation. Since we were conscious of the possible limitations and difficulties we may find on the process, we preferred to formulate generic objectives, that were not bound to any detail of the project, but rather to its general functionality. We had to make some decisions as the need arose, such as whether or not it was meaningful to implement authentication or a front end, but we decided to focus in the areas that were critical for our system analysis.

One of the most difficult aspects we had to discuss was the nature of the analysis. Since we were not sure about what the nature of the system would be until a relatively advance stage of the project, it was difficult to devise meaningful and adequate tests. We first considered different variables and metrics for our analysis,

but had to discard some of them, because of the time and complexity that using them would take.

With that said, thanks to how we formulated the objectives, we were able to keep them from start to end of the project. Even though there were a number of assumptions we had to modify, and some sacrifices in the model and tests, the significance of the project remained untouched, and the overall project justified.

## 2 Project Justification

*Once we have discussed the main reasons why this project was conceived, we need to address the question of why it is actually needed or useful. Why it is significant for the topic, and how it achieves it.*

As we just discussed in the previous section, there are a number of social, political and economical factors influencing an electronics market. Here is where the technological factor takes importance. A blockchain offers a way to avoid, or at least mitigate, some of the adverse effects of the traditional workflow in this market. Its inherent transparency, as well as its distributed nature, could ease the process of enforcing regulations, and would allow the electronics recycling and reuse systems to achieve its goal, which is to make a better use of resources, minimise the harm done to the environment, and play an important role in society, by including people with less resources in a community of users.

In the scope of the project funding this thesis, this project has the job of effectively bringing in the benefits of the blockchain to an end-to-end circular economy model. Once the model is completed and tested, the intention is to present it as candidate to become a standard. The reason for doing this is that standards are respected and widely used. This would give the model more visibility, and help it achieve its original goals. Explaining the broader scope in detail would require far more than this thesis can cover, and we would lose the focus on the actual meaning of this thesis. Nevertheless, it is important to give an overview of where is this thesis located inside its parent project.

As stated, the NGIAtlantic project aims to implement the end-to-end model discussed in this thesis. Doing it requires a wide and very complex set of technologies, as well as the collaboration of international teams, formed by experts in their fields. The idea is that once a device is registered in the system, an abstract identifier called Obit is created. This Obit has the role of being a unique reference to a single device, and exposing relevant information to each one of the parts in the circular economy model. At some point in that workflow, there is the need of storing business-critical information, in such a way that its integrity is guaranteed. This information could be, among others, proofs of the device function, or proofs of the recycling, once a device is no longer refurbishable.

Hence, in this project we can find two main contributions. First, it offers the necessary infrastructure to store critical information, as a sort of verifiable log of the model workflow itself. Second, it allows the funding of the system by means of the contributions of the entities involved. Thanks to the capabilities of smart contracts, it is possible to define a custom, internal currency, in such a way that the system has its own economical rules, and the movements of said currency are trustfully recorded.



### 3 Methodology

*This section serves as explanation of the work was carried out. It explains the tools used - both technical and communications-related, how the version control was maintained, and how the different entities involved in the project coordinated with each other.*

#### 3.1 Approach Taken

As stated earlier in this memory, this thesis is experimentally driven. Thus, its development was driven by questions, and the process for finding them an answer. At conceptual level, this meant discussing the assumptions we were taking at each step and when implementing, it meant focusing on meaningful results at each iteration, and expand the work afterwards. The advantage of this agile-like philosophy, is that we could invest time and resources in a practical manner, and focus on what the system was supposed to do, rather than what it could eventually do. For these reasons we prioritised certain features to the detriment of others.

#### 3.2 Version Control

In this thesis, the complexity did not reside in the amount of code <sup>1</sup> written, but rather on discussing what part should be responsible for each task. In a blockchain environment, this is not trivial, since it could be very attractive to simply overload the contract code with logic, and perform every single operation on-chain. Actually, not everything could be done in the blockchain, and trying to do so is not suggested. For this reason, much of the work was thinking the most convenient and optimised way to perform the core operations, so we could analyse the system performance with as less noise as possible from secondary functionalities.

Nevertheless, to conduct a minimal version control, we used Git <sup>2</sup>, via GitHub <sup>3</sup> and GitLab <sup>4</sup>. The research groups had a repository working in the former, and we used the latter as a new addition. In both of them, we stored the scripts used for the proofs of concept and analysis, and also as a collection of references, from which we could extract already implemented functionalities, and take them to our project.

#### 3.3 Communications

Regarding my communication with the director of the project, we appointed meetings on demand, taking into account the availability of both of us. On these meetings, we reviewed at what point the thesis was, and made a draft of what should be in the coming days or weeks. Another of the values of these meetings was the broad knowledge the director had on the subject, since it allowed me to ask all sort of questions and acquire a deep understanding of the topic and context.

---

<sup>1</sup><https://github.com/DSG-UPC/e2e>

<sup>2</sup><https://git-scm.com/>

<sup>3</sup><https://github.com/>

<sup>4</sup><https://about.gitlab.com/>

At team level, we fixed a weekly meeting, in which each one of us explained what we had been doing that week, as well as what we intended to do the following one. It was useful to keep everyone updated on the overall state of the research, and to create a sense of community between the members. Additionally, since we had the chance of listening what our colleagues were occupied with, we could propose side meetings that involved a number of members, in a different time slot than the fixed weekly meeting. Due to the pandemic situation, from my side all of the contacts were done remotely, by means of instant messaging, mails and video conferences.

### 3.4 Tools and Environment

With regards to coding, I chose Visual Studio Code<sup>5</sup> as IDE. It offered a highly customisable set of options, as well as a tree view of the project folders, and an integrated terminal, to explore directories and run applications easily.

To write this memory, I used LaTeX<sup>6</sup> via Overleaf<sup>7</sup>. One of the most important advantages was the ability to share the document, so both me and the director could edit it simultaneously, as we discussed about it in a conference call. Besides this, using LaTeX made it very easy to format the memory, so it met all the requirements imposed by the FIB.

For communication, we started using Google Meet<sup>8</sup>, to later transition to Jitsi<sup>9</sup>. Virtually, they offer the same main functionalities, but the Jitsi software was running and configured in a local fashion, meaning the team could control the parameters of the video conference system. This allowed us to not depend on third party software when communicating among ourselves.

---

<sup>5</sup><https://code.visualstudio.com/>

<sup>6</sup><https://www.latex-project.org/>

<sup>7</sup><https://www.overleaf.com/>

<sup>8</sup><https://meet.google.com/>

<sup>9</sup><https://meet.jit.si/>

## 4 Architecture

*This section explains what are the building blocks of this project. It explains each part, and justifies it in the context of the project as a whole. For each part, details are given as to how it was implemented, and what design decisions were taken.*

### 4.1 General Model

Before moving to more specific parts of the proposal, it is interesting to observe an overview of the system, in order to have some context, and allow the following sections to be understood in a holistic way. This project presents a module of a system aimed to provide framework for circular economy scenarios. This part has the specific task of enabling automated subscription-based payments in a custom token created for a permissioned blockchain.

This opens up several points of discussion. Firstly, there is the decision of using a private, permissioned blockchain. In these blockchains, there is a certain degree of centralisation, and also a limit regarding who can use the system. In our case, the centralised aspect would be represented by a central trusted authority -a mediator- that would have privileged access to the smart contracts, and to the managing of tokens and databases. Regarding the users, it is expected that they reach an agreement with the organisation by traditional means -that is off-chain- before they are allowed to use the system. For these reasons, the discussed permissioned blockchain would unite the centralisation of authority of traditional, non consensus-based systems, with the properties of the blockchain regarding transparency and logging of events.

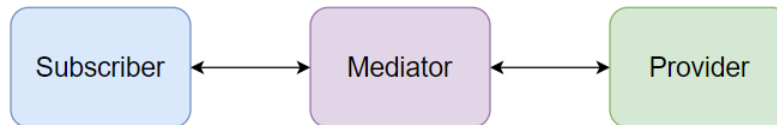


Figure 1: Relationship between stakeholders

With the blockchain as a central piece, we needed to decide what other elements would be capable of bringing the required functionality, and how to interconnect them appropriately. The following figure shows the general architecture of the model.

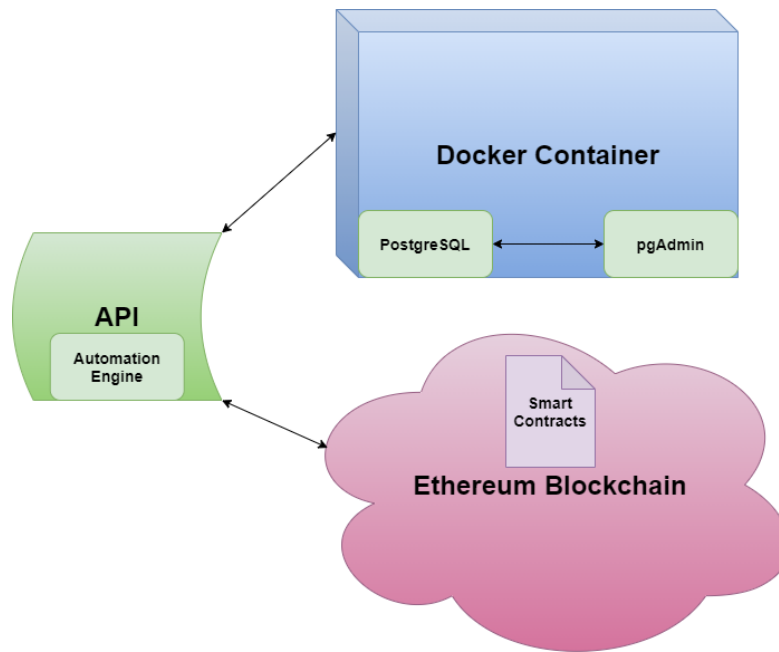


Figure 2: Model of the implementation

The most external part of the model is the API, that allows external entities to contact the system. It is the gateway to the blockchain, and as such, it enforces authorisation policies, so only correct actions arrive to the network. Additionally, there is a Docker container that holds both the database -to store user and subscription information-, and the pgAdmin -to manage the database easily with a GUI. The last part is the blockchain itself. It contains the Smart Contracts that make the token and subscription logic possible and secure. The Mediator acts as a representation of the organisation in the network, and the DDToken serves as a bank, where the registry of balances is kept, and token transactions are performed. Conceptually, there are two more entities involved: Subscribers and Providers. Although conceptually they exist, after several iterations we realised there was no need to have two more Smart Contracts representing each of them, and we limited their representation in the system to their corresponding addresses. This way, they can effectively participate in the model, but do not introduce unnecessary logic, and can be seen -simplifying- as bank accounts that spend and receive tokens.

## 5 State of the Art

*This section has the objective of presenting other ideas and projects about the topic. By observing them, we can have a better understanding of what is the current status of the technologies, and the research in this field.*

The following works represent efforts to seek architectures and use cases where a decentralised ledger could provide an innovative way of performing certain tasks. Much like in this project, the tasks that the blockchains perform in these proposals could be transferred to other, more traditional entities, like a simple relational database. Hence, it is useful to study how other researchers have been able to fit a decentralised ledger in a complex and modular system, so we can learn from them, and apply their experience in our own proposal.

### 5.1 Using a blockchain as intermediary to use Data as a Service

This work [1] proposes a Blockchain-based, private network model, to improve the way in which data is collected and shared. Data collectors tend to not share data unless it is done securely and with a relatively fair benefit, and data consumers find that acquiring data is costly, and often it is incomplete and heterogeneous. The proposed model uses Blockchain to provide security, transparency, and immutability, and uses the Data as a Service (DaaS) concept as a main axis. With regards to the relationship between businesses and customers, a subscription-based system is proposed, where consumers subscribe to a certain Data Producer, so the former can continuously access data, and the latter can keep receiving a revenue.

