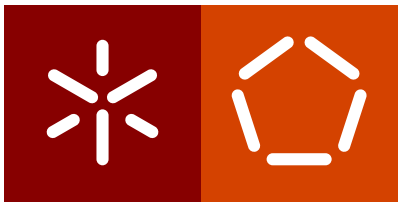
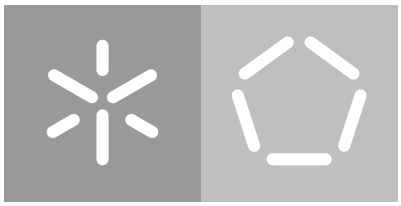


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Ricardo António Gonçalves Pereira

Adaptive Consensus for the Blockchain

November 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Ricardo António Gonçalves Pereira

Adaptive Consensus for the Blockchain

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor José Orlando Roque Nascimento Pereira
Dra. Ana Luísa Parreira Nunes Alonso

November 2019

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ACKNOWLEDGEMENTS

I would like to express my acknowledgments to some people who were very important and special to me during this year and during my academic life. Firstly, I want to thank very much my supervisors for all the help and support during this dissertation. To Professor José Orlando Roque Nascimento Pereira for guiding me, for clarifying all my doubts, and for all his availability to help. My special thanks to him for the research grant offer and the opportunity to join the HASLab Distributed Systems Group, which gave me more motivation to work. To Dra. Ana Luísa Parreira Nunes Alonso for all the support and time spent with me to clarify my doubts and help me to overcome the problems that I faced.

To HASLab Distributed System group, for the excellent work environment, for all the availability to help, and for the social activities and sports activities.

To my whole family, especially to my mother, Maria, my sisters, Liliana and Cindy, my uncles, José and Domingos, and my grandmother, Maria, for always being present to me, for helping me to grow, for giving me the best education, and all the support that I needed during these five years.

Finally, I would like to thank also my close friends for all their help and support, and for all the moments that we passed together during these five years.

ABSTRACT

Consensus is essential to the Blockchain as it enables participants to share a consistent view of the underlying distributed ledger. Currently existing protocols either rely on Proof-of-Work or similar economic incentive schemes, with high transaction latency but that can handle thousands of participants or on classical byzantine fault tolerant consensus protocols, with low transaction latency but that do not scale well with the number of participants.

In this work, one goal is to look at classical consensus protocols and assess the impact that protocol parameters can have on the behaviour of the system, considering different settings (e.g. network), scale (participants), load and trust assumptions, for example. Furthermore, we propose an adaptive consensus protocol for the Blockchain, using an optimization mechanism that configures the protocol automatically.

Keywords: Blockchain, Consensus, Optimization, Machine Learning

RESUMO

O consenso é essencial para a *Blockchain*, pois permite que os participantes compartilhem uma visão coerente do *ledger* distribuído subjacente. Os protocolos actualmente existentes baseiam-se em esquemas de incentivo económico como o Proof-of-Work da BitCoin ou similares, com alta latência de transações, mas que podem lidar com milhares de participantes ou com protocolos clássicos de consenso tolerantes a falhas bizantinas, com baixa latência de transações, mas que não escalam bem com o número de participantes.

Nesta dissertação, um dos objetivos é analisar os protocolos de consenso clássicos e avaliar o impacto que os parâmetros do protocolo podem ter no comportamento do sistema, considerando, por exemplo, diferentes ambientes (por exemplo, rede), escala (participantes), carga e suposições de confiança. Para além disso, nós propomos um protocolo de consenso adaptativo para a *Blockchain*, usando um mecanismo de otimização que configura o protocolo automaticamente.

Palavras-chave: Blockchain, Acordo Distribuído, Otimização, Machine Learning

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

A

ACK Acknowledgement.

AI Artificial Intelligence.

ANN Artificial Neural Network.

B

BFT Byzantine Fault Tolerance.

BTC BitCoin.

C

CFT Crash Fault Tolerance.

D

DL Deep Learning.

F

FIFO First-In, First-Out.

FLP Fischer-Lynch-Paterson.

H

HLF HyperLedger Fabric.

I

IBM International Business Machines Corporation.

IOT Internet of things.

M

MAE Mean Absolute Error.

MCP Mutable Consensus Protocol.

ML Machine Learning.

O

OS Operating System.

P

PBFT Practical Byzantine Fault Tolerance.

POC Proof of concept.

POS Proof-of-Stake.

POW Proof-of-Work.

S

SDK Software Development Kit.

U

UDP User Datagram Protocol.

1

INTRODUCTION

Blockchain technology changes the way the world stores, shares, and manages information and data, becoming a revolutionary technology to the finance and financial services industries [1]. The main feature of this technology is decentralization. It decentralizes all transactions and data between the participants. With this, it eliminates the need for trusted third parties to verify transactions, being validated by the participants through a consensus protocol. This new concept changed the business world, more precisely when *BitCoin (BTC)* [2] was introduced. BTC is a cryptocurrency, supported by blockchain technology that became very popular, getting a lot of people to invest in it, reaching a value of over $19000 [3]. Currently, the global value of all BTCs in circulation is almost $64 million [4].

After BTC's success, blockchain technology began to be introduced into new fields to explore its full potential. Currently, it is being introduced in banks, healthcare, *Internet of things (IoT)*, government, etc [5]. In order to fulfill the requirements of each of these fields, there are several blockchain implementations, with different trust and scale assumptions. Therefore, we can distinguish two main blockchain types: permissioned, where the access to the network is restricted and controlled by a group of known nodes, and permissionless, where any node can access the network and does not trust any other node. In the context of this dissertation, the focus is on permissioned blockchain, such as HLF [6].

Blockchain, like any other technology, has some downsides. One of blockchain's main problems is related to the consensus protocol, which may have problems with scalability. For permissioned blockchain, the current consensus protocols are based in the classic distributed consensus protocols like state machine replication, which causes a large communication overhead for a large number of nodes. Several protocols have been proposed that aim at improving scalability.

1.1 PROBLEM

The blockchain handles the maintenance of a distributed register by several participants with integrity guarantees. It is essential that participants agree on a consistent view of

the global state of the register, thus requiring a consensus protocol. However, consensus protocols limit its performance.

Blockchain implementations such as HLF target a different environment from BTC . The consensus protocols used in this type of blockchain implementation differ significantly from protocols based on economic incentives such as BTC 's PoW, but they typically do not scale well with increasing numbers of participants. These protocols must have resilience against a variety of threats, for example, the crash of process, malicious activity, etc. Therefore they are based on two fault models: *Crash Fault Tolerance (CFT)* and *Byzantine Fault Tolerance (BFT)*. CFT builds a degree of resiliency in the protocol so that the algorithm can still correctly reach consensus, even if certain components fail. On the other hand, BFT ensures that every peer receives the same guaranteed data. This is why even faulty components or malicious attacks are unable to breach the protocol, *i.e.*, as long as two-thirds of the nodes are safe, the consistency of consensus can be ensured. Fault-tolerant protocols are based on these models, however, in a blockchain context, which has a large number of participants that can be malicious, they face some problems. For example, the *Paxos* protocol, which is CFT, is **typically limited to 5 nodes**, because of the increase of the number of peers, the leader needs to handle more messages, which degrades the protocol performance [7]. Another approach for CFT protocols use an ordering service to guarantees the consensus between the peers, such as HLF v1.1 that use *Apache Kafka*. Performance analysis of HLF Platform v1.1 was performed and shows that the bottlenecks lie in the communication overhead between the execution and ordering phase, even scaling the Kafka cluster does not affect the overall performance [8]. Besides these scalability issues, these protocols do not tolerate Byzantines participants, which makes them a weak option in a blockchain context. Therefore, BFT protocols are more reliable option. However, they present scalability issues as well. The HLF Platform v0.6, which use the PBFT protocol [9] as consensus protocol, was analysed in terms of performance. It stops responding beyond 16 nodes due to the communication overhead between nodes in the consensus protocol [10]. This protocol is based on state machine replication to tolerate faults at the level of the network and the nodes, which need a large exchange of messages to guarantee the consistency of the consensus in the presence of faults. Therefore, the problems begin to emerge if the number of participants increases, which calls into question the scalability of the protocol. Besides, there is another approach for HLF v1, which uses a BFT ordering service based on the *BFT-SMART*[1] library, to improve the resiliency of the consensus. However, this approach only evaluated up to 10 orderer nodes, and shows that with the increase of the number of orderer nodes, the throughput decrease [11].

In sum, the consensus protocol becomes a problem when the blockchain network increase, limiting its performance, mainly in terms of scalability.

1 https://github.com/bft-smart/library

1.2 OBJECTIVES AND CONTRIBUTIONS

To improve the performance of blockchain implementations, we need to find a new consensus protocol that solves the previous problems. Thus, it is interesting to consider the *Mutable Consensus Protocol (MCP)* [12], which can adjust several parameters through a range of mutations encapsulated in the protocol itself for blockchain implementation. Each of these mutations has specific features, in particular, the *gossip* mutation that scale well to a large number of peers, overcoming the scalability issues. However, these mutations are implemented through the adjust of the protocol parameters, which requires manual intervention.

Therefore, we propose to create an **adaptive consensus protocol** applicable to a blockchain implementation, which makes the configuration simple and automatic. To achieve this objective, this dissertation presents the following contributions: **get the best mutation for a given environment** and **the automatic adaptation of the protocol for a given environment**. To achieve these goals we use *Machine Learning (ML)* as the optimization mechanism, which gives the adaptive component to the protocol. To this end, we made an **implementation of MCP in a simulation environment**, which allows us to evaluate the effect of the protocol parameters on the system in a variety of environments and get data to train the ML models.

Besides, we improve the *ring* mutation in MCP, making it more resilient to faulty peers.

1.3 DOCUMENT STRUCTURE

The present dissertation is divided into several chapters to separate and simplify the explanation and description of it. The first and current chapter presents an introduction of the dissertation theme and what we propose to do. In Chapter 2, a literature review of the blockchain, consensus protocols, and optimization mechanisms is made. This chapter explains in detail the concepts and technologies related to the consensus protocol and presents the current state of the art about the most used consensus protocols on blockchain technologies. For each of those protocols, an analysis is made, that explains how the protocol works and which advantages and disadvantages it has. The next chapter, Chapter 3, describe in detail all the implementation of the consensus protocol, which is based on the MCP. Finally, Chapter 4 presents all the machine learning strategies applied, the respective problems faced and the results of the integration with the implemented protocol. In Chapter 5, a conclusion about the work done and the results obtained is made.

2

BACKGROUND AND RELATED WORK

Blockchain is known as a simple concept to store records in a chain of blocks. However, from a deeper perspective, it is a complex concept that depends on other concepts, as the consensus protocol, which is an essential component to it. The present chapter discusses all these concepts and presents the most known consensus protocols on blockchain.

In order to contextualize all the concepts addressed in this thesis, in this chapter, will be presented in detail the definition of blockchain, an example of permissioned blockchain, *i.e.*, HLF, which the thesis is focused, a description of a consensus protocol with an adaptive characteristic called MCP and finally, some optimization mechanisms are presented too.

2.1 CONSENSUS

Consensus is the problem, in distributed and multi-agent systems, of achieving overall system reliability in the presence of a number of faulty processes. More formally, considering a set of processes, at least some of which propose values, it is expected that when the protocol ends, all (correct) processes have decided on the same value, from the set of those proposed. The protocol must ensure the following properties [13, 14]:

- **Agreement**: No two processes decide differently;
- **Termination**: Every correct process eventually decides some value;
- **Uniform Validity**: If a process decides v, then v was proposed by some process;
- **Uniform Integrity**: Every process decides at most once.

Achieving consensus in a distributed system is challenging as both the network and participating nodes can be unreliable. Different protocols can tolerate different numbers and/or types of faults. Additionally, nodes can behave maliciously to undermine or take advantage of the system. The assumption of how trustworthy processes can be is fundamental for selecting an appropriate consensus protocol.

An important result, known as *Fischer-Lynch-Paterson (FLP)* Impossibility [15], demonstrates that in a completely asynchronous system, no consensus algorithm can assuredly

achieve consensus if failures can occur. Practical implementations require choosing between compromising consistency for the protocol to make progress, favouring liveness, or compromising progress to guarantee consistency, favouring safety.

2.2 BLOCKCHAIN

Blockchain was first introduced in 2008 when BTC, a digital currency, was proposed by Nakamoto [2]. Blockchain is a distributed ledger of transactions, stored in a chain of connected blocks. The structure of a blockchain is represented in Figure 1.

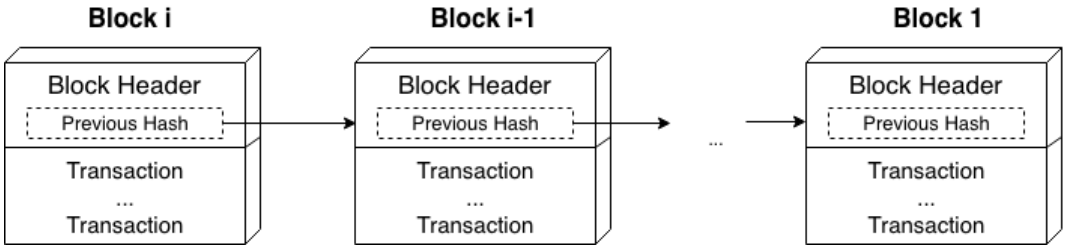


Figure 1: The blockchain structure.

Each block contains the cryptographic hash of the previous block in the chain that protects the blocks against tampering [11]. The technology allows records to be shared by all participants in the network, but no single participant maintains or owns the data [10]. The main feature of the blockchain is the decentralization that prevents an intermediary or centralized authority from controlling records and transactions. However, without a centralized authority, a way to persist records and validate each transaction is needed. As such, this technology uses a consensus mechanism, which makes the nodes in the network accountable for validating each transaction and add a group of validated transactions as a new block to the blockchain. Therefore, the consensus protocol ensures an unambiguous ordering of transactions and blocks and guarantees the consistency of the blockchain [1].

Blockchain can be classified based on the permissions of access to the network and add a new block, and the trust model. There are two main types [10]:

- **Permissionless**: any node has the permission to join the network, participate in the transactions and create new blocks. However, the nodes are untrusted, which makes the consensus mechanism a proof-based mechanism, like BTC;
- **Permissioned**: a solution which the network is controlled by a group of known nodes. Unlike the permissionless type, there is a group of trusted nodes that ensure the consistency of the blockchain. This type of blockchain is seen in platforms as HLF.

However, like any other technology, blockchain faces some challenges. The global performance of the blockchain is influenced by the different assumptions of blockchain types

[16], *i.e.,* the different trust and permission assumptions influence the global performance of the blockchain. On the one hand, the permissionless blockchain, allows anyone to access the network. Thereby, this blockchain type scales well for a large number of participants, handling thousands of nodes. On the other hand, this type of blockchain presents high latencies and low transaction rate caused by the consensus protocol, such as PoW, which forces the nodes to solve a hard cryptographic puzzle as proof to earn the right to add a new block. Unlike the permissionless, the permissioned blockchain presents low latencies, and high transaction rate, because the nodes do not need to do any proof and the consistency is ensured by a group of known nodes. However, in terms of scalability, this blockchain type does not scale well because of communication overhead caused by the consensus protocol, like PBFT.

Each blockchain type presents different consensus protocols. The permissionless blockchain has, for example, PoW[2] and PoS [17], and the permissioned blockchain has, for example, PBFT [18] and Kafka [19]. Each of these protocols are described in the following subsections. However, in the context of this dissertation, we are the focus on permissioned blockchains, such as HLF, which use classic distributed consensus as consensus protocol.

2.2.1 *Permissionless Blockchain*

A permissionless blockchain has fully decentralized identity management, where any node can join the network and add blocks to the blockchain. For example, in BTC, anyone can download the code to be a BitCoin miner, and start participating in the protocol [16]. Therefore, a consensus protocol for a permissionless blockchain should take into account this kind of access without any restrictions.

Proof-of-work

Proof-of-work (PoW), is a consensus protocol used in the BTC network. In this protocol, to add a new block to the blockchain, it is necessary to produce a proof that corresponds in doing some computer processing. The proof consists to calculate a hash of the block header that is smaller than a target value and begins with a set number of zero bits. This test is associated with a difficulty level that is dynamically adjusted by the BTC protocol, in order to guarantee that a new block is generated every ten minutes, regardless of the computational power of the network. Although significant computational effort is required to calculate a hash, the hash validation process is very fast. The nodes that calculate the hash values are called *miners* and the PoW procedure is called *mining* [20].

The PoW process flow is described in Figure 2. All the transactions submitted to the blockchain are validated by the miners. They gathers the transactions into a block and tries to add it to the blockchain, but they need to solve the puzzle challenge given by the

network. First the miners receive the challenge. The process to calculate a hash consist of getting different hash values, changing the nonce value that belongs to the block header until getting a hash value that matches the target value. Among all nodes, the first node to find a matching hash, *i.e.*, a winning hash, can add its proposed block to the blockchain, broadcast it to other nodes so that they confirm the correctness of the hash value and add the block to their copy of the blockchain. The award is processed and validated just like any other transaction, not explicitly sent to the winning node. After the block is added to the blockchain, it can not be changed without redoing the related work. If it has a set of chained blocks after it, the work required to change the block includes redoing all the chained blocks after it [2].

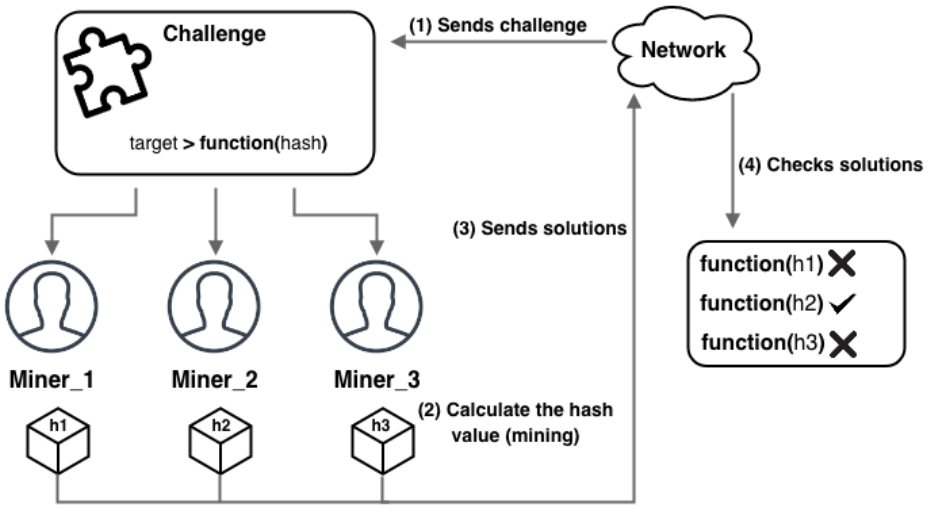


Figure 2: The PoW process flow.

As the system is distributed, all the mining processes are running concurrently, which may cause more than one node to find a winning hash. So, each node of these adds its own proposed block to the blockchain, creating a temporary fork in the blockchain. With time, the created branch increases. However, the protocol ensures that the branch with the maximum PoW, *i.e.*, the largest branch, will be the blockchain and the others will be discarded. Therefore, the PoW protocol guarantees eventual consistency in the BTC blockchain even though temporary forks [20]. This process is present in Figure 3.

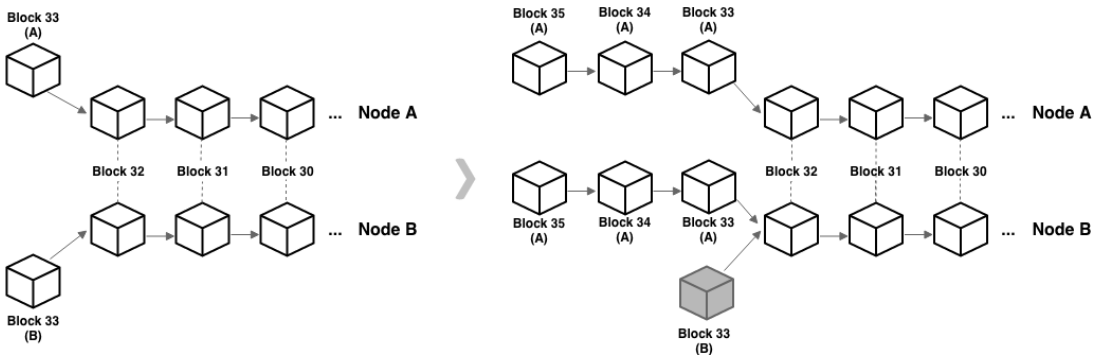


Figure 3: Example of a fork and conflict resolution.

Besides, by convention, a transaction is only considered committed after 6 subsequent blocks have been added, to avoid double-spending, *i.e.*, spend a certain amount of BTC twice in two separate transactions. Another problem is the 51% attack, which is a potential attack on a blockchain network, where a single entity can control the majority of the hash rate, potentially causing network disruption. However, since the blockchain is maintained by a distributed network of nodes, all participants cooperate in the process of reaching consensus. This is one of the reasons they tend to be highly secure. The bigger the network, the stronger the protection against attacks and data corruption.

