POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Computer Engineering

Master Degree Thesis

Optimal Endorsement Policy for network-wide distributed blockchains

Supervisors
Prof. Paolo Giaccone

Candidate
Vittorio Pellittieri

Academic Year 2021-2022

Abstract

Blockchain is a technology that allows storing data in a secure and decentralized way. One of the main reasons why the interest in this technology is growing is because it can provide trust between untrusted parties without the use of a centralized entity. Hyperledger Fabric is a permissioned blockchain project maintained by The Linux Foundation meant for environments in which system performance is a key factor. For instance, HF architecture faces three consecutive phases called Execute, Order and Validate. This behavior differs from other blockchain architecture like Bitcoin in which an Order-Execute-Validate procedure is followed. During the Execution phase, Simulation happens: a proposal of the transaction, called Endorsement request, is sent to specific peers named Endorser peers or EPs. EPs execute the received endorsement request and send the obtained output back to the client in a message called Endorsement. This phase is called Simulation because, while executing the transactions, EPs don't change the content of the data they are securely storing. The client needs to receive the Endorsement results from a combination of the above-mentioned EPs defined by a policy called Endorsement Policy. This policy is defined in a way that reflects the agreements taken between the participating organizations. This thesis describes Blockchain technology, HF architecture, and Fabric SDK, which is a Development Kit that enables the development of applications that interact with a Fabric network. Afterward, it is presented an optimal procedure of spreading the endorsement requests across the EPs to reduce the overall latency of the Endorsement procedure. This is done through an algorithm called OPEN that keeps track of the endorsement delay history of the network's EPs. While respecting the Endorsement Policy, OPEN chooses the peers that performed better in the past transactions because they are the most likely to be the fastest in the next one. To keep updated the response delay history of not selected EPs, OPEN sends endorsement requests to additional peers. This will facilitate the better making of assumptions about the peers to endorse during the next transactions. Finally, having verified that the Endorsing Peers can queue requests when higher loads of transactions are submitted, an evaluation of OPEN performances in choosing the least congested EPs is carried out in compliance with other endorsing peer selection algorithms.

Contents

Chapter 1

Introduction

1.1 Blockchain Technology

Blockchain is a technology that enables a network of entities, which do not trust each other, to maintain a shared secure information that can be modified only if everyone gives their consent. This consent is reached through the coordinated interaction of the entities and it makes it possible to not need someone who certifies the legitimacy of every action. For this reason blockchain can be used in several contexts where typically is needed someone who certifies the truthfulness of a statement. For example, if Bob wants to send a quantity of money to his friend Alice, the bank before doing the operation will check if Bob has enough money in its account to make this operation. If there was not somebody checking on the legitimacy of the financial movements, it would not be possible to have a global payment network that works. **Cryptocurrencies** are the financial declination of the blockchain as they enable to access financial services without the support of any bank as the technology in which they lean permits only legit actions. Blockchains can also be used by a group of cooperating but untrusted companies that want to interact securely without having to hire a certifying entity that assures that every issued action is legit. Since company's data must be maintained confidential there must be a way to let it access only by authorized person: **private blockchains**.

1.2 Hyperledger Fabric

Hyperledger Fabric is an open-source software implementing a private blockchain which offers several building blocks that can be composed depending on the need. HF was created having in mind other blockchain's flaws and it changes the way transactions are handled. In other blockchains all the transactions issued are chronologically ordered before being executed in order to avoid anybody from spending the same assets more times. HF instead first executes the transactions in a way that it can remove as soon as possible the ones that are not considered legit speeding up the whole process. As a matter of fact Fabric is one of the fastest private blockchains at executing transactions.

1.3 Endorsement

Endorsement is the first step to face for each transaction that has to be inserted inside the blockchain and consists of a preliminary execution of the transaction. As a matter of fact who wants to change the data in the blockchain must request to other participants of the network the endorsement of their transaction. The set of entities that will be asked to endorse the transaction is defined by a so-called **endorsement policy** which uniquely defines if the transaction can be considered legit. Since for a matter of network or service availability certain entities may require more time to be contacted, selecting the right combination could improve the overall performance of the transaction flow.

1.4 Goal of the Thesis

The goal of this thesis is to analyze an optimal endorsement selection algorithm called **OPEN** that aims at selecting the best entity to ask the endorsement for a transaction in a way that it can reduce the time needed for the endorsement phase. Choosing the right entity to ask for approval is not trivial since the way they behave varies with time making it difficult to predict. OPEN deals with the unpredictability of the endorsement time by accumulating information about how the entities behaved in the past, which enables better assumptions about the entities that could be the fastest in responding.

1.5 Content of this thesis

The thesis is structured as follows:

- **Chapter 2** describes blockchain technology and its application context
- **Chapter 3** introduce Hyperledger fabric architecture, transaction flow and endorsement policy
- **Chapter 4** introduce Fabric Node SDK and how it handles endorsement phase
- **Chapter 5** analyzes OPEN policy design
- **Chapter 6** describes how experiments were conducted and their consequent results.

Chapter 2

Blockchain Technology

2.1 Definition

Blockchain is a secure ledger shared across multiple untrusted entities in which if anybody tries to alter maliciously the content of the data stored, the other participants of the network will notice it. This technology differs in the way it stores the data: as a matter of fact blockchain takes its name from a unique way of storing it as the data gets shaped in the form of a **chain** made from **blocks**. The domain of the data stored inside the blocks depends on the specific blockchain: Bitcoin's blockchain for example stores transactional data which is nothing more than records that attest the exchange of a financial asset. The blocks store not only transactions but also a reference to the block which is preceding on the chain called **parent block**. It is also thanks to this particular structure that blockchains can be considered immutable and secure.

2.2 Origin of Blockchain

Blockchain technology was firstly unveiled in 1991 [7] by Stuart Haber and W. Scott Stornetta. They invented a way of creating timestamped digital documents that could not be compromised or later modified thanks to the use of cryptography. They later redesigned their project to work with Merkle trees which enabled to store more documents inside the block and, consequently, to be more efficient. In 2008 Satoshi Nakamoto revamped this technology with the whitepaper called "Bitcoin: A Peer-to-Peer Electronic Cash System" [31]. Nakamoto improved blockchain by removing the need for a trusted party that certifies timestamps and by shaping the rate of block creation [2].

With Bitcoin, he aimed at creating a financial system that was unlinked to banks and usable without any geopolitical restrictions. Bitcoin's software was later released in 2009 officially giving birth to the first of a series of numerous cryptocurrencies that are nowadays at the center of media attention

2.3 Blockchain types

Blockchain are classified depending on who can access their services. For instance, they are called:

- **Public** or **Permissionless blockchain** if they are open to anyone (cryptocurrencies like Bitcoin or Ethereum are part of this category).
- **Permissioned blockchain** if they allow only identified entities to access them. Thus their services may be limited only to the employees of a company. An example of a permissioned blockchain is **Hyperledger Fabric** which will be discussed in the next chapter.

2.4 Data Structure

Blockchain keeps its dataset inside blocks all sequentially linked together through the use of **hash functions**.

2.4.1 Hash functions

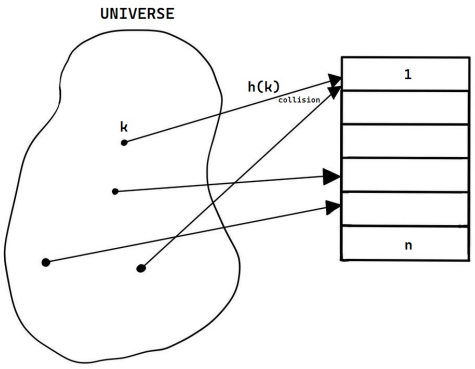


Figure 2.1: Hash function

A hash function is a deterministic function that looks random. As shown in Figure 2.1, hash functions map an input that belongs to an infinite set of elements into a space of finite outputs. The output of the hash function is called **digest** or simply **hash value** and it must be impossible to derive the input from which it was generated. As a consequence if just one bit of the input is modified, the output should look different from the one returned with the unchanged input in a way that it can not be defined a measure of distance between the inputs.

Properties

Hash function probability of giving a certain output j is:

$$Pr(h(k) = j) = 1/n$$

which means that it is uniformly distributed and there is not an output that occurs more than others.

In addition for a pair of not equal inputs k1 and k2:

$$\forall k1 \neq k2, Pr(h(k1) = h(k2)) = 1/n$$

it is possible to affirm that there is independence between their output values.

Collision

Hash functions should minimize the possibility of having **collision**, which happens when two elements have the same digest value.

2.4.2 Blocks

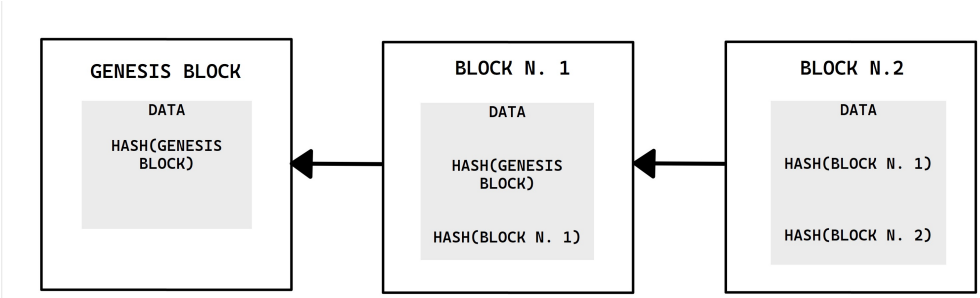


Figure 2.2: Chain of blocks

blocks are the building pieces of the blockchain's data structure. A block, as shown in Figure 2.2, usually contains:

- A set of **transactions**
- The **digest of the block**
- The **digest of the parent block**
- A **timestamp**
- A **nonce**, which is a random alphanumeric string needed to maintain the process secure.

The first block of the chain has particular properties and it is called **Genesis block**.Figure 2.2 shows the way blocks are linked and what they contain.

2.4.3 Links

It is by storing the hash of the parent block that the **links** are created. Having these links between the blocks makes the blockchain secure.

Supposing that an attacker wants to compromise the ledger's content by modifying the value of a block if also just one bit of the ith block's data changes his digest will change. Since the ith digest changed, the i+1th block does not store anymore the ith digest as parent hash. This makes the chain invalid from the i+1th block until its tail.

Fork

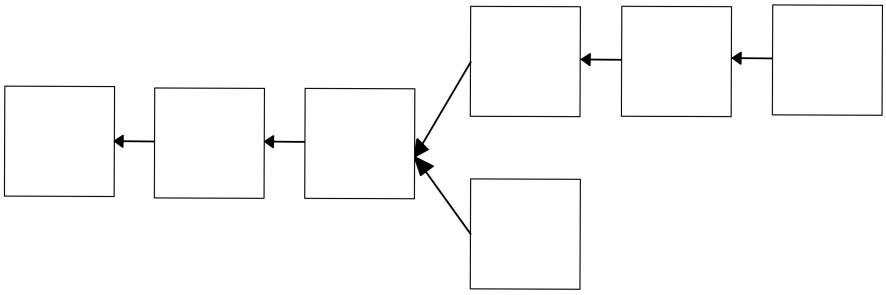


Figure 2.3: Fork

When two blocks refer to the same parent there is a so-called **fork** [13] which means that inside the network there are two different versions of the same ledger. This way two blocks may refer to the same parent is shown in Figure 2.3. Even if there is not a favorite chain, peers will always prefer the one which is the longest since it means that it took more effort to be calculated and consequently it is less likely to be altered. This principle, called **longest chain rule**, enables to maintain the same version of the ledger across the network's nodes.

2.4.4 Block time

Blocks, once created, are not directly added to the ledger. A new block can be appended at the tail of the chain only after a period of wait called **Block time**. This period lasts up to 10 minutes for Bitcoin and almost 15 seconds for Ethereum blockchain. Block time surely reduces the transaction throughput of the network but, together with the longest chain rule, prevents attackers from tampering blockchain creating a longer alternative chain.

2.5 Consensus Mechanism

Consensus mechanism is a set of procedures that allows the network's node to have the same version of the ledger. Before attaching a block to the chain the majority of the network needs to agree on its validity. There are different types of Consensus mechanisms:

2.5.1 Proof of Work

Blockchain implementing a **Proof of Work** mechanism requires a proof of having done an amount of computational work in order to add a new block into the ledger. The nodes in charge of creating new blocks are called **miners**.

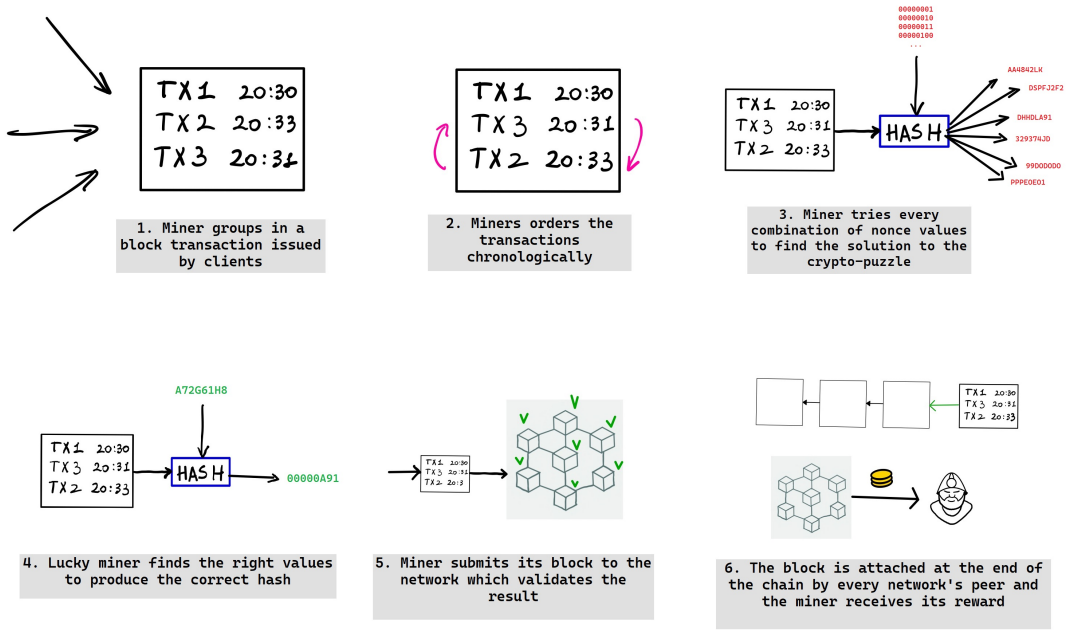


Figure 2.4: Proof of Work miner's block creation steps

A miner, as shown in Figure 2.4, must face several steps to create a block and earn a reward for it:

1. It collects transactions signed through cryptography by the users that want to send their assets [36]
2. It orders them chronologically. Ordering transactions is a necessary step as it prevents attacks like Double Spending. [14]
3. It tries every possible combination that could solve a puzzle associated with that block. Usually, this puzzle consists in finding a digest that satisfies some mathematical rule by using as input the block and by varying a nonce value. In Bitcoin for example this puzzle requires a digest that starts with a predefined number of zero called **difficulty**.
4. If the miner is lucky, it finds the right value of the nonce that generates the expected digest.

5. Having found the solution, it broadcasts the block it created to the whole network that executes and validates each transaction of the block.

6. If all the transactions are considered legit, the block is inserted into the ledger by all the peers and the miner receives a reward.

PoW is the first consensus protocol that was adoperated and for this reason, is the one that was more tested. PoW has been criticized for the waste of energy given by the mining process since every miner does computations to find the right block to append to the ledger. However, it is one of the most secure consensus protocol because attackers need to do unfeasible investments in hardware facilities to alter the content of the ledger. [16]

2.5.2 Proof of Stake

Proof of Stake is a consensus mechanism that does not require heavy hardware computation like PoW but instead asks for financial investments. The actors involved in the blocks' creation are no more called miners but **validators**. Participants to be eligible as validators must block an amount of tokens as collateral in a process called **staking**. In PoS the validator is chosen by an algorithm that prioritizes who staked more tokens in the blockchain. By being so, PoS does not require every node to be involved in the validation process but only the ones that are selected by the algorithm making it possible to reduce the overall computation.

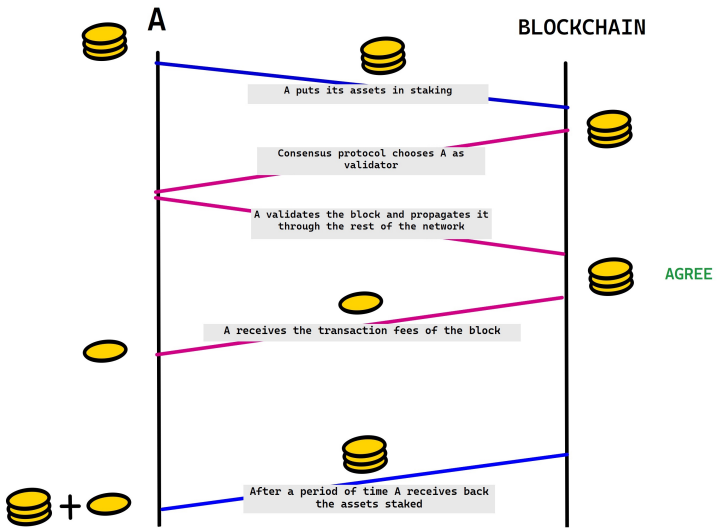


Figure 2.5: Proof of Stake validator's block creation procedure

A node, as shown in Figure 2.5, must face several steps to create a block and earn a reward from it:

1. It puts an amount of token in staking to be eligible to be a validator. These staked assets are blocked and can not be used for a period of time.

2. The consensus protocol chooses the node as a Validator.
3. The node validates the content of the block and then broadcasts it through the network. The rest of the network later agrees with the block's content and appends it to the back of the chain.
4. The validator receives the transaction fees of the block.
5. Later the node will receive back the assets it staked and it can decide if stake them again.

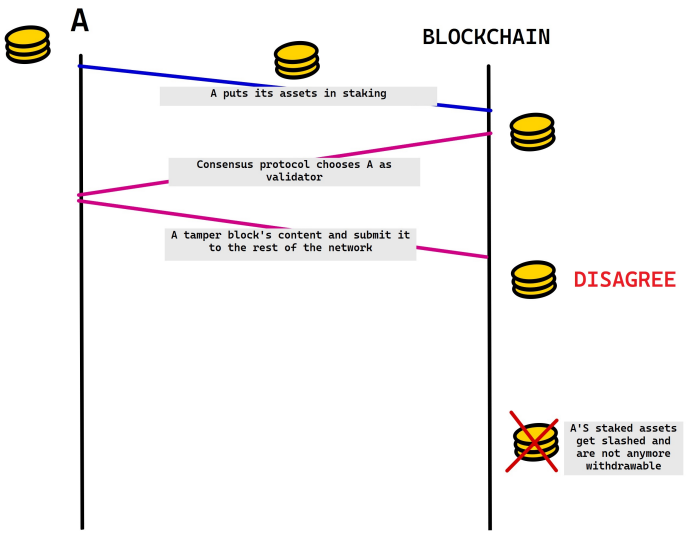


Figure 2.6: Proof of Stake validator's slashing procedure

The use of collateral discourages attackers from submitting an invalid block. As shown in Figure 2.6, if an attacker wants to change the ledger's content:

1. It puts an amount of token in staking to be eligible to be a validator.
2. The consensus protocol chooses the malicious node as a Validator.
3. The validator changes the content of the block in a way that it can profit from it. It then broadcasts the block to the whole network that executes it and notices how the transactions do not respect the actual state of the ledger. Since the node submitted a malicious block the assets it staked are automatically sent to an address that is not accessible by anyone making them no anymore withdrawable. This process is called **slashing**.

For its characteristics, PoS makes it possible to reach a higher throughput and scalability than PoW but it is a new technology that still needs to be tested in bigger networks.

Not requiring the use of hardware facilities which incentive cost increases quickly, PoS may phase centralization on the decision process. For this reason, some blockchain selects the validators randomly.