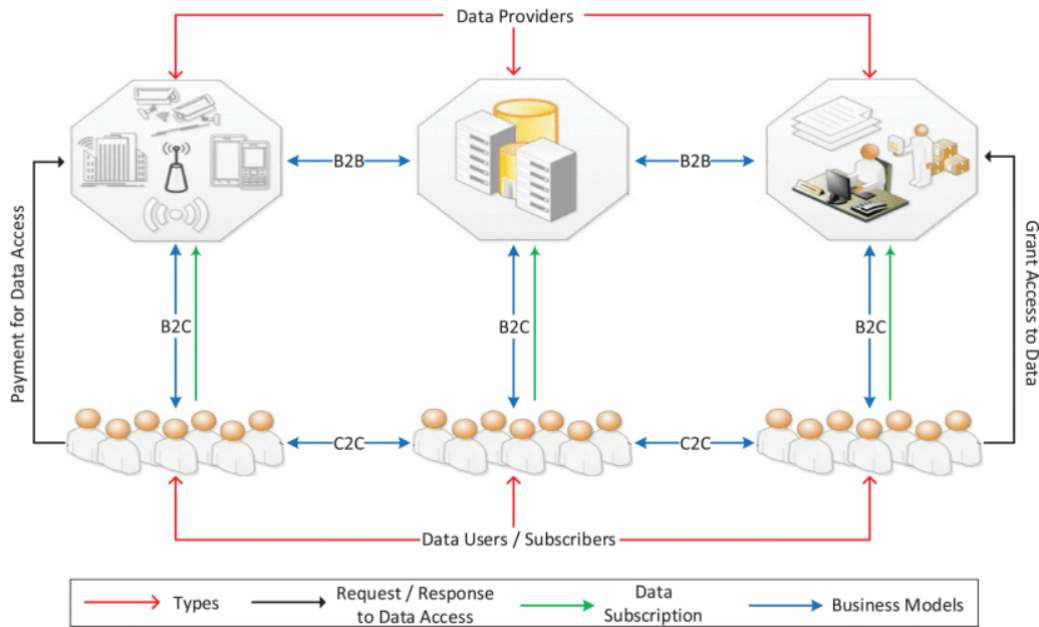


Figure 3: Stakeholders and Relationships

As can be seen in the model, both Data Providers and Data Subscribers are considered, as well as the relations between groups, and within each group.

- B2B: A Business acts as user of the data of another Business. This allows a Data Provider to subscribe to another Data Provider, and offer that acquired data as their own, so they can get a benefit via offering it to end users.
- B2C: When a Data User is interested in the data user by a certain Data Provider, it can initiate a subscription, so it will be granted access to the data as long as the payments continue happening at the agreed time.
- C2C: Similar to a B2B relationship, a Data User can offer its collected data to other Data Users by means of an API, with the payments being made in a digital currency.

To make these relationships effective, two main payment models are considered.

- Flat Rate Pricing (FRP): Subscribers pay after an agreed period of time. There is no additional charge related to the amount of data consumed. This model is administratively simple, and makes the prices easy to calculate, but since users pay a fixed rate no matter how much data they will consume, this model can lead to congestion, due to extensive use of the infrastructure. A subscriber that pays a fixed amount will most likely do exhaustive use of the system, rather than the opposite.

- Usage-Based Pricing (UBP): This model is a sort of remediation to the FRP model. Here, there are both a fixed, time-based payment, and a usage one. Infrastructure usage is more controlled here, since subscribers will better calculate what data they need to collect, but this fact is can also drive away users that are not willing to limit the amount of data they pretend to obtain.

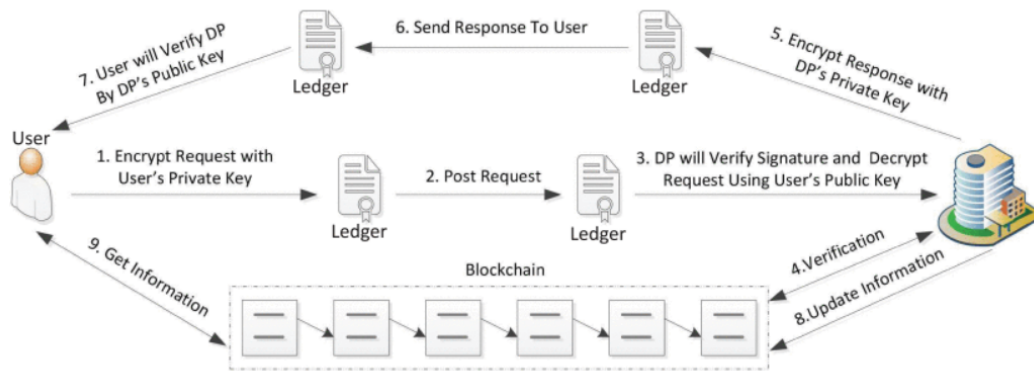


Figure 4: Access to the blockchain

Once a subscription is initiated, and the payment method is agreed, the model uses Public Key Cryptography to grant external users access to the data in the blockchain. As the image illustrates, the steps would be the following.

- A subscriber signs a request, and sends it to the Data Provider.
- The Data Provider will verify the signature in the request, and if it is correct, it will sign a response, and send it to the subscriber.
- The Data Provider will update the relevant information in the blockchain, and the subscriber will be able to collect it.

To encourage a honest use of the system, Data Providers are rewarded cryptocurrencies upon successfully provisioning to the users. Since this does not happen once, but rather for as long as the data continues being provided, the parties are encouraged to use the system periodically, and thus receive more incentives.

In the paper, the author concludes that DaaS is very beneficial in these scenarios, and that it significantly increases the benefit margin for Data Providers. Additionally, the properties of the blockchain, along with Public Key Infrastructure, and Authentication and Authorization mechanisms, guarantees that the system is highly secure and reliable. Computationally, the limitations imposed made the model highly efficient and scalable.

When compared to this project, we can see some comparison points. For instance, the use of a private blockchain as a trusted and immutable backend. After a Subscriber makes a request to a data Provider,

the latter verifies said request, and verifies that the desired information is in the blockchain, before letting the Subscriber collect it. In our project, the main use of the blockchain is not to store data, but to allow token operations, and to serve an immutable log of actions in the system. Regarding the considered business relationships, while the referenced work establishes three well defined types (B2B, B2C, C2C), our proposal does not define any clear type. Our aim is to offer a model that is applicable to a different range of use cases, and thus the possible business relationships are diverse. Finally, and with regards to the payment models, it is important to notice that, while the paper does mention two distinct types (Flat Rate Pricing and Usage-Based Pricing), we have not defined the specific payment models supported by our model. In this direction, we provide the technological foundation, leaving the door open to establishing different payment methodologies depending on the use case considered.

This research was useful to assess the theoretical validity of our model, and helped in the process of logically connecting the parts that composed our own infrastructure. In some aspects, the paper proposed more specific types or characteristics, but in general terms, we can see a number of similarities in the establishment of a subscription model, or in using the blockchain as a trusted registry.

## **5.2 Monitoring the status of subscriptions via a decentralised ledger**

This aim of this work [10] is proposing and testing a model to eliminate third parties such as banks in a cloud service subscription scenario. By offering different payment methods, supported by a blockchain backend, the system allows to perform low cost transactions, in which no central bank is needed. To do this, the authors implemented two smart contracts that enforce the distributed ledger logic. According to the authors, the main contributions of the paper are the following.

- Designing a protocol that allows cloud providers to charge their subscribers using two different payment models.
- Implementing said protocol in an Ethereum blockchain, and assessing its correctness.
- Conducting security and performance analysis.

The two payment models considered are very similar to the ones discussed in the previous section.

- Fixed Subscription: Fixed and periodic payment. Agnostic of how much the subscriber used the service.
- Pay As You Go: The amount paid is directly dependant on the usage done by the subscriber.

The authors explained the workflow of the system by dissecting a standard use case, in which an unregistered potential subscriber initiates the registration process, uses the system, and finally sees its subscription expire.



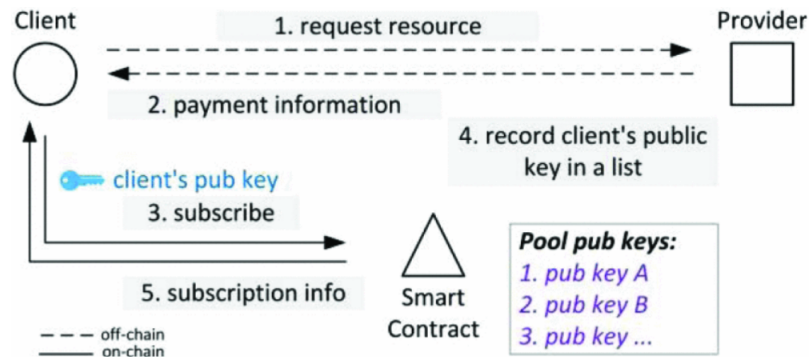


Figure 5: Registration

The first step is that a potential Client reaches the Provider to request a resource it stores. Since the service agreement does not exist yet, the Provider returns information about the required payment. After receiving this information, and agreeing on the terms, the Client uses its Public Key to subscribe in the smart contract, that stores the information, and returns to the Client the information about the successfully initiated subscription.

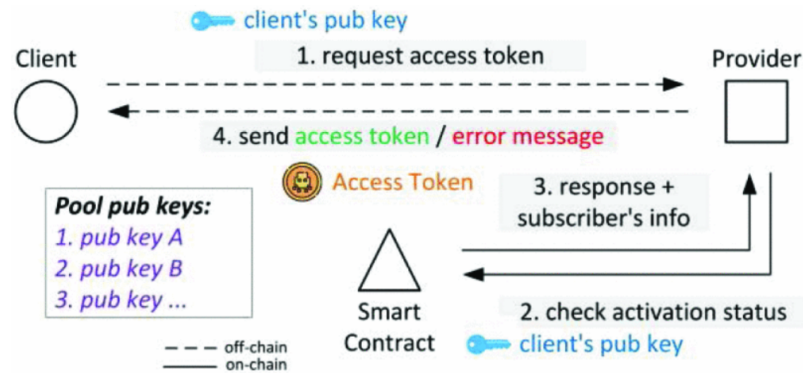


Figure 6: Requesting Access Token

Once the registration is initiated and active, the Client can request the Provider an Access Token to use its Cloud Services. When the Provider receives the Request, it reaches the smart contract, that checks whether the Client indeed has an active subscription. Finally, depending on what the response from the smart contract is, the Provider issues an Access Token, or an error message, indicating why the Client is not allowed to use the Cloud Services.

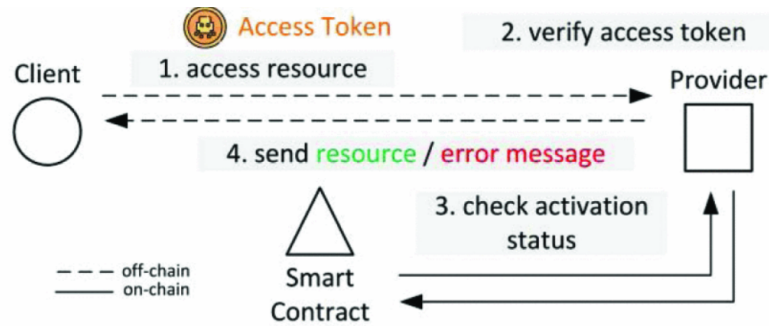


Figure 7: Using Access Token

Once the Client has the required Access Token, the process to use it is similar to the one followed to request it. The Client uses its Access Token to request access to the Cloud Services of the Provider, that reaches the smart contract to check if the Access Token is still valid. If it is, the Provider sends back the requested resource to the Client. Otherwise, the Client receives an error message, indicating the reason why it was not allowed to use the Cloud Services.