This consensus protocol has excellent node scalability and operates completely in a decentralized fashion. However, this protocol has a few drawbacks too. The requirement to solve a hard cryptographic puzzle causes high latencies, slow transaction rate (approximately 7 transactions/second), and significant energy expenditure, in which the participants, as miners, have to be conscious of the related costs. In the world of payments, namely *MasterCard* or *VISA*, the transactions rate achieves 10000 transactions/seconds, which is much higher than the previous value. [16].

Proof-of-Stake

Proof-of-Stake (PoS) is an alternative consensus protocol to PoW, which is designed to overcome the high electricity consumption that the PoW mining operations require. This protocol uses, as proof, a user's stake or ownership of virtual currency in the blockchain system, replacing the mining operation [17]. In this protocol, validators (instead of miners), have to prove the ownership of currency. In a different perspective, instead of a user investing around $2000 in mining equipment to use as a miner in PoW, the user can invest $2000 worth of cryptocurrency by becoming a validator and uses them as a stake to buy opportunities to create a new block in the blockchain system. In short, for a node in a blockchain network which uses PoS, the more it owns, the higher are the odds of processing the next block and receiving the transaction fees as a reward. In addition, the amount of coins that

is necessary to stake is specified by the network through an adjustment process, as in the PoW. In this way, this process guarantees an approximate constant block time too.

This protocol presents some differences in its process flow compared to PoW. The PoS process flow is present in Figure 4. Each validator who wants to stake, need to deposit in order to participate in the process of creating blocks. In the example, Node A invests more coins than Node C, increasing its chance to win. After that, the validator is selected in a deterministic (pseudo-random) way that takes its stake into account. In this case, Node A was selected. Once a validator is selected, it has the exclusive rights to create a new block, add it to the blockchain and receive the related reward. However, if the validator misbehaves then it will lose their stake. This validation process gives the validator the incentive to behave honestly.

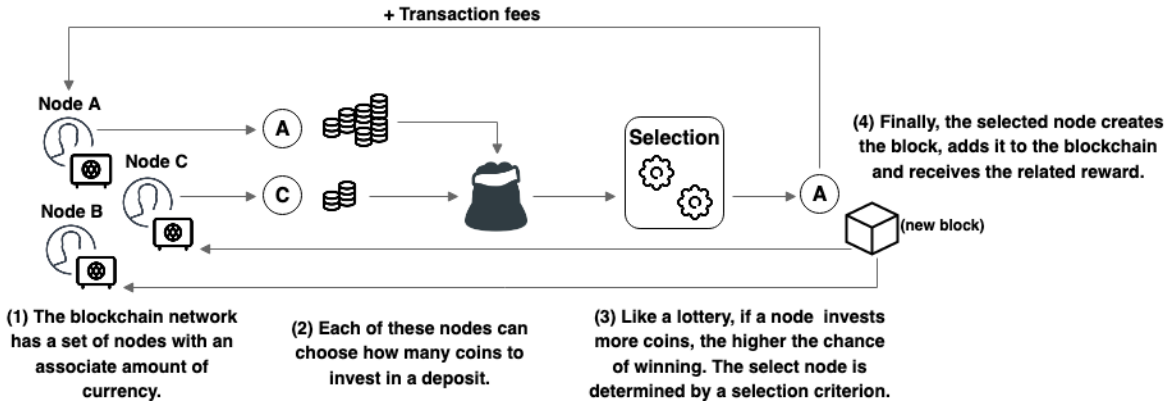


Figure 4: The PoS process flow.

Node selection based on account balance is unfair because the richest node would be dominant in the blockchain network. Two solutions are: randomization used by BlackCoin [21] and coin age used by Peercoin [17].

PoS has some problems too, but the one who stands out is the *Nothing at Stake*. Unlike PoW, where miners must choose which chain they want to mine in order to avoid waste of resources, the validators in PoS can stake their coins on every fork because it is free of costs. Therefore, if a node votes on multiple blocks supporting multiple forks, it can maximize their chances of winning transactions fees. This will be disruptive to consensus, preventing it from being achieved.

BlackCoin The BlackCoin blockchain uses a randomized block selection method. The validators on that method are selected based on a formula that looks for the nodes with a combination of the lowest hash value and the highest stake. Since the stakes from each node are public, the nodes can predict the next node to win the right to forge a block [21].

Peercoin The generation of blocks in Peercoin is based on coin age, which is a factor that increases with time and is defined as the currency amount times the retention time.

For example, if a node received 10 coins and held it for 10 days, the node has accumulated 100 coin-days of coin age. The coin age accumulated by these coins is consumed when the node spends the coins [17]. Once a stake of currencies has been used to create a block, they must start over with zero coin age. In order for a validator to be selected, it is necessary to satisfy the following condition:

$$proofhash < coin \cdot age \cdot target \tag{1}$$

2.2.2 *Permissioned Blockchain*

Unlike the previous blockchain type, a permissioned blockchain requires centralized identity management in which a trusted component ensures and sends respective identities and cryptographic certificates to the nodes. However, after an initial bootstrap of the blockchain, the system could be configured by the nodes themselves acting as a distributed trusted component. This feature may be a requirement for certain blockchain applications, for example, a bank's application that imposes it for legal and compliance reasons [16]. Thus, this blockchain type requires a consensus protocol that assumes to know the entire set of nodes participating in consensus. An example of permissioned blockchain is HLF.

HyperLedger Fabric

Hyperledger is an open source, permissioned, distributed ledger with several subprojects founded by Linux Foundation, which we only focus on the HyperLedger Fabric (HLF) project.

HLF, or simply Fabric, is a modular permissioned blockchain platform that is maintained by *International Business Machines Corporation (IBM)* and Linux Foundation. It was designed as a foundation for developing applications with a modular architecture, which supports pluggable implementations of different components such as consensus and membership services [11]. Furthermore, as HLF supports modular consensus protocols, it allows the system to be adapted to particular use cases and trust models. It also supports the execution of distributed applications, which makes it the first distributed operating system for permissioned blockchains [6].

Unlike BTC, in which the access is public and has a cryptocurrency, HLF has restricted access and does not have any cryptocurrency. A group of known participants controls access and transactions in this blockchain. In order to obtain the privacy of the transactions between the participants in the network, HLF uses an isolation mechanism known as Channel. This channel ensures that only the members of the channel have the permissions to access the transactions and data exchanged. In addition, HLF uses chaincode (smart con-

tract), which implement the application logic that interacts with the network and handles transactions [10].

Consensus in HLF covers the entire transaction flow, *i.e.*, since proposing a transaction to the network until committing the transaction to the ledger. Nodes assume different roles:

- **Clients**: They act on behalf of an end-user and they are responsible for invoking transactions. This node sends a message to an endorsing peer that invoke a chaincode function, and after some validations, creates a signed envelope as a transaction proposal to be delivered to the ordering nodes.
- **Endorsing Peers**: They maintain the ledger and receive ordered update messages for committing a new transaction to the ledger. Basically, they execute chaincode, access ledger data, endorse transactions and interface with applications.
- **Ordering Nodes**: Gather envelopes from all channels sent by the clients, ordering them using atomic broadcast and consequently ensuring the consistency of the blockchain. They also create signed chain blocks containing these envelopes.

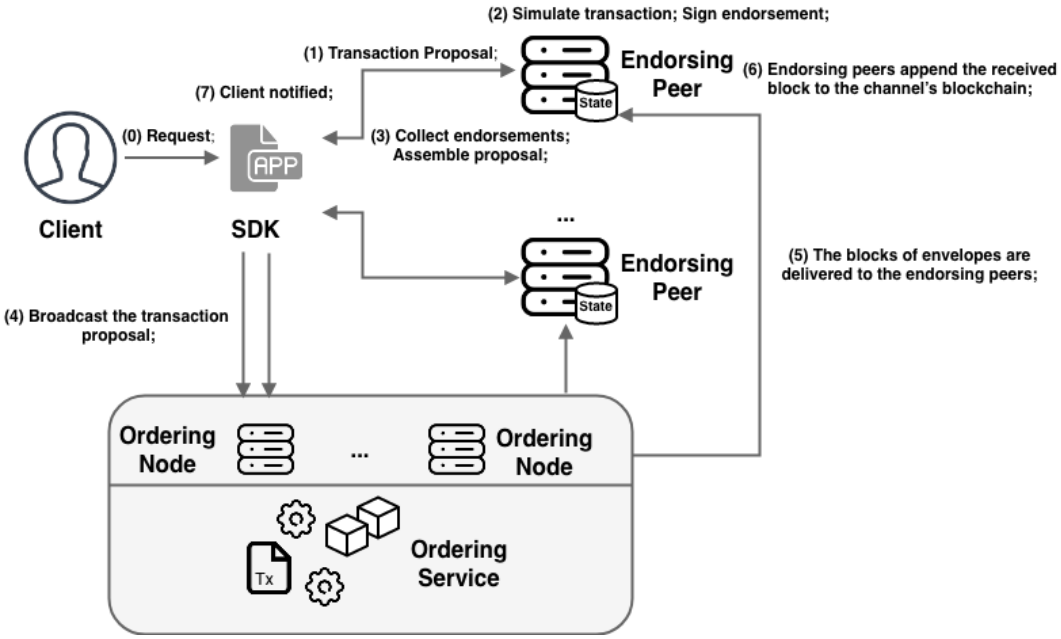


Figure 5: Illustration of the HLF transaction flow.

Figure 5[1][2] represents the HLF transaction flow. It can be represented by a client sending a request to purchase apples through the client application *Software Development Kit (SDK)*. The application builds a transaction proposal and sends it to the endorsing peers, where the

1 https://hyperledger-fabric.readthedocs.io/en/release-1.4/peers/peers.html
2 https://hyperledger-fabric.readthedocs.io/en/release-1.4/txflow.html

transaction proposal is a request to invoke a chain code function. So, when the endorsing peer receives the proposal, it validates the proposal. If the proposal is valid, the peer simulates the transaction with the current state of the database, without updating it. The result of the simulation and the endorsing peer's signature is sent to the client application as a response to the request. The application validates the response and if the request only queried the ledger, the response is inspected and nothing is broadcast to the ordering service. If the request has the intention to add transaction and update the ledger, the application broadcast the transaction to the ordering service as a signed envelope, through the channels. The ordering service gathers all the envelopes, ensure the order of them and creates blocks of transactions to deliver to the endorsing peers. The endorsing peers update the ledger and append the received block. Finally, a notification is sent to the client application, to alert of the transaction result.

In HLF, there are several consensus protocols, depending on its version: PBFT and Kafka. PBFT protocol is the only available in a previous version 0.6 of HLF, and Kafka is currently an option for the recent version 1.X, integrating the ordering service.

Practical byzantine fault tolerance

With the growth of systems and software complexity, malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit Byzantine behavior [22]. Because these arbitrary behaviors can happen in a distributed system, the consensus algorithm must be resilient against them. So, in order to prevent such behaviors and ensure the high availability of the system, a BFT algorithm is necessary [9]. PBFT is an algorithm proposed by Miguel Castro and Barbara Liskov [9], and also the first practical solution to reach a consensus that tolerates Byzantine faults.

This algorithm implements state machine replication, that is, it is implemented as a state machine that is replicated across different nodes in a distributed system, where each node has the same state and implements the same operations. This allows the system to handle a maximum of f Byzantine failures on $3f+1$ nodes. Although for f faulty nodes there may be more than $3f+1$ nodes, surplus nodes provide greater communication overhead due to more exchanged messages and larger messages, and do not improve system resiliency.

In HLF, only version 0.6, implemented this protocol as a consensus protocol. In version 0.6 of HLF, a new block is determined in a round. For each round, a node is selected as a primary node and is responsible for establishing the order of transactions in the block. In addition, the consensus process is complex and can be divided into three phases: *pre-prepared*, *prepared*, and *commit*.

The consensus process is present in Figure 6. The process begins when a Client sends a *request* to invoke an operation on the primary node, Node A, such as a chain code function. Node A multicasts a *pre-prepare* message to the replicas, containing the *request* message,

a sequence number, the view number and the message digest that guarantees message validity between phases. Each replica, Nodes B and C, accepts the *pre-prepare* message as long as it is valid by checking the message metadata, expect Node D that fails before to receive the message ($f = 1$). If the *pre-prepare* message is accepted by a replica, it follows up by sending a *prepare* message to everyone else. As in the *pre-prepare* message, the *prepare* message is checked based on its metadata. A node is considered *prepared* if it has received the original request from the primary node, has *pre-prepared*, and has received $2f$ *prepare* messages that match the *pre-prepare* message. Therefore, the *pre-prepare* and *prepare* phases guarantee that non-faulty replicas nodes agree on total order for the requests in a view. Once the nodes are *prepared*, they multicast a *commit* message. If a node receives at least $f+1$ valid *commit* messages, it will execute the client request and then finally send the reply to it. The Client waits for $f+1$ replies that ensure the response is valid and correct. In this way, these three phases guarantee the order of transactions with faulty nodes [9]. In the example, though the Node D failed during the process, the algorithm was able to reach the consensus and thus reply to the Client.

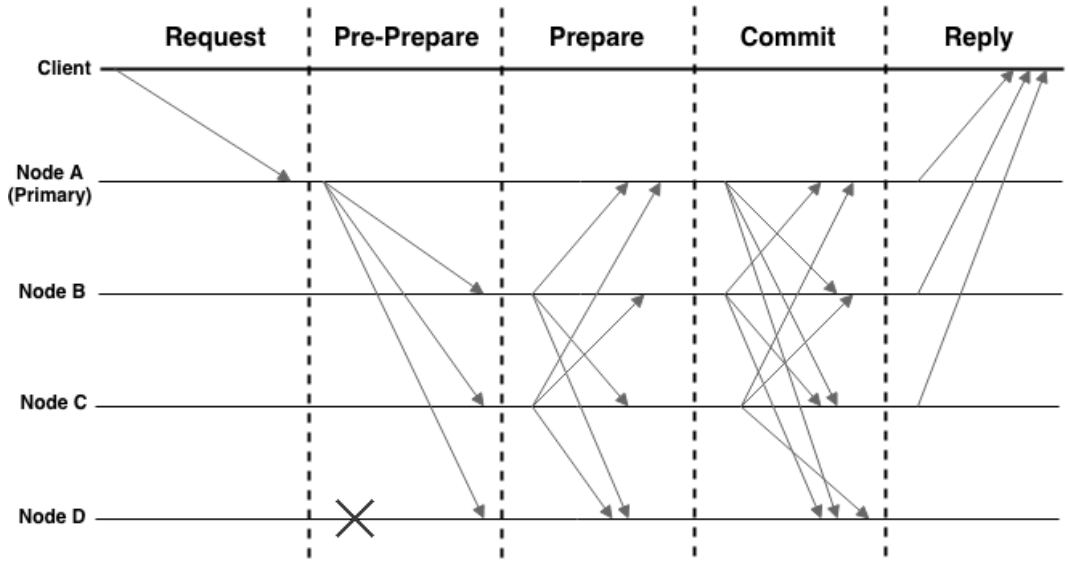


Figure 6: Consensus process in the PBFT algorithm.

Last but not least, although this algorithm sustains thousands of requests per second with low latency, those three phases require a lot of message exchange. So, if we increase the number of nodes in the blockchain system, this protocol does not scale well because of the exponential increase of number of messages exchanged by the peers, *i.e.*, a complexity of $O(N^2)$ messages per block.[16]. Indeed, it has only been scaled and analyses to 16 nodes [10].

Kafka

Consensus protocol in HLF has been reworked with the version v1, where nodes are divided into different components and services. The previous version was designed with state-machine replication to reach consensus, but now HLF has created an ordering service that is responsible for avoiding any conflicts, guarantees the order of transactions, and thus reaching the consensus.

The ordering service can be provided by an Apache Kafka [19] cluster. Kafka is a distributed messaging system that collects and deliver high volumes of log data with low latency. It combines the best of traditional log aggregators and messaging systems. The log aggregator only appends messages to the data structure, avoiding read/write locks, which makes the complexity O(1) the worst case. The messaging system is based in the popular *Publish-Subscribe* model. A stream of messages is defined by a *topic*. While a *Producer* publishes messages to a topic, a *Consumer* subscribes one or more topics to receive a new message. The messages are stored in a set of servers defined by the brokers. Therefore, when a Producer publishes a message on a topic, the message is stored in a broker. On the other side, a Consumer who subscribes to the topic, consumes the message, by pulling the data from the broker. When the topics get bigger, they are split into partitions to balance the load, and Kafka ensures that all messages inside a partition are sequentially ordered and immutable [19].

Kafka was designed considering the CFT fault model, where the system can still correctly reach consensus if components fail. The partitions are replicated among the multiple brokers. A leader owns a partition, and the follower has a replication of the same. When the leader dies, the follower becomes the new leader. Therefore, if one broker dies due to a software fault, data is preserved. However, when a consumer wants to subscribe a topic, he needs to know which leader to grab. The Zookeeper Service [19] solves this problem. Zookeeper is a distributed key-value store, used to store metadata and handle the mechanics of clustering. It allows the brokers to subscribe and have changes sent to them once they happen. With this, the brokers may know when to switch partition leaders.

In HLF, Kafka is integrated with the ordering service. The ordering service nodes can consume a partition and get an ordered list of transactions that are equal across all nodes. A timer service is used to batch the transactions in a chain. That is, a timer is set when the first transaction for a new batch arrives. In addition, each ordering service node maintains a local log for every chain (partition), and the new blocks are stored in the local ledger. Once all of them have a replica of the data, when one node crash, the relays can be sent by another node.

Figure 7 present the ordering process. Client broadcasts the transactions *Tx A* and *Tx B* to the ordering service nodes 1 and 2 respectively. These nodes forward the incoming transaction to the *Partition 1* on Kafka, which order these transactions. On the other side,

the *Partition* 1 is also consumed by an ordering service node which gathers the received transactions into blocks locally, persists them in its local ledger. Now, a peer can connect to an ordering service node and sends a request, saying that it wants to receive the new block. Then the ordering service node reads the block from its local ledger and sends it to the peer.

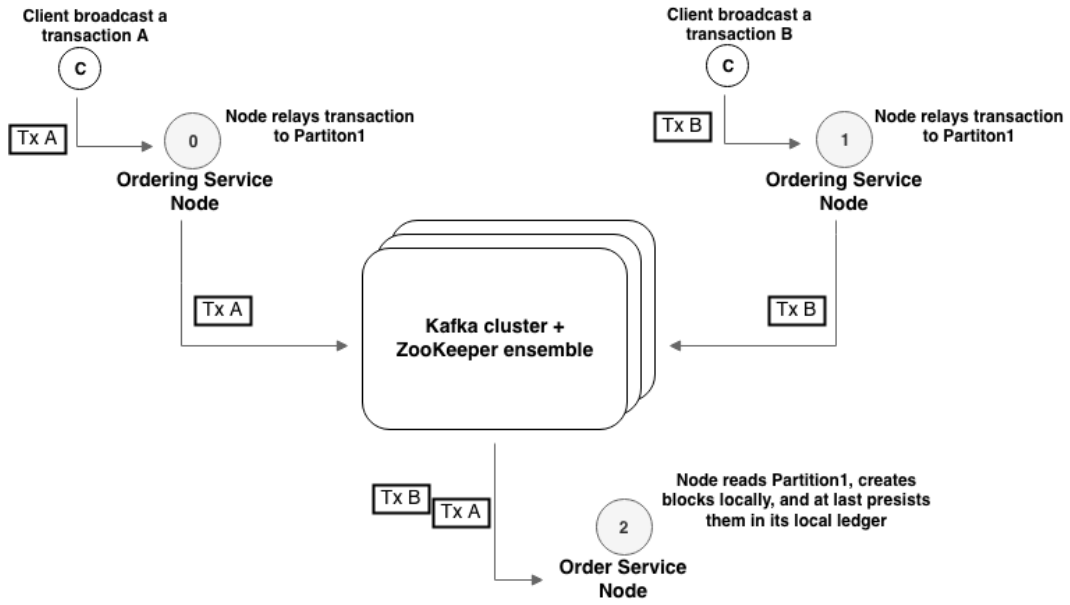


Figure 7: Example of interaction between Kafka cluster and Ordering Service Nodes.

Kafka helps HLF ordering service to guarantees the order of transactions with high performance, by connecting two ordering services nodes with a stream, avoiding any conflicts in the order of blocks. However, this protocol does not tolerate malicious nodes in the network, as PBFT does, becoming a weak option.

2.3 ADAPTATION

Nowadays, there are several consensus protocols for permissioned blockchains, but they do not scale well with increasing number of participants. An attractive option can be the MCP [12].

The MCP solves the consensus problem, tolerating the crash of a majority of processes. For such, it assumes an asynchronous distributed system model augmented with an eventual strong failure detector, $\Diamond S$ [12]. All process are connected through fair-lossy communication channels, which any message that is sent has a non-zero probability to be delivered. The topology can be considered a fully connected graph. Over these channels, this protocol uses Stubborn Channels [23], that abstract the communication implementation and offer a simple and powerful interface over them. In brief, to implement a Stubborn Channel over a

fair-lossy channel we only need to buffer the last message sent and retransmit it periodically until the destination confirms the reception.

