Trabajo Fin de Master
Master en Ingeniería Industrial

Reputation-Based Consensus on a Secure Blockchain Network

Author: Emilio Marín Fernández

Tutor: José María Maestre Torreblanca

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Departamento de Teoría de la Señal y Comunicaciones

Trabajo Fin de Master

Master en Ingeniería Industrial

Reputation-Based Consensus on a Secure Blockchain Network

Autor:

Emilio Marín Fernández

Tutor:

José María Maestre Torreblanca

Profesor Titular

Dpto. de Ingeniería de Sistemas y Automática

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Master: Reputation-Based Consensus on a Secure Blockchain Network

Autor: Emilio Marín Fernández
Tutor: José María Maestre Torreblanca

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

A mis padres, hermanos y pareja por su apoyo incondicional.
A mis abuelos por todo lo que me han enseñado.
A mi tutor Jose María Maestre por toda la ayuda prestada durante la realización de este proyecto.

Emilio Marín Fernández
Master en Ingeniería Industrial

Sevilla, 2020

Abstract

The aim of this thesis is to research new algorithms of consensus for distributed systems and running these consensus within a secure blockchain network built with Hyperledger Fabric allowing a full traceability of transactions and audit of the process.

We will describe every detail involved in the creation of a blockchain network, from the main concepts to developing a production-ready network with identity management and multiple nodes. Using Hyperledger Fabric and Composer we will be able to create simple and complex networks that will be reachable from any client through a REST API.

A new reputation-based consensus algorithm will be proposed taking full advantage of the blockchain technology. Each participant of the network will have a reputation level that increases or decreases depending on their behavior with the rest of the network. In every consensus round there's a verification phase within the blockchain smart contracts that would decide the reputation delta of each participant.

With everything developed we will compare the proposed algorithm with others learned in the literature in different scenarios such as when malicious agents try to prevent the consensus being reached or generating an error in the agreed value, we will study the speed, value offset and convergence.

Contents

1 Introduction

Every once in a while, a new technology emerges trying to disrupt the world as it is known. This is happening at this very moment and it is not known by many, the technology: *Blockchain*. We live in a world dominated by data, everything you are able to do with a device connected to the internet will most likely end with one or more companies gathering information about the actions that were done, while the user's fear about their privacy increases rapidly, a new opportunity for a new system or technology to change this appears. With the goal of removing intermediaries and making possible peer to peer (*P2P*) interactions, the idea of Blockchain was created alongside the concept of Bitcoin by Satoshi Nakamoto in 2008 in the whitepaper "Bitcoin: A Peer-to-Peer Electronic Cash System". [1]

> What is needed is an electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party. – Satoshi Nakamoto (2008)

Although the initial use case for the blockchain technology was around payments or monetary transactions, in the 11 years since its creation it has been understood that it is just the tip of the iceberg.

This document explores new possibilities for this technology to tamper proof consensus algorithm in distributed networks. The structure of this project will be as follows:

1. Description of the blockchain technology, what components does it have and how it works. We will also explore different use cases that companies are creating. Next, we will present the suite of tools that will be used in the project, Hyperledger, giving definitions of concepts and components.

2. In chapter 3, we will layout the consensus problem and will define the algorithms that will be later on developed. Here we will also propose a reputation-based consensus algorithm that will create a reputation for each agent and will be modified depending of how they follow the rules set for consensus. Based on this reputation, the rest of the network will take more or less into account the value received.

3. Once the technology is defined and the problem we want to solve we will enter into the development of the solution. First we need to create a blockchain network with Hyperledger where the participants of the consensus will communicate with each other. Here we will have to develop the smart contracts that will define how this communication is done and also the rules of consensus. We will also get into how a more complex network would be created for a production setup. Lastly, we will need to develop scripts that will simulate the behavior of participants and integrate it with the blockchain network so we can test different scenarios.

4. Now that we have created our blockchain network and consensus rules we will setup different configurations trying to prove the advantages and disadvantages of the proposed reputation-based consensus algorithm.

5. Lastly, we will discuss the conclusions and consider the effectiveness of the system and algorithm created as well as future steps.

1.1 Motivation and Objectives

Nowadays we are heading into a world where all the devices are connected with each other but, normally, the information processed by these devices end up in the hands of some company. Here is where we find the two main issues that the technology studied in this project will try to solve.

In the one hand, the privacy of our daily actions is disappearing. Either when searching something about a new gadget or asking Alexa to order a product, this information ends up being linked to our profile and later on sold to advertising companies.

In the other hand, this information usually is property of a few companies that are no more than centralized points of failures. If any company gets breached by a malicious agent, all of our data could end up in bad hands.

Blockchain comes trying to solve some of these aspects. By removing third parties, we reduce the information shared, interacting directly with the end user, P2P. By using cryptography and decentralization, the possibility of a breach or alteration of data is far less probable.

This project is a proof of concept that tries to find a solution to the current consensus problem using blockchain. The main objectives are to study the consensus problem, analyze current literature about consensus algorithms and create our own consensus algorithm based on reputation. In addition we will make the participants of the consensus communicate through a blockchain network to keep track of the status of each agent and avoid having the information altered by a third party.

We will create different scenarios where we will simulate a malicious agent trying to modify the result of the consensus for its own benefit or even trying to keep the participant out of an agreement.

2 Blockchain

2.1 What is Blockchain?

Blockchain is, as defined by the Merriam-Webster dictionary[2], *a digital database containing information (such as records of financial transactions) that can be simultaneously used and shared within a large decentralized, publicly accessible network*. At the same time, the word itself "Blockchain" (*chain of blocks*) gives some information about how the information is stored.

Figure 2.1 Representation of a Blockchain[3].

Blockchain can be defined as a distributed and decentralized ledger that holds information. To give more context to this definition there is a need to define the following concepts [4][5]:

- **Ledger**: It is a written or computerized collection of financial accounts, usually sorted by date.
- **Distributed Ledger**: Related with the location of the ledger, in Blockchain each participant of the network holds a copy of this ledger therefore keeping a record of all the transactions made.
- **Decentralized Ledger**: Refers to the level of control over the ledger, a centralized ledger would mean that a single person or entity has authority over all the actions possible in the network. On the other hand, a decentralized ledger implies that all the participants of the network have the same power.

As a consequence, each participant in the system interacts directly with other users, making it a peer-to-peer network, this way third parties are removed and, as a result, possible services charges are eliminated.

Each block, from the chain of blocks, contains a piece or collection of information inside in the form of transactions, this data is usually formed by multiple attributes, Figure 2.3 [5]:

- **Time-stamp**: Date when the transaction was submitted to the network.

Figure 2.2 Concepts of distributed and decentralized networks [6].

- **Hash[1] of the previous block**: Encrypted information of the previous block in the chain.
- **Data**: Actual information to store in the blockchain.
- **Hash of the current block**: All the mentioned parameters are encrypted into a hash that will identify the block.

With these attributes each block will be connected to the previous one, forming then a chain. The first block of the chain is know as the *Genesis* block and it is the only case where there is no reference to a previous block. Sometimes, the data included in the block 0 is fake or hardcoded to give the network a start.

For example, in the case of cryptocurrencies like Bitcoin, the information inside a transaction would include the address of the sender, the address of the receiver, the amount of BTC to send and a nonce[2]. All this information alongside the other parameters defined above would form a transaction, then the transaction is hashed and imported to the block, the block can store multiple transfers at the same time. It is important to note that any changes to the data would cause a completely different transaction hash.

Since the blocks are connected with each other it makes the data immutable, once the transaction is submitted to the network it cannot be modified. Like it was said, each participant in the network holds the whole chain of data, if a bad actor tries to tamper previous information in a block (e.g. by changing the amount he received in his wallet) he would need to build a new chain, different than what other participants are holding, and they would need to be convinced that the new chain is the correct one. This is only possible if the bad actor owns 51%, in addition the blocks need to be verified (more on chapter 2.2) and this is computational expensive, so with a big enough network it would be nearly impossible for a bad actor to own the majority of the system and then the cost involved in the tampering of the data would not be worth it.

2.2 How does Blockchain work?

To sum up some of the main concepts defined in the previous section, blockchain is a distributed and decentralized ledger used to store any kind of information, this data is stored inside blocks which are connected to other blocks by storing the hash of the previous one alongside the data. Each participant holds the whole chain of information and has the same power over the network [7].

The main components of a blockchain network are (usually):

[1] A hash is function that encrypts information into a fixed length output.

[2] Cryptographic value that can only be used once. In some networks there are two kind of nonces involved, an account nonce to prevent replay attacks and a validation nonce to support the mining algorithm.

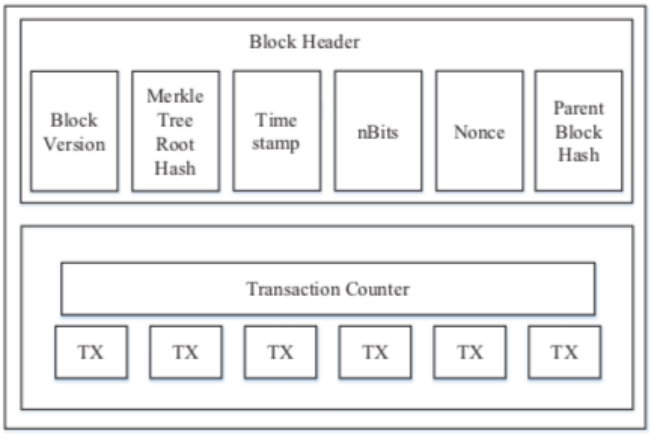


Figure 2.3 Block Structure [7].

- Peer to peer network that connects participants of the system and distributes the verified blocks holding the data to all members of the network based on a "gossip" protocol, each participant in the network is also called a *node*
- Transactions holding data.
- Consensus rules over the transactions, defines what is a transaction, how it should be formed and what makes a valid/verified transaction.
- State machine to process the transactions to determine whether they follow or not the rules.
- Consensus algorithm to decentralize control over the blockchain, participants must follow and cooperate with said rules.
- Incentive program to reward users that play by the rules or helps the self governance of the network validating transactions, in public blockchain these are also known as miners, they validate the blocks and get a prize in return (e.g. some kind of cryptocurrency with a fiat value)

All this components are combined to build an application or client to make easy for users to create a new node of the network.

If we talk about a generic blockchain, these are the steps that would take place in the event of a transaction submission:

1. User submits an action to platform or client.
2. The action's information is packed inside a transaction with all the needed data and then it is sent to the network.
3. The validators or miners of the network verify that the transaction is valid and import it inside a block.
4. Once the block's time has passed or the maximum number of transactions allowed is reached the block is distributed through the network to all the participants.
5. Each node update their chain to hold the latest information.

Usually, every blockchain network follow this flow to create a new transaction and block, the difference resides in the consensus algorithm used to verify the validity of the transaction submitted. The main algorithms used nowadays, development in this field is increasing and improving rapidly, are as follows [8][7][9]:

- **Proof of Work (PoW)**: Consists on the resolution of a difficult mathematical problem, hashing. Like

it was defined hashing consists on transforming data in a encrypted output (e.g. SHA256, keccak-256 hashing), the only way to get the hash without having the initial data is by trial and error, this proves that the miner has done certain amount of computational work. There is also a difficulty defined, it is related with the number of '0' known in front of the hash. The cost to compute one SHA256 hash is:

$$PoW\ Cost_{expected} = \frac{Cost_{Hash}}{Chances\ of\ success} = Cost_{Hash} * Difficulty \tag{2.1}$$

This cost is worth it since there is a reward involved to incentivize the first miner that verify the block. The difficulty is also handled by the system to create an equilibrium and make the system work for the specifications at creation (e.g. block speed, if computational power increases in the network the difficulty is increased to stay in the equilibrium stated). Most of the current cryptocurrencies use this consensus algorithm (e.g. Bitcoin, Ethereum, Litecoin, Monero). The disadvantages of this algorithm, in the example of Bitcoin, is that the hashing power needed at the time of writing of this project continues to grow and so does the computational power needed, that translates to a higher price to build a mining machine and a higher cost in electricity.

- **Proof of Stake (PoS)**: In this consensus algorithm, the computational power needed to verify blocks is minimal. Instead of using the computer's power to decide which transaction is valid, the participant's coins are used. The validators, concept of miners in PoW, lock a specific amount of coins, they take turns on proposing and voting which is the next valid block, the bigger the amount locked the higher the chances or weight in this process. When a validator is selected, he will validate a block and get a reward in return, if the validation was not done correctly he will lose the stake. This way it is in the validator's best interest to play by the consensus rules to hold the coins and keep participating in the network. On the other hand, the punishment for a bad miner in the PoW network is just the waste of electricity.

- **Byzantine Fault Tolerance (BFT)**: This concept comes from the Byzantines General's problem[3], a Byzantine Fault Tolerant system is able to continue operating even if some of the nodes fail to communicate or decide to do it maliciously. The main advantages of this algorithm is that it provides a high throughput, low transaction costs and good scalability. The disadvantage is that while it's not centralized it is not completely decentralized, it's often used in hybrid blockchain networks.

- **Proof of Authority (PoA)**: In this case, only approved accounts can validate transactions. The main disadvantage of this algorithm is the centralized nature, it can provide a high throughput but destroys one of the key attributes of blockchain, decentralization, which is not acceptable for public blockchain networks.

2.3 Types of Blockchain Networks

There are different types of blockchain networks that can be defined in terms of how users can join the system and how it operates, we can find two main kind of networks and then a combination of them [7]:

- **Public and Permisionless Networks**: Theses networks offer a completely open access, anyone can read and create transactions without the approval of a higher power entity. Public blockchains are the most common ones because it meets all the requirements for a blockchain. Usually these are open source, anyone can get involved in the development, find bugs in the code and suggest new changes. It is also common that this networks have some incentivizing mechanism to encourage users to join the network.
  The main problems resides in privacy and scalability, while some networks try to encrypt and hide as much as possible sometimes it is not feasible (e.g. in Ethereum user A transfer tokens to user B, the second user doesn't know the identity of the first one, but if B look in the transaction for the sender and explore the wallet it is possible to see the balance and transaction made by the user A). Some

[3] The Byzantine General's problem creates a scenario where all the parties involved (in this case the Commander and its Lieutenants) must agree in a single strategy to avoid failure. Some of them might be corrupted and distribute false information. [10]

blockchain projects specialize in privacy, in cryptocurrencies these are known as Privacy Coins (e.g. Monero, DAPS Coin).

- **Private and Permissioned Networks**: Newer alternative to public blockchain to use the technology in a group of known participants. Access to the network is only possible if it is granted by an administrator or superior entity of the system, some actions may be restricted to specific participants. This kind of network is more attractive to business, they don't want their data publicly shared out in the world but they want to take advantage of the key points of blockchain. Usually these networks don't rely on consensus algorithm based on compute power or wealth of the participant (e.g. PoW, PoS) but a collection of trusted nodes can validate the transactions. This alternative provides a better scalability, similar to cloud computing, privacy control and a higher throughput. They are not open source. Many people believe that the concept of blockchain loses meaning in this kind of network because the decentralized feature is destroyed, the company that builds this network still a central entity with power over the whole network.

- **Hybrid Networks**: This combination of both types of network defined could be the best solution for many businesses, while keeping some of the key features of public blockchains (people can easily join the network) and other strong points of private networks (privacy control). Also known as consortium networks, this provides a way for different organizations to work together privately inside a blockchain network. While it is not completely private, it is semi-private. Each organization can set up their nodes infrastructure and join the the network with other entities, it helps to bring together business in the same sector.

  One of the most important companies working in hybrid networks is Hyperledger, we will be working with their tools and framework for this project, more information will be detailed in later sections.

Table 2.1 Summary of Blockchain types.

Property	Public	Private	Hybrid
Consensus determination	Miners	One entity	Nodes of multiple organizations
Read/Write Permission	Public	Restricted	Semi-restricted
Immutability	Impossible to tamper	Could be tampered	Could be tampered
Efficiency	Low	High	High
Centralized	No	Yes	Partially
Consensus Process	Permissionless	Permissioned	Permissioned

2.4 What uses does Blockchain have in the real world?

We have defined a lot of concepts about this technology, the advantages and disadvantages it brings and the different configurations that can be found. Now we will look through different real examples that uses this emerging technology.

2.4.1 Cryptocurrencies

Cryptocurrencies are the first application of this technology, although the first ideas around the concept of a cryptographically secured chain of blocks can be found in 1991 by Stuart Haber and W. Scott Stornetta [11], the real first application was created by Satoshi Nakamoto[4] in 2008 in his paper *Bitcoin: A Peer-to-Peer Electronic Cash System* [1]. Since the creation of Bitcoin, many cryptocurrencies have been borned trying to solve some of the problems that Bitcoin could not. The total market capitalization for all cryptocurrencies, at the time of writting, is $270 bn, at the same time, Bitcoin holds a 60% of that amount, being then the coin with the highest capitalization [12].

[4] It is unknown whether this refers to one person or a group of person. Up to this date, the identity of the creator of Bitcoin still unknown.

Table 2.2 Current ranking of popular cryptocurrencies [12].

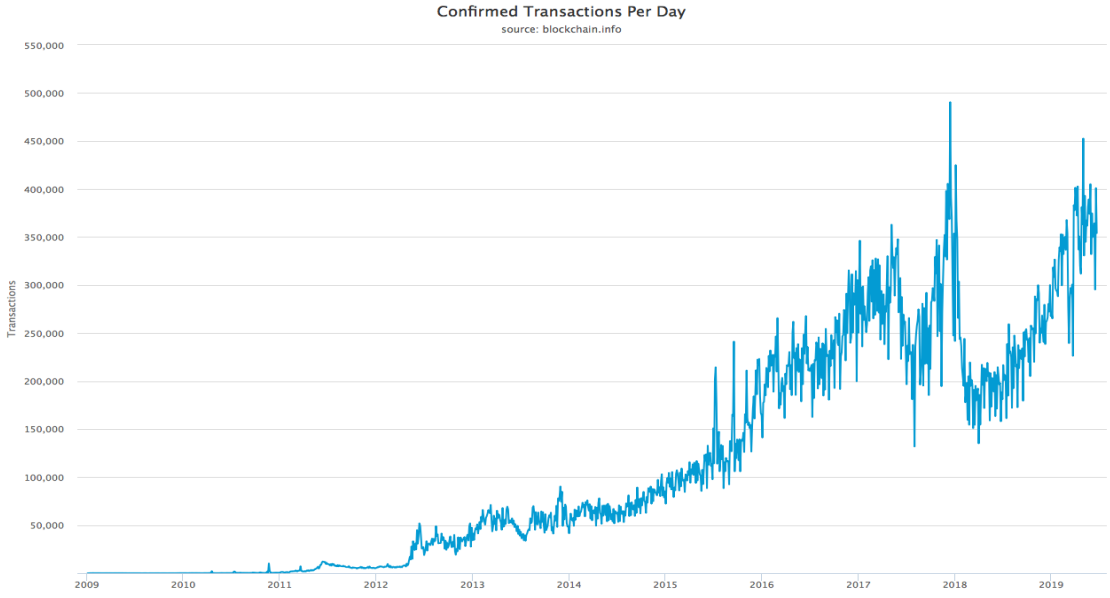
Name	Market Cap	Price	Circulating Supply
Bitcoin	$176 bn	$9,500	18,391,443 BTC
Ethereum	$26 bn	$238	111,170,930 ETH
Ripple	$9 bn	$0.20	44,166,596,173 XRP
Litecoin	$3 bn	$46	64,885,651 LTC

Although the main use for most cryptocurrencies is the ability to transfer the coins or money between users, other projects go one step further and build additional functionality to take advantage of the blockchain technology. For example, Ethereum is also a public distributed network based on blockchain featuring smart contract functionality. The token used in the network is the Ether, it can be transferred between users as a currency. A smart contract was first defined in the 1996 by the cryptographer Nick Szabo as "a set of promises, specified in digital form, including protocols within which the parties perform on the other promises", in this context it refers to a set of immutable computer programs that run in a deterministic way inside an Ethereum Virtual Machine as a part of the network protocol [13].

Many times, cryptocurrencies are used as a way to fund a new startup related with the crypto world. This way of crowdfunding had a huge impact in 2017 where over $6bn was raised in numerous ICO[5] [14]. The development teams get the funds and in return they give a new token to the users, this coin can be used in a speculative way with the only goal of selling it for a higher value than the initial investment, other times, that token will have a real use inside the project being developed.

[5] Initial Coin Offering

(a) Price.



(b) Confirmed transactions by day.

Figure 2.4 Bitcoin price and confirmed transactions history [12].

2.4.2 Supply Chain Management

Supply chain management includes all the different processes that convert raw material into ready to sell products trying to minimize the total cost. There are six components in the traditional supply chain management[15]:

- Planning: Managing all the resources required to created a product in an effective way.
- Sourcing: Choosing the provider of goods to created said product.

- Making: Organizing all the activities to transform raw goods into the final product, including quality testing.
- Delivering: Coordinating logistics, schedules and payments
- Returning: Processes to allow back unwanted or defective items.
- Enabling: Support and monitoring of all the processes involved.

In this scenario, blockchain allows to remove unneeded intermediaries in between the different components in supply chain management. Additionally it gives the ability for transparency and trust, improving the flow of information between all the participants. Unlike the cryptocurrencies, in this case a private networks is more correct since people outside the business should not have access to the data. This does not mean that there is only a single participant, but in order to form part of the network there would be a validation process.