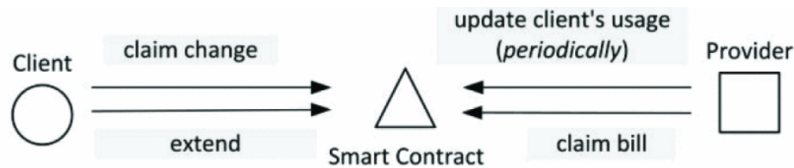


Figure 8: Subscription Expiration

Finally, when a subscription becomes inactive, there are different possible ways of proceeding.

- **Extend Subscription:** The Client can pay to the smart contract to extend its subscription, and also choose what type of subscription it wants. Depending on the payment model used, the payment is done as a single transaction, or as a recharge of the deposits that the Client has in the smart contract. This last option is the one used in the Pay As You Go payment model.
- **Fee Withdrawal:** The Provider is able to withdraw the deposits associated with the subscriptions of its clients.
- **Subscription Cancellation:** The Client simply lets its subscription to remain inactive, knowing that this implies it will not be granted further access to the Cloud Services offered by the relevant Provider. If the Pay As You Go model is used, the Client can claim back the deposit that was left in the smart contract.

Although their proposal can present some trust issue scenarios, they have been taken into account when designing and implementing the protocol, so they should not prevent the system to work as expected. Regarding integrity, they also designed unit tests to assess the smart contract worked as intended. Finally,

they conducted performance tests, proving that the system features low transaction costs, and that blocks are mined in a reasonable time frame of about 5 minutes.

Like we did in the previous example, we can find a number similarities and differences with respect to our project. The paper describes two smart contract used to enforce the logic of their blockchain system. In our case, we also have two smart contracts deployed, which enable token transfers (the Mediator and DDToken contracts). Like in the previous paper, here we can also see two well defined payment models (Fixed Subscription and Pay As You Go), while our proposal does not try to model any specific payment methodology, beyond offering the means for an automated direct debit subscription. If we observe the different steps in the workflow described in the paper, we can see that, except for the case in which a subscription expires, both the Client and the Provider have to reach each other before contacting the corresponding smart contract. For example, after issuing a request to a Provider, a Client has to reach the smart contract to make a subscription effective. Similarly, when a Provider receives an access request from a Client, it has to check in the smart contract if the subscription is active, before granting the client access to the data. In our model, the smart contracts are also a central part of the architecture (since both Subscribers and Providers have to contact them) although the paper mentions access tokens as authorisation method, and our proposal does not focus on such aspects of the model.

There are a number of similarities between this paper and our model. The use of smart contracts and the modelling of entities are similar, but there are differences in the authorisation mechanisms. We can observe an overall similarity in the purpose of some of the parts, but our approach of this project is more experimental, and focused towards offering a general, more adaptable architecture.

## 6 Implementation

*Now that we have gone through the different parts of the project, we will see how everything was implemented. The order on which each part was found necessary, and other details regarding design and development.*

### 6.1 Detailed View of the Model

#### Client library for interacting with smart contracts: Web3

To be able to interface with the smart contracts deployed in the blockchain, we needed a tool that allowed us to invoke functions and get information from them. There are different libraries focused on this, but two are the most widely used: web3.js<sup>10</sup> and ethers.js<sup>11</sup>. The web3 has been the standard option for a long time, and still remains so. It features several sub-modules, that as a whole offer a complete set of functionalities, that most developers will find suitable. Ethers features less sub-modules, but still offers the same core functionality, with the added value of featuring pre-written tests. Web3 can be seen as the obvious choice, but lately ethers has been growing in popularity. Since ethers it is much younger, it might be possible to see both libraries compete on equal terms in the future, once both can be considered equally mature.

#### Solidity Smart Contracts

One of the main reasons to use Ethereum<sup>12</sup>[4] was the ability to write code that would execute on the decentralised infrastructure, that is Smart Contracts. Although many other platforms offer similar capabilities, Solidity<sup>13</sup> Smart Contracts in Ethereum are the most common option. Other options include Solana<sup>14</sup>, Polkadot<sup>15</sup> or Eos<sup>16</sup>, but since the research group had worked with Ethereum in the past, we decided to keep using it, and take advantage of the gained experience.

The general idea of smart contracts is to write some sort of logic, just like it is done in traditional programming, but taking into account the characteristics of the blockchain. Once deployed, the logic contained in the code cannot be modified, and thus represents a trusted way of enforcing a certain behaviour, and leaving behind a trace of the events happened. Nevertheless, some of the assumptions we make when coding outside of the blockchain do not apply in Smart Contracts, and this is why many considerations have to be made when writing the code.

For instance, the more computationally intensive a code is, the more gas it need to runs. Gas is the unit with which Ethereum represents the effort done by the system to perform a certain task. Hence, while it is

---

<sup>10</sup><https://web3js.readthedocs.io/>

<sup>11</sup><https://github.com/ethers-io/ethers.js/>

<sup>12</sup><https://ethereum.org/en/>

<sup>13</sup><https://docs.soliditylang.org/>

<sup>14</sup><https://solana.com/>

<sup>15</sup><https://polkadot.network/>

<sup>16</sup><https://eos.io/>

important to optimise the code in all scenarios, it is of uttermost relevance in the blockchain. This is why Solidity code has to be concise, feature few loops, and be as compact as possible. With this in mind, we have that we need to design a complex system, but the main tool we have must be kept simple. This is where the concept of on-chain and off-chain come in.

While one could design a smart contract with all the business logic inside, and make it work, it would perform poorly and generate a lot of processing cost. In our permissioned scenario, the gas cost is zero, so there is no problem as to how much the code execution costs, but that does not free the engineers from having to keep it simple. Then, the logic that is moved outside of the Solidity code needs to be placed somewhere else inside the model. We call on-chain all the work done by the Smart Contracts and the blockchain infrastructure, and off-chain the work done by entities external to the decentralised system, such as a database or an API. In this scenario, we decided that the assignment of duties would be the following:

- On-chain: DDToken operations (approvals, transfers), Mediator operations (storing deposits, granting withdrawal permissions, refunding to Subscribers).
- Off-chain: API, Database (registration of subscriptions and stakeholders), automation engine.

This way, in a hypothetical production scenario, after the formal agreement between the parts, a task could be created, that would periodically contact the blockchain to perform the token transfers. Hence, we created a multi-interface process, where we use the API between the agreement and the off-chain system, and the automation engine between the latter and the on-chain infrastructure.

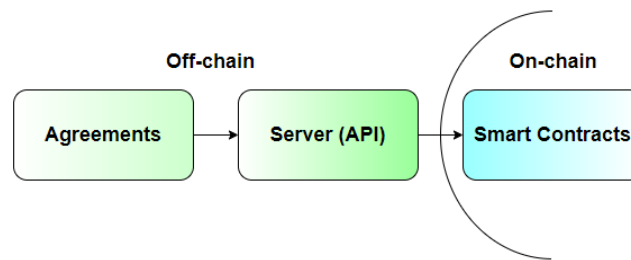


Figure 9: Interfacing

Regarding the specific Smart Contracts created, we decided to limit ourselves to two deployed smart contracts. After discussing with the team, it proved itself to be enough to fulfil the requirements. Although the existence of only two Smart Contracts has already been discussed, now we will see them in more detail. First, we will review the Mediator. Since the DDToken contract is derived from the OpenZeppelin libraries, it will be explained in the next subsection, to better appreciate the interaction between the parts. As global variables and data structures, the contract included the address of its owner -that is, the address that deployed the smart contract to the network, a reference to the deployed DDToken contracts -so the Mediator could effectively perform token operations, and a deposits mapping. The latter had the task of associating a given Subscriber to the set of Providers it had an agreement with. This allows the Mediator to store deposits coming from a certain Subscriber for a certain Provider, so eventually the latter could collect the tokens.

```

contract Mediator {
    address public owner;
    mapping(address => mapping(address => uint256)) deposits;
    DDToken public tok;

    constructor() public { ...
    }

    receive() external payable {}

    modifier onlyOwner() { ...
    }
    function setToken(address _tokAddress) public onlyOwner { ...
    }
    function depositPull(address _sub, address _prov, uint256 _amount) public onlyOwner { ...
    }
    function depositPush(address _prov, uint256 _amount) public { ...
    }
    function refund(address _sub, address _prov) public onlyOwner { ...
    }
    function payProv(address _sub, address _prov) public onlyOwner { ...
    }
}

```

Figure 10: Mediator Contract

When we observe the main functionality of the contract, we can find the following:

- `onlyOwner()`: Modifier that is used as authorisation method for critical operations, so only the trusted central organisation, whose address is the same that deployed both the Mediator and the DDToken, can perform such operations. This pattern is commonly used in Solidity, and allows for a better "checks-effects-interactions" logic. This means that all security checks are done in the first place, and if they are all successful, the local contract stores the effects of the operations, to finally interact with other contracts if needed. This helps reducing the attack surface, making the code more secure. Were any problem to happen, the abort produced by the blockchain would revert all changes.
- `setToken()`: Function used at migration time, that assumes a DDToken contract is already deployed, and updates the reference present in the Mediator contract.
- `depositPull()`: Function that first checks if there is an agreement between the specific Subscriber and Provider passed as parameters, and in case there is, the Mediator provokes a movement of tokens from the Subscriber to itself.
- `depositPush()`: Function that allows a Subscriber to proactively make a deposit in the Mediator for a specific Provider. Since it does not check if the sending address is the one of the owner, this function opens the gate to interfacing with external software such as Metamask, as a part of a Decentralised Application (dApp) that uses the system explained in this thesis.

- refund(): Function that allows the Subscriber to return a deposit to its original owner -a Subscriber. The Subscriber can request a refund off-chain to the API, and the system will check whether the tokens can be refunded or not prior to invoking the function in the smart contract.
- payProv(): Function that forwards a deposit to its intended destination -a Provider. In the off-chain module, thanks to the automation engine, when the corresponding period of time has passed, an action is triggered, so the method is invoked, and the Provider can receive the deposited amount of tokens.

## Smart Contract Templates

It is worth mentioning that in order to work more efficiently, we used OpenZeppelin's <sup>17</sup> well-known libraries as a basis for the Smart Contract code. By using this approach, we avoided working on something that was already done and tested, and saved time to invest it in other matters. As said previously, we developed a Mediator and a DDToken. We have already went through the details of the Mediator Contract. Now we will see in detail the nature of the DDToken contract.

```
contract DDToken is ERC20{
    address public owner;

    constructor(uint256 _initialSupply) public ERC20("DirectDebit", "DD"){ ...
    }

    function approveCentralized(address approver, address spender, uint256 amount) public returns(bool){ ...
    }

    function transferFrom(address sender, address recipient, uint256 amount) public override returns(bool){ ...
    }

    function increaseAllowance(address spender, uint256 addedValue) public override returns(bool){ ...
    }

    function decreaseAllowance(address spender, uint256 subtractedValue) public override returns(bool){ ...
    }
}
```

Figure 11: DDToken Contract

At first glance, it may seem very simple, and indeed it is, but it is worth noticing that it inherits from the ERC20 OpenZeppelin contract. This means that it features all its functionalities, plus the ones we added for our specific needs. For the sake of clarity, we will give an overview of the functionality of the original ERC20 implementation.

- Getter functions to obtain information such as the name, symbol, decimals, or total supply of the token.
- Getter functions to obtain the balance a specific account.

---

<sup>17</sup><https://openzeppelin.com/>

- Functions to allow an address to operate with the tokens of another address on its behalf. This is especially interesting in our scenario. Additionally, there are functions to modify the limit of tokens per operation, or to check the current approvals.
- Transfer functions that allow both to send tokens from the sender address to any other, or to move the tokens of another address to a receiver. For the latter, it is necessary that the address invoking the transfer operation has the permission of the spender address, in order to operate with its tokens.
- Other functionality like minting tokens, burning them, or changing the decimals.
- For all the core operations, there is an internal version that simply produces the desired effect, without doing further checks. In the original implementation, these are private, so no external address can invoke them. The public versions of the functions are the ones invoked, doing all the checks, and finally call the internal functions. In our custom implementation, this allows us to do our own checks, and afterwards calling the internal functions, thus bypassing the default security measures applied by the default public functions.