MCP only sends a subset of messages over the network, because of the delay added in the messages. This delay decides when a message is sent or not. In some mutations, some messages will be sent immediately while others only after a period of time e (an estimate on the time consensus will take). This delay prevents unnecessary sending of messages. This property can define a type of mutation, such as [12]:

- **Early**: assigns to each message zero delay. In other words, the process broadcasts the message to all other process;
- **Centralized**: only messages to and from the coordinator process are immediately sent while the others are delayed;
- **Ring**: only messages addressed to the next process (in a logical ring) are immediately sent;
- **Gossip**: each process has a permutation of the list of all processes and sends the message immediately to F processes (gossip fanout) and delays it to the others. This set of processes changes for each broadcast.

The mutable consensus algorithm does not depend on these mutations, *i.e.*, no matter what mutation is used, the algorithm's correctness is not affected. This makes MCP a good option for dynamic environments where settings may change. Thereby, this protocol can be a good option for a blockchain implementation, because it can adjust to a particular environment and tolerating faults avoiding the communication overhead. However, it can not currently adapt automatically because there is no way to determine the optimal settings, and it is also not easy to predict such configurations.

In short, MCP provides a framework that makes it possible to seamlessly adjust the trade-offs on network usage, processing load, and fault tolerance through a range of mutations encapsulated in the protocol itself. This makes it adaptable and an attractive tool to solve consensus in a wide range of environments.

2.4 Optimization mechanism

The goal is improve the protocol performance, but there is a problem in finding the best setup for a specific environment. The settings have an impact on the number of supported nodes because they influence the number of exchanged messages between the nodes, the decision latency and load balance among the different nodes. For example, if the load increase with the increase of the number of participants, the ideal would be to balance it

by all of the nodes, relieving some overloaded nodes. With optimization, the aim is to find the best setting for a given environment, making also the protocol adaptive to it. Machine learning (ML) [24, 25] is explored for the optimization mechanism.

ML aims to introduce the ability to learn on computers without these being explicitly programmed. More precisely, it is a branch of *Artificial Intelligence (AI)* based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention. Most industries working with large amounts of data have recognized the value of ML technology, using it to web mining, decisions involving judgment, screening images, load forecasting, diagnosis, marketing, and sales, etc [24].

The main goal of ML is to understand the structure of the data. It has developed based on the ability to use computers to probe the data for structure, even if we do not have a theory of what that structure looks like. As ML generally uses an iterative approach to learn from data, the learning can be easily automated. The steps are performed through the data until a robust pattern is found.

Finally, with ML it is possible to find patterns in the data. If the data are the parameters of the consensus protocol, it may find the best setting/mutation for a specific environment. The general idea would be to apply ML techniques to data collected from running mutable consensus in a variety of environments, with different parameter configurations with the goal of then formulating predictions for the best parameters/mutation for a specific environment. Another approach is from a mutation, collect all different states of the protocol parameters in each peer to make a *Deep Learning (DL)* model learns from it and replicate the mutation. However, this approach can be extended and to be possible replicate more than one mutation.

The following subsections describe widely used techniques/models and how their training process and prediction works.

2.4.1 *Machine Learning model*

Choosing a model depends on what kind of input data we have and what kind of output we want. **Random Forest** is a supervised learning algorithm and can be used both for classification and regression. It is called Random Forest because it creates multiple decision trees on randomly selected data samples. For each decision tree, it gets a prediction and selects the best solution of all the trees by means of voting. A decision tree can be compared to a series of yes/no questions about our data that eventually leads to a predicted class or, in the case of regression, a continuous value. It is an interpretable model because it makes classifications much as we do: we ask a sequence of queries about the available data we have until we arrive at a decision. Therefore, the more trees the Random Forest has, the more robust it is.

Training process

This step is the step where we use the data to incrementally improve the model's ability to predict better results until it reaches an accurate model that predicts the results correctly most of the time.

When the different sets of data have a linear relationship between them, the model can be represented by a straight line. The formula for a straight line is `Y = m*X + b`, where `m` is the slope of that line, `X` is the input, `b` is the `Y`-intercept, and `Y` is the value of the line at the position `X`. Only the values `m` and `b` are available for adjusting, which allows the model to be trained. In machine learning, there are many `m` since there may be many input parameters. Once we have several input parameters, they are represented as a matrix and denoted as `W`, the *weights* matrix. The same thing happens with `b`, the *biases*. The training process involves initializing some random values for `W` and `b` and attempting to predict the output from those values. Therefore, we can compare the model's predictions with the output that it should produce, and through an iterative process, adjust the values in `W` and `b` such that the model gets more correct predictions. However, in this case, we do not have a linear problem. The many combinations of parameters and the respective output give us a non-linear problem. Because of that we use the Random Forest algorithm. In a more detailed perspective, the Random Forest uses several decision trees. Each of them breaks down a dataset into smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with **decision nodes** and **leaf nodes**.

Prediction

In a decision tree, to predict the output value of a record, we start from the root of the tree. We compare the value of the root parameter with the record's parameter based on the node condition. We follow the branch corresponding to that value and jump to the next node. We continue comparing our record's parameter value with other internal nodes of the tree until we reach a leaf node.

2.4.2 *Deep Learning model*

The Deep Learning(DL) model works with an Artificial Neural Network (ANN), which is organized into layers of parallel nodes. The layers can be:

- **Input layer**: is the layer to which the features are passed as input. There is no computation in this layer. It serves to pass features to the hidden layers.
- **Hidden layer**: are the layers between the input layer and the output layer. These layers perform computations and pass the information to the output layer.

- **Output layer**: represents the layer of our ANN that will output the results after training the model. It is responsible for producing the output variables.

The ANN is a fully connected network, where it learns using **back-propagation** and **gradient descent**. Back-propagation is the technique used to calculate the error between a guess and the correct solution, provided the correct solution over this data. So any training step includes calculating the gradient (*i.e.,* derivatives) and then doing back-propagation (integrating the gradient to get back the way the weights should change). Gradient descent is an optimization strategy to minimize errors during the training process. Indeed, it is trying to reach the minimum of the *loss function*, with respect to the parameters, using the derivatives calculated in the back-propagation phase. The loss function is a measure of how well an ANN performs based on the output expected from it, *i.e.,* measure the difference between our model predictions and the output that we want to predict.

Training process

Each layer in the ANN has a set of nodes. Each node in a layer takes in some number of inputs, $x_1, x_2, ..., x_n$, each of which is multiplied by a specific weight, $w_1, w_2, ..., w_n$, resulting in what is called the internal value of the operation. These weighted inputs are, as before, summed together to produce the *logit* of the node, which can include a *bias*. This value is further sent to an *activation function* to map its output. This formula integrates all operations:

$$output = function(\sum_{i=0}^{n} w_i x_i + b) \tag{2}$$

The output of that function is then sent as the input for another layer, or as the final response of a network should the layer be the last.

Training a model is just minimizing the *loss function*, and to minimize we want to move in the negative direction of the derivative. Back-propagation is the process of calculating the derivatives and gradient descent is the process of descending through the gradient, *i.e.,* adjusting the parameters of the model to minimize the loss function. In other words, gradient descent is how randomly assigned weights in an ANN are adjusted by reducing the *loss function*. In gradient descent, you differentiate to find the slope at a specific point and find out if the slope is negative or positive — we are descending into the minimum of the *loss function*.

There are several types of optimization strategies that improve the model by minimizing the loss function, making the model learns faster. For example, a popular strategy used is **adam**. In brief, it is an adaptive learning rate method, which computes individual learning rates for different parameters.

Besides, the *loss function* is also an important component of the model. Depending on the problem that we are solving, the loss function can change. If we have a regression problem, the *Mean Absolute Error (MAE)* is a valid option as *loss function*. It calculates the average of the absolute difference between the actual and predicted values. On the other hand, if we have a binary classification problem, where we only have two possible outputs, the *Binary Cross-Entropy* can be used. The cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events.

Prediction

After complete the training process, the ANN has the best values for the *weights* and *bias*. Once they are defined, the prediction process is simple. The input parameters are given in the input layer, and the output of the activation function is the input for the next layer. The values are passed between the layer until they reach the output layer, and give the prediction.

2.5 DISCUSSION

We presented some consensus protocols that are used in blockchain implementations. However, they face some performance issues, namely the blockchain permissioned, which does not scale well with the increase in the number of peers. Therefore, we create an adaptive consensus protocol that overcomes these limitations, from the MCP. The MCP is a good candidate for that due to its mutation mechanism that allows changing the exchange message pattern of the protocol without compromise its correctness. Besides the flexibility that the protocol present, the protocol implements the mutation *gossip*, which makes the protocol scale well to increase the number of peers.

Even though the protocol presents good flexibility, its configuration is manual. Therefore, from the previous protocol, we integrate a ML component to make the protocol adaptive, with an automatic configuration and be able to set the best mutation to a given environment.

3

SIMULATION AND DATA COLLECTION

The ML model requires useful data that allow it to learn something. Therefore, the process to collect metrics from the protocol needs to be the most realistic and correct possible. Otherwise, the model will learn wrong patterns and features. So, we implement a simulator that simulates the protocol in a given environment, which allows collecting metrics from each node that runs the protocol. In this way, we can get useful data to the model.

The first step is to implement the whole protocol, *i.e.*, algorithms, message exchange, fault handling, and mutations. Subsequently, analyze their behavior in various environments and collect metrics that will feed the ML model.

3.1 IMPLEMENTATION

The protocol is implemented in *Go*[1], a programming language that makes it easy to build distributed systems and often described as a concurrent-friendly language. The reason for this is that it provides a simple syntax over two powerful mechanisms: **Goroutines** and **Channels**.

A Goroutine is a lightweight "thread" of execution, which is very easy to create with a few lines of code. Indeed, a Goroutine is comparable to a thread, but it is scheduled by *Go*, not by the *Operating System (OS)*, which improves its performance. Multiple Goroutines will end up running on the same underlying OS thread, which avoids the creation of more threads.

Channels are the mechanism that lets Goroutines communicate with each other. In other words, a Channel is a communication pipe between Goroutines, which is used to pass data. A Goroutine that has data can pass it to another one via a Channel. This feature makes it simpler to implement a distributed system that requires multiples processes that run simultaneously and communicate with each other.

Moreover, once the protocol is implemented in *Go*, there is a possibility of integration with HLF [6] in the future, which has the same codebase.

1 https://golang.org/

The protocol has two different operating modes: native with *User Datagram Protocol (UDP)* and with channels for simulation. The first one is designed with separate processes where they communicated through Stubborn Channels that use the UDP. The another one is a protocol simulator that runs the protocol in a single process, which supervises all the subprocesses that it creates. These subprocesses communicate through the channels that *Go* offers. This type of environment is better for testing communication parameters because it allows for greater control over the network environment.

In the Simulator, when the protocol runs, the supervisor is waiting for all peers to decide and reach consensus. After a peer decides, it replies with metrics collected during the protocol. The Simulator's architecture is present in the following image:

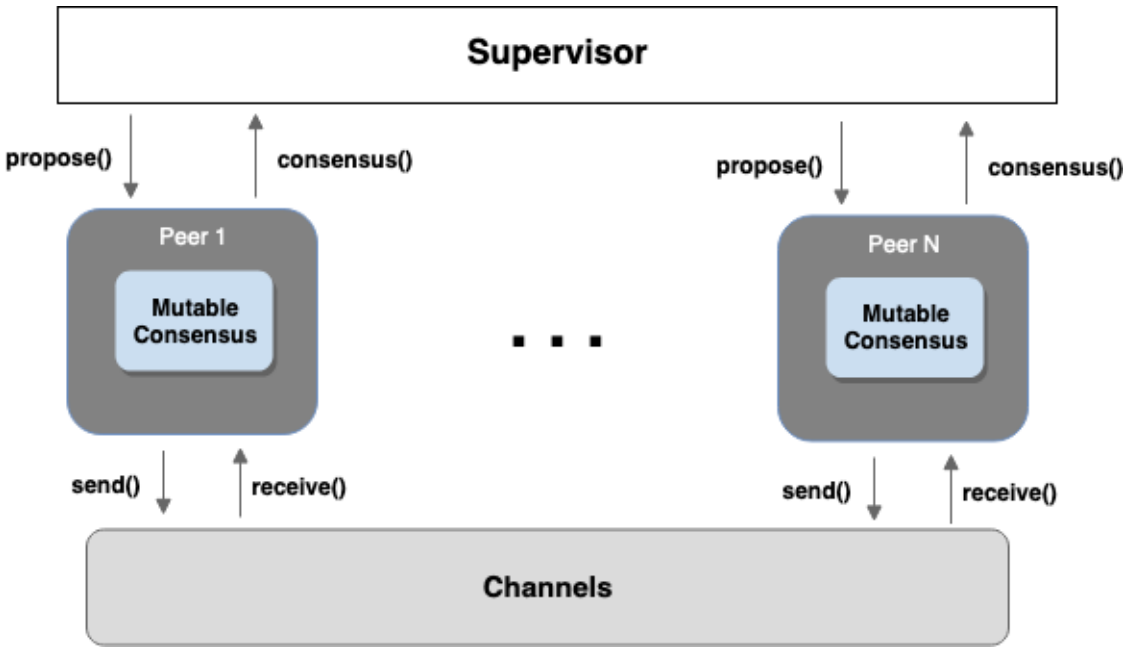


Figure 8: The Simulator's architecture.

Each peer runs the MCP with the same mutation, which defines the behavior of the peer in the consensus flow. Each mutation behaves differently in the same environment, but this point will be discussed in the next sections. When a peer reaches the consensus, it notifies the Supervisor. All messages exchanged between the peers, go through the Stubborn Channels, which are part of the implementation as well. The simulation ends when all peers report the consensus.

The implemented protocol is modular with different components, which allows the protocol to be analyzed quickly and clearly. The following sections will explain each protocol component to clarify the entire implementation.

3.1.1 *Peer*

The peer represents the process that runs the protocol and is the most complex component in the implementation. It aggregates all other components and uses them to make the protocol work. The following figure illustrates the peer structure:

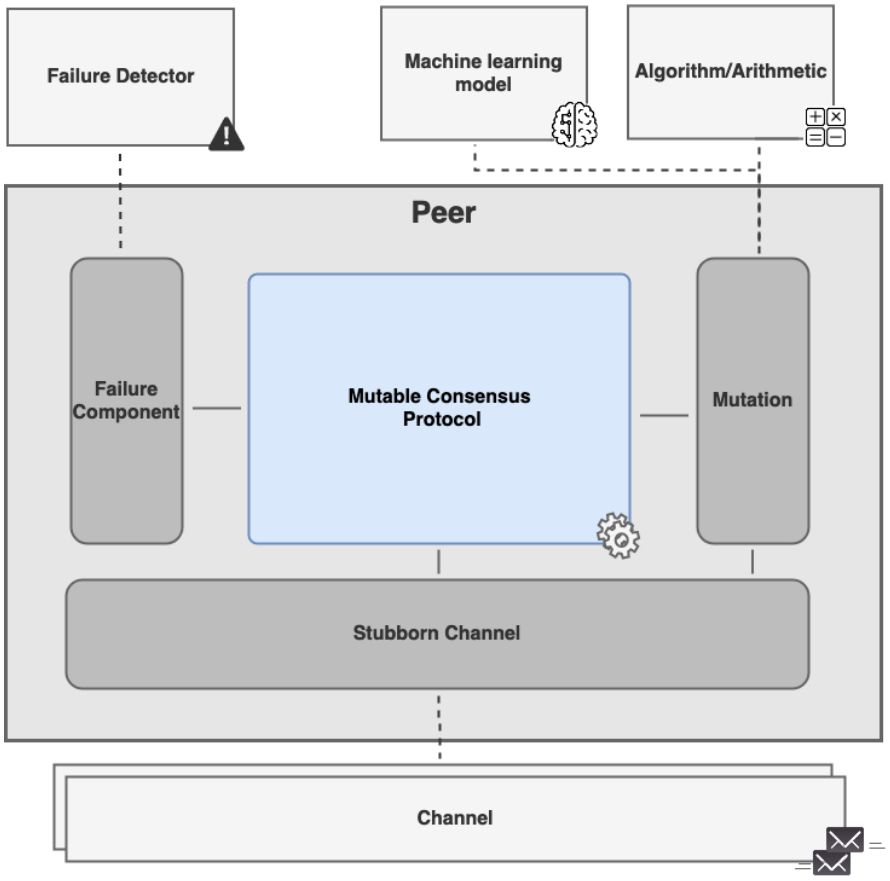


Figure 9: The Peer structure.

The MCP is the core of the peer. It defines all algorithms and decisions that a peer needs to follow and uses all other components, *i.e.*, the Stubborn Channel to send and receive messages, the Mutation to filter messages that are to send and the Failure Handler that handles the suspicions of failure given by the Failure detector. Also, each peer has a Failure Trigger component, which is responsible for simulating the peer failure. The Failure Handler and Failure Trigger are part of the Failure Component.

The implementation is well structured, allowing to isolate the functionalities in different components. When a feature implementation needs to be changed or removed, we replace its component. For example, there are several simulated components in the protocol that needs to be changed to allow the protocol to be deployed in a real environment. However, only Stubborn Channels and the Failure Detector need to be replaced with a real component.

The other components are smaller and only need to be removed. These components are responsible for simulating network limitations, *i.e.*, message loss, bandwidth, and latency.

The peer has several components, but most of them are related to the implementation of the MCP. Only the Failure Component is directly related to the peer, which enables failures to happen. These components are described below.

Failure Detector

The Failure Detector is responsible for guarantees that all peers know about eventual failures. This component is a shared structure between all peers, which contains all their status and controls the number of faults that can happen between them. Whenever a peer wants to know if a peer is still alive, it asks the Failure Detector. For that, there is a process running in a loop in each peer, called Failure Handler, that checks if a random peer still alive, every 100 milliseconds. Besides, the same process ensures the peer still alive by sending heartbeats to the Failure Detector, every 100 milliseconds as well. When the Failure Handler asks the Failure Detector about a random peer and it returns *false*, the Failure Handler triggers the **suspected** function defined in the protocol. This function makes the peer create a message with the *Phase* parameter equal to 2 and broadcasts it to information the other peers that a possible failure occurred.

As mentioned above, the Failure Detector controls the number of faults between the peers. When a peer tries to simulate its failure, it asks the Failure Detector if it can fail or not. This behavior is the responsibility of the Failure Trigger, which is described below. Depending on the number of current faults, the Failure Detector allows the peer fails if the number of faults is below to a given limit. This limit is a parameter that is given to the simulator, called **Percentage of faults**. It represents the percentage of peers that can fail in the protocol. This parameter is useful because it can guarantee and control the presence of faults. Per example, if the parameter has the value of 20, it means that at most 20% of the peers will fail during the consensus.

Failure Trigger

All peers running the protocol have a probability of failing, which is due to many reasons, for example, errors or failures. However, in a simulation environment, no process fails for a logical reason. Therefore, it is necessary to add the Failure Trigger component to trigger the peer failure according to a certain probability. This probability, called Probability to fail, is a parameter given to the simulator. When it is 0, the peer never fails. Otherwise, the peer may fail according to the value.

In more detail, this component is a process running in the peer that checks if it fails or not according a given probability. After that, if the peer checks that it needs to fail, it will check through the Failure Detector, if it can fail or not. Only after these two verifications,

the peer can fail and stop running. Before to stop running, it puts its status as failed in the Failure Detector to report the fail.

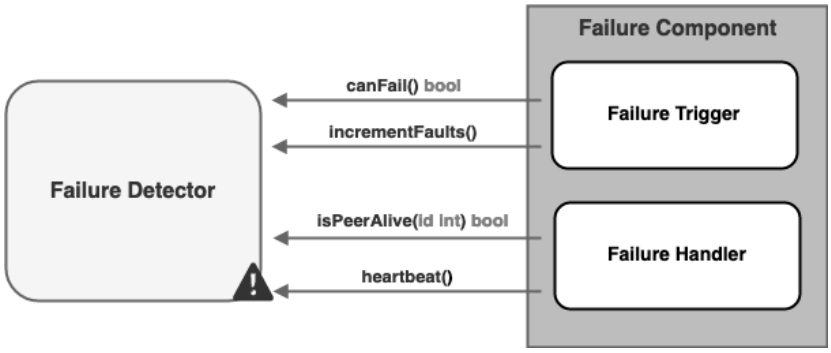


Figure 10: Overview of the interaction between the components of failures.

3.1.2 *Mutations*

MCP is a flexible protocol due to the capacity to exchange between several mutations. These mutations, based on a straightforward interface, filter the messages that are sent and are defined only by two functions:

- Δ_0: At the moment to send a new message to all peers, the function Δ_0 is invoked to validate the message transmission. In other words, the function checks whether to delay the message or not for each peer. If the message is set to be delayed, the message is not sent immediately. Otherwise, the peer sends the message immediately to the destination peer. In either case, the message is saved in the Stubborn Channels buffers until a retransmission mechanism tries to send the message again;
- Δ: All saved messages in the Stubborn Channels buffers represent the last messages sent. This message, after a certain amount of time, is retransmitted by the retransmission mechanism to ensure that the message was received. This mechanism tries to prevent problems caused by lost messages on the network and increases the probability of a message to be received. In this case, the function Δ is used in these retransmissions to validate the messages.

