Electro Points: A blockchain based loyalty system for vending machines owned by different merchants

André Soares Amaral Mendes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Miguel Ângelo Marques de Matos
Eng. Jorge Tiago Melo Barbosa da Silva e Vila

Examination Committee
Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Ricardo Jorge Fernandes Chaves

November 2019
To my best friend, to whom I owe everything that I am today
Acknowledgments

First I want to thanks my supervisor, Miguel Matos, for being willing to be part of this project, for all the knowledge and feedback provided, for being comprehensive and understanding when I was unable to report progress. I want to thanks Jorge Vila and Elecetco, for giving me the opportunity to work with them on this project, and for all the cursed Javascript knowledge I have today.

I want to thanks my University, for providing me with the knowledge required to finish this document, and most importantly required to shape who I am today.

I want to thanks my colleagues that were always by my side during my academic life, with whom I shared many special moments, and that kept me on track and always willing to learn. Special thanks to Nuno, Pedro for the late night projects, and Nelson, Genebra for all the skate sessions we shared together.

I want to thanks my Elecetco co-workers, for keeping me entertained during work hours, always willing to help me when required, and for all the meals we shared together. Thanks Filipe and Rúben for the late night gaming sessions and all the keyboard and Nintendo Switch discussions.

I want to thanks the online communities I am part of, for following my progress and for helping me procrastinate with random projects when I should have been working on this thesis. Without you I would probably have finished this document a long time ago.

I want to thanks my friends and family, for not letting me give up on this thesis when I thought it was the logical step to take given life circumstances. Thanks mom and dad for always supporting me when I was unmotivated and unwilling to move forward. Thanks Beatriz, specially in the last few chapters, for cheering me up when I was frustrated.

Last but not least, I want to thanks my best friend (you know who you are), to whom I owe everything that I am today. If I would have never met you, I am certain that I would not be writing this document at this very moment.
Resumo

As máquinas de venda não evoluíram muito, entregam produtos quando lhes é dado dinheiro. Numa altura onde os meios de pagamento digitais estão em crescimento, esta falta de modernização fez com que empresas desenvolvessem soluções que permitem melhorar as funcionalidades das máquinas de venda.

A Elecctro desenvolve uma solução que permite a qualquer máquina de venda, de aceitar pagamentos digitais. Esta solução é vendida a comerciantes que pretendem melhorar as suas máquinas.

A Elecctro reparou que a percentagem de pagamentos digitais em máquinas que já implementam esta solução ronda os 25%, e queriam aumentar este número.

Para resolver este problema, introduzimos um programa de fidelidade que recompensa clientes com pontos, se usarem pagamentos digitais quando compram produtos com a solução da Elecctro. Visto que estas máquinas pertecem a comerciantes de organizações diferentes, a falta de centralização é problemática.

Precisamos de uma solução distribuída que permita que os comerciantes colaborem sem terem que confiar uns nos outros. Devido ao elevado volume de vendas diário, o programa de fidelidade tem de ser capaz de recompensar muitos clientes em paralelo.

Propomos uma solução baseada em blockchain, em cima de Hyperledger Fabric [1], que serve como uma única fonte de verdade, que guarda os pontos de cada cliente e como estes mudam ao longo do tempo. Apesar de não conseguir servir milhares de clientes por segundo, é um passo em frente para uma implementação que permite atingir uma taxa de débito elevada, com uma latência menor que o tempo de espera tolerável de um cliente.

Palavras-chave: Máquina de venda, Programa de fidelidade, Blockchain, Hyperledger Fabric
Abstract

Vending machines remained the same throughout the years, dispensing products when given cash. In a world where digital currencies are getting more prominent, this technology gap has led companies to develop solutions that enhance the functionality of vending machines.

Electro develops a solution that turns vending machines into "smart" vending machines, capable of accepting other payment methods. Electro offers this solution to merchants that want to boost the functionalities of their vending machines.

Electro noticed that the percentage of cashless vends in vending machines that already deploy their solution is around 25%, and they wanted to increase this value.

To solve this problem we introduce a loyalty system that awards points to customers who use cashless payments when purchasing products using Electro’s solution. Since these vending machines belong to merchants from different organizations, the lack of centralization is problematic.

We require a distributed solution that allows merchants to cooperate without necessarily trusting each other. Due to the high volume of daily vends, the loyalty system needs to be capable of serving many customers in parallel.

We propose a blockchain-based solution, on top of Hyperledger Fabric [1], which serves as a single source of truth, that keeps track of all customer’s points and how they change over time. Although unable to serve thousands of customers per second like intended, it is a step towards an implementation that attempts to achieve a high request throughput while maintaining a latency lower than a customer’s tolerable waiting time.

Keywords: Vending machine, Loyalty system, Blockchain, Hyperledger Fabric
# Contents

Acknowledgments ........................................ v
Resumo ................................................... vii
Abstract ................................................... ix
List of Tables ........................................... xiii
List of Figures ........................................... xv
Nomenclature ............................................. 1
Glossary ................................................... 1

1 Introduction ........................................... 1
   1.1 Objectives .......................................... 3
   1.2 Thesis Outline ...................................... 5

2 Background ............................................ 7
   2.1 Vending Machine ................................... 7
   2.2 Vending Business ................................... 8
   2.3 Electro’s Vending Platform ......................... 8
      2.3.1 Vending Hub ................................... 9
      2.3.2 Communication Infrastructure ................. 11
      2.3.3 Customer interaction with a vending machine 11
   2.4 Blockchain ......................................... 14
      2.4.1 Blockchain definition .......................... 14
      2.4.2 Public vs Private Blockchain .................. 15
      2.4.3 Permissioned vs Permissionless Blockchain .... 16
      2.4.4 Bitcoin ...................................... 17
      2.4.5 Bitcoin Lightning Network ..................... 19
      2.4.6 Ethereum ...................................... 20
      2.4.7 ERC-20 Token .................................. 21
      2.4.8 Raiden Network ................................ 22
2.4.9 Have a snack, pay with Bitcoins ........................................ 22
2.4.10 Hyperledger Fabric ......................................................... 23
2.4.11 Summary ................................................................. 24

3 Electro Points ................................................................. 27
  3.1 Proposed architecture ..................................................... 27
  3.2 Orderer ............................................................................ 29
  3.3 Customer CA ................................................................. 29
  3.4 Chaincode ................................................................. 30
    3.4.1 machines chaincode .................................................. 31
    3.4.2 rates chaincode ...................................................... 32
    3.4.3 elepoints chaincode ................................................ 33
  3.5 Merlin .............................................................................. 34
  3.6 ElePoints ........................................................................ 35
  3.7 Components interaction ................................................... 36
    3.7.1 Customer creating and using accounts .......................... 36
    3.7.2 Customer transferring points ..................................... 37
    3.7.3 Merchant awarding points to customer ......................... 38
    3.7.4 Merchant updating rates and vending machines .............. 38
    3.7.5 Customer purchasing a discount ................................. 39

4 Evaluation ................................................................. 43
  4.1 Elen .............................................................................. 43
  4.2 Pupper .......................................................................... 44
  4.3 Testing environment ........................................................ 45
  4.4 Workload composition ...................................................... 46
  4.5 Results ............................................................................ 46
  4.6 Discussion ...................................................................... 51
  4.7 Scaling issues ............................................................... 52

5 Conclusions ................................................................. 53
  5.1 Achievements ............................................................... 53
  5.2 Future Work ................................................................. 54

Bibliography ............................................................. 55
List of Tables

2.1 Comparison between studied solutions ......................... 26
# List of Figures

2.1 Vending business layers ......................................................... 8  
2.2 Electro’s Vending Infrastructure ............................................. 10  
2.3 Customer interacting with a vending machine via payment terminal .......... 12  
2.4 Customer interacting with a vending machine via Electro’s Facebook Messenger 
   chat bot .................................................................................. 13  
2.5 Instance of a blockchain with 3 blocks ......................................... 14  
3.1 Electro Points’s architecture ..................................................... 28  
3.2 Customer creating/enrolling an account ......................................... 37  
3.3 Customer A transferring points to Customer B .................................... 37  
3.4 Merchant awarding points to a customer .......................................... 38  
3.5 Merchant updating rates .......................................................... 39  
3.6 Sequence diagram of a merchant adding/removing vending machines .......... 39  
3.7 Customer purchasing a discount .................................................. 41  
4.1 Throughput over time of the query only workload with using 30 DEV-L machines, 
   with 30 merchant nodes and 30 Pupper instances ............................... 48  
4.2 Average latency time of the query only workload with using 30 DEV-L machines,  
   with 30 merchant nodes and 30 Pupper instances ............................... 48  
4.3 Throughput over time of the mixed workload with using 5 GP-L machines, with 
   5 merchant nodes and 5 Pupper instances ......................................... 49  
4.4 Average latency over time of the mixed workload with using 5 GP-L machines, 
   with 5 merchant nodes and 5 Pupper instances .................................. 50  
4.5 Throughput over time of the mixed workload with using 5 GP-L machines, with 
   30 merchant nodes and 5 Pupper instances ....................................... 50  
4.6 Average latency over time of the mixed workload with using 5 GP-L machines, 
   with 30 merchant nodes and 5 Pupper instances ................................ 51
Chapter 1

Introduction

According to the European Vending Association, in 2015, the vending market generated an yearly revenue of 14.6 billion Euros across 33.42 billion vends, issued by 3.82 billion food and beverages vending machines [2]. Traditional vending machines are only able to accept cash as payment, apart from more modern ones which might accept other forms of payments such as credit and debit cards. This lack of modernization, in a world full of digital currencies, with phones being able to issue payments with technologies like Apple Pay [3] and Google Pay [4], led companies to develop solutions that try to close this technology gap.

One of these solutions is Electro’s vending platform. Electro offers this vending platform as a service for different merchants that are interested in boosting the features of their vending machines. With Electro’s solution, merchants are able to turn their vending machines, into ”smart” vending machines that are connected to the Internet. With an Internet connection vending machines are capable of reporting telemetry data and accepting other forms of payment that require a network connection.

Merchants are widely interested in Electro’s solution since it unlocks the ability to track the state of any vending machine in real time, as well as reducing the amount of cash a machine holds because of the additional cashless payment methods. This helps merchants plan their restocking and cash collecting routes, thus reducing transportation and maintenance costs.