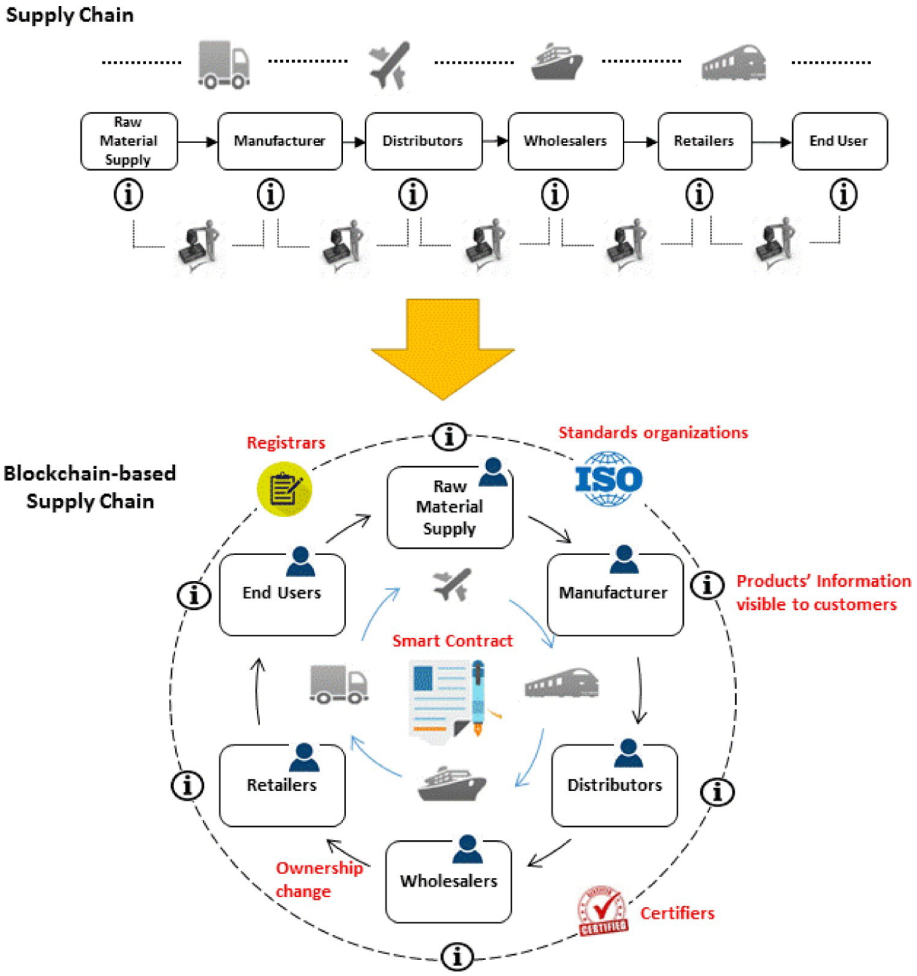


Figure 2.5 Supply Chain Network with Blockchain[15].

IBM Food Trust[16] is a real approach of this concept. It allows companies to easily include blockchain into their processes. Using this technology they can ensure end-to-end traceability, with food, for example, is crucial to be able to know who is affected and how to react if a food issue is reported. It allows the companies to show the end customer the journey of the item they are purchasing.

2.4.3 Internet of Things

The Internet of Things (IoT) is composed by any material object that is connected to the internet. This connectivity gives the possibility of getting information remotely out of the object or send actions to it. These activities do not need to be triggered by a person, other objects in the network can interact with each other. Blockchain gives security to these systems, the information is distributed along all the objects in the network instead of being in a central weak point. Companies like IBM are integrating blockchain within their line of IoT products, using the IBM Watson, enabling the devices to interact in blockchain transactions [17].

There are three main benefits that blockchain brings to IoT:

- Builds trust: Reduces the risk of collusion and tampering, creating trust between parties and devices.
- Reduces costs by removing middlemen and intermediaries.
- Accelerate transactions: Settlement time is reduced from days to nearly instantaneous.

2.4.4 Digital Identity Management

Identity is formed by a collection of attributes about a person such as their full name, date of birth, a national identifier, driver license, etc. The digital identity is currently handled by central authorities and users have no control over it, each time we have to identify ourselves our identity gets saved in the systems of banks, governmental institutions, credit agencies... and these represent weak points that can be breached. Blockchain attempts to reduce intermediaries, verify our identity without having to disclose it and give us the control to manage out identity. With blockchain there would be no central point that could be attacked in order to steal our identity thanks to the decentralization. [18][19]

2.5 Hyperledger

2.5.1 Introduction

Hyperledger is a project created in 2016 by The Linux Foundation, they main goal of this project was to bring the blockchain technology closer to businesses by providing a set of frameworks and tools (Figure 2.6) to facilitate the implementation. What started as a platform with a few founding members has now over 100 members with big companies supporting the project (e.g. IBM, Airbus, J.P. Morgan, BBVA).

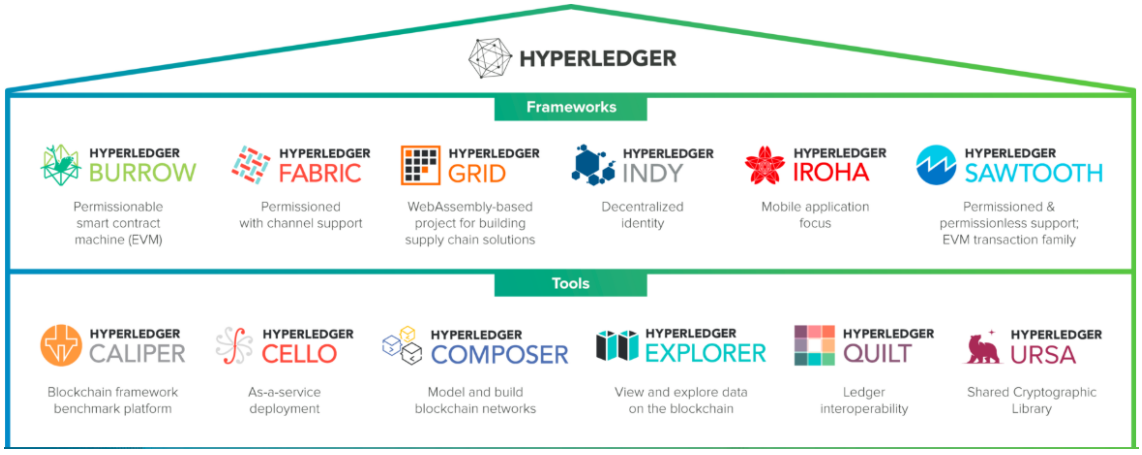


Figure 2.6 Hyperledger frameworks and tools [20].

With this strategy, Hyperledger incubates and develops a series of blockchain technologies, libraries, frameworks and applications. To mention some of the popular ones and others relevant to this project[20]:

- Hyperledger Fabric: Project developed by IBM, the goal is to allow companies to build a permissioned blockchain of their own that can quickly scale to over 1,000 transactions per second. It allows different organization to be connected inside a communication channel and the ability to have multiple channels in the same network [9].

- Hyperledger Composer: Tool for fast prototyping of blockchain business networks, smart contracts. This logic goes from creation of assets and relationships between assets and participants to creation of transactions and events. These smart contracts can then be deployed in a Hyperledger Fabric Network for participants to interact with them.

- Hyperledger Explorer: Tool designed to create a user-friendly web application to view the state of the blockchain network, number of blocks created, statistics about the transactions, etc.

- Hyperledger Sawtooth: Project developed by Intel, tries to bring modularity to blockchain, it is written in Python and the biggest difference with other frameworks inside hyperledgewr is that it supports both permissioned and permissionless networks. The consensus algorithm used is called Proof of Elapsed Time (PoET) which ensures a safe and random selection of a "leader", comparable to the miner in other Proof of Work algorithms [21].

- Hyperledger Indy: Project that provides a set of tools and libraries to help with identity-like application in blockchain. Decentralized identity is an uprising subject right now and this tool would help companies to implement it.

For this project, we have decided to use three frameworks and tools provided by Hyperledger:

- **Hyperledger Fabric** to create the blockchain network.

- **Hyperledger Composer** to develop the smart contracts and the main logic.

- **Hyperledger Explorer** to view data and statistics about the network and the interactions with the smart contracts.

2.5.2 Hyperledger Fabric

As we have explained, Hyperledger Fabric is one of the most popular frameworks from Hyperledger Suite with the main characteristics of being semi-private and permissioned. Instead of allowing unknown participants to the network, members have to be enrolled through a trusted Membership Service Provider (MSP). For this reason there is no need for algorithms like Proof of Work to validate transactions, removing the need also of a cryptocurrency to pay the miners that use their computing power for validation. In this section we will define the main concepts needed to understand how the infrastructure of a blockchain network built on Fabric works and lay then the foundation of the blockchain system that will be used in later sections.

In Hyperledger Fabric there is the concept of an asset, participants can interact with the attributes of said asset by submitting transactions to the network. These transactions are stored in a ledger which is distributed among the different nodes of the network. The ledger includes also the ability to do SQL queries for auditing. A node can be seen as a communication entity of the system, they are connected with each other as it is a decentralized network, there are three main types of nodes that can be found [22]:

- **Client node**: It is used by the application to submit transactions to the network

- **Peer node**: Used for synchronization of the blockchain data between participants and nodes in the network. There are three sub types of peer nodes:

  - General Peer: Distributes information within the organization, if any node is not updated and asks for the most recent data, these nodes will send over the needed data.

  - Anchor Peer: As general peers are only known inside the organization, there is a need for some peers to be known by outside organization, anchor peer take care of this task. If a new transaction is submitted to the network, the information is sent to the anchor peer of each organization and these will distribute it to general peers.

  - Endorser Peer: Peers marked as endorser have the obligation of approving or not a transaction submitted by the client node. When this peer receives the request to create a new transaction it will follow the next process:

    1. Validates the transaction, makes sure it has the correct certificates and roles of the sender.

    2. Executes the smart contract to simulate the outcome of the transaction but it does not update the ledger.

- **Orderer node**: Takes care of maintaining the state of the ledger across the organizations, it can be seen as the central communication channel for the network. When the node receives an approved transaction it will create a new block or insert it in an existing block, which will be sent to all the anchor peers in the network and then distributed within the peers of the organization. The orderer node can be a single node in the network, solo mode, or it can be formed by multiple nodes using Kafka[6]. Usually, the solo mode is used for development and to create prototypes since it is quite more simple than the kafka mode, which is used in production-like setup. When using solo mode the network has a single point of failure, so it is not recommended to use it for a real system.

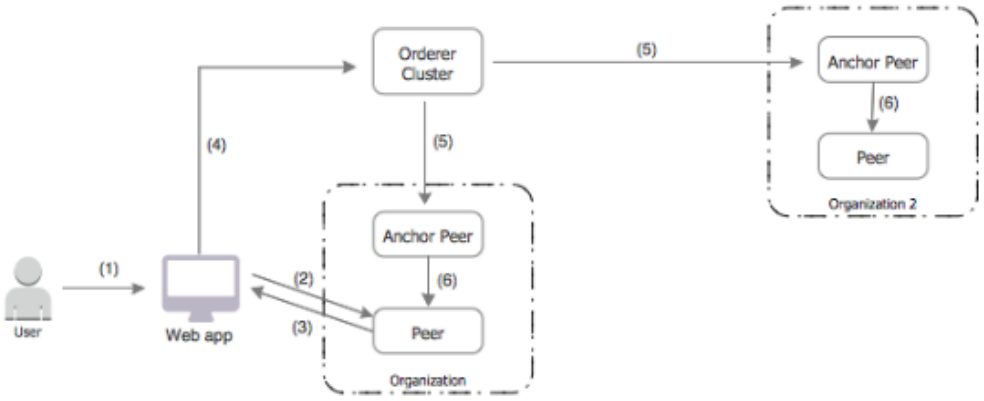


Figure 2.7 Process for creation and distribution of a transaction.

The process described in Figure 2.7 gives a better picture of what happens since the submission of a new transaction by a user until the new block reaches all the nodes in the network. Let's assume there are two organization in the network, Org1 and Org2. In this example, each organization has two peer nodes, one anchor peer and one general peer which is also an endorser. Then there is a cluster of orderer nodes working in kafka mode.

To give more context to Figure 2.7:

1. The user invokes a transaction request through the web application connected to the network via the hyperledger SDK or the REST Server.

[6] Kafka is a software developed by Apache that takes care of message distribution and queuing. It is horizontally scalable and fault-tolerant.

2. The application sends the transaction to one endorser peer.

3. This peer validates the certificates, simulates the chaincode and gives back the result.

4. If the transaction got approved, the application sends it to the orderer cluster for execution and block creation.

5. Orderer creates the block and sends it to the anchor peers on each organization.

6. Anchor peers broadcast the new block to all the peers inside the organization. All the peers are in synced now.

To manage identities in the network there is a Certification Authority (CA) that takes care of the certificates creation. Everyone that interacts in any way with the blockchain network needs valid certificates (e.g. users, peer nodes, organizations). Each accreditation is linked to a participant of the network and specifies the roles available for the participant.

Specific instructions on how to build a Hyperledger Fabric Network can be found in the section 4.

2.5.3 Hyperledger Composer

Hyperledger Composer is a tool that allows to quickly develop prototypes of chaincode to upload to a Hyperledger Fabric network, these prototypes are known as Business Network Application (BNA). The main goal of this framework is to make it easy and fast to develop blockchain applications. To build a BNA we need to create several files that define each section of the application, some files are written in a custom modeling language to simplify the way that assets and transactions are created. In order to form this BNA we will need to create [23] [24]:

- **Model File**: This file contains the definition of assets, participant and transactions involved in the blockchain application. It is written in a custom object-oriented modeling language that makes easy the development of these resources. The assets sets the object which can be created or altered, the participants are the agents that make those changes in an object and the transactions are the actions that trigger alterations in an asset.

  To define an asset we need to set the attributes that form the object, these can be primitive data types (e.g. Integer, String, Boolean) and relationships to other assets or participants, for example to say that some asset is owned by some user. This language also supports inheritance, abstractions and generic classes. To understand how an asset is built, we can look at this example extracted from the finished application:

Code 2.1 Definition of an asset.

```
asset Measure identified by measureId {
  o String measureId
  o String systemId
  o String consensusId
  o Integer subConsensusId
  o Double measure
  --> Agent agent
}
```

First we set *Measure* to be an asset identified by its *measureId*. Then we proceed to define the different parameters that form a measure. As it was said, there can also be relationships, in this case the measure will be linked to a participant, an agent, the one who took the measure. This participant is also defined in a similar way:

Code 2.2 Definition of a participant.

```
participant Agent identified by agentId {
  o String agentId
  o Double measure
  o Double reputation
  --> Agent[] inAgents
}
```

Similarly, to define a transaction, we need to set the parameters involved in the transaction and that will need to be filled by the user or participant:

Code 2.3 Definition of a transaction.

```
transaction add_measure {
  o String agentId
  o String systemId
  o Double measure
  o Integer subConsensusId
  o String consensusId
}
```

- **Transaction Functions**: Although the transaction, assets and participants are defined there is no logic that combines them yet. A Javascript file is needed to build the smart contract, when a transaction is called, assets are created or modified, this is done by the smart contract. This file receives the parameters set in the transaction definition and uses them to fill its purpose. There is a specific Composer SDK for Javascript with custom functionalities used to make queries, retrieve objects and make changes to these objects.

- **Access Control Rules**: Defines the rules over what can be done by each participant, it is built with a custom language, ACL (*Access Control Language*), that makes it really simple simple. Each rule need some parameters to be defined: which participant is the rule meant to, what kind of operation will allow/restrict (*WRITE,READ...*), which resources are involved and what kind of action (*ALLOW, DENY*). For example:

Code 2.4 Definition of a rule.

```
rule AllowAdmins {
    description: "Allow all Admin participants access to all
        resources"
    participant: "com.emilio.tfm.Admin"
    operation: ALL
    resource: "**"
    action: ALLOW
}
```

- **Query Definitions**: This file defines the queries allowed by the system. With another custom query language, we can easily define simple queries for the blockchain application. This language does not yet allow complex queries, only simple operators. For example:

Code 2.5 Definition of a query.

```
query selectAgent {
  description: "Select Agents based on agentId"
```

```
        statement:
            SELECT com.emilio.tfm.Agent
            WHERE (agentId == _$agentId)
    }
```

All these files are packed into a Business Network Application that will be inserted into a blockchain network, like the one defined in the Hyperledger Fabric section. Users with the correct credentials will be able to access to create assets, make transactions and read data from queries. The Figure 2.8 sums up the formation of a BNA.

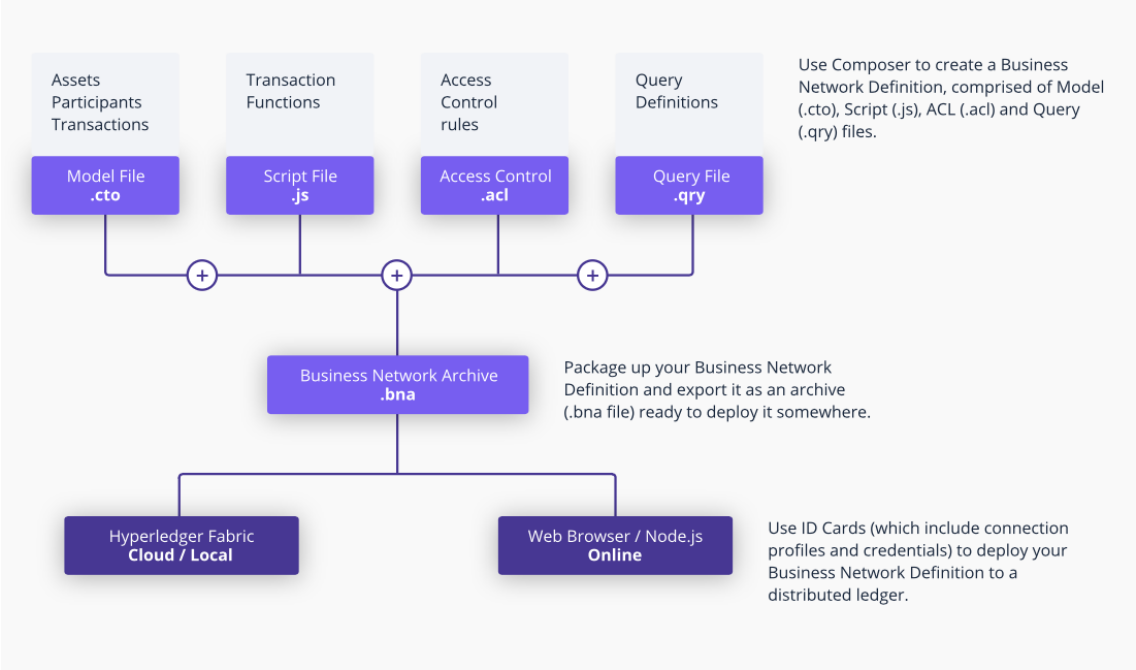


Figure 2.8 Architecture of a Business Network Application (BNA)[23].

2.5.4 Hyperledger Explorer

Hyperledger Explorer is a new project, still in incubation, designed to create a user-friendly interface for users to see blockchain data and statistics. We decided to use this tool to make it easy to view activity on the underlying blockchain network. It is a simple tool that connects to the Fabric network by settings the nodes built. An example of this tool can be seen in Figure 2.9.

As it can be seen, we can view the current activity of the network (e.g. blocks/hour, total transactions, status of the nodes).

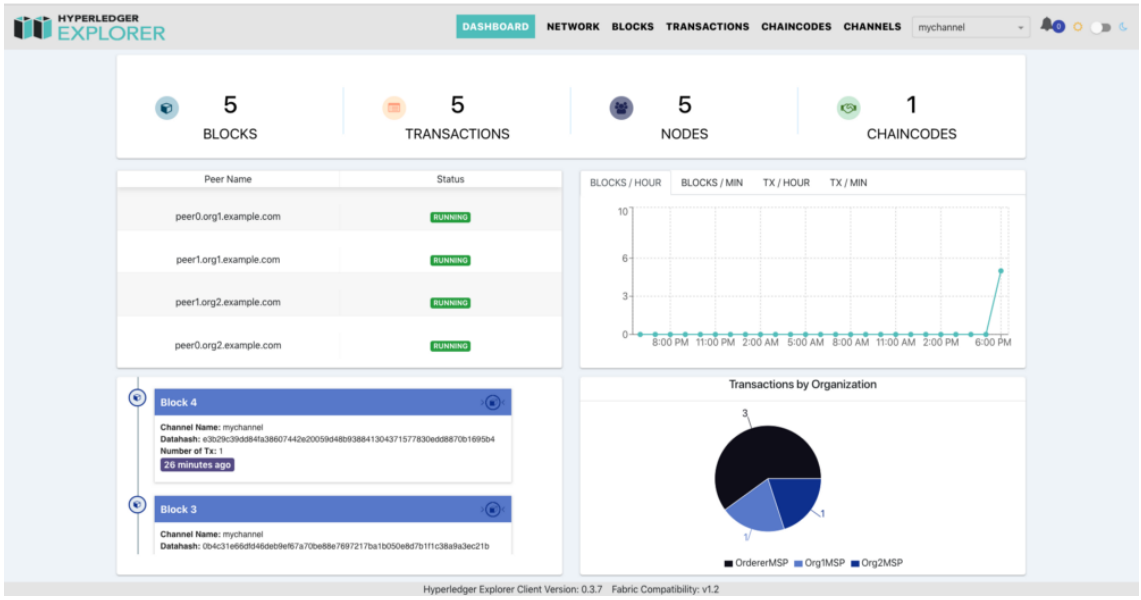


Figure 2.9 Example of Hyperledger Explorer.

3 Consensus problem

3.1 Introduction

With the latest technological developments in electronics, network communication and computation, a network of embedded system where the components can communicate with each other is a reality. From a cluster of sensors giving the most precise information about a system to a network of robotic vehicles sharing information for a seamless cooperation to solve a problem, many of these applications will start appearing in our daily lives.