2.6 Smart Contracts

2.6.1 Definition

Blockchains can run distributed programs called **Smart contracts** which can be seen as digital contracts that lean on the reliability and immutability of the blockchain infrastructure. Smart contracts are gaining popularity because they can automatize a variety of processes that otherwise would need a centralized entity super visioning their correctness. It is possible to define a smart contract by writing code that is expressed as set of conditions that could trigger a modification of the ledger's state. Ethereum [27] gives the ability to create Smart Contracts through the use of **Solidity** language.

2.6.2 Use case

An example of how smart contracts logically work could be the sale of a bike:

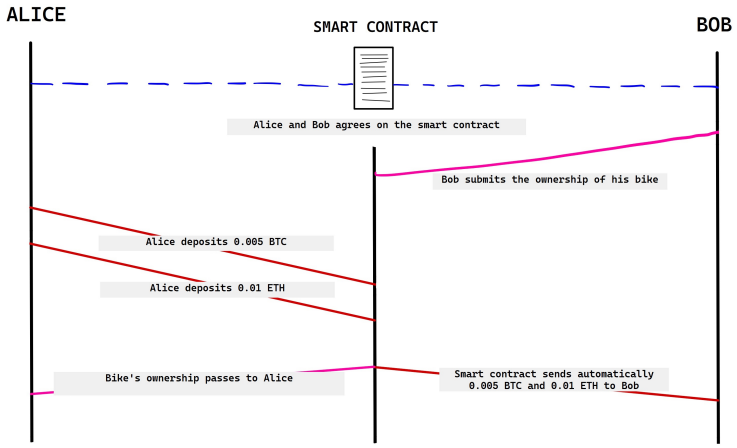


Figure 2.7: Successfully finished smart contract

Assume that Bob wants to sell his bike to Alice but he wants to put some conditions on the sale. Bob requires that Alice sends within 1 month a specific amount of crypto (0.05 BTC and 0.01 ETH) to him in order to have the ownership of the bike. If Alice does not respect one of these conditions she will not have the bike. They decide to automatize the sale through the use of a smart contract by expressing through code the agreed conditions of the exchange. They do not need indeed any intermediary that assures that the right amount of money is sent or that the bike's ownership has changed because blockchain will

do it. If Alice sends the amount of agreed assets, she will have the bike and its ownership will be immutably written on the blockchain, as happens in Figure 2.7.

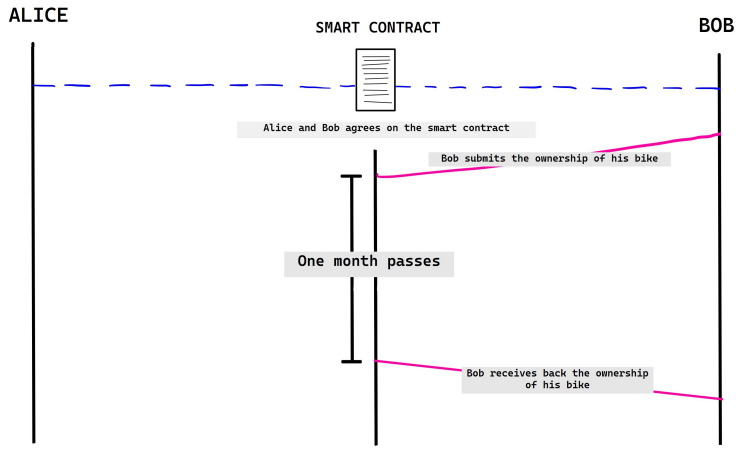


Figure 2.8: Unsuccessfully finished smart contract

Otherwise, as shown in Figure 2.8, if Alice does not send the quantity of cryptocurrencies within the chosen time frame she will not receive bike ownership.

2.7 Application contexts

One of the main innovative blockchains' features is that they enable to not rely on a trusted entity to exchange data securely. In the case of cryptocurrency, it is not needed an entity like a bank that certifies if everybody is issuing legit transactions. Not depending on a centralized entity indeed reduces strongly costs and overcomes geopolitical barriers. It also removes the existence of a single point of failure making it always possible to access the services. For these reasons there are many contexts where using blockchain could be beneficial:

2.7.1 Open access to financial services

Cryptocurrencies could help people who live in **developing countries** to have access to financial services and to issue financial transactions more quickly and with a lower risk of fraud or theft. [30]

2.7.2 Tracking food supply chain

Since this technology is not just about financing, blockchain can be used to efficiently track **agricultural products** from the moment they are planted until they are served to

consumers' tables, allowing to detect any failure inside the food chain and avoid dangerous food poisoning. [26]

2.7.3 Process automatization

Thanks to smart contracts, blockchain could be a game dealer for industries as it would reduce overhead given by the bureaucracy of the processes. Smart contracts can be used by companies to automatically issue a transaction whose price depends on some condition of the exchange. For example, its price could be related to the time that shipping took and the quality of the delivered goods.

Chapter 3

HyperLedger Fabric

3.1 What is Hyperledger fabric

Hyperledger Fabric is one of the leading projects sustained by the **Linux Foundation**, a no-profit consortium that strongly believes in open source software and aims at incentivizing technology development. [12] Fabric enables to execute coherently applications written in commonly used languages over multiple peers as it was in a single global blockchain. For this reason, Fabric is the first distributed operating system implementing a permissioned blockchain. Fabric being an **open source software** can be exploited in several contexts where trust can not be assured. Fields like food safety, dispute resolution, or trade logistic could be an example of application contexts.

3.2 Execute-Order architecture limitation

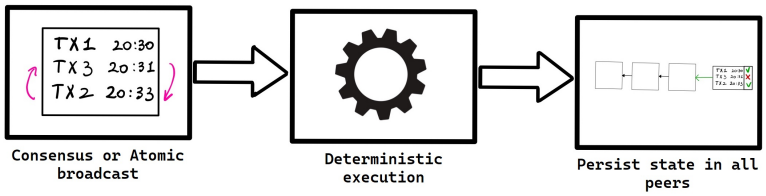


Figure 3.1: Order-Execute architecture

HF was built having in mind blockchain's (both public and private) typical flaws:

- Blockchains make every peer execute **sequentially** all the transactions contained inside a block implementing the so-called **Order-execute architecture**. This type of behavior is illustated by Figure 3.1. This architecture could make throughput performance degrade especially in the case in which computationally expensive smart contracts are intentionally submitted to the network with the goal of congestion it. Since every peer executes every transaction, the network could be subjected

to the risk of a Denial Of Service attack. The way Ethereum, a public blockchain, copes with this weakness is by using **gas**, a sort of a bill delivered to the submitter whose cost depends on how many resources were needed to execute the submitted Smart Contract.

- After having reached consensus, all the transactions executed by the peers must give a **determined output**. Otherwise, ledger will fork going against the replication requirement of the blockchain. This is usually solved by using domain-specific languages, that by restricting functionalities, make the execution return only determined outputs.

- Blockchains exploit a **fixed consensus protocol** that defines the rules to validate a transaction. This is hard coded into the protocol and it does not let define at application level a specific trust model that could fit the needs. Smart contracts must stick to the original predefined consensus protocol.

- Permissioned blockchains make all the peers run every smart contract. In certain contexts, it may be needed to have **confidentiality** of the ledger state and the smart contract logic. Public blockchain solves this problem by using cryptographic techniques but this requires a big overhead on the computation needed to maintain security properties.

3.3 Key concepts

3.3.1 Ledger

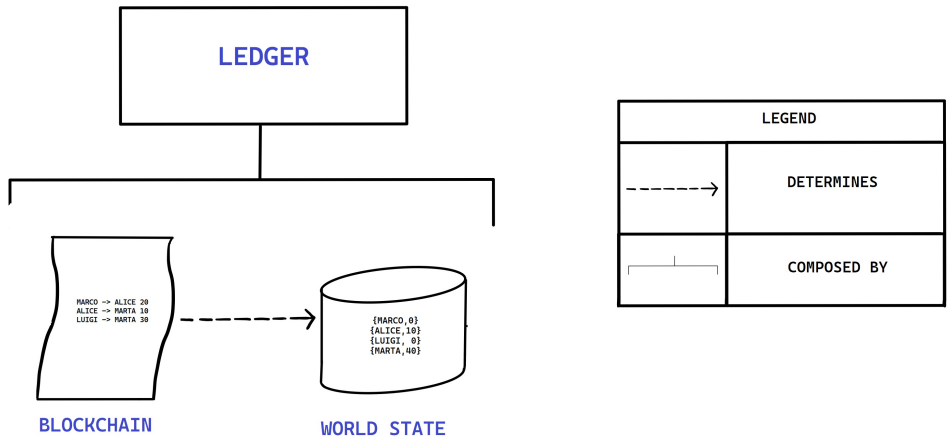


Figure 3.2: Ledger

Ledger stores the data of the blockchains and it is composed of two different but related parts [11], as shown in Figure 3.2:

- **Blockchain** preserves the blocks containing the transactions that have been made, both valid and invalid. Once written its content is immutable because of blockchain's properties.
- **World state** maintains the actual state of assets deriving it from the blockchain. This makes the current state of the asset more comprehensible by the application. Otherwise, the application should traverse the whole transactions log to derive the value of the current assets. World State usually stores its data in the form of a key-value tuple.

3.3.2 Chaincode

Smart contract contains the logic with which it is possible to query and modify ledger's data. Smart contracts may be written by untrusted developers. For this reason, every trial of modifying ledger's data must be validated first. **Chaincode** envelops one or more smart contracts and manages how they're deployed and available to be accessed. [18] For simplicity, it is also named chaincode the smart contract that is contained.

3.3.3 Channels

Channel enables to create a subchain with a part of the network making it possible to maintain confidentiality between them. Channel could be used to create private communication links between parties that do not want their transactions to be visible to every node of the network. [3]

3.3.4 MSP

Since Fabric is a Permissioned Blockchain, it requires every node to be known from the other ones. This is carried out by a component called **Membership Service Provider** (**MSP**) that associates each node with its identity.

3.3.5 Peer Gossip

Gossip enables each node to have an updated view of the network through the exchange of messages by using TLS encrypted tunnels. [6]

3.3.6 Organization

Every network's node is owned by an **organization** that can be defined as a group of entities that trust each other. Organizations can both represent a company or a single unit.

3.4 Fabric Nodes

Fabric network is composed by several modular components that cohoperates to build a permissioned blockchain. An example of a network composed by 3 organizations is illustrated in Figure 3.3.

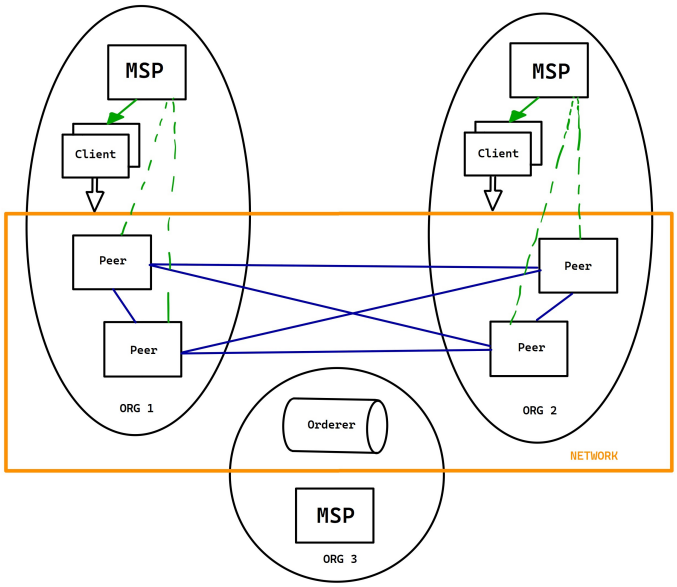


Figure 3.3: HyperLedger fabric dummy network composed by 3 organizations

3.4.1 Peer

Peer is the node that holds one or more instances of the ledger. Peer also can have installed one or more chaincode that can be used to query and modify ledger's data systematically. Every peer has a preinstalled system chaincode which provides necessary functionalities. Chaincode is placed on a separate docker container to keep the peer and its data isolated. [15]

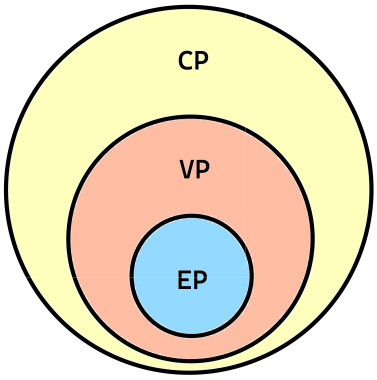


Figure 3.4: Subdivision of peer's types

It is possible to list different types of peers depending on the role they are assuming:

- **Committing Peer** or **CP** is the peer which stores an instance of ledger. CPs that do not store any chaincode may be used for keeping data replicated.

- **Verifier Peer** or **VP** participates in the validation process that occurs before a new block is added into the ledger.
- **Endorsing Peer** or **EP** is a peer that is involved in the **endorsement process** where single transactions are executed before being chronologically ordered.

Every peer is a Committing Peer since in order to maintain reliability in the validation or endorsing process it is needed an updated version of the ledger. The way peers are partitioned is shown in Figure 3.4.

3.4.2 Orderer

Ordering Service Node (**OSN**) or **Orderer** is the node that establishes the global order of the endorsed transactions in a way that only legit ones can succeed. Orderers do not have any knowledge about ledger's state, implementing a modular consensus protocol that can be easily replaced depending on the needs. All the orderers together create the so-call **Ordering Service**.

3.4.3 Client

Client represents the user that communicates with the network through the application. Clients need to be authenticated by the MSP in order to issue any action inside the blockchain and they have the responsibility to orchestrate the transaction during part of its process.

3.5 Transaction flow

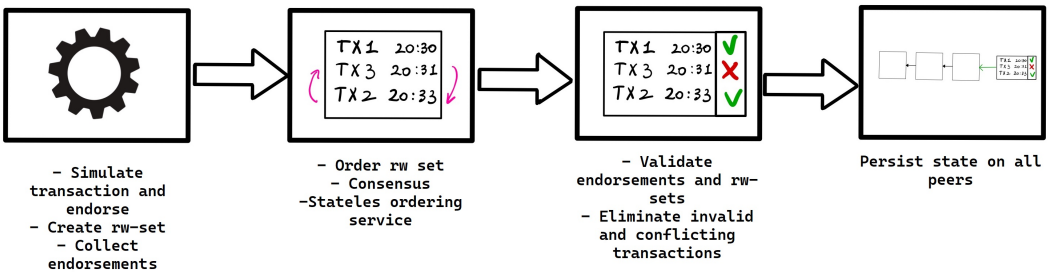


Figure 3.5: Execute-Order-Validate architecture

Fabric offers a way of executing transactions before globally ordering them enabling the parallelization of their execution. This new architecture is called **Execute-Order-Validate** and the steps it takes to commit a transaction are shown in Figure 3.5. The process of executing single transactions is called **endorsement** and it makes possible removing as soon as possible any malicious transaction that is not considered legit. In the EOV architecture every transaction must pass through three different phases to be inserted inside the ledger:

3.5.1 Execution

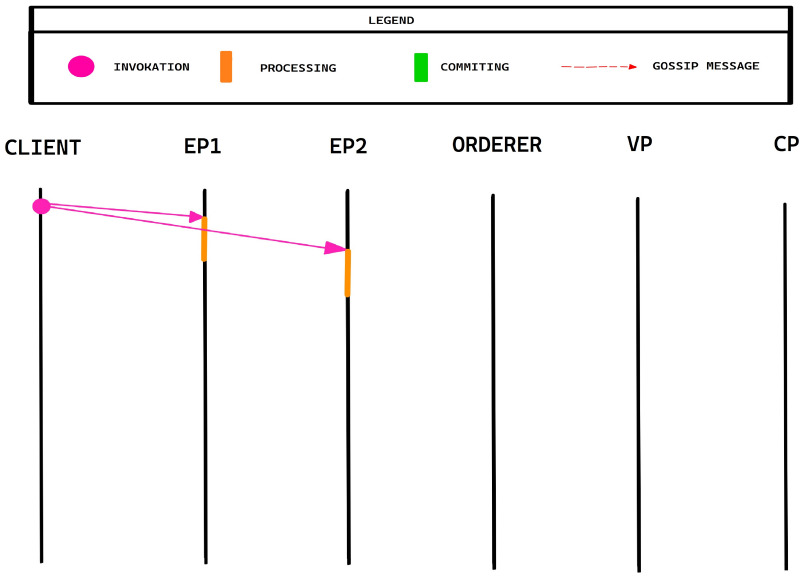


Figure 3.6: Execution phase

At the beginning of the **Execution** phase, the client signs the **transaction proposal** and sends it to a group of Endorser peers specified inside the chaincode.

The proposal is created by packing:

- **Client's identity**
- The transaction payload containing the **operation** that the client wants to do to the ledger
- **Parameters**
- Chaincode identifier
- Nonce
- Transaction identifier derived from the nonce and by the client identifier.

Once received the proposal, the EPs execute the submitted transaction in a process called **simulation** where peers produce an output not changing the actual content of their ledger. The output is composed of the **writeset**, which resumes the state changes that the operation does, and the **readset**, which refers to the values that were read during the execution and their version number. The transaction proposal is serviced by only accessing the local copy of the ledger. Furthermore, there is not any communication between peers during the simulation phase. Once finished the execution, the EP signs the output of the execution and sends back to the client the resulting packet which is called

endorsement. The transaction will be considered legit only if the outputs produced by all the simulations are identical. Otherwise, it means that it produces inconsistent states among the peers and it can not be accepted. The whole procedure is illustrated in Figure 3.6.

3.5.2 Ordering

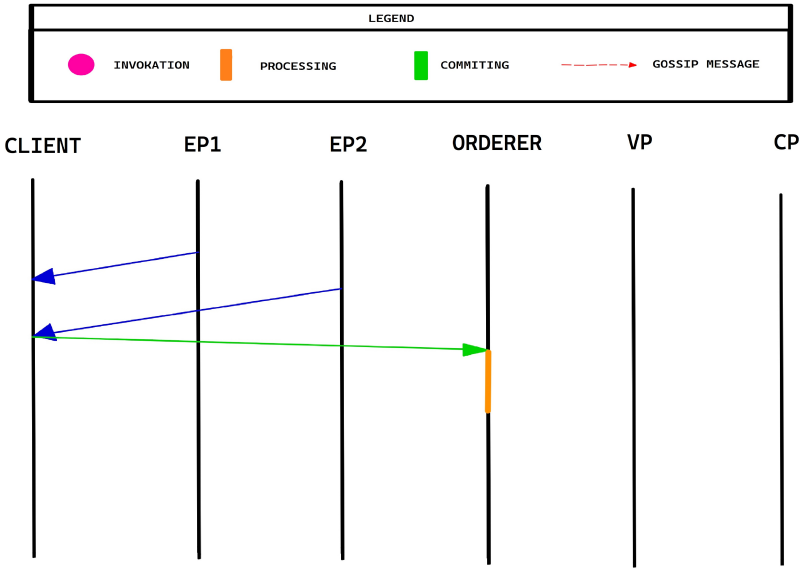


Figure 3.7: Ordering phase

After that the client collected enough endorsements from the EPs, it packs them in a transaction together with the transaction payload and metadata. Client sends it directly to the Ordering Service which has the only role to establish the global order among every transaction received. As a matter of fact orderers, for the modularity of HF, do not have any information about the ledger state and can not consequently affirm the legitimacy of the transactions. This enables the consensus to be separated from execution and validation. The transactions received are later grouped into blocks whose size is determined by a predefined parameter or by a timeout. Packing the transactions into blocks makes the blockchain's throughput higher. The whole procedure is illustrated in Figure 3.7.