After seeing what the original ERC20 does, we can compare it with what the custom DDToken offers. In essence, it adds a new functionality, and deactivates some functions, that were meaningless in the new implementation:

- `approveCentralized()`: Function that allows the owner of the contract (the organisation) to allow an address to spend the tokens of another address. Originally, only the address whose tokens would be spent could give such approval, but after reviewing our business case, we found out that this was the most convenient solution.
- `transferFrom()`: This function allows an address to spend the tokens of another. For this, a previous approval is needed. Originally, the ERC20 established an approved amount, so after that amount was moved by a non-owner address, the owner had to renew the approval for some new amount of tokens. We decided that was unnecessary for our needs, so we made the new function just check if the amount to be transferred exceeded a certain limit, but eliminated the notion of an approved amount or a renewal. There is only an upper limit for each single operation, as a security measure, not allowing the organisation to move an arbitrarily significant amount of tokens.
- `increaseAllowance()`: Function that returns true. Originally, it was used to increase the amount of tokens that a specific allowance had to operate, but since we removed that need, this function was unnecessary.
- `decreaseAllowance()`: Similar to the `increaseAllowance()` function, but for the case of decreasing the approved amount left to operate.

Some of the decisions taken led to a behaviour that was not very likely to adapt to a production system. Indeed, the approach taken gives a lot of power to the organisation, but since the purpose of the system, at the time of writing, was academic, we decided to take the shortest path to a working prototype, without being excessively concerned about security or end-user experience.



## Migrations

To complete the explanation of the Smart Contracts used, we will go through the process of migrating them to either the testbed, or the local development blockchain. The Truffle framework uses a default migration file, used to bootstrap the smart contract migration process. We focus only on the migration of the two relevant Smart Contracts.

```
const ERC20 = artifacts.require("DDToken");

module.exports = async function (deployer, network, accounts) {
  ...
  console.log(accounts)
  await deployer.deploy(ERC20, 1000000, { from: accounts[0] })

  const tok = await ERC20.deployed()
  console.log("tok:" + tok.address);

  for (i = 2; i < accounts.length; ++i) {
    | await tok.transfer(accounts[i], 10000, { from: accounts[0] })
  }
};
```

Figure 12: Migration of DDToken

The migration process serves the purpose of effectively making the code present in the network of choice, in such a way the contract can execute its logic and be contacted by other entities. Following this idea, it is important to arrange the deployment order correctly, in order to allow contracts that need the information held by a previously deployed contract to be instantiated successfully. In our case, this meant that, while the DDToken is agnostic of the other possible contracts in the system, the Mediator does include a reference to the token contract being used to handle the operations and management involving the internal token. This created the need to deploy first the DDToken, and later the Mediator. As can be seen in the corresponding image, the migration code is quite concise. It prints in console the set of accounts used in the node, deploys the contract to the network, takes the deployed contract abstraction, and prints in console the address on which the contract was deployed. In the process, there are a couple of actions regarding token movement, that are mainly related to the experimentation process that we conducted.

When the DDToken contract is deployed, we state both the name of the instance deployed -ERC20, as well as the initial supply of tokens held by the DDToken contract, and the owner address of the contract. Here, we decided address 0 to be the one the organisation uses to perform actions in the blockchain, and thus it is the owner of the DDToken contract.

At the end of the DDToken migration script, we iterate through the set of accounts, and transfer each one of them a specific number of tokens. We used the transfer() method without explicitly specifying a sender

account, so account 0, that initially held all the supply of tokens, "loses" 10000 units for every other account in the system. At this stage, the consideration of how many accounts would be present in the testbed, along with the nature of the experiments, is what determined the initial supply of tokens (so all non-organisation accounts could receive their tokens). After a number of iterations of the experiments, it was possible that some accounts were left without tokens. Since that would prevent the experiments from being correctly conducted, we had to find a way to easily refill the accounts with tokens, without having to migrate the contracts again, or manually transferring tokens from one account to another. For this, we wrote a separate script, that repeated the same action performed by account 0 at migration time. That is, to iterate through all other accounts, transferring a certain amount of tokens to each one, and thus making possible that all accounts can perform token transactions, and participate in the experiments.

```
const Mediator = artifacts.require("Mediator");
const DDToken = artifacts.require("DDToken");

module.exports = async function (deployer, network, accounts) {
  ...
  await deployer.deploy(Mediator, { from: accounts[0] })

  const tok = await DDToken.deployed()
  const med = await Mediator.deployed()
  console.log("med:" + med.address);

  await med.setToken(tok.address, { from: accounts[0] })
};
```

Figure 13: Migration of Mediator

To conclude the explanation of the migration process, we need to observe how the Mediator contract was deployed on the testbed. The first difference with respect to the DDToken migration is that this code requires the artifacts (an abstraction created by Truffle) of both contracts. As we will see, this is due to the fact that we needed to both deploy the Mediator, and use the DDToken contract.

The migration process in this case starts in the same way, and sets account 0 again as the owner of the contract. Afterwards, the deployed instances of both contracts are fetched, and the address where the Mediator contract resides is printed in console, to ease the development process. Finally, there is an invocation of the `setToken()` method in the Mediator, and here is where the use of the DDToken is justified in this code. To effectively associate both deployed Smart Contracts, we update the (yet null) reference in the Mediator, so it points to our DDToken contract.

The nature of this project made the overall migration process very simple. Other projects may need a far more complex logic, and the interdependence between deployments can become a problem. Truffle makes it easy to order the migrations, by simply including at the beginning of each file name the order that it occupies in the whole process. With this idea, our migration order was the following:

- 1\_initial\_migration.js
- 2\_deploy\_token.js

- 3\_deploy\_mediator.js

## Local Development Blockchain

For the development of the code and the initial proof of concept for the test, we decided it was better to keep everything local, rather than deploying it in a testbed. Some of the reasons included convenience -a local environment has less stakeholders involved, and thus can be adapted rapidly-, and coordination inside the group. As my main job in it was to develop the code, model and tests that would later compose my thesis, a good way to allow the rest to advance, while not having to wait for them, was to simulate a scenario in my laptop. Given the inherent qualities of the blockchain, and the immutability of the code once deployed, it is common practice to make the deployment the last step in the development process. Although in the specific context for this project the testbed belonged to the research group, and the objective was to analyze rather than building a market-ready application, it was still the most efficient way of working. As for the specific technologies used for this part, Ganache <sup>18</sup> was the chosen tool. The idea behind it is that it allows to rapidly create a blockchain in your own computer, and exposes it in a local user port, so it can be contacted by different applications in your device. Although Ganache offers a GUI to make the management of the blockchain more friendly, we decided to use its command line version (Ganache-CLI). This version of the tool allowed us to specify a number of parameters like the mnemonic, the number of accounts or the configuration to build the network, all with just one command. This made the whole development process more agile and focused. The following shows the configuration used for the local development environment.

```
development: {  
  provider: ganacheWeb3.currentProvider,  
  host: "127.0.0.1",      // Localhost (default: none)  
  port: 8545,            // Standard Ethereum port (default: none)  
  network_id: "*",       // Any network (default: none)  
  gasPrice: 0  
},
```

Figure 14: Ganache JSON Configuration

This is a typical configuration when developing in Ganache, but it features a detail that suited the real nature of the testbed. The provider field simply states the agent that will give us connectivity with the blockchain backend, and make us able to issue requests to the system. The host, port and network\_id parameters define that the device hosting the running blockchain is the local device, and that it will be exposed on port 854, with an unspecified network identifier. Lastly, the gasPrice parameter is set to 0. The reason is that in the scenario considered for this project, a production system using the module described in this thesis would be a permissioned blockchain where there is no need to reward miners, because the incentives to participate in the system are included in the economical model. The decentralised system is a means to provide immutable and transparent logging of device proofs and transactions, not a way to encourage the participants to mine blocks for a profit.

---

<sup>18</sup><https://github.com/trufflesuite/ganache>

```
web3 = new Web3('http://127.0.0.1:8545')
accounts = await web3.eth.getAccounts();
net = await web3.eth.net.getId()
chain = await web3.eth.getChainId()
```

Figure 15: Connecting to the local Ganache Blockchain

Once the Ganache-CLI is running, and there is an available blockchain configured, we can connect our code to it, and use it. In the figure we can see how we instantiate the Web3 library, and obtain an object that will let us retrieve the information and data of the local blockchain exposed in the shown URI. For our purposes, we used the chain identifier to check that we were using the correct system, but on the other hand, the net identifier has a more essential role. As the following figure shows, to make the code more parametrized and adaptable, we made the code obtain the information about the deployed contract. By using the net identifier, we can obtain the address of the contract in the specific network. Afterwards, we can use that address, and the Application Binary Interface, which is a representation of all the variables and methods in the contract, to create an interface that will allow us to invoke functions from that specific deployed contract directly in our code.

```
tokAt = DDToken.networks[net].address
tok = new web3.eth.Contract(DDToken.abi, tokAt)

medAt = Mediator.networks[net].address
med = new web3.eth.Contract(Mediator.abi, medAt)
```

Figure 16: Fetching the deployed Smart Contracts

One last meaningful detail is the accounts array provided by Ganache. When we started the Ganache service, it used a specific mnemonic phrase to generate a set of accounts. Both the mnemonic phrase, and the amount of generated accounts can be left by default, or customised, but the point is that Ganache will recognise all transactions issued by any of these accounts as signed, because the local blockchain knows the private key of all of them. This frees the developer from having to sign the transactions before sending them to the local blockchain, but introduced several considerations, regarding what was the most convenient approach, given the context of the project.

- **Setting accounts in the nodes:** In this approach, we would explicitly configure the set of available accounts to use. This way, transactions could be made mostly in one line, and there would be no need to write specific addresses as strings in our scripts, because the web3 injects in the code the array of available accounts in the contacted node. This would allow us to refer to each account by its position in the array, rather than its true alphanumerical value. With all its advantages, it still presented some problems. From an administrative point of view, the accounts should need to be configured in the node

beforehand, which forces the blockchain administrators to agree on the number of accounts needed and their values -losing flexibility. Besides, this approach was not realistic, since in a blockchain system, the most secure way to use the network is to sign all transactions before sending them, so the nodes do not need to store any private key -thus being less of a potential attack vector.

- **Signing all transactions before sending them:** Here, the nodes would not store any address or private key. They would only be an entry point into the system, and would require all incoming transactions to be signed. The problem with this approach is that, although the most realistic one, it introduced a certain degree of rigidity in the code and tests. The addresses should be written literally in the code, and that meant that they needed to be stored in an external file, generated by Ganache upon starting the blockchain. This in turn also meant that the mnemonic would need to be the same each time the local blockchain was started, or otherwise the file storing the addresses and keys should change each time.

Considering the benefits and drawbacks from both approaches, we decided to set the accounts in the node. It eased the development process, and the realism that the other approach would have brought was not a significant added value in this project. The goal was to build a prototype system, and assess how optimal its performance was when measured under different conditions. in a specific context, not obtaining a fully working solution ready for production. That is left for future work.

## Autonomous Blockchain

The research group built a private Ethereum blockchain using Geth nodes. Three of these nodes participated in a PoA algorithm, to decide how transactions could be fitted inside blocks and the order in which transactions should be recorded. Additionally, there were two other nodes that did not participate in the PoA, but also featured an EVM and had a copy of the blockchain in storage.

In public blockchains, cryptocurrencies are used as an incentive for nodes to contribute to the consensus algorithm. Since this was a private blockchain, the nodes were managed by the own organization, and there was no need for an incentive. This led the team to set the gas cost to 0. Since the goal is to maintain the system with the contribution received by the subscription payments, we discarded charging for gas usage.