The conditions filter the messages that are to send taking into account the destination peer. For example, the centralized mutation has a condition that checks if the destination peer is the coordinator or not. Each mutation has particular conditions, but there is a set of conditions that almost all mutations use:

- **fresh**: the function that checks if the new message is fresh, *i.e.*, if the message has a different *Round* or *Phase*;
- **majority**: the function that verifies whether the number of gathered votes is higher than $N/2$;
- **needAck**: for each received message, the peer replies with a *Acknowledgement (ACK)*. Therefore, when a peer receives a message, it checks if it needs to reply with an ACK;

The conditions *fresh* and *majority* prevent the transmission of useless information and thus to reduce the number of exchanged messages. The *needAck* condition is an optimization, which prevents the retransmission of a message that already was received, reducing the number of exchanged messages too and no influencing the correctness of the protocol.

Taking into account the two functions that define a mutation and the previous conditions, the four mentioned mutations are defined as follow:

Early The *early* implementation is the simplest. Every peer broadcasts its vote to all others. At some point, one of them gathers at least $N/2+1$ votes, which makes it decided and broadcast all the votes gathered to all others. These peers will receive this message and decide too. Therefore, this mutation allows decisions to occur in two communication steps, reaching the consensus after that.

```
1  func (early *Early) Delta0(id int, pack *stb.Package) bool {
2     channel := early.peer.GetChannel()
3     isFresh := fresh(channel.GetPackage(id), pack)
4     isMajority := majority(pack, early.peer.GetNumberParticipants())
5     needAck := early.peer.NeedAck(id)
6
7     return (isFresh || isMajority) || needAck
8  }
9
10 func (early *Early) Delta(id int) bool {
11    return true
12 }
```

Listing 3.1: The *early* implementation.

Centralized The coordinator has the responsibility to start the consensus, but also to control the consensus, exchanging all messages. Only the coordinator broadcasts messages to all peers, while all others sent messages only to the coordinator. When the coordinator has at least $N/2+1$ votes, it decides and broadcasts all the votes gathered to all others.

```
func (centralized *Centralized) Delta0(id int, pack *stb.Package) bool {
    coordID := centralized.peer.GetCoordID()
    peerID := centralized.peer.GetPeerID()
    channel := centralized.peer.GetChannel()
    isCoord := id == coordID || peerID == coordID
    isFresh := fresh(channel.GetPackage(id), pack)
    isMajority := majority(pack, centralized.peer.GetNumberParticipants())
    needAck := centralized.peer.NeedAck(id)

    return (isCoord && (isFresh || isMajority)) || needAck
}

func (centralized *Centralized) Delta(id int) bool {
    return true
}
```

Listing 3.2: The *centralized* implementation.

Gossip The *gossip* implementation aims a gossip-style message exchange pattern for consensus. This pattern consists of a peer that spreads a message to a random subset of its neighbors. Each neighbor does the same thing, but for a different subset of peers.

The gossip mutation, unlike the others, requires additional parameters that help to get the random subset of peers and thus send a message to them:

- **permuted list of peers**: Every peer generates a random permutation of the sequence of peers identities. This list is used to select F peers to gossip a message;
- **fanout** (F): number of peers to gossip a message;
- **turn**: the number of the turn. The value varies between $[0, F]$;
- **number of peers to skip**: this number is the number of peers to skip on the permuted list. For each broadcast, the value increase F, which allows gossiping the message to the next F peers.

Every peer only sent messages to a subset of peers, selected from the permuted list of peers. When a peer tries to send a message, it only selects F peers from the list, through the following formula:

$$\forall i \in [0, F] : permutedList[(i + peersToSkip) \mod numberOfPeers] \tag{3}$$

This formula allows using the permuted list of peers as a circular list.

Finally, the rest of the protocol is like the other mutations, *i.e.*, the peers exchange messages until all peers gather at least $N/2+1$ votes to make a decision. However, this mutation does not use any previous conditions (fresh and majority) and it still has an extraordinary performance compared to the others, in particular in terms of scalability and time to reach the consensus.

```
1  func (gossip *Gossip) Delta0(id int, message *stb.Package) bool {
2     needAck := gossip.peer.NeedAck(id)
3
4     return gossip.Delta(id) || needAck
5  }
6
7  func (gossip *Gossip) Delta(id int) bool {
8     numberParticipants := len(gossip.permut)
9     gossip.turn++
10
11    if gossip.turn == numberParticipants {
12       gossip.processesToSkip += gossip.fanout
13       gossip.turn = 0
14    }
15
16    processIndex := 0
17    for processIndex < gossip.fanout {
18       if gossip.permut[(processIndex + gossip.processesToSkip) %
             numberParticipants] == id {
19          return true
20       }
21       processIndex++
22    }
23
24    return false
25 }
```

Listing 3.3: The *gossip* implementation.

Ring Ring-style algorithms are usually in message exchange protocol, and they are easily implemented. In the *ring* implementation, only allows a message to be sent from the $peer_i$ to the $peer_j$ if the $peer_j$ is the successor of the $peer_i$. In other words, if the following condition is *true*:

$$peer_j = ((peer_i + 1) \mod N) + 1. \tag{4}$$

In other words, each peer only sends messages to its successor. The coordinator starts the consensus sending a message with its vote to its successor. The successor does the same thing, adding its vote to the message and sent it to the next successor. Therefore, the number of votes that the message exchanges increase until a peer has $N/2+1$ votes. At this moment, this peer sends all the votes gathered to the next successor and decide. After that, all peer successors start deciding as well, and in the end, the protocol reaches the consensus.

The ring mutation has an excellent performance, but it becomes a useless option for faulty environments. In the presence of faulty peers, it is very likely to stop responding until retransmissions occur. Therefore, a new approach of ring mutation was made to improve its resilience to faults — the next topic discuss this approach.

```
1  func (ring *Ring) Delta0(id int, pack *stb.Package) bool {
2     channel := ring.peer.GetChannel()
3     isFresh := fresh(channel.GetPackage(id), pack)
4     isMajority := majority(pack, ring.peer.GetNumberParticipants())
5     isIDEqual := id == ((ring.peer.GetPeerID() %
          ring.peer.GetNumberParticipants()) + 1)
6     needAck := ring.peer.NeedAck(id)
7
8     return (isIDEqual && (isFresh || isMajority)) || needAck
9  }
10
11 func (ring *OldRing) Delta(id int) bool {
12    return id == ((ring.peer.GetPeerID() %
13       ring.peer.GetNumberParticipants()) + 1)
14 }
```

Listing 3.4: The *ring* implementation.

In sum, each mutation has a different behavior, but a common goal: make peers reach consensus. Nevertheless, it is important to emphasize that there are mutations that fit better in certain environments than the others, and vice versa. To take an overview of each mutation behavior, we built a *bubble plot*, in which the size of the bubble represents the number of sent messages from the $peer_i$ to the $peer_j$. The larger the bubble, the bigger the number of messages sent. This variation of the bubble size, show us all the interactions between all peers and allow us to see the message exchange pattern that each mutation creates as well. Figure 11 shows the message exchange patterns for each mutation.

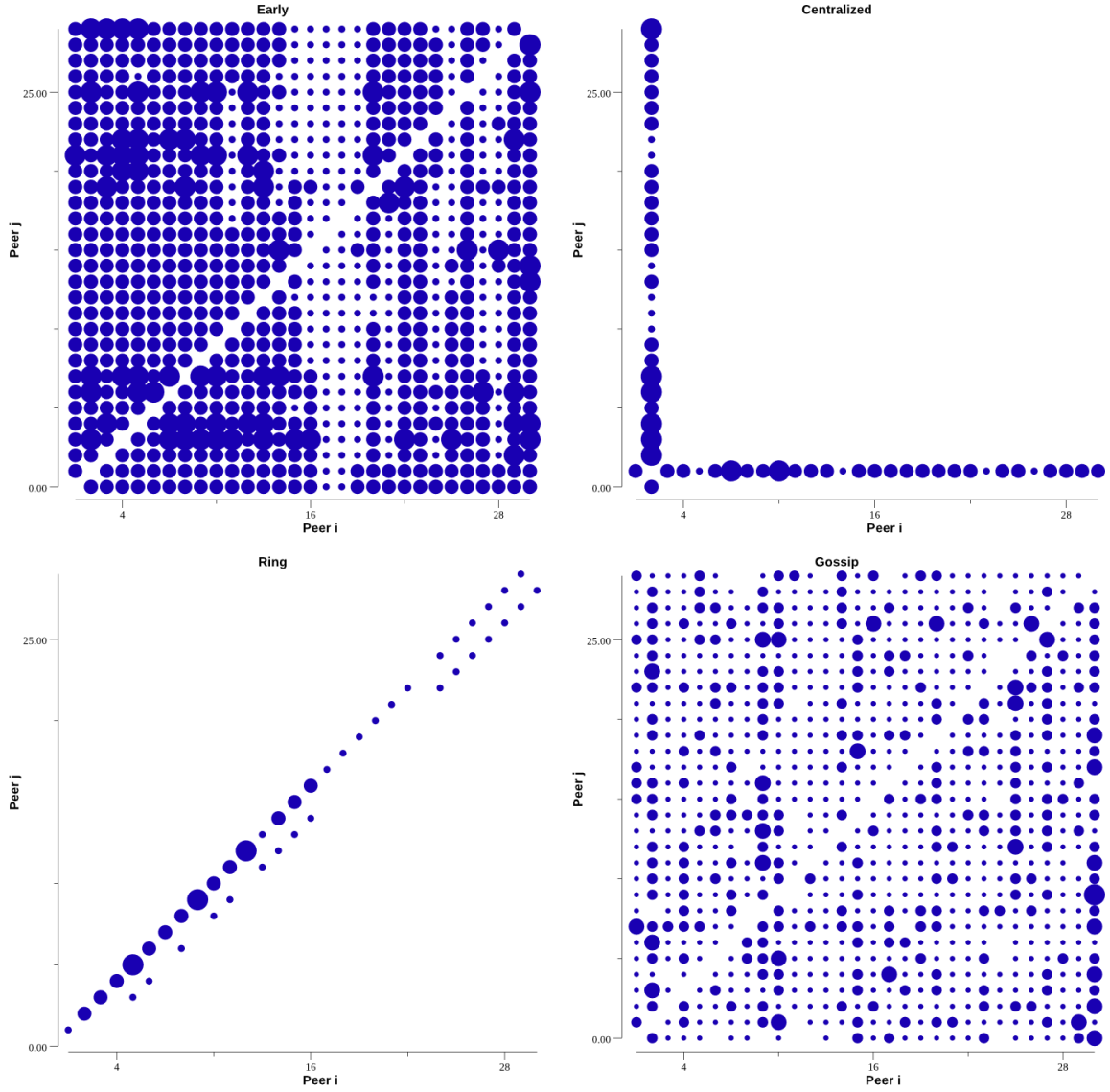


Figure 11: Overview of messages exchange in each mutation.

The previous figure presents the relevant details. The *early* mutation is the mutation that exchanges more messages, creating a denser plot. On the other hand, the *ring* mutation is the mutation that exchanges fewer messages due it only sends messages to its successor. In the *centralized* mutation we can see that the coordinator handles the majority of the messages. Finally, the *gossip* mutation, unlike the *early* mutation, there is a better distribution of the messages, which reduce the number of exchanged messages.

Improve ring mutation

The original ring mutation has few weaknesses in the presence of faulty peers, leading to not achieving the main goal: reach the consensus. It is a problem considering that a

mutation needs to be minimally resilient to environments with faulty peers. If it is not, it becomes useless in such environments. When a mutation stops responding, the MCP converges to the *early* mutation to ensure the protocol does not stop and thus reach the consensus. Therefore, a new Ring mutation was build to fix the weaknesses that the original version has.

Problem The problem lies in 1 or more peers fails before to spread a message to its successor. The ring mutation has only a single alternative to spread the message. It starts at the coordinator, who sends a message only to his successor. The following peers do the same thing, thus forming a ring. If a successor fails before to send a message to the next successor, the rest of successors will never receive a message. Although there is a retransmission mechanism, it does solve the problem. If the failed successor did not send an ACK message, the peer keeps retransmitting the message until it reaches the limit of retransmissions and thus converge to the *early* mutation. The Figure 12 illustrate the problem.

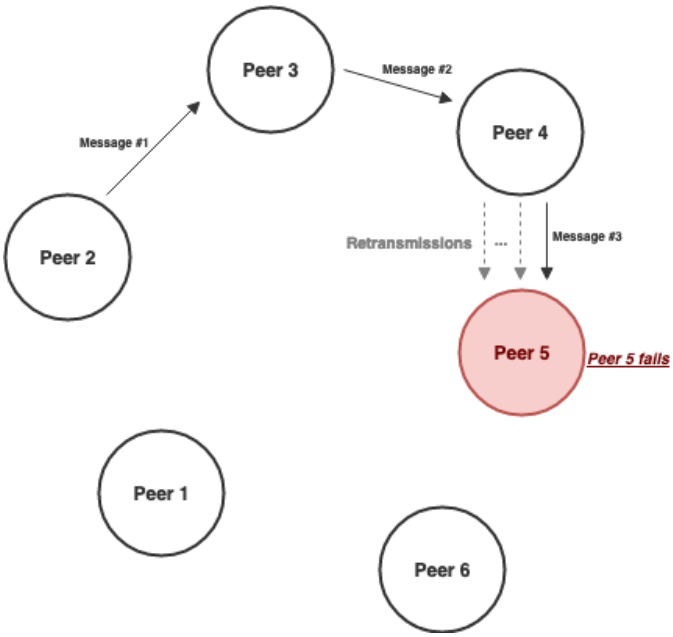


Figure 12: Illustration of the problem of ring mutation.

Solution A new approach of the *ring* mutation was created to fix the problem of faulty peers. This new version instead of just sending a message only to the successor, it also sends a message to its predecessor in order to increase the number of alternatives. Only with that, can we mitigate the problem by increasing the number of messages exchanged and thus increase the probability of a peer receive a message. However, this is not enough to solve the problem. When it has at least two failed peers, one being the successor and the other the predecessor, the rest of the peers will not receive any message until the protocol converges to the *early* mutation. Therefore, we introduce a new set of variables to the mutation:

- **stepsForward** and **stepsBack**: the number of steps to reach the target peer for both sides (forward and back);
- **suspectFailureForward** and **suspectFailureBack**: it is true if the peer suspects the peer failed (forward and back). The suspicion consists in checking if the last message arrived and if the *Phase* did not change.

The variables *stepsForward* and *stepsBack* start with the value 1 and only change when the peer suspects a failure. In other words, when a peer tries to retransmit its last message and suspects that its successor or predecessor failed, it increments the corresponding steps**X** variable and also turns to *true* the corresponding suspectFailure**X** variable. Next time, the peer will retransmit the message or send a new message to the next peer (successor or predecessor) and thus avoiding the failed peer. After that, the peer keeps skipping the failed peer and sends messages to the next one. The suspectFailure**X** variables ensure that the steps**X** value only increase once in the broadcast.

These new variables solve the problem of faulty peers, turning the *ring* mutation a good option, even for environments with faulty peers. The Figure 13 shows the messages exchanged in the new version of the *ring* mutation.

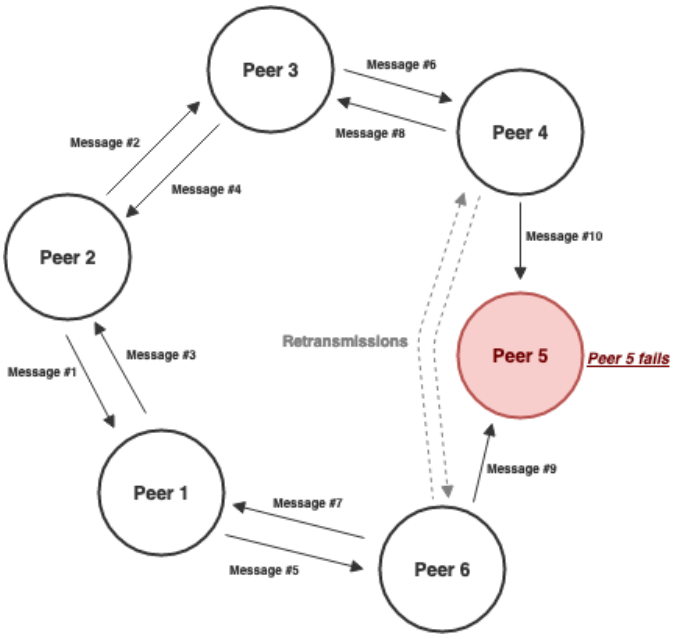


Figure 13: Illustration of the new ring mutation.

In short, the new version of the *ring* mutation improves the performance in environments with faulty peers. However, in environments with a low probability to a peer fails, the new version can be the worst option, because of the significant number of messages exchanged compared to the old version. Therefore, we give up some mutation performance in order to make it more resilient to faulty peers.

Below, we have the implementation of the new *ring* mutation:

```
func (ring *Ring) Delta0(id int, pack *stb.Package) bool {
   channel := ring.peer.GetChannel()
   isFresh := fresh(channel.GetPackage(id), pack)
   isMajority := majority(pack, ring.peer.GetNumberParticipants())
   next := ring.getNext()
   previous := ring.getPrevious()
   needAck := ring.peer.NeedAck(id)

   return ((id == next || id == previous) && (isFresh || isMajority)) || needAck
}

func (ring *Ring) Delta(id int) bool {
   next := ring.getNext()
   previous := ring.getPrevious()
   lastPackage := ring.peer.GetChannel().LastPackageBuffered(id)
   packageArrived := lastPackage != nil && lastPackage.Arrived

   if !packageArrived || ring.peer.GetConsensusInfo().Round == 2 {
      if id == next && !ring.suspectFailureForward {
         ring.stepsForward++
         ring.suspectFailureForward = true
      } else if id == previous && !ring.suspectFailureToBack {
         ring.stepsToBack++
         ring.suspectFailureToBack = true
      }
   }

   if ring.isLastNode(id) {
      ring.suspectFailureForward = false
      ring.suspectFailureToBack = false
   }

   return id == next || id == previous
}
```

Listing 3.5: The new *ring* implementation.

Machine learning integration

An important step was to integrate the ML model with the Simulator. All models are implemented in *Python*[2] with the *TensorFlow*[3] library, which builds, train, and tests the

2 https://www.python.org/
3 https://www.tensorflow.org/

models. This library offers several integrations with different languages, and with *Go* too. So the strategy is to build the model in *Python* and when it is ready, we export it. Finally, we import it into the Simulator, and it is prepared to be used. This process will be discussed in more detail in the next chapter.

3.1.3 *Message*

In consensus protocol, the *message* is a critical component, as far as the information that it contains is the only thing that the receiver knows about the sender. Therefore, it is essential the *message* being well structured, in order to report the current state of the peer to the others, and thus, the receiver can make the right decisions.

The Simulator contains just one type of message in order to keep it simple. The message structure is illustrated in Figure 14.

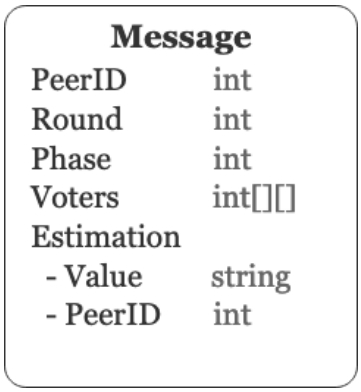


Figure 14: Message structure.

The message contains several variables, which ensure the identity of the sender and inform about the current that of it. These variables are described below:

- **PeerID**: It is the identifier of the sending peer;
- **Round**: It represents the round which the sender is;
- **Phase**: It controls the presence of faults. When a peer suspect that a peer fails, it changes the *phase* to 2 and send it to the other peers;
- **Voters**: It is the map with all votes gathered by the peer. It contains all votes and who voted;
- **Estimation**: It contains two parameters: *Value* represents the estimation of the peer, and *PeerID* represents the peer identifier.

Whenever $peer_j$ receives a message from the $peer_i$, in the next time that the $peer_j$ sends a message, it also will send a message to the $peer_i$, to confirm the message received, like a ACK. This only happens because of the condition *needACK* defined in the mutation. This condition, after receiving the message, is equal to *true* to the $peer_j$ and thus allows the $peer_j$ send a message to the $peer_i$. With that, the retransmission mechanism in the *Stubborn Channels* will not retransmit a message to the $peer_j$, because the message will act as an ACK and ensures to the $peer_i$ that the $peer_j$ already receives the message from it.

3.1.4 *Stubborn Channel*

The Stubborn Channel is responsible for transmitting all the message between the peers and ensuring that all messages are delivered. It use the **Go Channel** instead of the normal network, to be possible to manipulate the network in the Simulator, *i.e.*, to set up the network with given parameters:

- **percentage of message loss**: the percentage of messages that fail on the network. Each time a message is sent, it is checked, based in this percentage, whether or not it fails on the network.
- **bandwidth**: the measure that defines the ability to process message. In other words, the bandwidth represents the number of messages that a peer can process in a second. The bandwidth implements the *leaky bucket* algorithm [26].
- **latency**: the latency is the additional delay caused by the network in milliseconds. This delay is caused by put to sleep the process that will send the message.