3.5.3 Validation

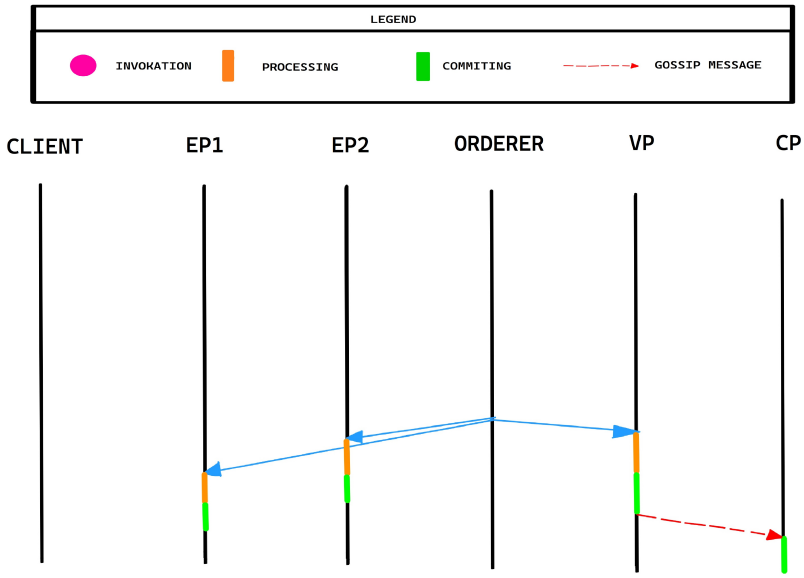


Figure 3.8: Validation phase

Once the VPs have received the transactions[1] that were previously grouped by the Orderer into a block, it is their role to:

1. Check in parallel for each transaction if the endorsement policy was satisfied

2. Search sequentially read-write conflicts for each transaction by comparing if the version of the readset is the same that is locally stored inside the peer. This makes possible detecting double-spending attacks and consequently considering the transactions invalid.

If any of these two steps end unsuccessfully the transaction is considered invalid and marked as so. Saving also invalid transactions enables to recognize quickly any identified entity that tries to do a Denial of Service attack. At the end of the validation process, each peer attaches the block into the locally stored ledger, changes the state by following the writeset, and updates the readset values. The whole procedure is illustrated in Figure 3.8.

[1]Peers can receive blocks from the ordering service or through **Gossip** messages received by the other participants of the network.

3.6 Endorsement Policy

3.6.1 Definition

Endorsement Policy sets constraints on the set of endorsers that are required to give an equivalent response to the invocation of a smart contract. Only if the EP is satisfied the transaction can be considered valid by the whole network. Endorsement Policies specify which organizations and how many peers inside of them must give their approval. These policies reflect the agreements taken between the organizations that co-operate inside the network and can not be chosen or changed by anyone that is not the system administrator. [33] [25] [20]

3.6.2 Endorsement Policies types

Hyperledger fabric makes possible defining Endorsement Policies at different levels of granularity. [4]

Chaincode Level

Every chaincode has a bounded endorsement policy which is applied to all the assets of the channel. These types of EPs are linked to the lifecycle of the chaincode and can not be changed unless a newer version of the chaincode is instantiated into the network.

Key level

Hyperledger Fabric lets also specify **key-level** endorsement policies referring to only an asset of the ledger. This type of policy is prioritized with respect to the chaincode EP. Key level EPs can be used in contexts where different EPs are needed for some assets of the ledger. For example critical assets may require more stringent EPs like the approval from every organization participating in the channel. Key-level EPs definition is stored in the write set of the transaction and can be changed through a transaction proposal.

3.6.3 Fabric EP language

Fabric offers a domain-specific language with which it is possible to define endorsement policies. This language consists of three logic operators:

- **AND** which requires every expression contained to be true
- **OR** which requires at least one expression contained to be true
- **Out-of-N** that is true only if at least k of the N conditions are satisfied

In this specific case, the expressions to be satisfied inside of the operators are the successful endorsement from an organization.

3.6.4 Real life example

Endorsement policies are expressive enough to be used for describing real-life scenarios. For example, whenever a person wants to sell his car to a buyer they must first agree on every detail of the agreement. The approval needed from both of them can be expressed with the Endorsement policy language:

$$AND('Seller','Buyer')$$

since this expression will return true only if both of the parties give their consent.

Chapter 4

JS application architecture

This chapter will introduce **Fabric SDK**'s architecture and **transaction flow**. Later it was studied how the **Fabric Node SDK** implementation faces the Endorsement phase.

4.1 Fabric SDK

Sometimes having a secure distributed ledger is not enough to fit companies' requirements. For instance companies might need a way to create more complex applications capable of interacting with the blockchain and that inherit its immutability and security properties. Fabric offers an **SDK** which enables to create applications that can communicate with a Fabric network. These SDKs are available for multiple programming languages such as **Java**, **Node.js**, and **Go**.

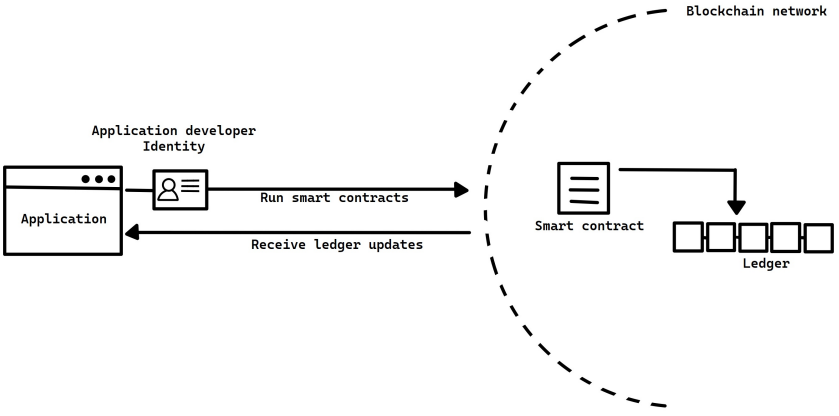


Figure 4.1: Application interaction with Fabric

4.2 Fabric Node SDK

Fabric SDK enables applications to interact with a Fabric network by offering several APIs that orchestrate the communication flow with the blockchain. Fabric SDK translates the invocation of its API into the execution of the corresponding smart contract's function by the peers of the network, as illustrated in Figure 4.1. Fabric SDK for Node.js offers several packages which can be used inside a node application: [10] [21]

- **fabric-ca-client** is an optional component for Fabric that enrolls peers and application users to enable them to be identified and access the network.
- **fabric-common** encapsulates the base code used by all the fabric-SDK-node packages. It provides low-level actions to the fabric network making it able to invoke transactions.
- **fabric-network** offers the API to connect to the fabric network. It gives the ability to query and modify the ledger content at a higher level than *fabric-common* does. Differently from fabric-common, it can not be used for installing, starting smart contracts or other administrative actions. [9]
- **fabric-protos** contains the necessary data structure to make gRPC communications possible. **gRPC** is a framework for reliable data communication between fabric network and client application.

4.3 Key concepts

In order to make an application communicate with a fabric network there are some classes belonging to the fabric-common package needed to be initialized.

4.3.1 Wallet

A **wallet** stores a set of user identities that provides access to a Fabric network. [22] These certificates together with the MSP associate to each client an identity and the role it assumes inside the network. As shown in Figure 4.2, identities are composed of:

- A **X.509 certificate**
- A descriptive **ID label**
- A **public key**
- A **private key**
- Fabric specific metadata

Wallets can store multiple certificates issued by different **Certification Authorities**.

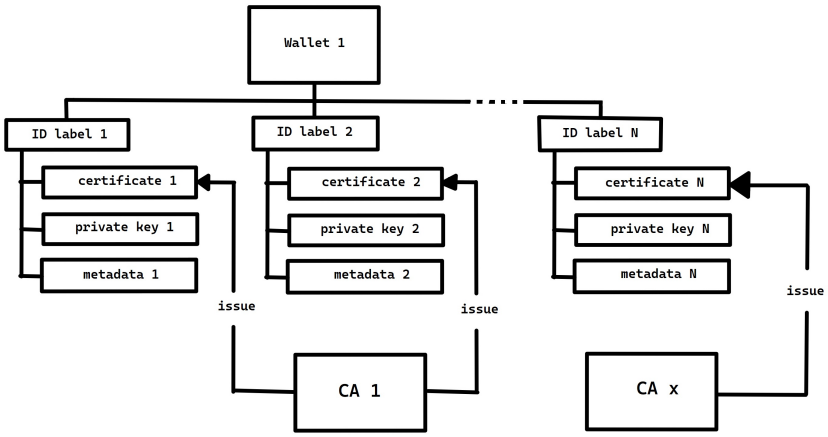


Figure 4.2: Wallet composition

4.3.2 Gateway

Gateways are used to manage the interaction with the Fabric network. [5] In order to work they need to be configured by inferring a partial or full description of the network topology. This configuration can be of two types:

- **Static** when the entire configuration is described in a **connection profile**. The SDK uses this statically encoded file to carry on the transaction proposals and the notification process. The inferred configuration must be complete in a way that the gateway can communicate with the network and have its transaction submitted. These network configurations need to be statically created by an administrator who understands the network's topology. Static configuration struggles to adapt to network changes that require the profile to be adjusted like when a new organization joins the network and the endorsement policy must be updated. The client may also have configured a profile with a peer that does not have the most updated version of the ledger and its transactions are a priori rejected because they do not respect the current value of the asset.

- **Dynamic** when the application uses the **Service Discovery** which automatically gathers data from the nodes and presents it to the SDK in a consumable way. [17] Service Discovery is installed inside the peers that by inspecting gossip communication layer messages indicates which nodes are online and how to contact them. Service Discovery can also be used to acknowledge the smart contract endorsement policy. The gateway still needs an entry point to be statically configured that can communicate and gather network information. Usually, it is inserted the profile of a peer that belongs to the same organization so that the data sent can be considered as trusted. A **connection option** configuration can be used to specify how the transaction process should be handled by the SDK.

4.3.3 Contract

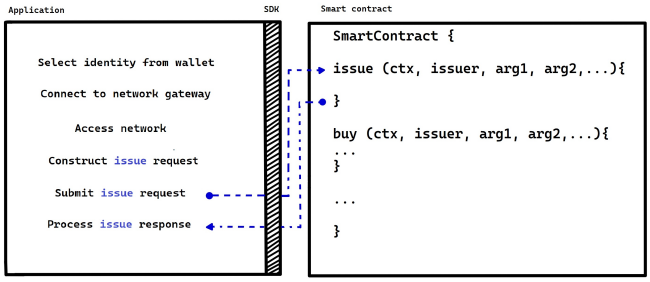


Figure 4.3: Fabric SDK interaction with EPs' smart contract

Contract class lets the application interact with a smart contract by offering two APIs [8]:

- **submitTransaction API** which enables to submit a transaction that aims to modify the ledger's content. Its parameters will be handled by the SDK in a way that the proper smart contract function will be called from the peers. The prototype of this API is:

  $$submitTransaction(name, [args])$$

  where

  - **name** is the name of the smart contract function that the client wants to execute. One example could be *insertAsset* or *deleteAsset*.
  - **[args]** are the argument of the smart contract and can be more than one. For example they can be the name of the asset and numeric values related to it.

  The function itself will return to the application the outcome of the transaction after the committing phase has finished.

- **evaluateTransaction API** does not send the received endorsements to the ordering service in a way that it does not change the content of the ledger. This API is used to query the world state of the blockchain. The prototype is equivalent to the submitting version:

  $$evaluateTransaction(name, [args])$$

Figure 4.3 shows how the API offered by the Contract class are later translated through the use of the SDK into chaincode function invocations.

4.4 Application transaction flow

This section summarizes how an application submits a transaction and how the SDK acts as an interpreter between the application and the network.

4.4.1 Pre-requisites

Before the application can run it is required that

- Peers have installed the chaincode
- The channel is already created
- The smart contract has its endorsement policy configured

In addition the application, before submitting transactions, needs to set up some objects offered by the Fabric SDK.

4.4.2 Fabric SDK initialization

When booting up the application needs to: [1] [23]

1. Have a **Wallet** that holds one or more X.509 certificates. Through them the MSP can identify the application and its right to read or modify the assets of the ledger. Only if the application user is correctly registered and enrolled by a CA of the organization, it can be authenticated into the network.
2. Define a **Gateway** to be able to communicate with the network.
3. Since the peers described in the gateway connection profile configuration can be part of multiple networks, the application has to select a specific **Network** with which it will interact.
4. From the network later the application can choose the selected smart contract through the **Contract** object.

4.4.3 Transaction execution

The user application can now communicate with the blockchain and issue a transaction. It is now described the interaction between the application and the blockchain when a new transaction is issued: [19]

1. The client retrieves the identity from a wallet and invokes the submitTransaction API which submits the request to the Gateway, as shown in steps 1 and 2 of Figure 4.4.

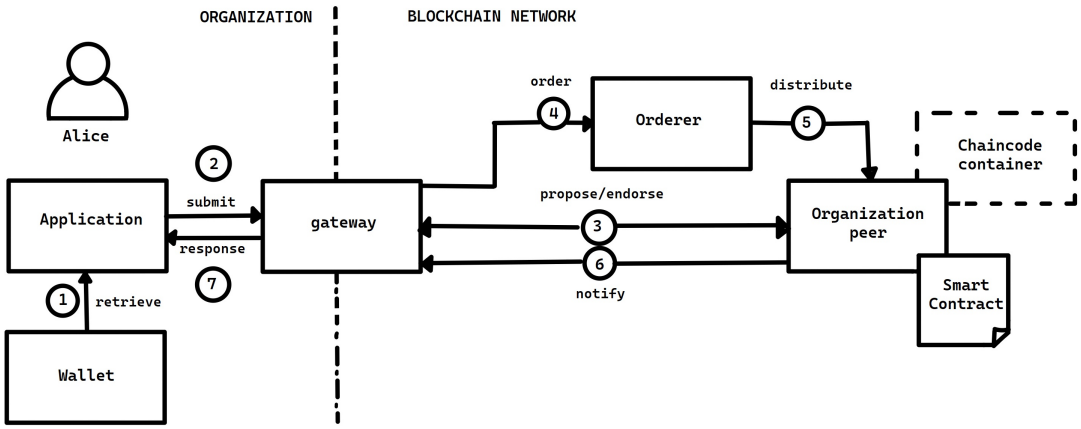


Figure 4.4: Application transaction flow

2. The Gateway acts as a shim that formats the inputs passed to the API, signs the endorsement request with the user's cryptographic credentials, and sends it to the right peers of the network. The endorsing peers check the correctness of the received request by inspecting its format if it has already been proposed (reply-attack), if the signature is valid and if the submitter has the right to modify or query the ledger. If everything is correct, the EPs execute the chaincode with the client's submitted inputs and return back to the Gateway the output containing the read-write set. This is shown in step 3 of Figure 4.4.

3. Having checked that the signature of the endorsing peers are legit, the SDK proceed to inspect if the output of the executions are equivalent. If the client wants its modification to be appended to the ledger, it must be sure to have enough endorsements to respect the endorsement policy. The Gateway creates a transaction message containing all the endorsements and broadcasts it to the ordering service that chronologically orders all the received transactions. This is represented by step 4 of Figure 4.4.

4. The ordering service delivers the assembled block to all the peers that verify if the endorsement policy is respected and if there is another attempt of modifying the ledger. If everything is correct the block gets added by the peers of the channel, as shown in step 5 of Figure 4.4.

5. When the execution is completed the Gateway is notified about the request finishing status. Finally, it parses the response payload in a comprehensible way for the application that is informed about the end of the transaction proposal. This is denoted by steps 6 and 7 of Figure 4.4.

4.5 Endorsers selection procedure

Having seen how the transaction flow proceeds, it is now discussed how the Fabric Node SDK implementation handles the endorsement phase and in particular how the endorsing peers are selected before sending the endorsement requests. The next sections will illustrate the data structure and functions involved in the endorsement phase carried by the SDK.

4.5.1 Layout

Layout is a data structure used by Fabric for representing a configuration of organizations that satisfy the endorsement policy. A **layout** associates to each specified group (organization) a number of peers belonging to it that must give the approval to satisfy the endorsement policy.

For example, if the layout of an endorsing procedure is:

$$[G0 : 1, G3 : 1, G4 : 2]$$

it means that 1 peer from G0, 1 peer from G3, and 2 peers from G4 must respond positively in order to endorse the transaction.

4.5.2 ProposalSendRequest

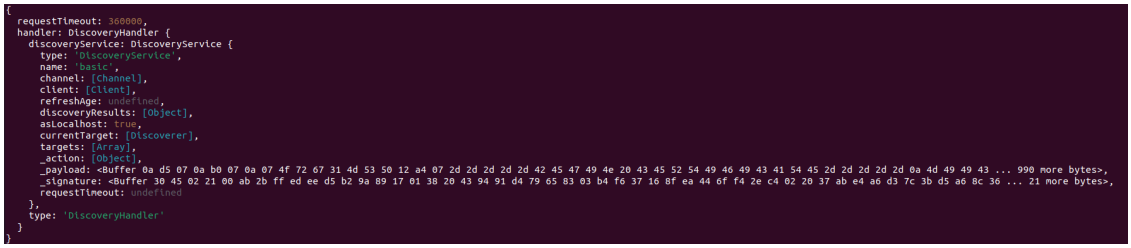


Figure 4.5: Example of a ProposalSendRequest object with a Dynamic configuration

ProposalSendRequest is an object that wraps the transaction request and packs details about how the endorsement phase has to be carried by the SDK. It can contain different attributes depending on the inferred configuration:

- **Request timeout** that indicates how much the SDK should wait for peers' responses before giving an error
- **Targets**, is an optional argument that statically identifies the target peers for the endorsement procedure
- **Discovery service Handler** is an optional argument and is typical of a dynamic configuration. Passing the handler of the discovery service is needed to make the SDK interact with it.

- **Peer selection criteria** is an optional argument where it is possible to define constraints on the peer endorsement selection process: some of them may be excluded, required, or preferred for certain characteristics.
- **Sort criteria**, that specifies if the endpoints should be sorted following certain criteria:
  - **BLOCK_HEIGHT** orders depending on the height of the ledger that a peer is storing. This can be a sign of reliability for the way blockchains are designed. This is the default value.
  - **RANDOM**, if the peer should be randomly ordered.

An example of an instance of a ProposalSendRequest object in the case of a Dynamic configuration is provided in Figure 4.5.

4.5.3 Endorsement plan

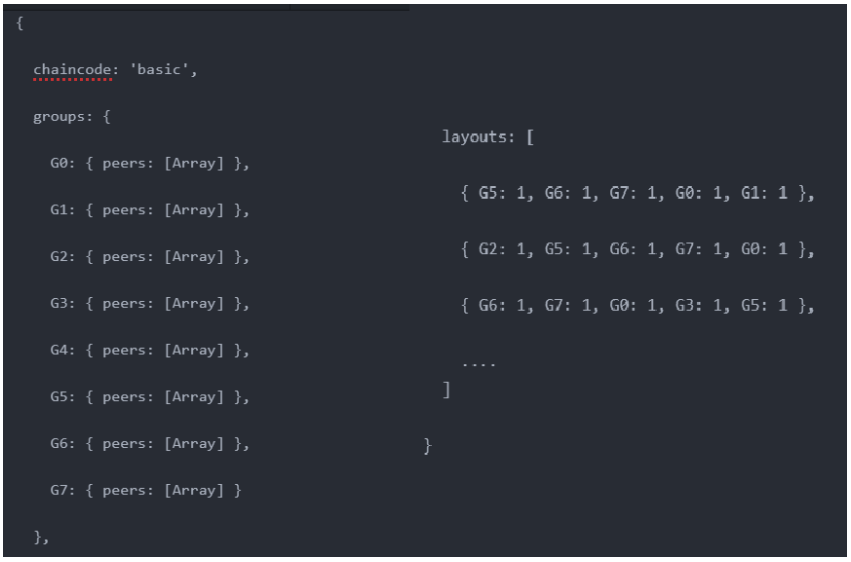


Figure 4.6: Endorsement plan structure

Endorsement plan is a key object as it maintains data about the network's topology and attributes that are used for sending endorsement requests to the right peers. An endorsement plan is composed of:

- Name of the **chaincode** with which the transaction wants to interact
- **Groups** which is the list of organizations beloging to the network. For each organization it identifies the contained peers specifying their:
  - MSPID

  - Endpoint URL
  - Name
  - Installed Chaincodes
  - Height of the ledger

- **Layouts** which contains a list of possible configurations of groups (organizations) that can satisfy the endorsement policy.

An endorsement plan can be retrieved by querying the service discovery which creates a response depending on the dynamically gathered data from the network. It can also be statically reconstructed by the SDK through the targets that are specified inside a proposal.

An example of an instance of a Endorsement plan object containing the chaincode name, groups and layouts is provided in Figure 4.6.