An interesting fact about this blockchain is the proximity between nodes. Since all nodes are geographically close to each other, the latency of the consensus process is significantly low, and as a consequence, so is the overall response time of the system. If a node were far away from the main core, the consensus would take more time. This is an aspect worth exploring in future versions of this work.

## Monitoring

As an event monitoring and alerting platform, Prometheus <sup>19</sup> allowed us to collect data from the testbed nodes, and export it to Grafana, so we could see how the system was performing.

---

<sup>19</sup><https://prometheus.io/>

## Visualisation Dashboard

To be able to monitor the activity of the testbed, we used Grafana <sup>20</sup>, a visualisation platform that used the data collected by Prometheus, and offered an adjustable dashboard. Thanks to Grafana, we were able to see how the server side was responding to our tests, and be able to contrast the results we were obtaining on the client side.

## API server

To be able to expose the functionality of the blockchain-based system to the potential users, we needed a convenient way to receive requests, and forward them to the blockchain or the database. Since the core of this project was not to offer a production system, we chose a framework that allowed us to fulfil our needs without adding too much complexity. Taking into consideration the use of Web3 and the Truffle framework, the use of an Express.js <sup>21</sup> API in the backend would ease the process of integrating those third party libraries with our code.

Although we have stated that we had ease of development as a main premise, the Express.js framework is a very complete tool, that can be perfectly suited for many production use cases.

---

<sup>20</sup><https://grafana.com/>

<sup>21</sup><https://expressjs.com/>

```

app = express()
app.post('/regSub', (req, res) => { ...
})
app.post('/regMed', (req, res) => { ...
})
app.post('/regProv', (req, res) => { ...
})
app.post('/addSubscription', (req, res) => { ...
})
app.post('/startSubscription', async (req, res) => { ...
})
app.delete('/stopSubscription', (req, res) => { ...
})
app.delete('/delSubscription', (req, res) => { ...
})
app.get('/subAgmt', (req, res) => { ...
})
app.get('/medAgmt', (req, res) => { ...
})
app.get('/provAgmt', (req, res) => { ...
})
app.get('/subLim', (req, res) => { ...
})
app.listen(3000, async () => { ...
})

```

Figure 17: Application Programmer Interface

The architecture of the API was rather simple. We needed to create an Express application, and then create as many routes as needed. Although we were in a lab environment, and we could have used POST as method for all requests, we decided to make it minimally realistic, and use different HTTP Verbs depending on the semantics of each request.

- POST: Requests that added records to the database, or start some type of process.
- DELETE: Requests that removed records from the database, or stops some type of process.
- GET: Requests that fetched information from the database.

Regardless of the HTTP Verb used, all routes followed the same pattern. They created a query for the database, and then checked whether the action has been successful or not. In the following example, we can see the route that registers a new Subscriber in the system. The code creates a query with the subscriber address passed in the request, and later prints out the error if the operation fails, or an informative message if it was successful. This is only an example, but it is enough to illustrate the approach taken in when designing the API. As stated, we did not perform any kind of authentication or further checking, because in the lab environment, it did not add any interesting insight.

```

app.post('/regSub', (req, res) => {
  pool.query(`\
    insert into subs (sub) values ('${req.query.sub}');`, (err, result) => {
      if (err) { res.send(err) }
      else { res.send(`Row inserted succesfully.`) }
    })
})

```

Figure 18: Example of a route

The functionality of each one of the API routes was the following:

- `regSub()`, `regMed()`, `regProv()`: Inserts a row in the corresponding `subs`, `meds` or `provs` table, to effectively register the entity in the system.
- `addSubscription`: Inserts a row in the subscriptions table, that relates three existing entities (a Subscriber, a Mediator and a Provider), and describes the nature of the payment associated (periodicity, amount, etc.)
- `startSubscription`: If the subscription was not already active, it changes the "active" field in the corresponding row of the subscriptions table to reflect the change, and initialises the automated cron process. Said process takes as parameters the information stored for the relevant subscription in order to perform the payments as agreed.
- `stopSubscription()`: If the subscription had its automated payment process initialised, this route stops it, and marks the subscription as inactive in the "active" field in the corresponding row of the subscription table.
- `delSubscription()`: Effectively stops the associated payment process, and deletes the row that represents the subscription in the database. The stakeholders involved would still exist in their corresponding tables, just in case they needed to start a new subscription in the future.
- `subAgmt()`, `medAgmt()`, `provAgmt()`: Retrieves from the subscriptions table all agreements involving the corresponding stakeholder.
- `subLim()`: Establishes the maximum amount that a Subscriber could be charged in one operation.

To make the API run as a service, and receive the requests made by its clients, Express features the `listen()` function. In it, developers can specify the port on which the API will be listening, as well as additional actions that will be conducted once the server boots up. We used this to prepare the database, and to connect with the blockchain. Additionally, in case the server was recovering from a failure, we also checked the state of the database, so we did not create tables that already existed, and we also recovered the automated subscription processes, so the direct debit payment system could continue working as planned.



```

app.listen(3000, async () => {
  pool.query(
    `create table if not exists subs
    (sub varchar not null,
    primary key (sub));

    create table if not exists meds
    (med varchar not null,
    primary key (med));

    create table if not exists provs
    (prov varchar not null,
    primary key (prov));

    create table if not exists subscriptions
    (sub varchar not null, med varchar not null, prov varchar not null, lim integer not null, charge integer not null, unit varchar not null,
    num varchar not null, fwd boolean not null, active boolean not null,
    primary key (sub, med, prov),
    foreign key (sub) references subs (sub),
    foreign key (med) references meds (med),
    foreign key (prov) references provs (prov));`, async (err, result) => {
    if (err) {console.log(err)}
    else {
      console.log(`Database initialized successfully. API is listening on port 3000.`)
      payments = new Map()
      web3 = new Web3('http://localhost:8545');
      accounts = await web3.eth.getAccounts();
      pool.query(`\
      insert into subs (sub) values ('${accounts[1]}'), ('${accounts[2]}'), \
      ('${accounts[3]}'), ('${accounts[4]}');\
      insert into meds (med) values ('${accounts[0]}');\
      insert into provs (prov) values ('${accounts[5]}'), ('${accounts[6]}'), \
      ('${accounts[7]}'), ('${accounts[8]}');`, (err, result) => {
        if (err) { console.log(`${err}\nAccounts not added.`) }
        else { console.log(`Accounts added to the database.`) }
        pool.query(`\
        select * from subscriptions where active = true`, (err, result) => {
          result.rows.map(row => {
            console.log(`Active subscription from ${row.sub} to ${row.prov} found on database.`)
            var task0 = cron.schedule('0,30 * * * *', async () => {
              console.log(`${row.prov} sets amount + Return previous amount to ${row.sub}.`)
            }, { scheduled: true })
            var task1 = cron.schedule('10,40 * * * *', async () => {
              console.log(`${row.med} pulls money from ${row.sub}.`)
            }, { scheduled: true })
            var task2 = cron.schedule('20,50 * * * *', async () => {
              console.log(`Final pull enabled. ${row.prov} can pull from ${row.med} the funds of ${row.sub}.`)
            }, { scheduled: true })
            var tasks = [task0, task1, task2]
            payments.set(`${row.sub}-to-${row.med}-to-${row.prov}`, tasks)
          })
        })
      })
    }
  })
})

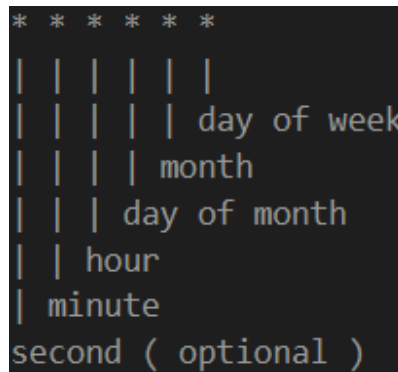
```

Figure 19: Initialising the API

In the example, we first created all databases if they did not already exist. Afterwards, we connected with the blockchain with Web3, fetched the accounts available, and updated the corresponding tables. Finally, if we found there was an active subscription in the database, we restarted the necessary logic to recover the automatic payment process associated with that subscription. In this case, we assign a fixed payment policy to all found active subscriptions. As can be seen in the image, we tested the recovery of active subscriptions by reinitialising three of them, belonging to a Subscriber, a Mediator and a Provider. This allowed us to see a minimal working example of how the different modules would behave when working together. A future work might be to refine this logic and make it more customized, maybe fetching the details of the

payment process from the database. For the purposes of this project, we found enough the fact of proving the combination of technologies was feasible, so we did not implement some of the required features for this API to be considered as complete. We provided the template for such a combination of modules, offering a service with simple management of subscriptions, blockchain-based payments, and basic recovery in case the service stopped.

One last aspect to discuss regarding the API, is the question of what was better to do on-chain, and what off-chain (as discussed previously). The main problem we faced regarding the separation of duties, was that at the time of writing, Ethereum did not inherently feature any type of automation or schedule mechanism. Thus, there was no direct way to program the payments to be done at a specific time, which is precisely one of the core features of any subscription-based, direct-debit mechanism. To solve this, we decided to move the automation logic off-chain to the API. Since it was located in a Node.js backend, we could take advantage of the different packages and capabilities that it offered regarding JavaScript code execution in the server, and build the automation engine there.



```
* * * * *  
| | | | |  
| | | | | day of week  
| | | | | month  
| | | | | day of month  
| | | | | hour  
| | | | | minute  
| | | | | second ( optional )
```

Figure 20: Cron rule syntax

As a basis for such engine, we used `node-cron`<sup>22</sup>, a package that emulated the behaviour of the cron tool in Linux systems. It allowed us to create objects that represented an automated task, allowing us to specify both the periodicity of the task, and the nature of the task itself. We limited ourselves to schedule basic payments, but if more functionality was added to the code, the automated tasks could feature arbitrary code, being able to perform much more complex tasks.

### Testing of the API calls

One more tool used when developing the API was Postman<sup>23</sup>. At its simplest form, it is a software that allows to make all kinds of HTTP requests, and observe the response of the server. Since we did not invest the time to create a frontend, we needed a way to test the functionalities of the API, without having to type in a web browser the full URI each time. By being able of easily defining the values included in the query, we

---

<sup>22</sup><https://github.com/node-cron/node-cron>

<sup>23</sup><https://www.postman.com/>

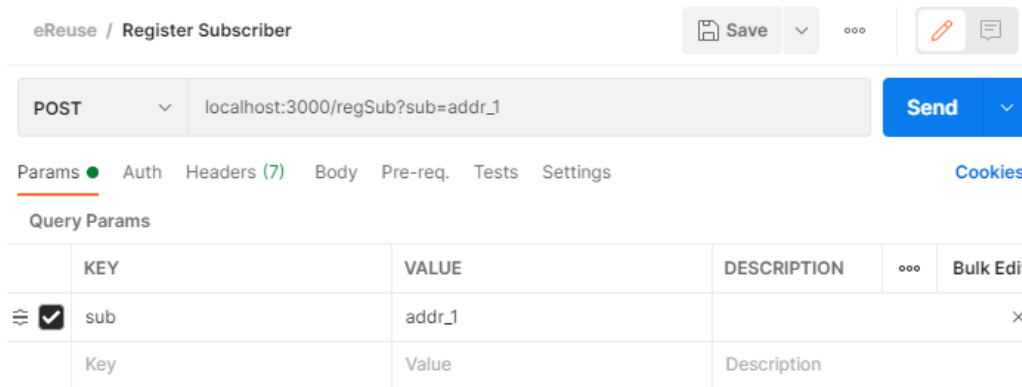


Figure 21: Postman request

could send request much more conveniently. Furthermore, the possibility to define environment variables made the handling of variables very similar to that of the code.

## Data storage and managing

For this project, we needed a convenient way to store the data of our system. Traditionally, this would have been a simple database, that connects to the API in order to perform basic CRUD operations. For this project, we decided to use the same approach that the research team had been using in a number of projects, that is to build a customised Docker<sup>24</sup> container. The reason why we took this path, was that we wanted to embrace virtualisation, and the benefits it brought. Besides, in our group, avoiding the repetition of work was of uttermost importance, and for this reason, building a parametrised module, that could be easily configured and recycled, was certainly convenient. In the configuration file, we defined two services that will be detailed in the next sections. With regards to the configuration file, we can first differentiate the db service.