The Stubborn Channel integrates all the messages exchanged by the peers. When a peer tries to send a message to all others, it uses the function *SendAll*. On the side of the Stubborn Channel, it creates a process that handles these transmissions, sending the message to each peer. In every transmission, the process uses the Mutation to validate if it is possible or not to send the message to the target peer. For that, it uses the function Δ_0 defined by the Mutation. For the Stubborn Channel, the Mutation acts like a filter in the process of sending a message.

Besides, this process also saves the message in a *Buffer* for each peer so that the retransmission mechanism can retransmit the message later. When this mechanism tries to retransmit a message, it uses the function Δ defined by the Mutation to validate the message as it does with the Δ_0 function.

In short, the Stubborn Channel interacts with all components as we can see in the Figure 15.

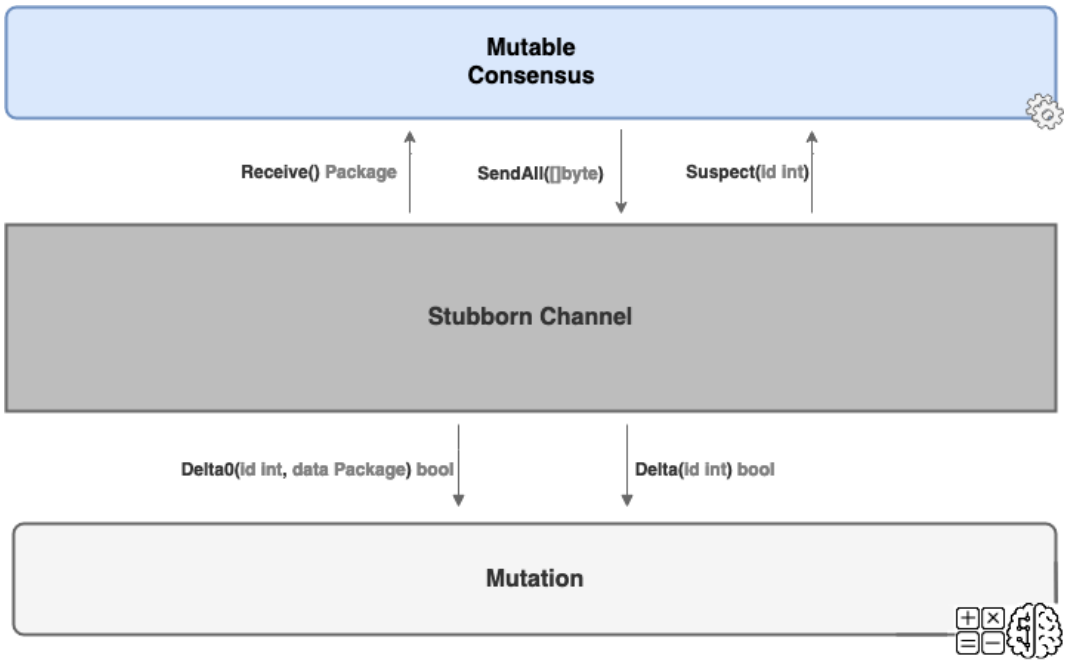


Figure 15: The interaction between the Stubborn Channel and the other components.

The Peers that are waiting for a message, receive it through the function *Receive* in a **Package**, which is defined by use. The Package is the only object transmitted by the Stubborn Channel and its structure is present in the Figure 16.

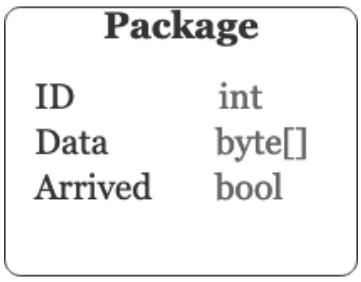


Figure 16: The Package structure.

The package contains several parameters to deal with the *ACK* messages and standard messages. Some of variables are:

- **ID**: It is the identifier of the sending peer;
- **Data**: It is the array of bytes of the message that the peer send. This array of bytes is the message which the protocol exchange between the peers;
- **Arrived**: It indicates if the current message arrived or not. When a peer receive an ACK message, it turns this parameter in the last message sent to this sender to *true*;

When a peer tries to send a message through the channel, it needs to convert the message to an array of bytes (*[]byte*). The channel receives this array and encapsulates it into a **Package**. In this way, the Stubborn Channel does not know what kind of data it transmits and thus keep the MCP and Stubborn Channel independent components.

The Stubborn Channel has different components to process the messages. As the channel is a simulation of real communication over the network, it is built taking into account all the network issues. There is a process, called *Writer*, that handles the messages that are to send and another process, called *Reader*, that reads the incoming messages. These processes communicate over a *Go Channel*, which acts like a *First-In, First-Out (FIFO)* queue where the message is put on by the *Writer* and removed the *Reader*. There is a channel for each peer, and all of them are instanced in the Stubborn Channel. Indeed, the Stubborn Channel is a shared structure between all peers, which contains all channels. This way, all peers have access to all peer channels.

In short, the *Reader* of the $peer_i$ keep reading messages from the $channel_i$, and when the $peer_j$ tries to send a message to the $peer_i$, it sends a message to the $channel_i$. In the end, the $peer_j$ saves the package sent in a *Buffer* in order to the Retransmission mechanism can retransmit it again. Every *DeltaDefault* seconds, this mechanism tries to retransmit all the packages in the Buffer. Before it sends a package, it checks the parameter *Arrived*, to check if the message already reaches the peer, and the function Δ, to validate if the target peer is valid. Therefore, the Retransmission mechanism only sends the message again if the message still not reach the peer, and the target peer is valid.

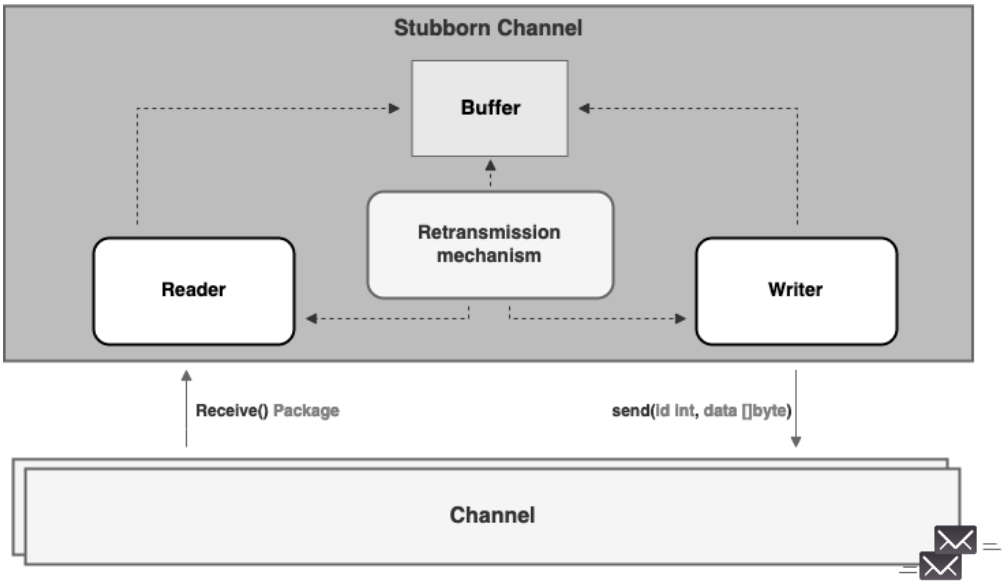


Figure 17: The Stubborn Channel structure in detail.

3.2 COLLECT METRICS

The Simulator integrates a few collectors to extract the data to feed the ML model. The MCP has several parameters that can be used to feed the model. In this phase, it is important to define which parameters to track, and when and how to collect them. The better the data we feed the model, the better the results we got. There is two types of parameters:

- **constant**: parameter which value never changes. For example, the environment parameters that the protocol is configured - *bandwidth*, *number of nodes*, etc.
- **variable**: parameter which value can change during the protocol. For example, the protocol variables - *votes*, *delays*, etc.

These parameters provide different types of information, which influence what and how the ML model learns. Therefore, we approach the problem in two different contexts: **Environment** and **Protocol**. These contexts will be described in the following subsections.

3.2.1 *Environment metrics*

The environment metrics represent all the parameters used to configure the protocol environment. These parameters are defined by the user at the beginning before the protocol starts. Table 1 shows all metrics used and an example of the respective values that each parameter can assume. This example is a row of an output used to feed one of the ML models, but it only will discuss in the next chapter.

nodes	defaultDelta	maxTries	percentageMiss	percentageFaults	probabilityToFail	latency	bandwidth	bandwidthExceeded	mutation	resultTime
200	1	3	0	25	25	150	50	TRUE	early	54437.25778

Table 1: All the environment parameters and an example of a row given by a simulation.

Almost all of these parameters are constants parameter, which are the values that configure the environment that the Simulator runs. However, one of these values is variable and can change in each run. In this example, we use the *resultTime*, which represents the time in milliseconds that the consensus takes to be reached. This metric is easily extracted from the Simulation, by register the time before the protocol starts and register the time when the *Supervisor* receives all the decisions from the *Peers*. As for the other metrics, as their values are constant we can get it at any moment. Therefore, the environment metrics can be easily gathered, and mixed with other metrics to create a dataset to feed a ML model.

3.2.2 *Protocol metrics*

Unlike the environment metrics, the protocol metrics are several variable parameters. These parameters represent the protocol variables and they change in each message exchanged. With that, we have another approach where is necessary save all these parameters and all the changes that they have for each peer, and thus to use them as a dataset to feed a ML model. For that, we define each row as a snapshot of the current state of a peer, as the peer's point of view at that moment. Table 2 represent all these parameters and an example of the values that they assume.

PeerID	CoordID	Round	Phase	EstimatePeerID	EstimateValue	Decision	Peer1Vote	...	PeerNVote	DelayToPeer1	...	DelayToPeerN
2	2	1	1	2	"accept"	null	1		1	0.00		0.00

Table 2: All the protocol parameters and an example of a row given by a simulation.

The parameter *PeerID* is the only one that never changes for a peer. However, as we have several peers, from a general point of view, this parameter change when the information comes from another peer, which is expected.

They have a few parameters that change frequently: *votes* and *delays*. When a peer receives a new message, it updates its votes with the votes from the message. The array of votes is a bitmap that contains all the peers that already vote, and not what they vote. Therefore, for a peer that gather votes from the others, a peer that already vote has the value 1, and a peer that does not vote yet has the value 0.

The delay values represent the current delay of the $peer_i$ to send a message to the $peer_j$. They are the most complicated metrics to collect in the protocol because they are always changing during the consensus and become hard to collect realistic values. However, with some mechanisms, it is possible to collect these values. Each peer has a list of delays, where each position represents the delay to send a message to a certain peer. The list starts with negative values equals to -0.000001. The negative values represent when a message is always delayed, whereas the positive or null values represent when the message is sent. When a peer sends a message to another peer, a process set the delay in the list for this peer to zero. From now, if the delay mechanism sends a new message, it updates the value with the delay between the last message sent and this retransmitted message. Every time that a peer stop sends a message to the other peer, the peer sets the delay between them to -0.000001 again.

Finally, the other parameters vary with little frequency and are defined at the beginning of the run, so it is not necessary to process them.

3.3 EXECUTION

The protocol is implemented in *Go*, so the Simulator is compiled into a single executable file called **simulator**. All the collected metrics from the protocol that the simulator runs are written to a *.csv* file. Depending on what kind of dataset we want to build, the protocol can generate two different datasets:

- **Single row**: the simulator can generate a single row. This is used for the environments metrics, to compare different environments. In this case, it is necessary run the protocol more than 1 time.
- **Entire dataset:** in another context, the simulator can generate an entire dataset to feed a DL model. This case is related to the protocol metrics, which generate several entries in the dataset, which each entry represent a snapshot of current state of a peer.

The simulator has several parameters, which configure the environment that the protocol runs. For that, all these parameters are defined as attribute to the *simulator* command, to be simple to define different environments and use different mutations. The simulator is using as follows:

```
simulator <mutation> <nodes> <default_delta> <max_tries> <percentage_msg_miss>
    <latency> <bandwidth> <percentage_faults> <probability_to_fail> <metrics>
```

- **mutation**: the mutation that was running;
- **nodes**: number of nodes;
- **default_delta**: default delta for retransmissions (seconds);
- **max_tries**: maximum tries until converging to the early mutation;
- **percentage_msg_miss**: percentage of missed messages ($0 - 100\%$);
- **latency**: network latency (ms);
- **bandwidth**: processing power of a peer (msg/s);
- **percentage_faults**: percentage of node failures ($0 - 100\%$);
- **probability_to_fail**: probability of a node fail ($0 - 100\%$);
- **metrics**: output metrics from protocol (true/false).

Take into account the example with the environment metrics and the *resultTime*, we get the following:

```
> simulator gossip 100 2 5 10 100 300 10 20 true
  100,2.0,5,10.0,10.0,20.0,100.0,300,true,gossip,158.310574
```

Another example is to run the simulator to generate an entire dataset with all states of each peer. For that, we get the following:

```
> simulator centralized 40 1 5 0 100 100 0 0 true
  2,2,,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,...,-0.000001,-0.000001,0.000000
  40,2,,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,...,-0.000001,-0.000001,-0.000001
  2,2,,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,...,0.000000,0.000000,0.000000
  1,2,,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,...,-0.000001,-0.000001,-0.000001
  (...)
```

4

APPLYING MACHINE LEARNING

ML allows a machine to learn from data, identifying patterns in it, and thus making decisions with almost no human intervention. Therefore, it is important to gather useful information/data from the MCP, because the better the input data is, the more accurate the ML model becomes. So, the task of gathering data to feed the ML model is equally important to the task of building the model, that applies a ML algorithm.

As mentioned above, it was possible to gather two types of metrics from the protocol: *Environment* and *Protocol*. Each of these metrics has a different approach to the problem and therefore the datasets created from these metrics have different features to learn. Therefore, different ML models were built, which learn different things from each type of metrics:

- **Learn from different environments**: the goal is to learn features in a variation of the environment and mutation, and thus can say what mutation should take less time to reach the consensus in a specific environment, for example.
- **Learn mutations**: the goal is to learn the features in a mutation, *i.e.*, the way it exchanges the messages, for example. Also, the goal extends in to learn more than one mutation and to exchange between them when an environment parameter change.

Besides, in the ML field it is common to use *Jupyter*[1] notebooks to build the models. Therefore, we used the *Colaboratory*[2][27] platform to build machine learning models. Colaboratory is a research project created to help disseminate ML education and research, owned by Google. It integrates Jupyter notebook environment and runs entirely in the cloud, which requires no setup to use. With Colaboratory we can write and execute code, save and share our analyses, and access powerful computing resources.

4.1 LEARN FROM DIFFERENT ENVIRONMENTS

Within the scope of the MCP, several parameters influence its performance, particularly the environmental parameters such as latency, bandwidth, percentage of failures, etc. Con-

1 https://jupyter.org/
2 https://colab.research.google.com/

sidering all these parameters as the input data for the model, making several different combinations of them, we will have a dataset which represents several examples of environments. The more examples of different combinations, the more the ML model learns. The idea is to apply ML techniques to this data, which is collected from running MCP in a variety of environments, with different parameter configurations.

nodes	defaultDelta	maxTries	percentageMiss	percentageFaults	probabilityToFail	latency	bandwidth	bandwidthExceeded	mutation	resultTime
200	1	3	0	25	25	150	50	TRUE	early	54437.25778
100	5	8	0	25	25	150	200	FALSE	ring	5995.506902
50	5	8	0	25	25	50	50	FALSE	ring	10004.69666
200	5	3	0	0	0	50	50	FALSE	ring	3355.443603
100	1	3	0	25	0	150	50	TRUE	centralized	7502.218138
50	1	8	12	25	0	150	50	TRUE	early	889.095608

Table 3: Sample of a dataset with environment parameters.

The ML model from this data can give several results, depending on the goal we want to achieve. Taking this into consideration, we can build two different models:

- **Regression**: The purpose of the model is to predict the default delay for a given environment through a regression algorithm.
- **Classifier**: The classification model has the goal of classifying the best mutation for a given environment.

4.1.1 *Machine learning model*

It is necessary to perform some steps to build an accurate ML model. The steps are: gathering data, preparing data, choosing a model, training, evaluation, parameter tuning and prediction.

Gathering data

The quality and quantity of data directly determines how good our predictive model can be. In the Classifier model, the data we collect will be the environment parameters of a peer and the time it takes to reach consensus. These parameters are collected during the execution of the simulation of the MCP, described in the previous chapter. Here is an example of execution and the respective output, which represents a row in the final dataset:

```
> simulator gossip 100 2 5 10 100 300 10 20 true
  100,2.0,5,10.0,10.0,20.0,100.0,300,true,gossip,158.310574
```

Preparing data

When we have all the data collected, the next step is to load the data into a suitable place and prepare it to be used in the ML model. Therefore, we put all the data together, and then randomize the ordering, because the order of the data can affect what the model learns.

Sometimes the data collected needs to have some of its parameters adjusted in order to be used in the ML model. In this case, only the categorical parameters need to be adjusted. Since we have categorical parameters, we need to encode them to avoid a misinterpretation by the model. For example, the parameter **mutation** can have four different values: *centralized*, *early*, *ring* or *gossip*. No entries can have string values, only numbers. Therefore, we need to encode them into different numerical values, but if we encode, for example, *centralized* to 1 and *early* to 2, the model interprets that the early mutation is higher than the centralized, which is not true. We apply the **One hot encoding** strategy to all categorical parameters to avoid this problem. This strategy creates a new column for each value that the parameter can have and, for each entry, it gives the value 1 to the column that corresponds to the value it had and 0 to the remaining columns.

Finally, when the data is ready to be used we split it into two different parts, one for the input parameters, and one for the target parameter. After that, we split the two parts into two other separate sets: train set and test set. The first set is used in training the model and has the majority of the dataset. The second set is used for evaluating the trained model's performance. We do not want to use the same data that the model was trained on for evaluation since it could then just memorize the information.

Choosing a model

As mentioned in chapter 2, choosing a model depends on what kind of input data we have and what kind of output we want. In this case, we have a non-linear problem, in which we have a single input data and two different outputs. This leads to building two different models. One of them is a **regression** model which deals with a continuous target value, and the other is a **classification** model, which deals with a target value that is discrete and limited to a set of values.The **Random Forest** algorithm supports both problems: classification and regression. Therefore, we use the *Python* library **Sklearn**[3][28] to get both Random Forest models: *RandomForestRegressor* and *RandomForestClassifier*.

Training

In this step, we use the input data to feed both ML models, which makes it learns and improves the ability to predict results. This process is iterative until it reaches an accurate

3 https://scikit-learn.org/

model that predicts the results correctly most of the time. The training process for the Random Forest algorithms is described in detail in chapter 2.

Evaluation

Once training is complete, the next step is to verify if the model is good, using Evaluation. This is where the test dataset that we split from the original one comes into play. Evaluation tests the model against data that has never been used for training and thus verifies how the model behaves against data that it has not yet seen. This is meant to be representative of how the model might perform in the real world.

Hyperparameter tuning

With the evaluation complete, the next step is to improve the model by tuning its parameters. The model has several parameters that can be changed. Therefore, by testing several different combinations of them, we will find the best combination that gives us the most accurate model. It is important to verify that the model is well adjusted. If the model is underfitting, the predictions will be very poor, which gives the impression that the model has not learned anything from the data. However, the model also can not be overfitted. Overfitting occurs when we have a very flexible model which essentially memorizes the training data by fitting it closely. The predictions for the training dataset may be good, but when we use a new dataset, that it has never been seen before, the predictions will not be as expected. In a Random Forest model, we can control this problem based on the number of leaf nodes in the tree.

Prediction

Prediction, or inference, is the step where we get to answer our questions. In other words, it is the step where we use the model to predict, for example, the best mutation for a given environment in a real-world case.

4.1.2 *Regression model*

Before we explain the regression model, it is essential to understand the importance of the default delay parameter. The default delay is the parameter that has more impact in all mutations, particularly in the performance, because it controls the frequency of retransmissions. On the one hand, when we have a low value for the default delay, the retransmissions are more frequent, which can improve the performance or overload a peer. On the other hand, when the default delay takes a high value, the retransmissions are less frequent, which can avoid overloading a peer but can make the performance worse. So, it is really

important to measure this parameter and make a trade-off between the performance and the load exerted on a node.

A ML model was built to predict the value for a given environment.

Dataset

The dataset that we use to feed the regression model is obtained through the encoding process for categorical parameters and split the target parameter, *i.e.*, default delta, from the input parameters. From the previous example, we have the following parameters:

nodes	defaultDelta	maxTries	percentageMiss	percentageFaults	probabilityToFail	latency	bandwidth	bandwidthExceeded	mutation	resultTime
200	1	3	0	25	25	150	50	TRUE	early	54437.25778
100	5	8	0	25	25	150	200	FALSE	ring	5995.506902
50	1	8	0	0	0	150	200	FALSE	gossip	143.339268
200	5	3	0	0	0	50	50	FALSE	ring	3355.443603
100	1	3	0	25	0	150	50	TRUE	centralized	7502.218138
50	1	8	12	25	0	150	50	TRUE	early	889.095608

Table 4: Sample of the input dataset of *regression* model.