As vending machines are often in unattended commercial zones, depending on the time of the day, they become easy targets of robbery and vandalism. Electro’s vending platform was extended with active security measures to help prevent these. An example of these measures is the existence of an alarm inside the vending machine, that is triggered when someone unauthorized opens the vending machine or when the vending machine gets tackled. Another example of these measures is a camera that takes pictures if it detects that the vending machine was improperly opened. The vending platform also has a GPS system that is capable of tracking
Despite the numerous amount of security measures built into Elecctro’s vending platform, a vending machine may still be robbed. If this happens a vending machine needs to be repaired, or even worst, fully scraped and replaced. The best way to stop robbery is to tackle the motivation behind it all. If Elecctro was able to remove the money off vending machines then this motivation would cease to exist for the most part.

As mentioned, Elecctro’s vending platform already provides cashless payments, but they noticed that only 25% of vends are cashless, meaning that customers would rather use cash than attempt the cashless variant. Elecctro could wait to see if adoption increases as their solution spreads among merchants and vending machines, but they realized that it might take some time, and decided to take a more proactive approach towards this problem.

In this thesis we introduce the creation of a loyalty system on top of Elecctro’s platform, that rewards customers with points when purchasing products using cashless payments. This motivates customers to consider the cashless route, thus reducing the amount of cash held by a vending machine at any point in time. Since Elecctro offers their platform as a service to a variety of merchants, we expect merchants to see this loyalty system as an opportunity to increase their customer base. Below we describe examples of beneficial scenarios that this new loyalty system can bring to merchants.

**Scenario 1.** Alice arrives late to work and she is not able to eat her breakfast at home. When she gets to work she purchases her usual cup of coffee from a vending machine that belongs to Merchant A. She instantly gets rewarded with points from purchasing using PayPal. Then she goes to a snacks vending machine that belongs to Merchant B. Not having enough PayPal balance to cover the full price of any product, Alice checks how many points she has, and realizes she qualifies for a discount in this machine. Alice notices that she just got enough points to purchase some cookies for a quarter of the price, which she can now purchase thanks to the discount. She then applies the discount, and is able to purchase the cookies right away, using the amount in her PayPal balance.

In the first scenario, if it was not for the points Alice got, she would have never bought any product from the second vending machine, rendering no profit to Merchant B.

**Scenario 2.** Bob is feeling like a snack, but he does not have enough money to cover the full price of a product. Bob asks Alice if she could help him, to what Alice responds that the vending machine downstairs has a chocolate with a discount
if he uses some points. Bob does not have enough points, so Alice transfers him the necessary amount of points to purchase the chocolate with a discount.

In the second scenario Alice is able to help Bob, and the vending machine’s merchant gains a new customer.

This loyalty system may seem prejudicial to merchants as they are allowing customers to get discounts in their products using points issued by other merchants. To prevent that, we utilized Electro’s telemetry capabilities to use the product’s expiration date to calculate the percentage of discount any point in time. This allows merchants to dispose of products whose expiration date is close, which when reached would render them useless and illegal to sell.

To sum it all up, the benefits merchants would get by partaking in this loyalty system are:

- Motivation for cashless vends, which would mean less robbery and as well as less maintenance and transportation costs when performing cash collections
- A partnership with other merchants, which could help increase their customer base
- Dispose of products whose expiration date is close

Given the number of vends in 2015 and the instant nature of vending machines, we need a scalable loyalty system, capable of rewarding many customers at the same time, while doing it instantly so it would not affect the user experience when earning/using points.

Because these merchants are usually from different institutions, the lack of centralization is problematic. Having different merchants, that do not necessarily trust each other, but all share the same goal, is a problem that is amenable to blockchain based solution. With a blockchain, no merchant would be responsible for all the points and rewards of the system, allowing merchants to reach consensus on how many points does each customer have, as well as having their own point rewards schemes that are transparent to both customers and merchants.

The goal of this thesis is therefore to design, implement and evaluate Elecctro Points, a blockchain based loyalty system for vending machines that belong to different merchants and that would allow them to participate without fully trusting each other.

1.1 Objectives

As a distributed loyalty system, our main goal is to achieve instant point rewards and instant usage of points. We consider "instant" to take no more than a customer’s tolerable waiting time which is around 4 to 5 seconds [5]. This would benefit the overall user experience of the customer, which, if unsatisfied, will not use the system at all.
Another requisite is the amount of requests our system needs to serve. According to the European Vending Association [2], in 2015, there were over 90 million vends per day across food and beverage vending machines throughout Europe. Electro’s focus in Europe is in Spain, Germany, France, United Kingdom and Italy, which among themselves have around 50 million vends per day. According to statistics of vending machines that already deploy Electro’s solution, splitting the day in one hour periods, we have that the biggest peak of the day is during lunch break and corresponds to roughly 8.3% of the total number of vends that happen throughout the day. That extrapolates to 1,152 vends per second (8.3% of the 50 million daily vends) during lunch break, if evenly distributed among that hour. Currently Electro’s solution holds an adoption rate of 25% in cashless vends but we should expect this number to increase over time, and therefore need to account for potential growth. If that number would increase to something like 80%, we would be looking at roughly 922 point transfers per second (80% of the extrapolated 1,152 vends per second).

Another aspect we need to keep in mind is the amount of merchants we need to support. According to Electro’s internal reports, there are 22 main merchants (500+ vending machines) across the European countries mentioned above. Merchants need to also be able to customize which vending machines and products should partake in the loyalty system. They also need to be able to create the rules that dictate the amount of points a product rewards based on its price, and the amount of discount a product has based on its expiration date. Ideally this would happen in a transparent way, in hopes that merchants would self-regulate the system by consulting each other’s rules.

Our system also needs to ensure customers are able to transfer points among themselves.

To sum it all up, our loyalty system needs to:

• Take no longer than 5 seconds when it comes to reward and use points

• Serve at least 950+ requests per second

• Support around 30 merchants with any number of vending machines to participate

• Allow merchants to select which vending machines and products should partake in the loyalty system

• Allow merchants to control products rewards and discounts based on their price and expiration date respectively

• Allow customers to transfer points to other customers
1.2 Thesis Outline

The remainder of the document is structured as follows. In Chapter 2, we delve into what is a vending machine, the vending business and discuss Elecctro’s current vending platform. We also analyse the more prominent blockchain systems as well as some work done in the micro-payments and vending area. Chapter 3 describes the implementation of our solution, the reasoning behind our decisions and how everything can be integrated with the Elecctro’s current platform. Next, in Chapter 4, we discuss the techniques used to evaluate our implementation and discuss the results of these evaluations. Finally, in Chapter 5, we finish with a conclusion and closing remarks.
Chapter 2

Background

In this chapter, we introduce some topics that will ease the transition into our implementation. We first start by explaining what a vending machine is, followed by how the vending business is composed. Then we go through Electro’s Vending Platform and its components to give some insight on how their platform turns vending machines into ”smart” vending machines. Last but not least, we define what a blockchain is and analyse some of the main blockchain systems that are currently deployed, as well as some work done in the vending and micro-payments area, summarizing them all in the end.

2.1 Vending Machine

A vending machine is an automated and unattended machine that sells products, such as snacks and beverages to consumers in exchange of any form of currency, usually cash. A few also accept cashless payments like credit/debit cards, and some even accept some special tokens that are usually related to workplace environments that provide their workers benefits, like free coffee.

Vending machines usually do not require any network connection, and only necessitate electric power to function. Interacting with one is as simple as inserting currency, selecting a product, waiting for it to be delivered and retrieving the change in case the amount of money introduced was more than the cost of the selected product.

As any unattended machine, vending machines require maintenance, to be restocked and the cash within them to be collected. This need entails the intervention of a maintainer, usually designated by the merchant of the machine, when it is not the merchant himself. The recurrence of such maintenance depends on the location of the vending machine, and is always uncertain, since the maintainer can only guess how many products and cash does a vending machine hold at a given point in time.
2.2 Vending Business

From Electro’s perspective the vending business is segmented in 3 main layers that can be seen in Figure 2.1.

The first layer is the vending machine manufacturers. They are the ones responsible for manufacturing the vending machines. These are then sold to the second and third layer - the merchants.

Merchants are devised in two layers. What differentiates each layer is the number of vending machines a merchant owns. Electro considers a merchant to be big, when the merchant has more than 500 vending machines under its possession.

Electro targets the big merchants. The reason why they do not target manufacturers, is because vending machines last decades before being replaced, which means Electro’s solution would not be seen until the majority of vending machines started being replaced. So, by targeting the merchants with 500+ machines, Electro is able to roll its solution to the consumer easily and faster, allowing merchants to modernize their machines without having to replace them with better and more expensive models. Bigger merchants can later have deals with medium and small merchants so they can have access to Electro’s vending platform.

Electro’s focus in Europe is in Spain, Germany, France, United Kingdom and Italy, which among themselves have around 2.66 million food and beverage vending machines [2]. The majority of them are owned by major merchants.

These machines in 2015 granted around 10 234 million euros in revenue throughout the whole year, issuing 50 million vends per day [2].

2.3 Electro’s Vending Platform

Electro’s vending platform connects a vending machine, no matter how old it is, to the Internet, allowing the development of management functionalities, new client interactions and even active
security measures like alarms on impact.

By connecting the vending machine to the Internet, merchants are now able to access information remotely which before was only available if they had physical access to the machine. This information is available by consulting a dedicated web or smartphone application. There they can list all their vending machines, their GPS locations, the amount of currency within them, their vends, product’s quantities and product’s expiration dates, among others. Not only merchants can access this information, they can also interact with the vending machines. This unlocks features like setting the price of each product remotely, or turning off the active security measures on demand in-case an unexpected maintenance occurs.

Another consequence of connecting the vending machine to the Internet is that consumers can now use cashless payments with a variety of payment gateways that were not available before. They are also able to start a dialogue with Electro’s Facebook Messenger [6] bot, Spencer, to purchase a product from a vending machine, as well as consult their account purchase history, nearby vending machines and the product’s nutritional information.

All of this is possible because of a major component in Electro’s solution, the vending hub. Composed by an Android device and a custom made controller board, it provides the interface between the information that the vending machine reports and Electro’s servers. It also provides the interface for the servers to interact with the vending machine.

This component is described in further detail in Section 2.3.1. Then, in Section 2.3.2, we describe the communication infrastructure that connects the vending hub to the servers in a bidirectional manner. Finally, in Section 2.3.3, we demonstrate the interaction between a customer and a vending machine that uses Electro’s solution.

2.3.1 Vending Hub

Electro’s vending hub is a component that provides an interface between the vending machine and the servers in a bidirectional manner. As shown in Figure 2.2), it is composed by an Android device, and a custom made controller board. The Android device manages all communication with both the servers and the controller board, using a dedicated application.