- image: Docker image pulled from Docker Hub. In this case, we pulled the postgres image.
- environment: Variables that define the data to access the database, as well as the location where the data in the tables will be stored.
- volumes: Variable that links a logical Docker volume with the location to persistently store the data between container restarts. If we hadn't defined volumes, all the information stored by the container would be reinitialised in a hypothetical crash.

Although we have stated that the system proposed in this thesis is not meant to be a final product, the potential reusability of this configuration file in future projects made us take this approach. It was also a convenient decision, since it allowed us to have the data at our disposal even when the containers were not running.

- ports: Docker allows to map a port in the host with one in the virtualised machine.

<sup>24</sup><https://www.docker.com/>

```

version: '3'
services:
  db:
    image: "postgres"
    environment:
      - POSTGRES_USER=****
      - POSTGRES_PASSWORD=****
      - POSTGRES_DB=****
      - PGDATA=/var/lib/postgresql/data/
    volumes:
      - db-data:/var/lib/postgresql/data/
    ports:
      - "5432:5432"

  pgadmin:
    image: "dpage/pgadmin4"
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@ereuse.com
      PGADMIN_DEFAULT_PASSWORD: ****
      PGADMIN_LISTEN_PORT: 80
    volumes:
      - pgadmin-data:/var/lib/pgadmin
    ports:
      - "5433:80"
    links:
      - "db:pgsql-server"

volumes:
  db-data: # named volumes can be managed easier using docker-compose
  pgadmin-data:

```

Figure 22: docker-compose file

In a similar way, we have the pgadmin service:

- image: To build this service, we pulled the dpage/pgadmin4 image from Docker Hub.
- restart: Configuration variable that makes the policy restart when the container exits.
- environment: Similar to the previous service, here we defined the variables that would allow us to use the pgAdmin dashboard.
- volumes: Location to persistently store data.
- ports: Mapping of ports in the host and the virtualised machine.
- links: Reference to the db service, so the pgAdmin dashboard could establish a connection with the PostgreSQL backend.

## Organising the data

To organise the subscription data, and relate the entities between themselves, we decided to use PostgreSQL<sup>25</sup> because it had been the choice for past projects, and it fulfilled our needs successfully. When put in context, the database has the task of tracking the existing stakeholders in the system, as well as the relationships (the subscriptions) between them. For each subscription, it lists all the relevant information to control its payments, like the periodicity, or the amount to charge. The use done through this project doesn't give room to an exhaustive explanation, since the database was not the core of the model, but it is interesting to justify the choice of tables and keys used, as well as the possibility of having more than one mediator.

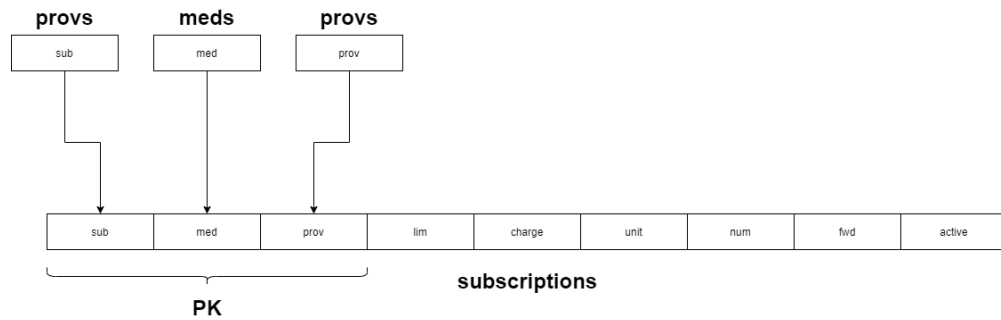


Figure 23: Primary Keys and Tables

The first important point is that the provs, meds and subs tables serve the purpose of making entities exist in the system and describing them, making it possible to relate them in the subscriptions table. In future iterations of this work, there may be the need to append additional information to a specific stakeholder, so we found that having a table for each type of entity in the architecture enabled an easy way of extending the definition of said entities, without necessarily affecting the subscriptions they were involved on. We did not store the token balance for each account here, because the smart contracts already did that. Then, for each of the three tables, the entity identifier is the primary key, and in the subscriptions table, they form the primary key, while referencing the corresponding entity in the other tables.

If we focus on the subscriptions table, we can see there are other fields beyond the primary key.

- **lim:** Effective upper limit in token operations, when the organisation moves tokens on behalf of an entity.
- **charge:** The next payment in the specific subscription. The corresponding provider must inform the organisation of the amount before it can be charged.
- **unit:** Granularity in the periodicity of payments. Added as a way to keep the system as much customisable as possible. By using this level of granularity, the automation engine in the server could define tasks that run exactly as the database states.

<sup>25</sup><https://www.postgresql.org/>

- num: For the time unit specified in the unit field, the value on which the payment occurs. For instance, if unit is month, and num is 5, it means that the payments will occur on day 5 on a monthly basis.
- fwd: This boolean value controls if the relevant Provider can withdraw the tokens deposited in the Mediator for this subscription. The Mediator does not allow this until the period in which a Subscriber can ask request for a refund is ended.
- active: Boolean value used to define if a subscription has an active automated payment task running. In case of being false, it would mean that the agreement exists, but is temporarily suspended for any reason the involved parts have discussed. Additionally, this field eases the process of restarting the services in case of failure, since the server would know the nature of the automated task, and its status before the crash.

## Easily handling the database

Although it was not an essential component of our model, during the development process we found out that an easy way to check whether the changes were happening in the database was needed. A simple command line query wouldn't have granted us that, so the next option was using a dashboard of some sort, to be able to execute queries and procedures without struggling with the command line.

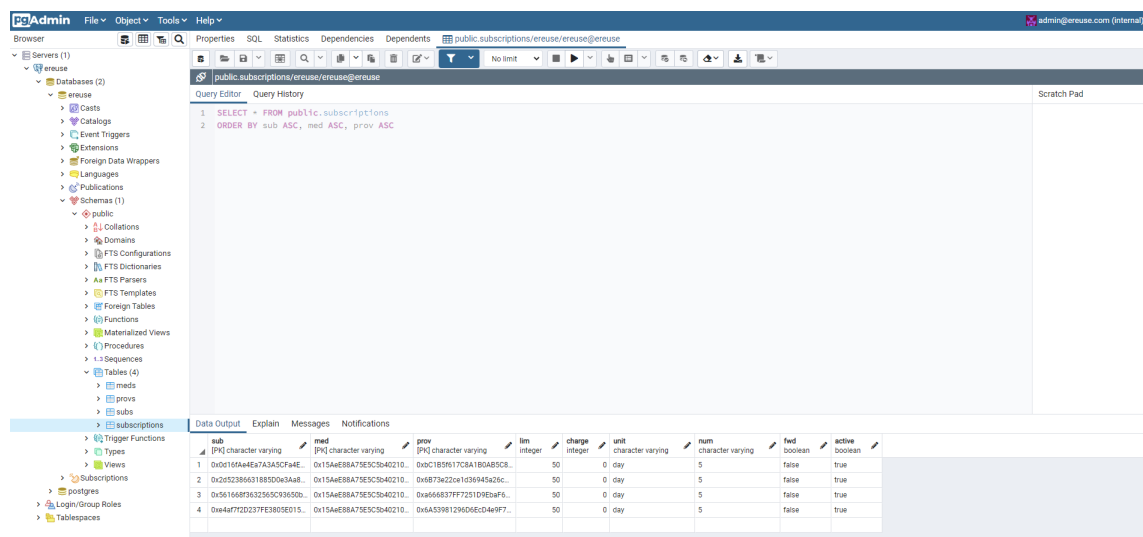


Figure 24: pgAdmin

In the image we can see an example of a query in pgAdmin<sup>26</sup>, that allowed us to observe the current data related with known subscriptions. This is a simple example, however by using the different options

<sup>26</sup><https://www.pgadmin.org/>

available, we could modify data by simply clicking in the table representation on the bottom of the GUI, or create custom queries and use them repeatedly in the development process.

## 6.2 Chronology of Tasks

As well as knowing the parts that compose the model, it is important to know the actual process by which it transitioned from an idea, to an actual prototype.

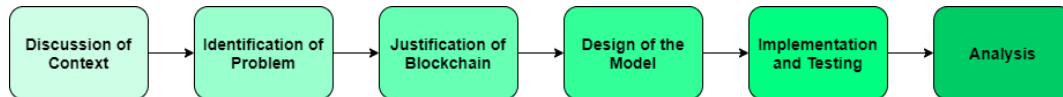


Figure 25: Chronography of the Project

The first step was to understand in detail the context in which the project resided. Not only NGIAtlantic, but rather the whole ecosystem of stakeholders, technologies and concepts relevant in the topic. Once this was understood, it was the moment of identifying the actual problem, and point out why was it caused, its effects, and potential solutions.

Having the background information documented, the design process could begin. In it, we reviewed how we could translate the needs that arose in the previous steps of the process, and plan how we could implement a prototype that fulfilled the known requirements. This is where most of the questions regarding the system appeared -since we had to make some assumptions related with the limitations of the technologies, and where we had to decide what the focus of the project would be.

During the implementation process, we started connecting each part in our design. At this time, we had to adapt to the situations that appeared, such as incompatibilities between technologies, or problems that made us modify our original plans and assumptions. In the end, we managed to implement a prototype, and test successfully if it was able to accomplish the intended tasks.

As a final step, we studied what experiments would better test the system, and started preparing the infrastructure that would make them possible. Once it was prepared, we designed test scripts and scenarios, in order to collect data, analyse how the system behave, and assess how suited it was for the original business case.

## 6.3 Security Considerations

In previous sections, we have already discussed the approach taken with regards to security. Being this an experimental prototype, its purpose is not to be a market product, ready for the end users. Thus, we prioritised the core system functionality and performance analysis, rather than its suitability as a final product. This is why we considered security, as well as other aspects like a front end, as secondary for this thesis. With that said, we did not neglect these aspects in design and development time. We estimated the dedication and benefits that investing time on them would yield, and decided to focus on the main aspects instead.

Examples of security in the project come in different forms. One example is the use of OpenZeppelin libraries for our Smart Contracts. While the main purpose for using them was not having to implement something already done and tested, internally the libraries use their own security measures, like SafeMath. This library implements simple mathematical operations, but does it in a way in which the possible attack surface in your code is reduced, avoiding vulnerabilities such as reentrancy. Another example is distributed architecture and the immutability of blockchain data, provided by the blockchain system. The idea is that while our system does not have security as a main axis, it certainly features secure components and practices across the whole infrastructure.