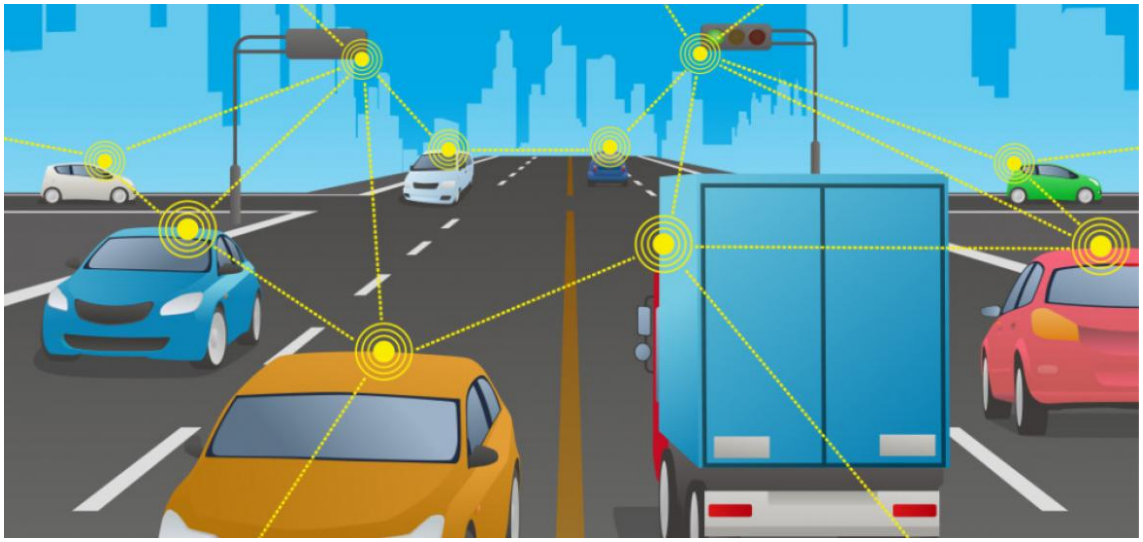


Figure 3.1 Representation of interconnected vehicles forming a cluster [25].

The main advantage of having multiple systems working on a problem in many situations is that while a single agent is not able to solve a difficult task, a group of agents have the ability to increase the computational power and provide robustness to failures, so the task is still solved even if a small number of agents fails or lose their communication. [26]

On the other hand, there are also main difficulties when using autonomous systems that communicate with each other since there is no central control point that makes sure that everything is working as expected. These problems open the possibility for research in the field of multi-agent systems. The problem that this project tries to solve is consensus one; the goal is to develop an application that helps agents of a network to reach an agreement based on the information they share.

The proposed solution consists on a series of consensus algorithms that will be analyzed to get convergence between agents as quick as possible. All of this will be done in a gossip communication system based on a blockchain network created with Hyperledger Fabric. The agents can also have restrictions for communication, so the system developed will make sure these access rules are followed. To avoid malicious agents trying to falsify information, modify the result of a decision and prevent the agreement, a reputation system will be implemented where any agent that does not follow the access rules, gives erroneous information or does not update its information correctly will be penalized by decreasing the reputation held. With this reputation scheme, malicious agents will not be taken into account by the rest of the network, it can be seen as an incentive to all the agents to play by the rules and have then power in the consensus decision.

3.2 Problem formulation

In this section we will detail the problem to be solved, what characteristics it has and in what cases it can be concluded as solved.

First, there is a need to define the notation to be used in the rest of this project when talking about the consensus problem. [27] [28] [26]

Let's define a network of $n>1$ agents represented by the digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{1, ..., n\}$ represents the set of nodes or agents. A directed connection from node i to node j is defined as (i, j) in $\mathcal{E} \subseteq \mathcal{V}x\mathcal{V}$. A connection from one node to itself is not allowed. The set of in-neighbors for the node i will be denoted as $K_i = \{j \in \mathcal{V} : (i,j) \in \mathcal{E}\}$

Definition of main graphs concepts:

- **Reachability**: Node i is reachable from node j if there exists a sequence of directed edges that connects node i to node j.

- **Strongly connected**: A digraph is strongly connected if every agent is reachable from every other node. Considering the digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and the subset of nodes $\mathcal{S} \subset \mathcal{V}$, $\mathcal{S}$ will be r-reachable if $\exists$i $\subseteq \mathcal{S}$ such that $\mid K_i \backslash \mathcal{S} \mid \geq r, r \in \mathbb{Z}$.

- **Robustness**: Measures the connectivity of a graph and how groups of nodes in a network are connected with each other. A digraph with two subsets $\mathcal{S}_1$ and $\mathcal{S}_2 \subset \mathcal{V}$ is r-robust, $r \in \mathbb{Z}$ if at least one subset is r-reachable.

- **(r,s)-Robustness**: Considering the digraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $n \geq 2$ nodes, define $r,s \in \mathbb{Z}$ such that $1 < s < n$. Now, define the set $\mathcal{X}_{S_m}^r = \{i \in \mathcal{S}_m :\mid K_i \backslash \mathcal{S}_m \mid \geq r\}$ for $m \in$ 1,2, where $\mathcal{S}_1$ and $\mathcal{S}_2$ are nonempty subsets of $\mathcal{V}$. $\mathcal{G}$ will be (r,s)-robust if any of the following conditions hold:

  1. $\mid \mathcal{X}_{S_1}^r \mid = \mid \mathcal{S}_1 \mid$

  2. $\mid \mathcal{X}_{S_2}^r \mid = \mid \mathcal{S}_2 \mid$

  3. $\mid \mathcal{X}_{S_1}^r \mid + \mid \mathcal{X}_{S_2}^r \mid \geq s$

At a given time $k \in \mathbb{Z}^+$ a node i will have the value of $x_i(k)$. $x(k)$ represents the array of states for the digraph at the time k.

Consensus, within a network of agents, is the agreement reached regarding a value of interest that depends on the state of the agents. The process that lead to this agreement being reached through the interaction and rules between agents and its neighbors is the consensus algorithm or protocol [29]. The theory around consensus in dynamic systems was first introduced by Bertsekas and Tsitsiklis in [30] and Olfati-Saber and Murray in [31]. The objective for any consensus algorithm is to update the value of each node with the information available from its connected neighbours. Consensus will be achieved once $x_i(k)$ converges.

3.3 Weight-Based algorithm

Next, we will define an algorithm well studied in the current literature [32] [33][26]. This is will serve us as a proof of concept for the blockchain network to be built in the next section and as a comparison for the proposed reputation-based algorithm.

Let's consider a digraph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ formed with n agents. For each agent i, we will define the group of *in-neighbours*, those agents with a directed edge towards i, as $N_i^+ := \{j \in \mathscr{V}; (j,i) \in \mathscr{E}\}$ and the group of *out-neighbours* as $N_i^- := \{h \in \mathscr{V}; (i,h) \in \mathscr{E}\}$.

In each iteration, the agent will send its state to the *out-neighbours* and will receive the state of the *in-neighbours*. With those states, the agent will update his state as follows:

$$x_i(k+1) = x_i(k) + \sum_{j \in N_i^+} w_{ij}(x_j(k) - x_i(k)) \tag{3.1}$$

where w_{ij} represents the updating weight and should satisfy $w_{ij} \in (0,1)$ and $\sum_{j \in N_i^+} w_{ij} < 1$.

3.4 Reputation-Based algorithm

There are several papers using the reputation approach for the consensus used within a blockchain network to decide who is responsible to decide the next block. In these scenarios, reputation defines the credibility of the user. In [34], depending on the reputation there are four distinguished trusted states that reflect more or less opportunities to be selected as a primary node. [35]

The following algorithm, proposed in this project, aims to remove some of the disadvantages of the others algorithm seen. This consensus algorithm focuses on keeping track of the behaviour of every participant in the network. With a reputation system, each agent has assigned a value from 0 to 100 marking its loyalty to the network. This reputation can also be defined as the trustworthiness of any agent given by the rest of the network. [36]

The participants must follow the consensus rules in order to reach an agreement, for each round of consensus those that do not follow this rules will get 30 points of reputation subtracted. If in this round the agent behaves normally, it will increase its reputation by 5 points, being 100 the higher limit. It's in the agent's best interest to follow the rules to keep the reputation as high as possible, serving this as the incentive.

What we want to achieve with this algorithm is that if an agent has low reputation, its value should not have an impact in the end result of the consensus. It is important to note that this algorithm would only be valid in systems where all the participants are equal.

In each round, each agent should update its value as follows:

$$x_i(k+1) = x_i(k) + \sum_{j \in N_i^+} w_j^*(x_j(k) - x_i(k)), \tag{3.2}$$

where w_j^* represents the updated weight related to the reputation,

$$w_j^* = w_{max} * r_j / r_{max}, \tag{3.3}$$

where w_{max} is the maximum weight that could be applied, r_j represents the current reputation for the agent i and r_{max} is the maximum reputation possible, 100.

If we write the Equation (3.2) with a matrix structure:

$$\begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \\ \vdots \\ x_i^{k+1} \\ \vdots \\ x_n^{k+1} \end{bmatrix} = \begin{bmatrix} 1-\sum_j w_j^* & w_2^* & \dots & w_i^* & \dots & w_n^* \\ w_1^* & 1-\sum_j w_j^* & \dots & w_j^* & \dots & w_n^* \\ \vdots & \vdots & & \vdots & & \vdots \\ w_1^* & w_2^* & \dots & 1-\sum_j w_j^* & \dots & w_n^* \\ \vdots & \vdots & & \vdots & & \vdots \\ w_1^* & w_2^* & \dots & w_j^* & \dots & 1-\sum_j w_j^* \end{bmatrix} \begin{bmatrix} x_1^k \\ x_2^k \\ \vdots \\ x_i^k \\ \vdots \\ x_n^k \end{bmatrix} \tag{3.4}$$

We can continue analyzing the convergence of this algorithm with a more simpler example. If we have 4 agents connected with each other, Figure 3.2, we can write the Equation (3.4) with numeric values, setting the Agent 1 as a possible malicious agent.

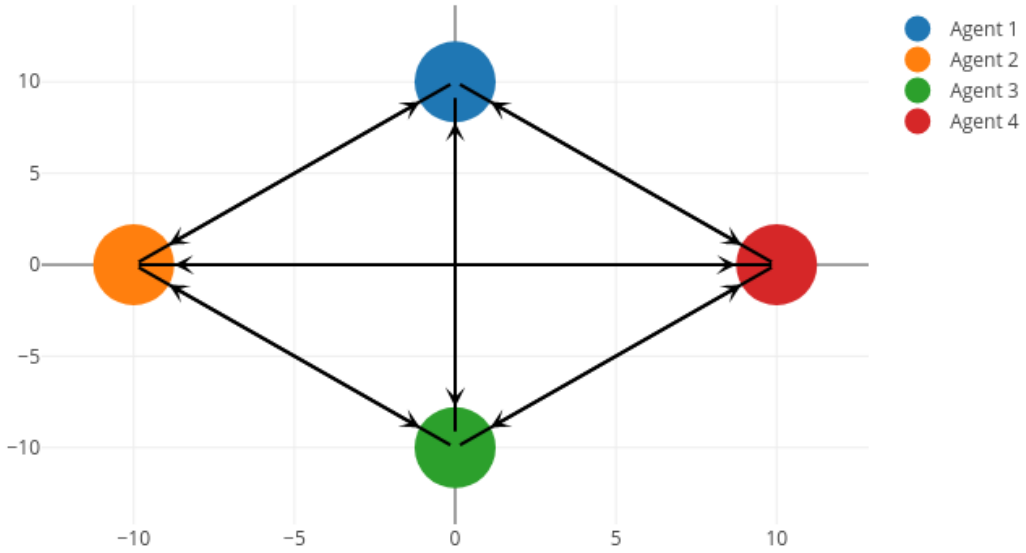


Figure 3.2 Network sample with 4 participants.

$$\begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \\ x_3^{k+1} \\ x_4^{k+1} \end{bmatrix} = \begin{bmatrix} 0.4 & 0.2 & 0.2 & 0.2 \\ w_1 & 0.6-w_1 & 0.2 & 0.2 \\ w_1 & 0.2 & 0.6-w_1 & 0.2 \\ w_1 & 0.2 & 0.2 & 0.6-w_1 \end{bmatrix} \begin{bmatrix} x_1^k \\ x_2^k \\ x_3^k \\ x_4^k \end{bmatrix} \tag{3.5}$$

We can find the eigenvalues for the weight's matrix to observe the convergence of the equation. First, if we set a scenario with no malicious agent, $w_1 = 0.2$, we calculate the eigenvalues, $|A - \lambda I| = 0$:

$$\lambda = \begin{bmatrix} 1.0 \\ 0.2 \\ 0.2 \\ 0.2 \end{bmatrix} \tag{3.6}$$

Since all the eigenvalues are inside, or in the limits, of the unit circle we can say that we will find convergence.

If we now set a scenario where the Agent 1 is acting maliciously, its reputation would decrease by 30 points in each round. We can find the eigenvalues for each of the reputations states. In $k = 0$ the result would be the same as the case we just studied.

$$A_0 = \begin{bmatrix} 0.4 & 0.2 & 0.2 & 0.2 \\ 0.2 & 0.4 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.4 & 0.2 \\ 0.2 & 0.2 & 0.2 & 0.4 \end{bmatrix}, \lambda_0 = \begin{bmatrix} 1.0 \\ 0.2 \\ 0.2 \\ 0.2 \end{bmatrix} \tag{3.7}$$

$$A_1 = \begin{bmatrix} 0.4 & 0.2 & 0.2 & 0.2 \\ 0.14 & 0.46 & 0.2 & 0.2 \\ 0.14 & 0.2 & 0.46 & 0.2 \\ 0.14 & 0.2 & 0.2 & 0.46 \end{bmatrix}, \lambda_1 = \begin{bmatrix} 1.0 \\ 0.26 \\ 0.26 \\ 0.26 \end{bmatrix} \tag{3.8}$$

$$A_2 = \begin{bmatrix} 0.4 & 0.2 & 0.2 & 0.2 \\ 0.08 & 0.52 & 0.2 & 0.2 \\ 0.08 & 0.2 & 0.52 & 0.2 \\ 0.08 & 0.2 & 0.2 & 0.52 \end{bmatrix}, \lambda_2 = \begin{bmatrix} 1.0 \\ 0.32 \\ 0.32 \\ 0.32 \end{bmatrix} \tag{3.9}$$

$$A_3 = \begin{bmatrix} 0.4 & 0.2 & 0.2 & 0.2 \\ 0.02 & 0.58 & 0.2 & 0.2 \\ 0.02 & 0.2 & 0.58 & 0.2 \\ 0.02 & 0.2 & 0.2 & 0.58 \end{bmatrix}, \lambda_3 = \begin{bmatrix} 1.0 \\ 0.38 \\ 0.38 \\ 0.38 \end{bmatrix} \tag{3.10}$$

$$A_4 = \begin{bmatrix} 0.4 & 0.2 & 0.2 & 0.2 \\ 0 & 0.6 & 0.2 & 0.2 \\ 0 & 0.2 & 0.6 & 0.2 \\ 0 & 0.2 & 0.2 & 0.6 \end{bmatrix}, \lambda_4 = \begin{bmatrix} 1.0 \\ 0.4 \\ 0.4 \\ 0.4 \end{bmatrix} \tag{3.11}$$

In all the rounds, while the reputation of Agent 1 decreases, we can observe how the convergence condition holds. When its reputations reached 0, we will stay in a state where the network can reach an agreement.

When a malicious agent applies a perturbation to its regular state, our system goes from $x = Ax$ to $x = Ax + Bu$, where B is a column vector used to apply the offset to the malicious agent and u is a numeric value that defines the perturbation. Following the example used for this demonstration, we can define:

$$B = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.12}$$

Then, we can proceed to analyze the model in each of its states. From $k = 0$ to $k = 4$ we know that the A matrix will be modified.

$$k = 0 \rightarrow x_1 = A_0 x_0 + Bu, \tag{3.13}$$

$$k = 1 \rightarrow x_2 = A_1 A_0 x_0 + (A_1 + 1)Bu, \tag{3.14}$$

$$k = 2 \rightarrow x_3 = A_2 A_1 A_0 x_0 + (A_2 A_1 + A_2 + 1)Bu, \tag{3.15}$$

$$k = 3 \rightarrow x_4 = A_3 A_2 A_1 A_0 x_0 + (A_3 A_2 A_1 + A_3 A_2 + A_3 + 1)Bu, \tag{3.16}$$

$$k = 4 \rightarrow x_5 = A_4 A_3 A_2 A_1 A_0 x_0 + (A_4 A_3 A2 A_1 + A_4 A_3 A_2 + A_4 A_3 + A_4 + 1)Bu, \tag{3.17}$$

For $k > 4$, the weights matrix will stay the same since the reputation of Agent 1 will stay at zero, for this example, then:

$$k = j, \forall j > 4 \rightarrow x_{j+1} = A_4^{j-3} A_3 A_2 A_1 A_0 x_0 + (A_4^{j-3} A_3 A2 A_1 + A_4^{j-3} A_3 A_2 + A_4^{j-3} A_3 + \sum_{i=1}^{j-3} A_4^i + 1)Bu \tag{3.18}$$

If we analyze similarly, the model without taking into account the reputation, $w_1 = 0.2 \; \forall k$:

$$k = j, \forall j > 0 \rightarrow x_{j+1} = A^{j+1} x_0 + (\sum_{i=0}^{j-1} A^i)Bu \tag{3.19}$$

With these equations we can now study how sensible the model is to changes in u. First, let's see the scenario without using reputation, our weights matrix would be:

$$A = \begin{bmatrix} 0.4 & 0.2 & 0.2 & 0.2 \\ 0.2 & 0.4 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.4 & 0.2 \\ 0.2 & 0.2 & 0.2 & 0.4 \end{bmatrix} \tag{3.20}$$

Continuing with the example of Figure 3.2, if we solve the system $x = Ax$ with no malicious agent and with the initial state x_0:

$$x_0 = \begin{bmatrix} 15 \\ 5 \\ 15 \\ 5 \end{bmatrix} \tag{3.21}$$

If we iterate through our system, we observe that by round 8 of consensus the agent's state is:

$$x_8 = \begin{bmatrix} 10.0 \\ 10.0 \\ 10.0 \\ 10.0 \end{bmatrix} \tag{3.22}$$

They have reached consensus as expected, this will serve as reference for the next scenarios.

Now we introduce the action of the malicious agent, Agent 1. It will have the following input:

$$u = \begin{cases} = 0, & \forall k = 0 \\ = 1, & \forall k > 0 \end{cases} \tag{3.23}$$

If we iterate through the Equation (3.19), we can observe:

$$x_{10} = \begin{bmatrix} 13.18 \\ 11.93 \\ 11.93 \\ 11.93 \end{bmatrix} \tag{3.24}$$

$$x_{100} = \begin{bmatrix} 35.68 \\ 34.43 \\ 34.43 \\ 34.43 \end{bmatrix} \tag{3.25}$$

So, even after 100 rounds of consensus, there is no agreement. Since the malicious agent keeps changing, the offset of the final state keeps growing in each round. We can see how each agent has a difference of 0.25 with their previous state.

Lastly, we can do the same experiment with our reputation-based algorithm, our weights matrix would be from Equation (3.7) to Equation (3.11). The different states that the model goes through while the reputation decreases are the ones defined from Equation (3.13) to (3.18). We can now look at different states:

$$x_{10} = \begin{bmatrix} 11.81 \\ 11.15 \\ 11.15 \\ 11.15 \end{bmatrix} \tag{3.26}$$

$$x_{100} = \begin{bmatrix} 11.81 \\ 11.15 \\ 11.15 \\ 11.15 \end{bmatrix} \tag{3.27}$$

As expected, Agents 2 to 4 converged and didn't take into account Agent's 1 changes after its reputation dropped to zero. The deviation of this solution compared to the reference model without malicious agent is 1.15. From round 10 to 100, the results stay the same since the consensus have been reached and Agent 1 has no effect over the rest of the network anymore.

Even though the reputation of an agent decreases while it doesn't follow the rules of the consensus, it is possible that the malicious agent had an impact in the agreement. Another advantage of using this algorithm is that if the consensus needs to be run for a second time, with the malicious agent at a null reputation, this

time the agent would have no impact. Reputations are saved over time and it is not a numeric value that holds only over the period of a consensus.

As we will see in the next chapter, the consensus will be executed inside a blockchain network, giving full traceability and the ability to audit the process. This reputation will be managed by the blockchain smart contracts. The algorithm of consensus will be implemented inside the logic in the network so when each participant submits the value for the round of consensus the smart contract is able to retrieve previous data and verify if the participant is following the algorithm as expected. If the offset with the expected value is higher than 1% the smart contract will decide that the agent is not following the rules and will decrease its reputation. Otherwise, it will increase its reputation.

4 Applications developed

In this chapter we will see the applications developed[1] for this project. First, there will be details about the blockchain model, what assets are involved, how are they related to each other, when are created, what transactions can be done by any agent, etc., in other words, the development of the smart contracts. All these assets and transactions in the form of smart contracts need to be inside a network in order to be useful, so then there will be a description of the different kind of infrastructures we can build with Hyperledger Fabric, we will go through the setup of the network, submission of smart contracts and the process of updating the configuration to allow further scalability in the future. Lastly, we will create agents to simulate the process of consensus, each agent will submit its measure and will try to converge with the rest of the participants.

With all of this, we will be able to test different formation and connection of agents, multiple algorithms and come to conclusions about what blockchain can or cannot do in this research field. Opening the possibilities for further research on a PhD.

4.1 Blockchain smart contracts

To build the smart contracts, we will use Hyperledger Composer to define a business network application, equivalent to the concept of a smart contract. The objective of this business network application, also known as BNA, is to provide a group of assets definitions, transactions and queries that can be consumed from a client application to interact inside a blockchain network. This way, an agent with the proper credentials can create an asset, interact with other agents in the network and view data from the queries responses, limited to what these credentials allow him to do.

In this section we will go through the development of the BNA for the specific use-case defined in this project. For this we need to set some technical specifications of what the business network application should achieve:

- When an agent is created, it should show the inside connections with its neighbours. At any moment the current measure and reputation of the agent should be visible (with proper credentials or connections).
- The user should be able to create a new system to which a group of agents would be linked to.
- At the start of a new consensus, a new asset should be created indicating what system it is related to, the round of the consensus (initially 0) and it should show at any time the status of the consensus.
- Each agent should be able to add a new measure to its round of consensus. After the first measure is submitted, when a new measure is submitted it should verify that the agent is following the algorithm according to the rest of the agents involved in the previous round.

[1] All the code will be accessible from the GitHub repository [37]

- Reputation of each agent should be updated after a measure is added, according to the result of the verification.
- Ability to consult data about the agents, system or consensus via queries to easily audit any consensus process.

If everything is designed as specified, we will have a complete application to which the client side can connect to to interact inside the blockchain network. This connection will be done with a REST API that will serve as a intermediary connection between the client and the network (e.g. to add a measure the agent could create a POST request to http://www.<network_url>/api/add_measure/ with the proper data)

To start with the development of the BNA we first need to setup and install the tools needed.