The board is responsible for all low-level interactions with the vending machine as well as peripherals that might be installed in it (alarm on impact, among others). The board is also responsible for providing power to the Android device and all the peripherals.
Figure 2.2: Electro’s Vending Infrastructure
2.3.2 Communication Infrastructure

Elecctro’s current infrastructure connects merchants, chat bot customers, vending hubs and servers together can be seen in Figure 2.2.

Merchants are authenticated and communicate with the system through a public API (Services API in Figure 2.2). This API is used by a Web front-end developed by Elecctro, that allows merchants to query and interact with their vending machines using any browser of their choice.

Chat bot customers are authenticated through Facebook Messenger [6]. Elecctro uses a unique generated identifier (page-scoped identifier or PSID) provided by Facebook that is only valid in the context of the conversation between the customer and the chat bot. We delve more into how chat bot customers interact with Elecctro’s vending platform in Section 2.3.3.

2.3.3 Customer interaction with a vending machine

Customers have two main flows of execution they can choose from when interacting with a vending machine that implements Elecctro’s solution. Both of them require the customer to start the flow, and both of them, when successful, result in a product being dispensed and in a vend getting reported to Elecctro’s servers.

Both flows in the beginning try to start a session with a vending machine upon customer interaction. A session is what defines the start and end of a flow. A vending machine can only have one open session at any point in time, and it is used so other customers do not try to purchase any items while a customer is already in the process of doing so. This way Elecctro can link a vend to a single customer.

The first flow, shown in Figure 2.3, requires the customer to first interact with a payment terminal that is embedded in the vending machine, and connected to the vending hub as a peripheral. The customer then picks the payment gateway he wants to pay with, selects the product he desires and then finishes the payment.

The second flow, shown in Figure 2.4, requires the customer to first start a dialog with Elecctro’s Facebook Messenger chat bot, Spencer, by scanning a special Facebook code that is on the front of the vending machine. Spencer greets the customer, and prompts him to pick the payment gateway he wants to pay with, then to select the product, and finally to finish the payment.
Figure 2.3: Customer interacting with a vending machine via payment terminal.
Figure 2.4: Customer interacting with a vending machine via Electro's Facebook Messenger chat bot.

1. Customer scans Facebook code opening a chat dialogue.
2. Asking which payment gateway to use.
3. Selects payment gateway.
4. Asking to pick a product on the chat.
5. Customer picks a product.
6. Reports required credit.
7. Requests credit.
8. Reports credit approval.
9. Releases the product.

Waits for customer to decide.

Servers
Vending Hub
Vending Machine
Gateway Instillation
2.4 Blockchain

In this section we provide a definition of a blockchain and how blockchain systems can be divided by two main properties. Then, we analyse the main blockchain systems as well as some work performed in the vending and micro-payments area, while discussing their advantages and disadvantages, finishing with a final summary.

2.4.1 Blockchain definition

A blockchain, is a list of blocks chained together. These blocks are connected, by references, just like a linked list, but they are linked with the previous block, instead of the next block. This means that given any block there is only one path to the first, original block, often called the genesis block.

Before diving onto what this reference is, and what properties it assures, lets first define a block in the context of a blockchain. A block is a data structured composed by its index in the chain, a nonce, some arbitrary data and a reference to the previous block. A block reference is its hash. A hash of a block, is a cryptographic hash of all contents of the block, which means this hash is calculated using the block’s index, the block’s nonce, the block’s data and the block’s previous block reference (the hash of the previous block). An instance of a blockchain with 3 blocks can be seen in Figure 2.5.

A hash is a "fingerprint" of some given data, this means that any modification to this data would alter this "fingerprint". Since blocks have in their hash, the previous block hash, any alterations in one of the past blocks, would cause a cascade change to the hash of every consequent block starting by the one that was originally modified. This means that once appended to the blockchain, a block cannot be altered unless we calculate the hash of every single block that comes after the modified block.

Of course if a blockchain were not to be distributed, one could easily modify the data within a block and re-compute the hash of every consequent block from that block on. However, if blockchains are usually distributed which makes it not so trivial. In a distributed blockchain
system, every peer keeps a copy (or some snapshot of the state) of the blockchain, and whenever a new block gets added to it, a mechanism ensures consensus in what block gets added, so every peer can end up with the same blockchain. Because there exists this consensus mechanism, in order for someone to alter the data of any block, not only would they have to re-compute the hash of every consequent block, but they would also need every peer to perform the same re-computation and agree on the altered version of the blockchain. Because blockchain systems are designed to keep this scenario very unlikely, blockchains are deemed immutable.

Not reaching consensus could result in an event called forks, which happens when two different blocks reference the same previous block among different peers. These are usually temporary, and eventually all peers will agree on the same chain of blocks.

Blockchain is best utilized for applications that require multiple peers cooperating together to keep track of some data, but no single peer is able to alter the data without having all peers agree on the modification. This allows trust to be created without having to trust in a single entity or a third party. Like Vitalik Buterin, creator of Ethereum [7], one of the most popular blockchain systems, once said “if you explicitly make it harder or impossible for yourself to do certain things, then others will be more likely to trust you and engage in interactions with you as they are confident that those things are less likely to happen to them” [8].

The first blockchain, Bitcoin [9], had the data field of a block assume the form of a collection of financial transactions that were bundled together, and were then appended to the blockchain after successfully validated. This validation may change depending on the data that the block holds, but in the case of financial transactions, this validation checks if the funds being transferred belong to the sender and if the sender has the amount of funds necessary to perform the transaction. After validated, the block is appended to the blockchain, and peers start working on the next block.

Blockchain systems can be divided by two main groups:

- Public vs private blockchains, which delimits who can participate in the blockchain (authentication)

- Permissioned vs permissionless blockchains, which delimits roles peers have in the blockchain (authorization)

Both properties will be further explained in Section 2.4.2 and Section 2.4.3 respectively.

### 2.4.2 Public vs Private Blockchain

A blockchain can be public or private, and this is often confused with the permissioned and permissionless definition that we will be discussing in Section 2.4.3.
Public blockchains are blockchains where anyone can join the network, and there may or may not be restrictions on the role they can perform within the network. They can leave and join the network whenever they want, without affecting the overall performance or availability of the network.

Because the network is public and anyone can join, everyone can see what is the current state of the blockchain, what blocks are currently awaiting confirmation, data within the blocks, among others. This can be considered a major feature combined with the immutability a blockchain provides, or a major setback when you want to keep data private.

An example of a public blockchain would be Bitcoin [9], where anyone can read, approve or issue transactions.

Private blockchains on the other hand, are blockchains that require an invite to be able to participate in them. Once invited, one can be a part of the blockchain just like they would in a public one. The main advantage of private blockchains is that now, since you are able to invite participants, there is a set of rules that defines who can participate in the network. That way, depending on the amount of trust the network has over its participants and the amount of participants in the network, we can leverage the detail of information that is available on the blockchain.

An example of a private blockchain would be an custom Ethereum network, behind a virtual private network (VPN), where only those with the VPN credentials are able to join and participate in the network. Once joined, they can perform any role they would like, just like on the Ethereum public network.

2.4.3 Permissioned vs Permissionless Blockchain

Permissionless blockchains are blockchains where no participant needs permission to perform any action. In other words anyone can perform any role within the blockchain. This is why permissionless and public blockchains are often confused and used interchangeably. Just because a blockchain is public, it does not mean everyone has permission to participate in certain mechanisms like consensus. An example of a private permissionless blockchain would be a private Ethereum network, like discussed in Section 2.4.2, where once joined anyone can perform any role, with no restrictions.

Because there are no restrictions to which roles participants can perform, anyone can contribute to it, malicious participants included. Therefore the network is susceptible to possible attacks, like double spend attacks - where an attacker tries to spend the same digital token more than once - or 51% network takeover attacks - where an attacker controls more than half the
network, giving him the ability to change the blockchain’s data. This means these blockchains need mechanisms like proof-of-work [10] to be implemented, to lower the probability of these sort of attacks succeeding [9]. Proof-of-work is explained in more detail in Section 2.4.4. This usually means that the time taken for a block to be confirmed (appended to the blockchain of every peer) is in the order of minutes if not hours. Because mechanisms like proof-of-work are needed, there needs to exist incentives for participants who help achieve consensus, given they are lending their computational resources. These participants are often called miners.

An example of a public permissionless blockchain would be Bitcoin [9], where anyone can be a miner or a user, approve or issue transactions, and be rewarded with bitcoins when successfully helps the network achieve consensus. We further discuss Bitcoin in Section 2.4.4.

On the contrary, permissioned blockchains assign roles and responsibilities, like working towards consensus, to certain participants who satisfy a set of rules. This set of rules might require participants to be identifiable so one could come up with restrictions like ”only A and B are allowed to contribute to the consensus algorithm”. This might be confused with private blockchains because one might think that identifying participants is required and that inviting is a way of identifying participants.

Because only some participants may perform certain roles, one of these roles is usually consensus. Because these need to comply to a set of rules, and are usually identified, mechanisms like mining might not be necessary since interactions are usually performed among participants that share a common goal. This translates into a significant gain in performance, which means the time taken to achieve consensus is orders of magnitude lower than the permissionless variant, resulting in faster confirmation of blocks, and more blocks per second.

2.4.4 Bitcoin

Bitcoin [9] was the first blockchain. Developed by Satoshi Nakamoto in 2008 it was announced as a peer-to-peer electronic cash system that solved double spending without requiring a trusted third party.

In the context of Bitcoin, such attack could be achieved by a malicious peer that tries to maintain two different blockchains, the original one where the attacker would append a block with a transaction $T$ where he transfers bitcoins to a merchant, and a malicious one with a transaction $T'$ where he transfers the same amount of bitcoins to himself. Once the merchant confirms $T$, and releases the goods he got paid for, the attacker can then release the malicious blockchain, hoping the remaining peers use it. If the network agrees to use the malicious blockchain, where $T'$ is present, then the attacker will get both the goods and the
bitcoins he originally sent to himself. The merchant loses the goods and does not get paid.

Bitcoin was the first blockchain to solve this by using something called proof-of-work [10] as a consensus mechanism by having peers pick always the blockchain that has more computing power put into it. Proof-of-work was originally conceived by C. Dwork and M. Nao [10] to help prevent spam in emails by having senders solve a computational difficult problem in order to send an email to a given recipient. Bitcoin used a similar idea by having its peers solve a difficult problem before being able to append a block to the blockchain. This difficult problem comes down to having the hash of a block begin with an arbitrary number of zero bits, by changing the nonce of the block and re-hashing it until this restriction is met. According to Bitcoin original paper [9] the average work required is exponential to the number of zero bits required, but can be verified by executing a single hash.