## 7 Analysis

*This section discusses how the system was analysed -what parameters were measured, by means of what, and why it is relevant. By offering numerical and performance-based conclusions, we can strengthen the conclusions of this thesis, so it is more relevant, and its impact is greater.*

### 7.1 Method used

At the beginning of the analysis, we were mostly concerned about being able to retrieve data from our testbed. Since at this point we were still defining what tests, metrics and graphics were most relevant. We started by creating a script that collected all the necessary environment information, created a transaction object, and sent it an arbitrary number of times to the testbed. This allowed us to have a first approximation of how the infrastructure would perform.

```
web3.eth.sendTransaction({
  from: accounts[0],
  to: tokAt,
  gas: 200000,
  data: tok.methods.transfer(accounts[2], 50).encodeABI()
})
```

Figure 26: Transaction object

To create the transaction object, we used the web3 `sendTransaction()` method. It allowed us to specify the account originating the transaction, the target address (in our case, the DDTOKEN contract), the maximum gas to spend, and the data. For this last field, we had to encode a token transfer function, so when the transaction arrived to the blockchain, the EVM would understand what we wanted to do, and the function would be executed. Following this approach, we could simulate scenarios where an arbitrary number of transactions was generated, and test how resilient the infrastructure and was.

```
.once('transactionHash', function (hash) {
  results[id]['hashTime'] = Date.now()
  results[id]['txHash'] = hash
})
```

Figure 27: Intercepting the events of a transaction

After creating the transaction, we sent it a predefined number of times, and used events to fetch valuable

information about each transaction sent. With this information, we created a JSON object, that we later saved for later analysis, and printed in console a summary of the experiment.

```
Completed 100 transactions in 1342ms.  
74.51564828614009 tx/s  
in 6 blocks: 798, 794, 795, 796, 799, 797,  
Average latency per transaction: 718.06ms  
Minimum transaction latency: 148ms  
Maximum transaction latency: 1315ms  
Successfully wrote file  
907425  
22575
```

Figure 28: Results of the first experiments

Above, we show how many transactions were performed per second, as well as the blocks in which they were located. Additionally, the output also features latency information, a confirmation of the correct creation of the JSON file, and the token balance for the accounts involved in the experiment.

After having a starting point from which to start our final experiments, we needed to find feasible ways of handling and representing the data, as well as meaningful experiments for performance evaluation. We decided that we were going to have the size of a batch of transactions as independent variable, and choose different metrics as dependent variables. Thus, the requirements we found for making the charts, as well as the solutions found, were the following.

- Repetitions: Due to the asynchronous nature of the test script, we needed a way to perform each experiment a number of times that didn't yield wrong or interrelated results. For this reason, we modified the script so it could send a single batch on each execution, or leave a certain amount of time between batches, depending on the experiment.
- Charts: We needed a simple way to generate and customise charts. Many libraries require some type of web front end to show the plots, but since our architecture didn't feature such interface, we needed to be able to simply generate a chart and export it as an image. We decided that QuickChart<sup>27</sup> could be a good option, because by taking advantage of its cloud service, we could programmatically specify how we wanted a chart to be, use the library to generate an URI, and then use that URI in our web browser to obtain the PNG image of the chart.

---

<sup>27</sup><https://quickchart.io/>

```

const myChart = new QuickChart();
myChart.setConfig({
  type: 'scatter',
  data: {
    datasets: [
      {
        type: 'scatter', label: 'Samples', data: rateSamples
      },
      {
        type: 'line', label: 'Median', fill: false, data: rateAverages
      }
    ]
  },
  options: {
    title: {
      display: true, text: 'Transaction Rate'
    },
    scales: {
      xAxes: [
        {
          display: true,
          scaleLabel: {
            display: true, labelString: 'Batch Size'
          }
        }
      ],
      yAxes: [
        {
          display: true,
          scaleLabel: {
            display: true, labelString: 'Transactions per second'
          }
        }
      ]
    }
  }
})
.setWidth(800).setHeight(400).setBackgroundColor('transparent');

```

Figure 29: Configuration of a chart

As the example shows, we were able to specify the main type of the chart, the datasets used, and some options like the title or the label for each axis. Regarding the datasets, it was possible to further configure them, by using different labels, chart types and data series on each one of them. In this case, we wanted to show in the same chart the measured samples for each value of the X, and the average Y value in each case.

- Formatting the data: After each replication of the experiment, we stored the values in a text file one line at a time, so we could later import the data in a more convenient way. In the plotting script, we read the data from the file, split it, sanitised it and converted it to JSON, since that was what QuickChart needed. To obtain the aggregated results, we filtered the data by its value in the X axis,

and then performed a simple median, which yielded a second data object. At this point, we had the two datasets needed to generate the charts.

In the sampling process, we observed that there were a number of outliers in each experiment. We could not determine what was the exact reason they existed, but provided that we ran the experiments remotely, probably they were caused by network latency across the internet and the own lab infrastructure. This led us to use a median rather than an average when aggregating results. While the former ignores sporadic outliers, the average used them in the calculation, affecting the aggregated value in the Y axis of our charts.

```
var rates, times
var timeSamples = []
var timeAverages = []
var rateSamples = []
var rateAverages = []

fs.readFile('times.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err)
    return
  }
  times = data.split('\n')
  times.pop()
  times.map(pair => {
    timeSamples.push(JSON.parse(pair))
  })
  console.log(timeSamples)

  steps.map(x => {
    let ys = timeSamples.filter(pair => pair['x'] == x)
    let partial = []
    ys.map(pair => partial.push(pair['y']))
    timeAverages.push(JSON.parse(`{"x":${x},"y":${math.median(partial)}}`))
  })
  console.log(timeAverages)
```

Figure 30: Preparing the input data

As a final comment on the method used, it is worth knowing the semantics of the scenarios. The goal of our system is to serve as a backend for a subscription service in a private blockchain environment. Taking this into account, along with the fact that the blockchain is used only in token movements, it made sense to tailor and limit the experiments to that specific moment in the system's workflow. Hence, the idea behind the experiments is that a number of accounts perform token transactions between themselves, and we read the metrics collected in the process. We started using only two accounts, but the system started showing some anomalous behaviour, related to the lost of sequential order in transactions sent by the same address. After investigating the possible reasons why this was happening, we found a number of possible reasons, but we finally chose to use several accounts as a solution. It effectively solved this problem, and made us able to broaden the possible use cases that our experiments could refer to.

## 7.2 Results

For our first experiment, we wanted to stress the system with batches of increasing number of transactions. Our goal here was to measure how the blockchain performed with an increasing transaction load. For this purpose, we conducted the experiment 10 times for each batch size, and calculated the median of all the sampled values. This experiment could be associated to a use case where the subscription system users are getting paid or charged at the same time, due to the agreed date recorded in the system. Measuring the system performance under this assumption was critical, since the system had to be able to process a relatively high number of almost-simultaneous requests. In this case, we conducted the experiments up to 1,000 transactions per execution. We decided to use this as upper bound, because beyond this point, we started seeing occasional anomalous behaviours on the system.

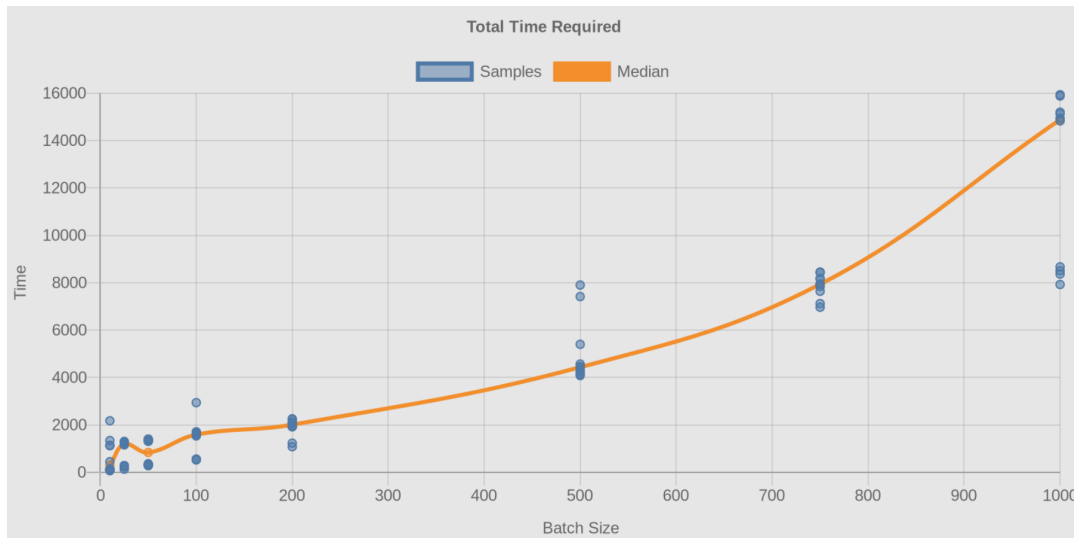


Figure 31: Time required for each batch

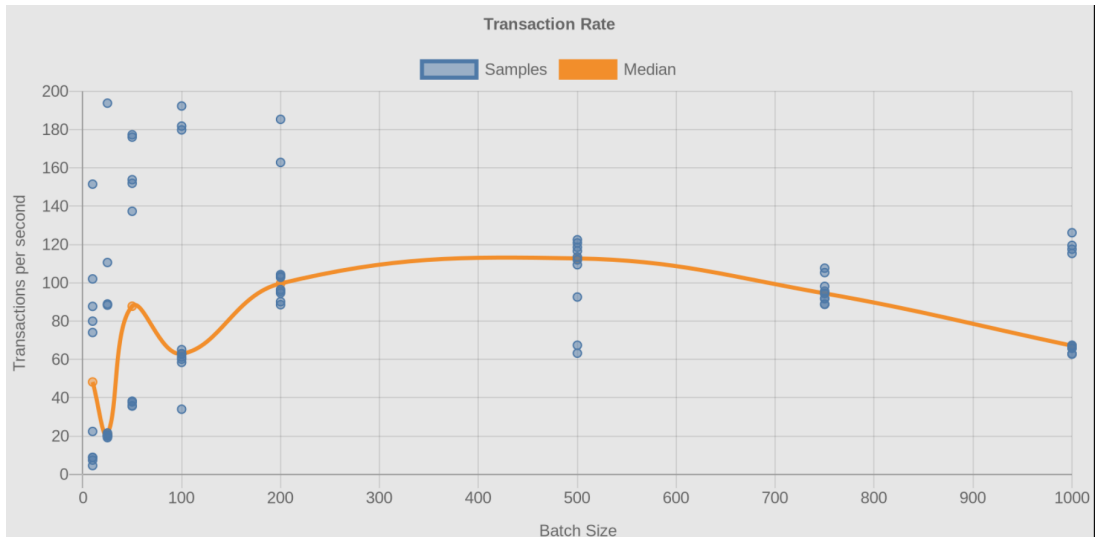


Figure 32: Rate of instructions per second

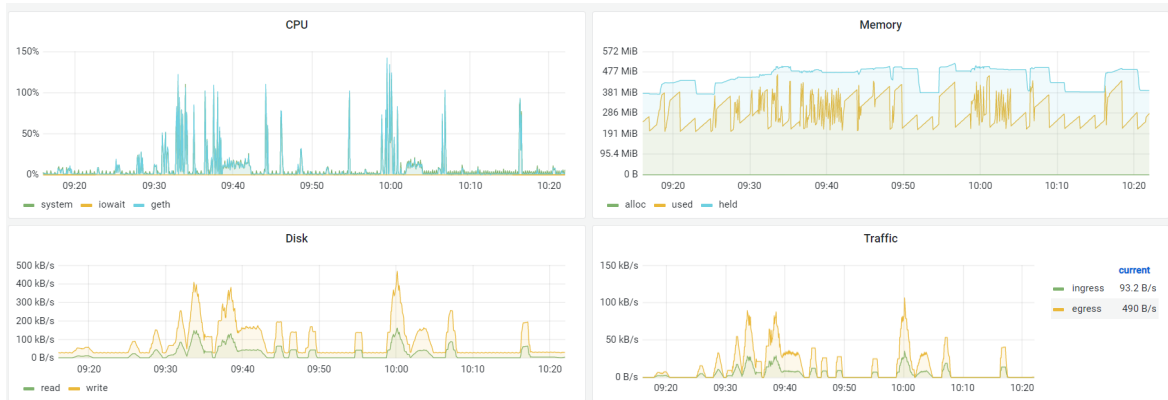


Figure 33: Resource usage in the first experiment

As we can see in the charts, the X axis represents the size of each batch of token transactions, while the Y axis represents the specific metric measured in each chart. The sampled measures can be seen coloured in blue on the charts. After calculating the median of the observed samples, we obtained an aggregated Y value that we included in the graphic, and linked them so the trend was easier to understand. Regarding the resource usage, we took advantage from the Grafana dashboard that the team built, and that was fed with the data provided by the Prometheus monitoring system.

