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

André Mendes
Instituto Superior Técnico, Universidade de Lisboa

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.

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

Elecctro 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 Elecctro’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

I. 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 Elecctro’s vending platform. Elecctro offers this vending platform as a service for different merchants that are interested in boosting the features of their vending machines. With Elecctro’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.

As vending machines are often in unattended commercial zones, depending on the time of the day, they become easy targets of robbery and vandalism. Elecctro’s vending platform was extended with active security measures to help prevent these. 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.

In this paper 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.

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 Elecctro’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

Because these merchants are usually from different institutions, the lack of centralization is problematic. We require a distributed solution that allows merchants to cooperate without necessarily trusting each other. Due to the number of vends in 2015, the loyalty system needs to be capable of serving many customers in parallel.
In this paper we design, implement and evaluate Electro Points, a blockchain based loyalty system on top of Hyperledger Fabric [1], for vending machines that belong to different merchants and that would allow them to participate without fully trusting each other.

II. 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. Electros 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 Electros 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 Electros 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 is the amount of merchants we need to support. According to Electros 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.

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

III. BACKGROUND

A. Electro’s Vending Platform

Electro’s vending platform connects a vending machine, no matter how old, 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, like products stock and vends.

Another consequence of connecting the vending machine to the Internet is that customers can now use cashless payments. Customers 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 Electros 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.

B. 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 1), 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.

C. 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.

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 they would 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 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 III-D and Section III-E respectively.

D. Public vs Private Blockchain

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.

### E. 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 III-D, 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 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. In blockchain it is used by having its peers solve a difficult problem before being able to append a block to the blockchain. 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.

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.

### F. 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. 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 enables the possibility of a pluggable consensus mechanism that can be chosen to better fit the application needs.

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 [9] or Ethereum [7], 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].

IV. IMPLEMENTATION

A. Proposed architecture

Figure 3 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 IV-D and Section IV-H 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 sub-sections.

B. Orderer

As discussed in Section III-F, the orderer is responsible for ordering transactions. There are a few implementations for an orderer service [11], 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 Elecctro to perform this role so that merchants do not have to decide who will be responsible for it. Although an orderer, in this case Elecctro, can read and block transactions, these can be encrypted so that Elecctro can not be selective about which transactions to block. This ensures that Elecctro 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.

C. 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 III-F, 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 Elecctro, 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) [12] 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.

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.

D. Chaincode

In Hyperledger Fabric’s core we have chaincode. Chaincode is code that can be installed in all endorsing peers that can be executed on demand by clients. Like mentioned in Section III-F 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

E. 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.

F. 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.

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.

G. 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.

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.

Customers can use this chaincode to transfer points to other customers and to purchase discounts. Merchants can use this chaincode to award customers with points when they purchase a product using a cashless payment.

H. 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 vend 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.
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.

I. ElePoints

ElePoints (stands for Electro 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.

Customers can 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 Electcro’s servers as described in Section IV-C.

V. Evaluation

A. Elen

Elen (stands for ELEctcro Network), is a command line tool that automates the deployment and maintenance of a Hyperledger Fabric network with the topology shown in Figure 3.

Elen uses Docker Swarm [13] 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 (e.g. merchant nodes and Merlin instances).

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.

B. 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 II 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 [14].

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.

C. Testing environment

To setup our testing environment we used a tool called Ansible [15] to automate the download of dependencies, pull Docker [16] 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 [13] 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 [17]. 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 instance per machine. We also had the Orderer and the Customer CA server in the same random machine.

D. 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.

E. 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 an 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, while having an average latency of less than 1 500 milliseconds per query, as seen in Figure 5. 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 6 and Figure 7, 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 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 8, while having an average latency of less than 1 500 milliseconds per query, as seen in Figure 9. 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 high of a CPU usage.

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

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.

**REFERENCES**


---

**VI. CONCLUSION**

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 Electroc’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.