Having proof-of-work increases the difficulty of re-hashing a block, let alone a whole chain. So if an attacker wants to alter the content of a block, not only would he need to re-calculate the hash of the block, he also needs to surpass the network since peers always use the longest chain that has the greatest proof-of-work effort invested in it. Therefore, an attacker would need to outperform the network. Citing Bitcoin original paper [9] if a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains.

To incentive a network of honest nodes, Bitcoin awards bitcoins to those who successfully solve the proof-of-work of a block. This process is called mining.

Because Bitcoin has proof-of-work, that means there are multiple miners spread throughout the network solving computationally intensive problems in order to append new blocks to the blockchain. This translates into energy efficiency problems and scalability issues when processing new transactions. Not only does it represent a lot of energy consumption put into solving this proof-of-work, it also means that once a new transaction is issued one has to wait quite some time for it to be confirmed i.e, be acknowledged by the majority of the network. According to K. Croman et al. [11], in 2016, Bitcoin’s maximum throughput was 3.3 to 7 transactions/sec. They also mention a latency of roughly 10 minutes for a transaction to be considered confirmed. This is unfeasible for our problem, where instant transaction confirmations are required, as well as a high throughput.

In the next Section we discuss a solution designed on top of the Bitcoin network to tackle these scalability issues.
2.4.5 Bitcoin Lightning Network

The Bitcoin Lightning Network [12] is an off-chain solution developed to tackle the scalability issues of Bitcoin. It allows the creation of a channel between two parties who wish to perform an exchange of funds over time without having every single transaction performed between them on the Bitcoin blockchain. Having a dedicated off-chain channel allows transactions to be instantaneous between participants, as they do not have to wait for any network confirmation after every transaction.

Opening a channel takes the form of creating a 2-of-2 multisignature [13] wallet that requires to be pre-funded by both parties with any amount of their choice. These amounts are then saved locally by both parties in a "spreadsheet" that contains which funds belong to whom, and that keeps track of all transactions between them for the lifetime of the channel.

After open, both participants may transfer funds to each other, always updating locally the balance of each in the spreadsheet. These updates are done cryptographically to ensure both parties agree on the new updated balances. They can perform as many transactions as they want, as long as they have enough balance to perform them.

This can go on until one of the participants decides to close the channel, claiming the funds on the multisignature wallet that belong to him. A participant is only able to collect the balance that belonged to him in the latest update of the spreadsheet. This claim results in an on-chain transaction with the respective amount of each participant transferred to them.

So in the end, what the Bitcoin Lightning Network provides is a way to defer the state of an arbitrary number of transactions between two participants, reflecting only in two transactions on the Bitcoin network.

To prevent the creation of a channel with everyone, the Bitcoin Lightning Network provides a network of channels. So in a scenario where A wants to trade with C, and both A and C already have a channel open with an intermediary B, the Bitcoin Lightning Network can resolve a path from A to C through B, as long as this path has the necessary funds to perform the transaction. In any path, nodes are motivated to cooperate as they are awarded with a fee for each transaction they perform. This fee increases with the amount of hops required to perform a transaction.

In case a payment does not fully route to its destination, or if one of the participants along the path is uncooperative, the sender must wait until the transaction expires before getting their funds back, resulting in having the funds temporarily locked.

This solution is only suited for small payments, since the network might not be able to resolve a route that has enough allocated funds to perform a payment.
Although this looks promising, the requirement of opening a channel and of paying fees might not compensate. Also since nodes may be non-cooperative, the sender might have their funds temporarily locked, needing to wait for the transaction to expire before re-sending them via an alternative route.

2.4.6 Ethereum

Ethereum [7] is similar to Bitcoin in many aspects: they are both public permissionless blockchains; they both have a mining mechanism that encourages peers to stay honest, helping the network achieve consensus; and they both have their own currency, ether and bitcoin respectively.

The main difference between Ethereum and Bitcoin is smart contracts. Although Bitcoin has scripting capabilities [14] that allows for more complex behaviours when performing transactions (like multisignature wallets [13]), it does not provide the developers the ability to use loops nor long term storage. This limitation allows peers to verify transactions knowing these verifications are finite computations.

Ethereum smart contracts are less restrictive, providing a Turing-complete programming language, for the Ethereum Virtual Machine (EVM), that allows the usage of loops and the usage of a long term key/value storage that enables them to store data between executions. Both code and storage of a smart contract reside on the blockchain, publicly available for everyone to read, query and interact with.

Just like Bitcoin, Ethereum has an account per wallet, where this account has an address and a balance. Like Bitcoin, the ownership of an account is verified using cryptographic public-private key pairs. Ethereum calls these externally owned accounts.

Ethereum has another type of account, named contract accounts, which are solely controlled by smart contracts. These accounts have within them, the code of the contract as well as their balance. Contract code is executed when the account receives a transaction. This means smart contracts can call other smart contracts.

Smart contracts enable a whole realm of decentralized applications such as: the creation of more complex multisignature wallets, that have multiple conditions to allow the withdrawal of all or part of the funds; the creation of a crypto-currency on top of a contract, in the form of tokens, that uses the long term storage to keep track of balances; the creation of gambling smart contracts that makes sure the house does not cheat because the contract’s code is publicly available for everyone to consult.

To prevent contracts with infinite loops trying to exhaust the executor’s computational resources, Ethereum limits how much computational time (gas unit in Ethereum) can be spent.
in executing a transaction. So transaction senders need to explicitly tell beforehand how much
gas do they want to allocate to the execution of a transaction. A transaction that runs out of
gas while executing is aborted, but the miner who devoted resources to execute it still gets the
fees for trying.

Much like Bitcoin, Ethereum has the same scalability issues, where every transaction needs
to be processed by every node in the network. This is of great concern because while Bitcoin
was only built as a crypto-currency and therefore only needs to verify financial transactions,
Ethereum is able to host a limitless number of decentralized applications, thus having more
time and space overhead, since nodes need to execute arbitrary code.

Bitcoin has a fixed maximum block size, where Ethereum does not. Ethereum has a varying
gas limit per block and therefore it is able to scale when it comes to throughput. As of today,
assuming a gas limit of 8,003,940, an average block time of 13.93 seconds [15] and a transaction
allocation of 21,000 gas, we have a throughput of 27.36 transactions per second. This is way
bigger than Bitcoin’s throughput of 7 transactions per second. Note however that this number
varies since transactions may allocate more gas, decreasing the number of transactions that can
fit in a block.

2.4.7 ERC-20 Token

The ERC-20 [16] is a smart contract standard for the Ethereum blockchain. It aims to uniformize
the token specification so applications that interact with tokens do not have to continuously
develop extensions to their systems to support non-uniform tokens as they are released.

A token is a possible application of an Ethereum smart contract. Token systems have many
applications, for example: assets such as company stocks; tokens representing tangible assets
such as cars; tokens representing points in a loyalty system.

The ERC-20 specification requires the implementation of 6 methods, namely:

- \texttt{totalSupply()}, which returns the total token supply of the contract
- \texttt{balanceOf(address \_owner)}, which returns the amount of tokens owned by a given ad-
dress
- \texttt{transfer(address \_to, uint256 \_value)}, which transfers to a given address the amount
  of given tokens
- \texttt{transferFrom(address \_from, address \_to, uint256 \_value)}, which transfers from
  a given address, to a given address a given amount of tokens. This is used for a withdraw
workflow, allowing contracts to transfer tokens on your behalf if you allow them to, using the `approve` method.

- `approve(address _spender, uint256 _value)`, which approves the withdrawal by a given address of a given amount of tokens

- `allowance(address _owner, address _spender)`, which returns the amount tokens that are approved by a given address to a given spender

Apart from these required methods, contracts are free to implement more functionality as they want. Because these tokens are stored within contracts in the Ethereum network, running the code in these contracts corresponds to issuing transactions to the contract with additional data that provides which function to call as well as its arguments. Therefore ERC-20 Tokens share the same performance and scalability issues as Ethereum.

2.4.8 Raiden Network

The Raiden Network [17] is the Ethereum counterpart of the Bitcoin Lightning Network previously discussed in Section 2.4.5. It supports both ether and ERC-20 compliant tokens, allowing instantaneous off-chain transactions. Instead of having a 2-of-2 multisignature wallet, both parties have a 2-of-2 multisignature contract that gets pre-funded with any amount of their choice. After that, both parties perform off-chain transactions until one of the parties decides to leave and claim the funds that belongs to them. Claiming the funds results in another on-chain transaction.

Just like the Bitcoin Lightning Network, the Raiden Network also deploys a network, so one party does not require a direct communicating channel with the other, and can use their network to perform routing of transactions.

Since it functions the same way as the Bitcoin Lightning Network, it also has the same issues. It requires both parties to have their funds locked up in a contract for the lifespan of the channel; depending on the amount of hops in the network fees may increase; and is only fit for small payments, since throughout the network, channels may not have enough funds pre-allocated to them.

2.4.9 Have a snack, pay with Bitcoins

Bamert et al. [18] explore the possibility of speeding up the confirmation time of a transaction in the Bitcoin network (they name it fast transactions) while applying it to the micro-transactions scenario, namely purchases in vending machines.
They propose to have an arbitrary number of nodes randomly distributed among the Bitcoin network, not actively participating, but rather just listening for all transactions that are currently being issued and broadcast in the network. If a transaction is of their interest, all they have to do is wait for more broadcasts from the network to increase the confidence of a transaction being non-malicious. Upon reaching a certain predefined threshold of seen transactions, within a predefined time interval, they notify the merchant to deliver the product.

Naturally the bigger the amount of connected nodes and the bigger is this threshold and time interval, then the smaller is the probability of a successful double spend attack.

Nodes have to be randomly distributed so they cannot be targeted by an attacker that could issue an invalid transaction \( T' \) to those specific nodes (where the attacker sends bitcoins to the merchant), while issuing a valid transaction \( T \) to other nodes in the network (where the attacker sends bitcoins to himself). Doing so would result in \( T' \) not being accepted by the network, and the merchant being a target of a double spend attack.

While listening to 100 nodes, using a threshold of 37 announced transactions with a time interval of 6.29 seconds, the authors estimate a 0.088\% chance of an attacker succeeding in a double spend attack.

Although 7 seconds does seem ideal for a transaction to be confirmed, the requirement of having to listen to 100 different nodes within the network, with the possibility of connecting to nodes that are also just listening and not announcing transactions, is costly. Not to mention that this would only allow for fast transactions from customer to merchant and not the other way around, unless customers are willing to come up with their own nodes to confirm transactions using the same method, which is highly unlikely.