4.1.1 Setting up the development environment

The Hyperledger Composer pre-requisites can be installed in both Ubuntu and Mac OS. In this project, we will use Mac OS 10.14.5 for the local environment. The tools that will need to be installed in the system are:

- Node.js: a Javascript runtime tool.
- Apple Xcode: Apple's development tool, with the installation some additional components will be installed which are needed to run Hyperledger Composer.
- Docker: a tool that helps to create, deploy and run applications in containers. It helps to package up applications with all the libraries and packages needed for it to run in a single container.
- VSCode: coding editor that supports the installation of syntax packages to ease out the development of the modelling files. It will need the extension *Hyperledger Composer* installed.

Once this tools are installed we can proceed with the installation of Hyperledger Composer:

Code 4.1 Installation of Hyperledger Composer.

```
npm install -g composer-cli@0.20
npm install -g composer-rest-server@0.20
npm install -g generator-hyperledger-composer@0.20
npm install -g yo
npm install -g composer-playground@0.20
```

With these tools installed, the development of the modelling files can start.

4.1.2 Assets and participants

As we explained in the second chapter of this project, *Blockchain*, the modelling in Hyperledger composer is done with a custom language created by this company, the file format used is *.cto*. A business network application can be formed by multiple cto files, each one defined by a unique *namespace* (e.g. com.emilio.tfm), all the resources defined inside a namespace would inherit the namespace.

The resources include:

- Assets, Participants, Transactions, and Events: These can be seen as class objects in other common programming languages.
- Enumerated types: Used to specify the data type of an attribute that may have *N* known possible values.

- Concepts: Abstract classes that do not fall in the first category of resources. Concepts may extend the data type of an attribute, much like structurs.

There is only one participant in the network, the agents, identified by its *agentId*. From every agent we will need to know its most updated measure, its reputation and the connections with its neighbourhood.

Code 4.2 Declaration of an agent participant.

```
participant Agent identified by agentId {
  o String agentId
  o Double measure
  o Double reputation
  --> Agent[] inAgents
}
```

Where --> marks a relationship between agents, in this case, a relationship between the agent defined and all those with a connection towards the agent.

A set of agents work around the same system, an asset, identified by its *systemId* and knows which agents are linked to the system.

Code 4.3 Declaration of a system.

```
asset System identified by systemId {
  o String systemId
  --> Agent[] agents
}
```

When agents are interacting in a consensus to agree in a measure, we need to keep track of what is happening in the consensus. For this, we create a consensus asset, identified by its *consensusId* and will let us know what round inside of the consensus is next (*subconsensusId*), the status of the process and the agreed measure.

Code 4.4 Declaration of a consensus.

```
asset Consensus identified by consensusId {
  o String consensusId
  o Integer subConsensusId
  o String systemId
  o ConsensusStatus status
  o Double agreedMeasure
}
```

Since the status of the consensus is a set of known values, we use a enum for that:

Code 4.5 Declaration of ConsensusStatus.

```
enum ConsensusStatus{
  o INACTIVE
  o COMPLETED
  o PENDING
}
```

- INACTIVE: The consensus process has not started yet.

- COMPLETED: The consensus process has finished.
- PENDING: The consensus process is currently in process.

In order to keep a full record of the measures taken by each individual agent, we can create a last asset for them. This is identified by its measureId and has attributes about what agent is creating the measure, what system is it linked to and what specific round of the consensus is it for. This way, when auditting the network, we can see with precision all the measures recorded.

Code 4.6 Declaration of ConsensusStatus.

```
asset Measure identified by measureId {
  o String measureId
  o String systemId
  o String consensusId
  o Integer subConsensusId
  o Double measure
  --> Agent agent
}
```

For easier identification when looking at records, it is recommended to use formatted identifications for the assets, for example, in the case of the measures we decided to use the following format:

$$\{agentId\} - \{consensusId\} - \{subconsensusId\} \tag{4.1}$$

this way, in a quick look we know who took the measure and at what specific round of consensus, this information can be later used to do a query for a more detailed information.

Next, we will see how transactions are defined and how the logic is created.

4.1.3 Transactions

Similar to assets and participants, transactions need to be defined in the cto model file, but the logic itself will happen somewhere else.

To define a transaction we need to specify what will the input parameters be:

Code 4.7 Declaration of a transaction.

```
@returns(Integer[])
transaction add_measure {
  o String agentId
  o String systemId
  o Double measure
  o Integer subConsensusId
  o String consensusId
}
```

For a *add_measure* transaction to be called, the agent will need to pass its agentId, the latest measure and then information about the system and consensus. Transactions can also incorporate decorators like *@returns* to indicate more information. In this case, we define that the transaction will return an array of integers, more specifically, it will be two integers [agentId, deltaReputation]. Basically, when the new measure is added, the smart contract will verify that the agent is following the rules of the consensus or not, making an update for the agent's reputation in either case.

In the table 4.1 there is a summary of the different transactions implemented in the system. All of them follow the same format definition as the code snippet 4.7. It is important to note that this is just the definition of the transaction, there is no actual logic yet.

Table 4.1 Summary of transactions developed and their input parameters.

Transaction	Data Type	Variable Name	Description
add_measure	String	agentId	Main transaction. Creates a new Measure asset for the agent sending the transaction. Lastly it verifies the measure with the information of the previous round to decide the reputation delta.
	String	systemId	
	Double	measure	
	Integer	subConsensusId	
	String	consensusId	
setup_agent	String	agentId	Initializes a new agent.
	String	systemId	
	String	consensusId	
	Double	measure	
register_system	String	systemId	Initializes a new system.
start_consensus	String	systemId	Starts a new consensus for a system.
update_subconsensus_id	String	systemId	Sets the round of consensus to what is indicated by the subconsensusId
	Integer	subConsensusId	
	String	consensusId	
update_reputation	String	measure	Updates the agent's reputation applying the specified delta. Used internally
	Double	delta	

Once we have modeled the transactions in our system, inside the cto file, we need to implement the logic that will be executed when a transaction is called. This is done in a different file. For this, in Hyperledger Composer, we use a Javascript file called *logic.js*. Using the Composer SDK [23] we can access the network to check queries, create assets, update them and other actions.

To start writing the logic of any transaction we need to create a Javascript function with the same name as said transaction. In the code snippet 4.8 we start writing the main skeleton for the *add_measure* transaction. As we can see there is a header that explain some of the information of the transaction, such as a small description and the transaction route (with the namespace). The input parameters come encapsulated inside the transaction structure, so inside the function we can access these parameters to use it as we need to.

Code 4.8 Starting the logic of a transaction.

```
/**
 * Create Measure for Agent
 * @param {com.emilio.tfm.add_measure} transaction
 * @transaction
 */
async function add_measure(transaction) {
    const factory = getFactory();
    const namespace = 'com.emilio.tfm';

    const agentId = transaction.agentId;
    const systemId = transaction.systemId;
    const measure = transaction.measure;
    const subConsensusId = transaction.subConsensusId;
    const consensusId = transaction.consensusId;

    ...
    }
```

As it was said, inside this logic it is possible to retrieve network information from queries, create new resources and update them. The code snipped 4.9 show some of the useful things that can be done.

Code 4.9 Useful Hyperledger Composer SDK functions.

```
// Adding new resources to the network
var newMeasure = factory.newResource(namespace, 'Measure', formattedId);
newMeasure.agent = factory.newRelationship(namespace, 'Agent', agentId);
newMeasure.systemId = systemId;
newMeasure.consensusId = consensusId;
newMeasure.subConsensusId = parseInt(subConsensusId);
newMeasure.measure = measure;
const measuresRegistry = await getAssetRegistry(namespace + '.Measure');
await measuresRegistry.add(newMeasure);

//Getting data from queries
const consensusQuery = await query('selectConsensusForSystem', {'systemId':
     systemId});

// Updating existing assets
const agentQuery = await query('selectAgent',{'agentId':agentId});
let agent = agentQuery[0];
agent.measure.measure = measure;
await agentRegistry.update(agent)
```

Although most of the logic created in the transactions is straightforward and can be found for further detail in [37], it would be of interest to elaborate how the double verification system works. The objective here is to verify that the participants are following the rules set for consensus, when an agent adds a new measure, it should result from the update of its previous measure with the information available to him from its neighbour agents. Since the consensus will be using a known algorithm, for example the one defined in section 3.3, each agent should follow this equation.

But, what would happen if a malicious agent enters the system? It is possible that at some point a malicious agent gets into the system and threatens to alter the result of the consensus. This could be premeditated, one of the agents gets hacked by an external agent to try to modify the final decision for his own benefit or it could be caused by a malfunction on one of the agents. Either way, the information given from this bad actor to the rest of the working agents should not alter the result of the consensus.

For this, we have developed a reputation system, each time an agents adds a new measure to the network, the smart contract internally checks if the agent followed the set algorithm. The smart contract can launch queries to see data from the past round in the consensus, so given the equation of consensus it can check what the new value should be and compare it with the given measure. If there is an error greater than a specified threshold, the agent is either lying or malfunctioning, so its reputations should be lowered. If the measure is correct, the reputation would increase, with a limit on 100%. As it was mentioned in section 3.3, the update weight gets substituted by the reputation of each agent. This way, the lower the reputation of the agent, the less it affects the result of consensus. At the same time, the higher we set the penalty to reduce the reputation, the quicker the agent's data gets discarded, this is something to modify in the simulations to find the best possible settings.

Once we have developed the needed transactions we move into the queries.

4.1.4 Queries

Queries in Hyperledger Composer are written in a specific query language in *.qry* files. The query will return all the pieces of data in the registry that follows a set of rules or conditions specific for said query.

Table 4.2 List of queries available.

Query	Input	Description
selectAgent	agentId	Select Agents based on agentId
selectAllAgents	-	Select all agents
selectSystem	systemId	Select system based on systemId
selectMeasures	-	Select all Measures
selectMeasuresForConsensus	consensusId	Select Measure for a specific consensusId and subconsensusId
	subconsensusId	
selectAllConsensus	-	Select all the Consensus
selectConsensusForSystem	systemId	Select PENDING Consensus for a System
selectSubConsensus	consensusId	Select specific subConsensus
	subconsensusId	
selectAllSubConsensus	consensusId	Select all subConsensus in a Consensus

Each query is formed by:

- Description, a string that gives information of query's function.
- Statement, contains the rules for the query. It can use a combination of the following operators [38]:
  - SELECT is a mandatory operator, and by default defines the registry and asset or participant type that is to be returned.
  - FROM is an optional operator which defines a different registry to query.
  - WHERE is an optional operator which defines the conditions to be applied to the registry data.

- AND is an optional operator which defines additional conditions.
- OR is an optional operator which defines alternative conditions.
- CONTAINS is an optional operator that defines conditions for array values
- ORDER BY is an optional operator which defines the sorting or results.

Code 4.10 Examples of queries in Hyperledger Composer.

```
query selectAgent {
  description: "Select agents based on agentId"
  statement:
      SELECT com.emilio.tfm.Agent
      WHERE (agentId == _$agentId)
}

query selectConsensusForSystem {
  description: "Select consensus for a system"
  statement:
      SELECT com.emilio.tfm.Consensus
      WHERE (systemId == _$systemId AND status == 'PENDING')
}
```

In the code snippet 4.10, we can see two examples of the queries. Description gives a brief string about what is the objective of the query. The statement sets the rules to follow, the *SELECT* operator is used to choose which asset are we looking to get information from, with the *WHERE* operator we specify the conditions the asset should have to be in the output.

In the query *selectAgent* we look for information of the agent identified by the *agentId* specified as input.

In the second example, we look for the consensus of a *systemId* which status still *PENDING*.

The table 4.2 specifies all the queries built in the Business Network Application, many of them are used internally for the transaction logic.

4.1.5 Access Control Rules

In Hyperledger Fabric, being a permissioned network, it is possible to set restrictions to certain users of the network or specific roles to control what can each individual participant see and do. For example, we may not want agents to be able to delete the measures they have created, only a user with the role of administrator could execute that task.

These rules are defined in a *.acl* file with an specific language.

Code 4.11 Examples of Access Control Rules in Composer.

```
rule AgentCantDelete {
    description: "Don't allow agents to delete Measures"
    participant: "com.emilio.tfm.Agent"
    operation: DELETE
    resource: "com.emilio.tfm.Measure"
    action: DENY
}
```

```
rule NetworkAdminUser {
    description: "Grant business network administrators full access to user
        resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "**"
    action: ALLOW
}
```

Each rule is identified by an unique name and a set of parameters [39][24]:

- **Resource** defines the things that the ACL rule applies to. This can be a class, all classes within a namespace, or all classes under a namespace.
- **Operation** identifies the action that the rule governs. Four actions are supported: CREATE, READ, UPDATE, and DELETE. We can use ALL to refer to all actions stated.
- **Participant** defines the person or entity that has submitted a transaction for processing.
- **Action** identifies the action of the rule. It must be one of: ALLOW, DENY.

For the simulations that will be conducted in the section 5 we do not have very strict rules since it is a controlled environment, but for a production-like setup it is very important to be concise with the access control rules to avoid data getting in the hands of bad actors.

4.1.6 Creating the BNA

Now that we have implemented all of the files needed in a BNA, see figure 2.8, we are ready to package all the information in a *.bna* file to deploy it in a Hyperledger Fabric network.

To make this *.bna* package is quite simple with the Hyperledger Composer tools we installed at the beginning of this section. From the terminal console inside the project folder [37] we just need to run a command that would archive all the data in a single file:

Code 4.12 Creating the BNA package.

```
# Create distribution folder if doesn't exist
mkdir ./dist

# Create BNA
composer archive create --sourceType dir --sourceName . -a ./dist/tfm.bna
```

The file *blockchain.sh* inside the *Development* folder in the project [37] would help to automate many of the tasks like the BNA creation and more we will see in the next section. This shell script was created to make it as quick as possible to create the blockchain network and install and upgrade the BNA.

4.2 Blockchain network infrastructure

When building a Hyperledger Fabric blockchain network to deploy the previously created business network application we need to take into account multiple factors to decide the configuration of the network. It is not the same to build a blockchain network to be used to test a proof of concept, small number of transactions and users, than to build a large scale network to be used worldwide with thousands of participants.

In this section we will build the most simple of the networks for pure testing and then we will go through the process of setting up a more complex network simulating a production-like scenario.

4.2.1 Testing network

Hyperledger Fabric provides a simple and ready to use network to skip most of the configuration steps and save time in development [40]. This way, we can get directly into building the BNA and testing it in a blockchain network.

Although we will see the different steps needed to get the network running, in the Github repository [37] there is a shell script, *blockchain.sh*, that automates all of this. There will be references to what command should be used in each step, e.g. **blockchain.sh <command>**.

First, we need to install all the Hyperledger Fabric tools. For this, we will clone the Fabric repository, select the version of Fabric we want to use and download the needed binaries, **blockchain.sh –installTools**.

Code 4.13 Installing Hyperledger Fabric.

```
mkdir ./fabric-dev-servers && cd ./fabric-dev-servers
curl -O https://raw.githubusercontent.com/hyperledger/composer-tools/
    master/packages/fabric-dev-servers/fabric-dev-servers.tar.gz
tar -xvf fabric-dev-servers.tar.gz

cd ./fabric-dev-servers

# We will be using Hyperledger Fabric 1.2
export FABRIC_VERSION=hlfv12
./downloadFabric.sh
```

Once everything is installed, we can easily start the network. In order to be able to interact with the network we need to have the proper credentials. In Hyperledger Composer, we have the concept of Business Network Cards, these cards hold the necessary information for connecting with the Fabric network. Only someone with a valid card can interact with the network. Not all the cards have the same privileges, some are just to create new transactions and make queries while others have the ability to modify the configuration of the network, deploy new BNA and upgrade the logic. When starting the network, we will need a card with enough credentials to interact directly with the peers to deploy the previously built BNA.

Code 4.14 Starting Hyperledger Fabric network.

```
cd ./fabric-dev-servers
./startFabric.sh
./createPeerAdminCard.sh
```

With a Fabric network started and a BNA already built, we need to deploy the BNA to the network and then start it. When deploying, we are sending a copy of the BNA to each peer in the network, in this simple example there is only one peer but in a production-like scenario there could be many peers in different machines. **blockchain.sh –createNetwork**

Code 4.15 Deploying and starting the BNA.

```
# Deploy the BNA to the peers
composer network install --archiveFile dist/tfm.bna --card PeerAdmin@hlfv1

# Start the BNA
```

```
composer network start --networkName tfm --networkVersion 0.1.1 --card
    PeerAdmin@hlfv1 --networkAdmin admin --networkAdminEnrollSecret adminpw
     --file adminNetwork.card

composer card import -f adminNetwork.card
composer-rest-server -c admin@tfm
```

When the network is started, an admin card is created to interact with the BNA as any user would. With this card we can also start the rest server API, this is really helpful to interact with the network as we would with any API, it allows us to make GET and POST requests that would create new transactions in the network or retrieve data from queries. This will be specially used when we try to simulate a scenario where multiple agents need to interact with the network, they will do it through the API. Figure 4.1.

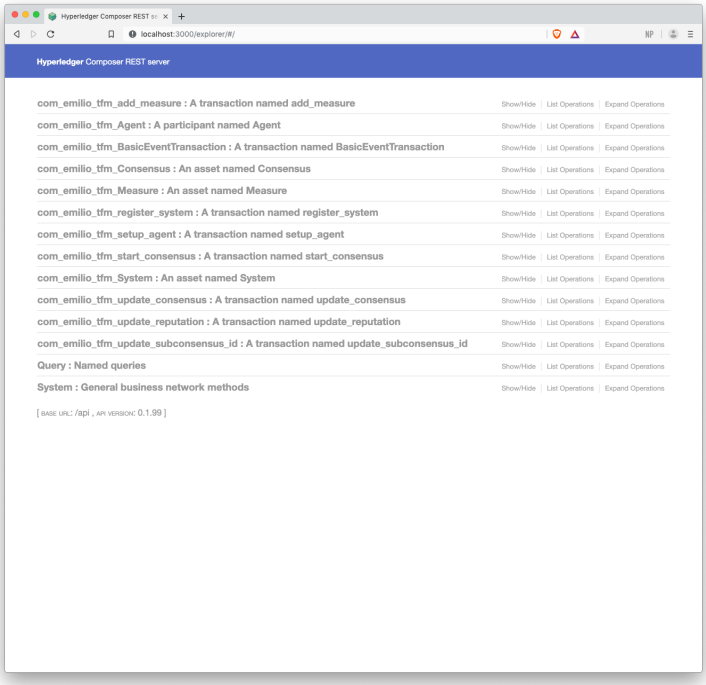


Figure 4.1 Composer Rest Server.

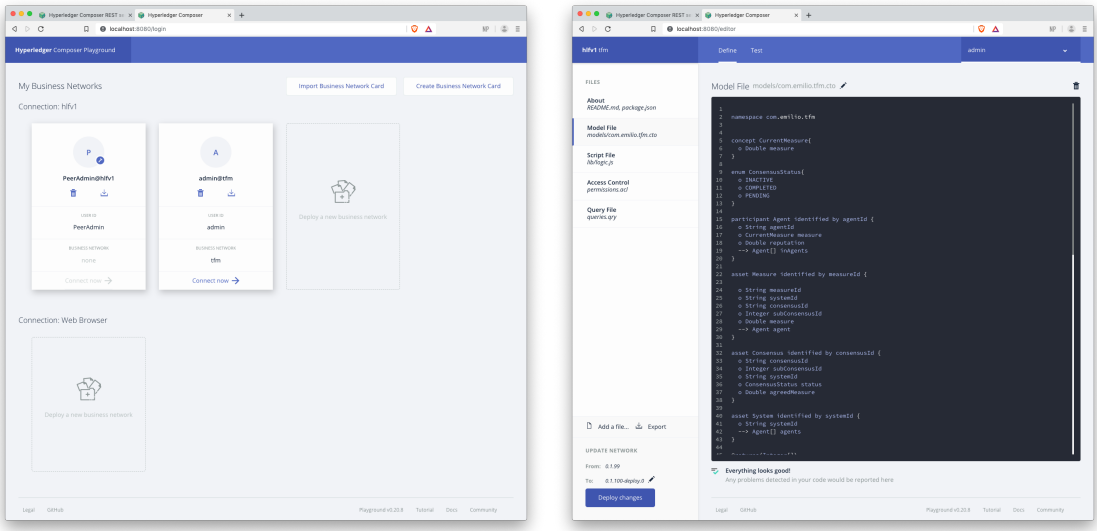
We can use this interactive UI, submitting request directly from the interface, or we can do it from any client by making the request to the proper url. For example, if we would want to make a query that returns all the agents in the network, we could make a GET request to the url *http://localhost:3000/api/queries/selectAllAgents* and this would return a JSON response with all the agents.

Code 4.16 Example of selectAllAgents query.

```
[
  {
    "$class": "com.emilio.tfm.Agent",
    "agentId": "1",
    "measure": {
      "$class": "com.emilio.tfm.CurrentMeasure",
      "measure": 20
    },
    "reputation": 100,
```

```
    "inAgents": [com.emilio.tfm.Agent#2]
  },
  {
    "$class": "com.emilio.tfm.Agent",
    "agentId": "2",
    "measure": {
      "$class": "com.emilio.tfm.CurrentMeasure",
      "measure": 25
    },
    "reputation": 100,
    "inAgents": [com.emilio.tfm.Agent#1}]
  }
]
```

Another useful tool provided by the Hyperledger Composer project for testing is the Composer Playground. It allows us to interact with the network, see transaction logs and even update the logic code. To run this tool we can simple execute in a terminal console: *composer-playground*. When we go into the playground, Figure 4.2a, we can choose the set of credential, the business network card, we want to connect to the network with. Once we are connected we can either update the current code explained in the previous section, Figure 4.2b, or test it out, Figure 4.3.



(a) Choosing Business Network Card for the playground.

(b) Code editor in Composer playground.

Figure 4.2 Hyperledger Composer Playground.

In the section Consensus integration we will see how we can connect external agents to the network so they can interact and send new transactions.