Applying the *One hot encoding* and split the *defaultDelta* parameter from the input parameters, we get:

Input Parameters

nodes	maxTries	percentageMiss	percentageFaults	probabilityToFail	latency	bandwidth	bandwidthExceeded_true	bandwidthExceeded_false
200	3	0	25	25	150	50	1	0
100	8	0	25	25	150	200	0	1
50	8	0	0	0	150	200	0	1
200	3	0	0	0	50	50	0	1
100	3	0	25	0	150	50	1	0
50	8	12	25	0	150	50	1	0

mutation_centralized	mutation_early	mutation_ring	mutation_gossip	resultTime
0	1	0	0	54437.25778
0	0	1	0	5995.506902
0	0	0	1	143.339268
0	0	1	0	3355.443603
1	0	0	0	7502.218138
0	1	0	0	889.095608

Target Parameter

defaultDelta
1
5
1
5
1
1

Table 5: Sample of the input dataset of *regression* processed and split.

Training

The model selected for training is *RandomForestRegression*. It receives the previous dataset as input and makes predictions about **defaultDelta** for a particular environment. The result is a continuous value that, for the specific model, seems to fit into the given environment. However, since the model predicts a continuous value, this value is not entirely equal to the actual value because the model uses regression for prediction. Because of this, we use the MAE to evaluate the model. The lower the value, the better.

In order to avoid underfitting and overfitting the model, we tested the model with different numbers of leaf nodes in the trees. This value varies between 100 and 1400 leaf nodes. In the end, we select the best result.

Results

After training the model, we obtained these results:

```
(Regression) Nodes: 100 | Error: 0.989415
(Regression) Nodes: 200 | Error: 0.974889
(Regression) Nodes: 300 | Error: 0.967260
(Regression) Nodes: 400 | Error: 0.968848
(Regression) Nodes: 500 | Error: 0.966422
(Regression) Nodes: 600 | Error: 0.964222
(Regression) Nodes: 700 | Error: 0.958534
(Regression) Nodes: 800 | Error: 0.959898
(Regression) Nodes: 900 | Error: 0.958422
(Regression) Nodes: 1000 | Error: 0.956761
(Regression) Nodes: 1100 | Error: 0.957326
(Regression) Nodes: 1200 | Error: 0.960465
(Regression) Nodes: 1300 | Error: 0.960019
(Regression) Nodes: 1400 | Error: 0.962164
(Regression) Min error: 0.956761 | Leaf nodes 1000
```

Figure 18: The results of the *regression* model.

As we can see, the best error we get is **0.956761** for 1000 leaf nodes. In the input dataset, the *defaultDelta* parameter has only three possible values: 1, 2, or 5. Comparing the error with these values, we can verify that the error is a bit high because it is almost equal to 1. For example, for an environment that was configured with the *defaultDelta* equal to 1, the ML model could predict a *defaultDelta* around **1.9**, which is almost double. One of the reasons for this value to be high is because one of the possible values for this parameter is 5, which is a huge number compared to the others and causes the error to increase.

4.1.3 *Classifier model*

Another case is the classification model that can predict the best mutation for a given environment. In other words, it predicts the mutation that takes the least time to reach consensus. This prediction is obtained through the parameter *resultTime*, which is the only result given by the execution of the protocol simulation.

Dataset

Unlike the regression model, this model needs to filter the data first. Therefore, for each combination of parameters that define an environment setting, each mutation has a result, unless it stopped responding. To get the best mutation for each environment, we need to select the mutation with the shortest result time and remove the others. At the end, we only have a dataset with the best cases.

From the input original dataset and applying the filter, we get the following sample:

nodes	defaultDelta	maxTries	percentageMiss	percentageFaults	probabilityToFail	latency	bandwidth	bandwidthExceeded	mutation	resultTime
200	1	3	0	25	25	150	50	TRUE	early	54437.25778
100	5	8	0	25	25	150	200	FALSE	ring	5995.506902
50	1	8	0	0	0	150	200	FALSE	gossip	143.339268
200	5	3	0	0	0	50	50	FALSE	ring	3355.443603
100	1	3	0	25	0	150	50	TRUE	centralized	7502.218138
50	1	8	12	25	0	150	50	TRUE	early	889.095608

Table 6: Sample of the input dataset filtered by the best results.

Only after filtering the input data, we apply the one hot encoding process to categorical parameters and split the target parameter from the input parameters. Applying the *One hot encoding* and split the *mutation* parameter from the input parameters, we get:

Input Parameters

nodes	defaultDelta	maxTries	percentageMiss	percentageFaults	probabilityToFail	latency	bandwidth
100	5	8	0	0	25	400	200
50	1	8	0	0	0	50	50
100	2	8	0	25	0	400	200
200	2	8	0	0	0	50	50
100	5	8	60	40	40	150	500
50	1	3	0	0	0	400	500

bandwidthExceeded_true	bandwidthExceeded_false	resultTime
0	1	195.423392
0	1	393.116436
0	1	208.588526
1	0	918.752037
0	1	3543.636213
0	1	135.568029

Target Parameter

mutation
gossip
ring
gossip
gossip
early
ring

Table 7: Sample of the input dataset processed and split.

Training

Since we have a classification problem, the model *RandomForestClassifier* is selected to be trained. Like the previous model, this model receives the input dataset and makes predictions. However, in this case, it predicts a discrete value. For a particular environment, the model predicts what mutation is the best to run over it, given one of these values: *centralized*, *early*, *gossip* or *ring*.

To measure the accuracy of the model, we only need to calculate the number of correct predictions over the total number of predictions:

$$accuracy = \frac{\#\ correctPredictions}{\#\ totalPredictions} \tag{5}$$

In the previous model, we submitted the model to different values of the number of leaf nodes to find the best result and to avoid underfitting and overfitting it. We repeat this strategy for this model, with the value varying between 100 and 900 nodes of leaves.

Results

The results of the classification model are:

```
(Classifier) Nodes: 100 | Accuracy: 0.988166
(Classifier) Nodes: 200 | Accuracy: 0.988166
(Classifier) Nodes: 300 | Accuracy: 0.988166
(Classifier) Nodes: 400 | Accuracy: 0.991124
(Classifier) Nodes: 500 | Accuracy: 0.988166
(Classifier) Nodes: 600 | Accuracy: 0.988166
(Classifier) Nodes: 700 | Accuracy: 0.991124
(Classifier) Nodes: 800 | Accuracy: 0.991124
(Classifier) Nodes: 900 | Accuracy: 0.988166
(Classifier) Max accuracy: 0.991124 | Leaf nodes 400
```

Figure 19: The results of the classification model.

We have achieved the precision of 0.991124 for 400 leaf nodes as the best result. In other words, we get, approximately, **99% of accuracy**. It seems a good result. However, it is necessary to emphasize that the input dataset with only the best results is almost 90% of the gossip mutation and the centralized mutation never occurs.

4.2 LEARN MUTATIONS

The other approach is to learn the mutation itself. For that, it is necessary to analyze the protocol parameters, because they are directly related to the mutation, *i.e.,* the mutation influences the way how the protocol parameters change during the consensus. Therefore, the protocol parameters are relevant data to use in ML. The mutation is defined by the delay added in each message in a specific moment. Considering all the states of each peer and the delays that they used in each state as the input dataset, it is possible to identify a pattern in the delays, which define the mutation used. The dataset consists of all the states of all peers, where each row is a state of a peer. A single run generates an entire dataset. A peer changes the state when it receives a message that changes the protocol parameters. Table 8 shows a sample of a dataset.

PeerID	CoordID	Round	Phase	EstimatePeerID	EstimateValue	Decision	Peer1Vote	...	PeerNVote	DelayToPeer1	...	DelayToPeerN
2	2	1	1	2	"value"	null	0		0	-1.00E-03		-1.00E-03
3	2	1	1	2	"value"	null	0		0	-1.00E-03		-1.00E-03
4	2	1	1	2	"value"	null	0		0	-1.00E-03		-1.00E-03
5	2	1	1	2	"value"	"value"	0		0	-1.00E-03		0.00
5	2	1	1	2	"value"	"value"	0		0	-1.00E-03		0.00
6	2	1	1	2	"value"	"value"	0		0	0.00		-1.00E-03

Table 8: Sample of the dataset with protocol parameters.

In this approach, we use **Deep learning** (DL). DL is a class of ML algorithms that allows computational models, composed of multiple processing layers, to learn representations of data with multiple levels of abstraction[29]. In other words, it uses multiple layers to progressively extract higher-level features from the raw input. DL structures algorithms in layers to create an ANN that can learn and make intelligent decisions on its own. While traditional programs build analysis with data in a linear way, the hierarchical function of DL systems enables machines to process data with a nonlinear approach. In DL mode is necessary to define all the layers of the ANN, which can be manipulated by adding or removing layers or nodes in a layer, which makes it a flexible model.

The following section describes all the models are built and used. At first, we show how to build a model using DL, the points to be taken into account and how to integrate the model with the Simulator. After that, we describe three different models: *ANN to learn Centralized mutation*, *ANN to learn Ring mutation*, and *ANN to learn both mutations*.

4.2.1 *Build a Deep learning model*

Deep learning (DL) is a sub-field of ML. Therefore, the process to build a model is almost equal. What changes is instead of select a ML algorithm, we need to define all the layers of the ANN. The following steps show how to build a DL model. We show how to integrate and use the model in the Simulator and the problems in doing it in a simulated environment.

Gathering data

The first step is to gather the data to the model, because of the better the quality of the data, the better the results. In this case, the data is a sequence of snapshots collected from each peer during a simulation. Each of these snapshots contains the protocol parameters and the delays that respective peer use. A single run generates an entire dataset as the following example:

```
> simulator centralized 40 1 5 0 100 100 0 0 true
  2,2,,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,...,-0.000001,-0.000001,0.000000
  40,2,,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,...,-0.000001,-0.000001,-0.000001
  2,2,,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,...,0.000000,0.000000,0.000000
  1,2,,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,...,-0.000001,-0.000001,-0.000001
  (...)
```

Preparing data

Like the ML model, all the gathered data need to be processed to feed the DL model with appropriate data. Therefore, some parameters need to be normalized and others to be encoded. Indeed, only the *Delays* and the *Round* parameters need to be normalized. The delays receive a special normalization: all the negative delays are converted to -1 which allows to ignore messages to some peers. The other parameters are defined in a set of known values and because of that they only need a correct encoding.

In the previous models, we encode the categorical parameters, which was the *mutation*. In this case, instead of having one parameter, we have almost all the parameters to encode. These parameters are defined in three different sets. The first one is the set of peer ids, which we have the parameters *Peer id*, *Estimate of Peer id*, and *Coordinator id*. Another set is a small set with the decision value and the *null* value, present in the parameters *Decision* and *Estimate value*. The last one is the parameter *Phase*, which represent the phase in consensus that can be 1 (no faults) or 2 (suspect a fault). All these parameters need to be encoded to equalize its value. For example, when we have the *Peer id* with the value 1 and 5, for the model the last one is 5 times higher than the first one. Therefore, we apply the **One Hot Encoding** strategy to all these parameters to avoid any misunderstanding of the deep learning model. For each of these parameters is created new columns with all the possible values of each set and fill the respective column with the value 1. However, it is possible to reduce 1 column, because the combination of all column with value 0 it is an option too. Apply the encoding strategy makes the dataset growing up. For a dataset in an environment with 40 nodes, the generated dataset has **87 columns**. After applying the *one hot encoding*, the dataset will have **205 columns**, which is more than the double. If we generate a dataset in an environment with 100 nodes, the number of columns grows even more. This becomes a problem because when we want to train a model and we change the number of nodes, we need to generate a new dataset. However, if we define by default a huge set of peer ids that includes all the sets from the dataset, maybe is possible to avoid this problem. But, in this dissertation, we skip this problem and focus on a specific number of nodes.

The final step is to split the dataset. At first, we split the *input parameters* and the *target parameters*. In this case, the target parameters are the delays and the others are the input parameters. At the same time, we split each row of the dataset. Each row is a snapshot of a state of a peer and is an isolated state and the model needs to read each snapshot as an independent input. When it has several snapshots of a peer it will learn what to do in each case. Therefore, we split the dataset by the row, creating M examples of input, where M is the number of rows in the dataset. In the end, we split the input parameters and the target parameters in another two different sets: train set and test set. After that, we have all the datasets ready to use.

Define the Artificial Neural Network

The ANN built has five layers: one input layer, three hidden layers, and one output layer. The input layer has the same number of nodes that the input parameters to pass each parameter to a single node. On the other hand, the output layer has the same number of nodes that the output parameters, *i.e.,* the *delays*. Lastly, the hidden layers are defined by a fixed number of nodes. As the number of input parameters changes when the number of nodes changes, we need to change this value too. For these layers, there is not a correct value to use, so, it is necessary to test with different values and find the best one. However, it is common the hidden layers have fewer nodes than the input layer to force the network to learn compressed representations of the original input. Besides, we define three hidden layers in order to the ANN can process more the data, compared a single hidden layer. Figure 20 represents a ANN generic used.

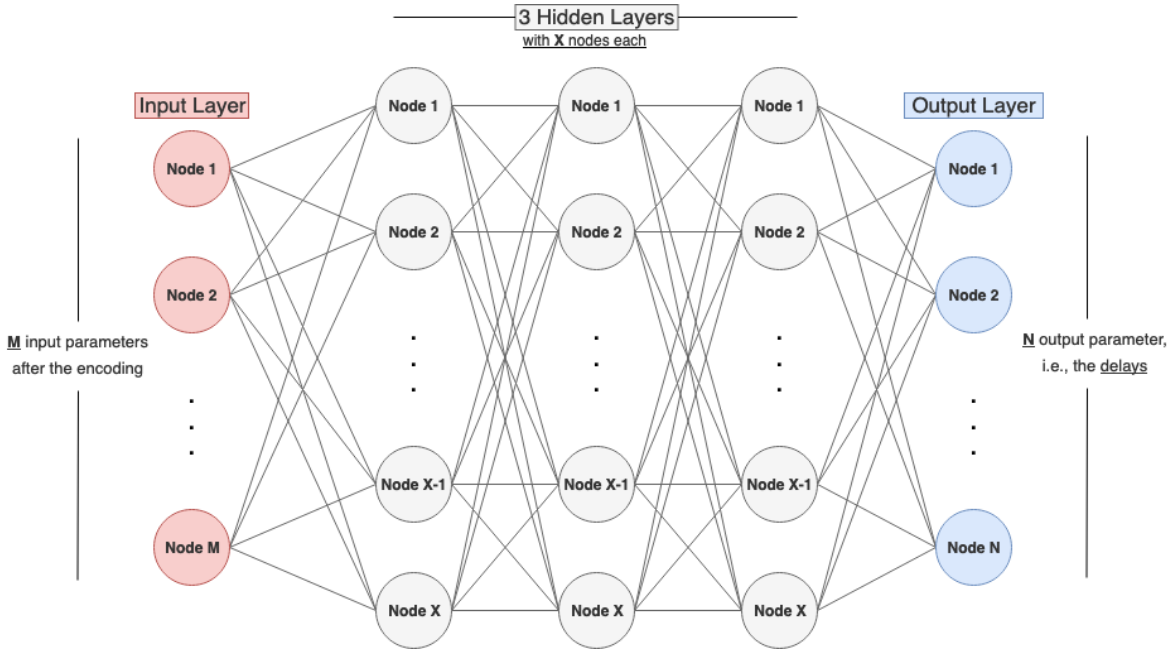


Figure 20: The ANN fully connected.

The hidden layers can be improved and consequently improve the model too. But in this dissertation, we will focus on a simple ANN to understand better the way it learns.

Training

Training an ANN is just minimizing the *loss function*, which requires to move in the negative direction of the derivative. For that, it uses the **back-propagation** process to calculate the derivatives and the **gradient descent** process to assign and adjust weights in the ANN to reduce the loss function, and, thus, descend through the gradient. In this case, we have

a regression problem, in which the ANN predicts continuous values. Therefore, the MAE was selected as the *loss function* to make the ANN learn from the continuous values. Besides, there are several types of gradient descents, but we use a popular one known as **adam**.

Evaluation

The Evaluation step checks if a model is good or not by using the *test* dataset. It is possible to measure the quality of the model by comparing the predictions given by the model and the expected values. Besides, during the training step it is also possible to analyze if the model is getting better after each iteration by analyzing the *loss function*, *i.e.,* the smaller the error, the more accurate the model. However, the measure given by the comparison with the *test* dataset is more reliable because it shows if the model is overfitting, underfitting or neither.

Hyperparameter tuning

In a DL model there are few configurable parameters. When we train the model with the *train* set, we define some parameters:

- **batch_size**: represents the number of samples that will go through the ANN at each training round;
- **epochs**: represents the number of times that the dataset will be passed via the ANN. The more epochs the longer it will take to run the model, which also gives better results. However, it is necessary to take into account if the model is overfitting;
- **optimizer**: is the gradient descent.

These parameters are defined at the moment of the training. There is not an ideal value for them. Therefore, it is necessary to test the model with different values and find the best one. Besides, more parameters can be changed, like the number of nodes in the hidden layer. Like the others, only testing we can find the best value of it.

Hyperparameter tuning is a good step to improve the model, but it must be careful to avoid the model is overfitting or underfitting, because of the over configuration based on only the train set and thus get bad results.

Prediction

In the Prediction step, we get the answer to our questions. In this case, the prediction will return the delays that a peer needs to use in a specific moment during the consensus. The positive or null delays represent the time in milliseconds that the peer needs to wait until sending the message. The negative delays represent all the target peers that the peer will not send a message.

4.2.2 *Integrating model in the Simulator*

The goal is the model predict the delays that a random peer normally used in a specific moment when it is running the mutation learned. Therefore, a strategy to test if the predictions are good enough is integrating the model with the Simulator. So, instead of using the common mutation that implements the Δ and Δ_0 functions, we load the model and reimplement both functions to use the model. If the protocol reaches the consensus with the same messaging pattern of the mutation learned, the model effectively learned the mutation. This approach not only allows you to test the model, but it also brings the protocol implementation closer to what is expected in the future.

The first step is to load the model in *Go*. The model is built with *TensorFlow*, which provides a *Go API* that is particularly useful for loading models created with *Python* and running them within a *Go* application. The model is built in the *Python* environment and when it is ready to use, it is exported and loaded in the *Go* environment. However, after this step, the problems appear.

Problem 1 In a real environment, each peer has the DL model loaded to use. However in a simulated environment, if each peer loads the model, the server that runs the simulation, crashes because the model takes a lot of memory, and requires a large amount of threads to execute. For that, it is impossible to load a model per peer in a simulated environment.

Solution 1 The solution is to reduce the number of models loaded without compromising performance. For that, we implement a **Load Balancer** which manages the requests to a small set of models. The load balancer is initialized at the beginning of the protocol and it is responsible to load M workers, each one with a loaded model ready to be used.

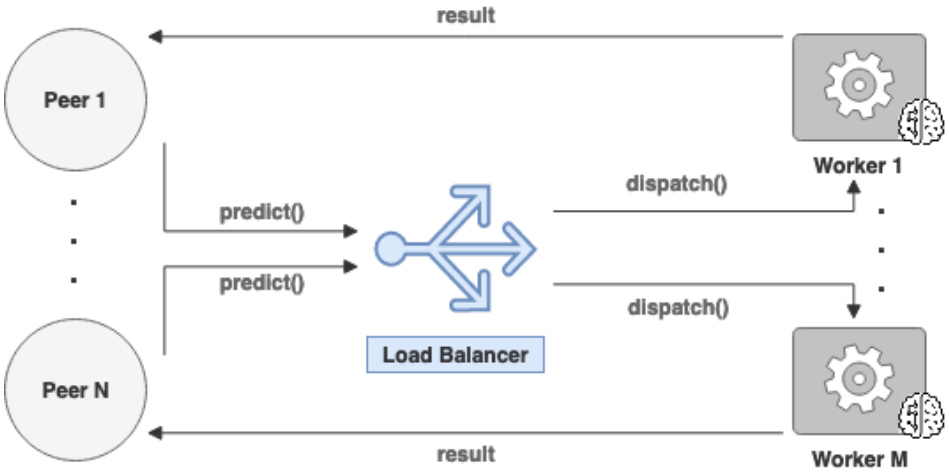


Figure 21: Load Balancer structure.

If we run an environment with N peers, the load balancer loads approximately $N/5$ models. There is not a perfect number of models. But it is necessary to take into account the number of models is not too low, which can overload the workers with requests, or too high, which can exhaust the memory of the server because of models that are not used at 100%. Figure 21 shows how the load balancer works.

The procedure is simple. A peer asks a prediction to the load balancer. The load balancer redirects the request to the worker with the fewer number of requests to process. Indeed, the load balancer redirects the request to the worker that is in the top of the list, which is a heap ordered by the number of requests. After the load balancer redirects the request, this request is put into the queue of the selected worker. Finally, the worker removes the request from the queue, process it through the model, and reply the result directly to the peer.