2.4.10 Hyperledger Fabric

Hyperledger Fabric [1] is an enterprise-grade permissioned blockchain framework that focuses on modularity which allows developers to craft a blockchain solution that fits their requirements. Since it is a framework, there is no cryptocurrency like the other variants we discussed previously.

Because it is a permissioned blockchain framework, Hyperledger Fabric enforces every participant to identify themselves, but since it is very modular there can be custom membership services that identify participants however they desire.

At its core is the notion of chaincode, which is another name for smart contracts. Chaincode can be written in a variety of languages and it is how participants interact with the blockchain (via transactions). Hyperledger Fabric allows for custom endorsement policies when validating
transactions for a specific chaincode, that dictate which peers need to endorse a given transaction for it to be considered by the network.

Participants can join multiple channels within the same network, that way developers can mediate which participants are allowed to see and interact with what information. It also allows for transactions to be private among different participants if required.

Hyperledger Fabric deploys a different architecture in comparison to other blockchain systems which unlocks better concurrency, and is therefore more performant than others. Where as other blockchains follow the order-execute architecture, Hyperledger Fabric came up with a different approach named the execute-order-validate architecture.

For example in Bitcoin or Ethereum, transactions are first validated and put into blocks. In the case of Ethereum, if a transaction is for a contract, the contract code is executed. If everything is successfully validated, then the miner performs a proof-of-work algorithm. Once successful, the peer broadcasts the block into the network, having every peer re-execute the contents of the winning block. If the block is indeed the winning block then every peer has to apply its transactions to their blockchain state. So essentially a order is first proposed, and then if everything seems correct, every peer executes and applies the state changes to their version of the blockchain.

In the execute-order-validate architecture, nodes in Fabric need to have one (or more) of three possible roles:

- Clients, who submit transaction proposals for the execution phase
- Peers, who may execute and/or validate transactions, maintaining a copy of the blockchain
- Orderers, who simply order transactions depending on the consensus algorithm

So with the execute-order-validate architecture, transactions are first sent to the peers specified in the endorsement policy and are executed by them. After executed, they are sent to the orderers, which decide the order of transactions among all peers. Orderers then broadcast the transactions to every peer. All peers then validate and apply the transactions in the same order.

With this architecture Fabric is capable of achieving, in popular deployment configurations, more than 3,500 transactions per second, with sub-second latency scaling well to over 100 peers [1].

2.4.11 Summary

In this section, we will compare each of the solutions discussed above. In Table 2.1 we compare each solution across 4 attributes:
Smart contracts support, since we could use them as a way to store reward rules by each merchant, as well as which vending machines are participating in the loyalty system;

Consensus mechanism, as it influences the amount of work each node has to perform

Transactions per second (throughput), since it dictates the amount of requests per second we will be able to serve

Confirmation time (finality), since it dictates the amount of time a request takes

Both Ethereum and Bitcoin are not candidates. They both have very low throughput and a really big finality, in the order of minutes, something unbearable for a customer when using points. But if they we were to extend them using the Bitcoin Lightning Network or the Raiden Network respectively, then they are definitely worth considering.

Between using the Bitcoin Lighting Network and the Raiden Network, the lack of smart contracts in the former makes it very difficult to store data in the blockchain, therefore we would use the Raiden Network and its interoperability with ERC20 tokens to host our loyalty system, which would allow for more complex interactions as well as an interface for merchants to easily change their reward rules and participating vending machines while having a instantaneous system.

Although instantaneous, there is a major setback to this approach, where customers would have to open a channel beforehand. Since opening a channel corresponds to one on-chain transaction, it means that a customer would have to open a channel before using a vending machine. So we would either have to somehow make that automatic with the user’s permission, or have the user explicitly tell beforehand that he plans to use a specific vending machine. Another way to tackle this is to create some nodes that would function as intermediaries for the Raiden Network, which would connect customers with any merchant, as long as customers would have an open channel with these intermediaries.

The idea of fast transactions discussed in in Section 2.4.9 is neat, but does not align with our goals since customers would have to spread some nodes among the network to listen to transactions from merchants, for the instant point awards. Not only that, but we would also have to adapt the idea to work on top of a blockchain that supports smart-contracts, like Ethereum.

A more interesting alternative is Hyperledger Fabric. A permissioned approach makes sense in this context since we only want some participants to validate transactions. For instance, we could have merchants as endorsing peers/validators and customers as clients. Also, the evaluation performed by Fabric, shows that the system has sub-second finality and good throughput.
scaling well to over 100 peers in popular deployment configurations. This conforms with our objectives of instant rewards, as well as high throughput and many participating merchants.

Each of these alternatives present some advantages and disadvantages, but in the end we decided to use Hyperledger Fabric as the backbone of our implementation, which we discuss in the next chapter.

<table>
<thead>
<tr>
<th></th>
<th>Smart contracts</th>
<th>Consensus mechanism</th>
<th>Transactions per second</th>
<th>Finality (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lightning Network [12]</td>
<td>No</td>
<td>Between channel using cryptography</td>
<td>No limit</td>
<td>Sub-second</td>
</tr>
<tr>
<td>Ethereum [7]</td>
<td>Yes</td>
<td>Proof-of-work</td>
<td>Around 27</td>
<td>0.5-5</td>
</tr>
<tr>
<td>Raiden Network [17]</td>
<td>Yes</td>
<td>Between channel using cryptography</td>
<td>No limit</td>
<td>Sub-second</td>
</tr>
<tr>
<td>Fast transactions [18]</td>
<td>No</td>
<td>-</td>
<td>-</td>
<td>0.1</td>
</tr>
<tr>
<td>Hyperledger Fabric [1]</td>
<td>Yes</td>
<td>Ordering service with pluggable consensus</td>
<td>3500</td>
<td>Sub-second</td>
</tr>
</tbody>
</table>

1 Average taken from last 7 days in Blockchain’s statistics website [19]
2 Upon opening a channel
3 According to ETH gas station [20]
4 Listening to 100 peers and recording 37 transactions
5 Using a popular deployment configuration, over 100 peers

Table 2.1: Comparison between studied solutions
Chapter 3

Elecctro Points

In this chapter we discuss our the design and implementation of Electro Points, blockchain based loyalty system on top of Hyperledger Fabric. We start with a high-level overview of our architecture. Then, we go through each component in more detail, their technologies and design choices. In the end of the chapter, we explain how each component interacts with each other across different scenarios throughout the lifetime of the network.

3.1 Proposed architecture

Figure 3.1 depicts the proposed architecture of Elecctro Points showing one customer, two merchants and Elecctro, all participating in a Hyperledger Fabric network, each with different roles.

Merchants take up the role of peers, therefore they require one node that participates in the network. In their node merchants will have three chaincodes installed as well as a component named Merlin. Both components are discussed in more detail in Section 3.4 and Section 3.5 respectively.

Customers take up the role of clients, they are able to query and submit transactions to the network through the usage of a mobile application, named ElePoints.

Elecctro will take the role of orderer responsible for ordering transactions. Since every participant needs an identity, Elecctro is also responsible for managing the customer’s membership service, in the form of a certification authority (CA) that issues certificates for each customer that wants to register and join the network.

We go through each component in more detail in the following sections.
Figure 3.1: Electro Points’s architecture
3.2 Orderer

As discussed in Section 2.4.10, the orderer is responsible for ordering transactions. There are a few implementations for an orderer service [21], namely: Solo, Raft and Kafka.

A solo ordering service, means that there is only one orderer instance across the network. Since there is only one orderer, this orderer is seen as a single point of failure for the whole network. If the orderer for some reason crashes, then transaction proposals will fail to be delivered, and therefore the network stops accepting new transactions.

With both Raft or Kafka ordering services, one can deploy multiple instances at the same time, to ensure fault tolerance. Both these implementations use a leader and follower approach, where there is a leader that orders transactions and its decisions are replicated by the followers.

Both Raft and Kafka are expected to be less performant as they must make sure that their followers are synchronized in case the leader crashes.

We decided to use the solo implementation since it unlocks more performance at the cost of a non-fault tolerant service. We think that our application does not have high availability requirements, in the sense that if the loyalty service is not working for a couple of minutes, it shouldn’t pose much of a problem. It is always possible to change this later to a Raft or Kafka service if availability becomes an important aspect of our system.

As a solo orderer, we picked Electro to perform this role so that merchants do not have to decide who will be responsible for it. Although an orderer, in this case Electro, can read and block transactions, these can be encrypted so that Electro can not be selective about which transactions to block. This ensures that Electro is not compelled by some merchants to "cheat" and block other merchant’s transactions.

This can change in the future with the usage of Raft or Kafka, where we can have a pool of merchants as orderers, to prevent centralization and to allow for a fault tolerant service.

3.3 Customer CA

Because Hyperledger Fabric is a permissioned blockchain framework, in order to assign roles to every participant, it requires them to have an identity. As mentioned in Section 2.4.10, Hyperledger Fabric allows the usage of custom membership services that are responsible to manage identities within an organization. An organization in Hyperledger Fabric is a group of participants with a particular purpose/role. In our implementation, we have the following organizations: one for all customers, one for Electro, and one for each merchant.

Since we have an organization for customers, we require a membership service that handles...
the identities of all customers. Because customers are not a fixed set of participants, we require a membership service that is able to register new members on demand.

We decided to use Hyperledger Fabric CA (Certification Authority) [22] as a membership service to manage the identity of all the customers, since it has the ability to register/deregister new members like we require. The Hyperledger Fabric CA provides an API that allows customers to register using a username and a secret. After registered, a customer can enroll in the CA, using his username and secret, to get a signed certificate that will be used as its identity throughout the network. This identity will also be used as a mean of authentication in some chaincodes, for example to make sure that a customer does not use other customer’s points.

Since it serves as a mean of authentication, customers should not share this certificate with anyone. If somehow a certificate gets compromised, a customer can revoke the certificate, and enroll again to get another one. If a customer’s secret gets compromised, the customer can also revoke its identity in the CA, meaning it will never be able to enroll again with the same identity.

We decided that Electro was the most appropriate entity to be responsible for this CA. This decision was driven by the fact that Electro already stores customer data, like payment information for cashless payments. We can use this information to link a customer’s identity in the network, to their cashless payment information. This is crucial to maintain the customer’s payment information private, while making sure that customers get awarded with points when using cashless payments. This will come later into play when we discuss how points are awarded.

### 3.4 Chaincode

In Hyperledger Fabric’s core we have chaincode. Chaincode is like smart contracts, code that can be installed in all endorsing peers that can be executed on demand by clients. Like mentioned in Section 2.4.10 a chaincode has a custom endorsement policy, that dictates which peers need to endorse a transaction in order for it to be considered valid by the network.