4.2.2 Production network

In this next subsection, we will see how we can customize the Fabric network to the needs of any project. The main goal of this part of the chapter is to see how to make the changes to add more complexity in the network. The default blockchain network, previous chapter, has a single organization and only one peer node, in a real world example there's always a need for multiple nodes to maintain decentralization and to avoid having the connection down if one of the nodes fail. We will see the most important sections of the configuration to

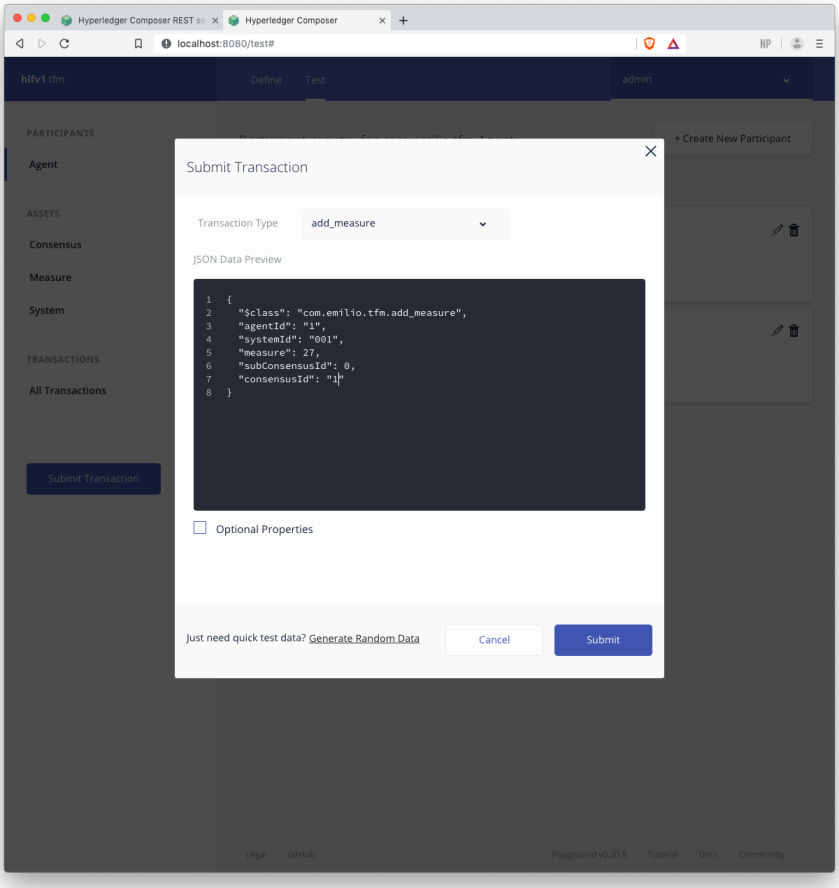


Figure 4.3 Composer playground, sending a new transaction.

accomplish a network with the following attributes:

- Two organizations to understand how organizations speak with each other and at the same time to have a higher number of nodes.
- Each organization will have 2 peer nodes, one of them being an anchor peer, and a certification authority node to manage the credentials.
- Updated block parameters for faster transaction and a higher throughput of them.

This setup could be done either in multiple servers, for example in AWS, or locally for testing with real world configuration. To simplify, in this project we will be setting up said configuration locally in a single machine, although we will understand the changes that would need to be done in order to adapt the network for the cloud. A great number of resources have been used to compile information about how to properly setup a real Fabric network [22] [41] [42] [24].

All the files mentioned in this section will be stored in the Github repository [37] inside the folder *Production*. Some of the files are configuration files and others shell script to automate the process. We will not go into detail of each independent command executed but the general objective of the scripts.

Identity Management

As discussed before, only certain users have access to the network configuration. We will be using the Certification Authority in Hyperledger Fabric to create the credentials for administrator and the nodes. The Certification Authority or CA consist on a server and a client:

- CA Server: Service used for certificate creation and management, it allows to renew and revoke identities too.
- CA Client: Utility to interact with the CA Server.

When a new identity wants to join the network, the process that will need to be followed is:

1. Registrar[2] registers a new identity and generates ID + Secret.

2. The user enrolls with the ID + Secret provided and generates the certificate.

In the case of the actual nodes, the registrar itself is the one that registers the new identity and enrolls it.

The files that hold the configuration for the CA Server and the CA Client are *fabric-ca-server-config.yaml* and *fabric-ca-client-config.yaml*, respectively, inside the config folder.

To quickly set up the identities needed for the proposed network we can simpy run the script *tfm-fabric-ca.sh* which will take care of all the needed steps, without getting in too many details, the process would be:

- Initializes and starts the CA Server.
- Generates the credentials for the admin and enrolls him.
- Creates the certificates for the two organizations.
- Sets up the credentials for the orderer node and enrolls it.
- Generates the certificates for all 4 peers (two in the TFM organization and the other two in the support organization) and enrolls them.

Each individual script can be found in the same folder with the specific commands used for generation and enroll.

Orderer node setup

As discussed before, the orderer takes the new transactions submitted, packs them in a new block and distributes it to all the anchor peers. In order to start the orderer node, we need to create the genesis block and the channel transaction file.

The configuration file *configtx.yaml* is used to create the network artifacts, this file has three main section worth mentioning:

- Organizations: Lists all the organizations that are part of the network. It includes the Name, ID, Directory for the MSP, Policies and the address for the anchor peers.
- Orderer: Configuration for the orderer. Type (solo or kafka), address for listening, policies[3] and block configuration. It also contains the anchor peer address of each organization, known to the network. The block configuration sets the rules for when a new block is created, depending on: time passed since last transaction, maximum number of transactions allowed in a block or a size limit.
- Profiles: Helps to automate some processes. We will have a profile for the Genesis Block and other for the Channel Tx file.

By default the block configuration is:

[2] Admin with registrar role

[3] Policies dictate what changes can be made, who can make them and how

Table 4.3 Default block configuration.

Parameter	Value	Description
BatchTimeout	2 seconds	The amount of time to wait before creating a batch
BatchSize - MaxMessageCount	10	Controls the number of messages batched into a block Number of transactions per block
BatchSize - AbsoluteMaxBytes	98 Mb	Controls the number of messages batched into a block Maximum block size

Inside the folder *orderer/bins* the script *tfm-orderer.sh* will:

1. Create the genesis block; *tfm-genesis.block* in *artifacts* folder.

2. Generate the channel transaction file; *tfm-channel.tx* in *artifacts* folder.

3. Start the orderer node, running on port 7050.

The channel transaction file will be used to create the channel used by the peers for comunication. Each peer will need to join the channel in order to sync and participate in the network.

Peer nodes setup

The process of setting up the peers include:

- Creating the channel in the network, this only happens once when the network is created. Other peers will join that channel to participate in the network. This process has to be done by the admin of one of the organizations.
- Start the anchor peers and make them join the network. The anchor peers are known to all the members in the network as they are defined in the *configtx.yaml* file and then embedded in the genesis block.
- Start the regular peers and make them join the network.

creating the channel in the network, starting the peers and joining the channel. The configuration of each peer is defined in the configuration file *core.yaml*.

By running the script *tfm-peers.sh* in the folder *peers/bins* the whole process described will happen. The script takes care of setting the environment variables needed for each peer and running the commands.

Peers that are not defined as anchor are used for scalability, performance and availability. They are not know to other peers on the network and they can be added or removed without network changes. The anchor peer works as a bootstrap peer so they can participate in the network communication. When a regular peer launched it will connect to the anchor peer and update its data to sync with the rest of the network.

Up to this point, we have built a complex network formed by multiple organization, identities are controlled by a Certification Authority and each organization owns multiple peers. The last step in building the infrastructure is to install and start the BNA developed in the previous section to the network. This is done the same way as explained before, submitting the packaged smart contracts to the network with the proper Business Network Cards.

4.3 Consensus integration

The blockchain network is now fully deployed and it is running the developed smart contracts. It is accessible through the API rest server, so the application developed in this section can make POST and GET request to

the API and interact with the network in an easy way.

The goal of this section is to make really simple how virtual agents interact with the network in order to be able to create different scenarios with a variable amount of agents, their connections and the initial measure of each agent. With all of this, we lay the groundwork for the next chapter where we will conduct experiments trying to see the benefits of using the new technology discussed in this project.

The code for this section is divided in three main files, inside the Consensus-Integration folder in the *Consensus-Integration* folder inside the project [37]:

- **system.py**: The system will be initialized based on a series of parameters such as the number of agents and how they are connected with each other. Then, with this information, it will create a blockchain object that will serve as an intermediary between the rest server and the client. Lastly it will make automatically graphs to display the progress of the consensus and the agent's structure, Figure 4.4 ,with the *plotly* library[43].

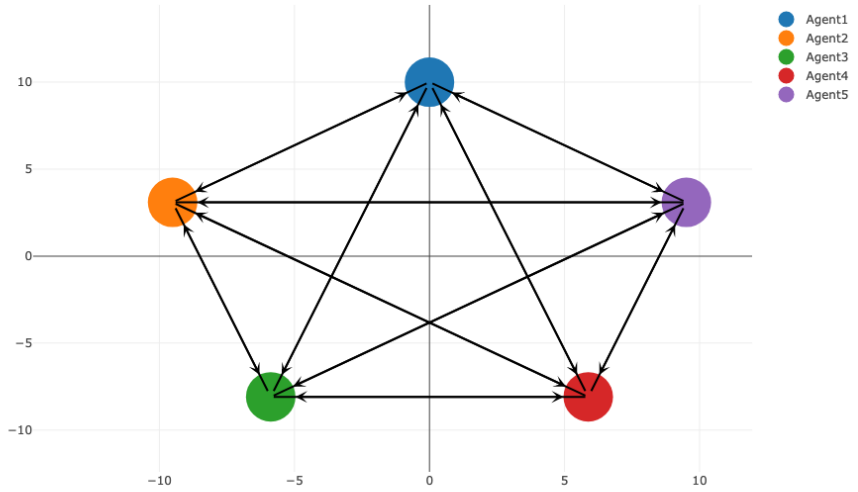


Figure 4.4 Sample of a 5 agents structure.

- **blockchain.py**: Consist of a blockchain class that gets initialize by the system client, it allows the client to call functions that will make GET or POST requests to the blockchain network. Each of these methods receive parameters as input that get then packed into a data variable to pass through the HTTP request. The status code in the response gets analyze to make sure that it was succesfull, Table 4.4.

Table 4.4 Most meaningful status code .

Status Code	Definition
200	OK
401	Not Authorized
403	Forbidden
404	Not Found
408	Request Timeout
500	Server Error

The status code is then analyzed, if the request didn't get completed successfully there will be another try. The blockchain class also is who call the consensus class to apply the specified rules.

- **consensus.py**: The consensus class implements different kinds of consensus algorithms. In the blockchain class we can select which algorithm we want to apply. When calling the consensus method, each agent will update the previous measure trying to get into agreement with the rest of agents.

In the next chapter we will put all the developed applications to test trying to simulate multiple scenarios where there might be one or more bad agents behaving differently, also we will be able to test different structures defining the number of agents and how they are connected to each other.

5 Simulations

Following the development of the applications, we illustrate the benefits of using a blockchain reputation-based consensus through numerical examples. We will set up different scenarios in which there will be changes in the network of agents used, the behaviour of the malicious agent or agents and the kind of consensus being used. With all of this information, we will be able to get conclusions knowing the pros and cons of using the proposed reputation-based consensus algorithm.

For the different configurations, we will be using two kind agent's structure:

- Strongly connected: A digraph is strongly connected if every agent is reachable from every other node, Figure 5.1.
- (2,2)-robust: Figure 5.2

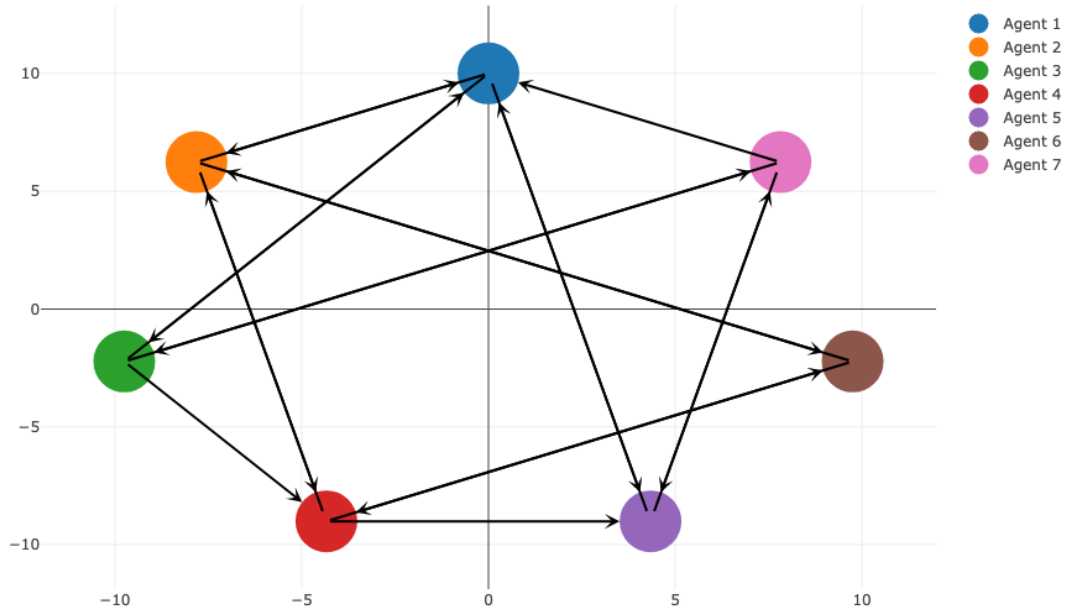


Figure 5.1 Strongly connected graph with 7 agents.

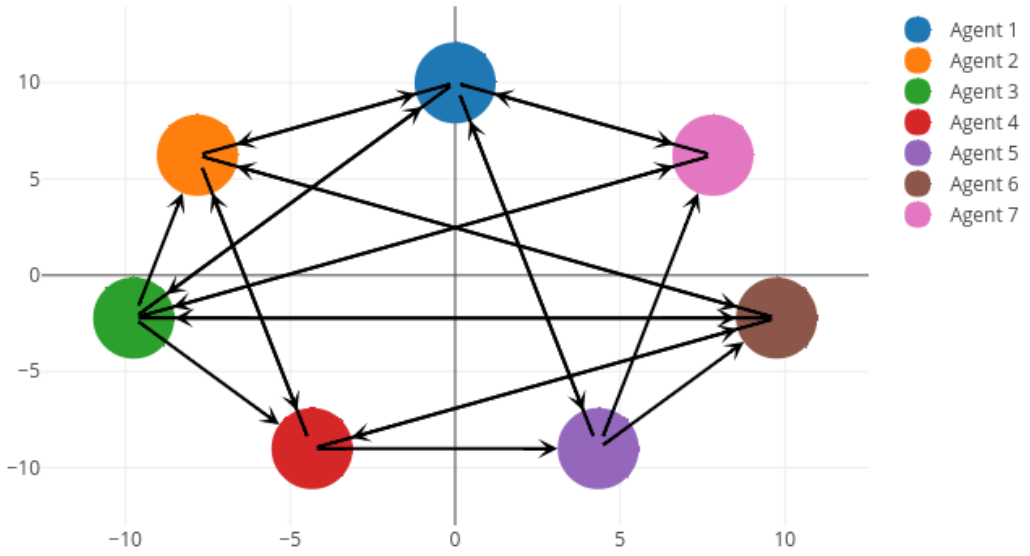


Figure 5.2 Robust graph with 7 agents.

There are three experiments that will be conducted around agent's behaviour:

1. No malicious agents present in the network.
2. There will be one malicious agent that will not be following the rules of consensus and will try to prevent the consensus to be reached.
3. There will be malfunctioning issue in one of the agents creating a deviation in the end consensus.

With each of these scenarios, we can find three main simulations that should be done:

- Average consensus: Each agent calculates the average of its in-neighbours measures.
- Weight based algorithm: Each agent uses the same weight parameter to get the calculation based on its measure and its in-neighbours measures.
- Reputation based algorithm: Each agent has linked an internal reputation that varies depending on the behaviour of the agent in the network. If it follows the rules the reputation will stay high, if not it will drop. Every agent will use each in-neighbour measure and reputation to calculates its next measure for the next subconsensus round.

Through the following experiments, we assume that the initial state of the agents will be:

$$x(0) = \begin{bmatrix} 5 & 15 & 5 & 15 & 5 & 15 & 5 \end{bmatrix} \tag{5.1}$$

For the second experiment, when any malicious agent comes into action it will alternate its state in each subconsensus round, going from one limit $x_i(2m) = 5$ to the other $x_i(2m+1) = 15$ where $\forall m \in Z_+$. In these cases, we will avoid printing the bad agent's result into the graphs as it tends to change frequently.

For each experiment we will conduct a total of six simulations (three types of consensus in two network graphs).

5.1 Experiment 1: No malicious agents

In this first experiment, we want to build a scenario where there are no malicious agents in the network, everyone will follow the set of rules specified. This will serve as reference for the following simulations where good and bad agents coexist in the same network.

5.1.1 Strongly Connected Network

Following the graph defined in Figure 5.1

Average Consensus

As defined, the agent will calculate the average between its measure and its in-neighbour's state.

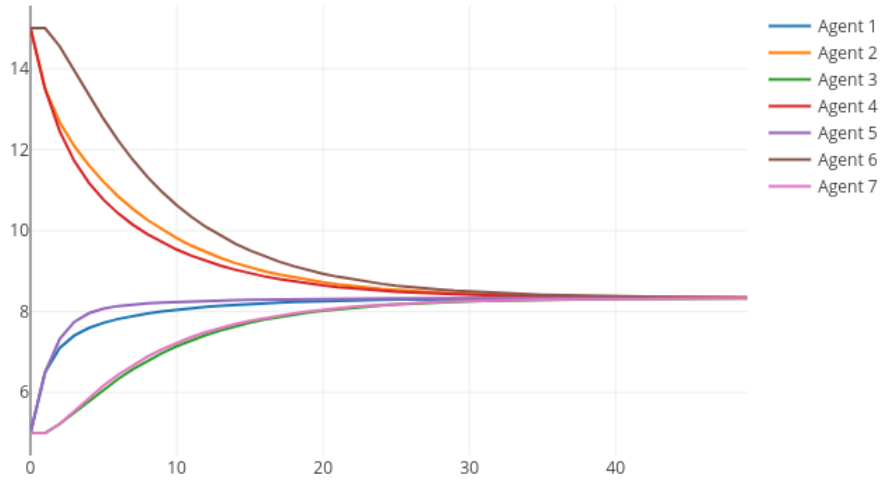


Figure 5.3 Average Consensus with no malicious agents.

We can observe in Figure 5.3 how the agents 1, 2, 4, 5 update in a more rapid way into the agreed consensus due to a greater amount of in-neighbours, they have more information than the rest of the network. The agreed state for this consensus is **8.340** and it took 49 subconsensus rounds.

Weight-Based Consensus

This simulation is based in the weight based algorithm in Section 3.3. We will assume that every neighbour has the same weight, which will be modified in multiple simulations. The next graphs shows a weight-based consensus with a fixed weight of 0.15, 0.2, 0.25.

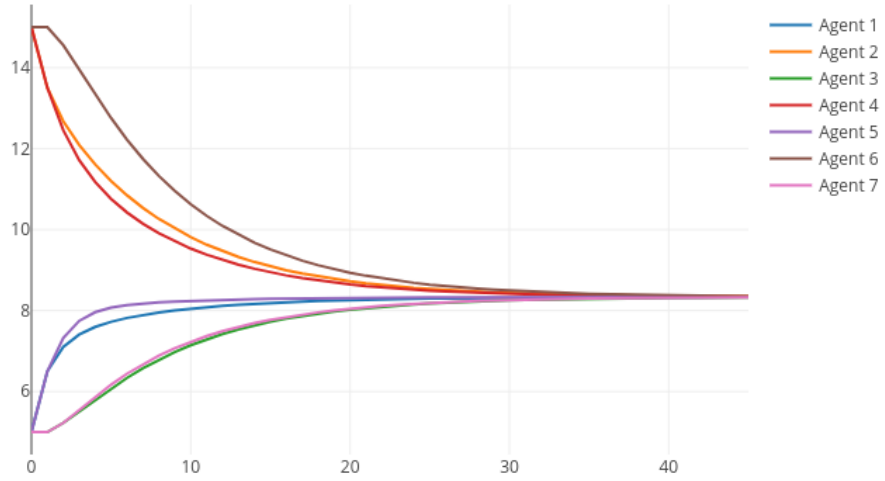


Figure 5.4 Weight-Based Consensus with no malicious agents, weight=0.15.

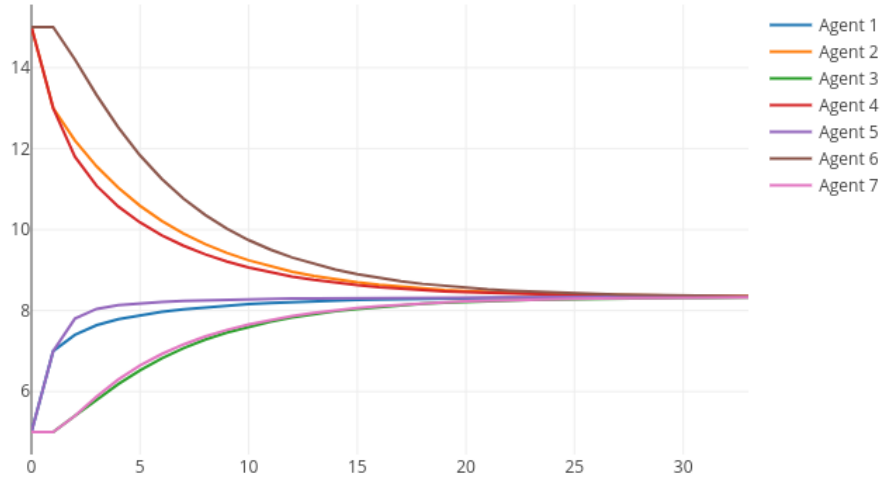


Figure 5.5 Weight-Based Consensus with no malicious agents, weight=0.2.