Introducing a load balancer solves the problem with several models which crash the server. However, the proposed solution is not perfect yet, because all peers use the same load balancer. When a peer tries to send a message, it uses the model to know if it needs to delay or not the message. However, all the peers do the same thing, which makes the number of requests grow quickly over time and consequently overload all the workers. For example, the time to a message get in the queue and to be processed, can take more than 1 *second*. This happens because of the peer makes N request for each message that it tries to send. This is solved by adding a cache mechanism before the peer starts sending messages. Therefore, when the peer tries to send a message to all others, it puts the prediction result in cache and then starts sending the messages. This simple change improves a lot the performance for **reducing the number of request from N to 1** for each message.

Problem 2 The model built is a regression model that returns approximate results, *i.e.,* each value of the prediction has a threshold which makes the values variate a little from the expected. Because of that, when the peers checks the Δ function it gets wrong results. For example, in a correct case, the $peer_i$ has a delay **0 seconds** to the peer $peer_j$, but the prediction returns a delay with the value $-$**0.0151**. The negative delays represent a message to ignore. Therefore, several messages have been ignored, when they should not. An example of an expected prediction and respective real prediction:

```
[ 0 0 ... -1(coordID) ... 0 0 0 ]                                  (expected)
[ -0.024 0.015 ... -0.899(coordID) ... 0.011 -0.002 -0.012 ] (real)
```

The values are almost the same, but it varies a little because of the threshold, and once the negative prediction indicates that the message is to be ignored, it leads to wrong results.

Solution 2 The solution to this problem is simple. It is only necessary to use rounded values.

Finally, after getting around all these problems, the protocol is ready to be used with the deep learning model.

4.2.3 *Artificial Neural Network to learn Centralized Mutation*

The first model aims to learn the centralized mutation, *i.e.*, how the peers exchange the messages with the others, mainly the coordinator in this case. In the centralized mutation, the coordinator is responsible by exchange all the message in the protocol between all peers. The coordinator, after sending the first message, starts to receive responses from the other peers and gathers their votes. When it gathers at least the majority of votes, it decides and broadcast its decision as well.

This model aims at predicting the delays through the protocol parameters and replicates the exchange pattern of the centralized mutation present in the Figure 22.

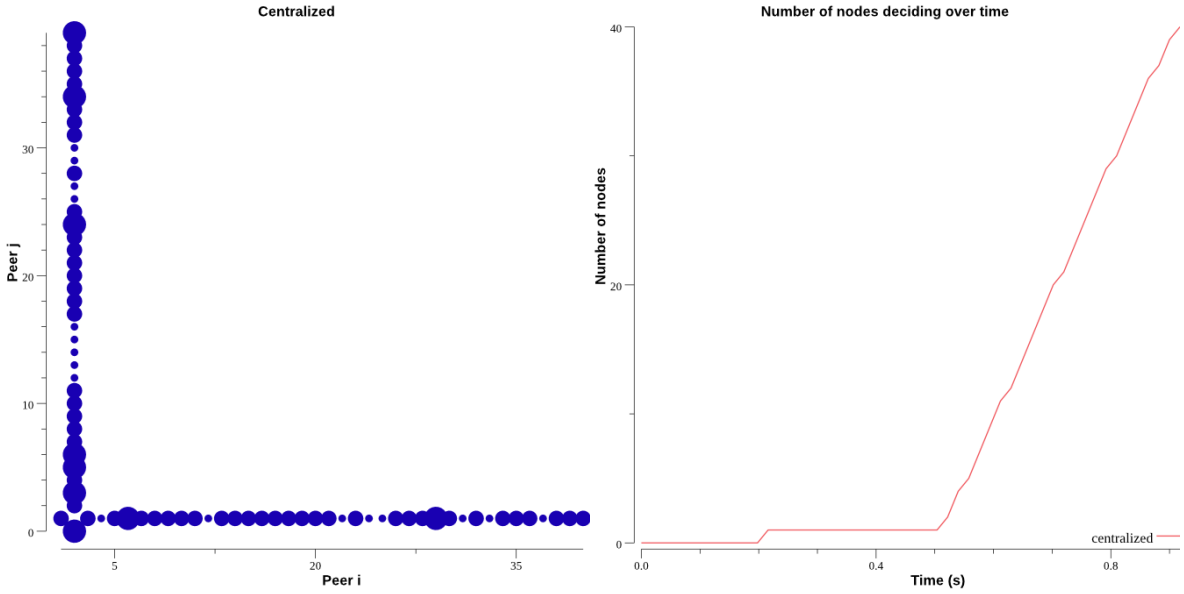


Figure 22: Message exchange pattern and respective time to reach consensus of *centralized* mutation.

Dataset - 1st iteration

For a first experience, the first dataset used all protocol parameters to evaluate the efficiency of the model, *i.e.*, if the model can learn from it. Therefore, we have all the protocol parameters as the input parameters and the delays as the output parameters, which are represented with gray and green respectively in Table 9.

PeerID	CoordID	Round	Phase	EstimatePeerID	EstimateValue	Decision	Peer1Vote	...	PeerNVote	DelayToPeer1	...	DelayToPeerN
2	2	1	1	2	"value"	null	0		0	0.00		0.00
40	2	1	1	2	"value"	null	0		1	-1.00E-03		-1.00E-03
2	2	1	1	2	"value"	null	0		1	-1.00E-03		0.00
20	2	1	1	2	"value"	null	0		0	-1.00E-03		-1.00E-03
2	2	1	1	2	"value"	null	0		1	-1.00E-03		-1.00E-03
30	2	1	1	2	"value"	null	0		0	-1.00E-03		-1.00E-03

Table 9: Sample of the dataset generated by the centralized mutation.

Over the previous dataset, we apply the *one hot encoding* strategy to the parameters that are a category value and normalize the continuous values as well. In the end, the input parameters and the output parameters are separated into two different data, like the previous models. Table 10 illustrates the processed dataset.

Input Parameters

PeerID1	...	PeerIDN	CoordID1	...	CoordIDN	Round	Phase0	Phase1	Peer1Vote	...	PeerNVote
0		0	0		0	1	0	1	0		0
0		1	0		0	1	0	1	0		1
0		0	0		0	1	0	1	0		1
0		0	0		0	1	0	1	0		0
0		0	0		0	1	0	1	0		1
0		0	0		0	1	0	1	0		0

EstimatePeerID1	...	EstimatePeerIDN	EstimateValueValue	DecisionValue
0		0	1	0
0		0	1	0
0		0	1	0
0		0	1	0
0		0	1	0
0		0	1	0

Target Parameters

DelayToPeer1	...	DelayToPeerN
0.00		0.00
-1.00		-1.00
-1.00		0.00
-1.00		-1.00
-1.00		-1.00
-1.00		-1.00

Table 10: Sample of the dataset generated by the centralized mutation after applying all preprocessing strategies.

Training - 1st iteration

The ANN selected for training has 165 nodes in the input layer, 128 nodes in each hidden layer and 40 nodes in the output layer. It receives the previous dataset of input parameters as input and makes predictions about **delays** for a particular state of a peer. Therefore, the result is an array of continuous values.

To avoid overfitting the ANN, we run the training process until 150 *epochs*, which, after testing the ANN, seems to us a good value.

Results - 1st iteration

The first experiment, although the ANN does not learn the message exchange pattern from the centralized mutation, the results are interesting. Figure 23 represents the results achieved. It shows that the ANN just learned that all peers except the coordinator only

sent messages to the coordinator. However, the coordinator does not learn that it needs to send messages to the other, which makes the mutation breaks until the retransmission mechanism reaches the maximum tries and converge to the early mutation.

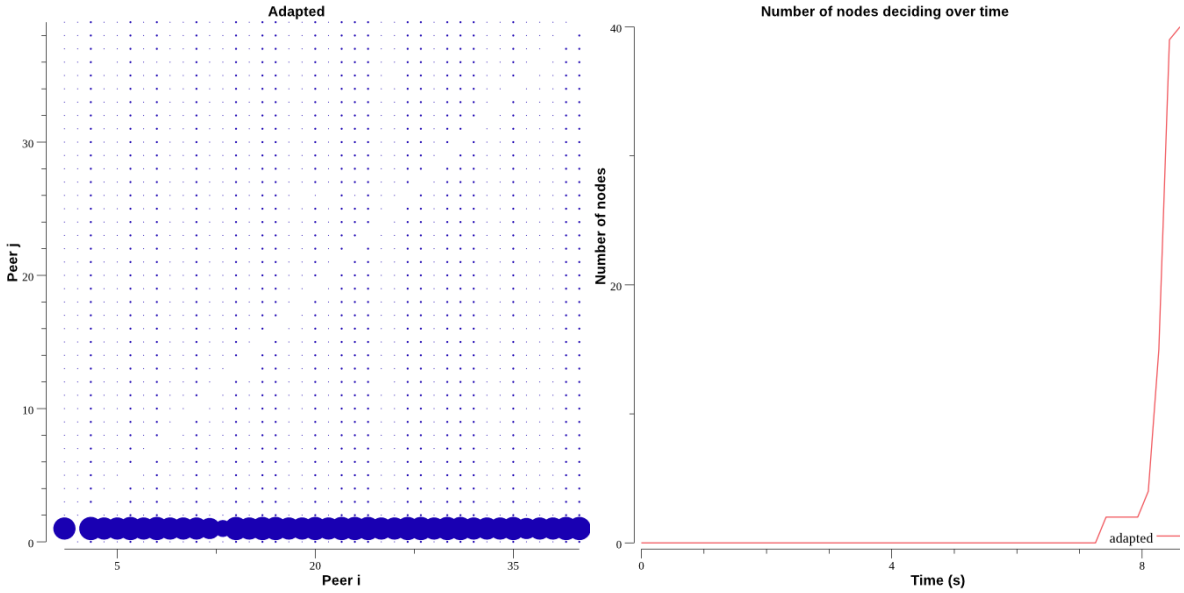


Figure 23: Message exchange pattern and respective time to reach consensus of *adapted* mutation derived from the *centralized* mutation - 1st iteration.

The previous results show us that the model started to learn some features, *i.e.*, it realized that all nodes only send a message to the coordinator. However, it does not learn the necessary to replicate the centralized mutation, which does not allow it to reach the consensus. Therefore, a more complex analysis of the previous dataset was made to find what it is wrong with.

Dataset - 2nd iteration

The previous experiment shows us that the model needs to be improved. In ML, some strategies help to understand how the data are correlated. In this case, we used the heat map which works by correlation. It shows the variables that are correlated to each other from a scale of 1 being the most correlated and -1 is not correlated at all. However, it is not possible to correlate strings, we can only correlate numerical features. However, once we have our categorical data encoded, it is not a problem. Figure 24 illustrates the heat map generated for an example with only 10 nodes. The heat map for a bigger environment grows a lot and makes the heat map imperceptible.

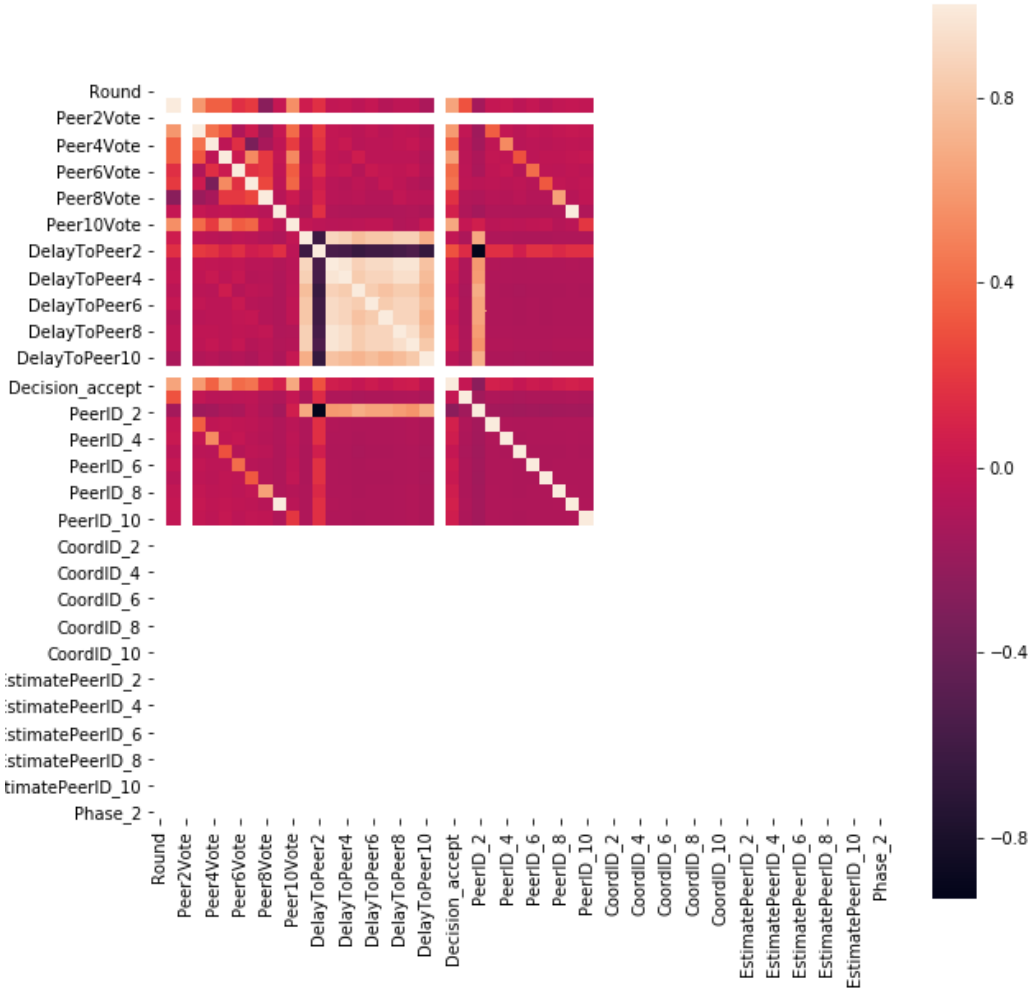


Figure 24: Heat map of all protocol parameters.

As we can see, there is a subset of parameters that are correlated, but there is another subset of parameters that are not correlated with anyone. The most important part of this diagram is to see what parameters are correlated with the delays (output parameter). So, we start to remove these parameters that have not a correlation and keep the parameters that have. So, we select the following parameters: *peerID*, *coordID*, *Peer vote*, and *Decision*. We keep the *coordID*, in spite of it not having any correlation, as it is an important parameter in the protocol. The heat map does not detect any correlation because its value was always the same. However, for an environment with faults, the parameter may change and thus start to correlate with the others. In this dissertation, we are focused in a simple environment without faults though.

We also analyze the parameters that the centralized mutation use. The centralized mutation has in the Δ_0 function several conditions that must be true for the sending process

to be allowed, which some of these are logic expressions calculated in the moment. These conditions are:

- the target peer ID is the coordinator;
- if the message is fresh;
- if the majority ($2N + 1$) of the peers has already voted.

In the centralized mutation, in the first iteration, the coordinator broadcasts the message to all peers and then receives the answers. However, the problem starts from here. The coordinator waits until gather at least the majority of votes and at this moment, start to broadcast the votes gathered and decides. The current parameters do not allow the ANN to understand when the waiting process happens and why. Therefore, we add the *isMajority* parameter to show to the ANN that when this parameter is *true* the peer stops gathering votes and starts sending messages again. Additionally, after analyzing the dataset we found a problem common in ML.

Problem The ANN did not learn all features, because the input data is not balance. In the ML field, it is called **data imbalance**. For an environment of 40 nodes, a run of the simulator generate a dataset with almost 150 snapshots. In these snapshots, the coordinator only appears 20 times and in these cases, it only broadcasts a message to the others 2 times. Therefore, the number of rows that shows that the coordinator broadcasting a message is very small. For a broadcast of the coordinator, the delays are:

```
[ 0 0 ... -1(coordID) ... 0 0 ]
```

In the other cases, the coordinator does not send nothing until it gathers $2N + 1$ votes. In theses cases the delays are:

```
[ -1 -1 -1 .. -1 -1 ]
```

The other peers, only send message to the coordinator, so their delays are:

```
[ -1 -1 ... 0(coordID) ... -1 -1 ]
```

The input dataset has very few rows of the coordinator broadcasting a message to the others and as such the ANN converges to the other more common rows. There is a huge difference between a row that represents a broadcast of the coordinator and the row that represents a message sent only to the coordinator. However, the number of rows which the coordinator stops sending messages is higher than the rows which it broadcasts a message. and more similar to the row that sends a message only to the coordinator. Because of

that, the results converge to the most common row which is sending a message to the coordinator.

Solution To solve this problem, we increase the number of snapshots by increasing the number of runs. Instead of using a dataset generated by a single run, we will use a dataset generated by, approximately, 30 runs. Increasing the size of the dataset, increase the number of cases that the coordinator broadcasts a message.

Taking into account the previous analysis, we build a new dataset. The new dataset has fewer parameters, but a lot more entries. The dataset is presented in Figure 11.

Input Parameters

PeerID1	...	PeerIDN	CoordID1	...	CoordIDN	isMajority	Peer1Vote	...	PeerNVote	DecisionValue
0		0	0		0	0	0		0	0
0		1	0		0	0	0		1	0
0		0	0		0	0	0		1	0
0		0	0		0	0	0		0	0
0		0	0		0	0	0		1	0
0		0	0		0	0	0		0	0

Target Parameters

DelayToPeer1	...	DelayToPeerN
0.00		0.00
-1.00		-1.00
-1.00		0.00
-1.00		-1.00
-1.00		-1.00
-1.00		-1.00

Table 11: The new dataset generated by centralized mutation and processed by all preprocessing strategies.

Training - 2nd iteration

The training process is the same as the previous with only a little change. Removing parameters in the dataset, changes the ANN structure in the input layer. Instead of **165** nodes in the input layer, we use **122** nodes. As there less than **128** nodes in the input layer, we change the number of nodes in the hidden layer to **100**. The output layer stays the same.

In the first try, we feed the ANN, with a dataset with 10 runs. The delay values converge to the correct case for each peer including the coordinator. However, it was not perfect, so we increase the dataset even more until getting something useful.

Once we increase the dataset size, the time of training increases as well. It is important to refer that the first dataset with a single run had **100 KB**, while a dataset with 30 runs has **7.9 MB**. Because of that the training process instead of taking **5 seconds** to train, it takes, approximately, **3 minutes**.

Results - 2nd iteration

After testing several times and improving the dataset, we achieved the target result: the ANN learns all the features of centralized mutation and replicates it when we run the simulator with the *adapted* mutation that use the ANN. An interesting fact is that the adapted mutation **got a better time to reach the consensus than the centralized mutation**. Figure 25 represents these results.

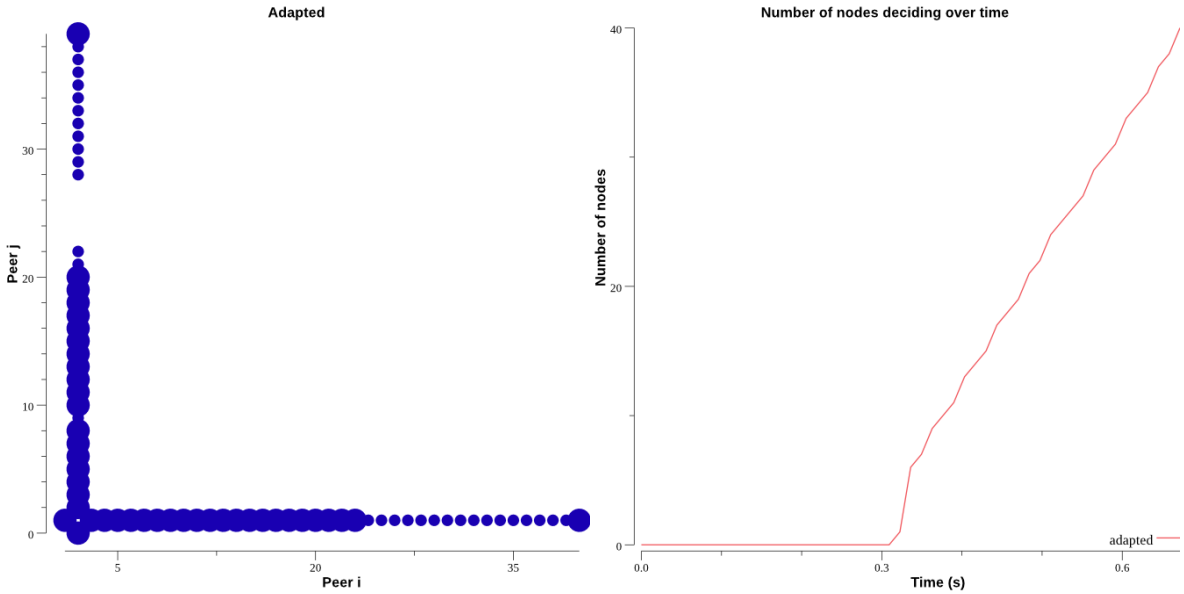


Figure 25: Message exchange pattern and respective time to reach consensus of *adapted* mutation derived from *centralized* mutation - 2nd iteration.

As we can see in the plot in the left, which represents the message exchange pattern, some messages that should have been sent, was not sent. However, this is not a problem, because the mutation managed to reach the consensus. Additionally, if the DL model replicates the centralized mutation in its entirety, probably we would face an overfitting model. Therefore, these results are good and reliable.