If a client wants to execute a chaincode function, it needs to contact the peers in the endorsement policy, specifying which function of the chaincode to execute. The output of the function needs to be the same in all endorsing peers which means that chaincode functions must be deterministic and can only use the input provided by the client.

Peers specified in the endorsement policy are required to have the chaincode installed in their system so they can endorse transactions for that chaincode. Once installed, the only way a chaincode can change is via upgrade, which requires every endorsement peer to install the new updated version of the chaincode, in order for it to be used by the network. Chaincode
can also interact with other chaincodes while executing, meaning we can perform more complex interactions and abstractions that only require a subset of chaincodes to change when updating the overall functionality of the network.

At the core of our implementation, we have three chaincodes:

- **machines** chaincode, responsible to store which machines are participating in the loyalty system
- **rates** chaincode, responsible to store the point awards and discounts of every merchant per product
- **elepoints** chaincode, responsible to store the points of each customer, as well as all interactions with it

We discuss in detail each of these chaincodes in the following subsections.

### 3.4.1 machines chaincode

The **machines** chaincode is responsible to store information about which machines every merchant has participating in the loyalty system. The endorsement policy, is to only require one merchant to endorse a transaction in order for it to be considered by the network. This allows merchants to change the machines that are participating in the loyalty system at any point in time without impediments.

Although we only require one merchant to endorse a transaction, all functions in the chaincode are authenticated, to ensure that merchants do not modify other merchant’s vending machines. All functions are also authorized, to ensure customers are only allowed to query participating vending machines, whereas merchants are allowed to both query and change this information.

The chaincode implements the following functions:

- **queryAllMachines(merchant:string)** which given a merchant, returns all vending machines that the merchant has participating in the loyalty system;
- **queryMachineOwner(machine:string)** which given a participating vending machine, returns the merchant to which that vending machine belongs to;
- **addMachine(machine:string)** which adds a given vending machine to the participating vending machine list of the merchant who called the function;
- **removeMachine(machine:string)** which removes a given vending machine from the participating vending machine list of the merchant who called the function;
• clearMachines() which removes all vending machines from the participating vending machine list of the merchant who called the function.

3.4.2 rates chaincode

The rates chaincode is responsible to store information about point awards and discounts for every merchant. The endorsement policy, is to require the majority of the merchants to endorse a transaction in order for it to be considered by the network. This allows merchants to agree or disagree if a change in point awards or discounts is similar to the already existing ones.

Similar to the machines chaincode, all functions are authenticated to ensure that merchants do not modify other merchant’s awards and discounts. All functions are also authorized, to ensure customers are only allowed to query awards and discounts, whereas merchants are allowed to both query and change this information.

Award and discounts information is stored in the form of rates. Rates are composed of the following elements:

• An award formula, that given the price of a product, it calculates the amount of points a customer should be awarded;

• A discount formula, that given the the amount of days until a product’s expiration date, it returns the discount percentage as well as how many points are required to purchase the discount;

• An enabled flag, which can be used to toggle the rate at any point in time.

Merchants have a generic rate that applies to every product. This generic rate can be overridden with product specific rates for more fine-grained control. For example, this allows to blacklist a few products or have "happy" hours where some products award more points.

The chaincode implements the following functions:

• queryDiscounts(merchant:string, product:ProductDiscountQuery[]) which given a merchant and a list of products with the days until their expiration dates, returns the percentage of discounts of each product alongside the cost of the discount in points;

• queryAward(merchant:string, product:ProductAwardQuery) which given a merchant and a product with its price, returns the amount of points a customer is awarded with;

• queryRates(merchant:string, product:string[]) which given a merchant and a list of products, returns the rates of each product.
• **addRates(productRates:ProductRate[])** which given a list of product specific rates, adds them to the list of rates of the merchant who called the function;

• **removeRates(products:string[])** which given a list of products, removes them from the list of rates of the merchant who called the function;

• **editRates(productRates:ProductRate[])** which given a new list of product specific rates, updates the old product specific rates of the merchant who called the function;

• **editDefaultRate(productRate:ProductRate)** which given a new rate, replaces the old default rate of the merchant who called the function;

### 3.4.3 elepoints chaincode

The **elepoints** chaincode is responsible to store the points of every customer as well as to mediate point transfers between customers and merchants. The endorsement policy, is to require the majority of the merchants to endorse a transaction in order for it to be considered by the network.

Similar to the **machines** chaincode, all functions are authenticated to ensure that customers cannot interfere with other customer’s points. All functions also have authorization checks, to make sure customers are only allowed to transfer points, whereas merchants are only allowed to award them.

At the core of the chaincode is a map that stores the points of every customer. This map is required to keep track of each customer’s balance.

The chaincode implements the following functions:

• **queryCustomer(customer:string)** which given a customer, returns the amount of points and ongoing purchases of a customer. Customers can only query their accounts whereas merchants can query any customer;

• **awardPoints(customer:string, points:integer)** which credits the given customer with the given amount of points. This can only be called by merchants;

• **transferPoints(customer:string, points:integer)** which transfers the given amount of points to the given customer. The points are transferred from the customer who called the function. This can only be called by customers.

• **purchaseDiscount(vendingMachine:string, product:string, points:integer)** which purchases a discount for the given product removing the given amount of points. The
points are removed from the customer who called the function. This can only be called by customers.

- `refundCustomer(customer:string, purchaseId:string, refundAmount:integer)` which refunds a given amount to a given customer. This function can only be called by merchants.

Additionally, `purchaseDiscount` creates a unique purchase identifier that can be later used by merchants to refer to that discount purchase, for example to refund a customer. Refunds can happen for three reasons:

- Customers may not send enough amount of points when purchasing a discount. In this case the merchant refunds the whole amount.

- If a discount gets applied to a vend that failed (e.g. product was not delivered). In this case the merchant refunds the whole amount.

- If the customer sent more points than required when purchasing a discount. In this case the merchant refunds the unused amount.

### 3.5 Merlin

Merlin (stands for MeRchant LIsteNer) is a component in every merchant’s node. This component is responsible to listen to two types of events, each with its own listener.

The first listener, listens to the network, more specifically for incoming discount purchases. When a customer purchases a new discount, every merchant node eventually receives an update on their blockchain. A peer can choose to be notified as soon a specific blockchain update happens locally. So every time a customer purchases a discount, Merlin gets notified and can check if the purchase is meant for him. If it is, Merlin can then ask Electro to apply a discount to the given product in the given vending machine.

The second listener, listens to Electro’s servers, more specifically for cashless vends events. Merlin uses this listener to award customers when performing a cashless purchase. Electro also includes in the event whether the vend was the result of a requested discount or not. That way Merlin instances can refund the customer if required.

One problem with this approach is that if a Merlin instance crashes at the moment of a discount purchase, the customer loses both the discount and the points. To prevent that, Merlin instances have a checkpoint mechanism to record the last handled discount purchase. This way, if they ever crash, they can just look at the state of the blockchain, check if there were any
discount purchases meant for them, and refund the customers (and maybe even compensate them for the frustration).

Our initial idea was to have the vending hub award points to the customer on the behalf of the merchant. But in order for that to happen, Elecctro required an identity that belonged to the merchant’s organization. This means that Elecctro would get access to a set of credentials that could use to impersonate a merchant, allowing Elecctro to perform actions that only merchants are allowed to perform, like awarding points. Not only that, but Elecctro would also have to be responsible to listen to all discount purchases on-going on the network.

This way we distribute the load across merchants without leaking credentials. Merchants are then responsible of awarding and using customer points. Since all participants have an identity, we can mediate who used or awarded points to a customer, and use this information to cross-check whether a merchant may be dishonest, or ignoring discount purchases.

This approach has a drawback. Merlin is seen as a single point of failure. If a customer purchases a discount in a vending machine that belongs to a merchant whose Merlin instance is down, then the customer will lose their points until the merchant refunds them. This could be solved by spawning more Merlin instances per merchant, but further analysis is required to prevent race conditions where more than one Merlin instance serves the same discount purchase.

## 3.6 ElePoints

ElePoints (stands for Elecctro Points) is a mobile application used by customers to interact with the network. A customer can use the mobile application to check its balance, transfer points to customers, purchase discounts with merchants and to link their "account" to their cashless payment information as well as Facebook Messenger identifier.

Initially, customers are introduced with a login screen where they login and register themselves in the loyalty system. This contacts the Customer CA to either enroll, or register a customer given its username and secret. After enrolling, ElePoints stores the enrolling certificate in the phone’s storage and uses it as an identity when interacting with the network.

After login, customers are presented with their balance and the ability to transfer points to other customers. They can also link their Facebook Messenger identifier, as well as cashless payment information, to make sure they get awarded with points when using those payment methods. All this linkage information is kept secure and private in Elecctro’s servers as described in Section 3.3.

Purchase discounts are triggered when communicating with Elecctro’s Facebook Messenger chatbot. This means that discount information is only available when purchasing via Facebook
Messenger, and can only be consulted once in front of a vending machine.

In order for the application to interact with the network, it needs to use one of the Hyperledger Fabric SDKs [23]. These SDKs are implemented in two languages: Javascript, via Node.JS [24]; and Java [25]. After looking into each, neither implementation officially supports the mobile environment.

Although there is a Java version, its implementation relies too much on API that is not available in the Android ecosystem, and therefore we would have to adapt the SDK to remove this restriction, by making its implementation platform independent. Even if we managed to do so, this solution would only work for the Android platform, not allowing iOS users to participate in the loyalty system. This also means that every time a major update to the SDK comes through, we had to port everything all over again. This process is very time consuming, even more in a SDK that is still far from mature.

We took a different approach. We found a Node.JS fork with the name of nodejs-mobile by Janea Systems [26], that ported the original Node.JS implementation to work in the mobile environment. Since it integrates with React Native, we used it to build a platform independent application, where the same codebase is compatible with both Android and iOS. Updates to the SDK would not cause a problem with this approach, unless native dependencies that are incompatible with ARM processors are added.

3.7 Components interaction

In this section, we describe the interactions among components in our Elecctro Points. Some of these interactions already existed and were modified to account for the new loyalty system, some of them are brand new and a consequence of it.

3.7.1 Customer creating and using accounts

Figure 3.2 depicts the different interactions between a customer and the Customer CA.

For creating an account, a customer contacts the Customer CA server providing an username, and optionally a secret. If no secret is provided, the Customer CA server generates one and returns it in the response. If an account with the given username already exists, the customer is notified and no account is created.