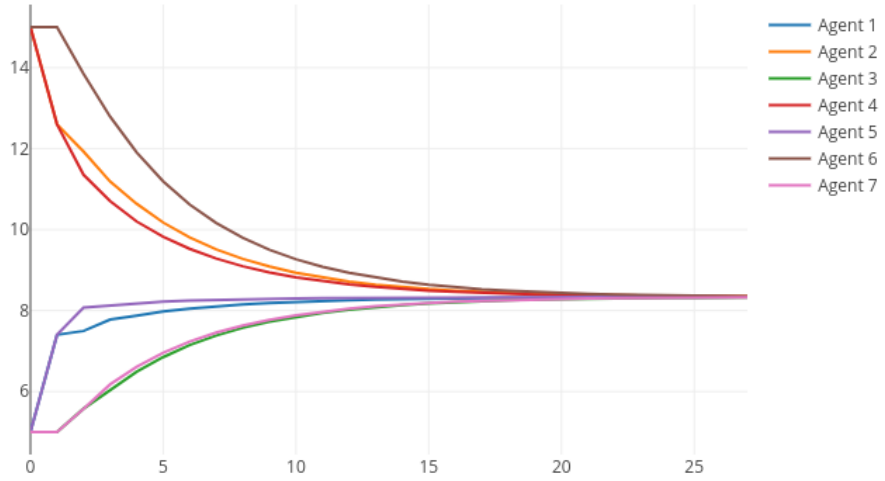


Figure 5.6 Weight-Based Consensus with no malicious agents, weight=0.25.

In Table 5.1 we can see the summary of the results for this experiment. It was expected that the higher the weight for each neighbour, the more aggressive the consensus becomes. The biggest difference can be found in the number of subconsensus rounds it takes to agreed on a state.

Table 5.1 Agreed state and Nº of rounds for each weight-based consensus.

Weight	Agreed State	Nº of rounds
0.15	8.338	45
0.2	8.338	33
0.25	8.338	27

Reputation-Based Consensus

As previously defined in Section 3.4, this algorithm starts off similarly to the Weight-Based consensus where each agents has a fixed weight. We have decided that the maximum weight assigned to a participant of the network is 0.20. If the agent follows the rules of consensus it will maintain that maximum weight, otherwise, its reputation will decrease and so will the weight, as defined by the Equation (5.2)

$$w_i^* = w_{max} * r_i / r_{max}, \tag{5.2}$$

where w_i^* is the updated weight for the agent i, $w_{max} = 0.20$ is the maximum weight, r_i is the current reputation of the participant and $r_{max} = 100$ is the upper limit for the reputation.

Running the same experiment with this consensus algorithm when there are no malicious agents in the network, we can observe in Figure 5.7 how the behaviour of the agents is similar to what we have observed in the previous consensus. The agents reach an agreement around the value 8.340 in 38 rounds of consensus. As all the participants followed the rules, the reputation stayed at the maximum value during the whole process. When there are no anomalies in the network there are no benefits of using this presented method.

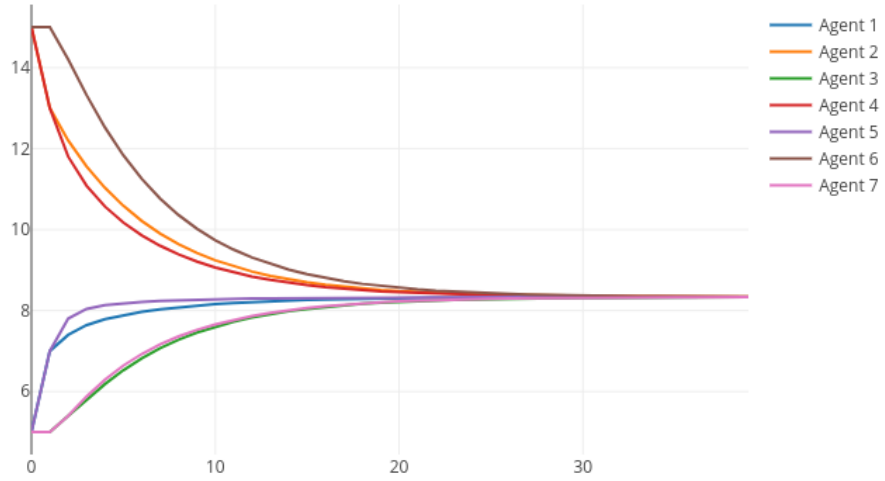


Figure 5.7 Reputation-Based Consensus with no malicious agents.

5.1.2 Robust Network

Following the graph defined in Figure 5.2. We should expect and improvement in the speed to reach consensus, this is due to the fact that most of the agents have more in-neighbour, giving them more information to compute in each round and giving then a more precise output.

Average Consensus

For this experiment, in Figure 5.8, we observe that the agreed value is **8.854** in 16 rounds of consensus. The same scenario with the previous network studied had a consensus around 8.340 in 49 rounds. Some agents took more time to get into the agreement because they lacked the information.

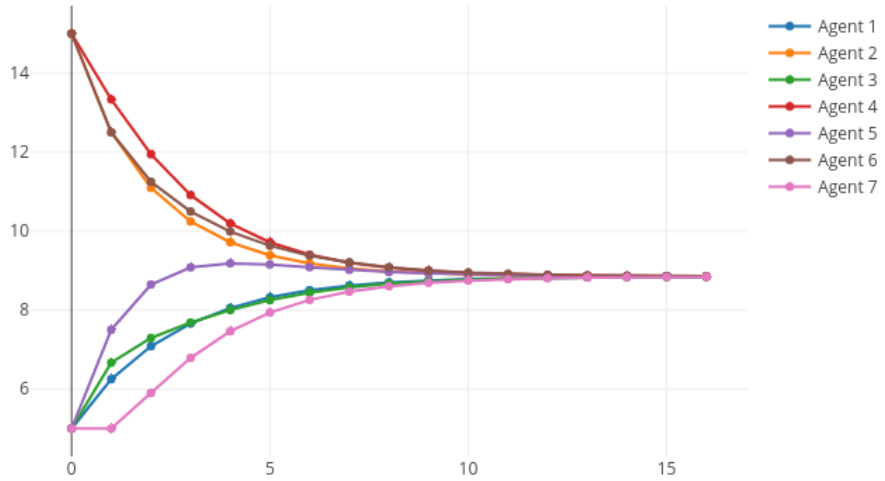


Figure 5.8 Average Consensus with no malicious agents.

Weight-Based Consensus

In the figures 5.9 and 5.10, we see the results for the Weight-Based consensus for the weights 0.15 and 0.20 respectively. For a weight of 0.15, we observe a consensus being reached around **8.376** in 17 rounds and for the weight 0.20 around **8.378** in 12 rounds. We observe again a vast improvement in the number of rounds to reach agreement.

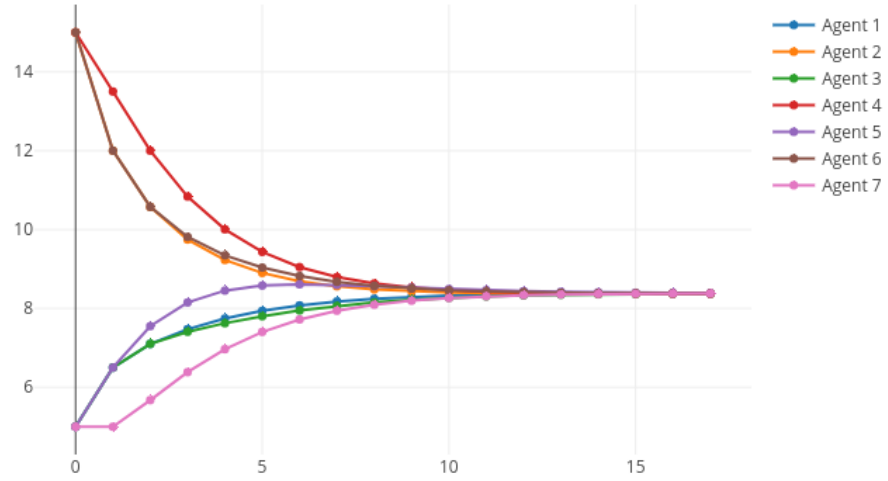


Figure 5.9 Weight-Based Consensus with no malicious agents, weight=0.15.

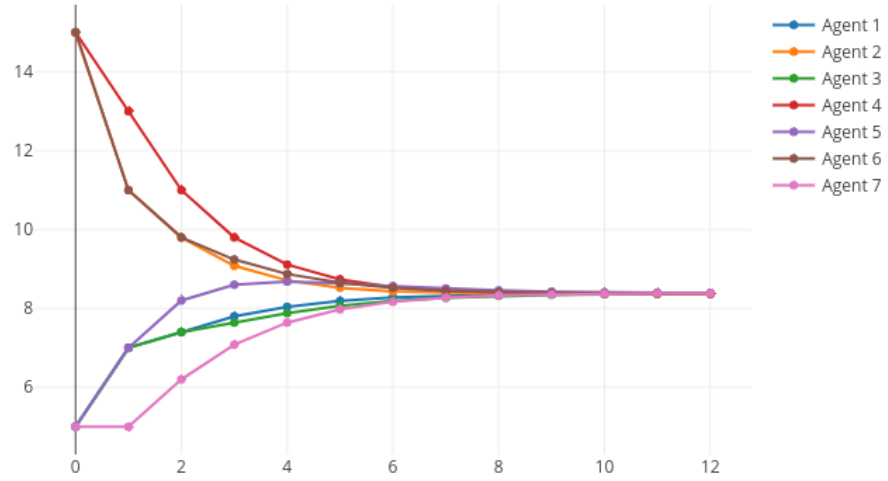


Figure 5.10 Weight-Based Consensus with no malicious agents, weight=0.20.

Reputation-Based Consensus

Lastly, for the Reputation-Based Consensus we expect a result similar to the previous scenario, Weight-Based Consensus with a 0.20 weight. In the Figure 5.11 we observe an agreement in **8.378** in 12 rounds.

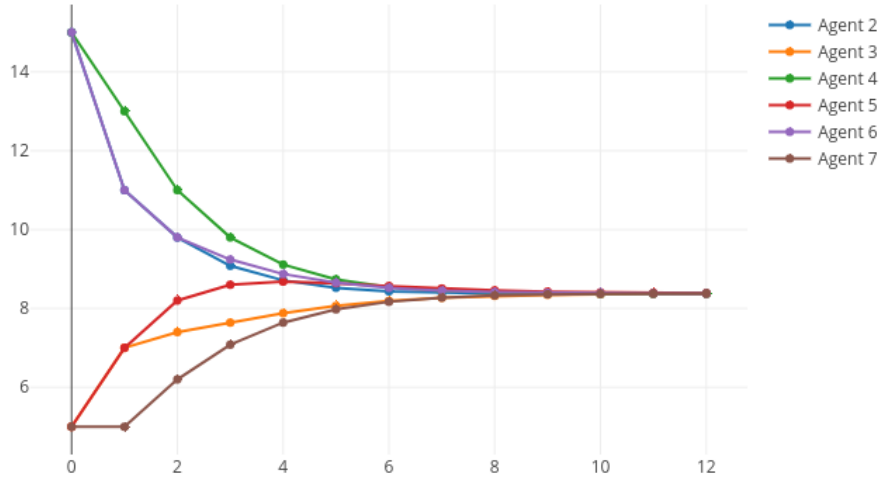


Figure 5.11 Reputation-Based Consensus with no malicious agents.

5.2 Experiment 2: 1 Malicious agent

For this experiment, one agent will not follow the consensus rules and will try to prevent an agreement to be reached. As we defined, the malicious agent will follow the next behaviour: $x_i(2m) = 5, x_i(2m+1) = 15$ where $\forall m \in Z_+$. The malicious agent selected for this experiment is the *Agent 1* in our graph structure, its measure in each subconsensus round will not appear in the plots to keep them clean.

5.2.1 Strongly Connected Network

Following the graph defined in Figure 5.1

Average Consensus

For this consensus algorithm, the agents have not agreed on a value. The malicious agent has completed its objective of altering the consensus result. As we can observe in the Figure 5.12, the bad agent rapidly affects the rest of the network and doesn't allow it to reach consensus by changing its value.

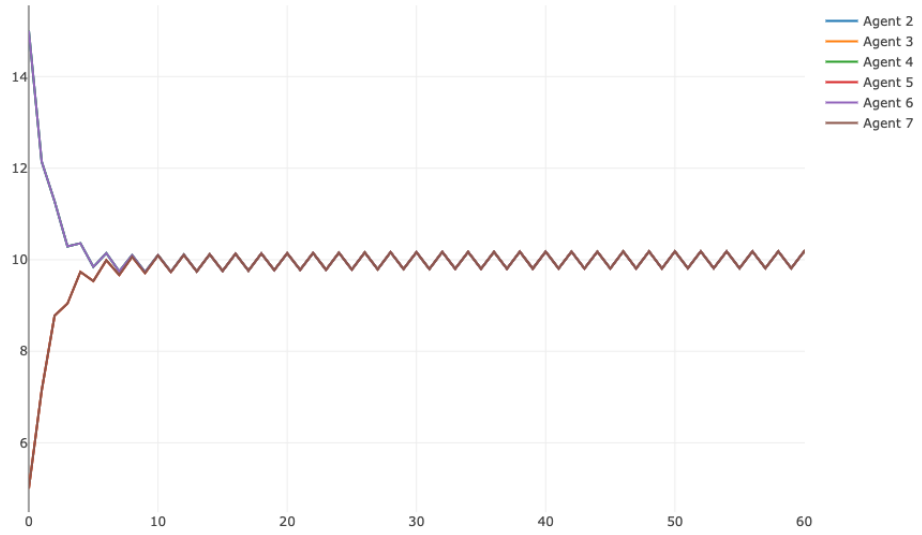


Figure 5.12 Average Consensus with one malicious agent.

Since the *Agent 1* is an in-neighbour of a great part of the network the effects of its malicious intention have more effect than if it was a worse connected agent like *Agent 6*:

Weight-Based Consensus

Like in the previous scenario, we will simulate a weight-based algorithm for multiple weights to see how the network in affected by the malicious agent, Figures 5.13 and 5.14 .

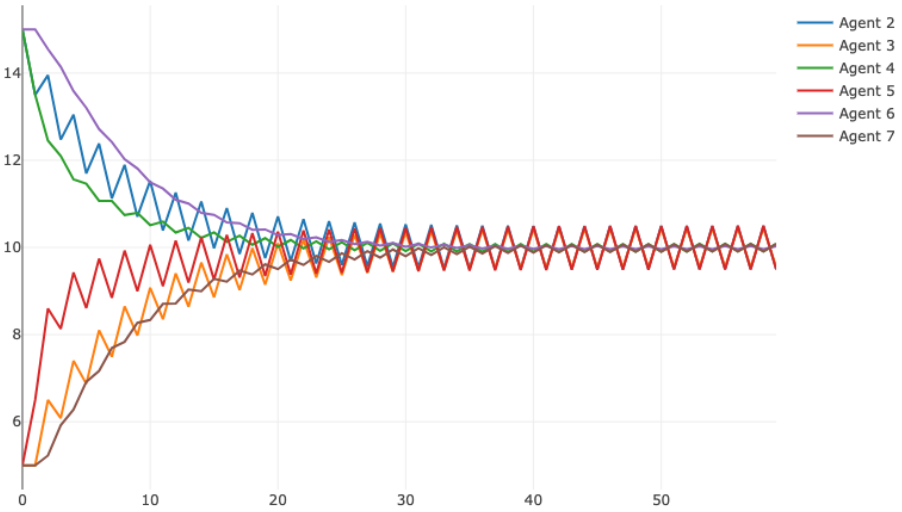


Figure 5.13 Weight-Based Consensus with one malicious agent, weight=0.15.

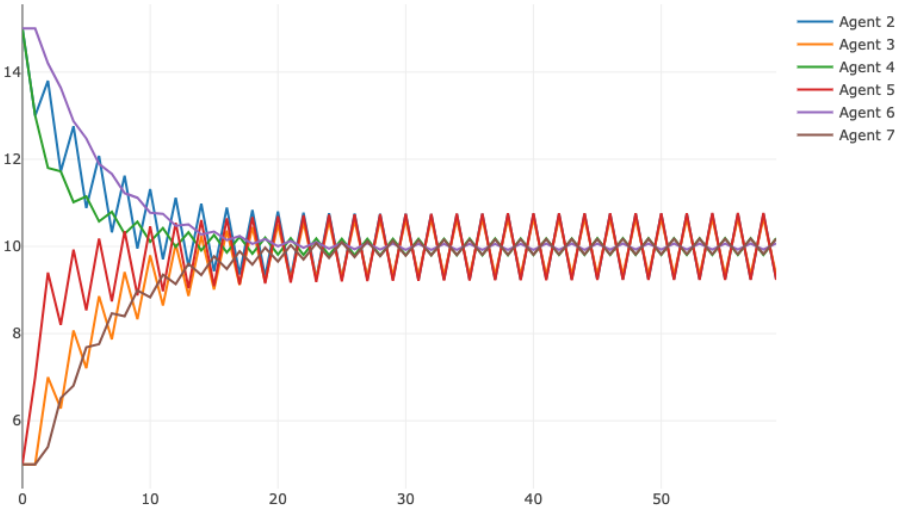


Figure 5.14 Weight-Based Consensus with one malicious agent, weight=0.20.

Similarly to the average consensus algorithm, the consensus has not been reach because the malicious agent is able to control the rest of the network. We can see how some of the agents have bigger oscillations than others. This happens on the agents that have the malicious agent as an in-neighbour. The rest of the network would have the malicious agent as a second layer in-neighbour, i.e. these agents are affected by agents who have the malicious agent as neighbour, so the frequent changes they see are reduced.

Reputation-Based Consensus

Unlike the previous consensus algorithm where there was one malicious agent in the network, in Figure 5.15 we can observe how an agreement have been reached. Since Agent 1 does not follow the defined rules, we can see how its reputation starts decreasing right away, reaching the lower reputation limit of 0 by the fourth round of consensus, Figure 5.16. This agent is able to manipulate the consensus for a short amount of time; from round 5 until the end of consensus the agents that have Agent 1 as a in-neighbour will ignore the measures received. Although it takes longer than when there are no malicious agents, the rest of the participants are able to agree on the value 8.755 in 71 rounds of consensus. We can see how the initial manipulation of the agent has an effect on the final result but at least it did no prevent them from reaching a consensus. The next time these agents start a consensus, the malicious agent will start with a reputation of 0, being ignored by the rest, so the consensus will look more like Figure 5.7.

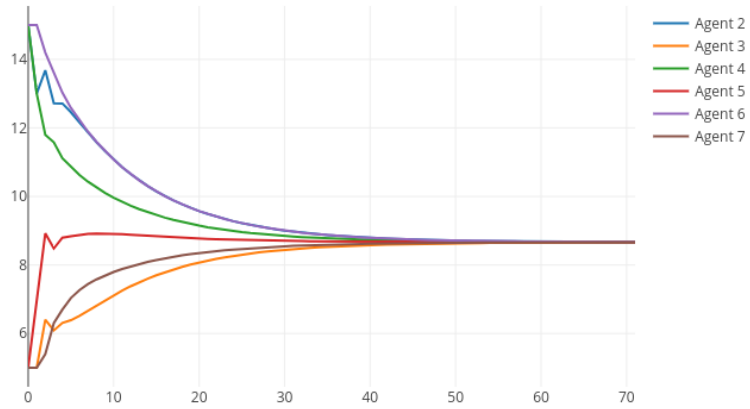


Figure 5.15 Reputation-Based Consensus with one malicious agent.

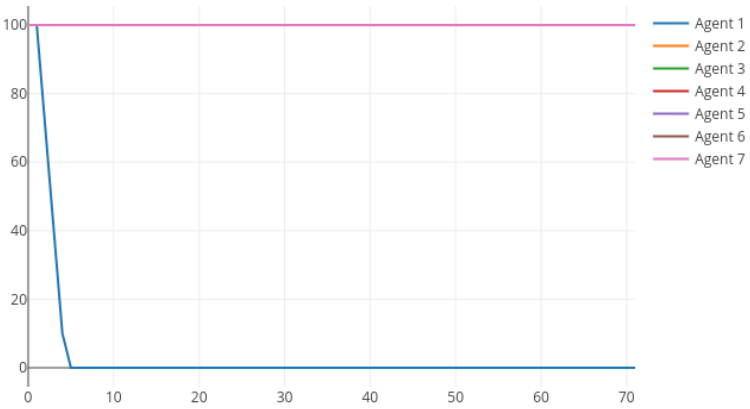


Figure 5.16 Reputation of the agents during consensus with one malicious agent.

5.2.2 Robust Network

Average Consensus

Similar to the same scenario with the strongly connected network, the agents do not reach a consensus and keep oscillating due to the malicious agents changing its measure in each round. We see maximums of 10.848 and minimums of 9.152, Figure 5.17.

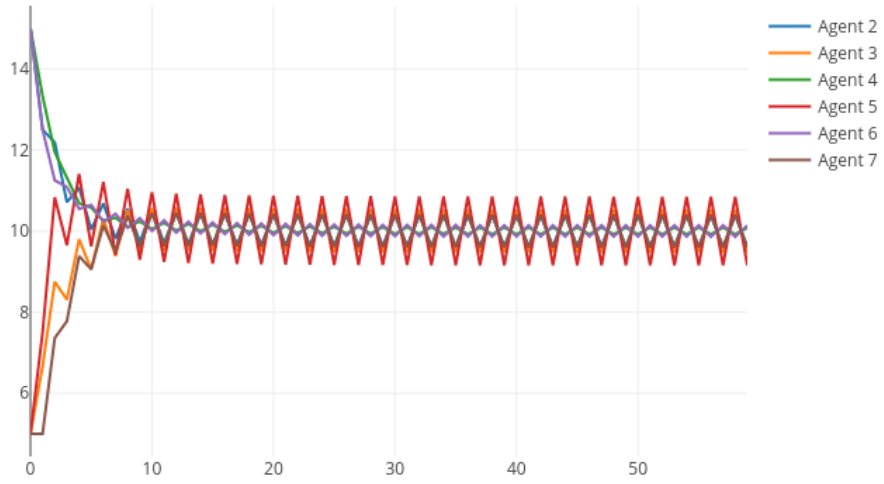


Figure 5.17 Average Consensus with one malicious agent.

Weight-Based Consensus

We see the exact same behaviour when it comes to the Weight-Based Consensus, agents do not reach an agreement and keep oscillating around 10.0, as seen before, the higher weight causes a more abrupt movement.