4.5.4 Endorsement phase

The actual endorsement phase starts when the **endorse** function is invoked. It is now illustrated the flow of the endorsement phase in the case in which a dynamic configuration is inferred.

1. Once called, the **endorse** function creates an endorsement plan by querying the Service Discovery. This is done by using the handler defined inside the ProposalSendRequest object passed as a parameter. For the way the Service Discovery's implementation works, a set of layouts composed of the disposition of groups that satisfy the endorsement policy are taken into account. Having correctly initialized the endorsement plan structure the _endorse function is called.

2. The **__endorse** function filters the endorsement plan's layouts in a way that the Peer selection criteria stored inside the ProposalSendRequest are respected. The resulting layouts are then shuffled and a loop through all the layouts begins trying to successfully endorse one of them.

3. **__endorse__layout** is called for one of the layouts. The function retrieves for each group(organization) the number of peers that are required to endorse the transaction. If the number of peers contained in the organization are less than the required ones the function returns an error and the flow goes back to step 3. Otherwise it calls **__build__endorse__group__member** for each required peer.

4. **__build__endorse__group__member** is called for every peer that has to endorse a transaction and it:

   - Checks if the peer has already responded to a previous endorsement request for the same transaction. This can happen because the same peer could have been in another layout that happened to fail. This makes possible speeding up the endorsement procedure not requiring an already endorsed peer to re-simulate the outcome of the transaction.

- Otherwise the SDK proceeds to connect to the peer through its endpoint URL and inspect if it is currently involved in other computations. If the peer is available the SDK connects to it and send finally the created proposal.
- If the outcome is successful the promise returns the response of the endorsement, otherwise the SDK retrieves the peer's characteristics and returns them with an error.

5. Having received the responses from every peer of the layout, the **_endorse** function checks if the endorsement requests went all successfully and if the responses received have equal output results. If an error occurred in one of the proposals the flow goes back to step 3. Otherwise, the function returns the endorsements and the endorsement phase is completed.

4.5.5 Layout dimension

Fabric Peer binary (version 1.4.9) makes the Service Discovery create and return layouts containing a fixed number of groups equal to:

$$NumGroupsInLayout = \frac{|NUM\_ORG|}{2} + 1 \tag{4.1}$$

which are composed in a way that they can satisfy the chaincode's endorsement policy.

For example, if there are 6 organizations inside the network, the layouts will be composed of 4 different groups.

Chapter 5

OPEN

In this chapter it will be discussed what is **OPEN** and how it works. Later it will be illustrated how OPEN can be implemented inside a Fabric Node Application. OPEN algorithm was designed by Doctor Lotfimahyari Iman in the "Optimal endorsement for network-wide distributed blockchains" paper.

5.1 Endorser peer selection algorithm

For instance, it is defined as **Endorsing peer selection algorithm** the procedure that selects while respecting the endorsing policy the endorsing peers to which the transaction proposal will be sent. In most of the cases it is possible to find multiple sets of peers that satisfy the endorsement policy and for this reason selecting one set rather than another could positively impact on the time experienced to finish the whole endorsement procedure. Choosing the best EPs is not trivial as their behavior together with the one of the network is not deterministic as it varies with the time and depends on many non-predictable factors.

5.1.1 Endorsing Time

Endorsing time can be defined as the time that the client waits from when it creates a transaction request to when it has received enough endorsements to be able to satisfy the endorsement policy.

The endorsing time experienced by the client is mainly influenced by two factors:

- **Network time** which is due to the fact that the endorsement request has to be sent to the EPs through the network. This delay is composed of the time needed for the actual propagation of the request from the sender to the receiver and by the additional time caused by the congestion of the network.

$$NetworkTime = PropagationTime + CongestionTime \tag{5.1}$$

- **Waiting time** which is the time required from when the EPs receive the endorsement request to when they simulate the transaction and provide the outcome of

the transaction. This delay is not fixed as it depends on many variables like the computing capabilities of the peers (that set a limit on the number of instructions that can be executed) or the received requests that are queued inside the peers (that shifts the execution of a newly received request).

$$Waitingtime = QueuingTime + ExecutionTime \tag{5.2}$$

For this reason **endorsing time** can be considered as sum the contribution:

$$Endorsingtime = Networktime + Waitingtime \tag{5.3}$$

given by the number of EPs that are requested to endorse a transaction.

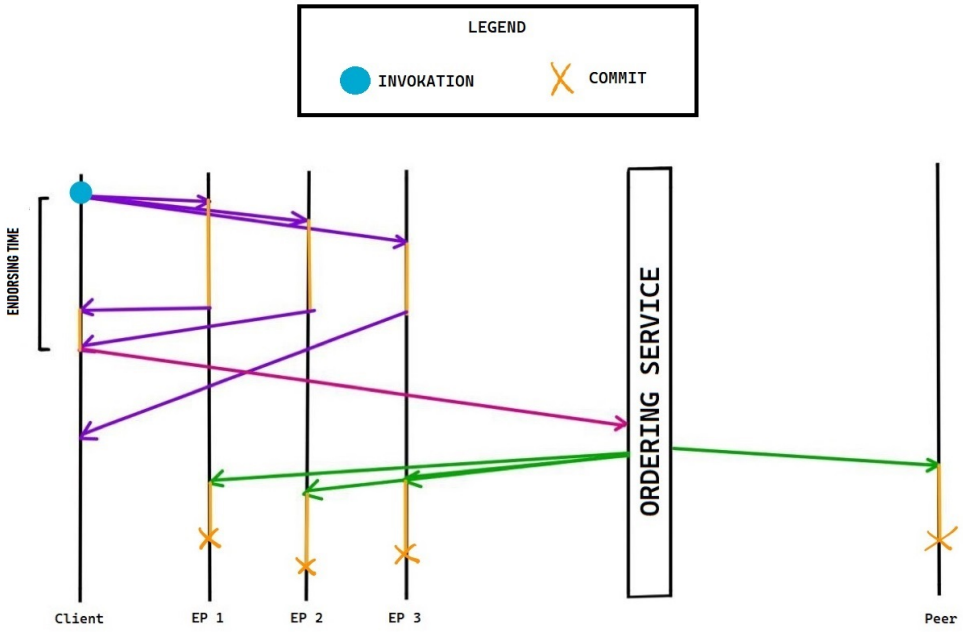


Figure 5.1: Endorsing time graph with a 2-out-of-N policy

Figure 5.1 illustrates how the endorsing time is measured in the scenario of a configured 2-out-of-N policy. In this case the endorsing time would be the time needed to receive at least 2 endorsements by taking into account the contributions given by the network time and the waiting time of the second fastest EP of the ones that were contacted.

5.2 OPEN

5.2.1 Definition

OPEN is a *state-based endorsing peer selection algorithm* that aims to minimize the *endorsing time* by selecting the EPs depending on how they behaved in the previous

transactions.

5.2.2 Functioning

OPEN exploits response delay data from previous transactions to decide which EPs are more suitable to be chosen. Because of the queuing that happens on the network and EPs, it can be experienced high correlation between the times that it takes to receive a response from an EP. This correlation can only be assumed only for EPs that were recently selected otherwise the information given by the response time should be considered obsolete. For instance, an EP that was not recently selected because highly loaded could be the one with the least number of pending requests when a new endorsement procedure is started since it was not selected anymore. To address this issue OPEN sends redundant endorsement requests to EPs that are not strictly required for satisfying the policy in order to gather data about their occupation status. [1] Furthermore, redundant requests are still considered in the evaluation of the endorsement policy. Finally, OPEN labels pending requests as a sign of the congestion of an EP that will not be considered in the decision process once another transaction is started.

5.2.3 Pseudo code OPEN

The pseudocode of OPEN is provided in Figure 5.2. Let TX_n be the n_{th} transaction locally for a client. Let $BEST\_N$ the number of selected EPs by OPEN. Let X_p^n be the measured response delays of TX_n for any EP $p \in P$. Let P_e^n be the set of selected EPs for TX_n. Finally let T be the sampling period which is measured in terms of transaction numbers.

```
1:  procedure OPEN(n)                                          ▷ Process TX^n
2:      e_p^n ← true, ∀p ∈ P                                   ▷ Init the eligibility vector for TX^(n)
3:      if n = 1 then                                          ▷ Just for the first transaction
4:          for p ∈ P do
5:              x_p^0 ← x_p^1 ← −1                             ▷ Init the response delay history
6:          P_e^1 ← P                                          ▷ Select all the available peers
7:      else                                                   ▷ Consider a generic transaction
8:          for p ∈ P do
9:              x_p^n ← −1                                     ▷ Init the measured delays for TX^(n)
10:         P_e^n ← Select-Endorsers()
11:         if (n mod T = 0) then                              ▷ Check if it is the time for a probe
12:             p ←Random-peer(P \ P_e)                        ▷ Select random peer ∉ P_e
13:             P_e^n ← P_e^n ∪ {p}                            ▷ Augment selected EPs with the probe
14:     Send-Endorsement-Requests(TX^n, P_e^n)
15:     X^n ←Update-Response-Delays()
```

Figure 5.2: Pseudocode of the `OPEN` algorithm for TX^n which was transcripted from "Optimal endorsement for network-wide distributed blockchains" paper

[1] Redundancy could reduce the response time and mitigate server-side variability as it samples more frequently nodes that may be temporarily slow because of garbage collection, background load or network interrupts. [32] [35] [34] [28]

1. In each transaction all the peers are firstly marked as eligible to be selected (line 2).
2. Just for the first transaction OPEN initialize the delay structure with dummy values and selects all the peers (line 3-6). Otherwise for every other transaction it just initializes the response delay with a nonsignificant value (lines 8-9).
3. It select the EP for the n_{TH} by following the *SelectEndorsers* procedure (line 10).
4. Every T transaction a random EP is added inside the set of endorsers(lines 11-13).
5. OPEN sends to the selected EPs the endorsement requests of TX_n (line 14) and stores the measured delays when a response from each EP is recorded (line 15).

SelectEndorsers function selects $BEST\_N$ EPs prioritizing the ones that responded faster during the TX_{n-1} transaction. In the case in which in TX_{n-1} one or more EPs did not respond, the selection becomes harder since the EPs that are considered congested are not selected by the OPEN algorithm.

```
1:  procedure SelectEndorsers( )
2:      d_max = max_{p∈P_e^{n-1}} {x_p^{n-1}}          ▷ Max measured delay for TX^{n-1}
3:      for p ∈ P_e^{n-1} do                            ▷ For EPs used for TX^{n-1}
4:          if x_p^{n-1} = −1 then                      ▷ Not yet response from EP p
5:              e_p^n ← false                           ▷ Make the EP Not-eligible for TX^n
6:              if d_max = −1 then                      ▷ No delay measured for TX^{n-1}
7:                  x_p^{n-1} ← x_p^{n-2}               ▷ Use past delays
8:              else
9:                  x_p^{n-1} ← d_max + ϵ               ▷ Speculate the delay
10:     for p ∈ P \ P_e^{n-1} do                        ▷ For EPs not used for TX^{n-1}
11:         x_p^{n-1} ← x_p^{n-2}                       ▷ Use past delays
12:     p ← Select-Eligibile-EPs-min-delay(BEST_N, X^{n-1})   ▷ Selects BEST_N EPs
13:     if |p| < BEST_N then   ▷ If the number of EPs selected is lower than BEST_N
14:         for i ∈ range(0, BEST_N − |p|) do           ▷ For every missing EP
15:             p ←Random-peer(P \ p)                   ▷ Select random peer ∉ p
16:     return temp
```

Figure 5.3: Pseudocode for `SelectEndorsers` which was partially transcripted from "Optimal endorsement for network-wide distributed blockchains" paper

The pseudocode of the procedure is provided in Figure 5.3.

1. At first *SelectEndorsers* calculates the maximum delay d_{MAX} that was measured during TX_{n-1} transaction. (line 2)
2. Every peer that was selected as EP during the previous transaction and that did not respond to the request is marked as not eligible. (lines 3-5)
3. From now on the procedure speculates the value of the delay of each EP for which it is not available:
    - If no responses were recorded during the previous transactions the algorithm speculates delays value by copying the values that were recorded during the

TX_{n-2} (line 6-7). In this case the other $|P| - BEST_N$ peers would be selected as none of the previous $BEST_N$ EP responded.

- The other case is the one in which at least a response was recorded during TX_{n-1} (line 8). For the previously selected EPs that did not respond the speculated delay is equal to the maximum delay recorded d_{MAX} summed to a constant ϵ that is small enough to be negligible with respect to the average network and processing delays. By doing this the algorithm assigns a value strictly larger than d_{MAX} for the EPs whose delay is unknown. Finally for all the other EPs that were not selected for TX_{n-1} their delays are speculated to be equal to X_{n-2} (lines 10-11).

4. $BEST\_N$ EPs are chosen among the eligible ones with the minimum measured or speculated delay, if available.

5. If not enough eligible EPs are available, OPEN randomly selects the remaining EPs (lines 13 - 15)

5.3 OPEN implementation into the application

5.3.1 Introduction

This section will show how OPEN policy can be implemented inside Fabric Node SDK architecture. **OptimalEndorsementPolicy** is a class that by following OPEN policy creates dynamically layouts containing the EPs to which ask to endorse a transaction.

5.3.2 Attributes

OptimalEndorsementModule attributes are:

- **peers** attribute keeps memory about the EP selected for the endorsement phase. It is a matrix having 2 rows one representing the EP for TX_{n-1} and the other one the EPs for TX_{n-2}. There is a column for each peer of the network and the value of each cell represents if that specific peer was selected for that transaction. Cell value is -1 if that EP was not selected. Otherwise, the value can range from 0 to $|P|$ and it represents the priority with which that EP was contacted (0 means max priority). Keeping memory of the past peer's selection is required as it is used as a selection criteria by the OPEN process.

- **delays** stores the measured/speculated delays for every peer and it has the same dimension of peers' data structure. This matrix will be used to decide which peers responded faster and consequently who will be the one to endorse.

- **eligibility** vector indicates if a peer is eligible for being selected as an endorser. The value depends on how it behaved in the previous transaction: if it did not respond its value will be False as it means that it should not be contacted for the next transaction because probably occupied by other computations.

5.3.3 Methods

OptimalEndorsementModule most relevant methods are:

- **GetLayout()** is a function offered from the module to retrieve the layout containing the peers to contact depending on the selection criteria. In order to gather data about the response delays, this function for the first **INITIAL_DELAY_ACQUIRING** steps sends the endorsement requests to all the peers. After these initializing steps, GetLayout() will compute the layout following the OPEN policy (or another, if specified).
- **SetEndorsementDelays(delay_data)** is a function that prepares and fills the delays structure with the response delays experienced in the current transactions. These values will be later used to decide which EPs will be asked to endorse.

5.3.4 Modifications of the SDK

In order to integrate OptimalEndorsementModule inside SDK architecture, there are three main parts to be modified:

- **Generation of the endorsement plan** that, as shown on section 4.5.4, happens during the **endorse** function. A statically created endorsement plan is used as it enables to have a fixed mapping between the groups and the organizations that otherwise would not be respected since the Service Discovery reconstructs the topology of the network through the gossip message system every time it is asked. Having a fixed mapping is necessary to associate an EP with its history of delays.
- **The generation of the layout** as it that will not be retrieved anymore from the response of the Service Discovery. A layout will be created by invoking **GetLayout()** each time the endorse_layout function is called. With OptimalEndorsementModule the layout is created each time endorse_layout function is invoked depending on how it behaved during the previous invocation. As a matter of fact, if in the previous layout a specific peer did not respond, it will not be considered as an endorser for the new transaction. This does not happen with the standard implementation as all the layouts are statically created and then iterated until one of them succeeds.
- **The registration of the delays**, once the requests are sent to the peers selected by the OptimalEndorsementModule, for each of them the time needed for the peers to respond is registered. Once the invocation of endorse_layout has finished, the value of the delays are stored inside OptimalEndorsementModule through the use of the **SetEndorsementDelays** function which facilitates the interaction with the module.

5.3.5 Additional features

- OptimalEndorsementPolicy supports the modular implementation of multiple Endorsers peer selection algorithms like RND_2.

- It implements the **Singleton pattern** in a way that only one instance of the class can be defined, reducing the possibility of conflicting code.

Chapter 6

Experimental validation

In this chapter it will be described the experimental scenario, methodologies and results of the experiments made to evaluate OPEN performances.

6.1 Experimental scenario

The **machine** that was used is a laptop with this configuration:

- **Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz** 4-Core Processor, 8 Threads
- **8 Gb DDR4 Ram**
- GTX 1050ti GPU
- Host OS: Windows 11
- SSD 64GB

All the experiments were run inside a **virtualized operating system** through the use of Oracle VM VirtualBox (6.1.16 r140961 (Qt5.6.2)):

- Ubuntu 21.04 64bit
- 6 Threads
- **5 Gb Ram**
- 150 Mb of GPU

The **fabric network** that was used for these experiments was configured with:

- Hyperledger Fabric version 2.2.2
- Fabric CA version 1.4.9
- 8 organizations, each one containing a peer
- 1 orderer

- 8 CA, one for each organization
- 1-outOf-N endorsement policy
- Max CPU usable for a single peer = 10%

The **libraries** required for booting up Fabric network are:

- Docker version 20.10.7, build 20.10.7-0ubuntu5 21.04.2
- docker-compose version 1.29.2
- git version 2.30.2
- curl 7.83.1 (x86_64-pc-linux-gnu)

The **programming languages** used for developing the application and the programs for conducting the experiments are:

- Javascript Node.js v12.21.0
- Python 3.9
- Bash

In order to function, it was necessary to configure properly **Fabric SDK** in a way that the application could wait for the EPs' response before exiting giving a timeout. As a matter of fact it was set:

- Request timeout = 36000 ms

6.1.1 App limitations

There are two ways of issuing transactions to the Fabric network: through the CLI (by using a bash command) and by using an application (in this case a Node.js one). These two methods differ in the way they behave:

- **CLI**: thanks to the **peer chaincode** command the administrators can perform operations on the chaincode like installing it, querying it, or submitting transactions to it. CLI is fast as it communicates directly through a so-called issuer peer which takes care of doing the request action. By using the CLI the client can only select the endorsing peers and wait for the response of the execution. As a matter of fact, since the entire communication with Fabric is handled by the issuer peer, it is impossible to distinguish the delays of each endorsing peer. *Consequently it is not possible to implement OPEN policy by using the CLI method.*
- **Application**: as previously discussed the application exploits an SDK that handles the interaction with the Fabric network. With the application, modifying the SDK code, it is possible to map every endorsing response to the associated EP and this enables to implement custom endorsers selection algorithms like OPEN. In contrast, the overhead given by the use of the application (and the use of the CAs) and by the inefficiency of its code does not let reach a high rate of transactions. The application can only issue a transaction every 4 seconds (throughput 0.25).

6.1.2 Experiments

The next sections will be divided in two main parts:

1. The first one called **Load of the system** will inspect endorsing peers' behavior when they're put under stress. Different analysis will be provided in order to better understand if EPs can be brought in a status where they struggle with the execution of the endorsement requests. This study is necessary as it validates OPEN's assumptions about the queuing that happens in the network and in the EPs. As a matter of fact, because of the experienced queuing, some EPs may be slower in returning the endorsement back to the client making the endorsing time grow. For this reason if queueing is proven a state-based selection can benefit of the data gathered from recent transactions. Otherwise, if queuing can not be proven, selecting EPs depending on how they responded previously would be worthless as correlation between the times of consecutive requests can not be assumed.

2. The second part instead is called **OPEN performance evaluation** and will inspect how OPEN algorithm performs with respect to other endorsing selection algorithms.

6.2 Methodology - Load of the system

A focus on understanding how EPs behave in different stress scenarios is provided. CLI method will be used to be able to issue a high rate of transactions making the EPs struggle with executing endorsement requests.