Later, if the customer plans to interact with the network it needs an identity. To get an identity, the customer needs to enroll with the Customer CA server. To enroll a customer needs to provide its username and secret to the Customer CA server, that replies with a enrollment certificate that can be used as an identity to interact with the network.
As mentioned in Section 3.3, the customer can revoke both an enrollment certificate (if an enrollment certificate is compromised), or its account entirely (if the secret is compromised).

### 3.7.2 Customer transferring points

Figure 3.3 depicts a customer transfer points to another customer.

Transferring points is as simple as calling the `transferPoints` function in the `elepoins` chaincode. If a Customer A wants to transfer points to Customer B, Customer A needs to open the ElePoints application on their mobile device, select the transfer option and input Customer B’s username.

To achieve this, the application uses the enrollment certificate as an identity, connects to the network and calls the `transferPoints` function. If Customer A does not have enough points to cover the transfer, no transfer takes place and the customer gets notified via an alert in the application that it lacks the sufficient funds.
3.7.3 Merchant awarding points to customer

Figure 3.4 depicts a merchant awarding points to a customer.

Customers can link cashless payment information to their username in the loyalty system. Every time a customer purchases a product using cashless payments, Elecctro tries to resolve an username based on the cashless payment information. If an username is resolved, Elecctro notifies the respective merchant’s Merlin instance.

When notified, Merlin is provided with the customer’s username, the vending machine where the purchase was made, the product that was purchased, and the price of the product. With this information, Merlin queries if the machine in question is present in the machines chaincode using the queryMachineOwner function. At the same time, Merlin queries the award rate for the product in question using the queryAward function in the rates chaincode.

If the machine is belongs to the loyalty system, Merlin then awards the customer using the awardPoints function in the elepoints chaincode.

Note that Merlin only gets the bare minimum information in order to award points to the customer. That means that the customer’s payment information is kept private on the Elecctro’s side.

3.7.4 Merchant updating rates and vending machines

Figure 3.5 and Figure 3.6 depict a merchant updating rates and vending machines, respectively.

Updating vending machines consists of adding or removing vending machines identifiers from the merchant’s list in the machines chaincode. Rates on the other hand are a bit more complex. Merchants have a fallback rate that is applied to every product by default. If more fine-grained control is required, merchants can use product specific rates. Rates can also be enabled/disabled if required. To add/update a rate, merchants need to provide two rules, an award rule and a
3.7.5 Customer purchasing a discount

Figure 3.4 depicts a customer purchasing a discount.

Discount rules calculate using the amount of days until expiration date, how many points does a discount cost, and what is the percentage of the discount. This way merchants can provide bigger discounts and/or have them cost less, as a product reaches its expiration date. Merchants can also specify within this rule, the minimum amount of days until a product expires in order for a discount to be considered. Once updated, both machines and rates, take immediate effect affecting future purchases and interactions from that point on. Rates are updated in the `rates` chaincode.
Purchasing a discount is by far the most complex interaction in our solution in terms of how many components are involved from the second a customer scans a Facebook Messenger code, to the moment where customer’s points are actually used.

When a customer scans a Facebook Messenger code, Elecctro servers are notified. Since customers can link their Facebook Messenger id to their username in the loyalty system, Elecctro is able to know whether or not the customer who scanned the code is participating in the loyalty system.

If the customer is a participant of the loyalty system, Elecctro can query the customer’s points using its username by calling the `queryCustomer` function in the `elepoints` chaincode.

Elecctro then queries if the machine in question is present in the `machines` chaincode using the `queryMachineOwner` function. At the same time, Elecctro queries the discount rates for all products in the vending machine using the `queryDiscounts` function in the `rates` chaincode.

Elecctro then filters the ones that the customer can purchase and displays them to the customer in the Facebook Messenger chat. The customer can then decide whether or not it wants to purchase a discount.

If the customer decides to purchase a discount, it clicks the discount it wants to purchase, which redirects it to the ElePoints application. There the customer is asked to confirm the purchase. Behind the scenes, the ElePoints application purchases the discount by calling the `purchaseDiscount` function in the `elepoints` chaincode.

Once the chaincode function is called, the respective Merlin’s instance gets notified of a purchase discount.

Once notified, Merlin queries if the machine in question is present in the `machines` chaincode using the `queryMachineOwner` function. At the same time, Merlin queries the discount rates for the product in question using the `queryDiscounts` function in the `rates` chaincode. Merlin then checks if the amount of points the customer transferred is enough to cover for the discount.

If the customer has enough balance, Merlin requests Elecctro to apply the discount to the given product. Elecctro applies the discount and tells the customer which product to pick. The customer, picks and pays for the product, which results in a vend being registered in Elecctro’s servers. Elecctro is able to resolve the username of the customer who purchase the product with a discount, and therefore notifies Merlin that a vend with a discount was performed.

If required Merlin can refund the customer using the `refundCustomer` function in the `elepoints` chaincode.
Figure 3.7: Customer purchasing a discount

1. Scans Facebook Messenger code
2. Looks up customer's points
3. Points of Customer A
4. Query vending machine and discount rates
5. Discount rates
6. Presents discounts the customer A can purchase
7. Sends filtered discounts
8. Purchases discount
9. Modify discount purchase
10. Query vending machine and discount rates
11. Discount rates
12. Apply discount to product
13. Displays customer A which product to pick
14. Tells Customer A which product to pick
15. Picks product and pays
16. Register vend with discount
17. Notify vend with discount from customer A
18. Refund customer if required
Chapter 4

Evaluation

In this chapter, we discuss how we tested our implementation. In Section 4.1 and Section 4.2 we start by mentioning a few tools that we created to help us launch and test Electro Points. Then, in Section 4.3 and Section 4.4, we describe the testing environment and workloads we used to conduct our tests. in Section 4.5 and Section 4.6, we analyse and discuss the obtained results, compare them with the objectives discussed in Section 1.1, and come up with potential ways to improve our solution. We end the chapter, in Section 4.7, where we enumerate some scaling issues that we ran into while setting up and using our testing environment.

4.1 Elen

Elen (stands for ELElectro Network), is a command line tool that automates the deployment and maintenance of a Hyperledger Fabric network with the topology shown in Figure 3.1.

Elen uses Docker Swarm [27] to deploy and distribute all entities across multiple physical machines in a round-robin fashion, while following a set of restrictions that make sure that a few components remain on the same machine. For example, each merchant node requires to have in the same physical machine a Merlin instance, also, both the Customer CA instance and the orderer instance are responsibilities of Electro, and therefore are confined to remain in the same machine.

Elen has the following commands:

• **generate**, which is used to generate all configuration files required to launch a network. It generates Docker compose files, Merlin and Pupper (described in Section 4.2) configurations, as well as certificates for each entity, and some network required elements like the genesis block. With this command users can specify the size of the network in terms of participating merchants, allowing the generation of network topologies with different
sizes. This command is also responsible of assigning labels to machines, which restricts the distribution of instances when the network is launched.

- **up**, which is used to start up the network based on the configuration files generated by the `generate` command. This command is responsible to start all component instances in the assigned machines, as well as having all merchants join the same network, all Merlin listeners start listening to both Electcro and their node’s blockchain updates.

- **install**, which is used to install chaincode in all nodes. With this command, users can specify which chaincode to install, as well as which endorsement policy to use.

- **down** command, which is used to bring down the network. This is the opposite of **up** command.

- **destroy**, which is used to delete any configuration files. This is the opposite of the `generate` command.

- **shell**, which spawns an interactive shell that connects to a given merchant’s node instance. This allows to debug Hyperledger Fabric specific behaviour via their command line tools.

We use Elen to ease the deploy of multiple network topologies, so we can easily replay tests across different sized networks. Since Hyperledger Fabric requires many configuration files to be launched, we used this tool to reduce the development and testing time.

### 4.2 Pupper

Pupper (stands for PUPPeteER), is a sieging tool created to help us evaluate the performance of ElePoints. Pupper performs interactions with the network, faking customer requests, that try to stress the network resources to see if the objectives listed in Section 1.1 are met.

Pupper works by reading and executing “interaction scripts” that specify how customers interact with the system. These scripts are ran continuously until stopped. We interact with Pupper instances via a REST API, meaning Pupper has a lightweight HTTP server to serve it. Pupper interacts with the network using the Node SDK [24].

When a script is ran, every interaction is recorded. We therefore get metrics on which operations were executed as well as each operation latency and timestamp. We later query the metrics, and aggregate every Pupper instance’s metrics to know how many transactions happened at any point in time.

With that information, we can infer metrics like average latency and average transactions per second, and then evaluate whether or not we met our objectives.
A script is nothing more as a JSON message, that defines customer interactions and their frequency. An interaction is an action, or the consequence of an action performed by a customer that originates an interaction with ElePoints.

Pupper supports four types of interactions:

- **QUERY_CUSTOMER** - Querying a customer
- **TRANSFER_POINTS** - Transferring points from a customer to another
- **AWARD_POINTS** - Customer purchasing a product, which results in a merchant awarding points
- **PURCHASE_DISCOUNT** - Customer purchasing a discount

Using scripts we are able to reproduce results (at some extent), and to easily siege at a scale. We also have a way to parameterize scenarios, and we can therefore tune them to simulate real workloads. This means that this tool can be used to also test the network in the future, once we have a model of real workloads.

To orchestrate all Pupper instances we created Wick, a command line tool able to communicate to every Pupper instance via their HTTP API. Wick allows us to start and stop scripts at a scale with a single command, as well as collect and aggregate metrics reported by each Pupper instance for later analysis.

### 4.3 Testing environment

To setup our testing environment we used a tool called Ansible [28] to automate the download of dependencies, pull Docker [29] images, upload and compile chaincode as well as all configuration files all nodes required to participate in the network. We also used it to have all nodes join the same Docker swarm [27] to ease the distribution and deployment of every component of our architecture.

Machine wise, we settled with two configurations. We conducted our first set of tests using 30 DEV-L machines from Scaleway [30]. These machines are equipped with 4 vCPUs and with 8GB of RAM. We then conducted a second set of tests using 5 GP-L machines also from Scaleway. These machines are equipped with 32 vCPUs and 128GB of RAM.

In all tests we used one Pupper instances per machine. We also had the Orderer and the Customer CA server in the same random machine.
4.4 Workload composition

As for the workload, we decided to test two main workloads, one consisting simply of queries to customers, and a more mixed workload composed mainly of changes to customers (transfers, point awarding, discount purchasing), followed by some queries.