Reputation-Based Consensus

In this case, despite the efforts of the malicious agent to affect the end result of consensus we can observe how reputation algorithm started to penalize Agent 1, Figure 5.22, until it is completely ignored by the rest of the network. As a result, the agents reach a consensus in 18 rounds around the value **9.212**, Figure 5.21.

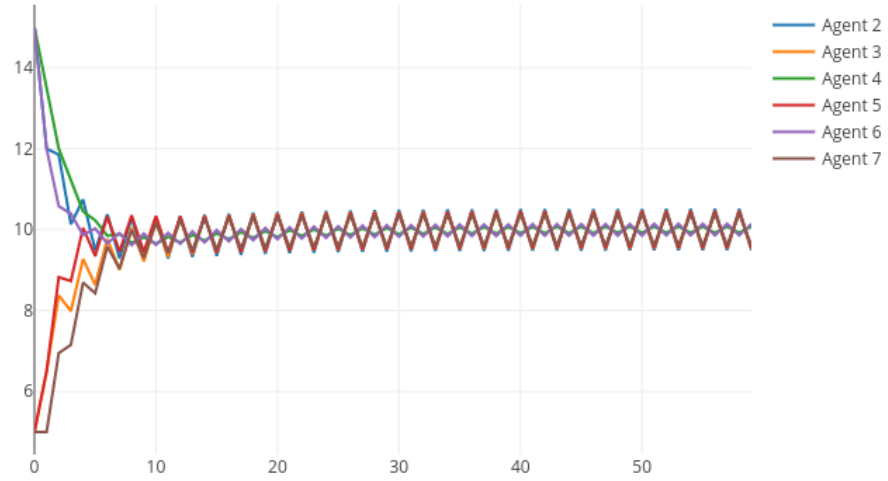


Figure 5.18 Weight-Based Consensus with no malicious agents, weight=0.15.

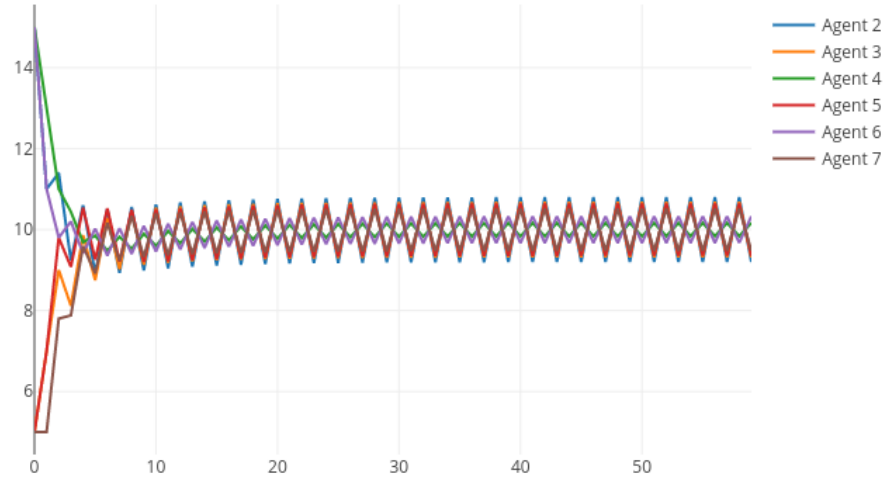


Figure 5.19 Weight-Based Consensus with no malicious agents, weight=0.20.

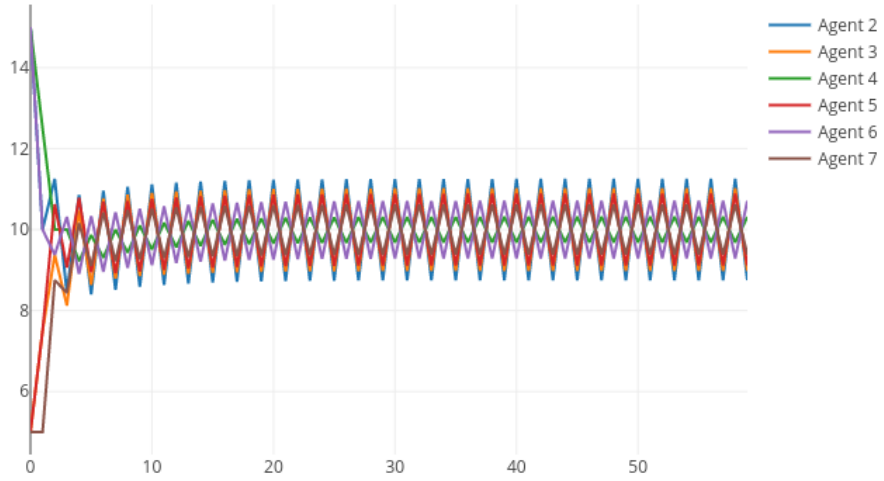


Figure 5.20 Weight-Based Consensus with no malicious agents, weight=0.25.

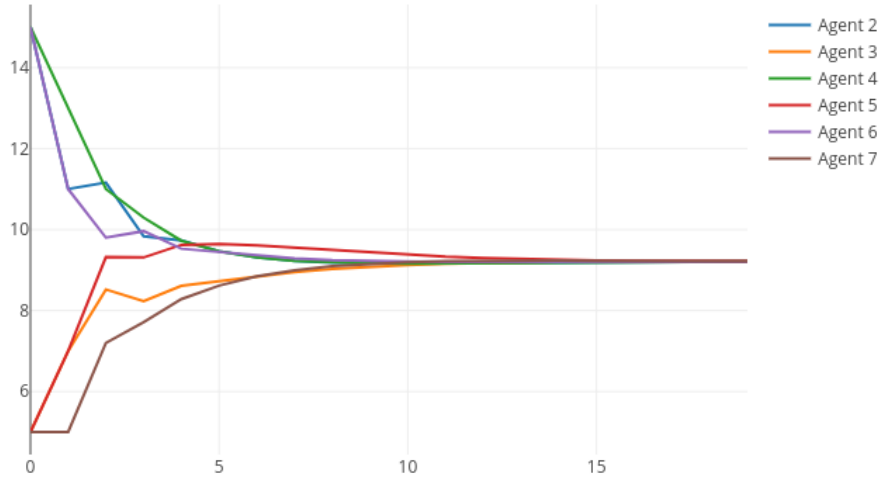


Figure 5.21 Reputation-Based Consensus with one malicious agent.

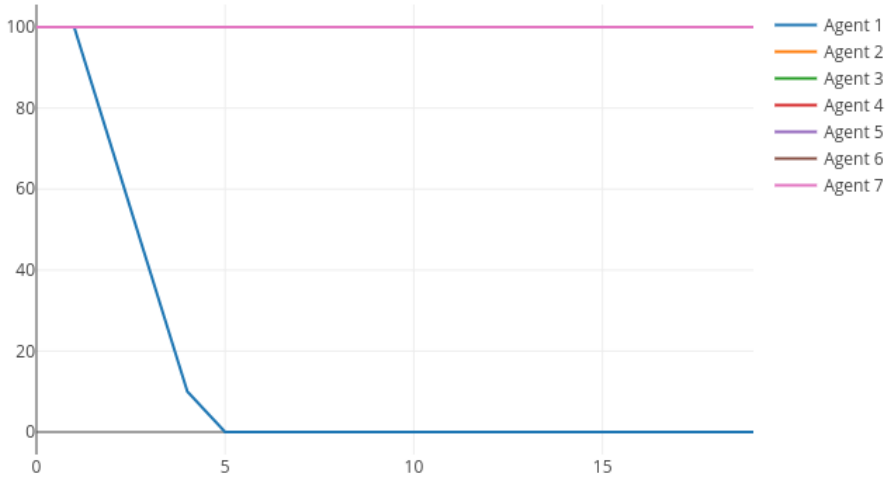


Figure 5.22 Reputation of agents during consensus with one malicious agent.

One of the benefits of this Reputation-Based Consensus is that reputation is saved over time. Therefore, if we were to run the exact same experiment after the previous result, with a null reputation for Agent 1, we see how the consensus is reached, Figure 5.23, to a closer value from our reference state, Figure 5.11, the agreement was attained around **8.556** in 20 rounds. Additionally, we can observe how the reputation evolved as expected in Figure 5.24.

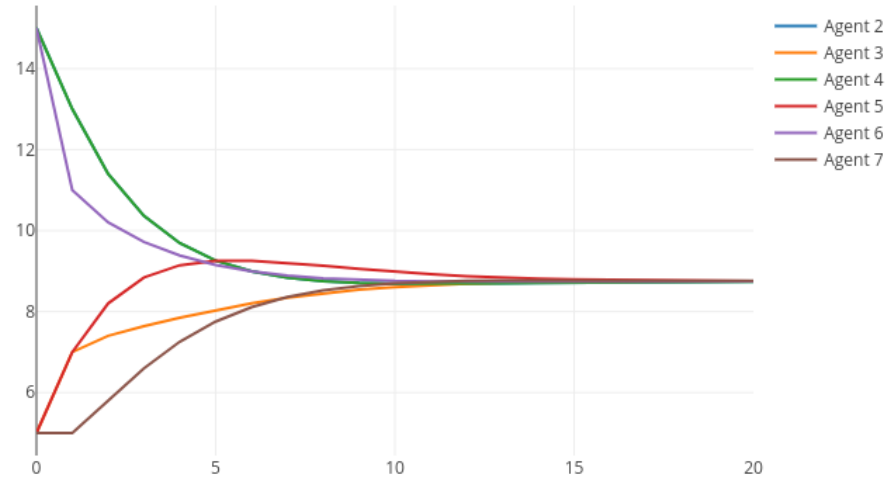


Figure 5.23 Second round of Reputation-Based Consensus with one malicious agent.

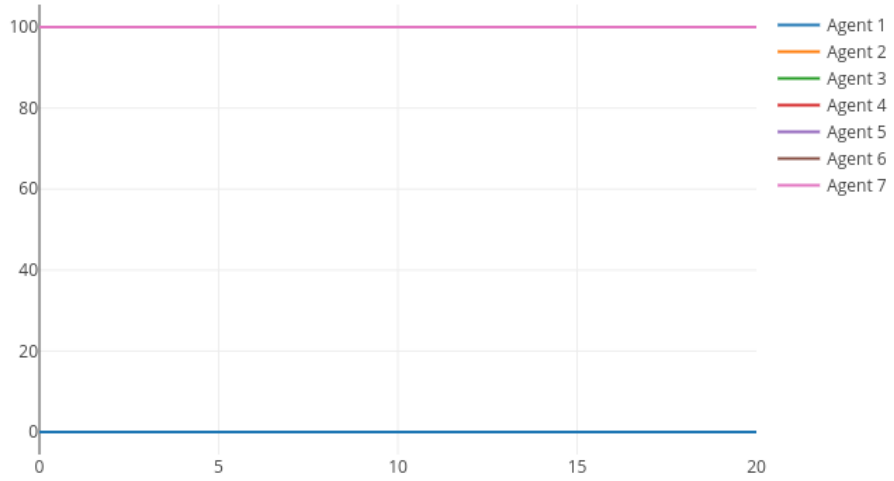


Figure 5.24 Reputation of agents during consensus with one malicious agent, second round.

5.3 Experiment 3: Agent temporal malfunction

For this last experiment, instead of thinking that an agent has been hacked to modify the result of a consensus we will approach a more realistic case where one agent is malfunctioning for a set number of subconsensus rounds. During this period, the agent will not update its measure, it will stay still.

In the following simulations, the agent 1 will malfunction from round 2 to round 15 of the consensus. We will study how each algorithm reacts to this change.

5.3.1 Strongly Connected Network

Following the graph defined in Figure 5.1

Average Consensus

Similarly to previous experiments, in Figure 5.25 we can see how the malfunction has affected the overall consensus by bringing the agreed measure closer to the state in which the agent had a problem. In difference with the scenario where an agent was not following the rules of consensus for the whole period of time, the consensus was achieved around the value **8.110** in 38 rounds of consensus.

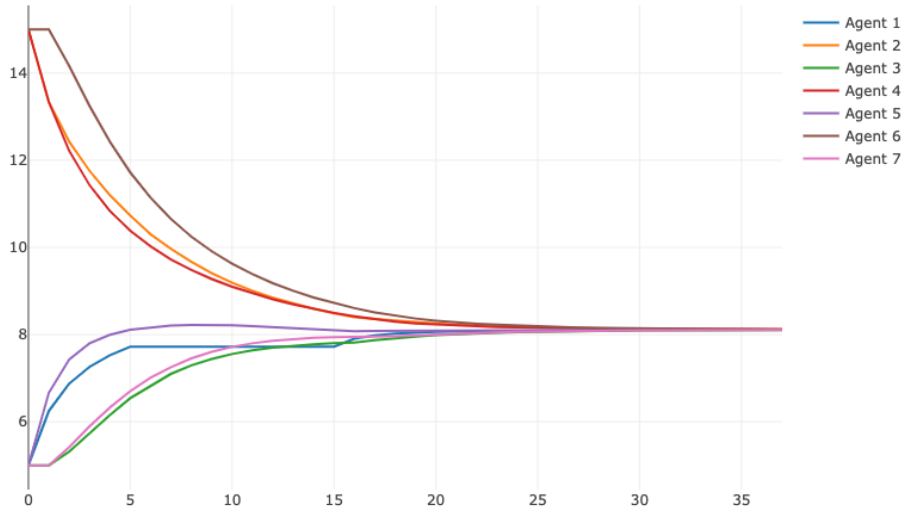


Figure 5.25 Average Consensus in temporal malfunction.

Weight-Based Consensus

We can find the same evolution when it comes to the weight-based algorithm, both consensus (Figure 5.26 and 5.27) reached an agreement although affected by the period of malfunctioning of agent 1, the measure agreed on was **8.115** and **8.188** respectively.

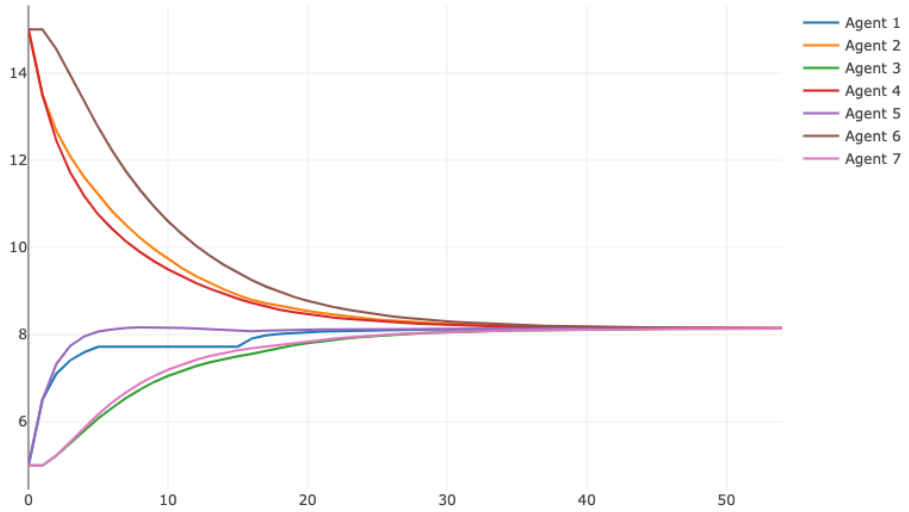


Figure 5.26 Weight-Based Consensus in temporal malfunction, weight=0.15.

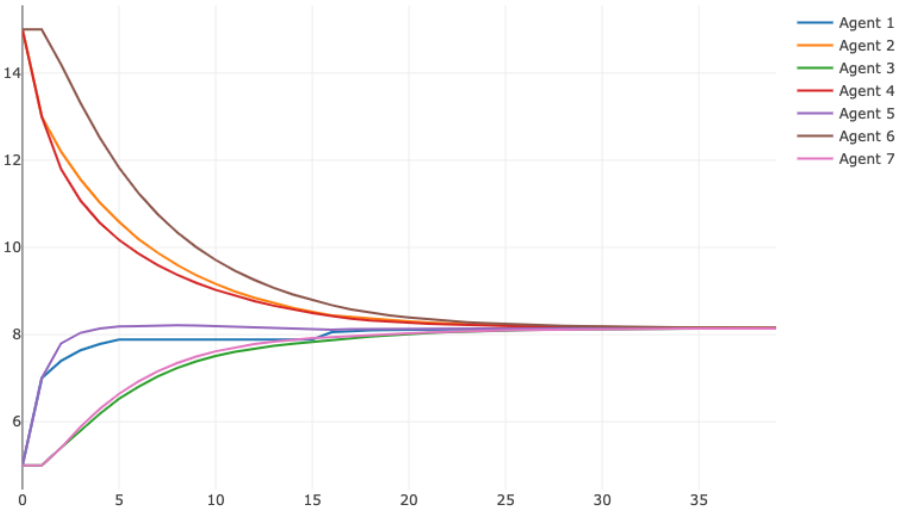


Figure 5.27 Weight-Based Consensus in temporal malfunction, weight=0.20.

Reputation-Based Consensus

In Figure 5.28, we see the experiment for temporal malfunction during a Reputation-Based algorithm. In 47 rounds of consensus the agents reached an agreement around the value 8.259. We can observe the reputation of Agent 1 decreasing in Figure 5.29 until the lower limit is reached in round 7 and then, when the agent works as expected, the reputation starts to increased slowly until the higher limit is reached in round 36. The impact that the malfunction had in the agreement was far less severe than what we observed in the other algorithms

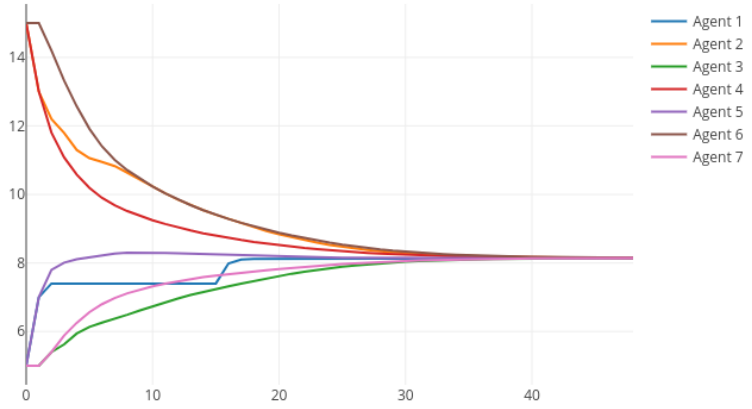


Figure 5.28 Reputation-Based Consensus in temporal malfunction.

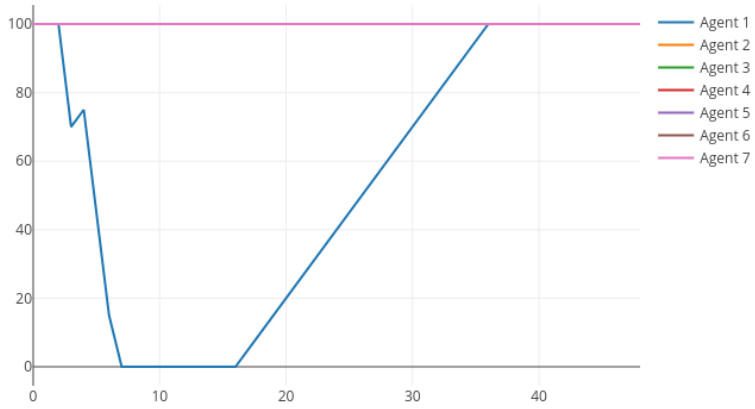


Figure 5.29 Reputation of agents in temporal malfunction.

5.3.2 Robust Network

Average Consensus

In this scenario, consensus was reached although it was affected by the temporal malfunction of Agent 1. In Figure 5.30 we can see how the the agents agreed around **7.555** in 24 rounds.

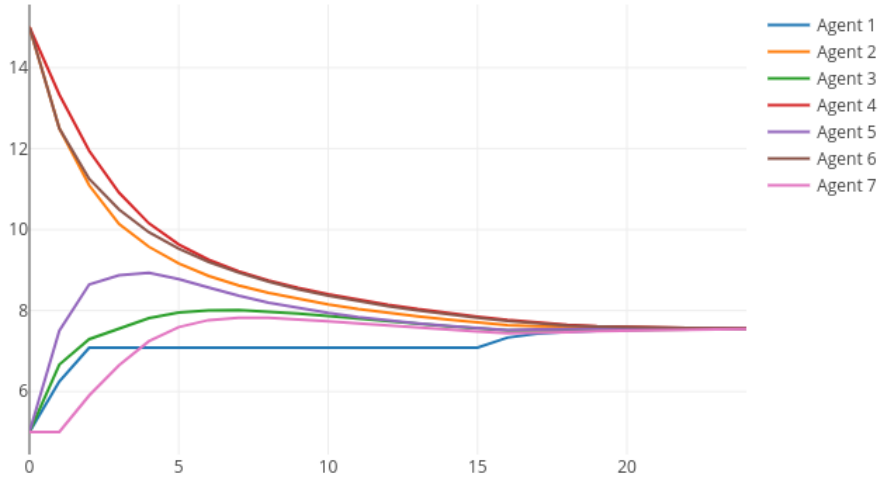


Figure 5.30 Average Consensus in temporal malfunction.

Weight-Based Consensus

Next, for the Weight-Based Consensus we observe a similar behaviour. For the weight 0.15 consensus was reached in 23 rounds around **7.495**, Figure 5.31, for the weight 0.20 it was reached in 19 rounds around **7.582**, Figure 5.32 and, lastly, for 0.25 the agents agreed on **7.595** in 18 rounds, Figure 5.33. As expected, the temporal malfunction affected the value by bringing it closer to the point of failure.