6.2.1 EP's log

It is defined as **Waiting Time** the time that it takes to a peer from when it receives a transaction to when it completes the endorsement procedure. This time includes both the time that an endorsement request has to wait before being executed, which for instance is called **Queueing time** and the time for executing the transactions, which for instance is called **Execution time**.

$$WaitingTime = QueueingTime + ExecutionTime \tag{6.1}$$

This value can be accessed by each of the peers by issuing the command:

```
docker logs peer{NUM_PEER}.org{NUM_ORG}.example.com
```

which returns the online log of the peer where several useful information is written. This is an example of the returned log:

```
2022-06-09 20:13:34.595 UTC [endorser] callChaincode -> INFO bfd finished
chaincode: basic duration: 2ms channel=mychannel txID=dddc8023
```

```
2022-06-09 20:13:34.596 UTC [comm.grpc.server] 1 -> INFO bfe unary call
completed grpc.service=protos.Endorser grpc.method=ProcessProposal
grpc.peer_address=172.18.0.1:60644 grpc.code=OK grpc.call_duration=4.414606ms

2022-06-09 20:13:34.785 UTC [gossip.privdata] StoreBlock -> INFO bff Received
block [180] from buffer channel=mychannel

2022-06-09 20:13:34.794 UTC [committer.txvalidator] Validate -> INFO c00
[mychannel] Validated block [180] in 8ms
```

and each line is meaningful as it affirms that a specific event has been completed from that EP.

1. The first line indicates that the execution of the chaincode associated with a tx request has been completed. The txID is specified and in this case is dddc8023.

2. The second line instead tells that the entire endorsing procedure has been completed from the EP. From the **gprc.call_duration** it is possible to understand how much it took for that EP from when it received the request to when it completed the procedure. This is what was defined as **Waiting Time** previously and it gives a hint about EP's occupation. In this case, the Waiting Time equals 4.4146606ms. Here the txID is not specified and this does not let acknowledge the endorsing time since it is not possible to link the Waiting Time from the different peers to the transaction they belong.

3. The third line indicates that a new block has been received from the EP. Each block is associated with an ID (180 in this specific example).

4. The last line indicates that the received block has been validated and its content is immutably inserted into the locally stored blockchain of the EP.

6.2.2 Generator

The transactions are generated by a python program that for instance will be called **Generator**. The generator sends transactions following a **periodic** distribution which depends on the value of the desired rate R. Once started, the generator will issue a transaction to Fabric each 1/R second. The transactions are issued by creating a thread that will use the CLI commands to insert a unique asset inside the blockchain. Figure 6.1 shows how the generator interacts with the Fabric network by creating multiple threads that issue each one a transaction.

6.2.3 Experiment 1 - Metrics analysis

Introduction

For instance, it is necessary to define some metrics that will be used to evaluate both the Generator and the Fabric network. Let t_{Gi} the time in which the Generator asks for the

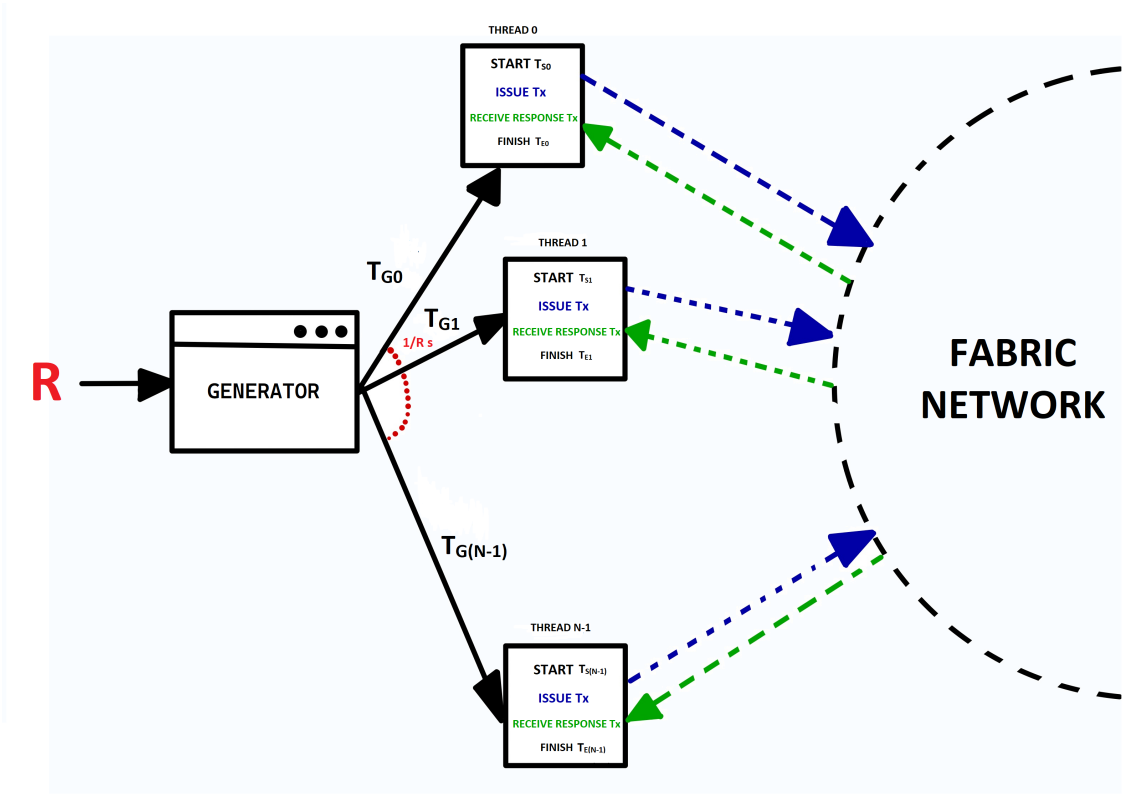


Figure 6.1: Generator logic

creation of i_{TH} thread. Let t_{Si} the time in which the i_{TH} thread starts running. Finally, let t_{Ei} the time in which the i_{TH} thread has correctly finished its task.

For instance:

- **Set Rate [tx/s]**, is the value of the desired transaction rate given to the generator as input. For obvious machine limitations it is possible to issue a limited number of concurrent transactions.
- **Offered Load [tx/s]** is the actual number of concurrent transactions per second that are issued by the generator. The offered load is calculated as

$$OL = \frac{N}{t_{G(N-1)} - t_{G0}} \tag{6.2}$$

and depends from the time needed for the Generator to spawn N threads.

Python threads are not executed in parallel but there is a single process that switches the execution between simpler tasks implementing the so-called **Concurrency**. A decrease of the offered load value means that between two spawning of a thread, a period longer than 1/R seconds passes. This happens when the tasks belonging to the previously generated threads are so many to unable the generator to create other threads.

- **CPU occupation** [%] indicates how much CPU is utilized by the system and its value ideally could be seen as the sum of two components:
  - The **transaction Generator** as higher values of offered load inevitably bring higher value of CPU occupation. More threads are generated per second and consequently the number of tasks to execute.
  - The **fabric network** since the occupation grows also if there is a higher number of EPs asked to endorse since they all must execute the requests and ask for CPU resources.
- **Throughput rate [tx/s]** is the rate at which Fabric network services transactions. The throughput rate is calculated as

$$TH = \frac{N}{t_{D(N-1)} - t_{S0}} \tag{6.3}$$

  and depends on how much it takes for Fabric to finish the execution of the submitted endorsement transactions. Throughput measure enhances how much the generator is able to put at stress the Fabric network. Lower throughput means fewer requests serviced per second by Fabric with respect to the ones that were submitted.
- **Average Waiting Time [ms]** is the average time in which the peers of the network respond to an endorsing request and it indicates the occupation of the network. The AWT is calculated as:

$$AWT = \sum_{p=0}^{P} \frac{\sum_{i=0}^{N} WaitingTime(i)}{N} \tag{6.4}$$

  AWT analysis makes it possible to do assumptions about the behavior of the EPs during the endorsement phase rather than during the whole transaction. For example, if for an offered load equal to 18tx/s the RND_2 algorithm has an AWT of 50ms it means that if this rate of transactions is sustained, the randomly selected EPs will endorse the transaction in averagely 50ms.

Experiment settings

This experiment will analyze the above-mentioned metrics in order to understand how both the Generator and Fabric behave if a growing number of transactions per second is issued. The experiments will be done by using four different endorsing peer selection algorithms:

- **ALL_PEERS**, where the request is sent to all the peers of the network
- **RND_4**, where the request is sent to 4 randomly selected peers
- **GEOMETRIC**, where the request is sent to the first k peers (starting from peer0.org1 to peer0.org8) with k_i is calculated by using a truncated Geometric

distribution with average value of N=NUM_PEERS/2. The value that can be returned by this geometric ranges from 1 to 8. Its average value equals to 4 and the distribution is represented as:

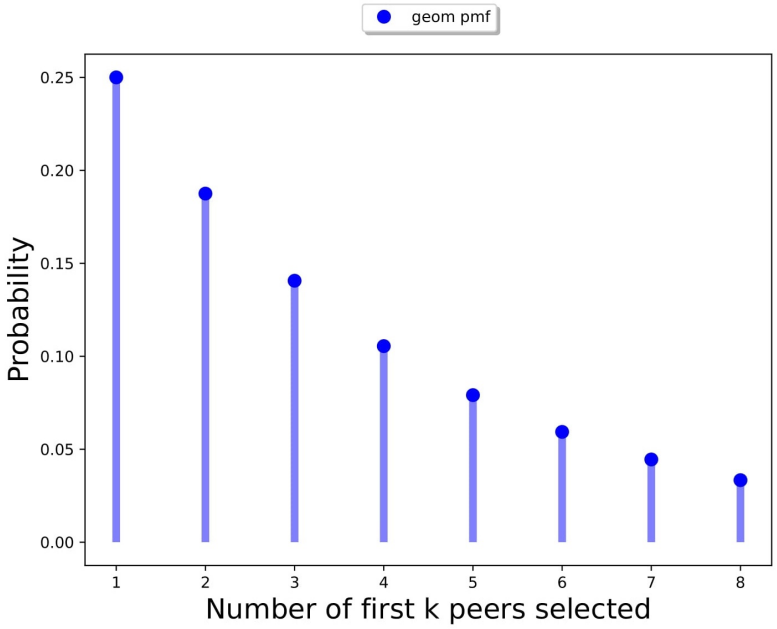


Figure 6.2: Truncated Geometric distribution

Figure 6.2 shows how probabilities are distributed in the truncated Geometric.

- RND_1, where the request is sent to 1 randomly selected peer

The quantity of CPU that each EP can access is limited by a so-called **resource usage limiter** to be 10%. The behavior of these utilities is not ideal as it can happen that a process can in some moments access more resources than the assigned ones. Anyway, it is possible to affirm that the **cpulimit command** sets boundaries on the Fabric peers in a way that they struggle with accessing the resources and "experience" the scenario in which a big amount of tasks should be executed by the peers but there are not enough resources to do it immediately. Otherwise, it would not be possible doing this experiment from the moment in which in an environment where no limitations are set, the EPs would saturate immediately the resources not making the Generator able to sustain a high rate of transactions.

Procedure

The pseudocode of the procedure of Metric Analysis is shown in Figure 6.3: for each **endorsing peer selection algorithm**, the network was (re)booted, and three measurements were done by issuing for 100 seconds an **increasing value of set rate R** between the range from 1tx/s to 20tx/s. For every value of set rate the Generator measured the

```
1: procedure METRIC ANALYSIS(n)
2:     for algorithm ∈ EndorsingPeerSelectionAlgorithms do      ▷ For every endorsing selection algorithm
3:         boot_network()                                        ▷ (Re)boot the network
4:         for repetition ∈ range(0,3) do                        ▷ Repeat 3 times
5:             for R ∈ range(1,20) do                            ▷ For every value of set rate between 1 and 20
6:                 measure_metrics(R, seconds = 100, algorithm)  ▷ Measure the metrics
7:                 sleep(60)                                     ▷ Sleep for 60s between every value of set rate
8:             sleep(200)                                        ▷ Sleep for 200s between every repetition
```

Figure 6.3: Pseudocode of the `Metric Analysis` procedure

metrics following the procedures mentioned previously. Having finished the i_{TH} value of set Rate R, the generator cooled down by waiting for **60 seconds** in a way that there was no influence between the experiments.

The shown values were calculated by averaging the results of the three measurements and by calculating the error through a **Student function** with 2 degrees of freedom and with a **confidence level of 90**%.

6.2.4 Experiment 2 - Execution Time distribution

Introduction

Execution time is the time needed for an EP to purely execute an endorsement request. Since the EPs give only indication about the waiting time, which contains both the execution time and the queueing time, it is needed a way to minimize the second component. To minimize the time that a request has to wait before being executed, a low rate of transactions can be sent to a single EP making it possible to do assumptions about the execution time behavior. Together with the average value of the measurements it will be computed also the so-called **coefficient of variation** C_v which is defined as:

$$C_V = \frac{\sigma}{\mu} \tag{6.5}$$

where μ and σ are respectively the mean and the variance of a set of measurements. The coefficient of variation provides a measure of the sparsity of the measured points. If C_v is near or equal to zero, it means that the execution time can be considered as deterministic. Otherwise Execution time should not be considered as fixed.

Experiment Setting

To understand how EP's execution times are distributed, it was set an experiment where 1000 transactions were issued to a single EP with a rate of 0.25tx/s.

6.2.5 Experiment 3 - Load of the last k peers

Introduction

The third experiment focuses on the waiting time experienced by every single EP that belongs to the network. The goal is to differentiate the rate of transactions issued to each EP in a way that can be observed. If the AWT is affected by the intensive use of the

CPU rather than the higher transaction rate. If the EPs that receive a higher rate of transactions require more time to respond, it could mean that there is correlation between the times that it takes to complete consecutive endorsement requests. Otherwise, if the AWT does not reflect the rate values, it means that the time of response is independent from the number of transactions submitted per second, and consequently queueing can not be assumed.

Uniform

To achieve this type of discrepancy in the transaction rates, it is defined a probability distribution called **Uniform** where the probability of selecting a value of **k** is uniformly distributed among the range k=[1,8], as shown in Figure 6.4. This range is defined accordingly to the EPs' enumeration it is been used until now.

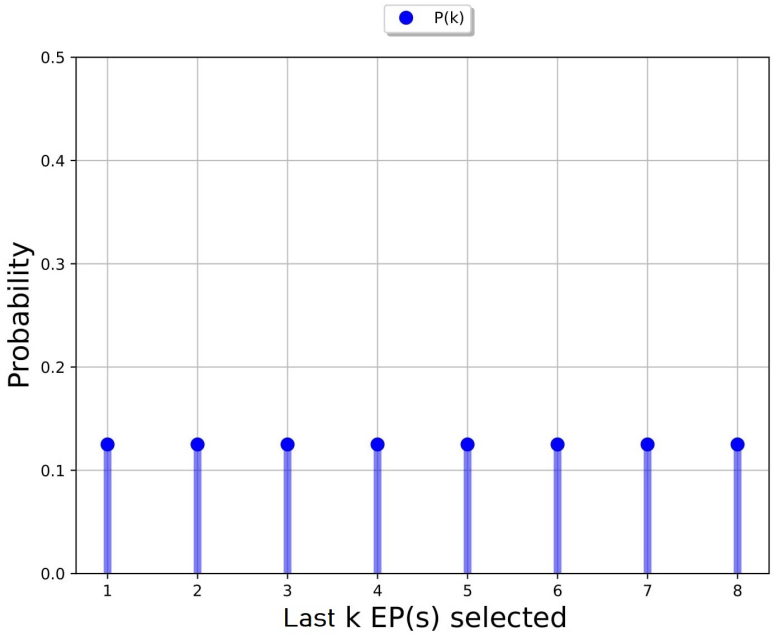


Figure 6.4: Uniform distribution

The output k returned by the Uniform distribution represents the last k EPs that will be asked to endorse the transaction. For instance, the order of the EPs is defined as:

$$[EP1, EP2, EP3, EP4, EP5, EP6, EP7, EP8] \tag{6.6}$$

where EP1 is the first one and EP8 is the last one. For example, if the output k of the Uniform distribution equals 4, the EPs that will be asked to endorse the transaction will be EP8, EP7, EP6, and EP5. For the way this distribution is defined, EP8 will be asked to endorse every transaction and consequently should be the most loaded among the network. EP1 instead will be selected only in the case in which the k equals 8. Uniform distribution should make it possible to redistribute the load among all the EPs of the

network in a way that a growing number of endorsed transactions are observed starting from EP1 and finishing with EP8.

Mathematical queue model

The Average Waiting time can be estimated for each EP through the use of a mathematical queue model. These types of models associate to the number of transactions that a server receives the expected time that a request has to wait to be serviced. To choose the correct queue mathematical model with which it can be theorized EP's behavior, it must be analyzed the characteristic of the system: the discriminators for deciding which model to use will be the transactions inter-arrival, the EP's execution time distribution, and the number of servers. [24]

- The **transaction inter-arrival distribution** is Poisson distributed since from EP's point of view transactions are arriving randomly with an average value equal to their arrival rate.
- The **number of servers** equals one as it is studied the AWT from each EP point of view.
- The **execution time**, as observed in 6.5.2, can not be considered deterministic since its coefficient of variation is high. This requires the use of a model that takes its variability into account.

The **Pollaczek-Kinchin formula for M/G/1 queue model** [29] can be used to model systems in which the execution times has high variability. This formula associate to an arrival rate λ an expected value of the average waiting time which is calculated as:

$$WT = (1 + \frac{1 + C_v^2}{2} * \frac{\rho}{1 - \rho}) * \frac{1}{\mu} \tag{6.7}$$

where

- μ is calculated as the inverse of the execution time when the system is not stressed by a high rate of transactions
- C_v is the coefficient of variation
- λ is the arrival rate and it calculated for each EP as

$$ArrivalRate = \frac{numberOfRequests}{experimentDuration} \tag{6.8}$$

- ρ is the utilization factor, which is defined for each EP as

$$UtilizationFactor = \frac{\lambda}{\mu} \tag{6.9}$$

Experiment settings

The experiment was made by issuing transactions with a rate of **10 tx/s** for **60 minutes** following a Uniform distribution. For each EP it will be analyzed the value of the Arrival Rate λ and the Average Waiting Time acknowledging if there is any relationship between these two measures. Later it will be theorized the measured sequence of Average Waiting Times through the use of the just described **mathematical queue model**. For each EP it will be calculated the expected value of average waiting time and it will be confronted with the measured one to see if it is possible to observe a similar behavior.

6.3 Methodology - OPEN performances

6.3.1 Introduction

This section compares OPEN algorithm **average endorsing time** with respect to the original Fabric Endorsing Peer Selection algorithm. In order to create the correct environment in which the algorithms can be evaluated, it is necessary to create load inside the EPs. As a matter of fact if no load is created, the EPs will not be stressed enough to queue requests and will not vary significantly the time they need to respond to an endorsement request making it useless to select one EP rather than another. As already mentioned Fabric Node Application is not able to sustain high rates of transaction making it unfeasible to overload EPs through its use. For this reason it is deemed necessary using an **hybrid approach** where both the application and the Generator cooperate: this makes it possible creating load inside the network while benchmarking OPEN.

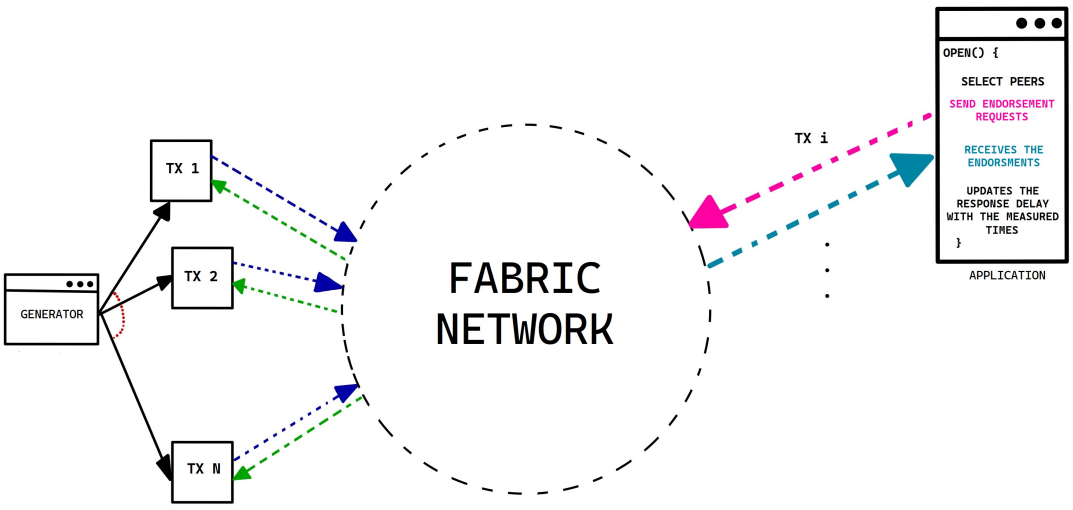


Figure 6.5: Second experiment

The schema of the experiment is represented by Figure 6.5: while the Generator creates a load of transactions, the application selects the endorsers for its requests by

following OPEN policy.