For our second experiment, we wanted to assess what was the maximum number of transactions that

the blockchain could process per second. With this in mind, we decided to send bursts of transactions, separating the beginning of said bursts by one second, but not controlling when they ended. In effect, what this produced was a steady generation of bursts, and as the size of the burst grew larger, the blockchain started overlapping one set of transactions with the next. The idea was testing at which point the processing power of the blockchain became insufficient. We would know that by either a high resource usage, or by seeing anomalous behaviours as a consequence of the collapse of the equipment. We first used HTTP as transport protocol.

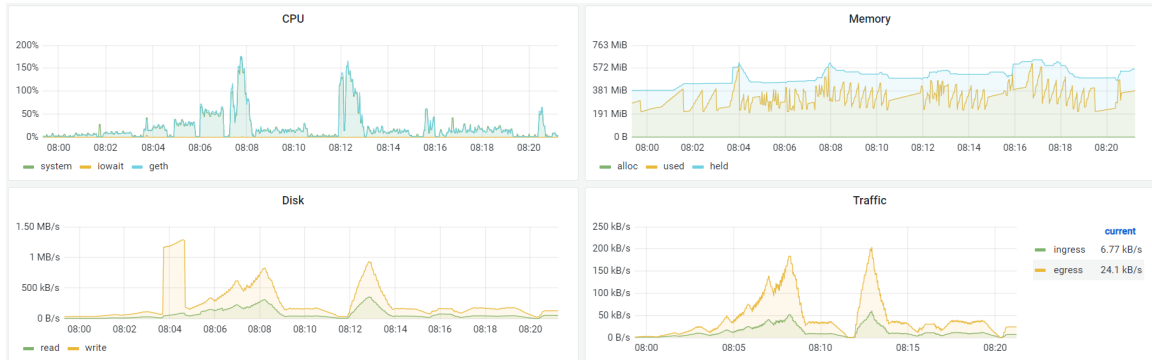


Figure 34: Resource usage in the second experiment (HTTP)

And afterwards repeated the same exact experiment using WebSocket instead.



Figure 35: Resource usage in the second experiment (WebSocket)

### 7.3 Discussion

After conducting the experiments, we were able to take insights as to why we observed such figures. Due to the remote conditions on which we had to conduct the experiments, there were several factors that were out of our control, like the latency in the consensus process, the delay caused by the propagation of messages through the Internet, or the internal behaviour of the blockchain algorithm. This introduced some variance and outliers, that can be observed in all of the charts. This is part of the reason why we decided to use the median as an aggregator function for our sampled values.

Regarding the first experiment, in a real scenario similar to the one we assessed, the system's performance would be high enough to meet the needs. The time function suggested that the evolution of the curve was super-linear, with an increasing slope as the size of the batches was incremented. Furthermore, the rate function was somehow symmetrical, although it featured some interesting differences. For smaller batch sizes, it showed high degree of variance, probably due to the way the blockchain closed the blocks. As the size of the bursts became larger, the values began stabilising, and we could see that the rate of transactions per second decreased more with each increment in the burst size. The resource usage provided by Grafana suggest that the load did increase in every possible aspect during the execution of the experiments, but was far from reaching its maximum value. The maximum CPU usage measured was of the 142% out of an available 400%, since we had 4 cores available in the node, and the memory, disk and traffic also showed a remarkably good performance, far from compromising the availability and timeliness of the system. Finally, it is worth mentioning that at the end of our experiment, the values measured suggested a quick decrease in the performance for bigger batch sizes. This leaves the door open to considering how large a batch of transactions the system would be able to process without collapsing.

Regarding the second experiment, the resource usage observed in the Grafana dashboard suggested a maximum effective throughput of approximately 200 transactions per second using HTTP. This gives enough room for a successful system behaviour, but should be taken into account when studying the production scenarios the system would need to face. If the average expected load is supposed to be close to that value, there is the risk of a service outage, so further measures have to be taken with regards to this. If the load is expected to be lower, the risk is somehow relative, but having such a defined limitation is a symptom that the system still needs to be polished and optimised, or the use case revised. The previous is derived from the readings provided by Grafana, as well as the fact that the own dashboard started malfunctioning when we arrived to a certain threshold. In the resource usage plots we can see how the CPU reaches a maximum at around the 150% of the capacity. That corresponds to the experiments with a burst size of 200 transactions. The processing power used had been growing steadily (as can be seen by the different valley areas in the plot), but after 200 transactions per second, the `geth` process collapsed, the logging of the testing script started showing errors, and the entire dashboard crashed. If we go back to the plots from the first experiment, we will see that some outliers indicated a maximum of around 200 transactions per second, which is consistent with the situation being described here. From these facts and results we can conclude that there is a breaking point in the performance of the system, and that its correct workflow cannot be guaranteed with such a load.



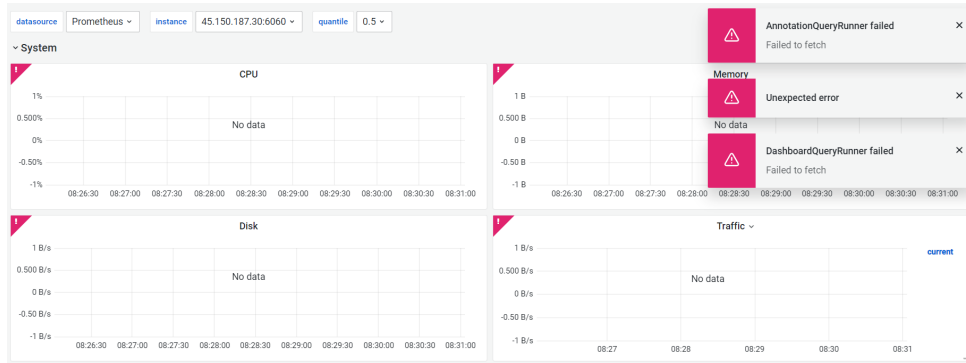


Figure 36: Grafana failure

With WebSocket, Grafana showed a substantially different behaviour. The resource usage grew in a similar way than it did with HTTP, but after bursts of 200 transactions, the geth process did not collapse, nor did Grafana. We were able to keep observing the charts evolving, and even increase the burst size to 750 and 1000 transactions without any critical failure appearing. While significantly positive, the values also showed that, after 200 transactions per burst, the CPU consumption trend reversed, the system began consuming less CPU overall as we increased the independent variable, and the logging showed a clear overlap between bursts. That might be caused by the sequentiality with which the geth process itself processes the incoming requests, or a need to revise the architecture and configuration of the distributed Ethereum ledger.

For the second experiment, we could safely conclude that the current state of the infrastructure limited the correct functionality to 200 transactions per second. Beyond this threshold, the system began showing undesirable behaviours with both HTTP and WebSocket.

## 8 Conclusions

*We finally draw some insight from the whole experimentation process, and formulate it as a set of statements, that clearly summarise what we learnt, and proved, during this project. Additionally, we offer ideas for further research on this topic, and mark the point up to which this thesis has worked in a number of aspects.*

In this project, we proposed a modular architecture that enables a blockchain-based subscription payment system. We integrated the technologies, and tested the correctness of the system as a whole. To complete the analysis, we experimented with the private blockchain and the implemented smart contracts, so we could assess how feasible the approach was. Based on the results, the infrastructure as a whole proved itself both feasible and convenient. While functional in certain scenarios, the private blockchain behave undesirably at high stress levels, so at this point, the availability of the service can not be guaranteed under all possible scenarios. In the future, the configuration of the private blockchain could be improved in terms of stability of the service at high loads, and consistency when using different transport protocols. Additionally, the functionality of the off-chain model could be extended to support more complex management capabilities, as well as interfacing with end-user interfaces, more suited for production scenarios.

### 8.1 Future Work

To make it easier for future research to continue working on this topic, we will enumerate different aspects of this project, and describe at which state they have been left, and how to further improve them.

- **General Model:** Possible improvements of the current system may include, for instance, a more refined automation engine, that is able to perform more advanced tasks, or a more complete database architecture. For example, by adding stored procedures to make the table handling easier, or by extending the definitions of the stakeholders, to make them more detailed and complex. Additionally, as the production scenario for this scenario becomes clearer, the interaction between the stakeholders could impose changes in who accessed the blockchain, and how. This may lead to a modification in the capabilities that the central organisation has regarding accounts and token handling.
- **Autonomous Blockchain:** Regarding the location of nodes, it would be interesting to assess how distance affects the system performance, and how the problem should be tackled in a global scenario. Besides this, the internal parameters of the decentralised ledger could be revised and optimised, to tackle the challenged observed in the experiments.
- **Security:** As stated in previous sections, since this project did not aim to produce a market-ready system, there were certain features that were out of the scope of our research. A future version of this work may include explicit security measures, as well as authentication and authorisation mechanisms. How the system evolves towards an end-user application will dictate the final security requirements.
- **End User Accessibility:** The work done in this thesis is contextualised in a laboratory environment, where the people involved had a level of technical skills that allows them to interact with the system in a more direct manner. That is why we did not find the significance of developing a front end for our tests. A possible evolution of this project may take end user needs more into account, and develop a front end, in such a way that it is not necessary to be an expert in the field to use the system.

Overall, this project worked to develop the core of the system and assess its functionality while implementing additional features like the database and the automation engine.

## References

- [1] Fahad Ahmad Al-Zahrani. 2020. Subscription-Based Data-Sharing Model Using Blockchain and Data as a Service. *IEEE Access* 8 (2020), 115966–115981. <https://doi.org/10.1109/ACCESS.2020.3002823>
- [2] Earl Cook. 1976. Limits to Exploitation of Nonrenewable Resources. *Science* 191, 4228 (1976), 677–682. <http://www.jstor.org/stable/1741483>
- [3] eReuse. 2021. eReuse. <https://www.ereuse.org/>
- [4] Ethereum.org. 2021. Ethereum Whitepaper. <https://ethereum.org/en/whitepaper/>
- [5] Ellen McArthur Foundation. 2017. *What is the circular economy?* Ellen McArthur Foundation. Retrieved April 1, 2021 from <https://www.ellenmacarthurfoundation.org/circular-economy/what-is-the-circular-economy>
- [6] David Franquesa and Leandro Navarro. 2018. Devices as a Commons: Limits to Premature Recycling. In *Proceedings of the 2018 Workshop on Computing within Limits* (Toronto, Ontario, Canada) (*LIMITS '18*). Association for Computing Machinery, New York, NY, USA, Article 8, 10 pages. <https://doi.org/10.1145/3232617.3232624>
- [7] David Franquesa, Leandro Navarro, David López, Xavier Bustamante, and Santiago Lamora. 2015. Breaking Barriers on Reuse of Digital Devices Ensuring Final Recycling. In *29th International Conference on Environmental Informatics, EnviroInfo 2015 / 3rd International Conference on Information and Communication Technology for Sustainability, ICT4S 2015, Copenhagen, Denmark*. Atlantis Press, Amsterdam, Netherlands, 281–288.
- [8] Leandro Navarro. 2021. *Circular economy research*. UPC. Retrieved May 1, 2021 from <https://dsg.ac.upc.edu/circular/> Distributed Systems group.
- [9] Obada. 2021. The Open Blockchain for Asset Disposition Alliance. <https://www.obada.io/>
- [10] Yustus Eko Oktian, Elizabeth Nathania Witanto, Sandra Kumi, and Sang-Gon Lee. 2019. Block-SubPay - A Blockchain Framework for Subscription-Based Payment in Cloud Service. In *2019 21st International Conference on Advanced Communication Technology (ICACT)*. IEEE, 153–158. <https://doi.org/10.23919/ICACT.2019.8702008>
- [11] Waterford Institute of Technology. 2021. *NGIatlantic.eu - A Collaborative platform for EU-US Next Generation Internet Experiments*. Next Generation Internet program. Retrieved May 1, 2021 from <https://ngiatlantic.eu> EU funded project.