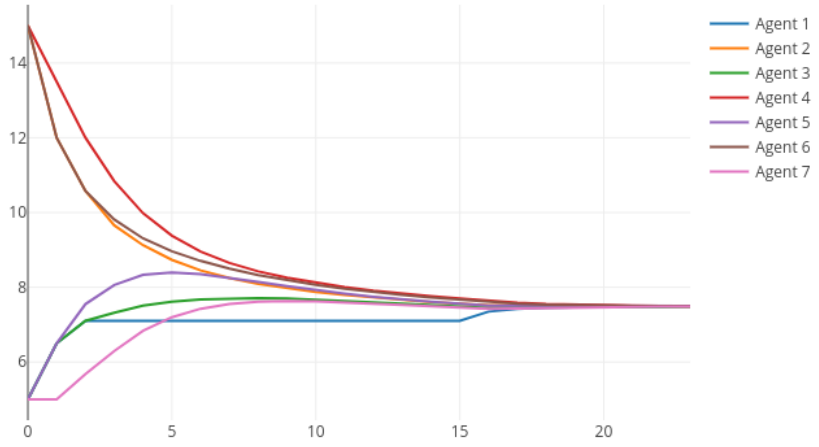


Figure 5.31 Weight-Based Consensus in temporal malfunction, weight=0.15.

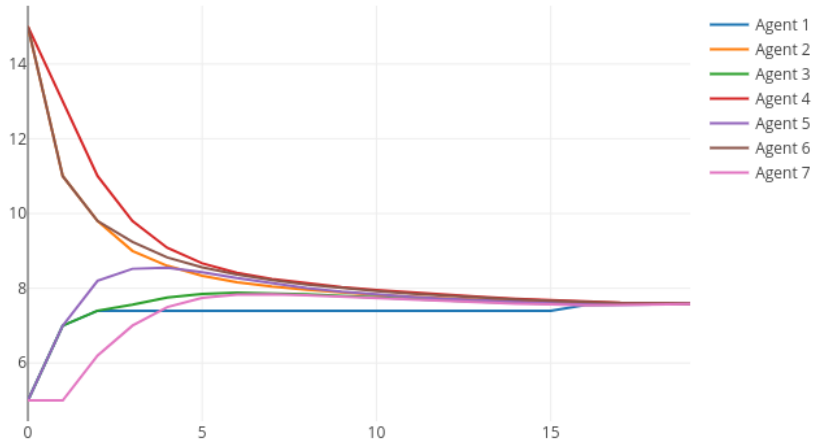


Figure 5.32 Weight-Based Consensus in temporal malfunction, weight=0.20.

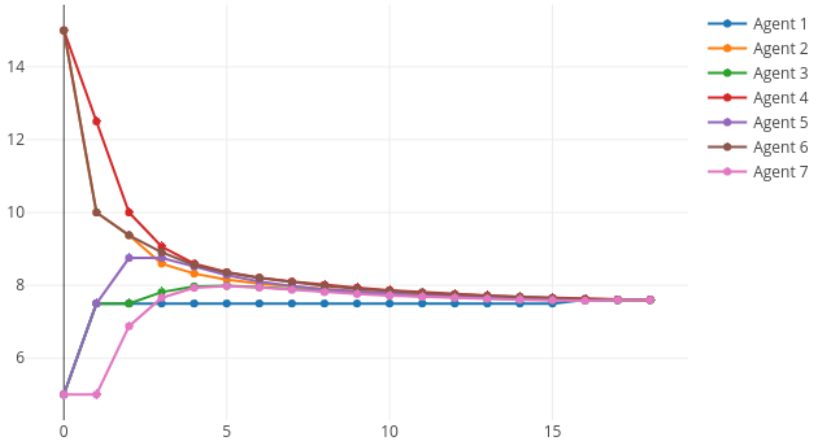


Figure 5.33 Weight-Based Consensus in temporal malfunction, weight=0.25.

Reputation-Based Consensus

In the next scenario, we observe how the reputation of Agent 1 rapidly decreased and started to recover after the malfunction was over, although it only recovered 10 points of reputation, Figure 5.35. The consensus was reached around **8.549** in 18 rounds of consensus.

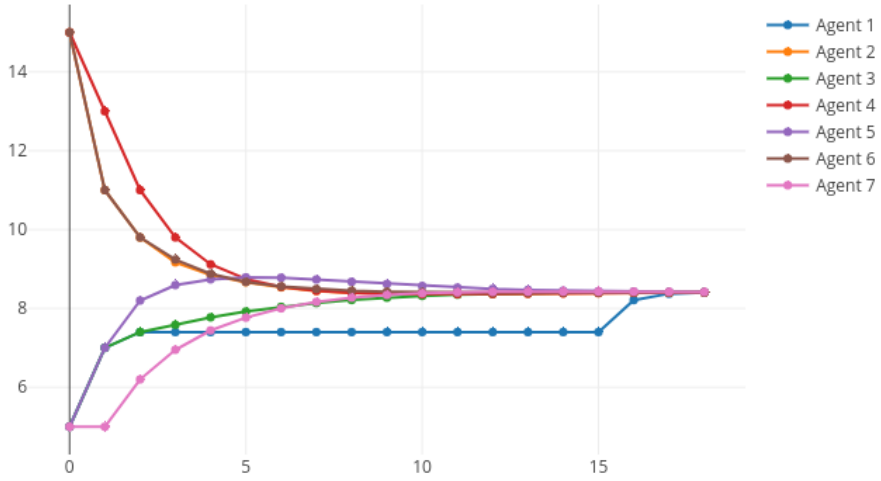


Figure 5.34 Reputation-Based Consensus in temporal malfunction.

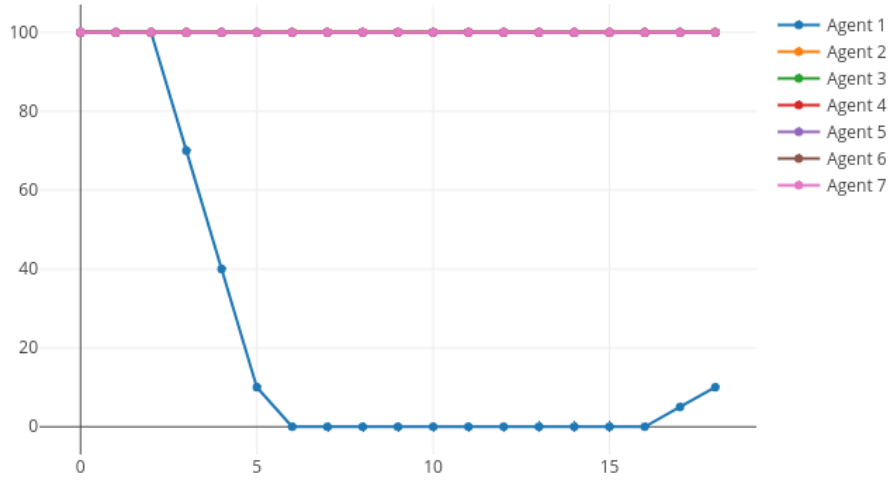


Figure 5.35 Reputation of agents in temporal malfunction.

If we launch the same experiment a second time, we see how the reputation starts at 10 and gets to 0 in a earlier round than before, having less impact in the consensus. The consensus was reached around **8.409** in 19 round.

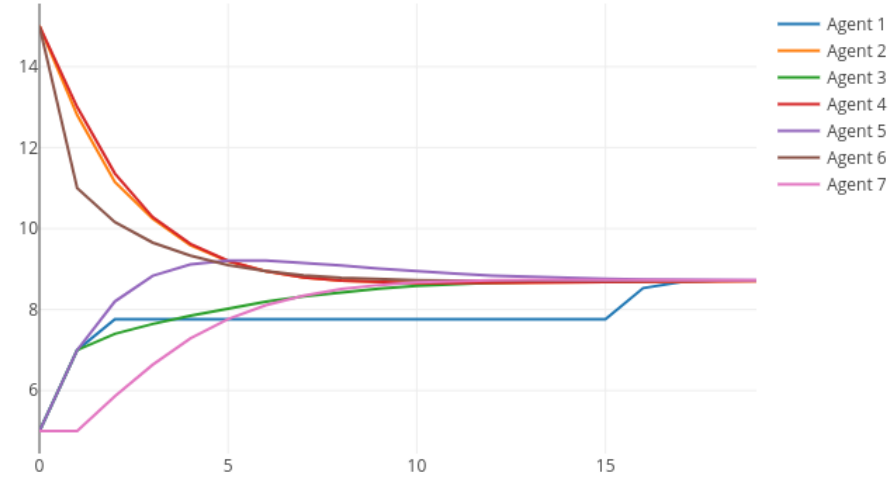


Figure 5.36 Second round of Reputation-Based Consensus in temporal malfunction.

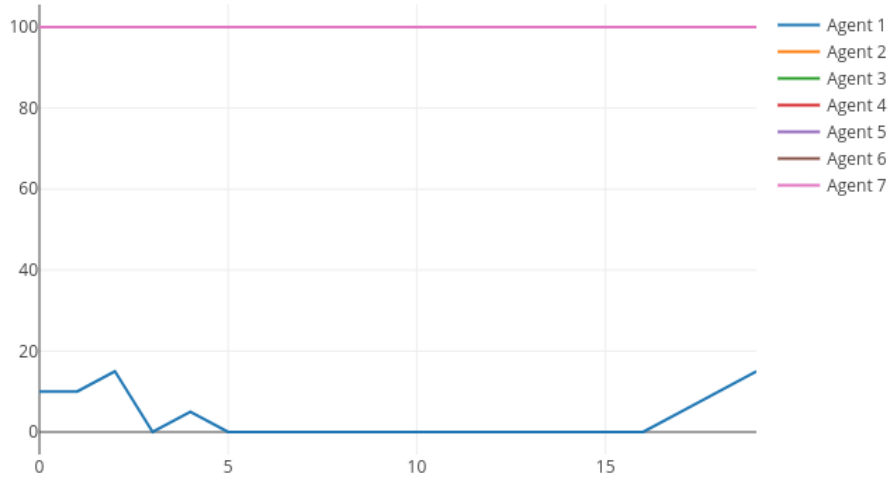


Figure 5.37 Reputation of agents in temporal malfunction, second round.

5.3.3 Experiments conclusions

In this chapter we have been analyzing different scenarios with multiple type of consensus algorithm to get conclusions around the effectiveness of the proposed Reputation-Based Consensus using the Blockchain technology.

Here we can observe two tables that summarize all the information retrieved during the experiments. This will help us to visualize all the scenarios in one place and make conclusions.

Strongly connected network

Table 5.2 Summary of results for the experiments with a strongly connected network.

Experiment	Type of Consensus	Reached Consensus?	Agreed Value	Number of Rounds	Error (%)
No Malicious Agents	Average Consensus	Yes	8.340	49	-
	Weight-Based Consensus	Yes	8.338	33	-
	Reputation-Based Consensus	Yes	8.340	38	-
One Malicious Agent	Average Consensus	No	-	-	-
	Weight-Based Consensus	No	-	-	-
	Reputation-Based Consensus	Yes	8.755	71	4.976
Temporal Malfunction	Average Consensus	Yes	8.110	38	2.758
	Weight-Based Consensus	Yes	8.188	37	1.799
	Reputation-Based Consensus	Yes	8.259	47	0.971

Robust network

Table 5.3 Summary of results for the experiments with a robust network.

Experiment	Type of Consensus	Reached Consensus?	Agreed Value	Number of Rounds	Error (%)
No Malicious Agents	Average Consensus	Yes	8.854	16	-
	Weight-Based Consensus	Yes	8.378	12	-
	Reputation-Based Consensus	Yes	8.378	12	-
One Malicious Agent	Average Consensus	No	-	-	-
	Weight-Based Consensus	No	-	-	-
	Reputation-Based Consensus	Yes	9.212/8.556[a]	18/20	9.955/2.125
Temporal Malfunction	Average Consensus	Yes	7.555	24	14.671
	Weight-Based Consensus	Yes	7.582	19	9.501
	Reputation-Based Consensus	Yes	8.549/8.409	19/18	2.041/0.370

[a] The notation references the two times the algorithm was run. In the second attempt the rest of the network knows about the malicious agent

With all this data summarized together we can easily see how the Reputation-Based Consensus improves the biggest disadvantages of the other algorithms analyzed.

- When the other algorithms are unable to reach an agreement, the proposed consensus algorithm is able to find the culprit and lower its reputation so the neighbours can ignore the provided measure and finish the consensus.

- When any of the participant is malfunctioning or has become malicious for a set amount of time, the offset induced into the agreed value is far less noticeable in the Reputation-Based Consensus.

- If the consensus were to be run multiple times, the proposed algorithm will give an improved output because the network will already know who the malicious agent is and who follows the rules.

6 Conclusions and next steps

In this last chapter, we present a summary of all of the conclusions obtained during the project, and analyze whether the outcome was as expected and successful.

The main objective of this document was to create a reputation-based consensus algorithm that analyzes if the behaviour of the participant in the consensus is valid, i.e. if the set of rules are being followed, and modifies the reputation linked to that agent. All of this was to be built in a blockchain network to take advantage of the security that it provides, among other benefits.

In the first chapters of this project, we revealed a new technology and laid down the main concepts that helped us to understand in more detail the possibilities blockchain provides. Next, we learned about the Hyperledger suite and the ability to quickly create a proof of concept on a blockchain network with the Byzantine Fault Tolerant consensus protocol. In addition, we explored the attributes that a real blockchain network in production should have and how we could create it, although for the objectives of this project using a local test network was enough.

With the definition of the problem we solved, we introduced the concepts around consensus algorithms and what should be accepted as a valid solution. We presented an algorithm used in many papers, a Weight-Based algorithm, to compare it later on with our new Reputation-Based algorithm.

Following the problem definition we started to develop the main application. On the one hand, we had to create the smart contracts for the blockchain network using Hyperledger Composer. This tool helped us to create the assets definition, the transaction logic to interact with them and the queries in a fast manner. Once our network and smart contracts were ready, we created a set of python scripts to help with the simulations in different scenarios. This script took care of creating the agents of the network and making them interact with each other until a consensus was achieved.

When running the simulations, we created three different scenarios to test the Reputation-Based algorithm and compare them with other known consensus algorithms. First, a reference setup where everyone follows the rules and there is no interference for the consensus. Then, one of the participants will change the value they report in each round, trying to prevent the consensus to be reached. Lastly, one of the agents will temporarily hold the value reported trying to take the end consensus closer to the measure.

We can clearly observe that thanks to the reputation system the consensus is always reached. When the malicious agent does not follow the rules, its reputations starts to decrease, getting to zero after 4 rounds. Once at zero, neighbours will just ignore the value received, continuing the consensus as if the participant didn't exist. In the case of the temporal malfunction, the damage done to the final value of consensus is far less noticeable than with the Weight-Based Consensus. In addition, if the consensus were to be run twice the effect will be even less perceptible.

This project opens possibilities not yet explored in the area of consensus algorithms, here we expose some subjects that have great interest within the initial concepts of the project:

- Researching the possibility of canceling an ongoing consensus when any participant drops a set percentage of reputation. As we have discovered, when any participant decreased its reputation until the limit was reached the consensus was not affected anymore by the measures but there was a initial deviation of the final agreement. It was seen how if the consensus was run twice that the second time got better results since it started with said agent ignored. The ability for the consensus to roll back completely, starting over, or a number of rows to decrease the effect without restarting would be a good subject to study.

- In this case, we used a private/hybrid blockchain network, but it would be interesting to find a specific case where a public blockchain network, i.e. Ethereum, makes sense to run a similar experiment and compare the results obtained with the analysis done in this project, specially to see the difference in complexity and speed of the blockchain network.

- Comparing this newly introduced Reputation-Based consensus algorithm with more advanced algorithms would also help to verify further this project.

List of Figures

List of Tables

List of Codes

Bibliography

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[2] Blockchain definition - merriam webster dictionary. *https://www.merriam-webster.com/dictionary/blockchain?utm_campaign=sd&utm_medium=serp&utm_source=jsonld*, . [Online; accessed 13-April-2020].

[3] Blockchain coding resources. *http://rasimsen.com/index.php?title=BlockChain*, . [Online; accessed 13-April-2020].

[4] Melanie Swan. *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[5] Marc Pilkington. 11 blockchain technology: principles and applications. *Research handbook on digital transformations*, 225, 2016.

[6] quantalysus. Choosing between centralized, decentralized, and distributed networks. *https://steemit.com/cryptocurrency/@quantalysus/choosing-between-centralized-decentralized-and-distributed-networks*, 2018. [Online; accessed 6-August-2019].

[7] Imran Bashir. *Mastering blockchain*. Packt Publishing Ltd, 2017.

[8] Andreas M Antonopoulos and Gavin Wood. *Mastering Ethereum: building smart contracts and dapps*. O'Reilly Media, 2018.

[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[10] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982. URL *https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/*.

[11] Stuart Haber and W Scott Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.

[12] Coinmarketcap - cryptocurrency market cap rankings, charts, and more. *https://coinmarketcap.com*. [Online; accessed 6-August-2019].

[13] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18:2, 1996.

[14] $6.3 billion: 2018 ico funding has passed 2017's total. *https://www.coindesk.com/6-3-billion-2018-ico-funding-already-outpaced-2017*. [Online; accessed 6-August-2019].

[15] What is supply chain management. *https://www.ibm.com/topics/supply-chain-management*. [Online; accessed 8-March-2020].

[16] What is supply chain management. *https://www.ibm.com/blockchain/solutions/food-trust*. [Online; accessed 8-March-2020].

[17] Iot for blockchain. *https://www.ibm.com/blogs/internet-of-things/iot-new-blockchain-service/*, . [Online; accessed 4-May-2020].

[18] Blockchain for digital identity - ibm. *https://www.ibm.com/blockchain/solutions/identity/*, . [Online; accessed 4-May-2020].

[19] Self-sovereign identity - sovrin. *https://sovrin.org*. [Online; accessed 4-May-2020].

[20] Hyperledger website. *https://www.hyperledger.org*, . [Online; accessed 6-August-2019].

[21] Kelly Olson, Mic Bowman, James Mitchell, Shawn Amundson, Dan Middleton, and Cian Montgomery. Sawtooth: An introduction. *The Linux Foundation, Jan*, 2018.

[22] Hyperledger fabric documentation. *https://hyperledger-fabric.readthedocs.io/en/release-1.2/*, . [Online; accessed 6-August-2019].

[23] Hyperledger composer main documentation. *https://hyperledger.github.io/composer/latest/introduction/introduction.html*, . [Online; accessed 6-August-2019].

[24] Nitin Gaur, Luc Desrosiers, Venkatraman Ramakrishna, Petr Novotny, Salman A Baset, and Anthony O'Dowd. *Hands-On Blockchain with Hyperledger: Building decentralized applications with Hyperledger Fabric and Composer*. Packt Publishing Ltd, 2018.

[25] Shutterstock interconnected vehicles image. *https://www.shutterstock.com/es/image-vector/signalized-intersection-various-vehicle-traffic-wireless-375559120*. [Online; accessed 20-April-2020].

[26] S. S. Kia, B. Van Scoy, J. Cortes, R. A. Freeman, K. M. Lynch, and S. Martinez. Tutorial on dynamic average consensus: The problem, its applications, and the algorithms. *IEEE Control Systems Magazine*, 39(3):40–72, 2019.

[27] James Usevitch and Dimitra Panagou. r-robustness and (r, s)-robustness of circulant graphs. *CoRR*, abs/1710.01990, 2017. URL *http://arxiv.org/abs/1710.01990*.

[28] Heath J. LeBlanc, Haotian Zhang, Xenofon D. Koutsoukos, and Shreyas Sundaram. Resilient asymptotic consensus in robust networks. *IEEE Journal on Selected Areas in Communications*, 31:766–781, 2013.

[29] R. Olfati-Saber, J. A. Fax, and R. M. Murray. Consensus and cooperation in networked multi-agent systems. *Proceedings of the IEEE*, 95(1):215–233, 2007.

[30] Dimitri Bertsekas and John Tsitsiklis. Parallel and distributed computation : numerical methods / dimitri p. bertsekas, john n. tsitsiklis. *SERBIULA (sistema Librum 2.0)*, 01 1989.

[31] Reza Saber and Richard Murray. Consensus protocols for networks of dynamic agents. volume 2, pages 951– 956, 02 2003. ISBN 0-7803-7896-2. doi: 10.1109/ACC.2003.1239709.

[32] Seyed Mehran Dibaji, Hideaki Ishii, and Roberto Tempo. Resilient randomized quantized consensus. *CoRR*, abs/1710.06994, 2017. URL *http://arxiv.org/abs/1710.06994*.

[33] Kai Cai and Hideaki Ishii. Average consensus on general strongly connected digraphs. *CoRR*, abs/1203.2563, 2012. URL *http://arxiv.org/abs/1203.2563*.

[34] Kai Lei, Qichao Zhang, Limei Xu, and Zhuyun Qi. Reputation-based byzantine fault-tolerance for consortium blockchain. 12 2018. doi: 10.1109/PADSW.2018.8644933.

[35] Jiawen Kang, Zehui Xiong, Dusit Niyato, Dongdong Ye, Dong Kim, and Jun Zhao. Toward secure blockchain-enabled internet of vehicles: Optimizing consensus management using reputation and contract theory. *IEEE Transactions on Vehicular Technology*, PP:1–1, 01 2019. doi: 10.1109/TVT.2019.2894944.

[36] Fangyu Gai, Baosheng Wang, Wenping Deng, and Wei Peng. *Proof of Reputation: A Reputation-Based Consensus Protocol for Peer-to-Peer Network*, pages 666–681. 05 2018. ISBN 978-3-319-91457-2. doi: 10.1007/978-3-319-91458-9_41.

[37] Tfm github project. *https://github.com/emiliomarin/TFM-Reputation-Based-Consensus/*. [Online; accessed 18-April-2020].

[38] Hyperledger composer queries documentation. *https://hyperledger.github.io/composer/v0.19/reference/query-language*, . [Online; accessed 19-August-2019].

[39] Hyperledger composer access control documentation. *https://hyperledger.github.io/composer/v0.19/reference/acl_language*, . [Online; accessed 19-August-2019].

[40] Deploying single organization fabric network. *https://hyperledger.github.io/composer/latest/tutorials/deploy-to-fabric-single-org*, . [Online; accessed 21-August-2019].

[41] Hyperledger fabric network design setup. *https://courses.pragmaticpaths.com/p/hyperledger-fabric*, . [Online; accessed 21-August-2019].

[42] Setup multi organization hyperledger fabric network locally on the cloud. *https://courses.pragmaticpaths.com/p/tutorial-hyperledger-fabric*, . [Online; accessed 21-August-2019].

[43] Plotly library for graph creation in python. *https://plot.ly/python/*. [Online; accessed 10-September-2019].