The Generator is used in multiple configurations to study how OPEN works in **different stress scenarios**. The load is created through the previously introduced **Uniform**. Since both the application and the Generator are run, the rates of transaction achievable by the Generator are lower than the ones reached in the previous experiments. By using Uniform the maximum rate achievable by the system is **7 tx/s**. Otherwise if a higher transaction rate is set, the application will not boot.

6.3.2 Experiment setting

For each scenario **5 experiments** were conducted in which the application **added 1000 assets** following the configured algorithm. The inspected metric is the average endorsing time in the case of an endorsement policy *1-out-Of-8*. Also the average time of response will be indicated and it consists in the average time in which each EP of the layout responded. The values will be indicated with their associated value of uncertainty that was calculated through the use of a Student function with a **confidence level of 90%**.

OPEN configuration

OPEN algorithm was used with this configuration:

- **Probing factor T** = 10
- **eps** = 0.001 ms
- $BEST_N = 5$

INITIAL_DELAY_ACQUIRING instead was set to 2.

Scenarios

Different stress scenarios will be analyzed:

- **High rate of transactions** created by the Generator (7tx/s).
- **Low rate of transactions** created by the Generator (3tx/s).
- **No rate of transactions** created by the Generator (0 tx/s).

OPEN performance evaluation

For each scenario it will be analyzed how OPEN performed with respect to the ORIGINAL algorithm computing:

$$Gain = \frac{avgEndTime(ORIGINAL)}{avgEndTime(OPEN)} \tag{6.10}$$

which indicates how OPEN performed with respect to the ORIGINAL algorithm. If its value is higher than 1 it means that OPEN outperformed ORIGINAL algorithm. Otherwise if its value is lower than 1, ORIGINAL was the best.

Application's EP selection analysis

A further analysis will be provided about the EPs that were selected by the application during the experiments. This enables making assumptions about OPEN's behavior when choosing the EPs. In the scenarios where the Generator was used, it is also provided the measurements of the average waiting time for each EP. This enables doing a global evaluation of the occupation status of the network during the experiments.

6.4 Results - Load of the system

6.4.1 Experiment 1 - Metric analysis

Offered Load Analysis

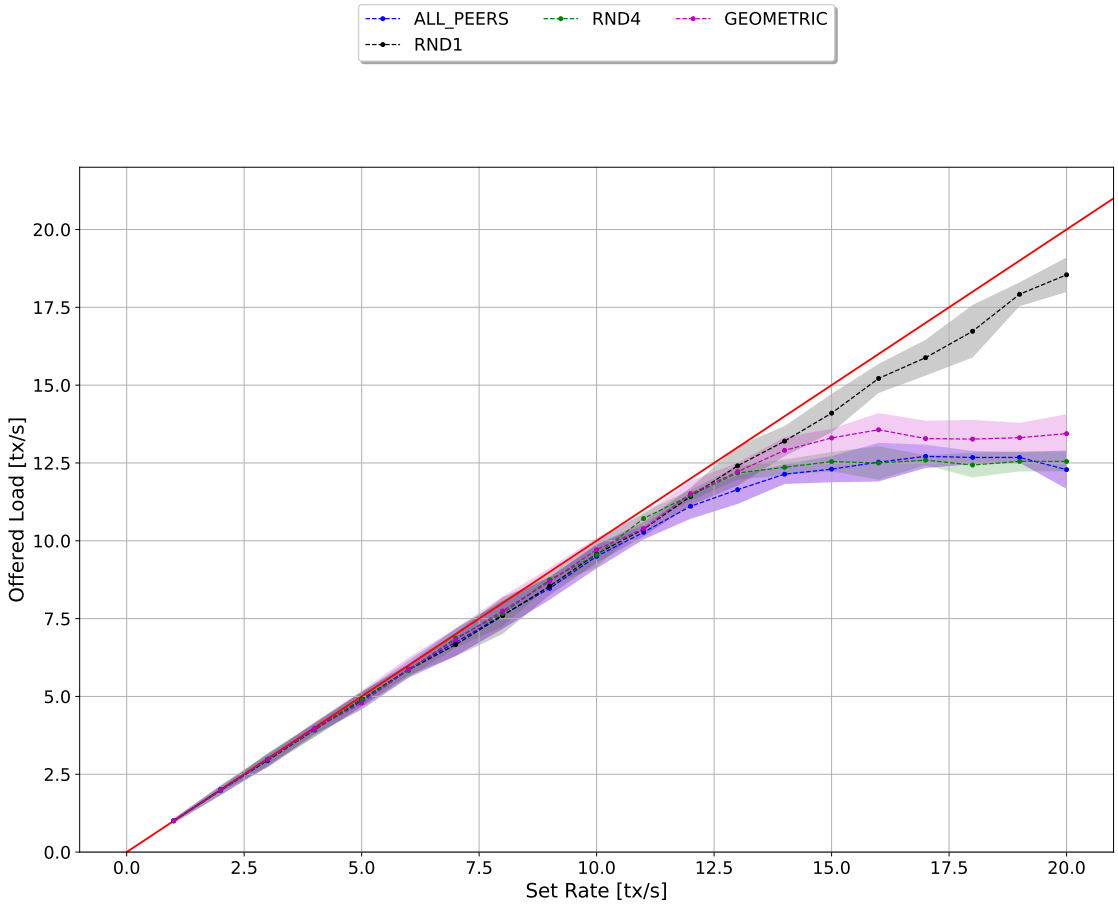


Figure 6.6: Offered Load graph with respsect to the Set Rate

Figure 6.6 illustrates the values of Offered Load that the Generator assumed in all the inspected algorithms depending on the Set Rate.Once the Set rate becomes higher the way the different algorithms operate changes: RND_1 algorithm is the only one of

the studied that keeps growing in the inspected range of rates. For the other algorithms, the value of set load for which the offered load has a **decrease of 10%** are:

ALL	RND4	GEOMETRIC
13	14	15

These values give an indication of how many concurrent transactions the Generator can issue, depending on the algorithm.

CPU occupation

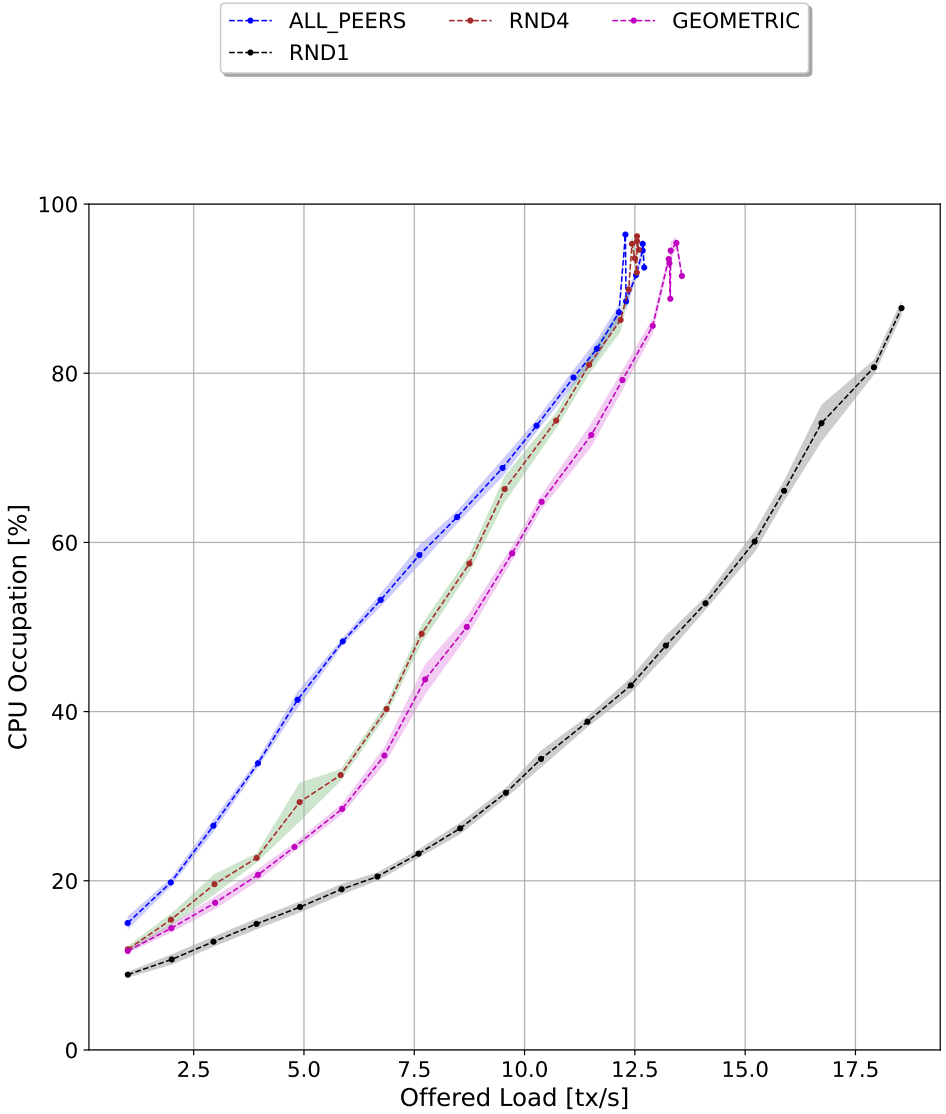


Figure 6.7: CPU Occupation[%] with respect to the Offered Load

Figure 6.7 shows the values of CPU occupation that the system reached in all the inspected algorithms depending on the Offered Load. It is clear that once more EPs are asked to endorse a transaction, the way the occupation value changes is different. It can be observed how in **Geometric**, **RND_4** and **ALL_PEERS** algorithms the occupation level peaks when certain values of offered load occur. These peeks happens concurrently with the saturation of the offered load. This means that the resources were not enough for both Fabric and the generator that competed for accessing them resulting in uncertain results. **RND1** represents the upper limit of the contribution given by the Generator as it is the algorithm that requires just one random peer to endorse and so it minimizes the Fabric network occupation component.

Throughput Rate

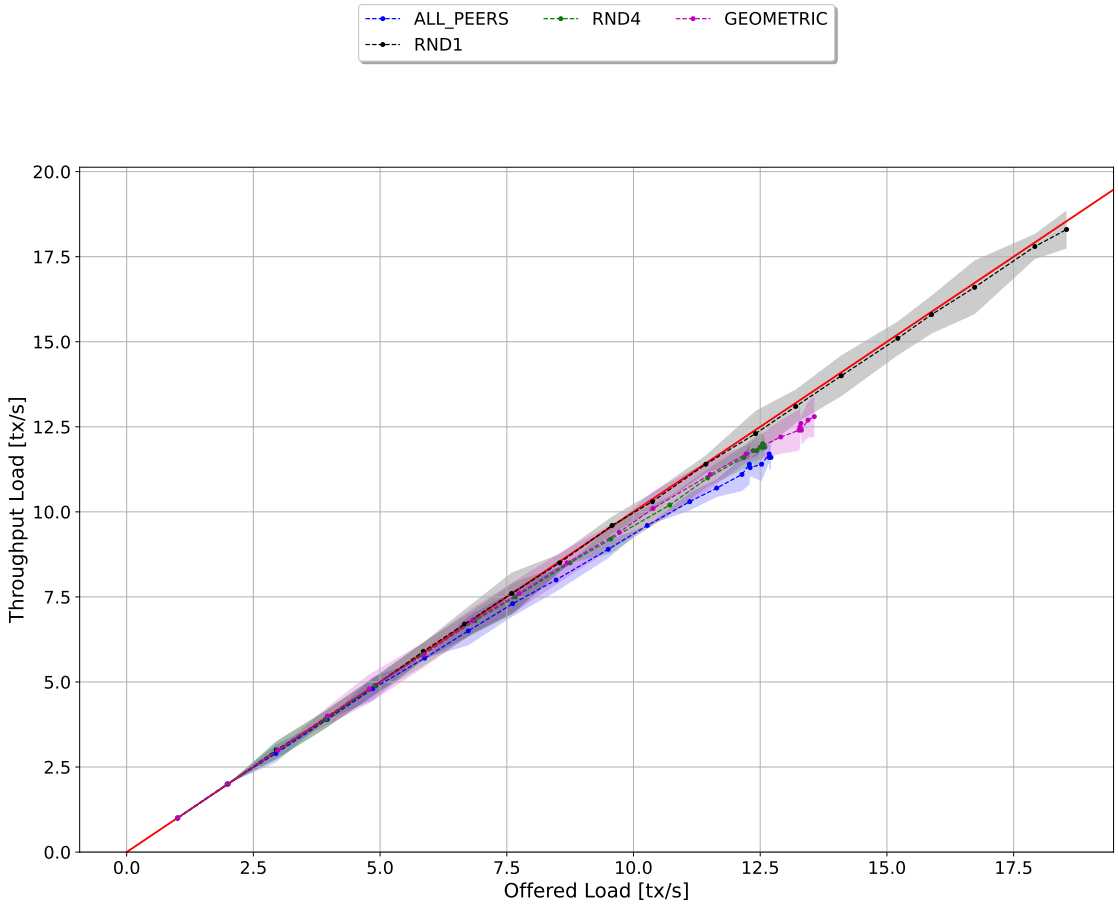


Figure 6.8: Throughput rate with respect to the Offered Load

Figure 6.8 shows the values of throughput that Fabric reached in all the inspected algorithms depending on the Offered Load. Every curve is always under the value of offered

load and this validates the results as it would be incorrect if Fabric solved more transactions than the ones that were actually submitted. The curve representing ALL_PEERS has the lowest slope as each request took more time to be finished since its delay of execution depends from the slowest EP.

The value of offered load for which the throughput has a **decrease of 5%** are:

ALL	RND4	GEOMETRIC
5.88	9.55	9.71

Average Waiting Time

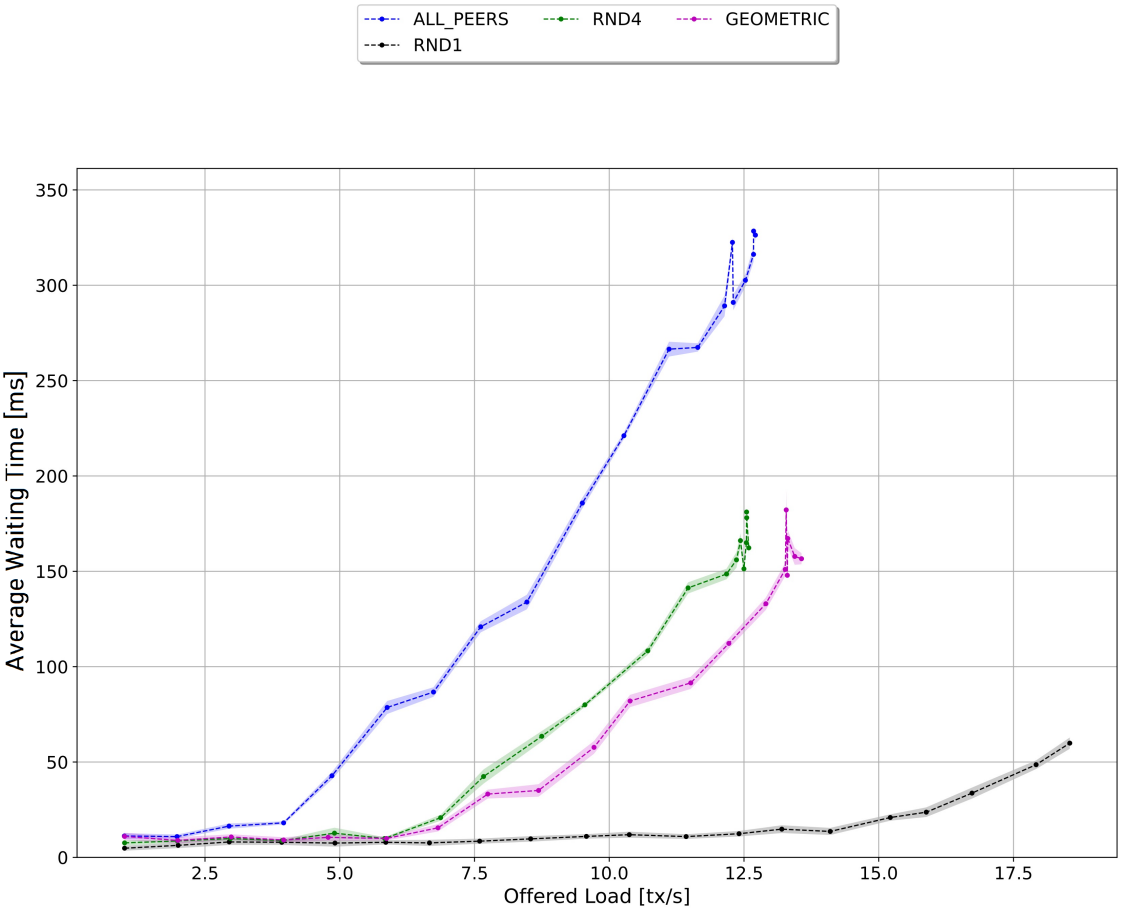


Figure 6.9: Average Waiting Time with respect to the Offered Load

Figure 6.9 shows the values of Average Waiting Time that the endorsing peers experienced in all the inspected algorithms depending on the Offered Load. The value of offered loads for which the AWT was quadruplicated (4X) are:

ALL	RND4	GEOMETRIC	RND1
5.87	7.67	9.71	15.21

It is evident how an increase of the transaction rate makes the average waiting time rise. In addition the number of EPs that are utilized by the system influences both the growth factor and the starting point of the curves.

Conclusions

This study proved how the Fabric network requires more time to endorse a request if a high enough rate of transactions is submitted into the system. The way this growth happens is related to the number of concurrently used EPs. It is worth noting how also in states in which the system can not be considered as saturate, the average waiting time is more than doubled. This means that the bigger AWT is not due to the (extreme) lack of resources but to the incremental number of requests to serve.