The Pupper scripts we used for the two workloads are the following:

- **Query workload:**
  - `QUERY_CUSTOMER = 1`
  - `TRANSFER_POINTS = 0`
  - `AWARD_POINTS = 0`
  - `PURCHASE_DISCOUNT = 0`

- **Mixed workload:**
  - `QUERY_CUSTOMER = 0.33`
  - `TRANSFER_POINTS = 0.22`
  - `AWARD_POINTS = 0.23`
  - `PURCHASE_DISCOUNT = 0.22`

Although it may seem we are only interested in how many writes our application can withstand, we thought it would be interesting to see how the network behaves in a query only workload, to analyse the difference between the two in terms of latency and throughput.

For the mixed workload, since we do not have real world data on how customers would behave, we decided to have 1/3 of queries and 2/3 in writes. Among writes, we decided to perform them in a uniform basis. We speculate that point awards and transfers would happen more frequently than purchase discounts (since customers need to accumulate points before purchasing discounts), but we wanted to try the more complex customer interaction as often as the simpler ones.

4.5 Results

After successfully setting up the network we started the evaluation. First we tested the mixed workload over the 30 DEV-L machines. We distributed 1 merchant node per machine, as well as 1 Pupper instance per machine, each performing 25 concurrent requests. To our surprise, all Pupper instances were unable to connect to some merchant nodes when asking for transaction
endorsements. Since we have an endorsement policy that requires the majority of merchants to endorse a transaction, failure to communicate to merchant nodes means that we are unable to gather endorsements and therefore unable to modify the state of the blockchain.

First we thought we had some issues in our network setup, but after stopping the script and manually pinging each merchant node we realised everything was working as intended. We tried re-running the script, this time while monitoring each machine resources, and found out that in all of them there was a process "peer node start" that was constantly over 300% CPU usage.

We hypothesize that the machines did not have enough resources to withstand the generated load, and therefore, started rejecting requests. Given that we were unable to modify the blockchain, we excluded this test completely. Regardless, at 25 concurrent requests per Pupper instance, across 30 machines, we only have 750 concurrent requests arriving at our network. Unless these would take less than 750ms, this approach would never be able to produce the amount of requests listed in our objectives.

To investigate how the network would behave in the case of a query only workload, we performed an hour and a half test, shedding the first and last 15 minutes of the results. In this test, we used 30 Pupper instances, each issuing 120 concurrent requests, in a network with 30 merchant nodes. We were able to average more than 2 000 queries per second as seen in Figure 4.1, while having an average latency of less than 1 500 milliseconds per query, as seen in Figure 4.2. The variance of both throughput and latency over time could be an indication that Pupper is not generating a uniform load correctly due to the lack of proper thread support in Node.js. During the test, CPU usage remained under 90%, which makes sense since querying only requires to read the state of the blockchain, and does not require any endorsement collection, making it a non CPU intensive operation.

Convinced that the problem was the lack of resources, we decided to try the more powerful GP-L machines. Since these machines are beefier and more expensive, we settled with 5 GP-L machines. Instead of trying to launch 30 merchant nodes, 6 per machine, we decided to be more conservative and only have 1 merchant node per machine.

Using this environment we tested the mixed payload, using 5 Pupper instances, each issuing 380 concurrent requests, in a network with 5 merchant nodes. As expected no connection errors happened. We tested for one hour, shedding the first and last 10 minutes of the results. Results varied per Pupper instance in both throughput and latency as seen in Figure 4.3 and Figure 4.4, respectively. Over time our system was able to withstand a total of 60 to 130 requests per second, while having the majority of Pupper instances report a latency of less than 4 to 6 seconds. Although some Pupper instances reported a latency as high as 2.5 minutes, they were
Figure 4.1: Throughput over time of the query only workload with using 30 DEV-L machines, with 30 merchant nodes and 30 Pupper instances

Figure 4.2: Average latency time of the query only workload with using 30 DEV-L machines, with 30 merchant nodes and 30 Pupper instances
able to eventually converge to less than 20 seconds. While conducting the test, we monitored the CPU usage of all machines, and we found the same "peer node start" process constantly using more than 150% of the CPU, although all the usage seemed to be single core. This usage percentage was enough for merchant nodes to not reject requests and therefore we were able to perform our test.

For the last test we tried the query only workload in the 5 GP-L machines. Since the query only workload puts less stress on the CPU, we decided to use a network with 30 merchant nodes, 6 nodes per machine, again using 5 Pupper instances per machine, each issuing 380 concurrent requests. We were able to average more than 2 000 queries per second as seen in Figure 4.5, while having an average latency of less than 1 500 milliseconds per query, as seen in Figure 4.6. During the test CPU usage remained under 50%, similar to the first query only workload test we conducted.

In all tests, we monitored the orderer process to make sure that it was not the bottleneck (since we have a solo orderer), but the process always remained under 40% CPU usage.

We realised that this "peer node start" process, is the process that every merchant node runs when participating in the network. Given we only have one node per merchant, that node is responsible for all endorsement requests on behalf of the merchant. Since we need the majority of merchants to endorse a transaction, for every chaincode execution, we require 3 merchant nodes to execute the chaincode. At the rate of 1 900 (380 concurrent requests over 5 Pupper instances) concurrent requests, it seems normal that every merchant node process reaches that
Figure 4.4: Average latency over time of the mixed workload with using 5 GP-L machines, with 5 merchant nodes and 5 Pupper instances

Figure 4.5: Throughput over time of the mixed workload with using 5 GP-L machines, with 30 merchant nodes and 5 Pupper instances
Figure 4.6: Average latency over time of the mixed workload with using 5 GP-L machines, with 30 merchant nodes and 5 Pupper instances.

high of a CPU usage.

4.6 Discussion

The experiments we conducted revealed an unforeseen bottleneck in our architecture. Having 1 node per merchant, responsible for all endorsement requests on behalf of the merchant, is clearly not enough to serve the amount of requests we desire to serve. Vertical scaling helped achieve some results, but this seems to be a problem that needs to be solved via horizontal scaling.

One way to tackle this bottleneck is to increase the numbers of nodes that each merchant has in the network. As long as these nodes all have the same chaincodes installed, customers can rely on them when requesting endorsements.

Each customer could then implement a selection strategy, that would help pick which node to use when requesting an endorsement. We can try multiple strategies, from a geographical strategy that can be used to try to minimize latency, to a round-robin strategy that can be used to distribute load among nodes.

This approach requires further evaluation to reveal if there are any other bottlenecks in the system. It is also not clear if having more than one node per merchant is desirable. This results in more hardware costs for each merchant (depending on how many more nodes are required in the new architecture), which might outweigh the benefits of having the loyalty system entirely.

Nonetheless, the proposed approach shows it is possible to build a decentralized blockchain-
based loyalty system that is able to integrate with Elecctro’s vending platform, that although not able to achieve the throughput and latency we intended on a mixed workload, it was still capable of reaching more than 2,000 customer queries per second with a latency lower than 2 seconds. This is a major stepping stone towards an implementation that attempts to achieve a high request throughput while trying to maintain a latency lower than a customer’s tolerable waiting time.

4.7 Scaling issues

Throughout evaluation we ran into some scaling issues. When installing Node.js chaincode, Hyperledger Fabric delays initialization until the first interaction with the chaincode happens. This initialization can take minutes to finish, since each chaincode initialization spawns a Docker process per merchant node. Since we could not afford to have some chaincodes instantiating while we were testing our implementation, we had to manually interact with every chaincode that was in the merchant nodes by calling a chaincode function that asks every merchant node for endorsement. This triggered a mass chaincode initialization that took 40 minutes to 1 hour to complete. Sometimes some Docker processes would not start at all, so we had to bring down the network, so we could install initialize them all over again.

Another issue we ran into, was mass metric collection. When we asked to collect metrics, Wick reaches every Pupper via their REST API. For metric collection, Wick expected a JSON response with a list of metrics collected so until that point. After longer periods of time, the metrics volume was so high, that each Pupper HTTP server would crash while trying to serve the request, losing access to the data. To tackle this we had each Pupper, when requested, write their metrics into a file, that it would statically serve for Wick to download.
Chapter 5

Conclusions

We conclude our thesis with this chapter. We start by going through some achievements that we were able to accomplish while implementing our Eleccctro Points, with a focus on the lessons we learned. We finish the chapter by shedding some light on future work that can be performed in order to improve our solution.

5.1 Achievements

The main goal of this thesis was to implement a scalable and performant blockchain-based loyalty system that would integrate with Elecctro’s vending platform. We were able to partially accomplish this objective in the sense that, by the end of the work, we had a working proof of concept of a loyalty system on top of a Hyperledger Fabric network that fully integrated with Elecctro’s vending platform.

Although we were unable to test its full performance due to a limitation in the proposed architecture, we think that this implementation was still an achievement. We were able to integrate with Elecctro’s current infrastructure, while allowing merchants to cooperate together without trusting each other. With this system, merchants are able to mediate discount and awards rules, as well as which machines are participating in the system.

Remarkably our solution does not drastically modify the way customers currently use Elecctro’s platform to account for the new functionalities.

Another achievement was the whole toolset that we developed throughout the thesis. Elen, that helped us save countless hours of configuration and deployment, by automatically configuring and launching a network from scratch simply by issuing a couple of commands with the help of Ansible and Docker Swarm. Pupper and Wick helped us inject load so we could evaluate our implementation at a scale, again just by issuing a couple of commands to run scripts and
fetch metrics. With some effort, Pupper could perhaps be open-sourced to help others evaluate their Hyperledger Fabric networks.

Last but not least, we learnt a few lessons throughout our thesis, the most important ones being:

- Sieging systems at a scale is not as easy as it seems, and is also very expensive. Being able to generate a load that complied with our objectives was not easy, and required multiple workers that needed to be orchestrated in order to launch workloads and collect metrics.

- Test before implementing. We think that the outcome of this thesis could have been much better if we were to run preliminary tests with some very simple chaincodes on top of our architecture. We would probably have noticed the bottleneck of the system earlier, granting us the ability to get rid of it in a much earlier stage of our implementation.

5.2 Future Work

Given the limitation of our architecture, a clear step towards improving our implementation would be to re-iterate the architecture to try mitigate this bottleneck. Since this means increasing the number of nodes per merchant, we would first need to analyse if the cost of the extra hardware outbalances the benefits of having a blockchain-based solution.

Our implementation would also benefit from a re-write of some of its components in a more performant language. The chaincode could be written in Go or Java, which would allow chaincode initialization to be instant, saving time when installing/updating chaincode.

Pupper could also be written in a language with proper threading support, enabling us to fully use each machine resources to achieve as many concurrent requests as possible. We could also have Pupper use the Hyperledger Fabric command line tools to interact with the network to prevent overhead of using the SDK (additional logic to help developers interact with any network).
Bibliography