6.4.2 Experiment 2 - Execution time distribution

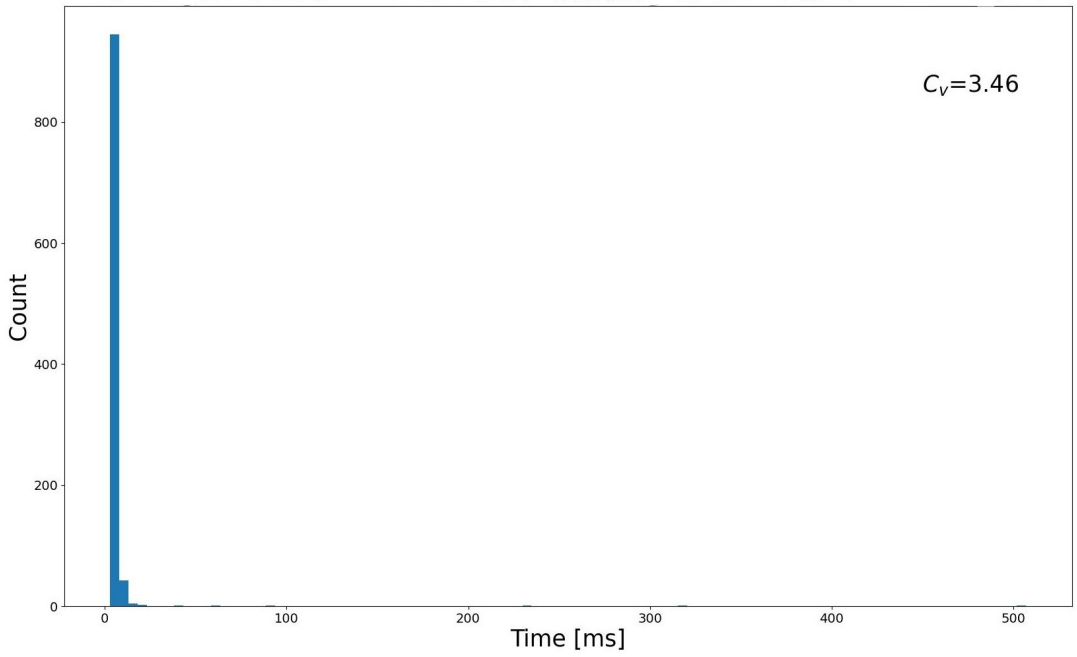


Figure 6.10: Histogram of the measured waiting times

Figure 6.10 shows how the measurements of the execution time are distributed. The average value is **6ms** and its coefficient of variation equals **3.46**. C_v value is far from being zero meaning that there are many outliers inside the dataset. It happened that the execution time assumed values higher than 500ms in an almost unused network. This means that the time that it takes the EP to execute a transaction can not be assumed to be deterministic as it significantly varies. The average value of the execution time strongly depends on the number of EPs that are concurrently utilized by the system, as shown in the first experiment. For this reason it is not possible to uniquely determine the

execution time of Fabric's EP since it depends on the studied case. *Ultimately this analysis provided through the study of the coefficient of variation a measure of the variability of the execution time.*

6.4.3 Experiment 3 - Load of the first k peers

Number of requests

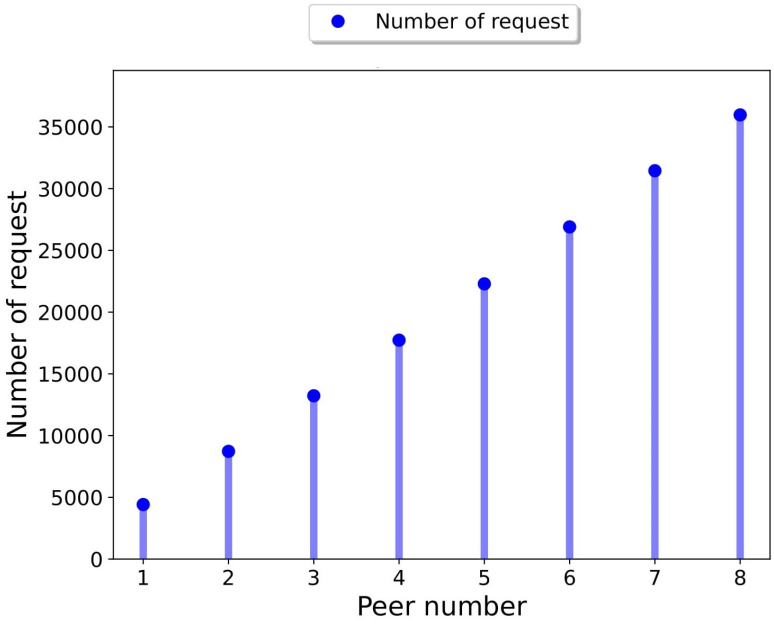


Figure 6.11: Number of requests issued to each EP

The experiment lasted 3806 seconds in which averagely 77% of the CPU was occupied. The distribution of requests received by each EP respects the expected trend as they grow in number starting from EP1 until EP8, as shown in figure 6.11.

Arrival rate

By confronting the arrival rate λ at each EP:

EP	Number of Requests [tx]	Arrival rate [tx/s]
EP1	4417	1.16
EP2	8724	2.29
EP3	13225	3.47
EP4	17728	4.66
EP5	22289	5.86
EP6	26899	7.07
EP7	31450	8.26
EP8	35971	9.45

it can be observed how the arrival rates present a well defined sequence where the value of λ for an EP equals to the one of the previous EP plus a constant. This constant is the arrival rate value for EP1 and this happens thanks to how Uniform was defined. For example EP5 arrival rate is approximately EP4 λ + EP1 $\lambda = 4.66 + 1.16 = 5.82$ tx/s, which value differs 0.7% to the measured one.

Average Waiting Time analysis

By confronting for each EP the arrival rate value with the Average Waiting Time measured:

EP	Arrival Rate [tx/s]	Average Waiting Time [ms]
EP1	1.16	56
EP2	2.29	85
EP3	3.47	113
EP4	4.66	173
EP5	5.86	223
EP6	7.07	316
EP7	8.26	371
EP8	9.45	422

It can be noticed how there is a coherent trend of growth between this two measured. However, this growth does not happen with the same factor between the two sequences and this, as seen during the first experiment, may depend from the fact that significant variation of waiting times can be observed only when a high enough rate of transactions is sustained.

For the way Uniform was defined, when EP1 is asked to endorse, all the other EPs will have to do it too. If the AWT depends just from the higher occupation of the CPU (which peaks when all the EPs are active), each measurements of the waiting time for EP1 should be high. Nevertheless EP1's AWT value equals 56 ms which is 7.5 time smaller than EP8's AWT so it can be affirmed that the AWT is influenced by the arrival rate perceived.

Mathematical queue model

As already mentioned during the second experiment, execution time strongly depends on the scenario in which it is measured and for this reason, the value that will be used for the queue model will be approximated with the one measured for the least loaded EP, which is the first one. EP1 does not experience queuing since it receives 1.16 transactions per second, value for which, during the first experiment, it was possible seeing how the AWT did not have appreciable changes.

Figure 6.12 illustrates the theoretical curve measured with the M/G/1 formula and the measured value for each EP. The values of the theoretical curve are growing monotone as expected. Both the theoretical and measured curves of the average waiting time present the same trend of growth. This means that the AWT for each EP can be correctly associated with the number of transactions per second they are receiving. This finally

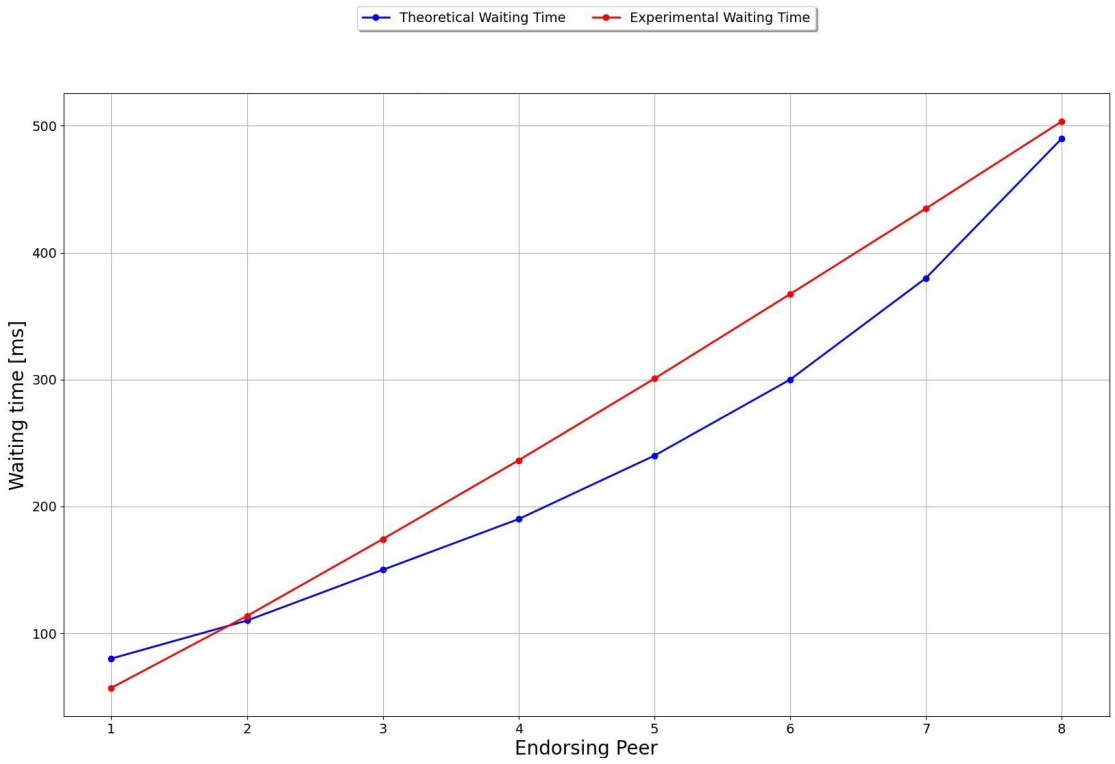


Figure 6.12: M/G/1 model for the waiting time

demonstrates how the rise of the AWT must be due to the fact that EPs, because of resource shortage, queue not yet serviced requests and consequently lengthen the time that it requires to execute them.

6.5 Results - OPEN performances

It is now presented the results of the OPEN performance evaluation with respect to the ORIGINAL algorithm.

6.5.1 High background load

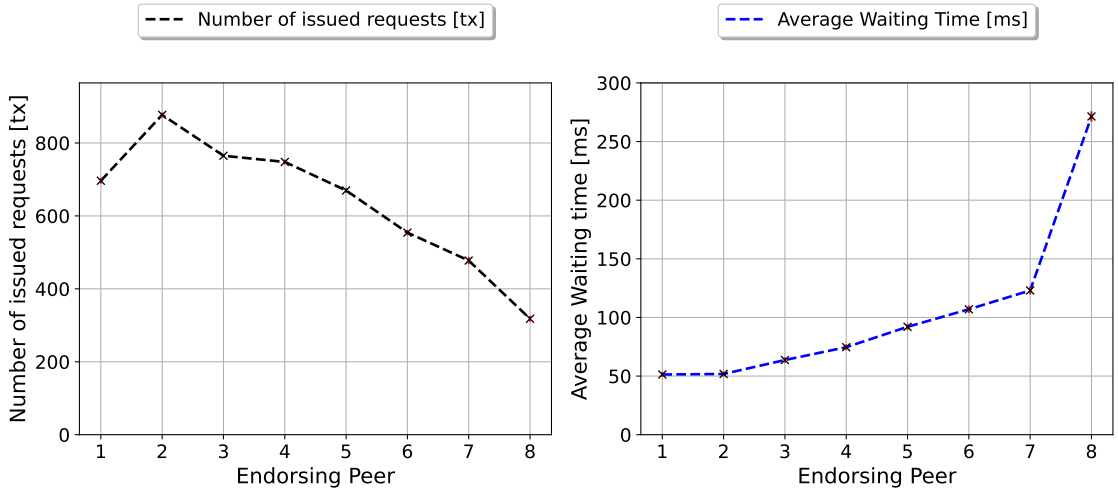
Results

In this scenario the application was running together with the Generator that sustained a transaction rate of 7 tx/s.

Algorithm	AVG(E. time) [ms]
ORIGINAL	115 ± 5
OPEN	94 ± 7

Gain
1.22

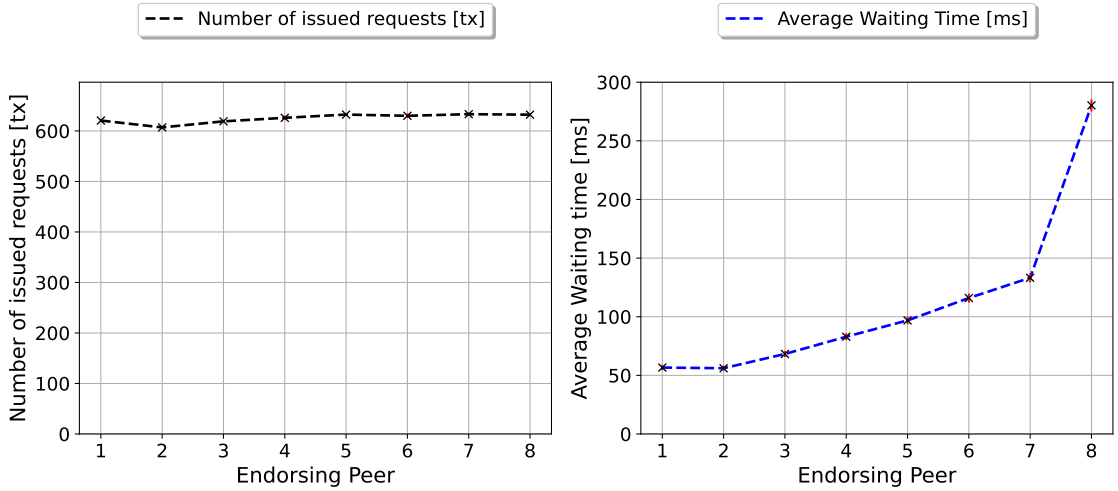
Detailed Analysis



(a) EPs selected for endorsing by the application (b) Average Waiting Time for each EP

Figure 6.13: OPEN algorithm experiment

From the graph representing how many requests were averagely sent to each EP by the algorithm (Figure 6.13 (a)), it is possible to see how OPEN preferred the EPs that were averagely less loaded. As a matter of fact figure 6.13 (b) shows the AWT experienced by each EP. Even if the first 2 EPs have similar AWT value, there is still a growth in the AWT from EP2 to EP8.



(a) EPs selected for endorsing by the application (b) Average Waiting Time for each EP

Figure 6.14: Original algorithm experiment

The original algorithm issued almost the same number of requests to each EP. This

reinforces the claim for which ORIGINAL chooses the endorsers for a transaction by creating a random set of 5 EPs to contact not exploiting past state data.

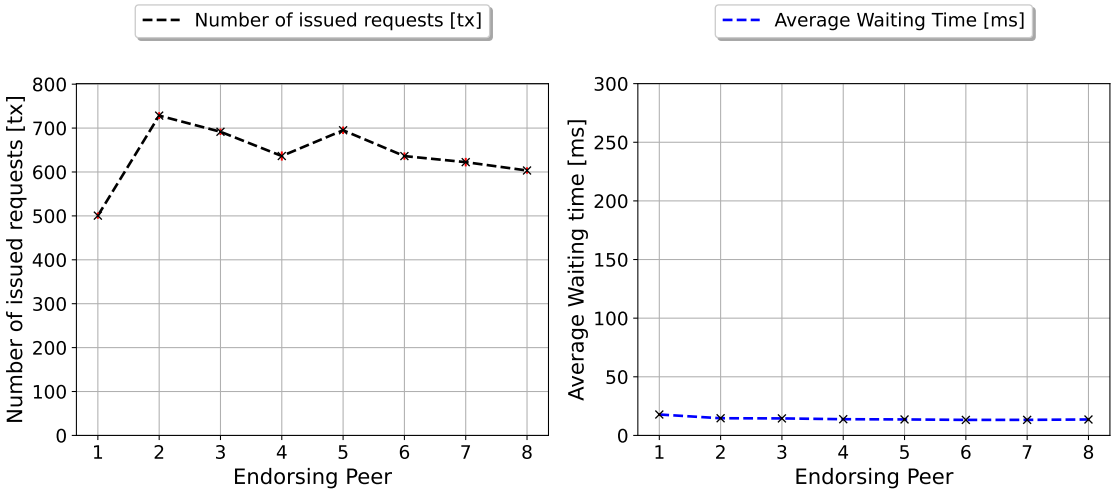
6.5.2 Low background load

In this scenario the application was running together with the Generator that sustained a transaction rate of 3 tx/s.

Algorithm	AVG(E. time) [ms]
ORIGINAL	26 ± 4
OPEN	22 ± 2

Gain
1.18

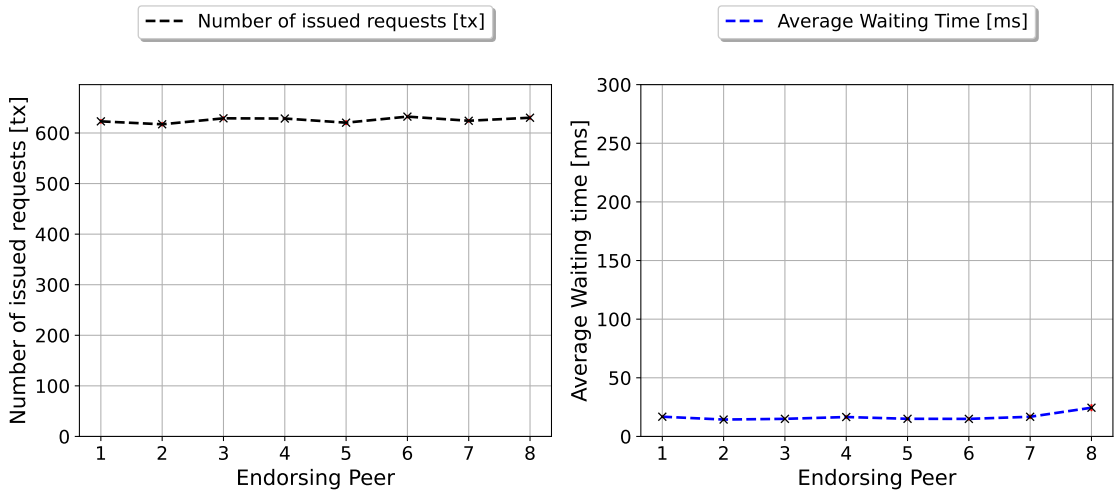
Detailed Analysis



(a) EPs selected for endorsing by the application (b) Average Waiting Time for each EP

Figure 6.15: OPEN algorithm experiment

In a low background load scenario, there is not a big difference in the AWT measured for each EP. Despite the low rate it was possible to create a partial correlation on the way the EPs responded enabling OPEN to select efficiently the EPs. The result obtained by the Low Background scenario is near to the ones obtained with the High Load Scenario. *However, the uncertainty of the measured values is higher meaning that the results are less stable and they may depend more on the execution conditions.*

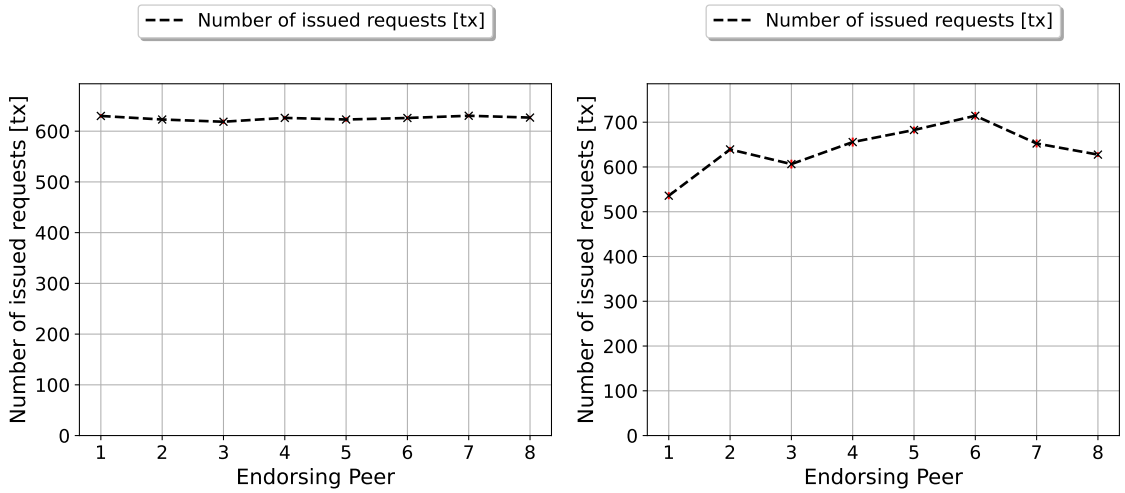


(a) EPs selected for endorsing by the application (b) Average Waiting Time for each EP

Figure 6.16: Original algorithm experiment

6.5.3 No background load

In this scenario the application was running without the use of the Generator.



(a) EPs selected by the application running ORIGINAL (b) EPs selected by the application running OPEN

Figure 6.17: Original algorithm experiment

Figure 6.17 illlustrates how many requests were averagely sent to each EP by OPEN and ORIGINAL algorithms. For this scenario no AWT graph is presented since the Generator was not used. When no background load was created OPEN performed worse than the ORIGINAL algorithm. This is probably due to the **high variability of the**

Algorithm	AVG(E. time) [ms]
ORIGINAL	13 ± 1
OPEN	17 ± 1

Gain
0.76

waiting time. When no queuing is experienced, the information given by the AWT does not offer any useful data for OPEN to choose a "better" EP for the next transactions. OPEN did not follow any particular pattern meaning that it switched several times the EP selection resulting in worse results.

Chapter 7

Conclusion

This thesis introduced **Blockchain technology** and **Hyperledger Fabric**'s architecture with all the steps that it faces to correctly accept a transaction in its ledger. Later the study focused on the **Endorsement phase** and in specific how the **Fabric SDK** faces it. Afterward, it was presented an algorithm called **OPEN**, theorized by Doctor Lotfimahyari Iman, that proposes a way of reducing the endorsing time experienced by the Client by selecting efficiently the EPs depending on their past behavior. A a way of implementing it inside the Fabric Node SDK was presented. Finally, the result chapter showed the domain of working of the Generator, how much it could put to the stress the Fabric network and how that influenced the average value of the waiting time. Through the definition of the **Uniform** distribution, it was possible to create an environment in which a strategic analysis of the AWT of each EP could be done. Despite the contribution given by the high occupation of the system and of the overload for the committing and validating of the blocks, this study confirmed how higher rates of transaction could influence the time that it takes to service an endorsement request from when it is received. The AWT measured for each EP was confronted with the theoretical ones obtained by using the **Pollaczek-Kinchin formula for the M/G/1 queue model**, resulting in similar trends. This enabled to assume that it is possible to create queuing inside the EPs. Having proven that, the study focused on benchmarking OPEN performances in different scenarios to understand how it performs in distinct situations. To create those stress scenarios, the Generator was exploited to issue predetermined values of rate (which defined the stress levels) while the application was running. For the limitations imposed by the hybrid approach, it was not possible to create the ideal environment in which to test OPEN. Anyway, it could be observed how in the high load scenario the graphs of AWT showed an appreciable growth that follows the distribution of transactions given by the Uniform. In addition for each scenario, both OPEN and ORIGINAL had similar trends in the AWT's values distribution meaning that the experiments were done through the same exact condition of the network. **OPEN performed up to 22% better than the ORIGINAL** algorithm when a background load was issued into the network. This means that as long as the EPs respond slower for a matter of network or resource availability, using a state-based endorsing peer selection algorithm can make the Client experience a lower endorsing time. However OPEN potentiality can be better

evaluated in wide distributed networks where more EPs are available with respect to the dimension of the layout of endorsers created by the Fabric SDK.

Chapter 8

Appendix

8.1 optimalEndorsementModule

```
//#### OPEN PARAMETERS
let T = 10;
const eps = 0.001; /// (1ms)
let count = 0

class optimalEndorsementModule{
  // transaction number
  n = -1;

  // Data structures for OPEN algorithm
  peers = [];
  delays = [];
  eligibility = [];

  // parameters
  constructor(){
    this.initialized = false;
  }

  isInitialized(){
    return this.initialized;
  }

  /**
   * POSSIBLE OPTIONS:
   *   1. f("OPEN"), SENDS FOR 2 TXS ENDORSEMENT
   TO ALL THE PEERS AND THEN USES OPEN ALGORITHM
   *   2. f("RND", NUMBER_OF_RANDOM_PEER(S))
   <=== IF NOT SET THE DEFAULT RANDOM PEER 2
   */
  bootUp(mode, totalRandom){
    console.log("[Bootup phase]: Initializing module");
```

```
  // TELLS US IF THE PEER IS ELEGIBLE TO ENDORSE THE NEXT TRANSACTION
  this.eligibility = new Array(NUM_ORG)

  for (var i = 0; i<this.eligibility.lenght; i++) {
    this.eligibility[i] = false
  }

  this.delays = new Array(2);
  this.peers = new Array(2);

  for (var i = 0; i<this.peers.length; i++) {
    this.delays[i] = new Array(NUM_ORG);
    this.peers[i] = new Array(NUM_ORG);
  }

  // PEER STRUCTURE INITIALIZATION
  for (var i = 0; i<this.peers.length; i++) {
    for (var j = 0; j<NUM_ORG; j++) {
      this.delays[i][j] = -1;
      this.peers[i][j] = -1;
    }
  }

  this.n = 1
  this.initialized = true;

  this.mode = process.argv.slice(2)[0]

  if (this.mode!= "RND"){
    console.log("[ALGORITMH SELECTED]: " + mode + "." )
  }

  if(this.mode == "RND"){
    if(totalRandom != undefined){
      this.totalRandom = totalRandom
      this.INITIAL_DELAY_AQUIRING_STEP = 0
    } else {
      console.log("[BOOTUP] : RND_4 will be used")
      this.totalRandom = Math.floor(NUM_ORG/2)
    }
    console.log("[ALGORITMH SELECTED]: " + mode + "_" + totalRandom + "." )
  } else if (this.mode == "OPEN") {
    this.INITIAL_DELAY_AQUIRING_STEP = Math.floor(NUM_ORG/2)
  }


}

/*#####################  GETTERS  ###################*/
//#### getLayout() RETURNS THE LAYOUT ACCORDING TO THE CHOSEN POLICY.
```

```
  getLayout(){
    console.log("----------------------------------->" + this.n);
      if ( this.n < this.INITIAL_DELAY_AQUIRING_STEP ){
        OEM.selectEndorsers("INITIAL_STEPS")
      } else {
        OEM.selectEndorsers();
      }

      this.n = this.n + 1;

      let result = []
      let i = 0
      let a = {}

      let dim = Math.max(...this.peers[0])+1


      while (i<dim){
        for (let j=0; j < NUM_ORG ;j++){
          if(this.peers[0][j] == parseInt(i) ){
a["G"+j]=1
            i=i+1
            break
          }
        }
      }
      return a;
  }


  /* ######################## SETTERS  ##################### */
  //#### setEndorsementDelays() sets the delays that were just measured.
  setEndorsementDelays(delay_data){
    OEM.prepareDelays()
    this.delays[0] = extractDelays(delay_data)
  }

  /* ######################## UTILS  ################### */
  /**
   * selectEndorsers(MODE) chooses the next endorsement depending on the MODE
   * MODE is optional and if not specified uses the MODE chosen at construction time.
   *   -OPEN
   *   -SELECT_ALL
   *   -INITIAL_STEPS
   */
  selectEndorsers(ENDORSERS_SELECTION_ALGORITHM){
    if (this.n != 1){
      OEM.preparePeers()
    }
```

```
// IF NO MODE GETS SPECIFIED IT DOES THE ONE IT WAS AT FIRST
//DEFINED ON THE CONSTRUCTOR
if (ENDORSERS_SELECTION_ALGORITHM == undefined){
  ENDORSERS_SELECTION_ALGORITHM = this.mode
}

if (ENDORSERS_SELECTION_ALGORITHM == "OPEN"){
  // Max delay among endorsements to be used later
  let dMax = -1
  for (var i = 0; i<NUM_ORG; i++){
    if (this.peers[1][i] != -1 && this.delays[0][i] > dMax){
      dMax = this.delays[0][i];
    }
  }

  if (dMax == -1){ // no response from TX-1
    for (var i = 0; i<NUM_ORG; i++){ // all peers
      this.delays[0][i] = this.delays[1][i]; // use previous delays
      if (this.peers[1][i] != -1){ // if was endorser
        this.elegibility[i] = false; // make it NOT-eligible
      }
    }
  } else { // We have response from TX-1
    for (var i = 0; i<NUM_ORG; i++){ // all peers
      if (this.peers[1][i] != -1){ // if was endorser
        if (this.delays[0][i] == -1){ // if no response
          this.delays[0][i] = dMax + eps; // speculate it
          this.elegibility[i] = false; // make it NOT-eligible
        }
      } else { // if was NOT endorser
        this.delays[0][i] = this.delays[1][i]; // use previous delays
      }
    }
  }

  this.peers[0] = this.selectBestPeers(Math.floor(5),
  this.delays[0].slice()).slice();

  if (this.n%T == 0){
    OEM.endorseRandomPeer();
  }

} else if (ENDORSERS_SELECTION_ALGORITHM == "SELECT_ALL"
 || ENDORSERS_SELECTION_ALGORITHM == "INITIAL_STEPS"){
  this.peers[0] = [0,1,2,3,4,5,6,7].map(x=>parseInt(x))
} else if (ENDORSERS_SELECTION_ALGORITHM == "RND") {
  for (let i = 0 ; i < this.totalRandom ; i ++ ) {
    this.endorseRandomPeer()
  }
}
```

```
}

//#### prepareDelays() shifts this.delays structure.
prepareDelays(){
  // INITIALIZING DELAYS VECTOR
  this.delays[1] = this.delays[0].slice();
  this.delays[0] = this.delays[0].map(x => x = -1);
}

//#### preparePeers() shifts this.peers structure.
preparePeers(){
  // Initialize eligibility vector for all peers
  this.eligibility = this.eligibility.map(x => x = true);

  // INITIALIZING PEERS VECTOR
  this.peers[1] = this.peers[0].slice();
  this.peers[0] = this.peers[0].map(x => x = -1);

}

//#### selectBestPeers(n,vector) selects n peers with lowest delays in vector
selectBestPeers(n, vector){
  // EMPTY STRUCTURE WHICH WILL CONTAIN THE PEER THAT ARE SELECTED
  let selectedPeers = new Array(NUM_ORG).fill(-1)

  let dMax = Math.max(...vector);
  for (var i = 0; i<n; i++){
    let dMin = dMax;
    let indMin = -1;
    // SELECTS THE PEER WITH MINIMUM LATENCY(REAL MEASURED OR ESTIMATED)
    //(BEING CAREFUL TO NOT CONSIDER VALUES SMALLER THAN 0 (LIKE -1, WHICH MEANS
    //THAT NO RESPONSE WAS RECEIVED.))
    for (var j = 0; j<NUM_ORG; j++){
      if (vector[j] != -1 && vector[j] <= dMin && selectedPeers[j]==-1){
        dMin = vector[j];
        indMin = j;
      }
    }
    if (indMin!= -1){
      selectedPeers[indMin] = Math.max(...selectedPeers)+1
    }
  }
  return selectedPeers;
}

//#### endorseRandomPeer() adds a random peer in the endorsement
endorseRandomPeer(){
  let tmpRndPeers = []
  for (var i = 0; i<NUM_ORG; i++){
    if (this.peers[0][i] == -1){
```

```
        tmpRndPeers.push(i);
      }
    }
    let done = false
    if (tmpRndPeers) {
      let randomIndex = this.getRandomInt(0,tmpRndPeers.length);
      this.peers[0][tmpRndPeers[randomIndex]] = Math.max(...this.peers[0])+1;
    }
  }

  //#### getRandomInt(i) generates a random int value in the [min,max) interval
  getRandomInt(min, max) {
    min = Math.ceil(min);
    max = Math.floor(max);
    //The maximum is exclusive and the minimum is inclusive
    return Math.floor(Math.random() * (max - min) + min);
  }



function extractDelays(delay_data) {
  let end_delays = new Array(NUM_ORG).fill(-1);
  // let end_delays = {}

  for ( let i = 0 ; i < end_delays.lenght; i++ ){
    end_delays[i] = -1;
  }

  for (const key of Object.keys(delay_data[3][0])) {
    // console.log("key:" + key)
    let peer_num = parseInt(key[1]);

    if (delay_data[3][1].hasOwnProperty(key)){
      end_delays[peer_num] = parseFloat(delay_data[3][1][key]-delay_data[3][0][key]);
    }
  }
  return end_delays
}

let OEM = new optimalEndorsementModule();

function getOEM(){
  if(MODE[0] == "ORIGINAL"){
    return undefined;
  } else if ( MODE[0] == "OPEN" || MODE[0] == "RND"){
    if (OEM.isInitialized() == false){
      OEM.bootUp( MODE[0], MODE[1] )
    }
    return OEM
  } else {
```

```
    console.log("[ERROR]: You inserted a wrong algorithm. Exiting.")
    exit(0)
  }
}
```

8.2 Generator

```
def generator(R,seconds,SELECTION_ALGORITHM,N):

        PRE_JOIN_THREADS = True

        NUM_ERROR = 0

        Actualtimestamp = timestamp()

        y=[]
        executionDurations=[]
        periodDurations=[]
        committedBlocks=[]
        txIDs=[]
        delays=[]
        responses =[]

        totalDuration = 0
        totalElapsingTime = 0
        preJoined = 0
        afterJoined = 0
        txNumber = int(R*seconds)


        threadStartTimes= [None] * (txNumber)

        threadFinishTimes=[None] * (txNumber)



################################### THREAD GENERATION
        CPUoccupation = []


        averageCPU=0
        cpucount=0
        txCount = 0

        for a in range(0, int(txNumber)):
                 activeThreads=0
                 deadThreads=0
                 start  = datetime.datetime.now()
                 if THREADS:
```

```
                    if (a==0):
                        startExperiment = datetime.datetime.now()

                    x = threading.Thread(target=insertTransactionFunction,
                    args=(a,SELECTION_ALGORITHM,N))
                    x.start()


                    if (a == txNumber - 1):
                         finishSpawningTime = datetime.datetime.now()

                    CPUoccupation.append(float(measureCPUoccupation()))

                    y.append(x)
                    if PRE_JOIN_THREADS:
                        if float(R) > 1:
                            interval = int(R)
                        elif float(R)< 1:
                            interval = int(1/float(R))
                        else:
                            interval = 10
                        if (a%interval == 0 and a > 2* interval):
                            for i in range(a-2*interval, a-interval, 1):
                                if not y[i].is_alive():
                                    y[i].join()
                                    preJoined = preJoined + 1

                else:
                    insertTransactionFunction(a,R)

                timeExecution = (datetime.datetime.now()-start).total_seconds()
                remaining = 1/R - timeExecution
                if remaining > 0:
                    time.sleep(remaining)

                timePeriod = (datetime.datetime.now()-start).total_seconds()
                executionDurations.append(timeExecution)
                periodDurations.append(timePeriod)

################################### METRICS CALCULATION

        offeredLoad =
        ((txNumber)/(finishSpawningTime - startExperiment).total_seconds())

        if THREADS:

            print("Joining thread....", end="")

            for a in range(0,len(y)):
                if not y[a].is_alive():
```

```
            y[a].join()
            afterJoined = afterJoined + 1

    print("...finished!")


threadStartTimes = [elm for elm in threadStartTimes
if isinstance(elm, datetime.datetime)]

threadFinishTimes = [elm for elm in threadFinishTimes
if isinstance(elm, datetime.datetime)]


firstStart = min(threadStartTimes)
lastFinish = max(threadFinishTimes)

throughput = (txNumber) / ((lastFinish - firstStart).total_seconds())

measurament = compute_metrics()
return measurament
```

Bibliography

[1] Application — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/application.html. (Accessed on 05/28/2022).

[2] Blockchain - wikipedia. https://en.wikipedia.org/wiki/Blockchain. (Accessed on 05/08/2022).

[3] Channels — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/channels.html. (Accessed on 05/18/2022).

[4] Endorsement policies — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/endorsement-policies.html. (Accessed on 05/18/2022).

[5] Gateway — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/gateway.html. (Accessed on 05/28/2022).

[6] Gossip data dissemination protocol — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/gossip.html. (Accessed on 05/18/2022).

[7] History of blockchain - javatpoint. https://www.javatpoint.com/history-of-blockchain#:~:text=The%20blockchain%20technology%20was%20described,not%20be%20backdated%20or%20tampered. (Accessed on 05/08/2022).

[8] Hyperledger fabric sdk for node.js class: Contract. https://hyperledger.github.io/fabric-sdk-node/release-1.4/module-fabric-network.Contract.html. (Accessed on 05/28/2022).

[9] Hyperledger fabric sdk for node.js module: fabric-network. https://hyperledger.github.io/fabric-sdk-node/release-1.4/module-fabric-network.html. (Accessed on 05/28/2022).

[10] hyperledger/fabric-sdk-node: Hyperledger fabric sdk for node https://wiki.hyperledger.org/display/fabric. https://github.com/hyperledger/fabric-sdk-node#readme. (Accessed on 05/28/2022).

[11] Ledger — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/ledger/ledger.html. (Accessed on 05/18/2022).

[12] Linux foundation - wikipedia. https://it.wikipedia.org/wiki/Linux_Foundation. (Accessed on 04/08/2022).

[13] The longest chain - blockchain guide. https://learnmeabitcoin.com/technical/longest-chain#:~:text=The%20longest%20chain%20is%20what,on%20the%20same%20transaction%20history. (Accessed on 05/11/2022).

[14] Order of transactions and how blockchain avoids double spend | by karthik margabandu | medium. https://medium.com/@karthikmargabandu7/order-of-transactions-and-how-blockchain-avoids-double-spend-9daf9f697b8f. (Accessed on 04/08/2022).

[15] Peers — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/peers/peers.html. (Accessed on 05/18/2022).

[16] Proof of stake vs. proof of work: What's the difference? https://www.businessinsider.com/personal-finance/proof-of-stake-vs-proof-of-work?r=US&IR=T#:~:text=Proof%20of%20stake%20requires%20participants,slower%20and%20consumes%20more%20energy. (Accessed on 05/11/2022).

[17] Service discovery — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/discovery-overview.html. (Accessed on 05/28/2022).

[18] Smart contracts and chaincode — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/smartcontract/smartcontract.html. (Accessed on 05/18/2022).

[19] Transaction flow — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/txflow.html. (Accessed on 05/28/2022).

[20] Understanding hyperledger fabric — endorsing transactions | by kynan rilee | koki | medium. https://medium.com/koksterhyperledger-fabric-endorsing-transactions-3c1b7251a709. (Accessed on 05/18/2022).

[21] Walkthrough of hyperledger fabric node sdk and client application | by salman dabbakuti | datadriveninvestor. https://medium.datadriveninvestor.com/walkthrough-of-hyperledger-fabric-client-application-aae5222bdfd3. (Accessed on 05/28/2022).

[22] Wallet — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/wallet.html. (Accessed on 05/28/2022).

[23] Writing your first application — hyperledger-fabricdocs main documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.2/write_first_app.html. (Accessed on 05/28/2022).

[24] Mounir Mahmoud Moghazy Abdel-Aal. Survey-based calibration of a parking entry as a single-server mathematical queuing model: A case study. *Alexandria Engineering Journal*, 59(2):829–838, 2020.

[25] Elli Androulaki, Angelo De Caro, Matthias Neugschwandtner, and Alessandro Sorniotti. Endorsement in hyperledger fabric. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 510–519, 2019.

[26] Francesca Antonucci, Simone Figorilli, Corrado Costa, Federico Pallottino, Luciano Raso, and Paolo Menesatti. A review on blockchain applications in the agri-food sector. *Journal of the Science of Food and Agriculture*, 99(14):6129–6138, 2019.

[27] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[28] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. Reducing latency via redundant requests: Exact analysis. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, page 347–360, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Liang Huang and Tony Lee. Generalized pollaczek-khinchin formula for markov channels. *Communications, IEEE Transactions on*, 61:3530–3540, 08 2013.

[30] Nir Kshetri and Jeffrey Voas. Blockchain in developing countries. *It Professional*, 20(2):11–14, 2018.

[31] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system bitcoin: A peer-to-peer electronic cash system. *Bitcoin. org. Disponible en https://bitcoin. org/en/bitcoin-paper*, 2009.

[32] Nihar B. Shah, Kangwook Lee, and Kannan Ramchandran. When do redundant requests reduce latency? *IEEE Transactions on Communications*, 64(2):715–722, 2016.

[33] Mark Soelman, Vasilios Andrikopoulos, Jorge A. Perez, Vasileios Theodosiadis, Karel Goense, and Arne Rutjes. Hyperledger fabric: Evaluating endorsement policy strategies in supply chains. In *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 145–152. IEEE, August 2020. The 2nd IEEE International Conference on Decentralized Applications and Infrastructures ; Conference date: 03-08-2020 Through 06-08-2020.

[34] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, page 283–294, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Ashish Vulimiri, Oliver Michel, P. Brighten Godfrey, and Scott Shenker. More is less: Reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, page 13–18, New York, NY, USA, 2012. Association for Computing Machinery.

[36] EyalI ZhangF et al. Rem: Resource efficient miningforblockchains. *Proceedingsoftheciteseerpress securitysymposium (usenixsecurity17). Vancouver, Canada*, 1427:1444, 2017.