Once we get a ANN that learns the centralized mutation, the next step is to learn another mutation: *ring* mutation. The ring mutation is completely different from the centralized mutation because of the patterns its present. This difference requires a different approach to the problem, which is described in detail in the following subsection.

4.2.4 *Artificial Neural Network to learn Ring Mutation*

In the ring mutation, each peer has the same importance, while in the centralized mutation the coordinator peer is the most important because it is responsible for exchanging all mes-

sages in the protocol. Without it, the protocol does not work and does not reach consensus. In the previous model, it learns that there is a peer that exchanges the message between all other peers and these only reply to it. In other words, there are only three patterns in the previous model: the coordinator broadcasts a message, a normal peer replies to the coordinator and the coordinator wait until gather the majority of votes. In the ring mutation there are more than three patterns and it is hard to identify them. In the this mutation, each peer sends a message to its successor and predecessor. Therefore, the successor and predecessor change for each peer, which creates a pattern for each one. This mutation increased the problem complexity and so it will be a problem for the model to learn all the patterns properly.

In this section, we present several models that aims at to learn the ring mutation patterns and may able to replicate its message exchange pattern as it is present in Figure 26. We also present the problems faced in building the models.

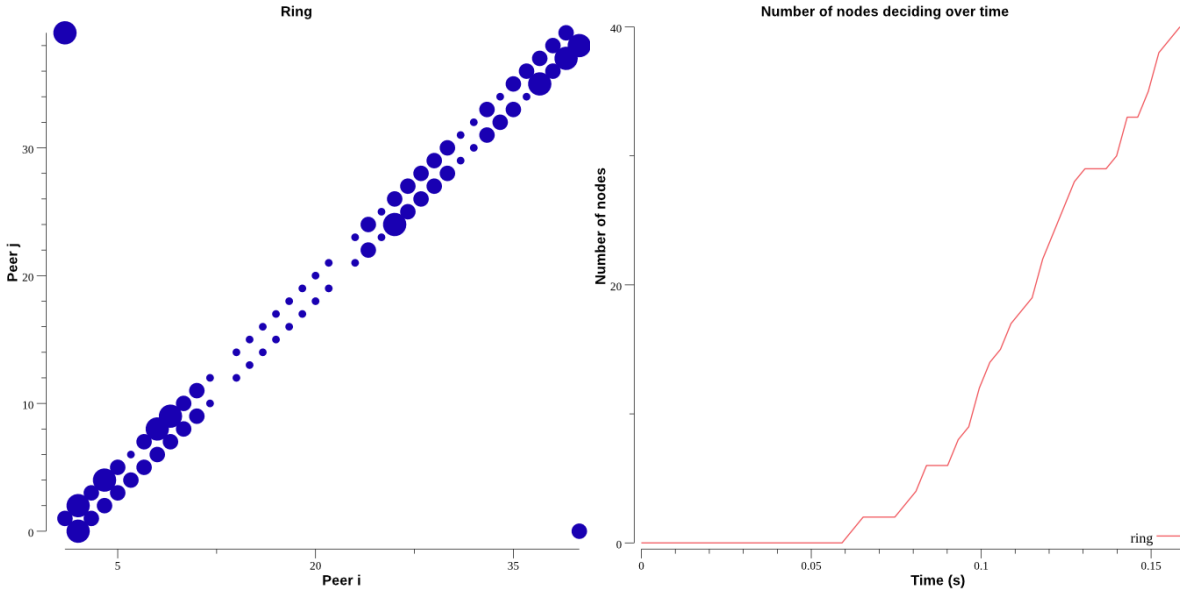


Figure 26: Message exchange pattern and respective time to reach consensus of *ring* mutation.

Dataset - 1st iteration

The previous dataset gave good results, because of that we will continue using the previous parameters that it has as the input parameters. The dataset is present in Table 12.

PeerID	CoordID	isMajority	Decision	Peer1Vote	...	PeerNVote	DelayToPeer1	...	DelayToPeerN
2	2	0	null	0		0	0.00		-1.00E-03
1	2	0	null	1		0	-1.00E-03		0.00
40	2	0	null	1		1	0.00		-1.00E-03
39	2	0	null	1		1	-1.00E-03		0.00
40	2	0	null	1		1	-1.00E-03		-1.00E-03
3	2	0	null	0		0	-1.00E-03		-1.00E-03

Table 12: Sample of the dataset generated by the ring mutation.

Over the previous dataset we apply the encoding and normalization strategies and split the input parameter and output parameters. Table 20 represent the result of it.

Input Parameters

PeerID1	...	PeerIDN	CoordID1	...	CoordIDN	isMajority	Peer1Vote	...	PeerNVote	DecisionValue
0		0	0		0	0	0		0	0
1		0	0		0	0	1		0	0
0		1	0		0	0	1		1	0
0		0	0		0	0	1		1	0
0		1	0		0	0	1		1	0
0		0	0		0	0	0		0	0

Target Parameters

DelayToPeer1	...	DelayToPeerN
0.00		-1.00
-1.00		0.00
0.00		-1.00
-1.00		0.00
-1.00		-1.00
-1.00		-1.00

Table 13: Sample of the dataset generated by the ring mutation after applying all preprocessing strategies.

Training - 1st iteration

For a first experiment, we use the same ANN that was used in the centralized mutation. Therefore, the ANN will have 122 nodes in the input layer, 100 nodes in each hidden layer and 40 nodes in the output layer. The dataset used has **13.5 MB**, which makes the training process takes, approximately **5 minutes**. As the previous models, we run the training process until 150 *epochs*.

Results - 1st iteration

The ANN learned almost nothing from the dataset. Only some peers were able to send messages to their neighbors, but it was only a small set of them. The mutation could not

reach to the consensus and because of that, it converged to the early mutation. Figure 27 illustrates the results.

As mentioned above, this mutation has more complex patterns. Even the strategy of increasing the dataset with more snapshots to give more example to the model does not solve the obstacles. Therefore, we need to dive into the dataset and the mutation and figure out the problems for this mutation.

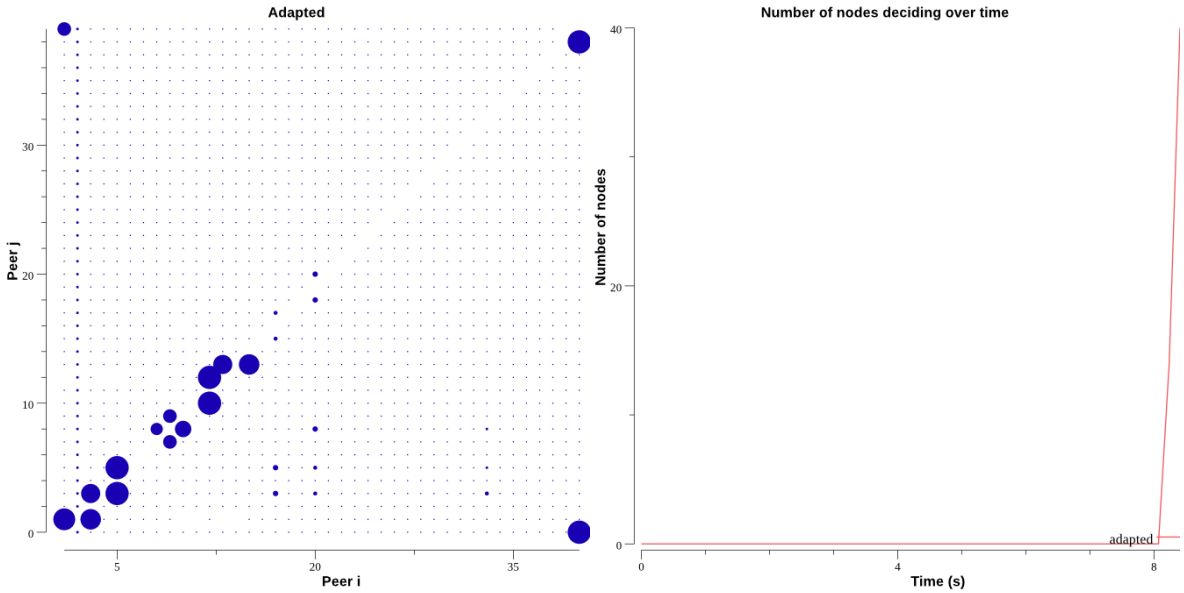


Figure 27: Message exchange pattern and respective time to reach consensus of *adapted* mutation derived from *ring* mutation - 1st iteration.

Dataset - 2nd iteration

The current ANN is having trouble identifying the patterns from the ring mutation. Therefore, a more complex analyze is made to find the problem. After analyze the delays that were generated, we find the problem.

Problem The ring mutation has a common pattern that frequently repeats at each peer, which is to send a message to its predecessor and successor. This pattern generates an array of delays that only has two different values in N. For example, the array of delays of the coordinator when it starts the protocol is:

```
[-1  -1  ...  -1  0  -1(coordID)  0  -1  ...  -1  -1]
```

After it sends its first message, it will wait until it gather the majority of votes and therefore stop sending messages. In other words, its array of delays will be a list with values equal to -1:

```
[ -1 -1 ... -1 -1 -1(coordID) -1 -1 ... -1 -1 ]
```

In the centralized model, when the coordinator broadcasts a message, the values of the array of delays change almost all to 0, keeping only its value equal to -1, because it does not send messages to itself. On the other hand, when the coordinator or other peer is waiting for new votes, the array of delays is an array of values equal to -1. For the ANN, the array when the coordinator broadcasts a message is easy to learn, because there is a huge difference between this pattern and when it stops sending messages to wait for new votes. This difference is associated with an error that makes the ANN learns. The bigger the difference, the bigger is the probability for the model to learn.

In a more detailed perspective, this is due to the *gradient descent*, which aims to get the point where the error, given by the *loss function*, is minimal. The *loss function* used is the MAE, which calculates the error by measuring the difference between two continuous variables (Real and Predicted). Since the ANN learns by minimizing this error, when we have a big discrepancy in an entry, the ANN get a huge error if it tries to converge this case to a common case. Therefore, it starts to diverge this case to minimize the error and thus starts to identify the pattern and learn it.

However, in the ring mutation, this is a problem, because the difference between rows is minimal. At most, there are only two values in the array that change. This causes most patterns not to be recognized because the associated error will be too small. Therefore, the ANN converges to the most common case, which is when a peer is waiting for new votes. As expected, this problem replicates to all other peers as well.

Solution The current problem is to make the model understand that there is a variation in the delays around the position of the peer. The only parameter that can help it to identify this variation is the peerID. For example, for the $peer_2$ we have these arrays:

```
[ 0 1 0 0 0 ... 0 ]      (peerID)
[ 0 -1 0 -1 -1 ... -1 ]   (delays)
```

The peerID parameter, after the encoding process, it is converted to an array of N values, which has the value 1 in the position of the specific peer. As we can see, in the previous example, the $peer_2$ has the value 1 in the second position. This array is related to the array of delays, as this array can identify the peer position in the array of delays. Around this position are the delay used in the ring mutation. The current model can not identify this pattern, but there is a solution. Some parameters are more important and influential than others. In this case, the peerID parameter influences more than the other parameters, because it can identify the peer in the array of delays. However, all current parameters have the same weight in the model. Therefore, the solution is **denormalize** the model, and

apply more weight to the parameters that more influence the model to make the model understand the existing variation in specific situations.

The process to denormalize the model consists in applying an extra factor to the parameter, multiplying its value by the factor. Taking the previous example and a factor of 1000 we have:

```
[ 0 1000 0 0 0 ... 0 ]
```

The denormalization process will give another perspective of the dataset to the ANN. Therefore, we decide to denormalize all the parameters that represent the identification of peer (IDs) and keep the others equal. So, we apply a factor of 1000 to the parameters *peerID* and *coordID*. The factor is a little exorbitant, but we decide to apply a big value to see the impact of the denormalization in the model. Besides, it is expected that the peerID parameter will influence more than the coordID because it is very related to the position where the variation of delays happens. The resulting dataset is:

Input Parameters

PeerID1	...	PeerIDN	CoordID1	...	CoordIDN	isMajority	Peer1Vote	...	PeerNVote	DecisionValue
0		0	0		0	0	0		0	0
1000		0	0		0	0	1		0	0
0		1000	0		0	0	1		1	0
0		0	0		0	0	1		1	0
0		1000	0		0	0	1		1	0
0		0	0		0	0	0		0	0

Target Parameters

DelayToPeer1	...	DelayToPeerN
0.00		-1.00
-1.00		0.00
0.00		-1.00
-1.00		0.00
-1.00		-1.00
-1.00		-1.00

Table 14: Sample of the preprocessed dataset of the *ring* mutation after denormalize the *peerID* and *coordID*.

Training - 2nd iteration

The training process is the same as the previous. The ANN has 122 nodes in the input layer, 100 nodes in each hidden layer and 40 nodes in the output layer. Besides, the dataset and the number of *epochs* are also the same, but because of the denormalization the training process takes, approximately, **6 minutes** instead of 5 minutes.

Results - 2nd iteration

The results show that this model managed to learn every single pattern of each node, which makes the protocol reach the consensus without to converge to the early mutation. Therefore, we achieved another goal, making an **ANN learn the ring mutation**. The results are present in Figure 28. However, this model is not entirely complete. The model only learns the main pattern of each peer, which makes the peers sending messages constantly to its predecessor and successor. The pattern to wait for the majority of votes it does not learn it from the dataset. This happens because of the denormalization process we only change the peerID and keep the other parameters equal. However, other parameters are also important and due to the peerID they make a minimal difference.

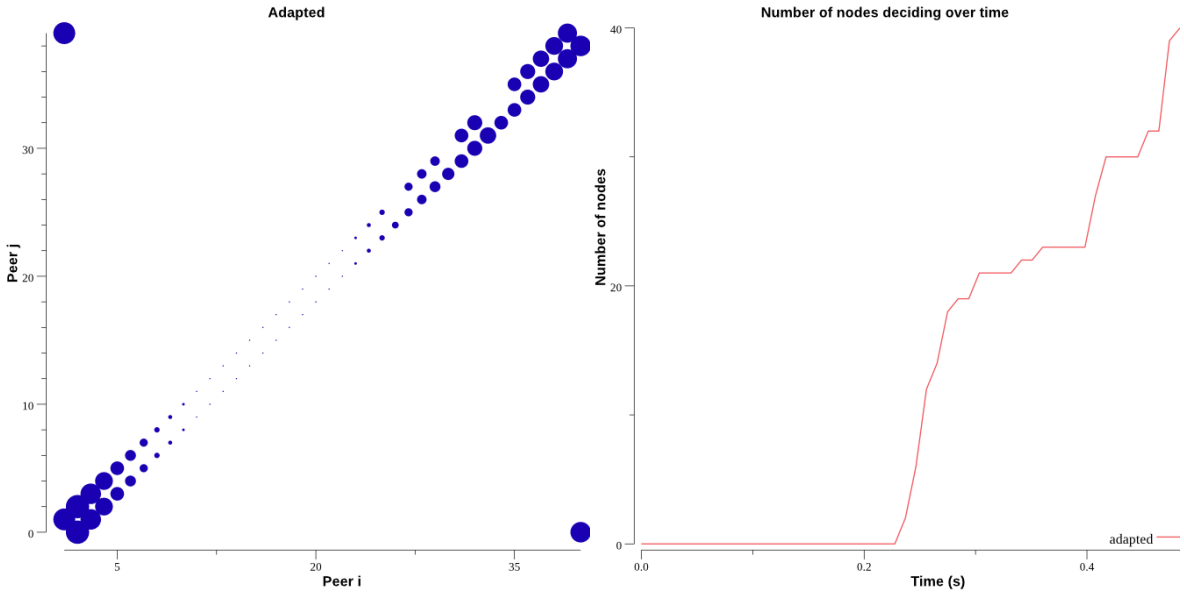


Figure 28: Message exchange pattern and respective time to reach consensus of *adapted* mutation derived from *ring* mutation - 2nd iteration.

Dataset - 3rd iteration

To solve the previous problem, we start to analyze all parameters that can identify new patterns for the model. The model already learns the pattern of sending a message to the predecessor and successor, because of the peerID. Therefore, we only need to detach the parameter that is related to the pattern to wait for new votes until gathering the majority of the votes. There are only two parameters that are most related to this pattern: *votes* that change always that a peer gathers a new one and *isMajority* that change the value when the peer has the majority of the votes. We will denormalize these parameters as we did with

the peerID and coordID. However, in this model, we also will try to find the best value of the factor for each parameter.

The first step is to reduce the previous factor to 100 because a factor of 1000 causes a big discrepancy with the other parameters. After that, we test several combinations of these parameters to find the best value for each one. Indeed, we will not find the best value but a value that it is close to it, because finding the best values will take a lot of time.

After making several tests, we find a combination of factors that fit the parameters. The parameters that will have a factor are: *peerID*, *votes*, and *isMajority*. We remove the factor to the coordID because during the tests we realized that it influences almost nothing the model. Therefore, we apply the following factor:

Parameter	Factor
peerID	100
votes	10
isMajority	5
coordID	1
decision	1

Table 15: The factors applied to each parameter.

The denormalization originated the following table:

Input Parameters

PeerID1	...	PeerIDN	CoordID1	...	CoordIDN	isMajority	Peer1Vote	...	PeerNVote	DecisionValue
0		0	0		0	0	0		0	0
100		0	0		0	0	10		0	0
0		100	0		0	0	10		10	0
0		0	0		0	0	10		10	0
0		100	0		0	0	10		10	0
0		0	0		0	0	0		0	0

Target Parameters

DelayToPeer1	...	DelayToPeerN
0.00		-1.00
-1.00		0.00
0.00		-1.00
-1.00		0.00
-1.00		-1.00
-1.00		-1.00

Table 16: Sample of the preprocessed dataset of the *ring* mutation after denormalize the parameters that most influence.

Training - 3rd iteration

The same process as the previous. The model trains a ANN with 122 nodes in the input layer, 100 nodes in each hidden layer and 40 nodes in the output layer. However, for this model, the dataset was increased to **16.3 MB** to improve the learning process. Finally, the ANN was configured with 150 *epochs*, same as the previous models and takes, approximately, **13 minutes** to train. It is a big difference compared to the previous models.

Results - 3rd iteration

After integrating the model with the simulator, we realized that it almost achieved a complete model that learns all patterns. The model can replicate the ring mutation in most of the cases, but there are some cases that it does not behave as expected. For example, in some cases, a peer that suppose to wait until it gathers the majority of votes, it sends a message sometimes during this phase. However, since it only happens sometimes, it is not a serious problem.

Finally, we have a model that can replicate the ring mutation and reach the consensus, but **this model takes more time to do it** than the original ring mutation. The results of this model are illustrated in Figure 29.

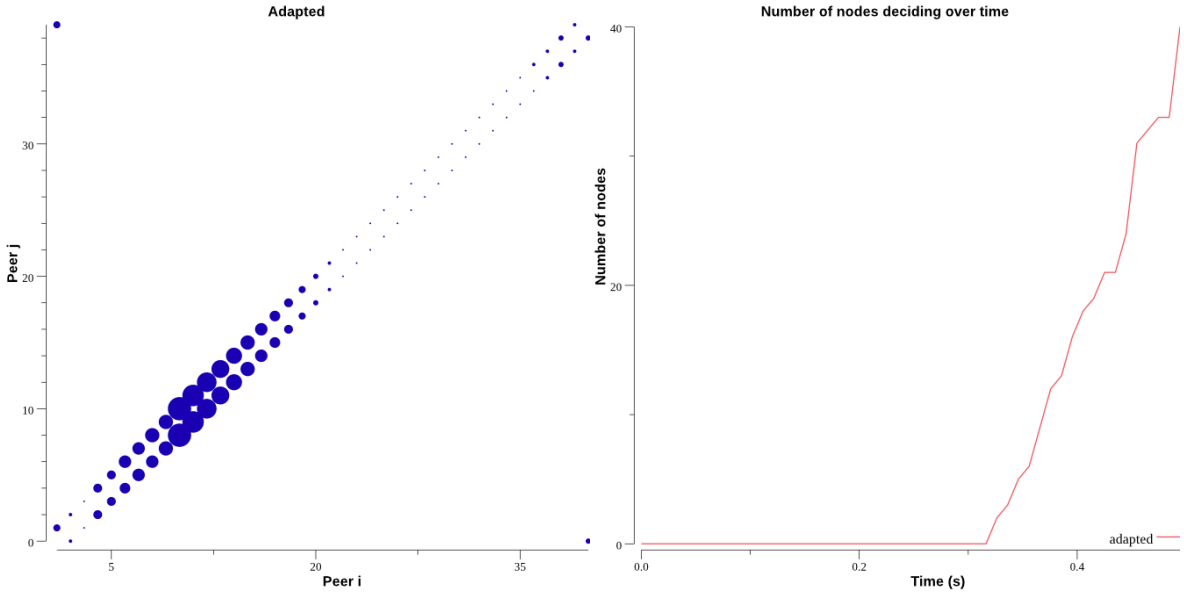


Figure 29: Message exchange pattern and respective time to reach consensus of *adapted* mutation derived from *ring* mutation - 3rd iteration.

4.2.5 *Artificial Neural Network to learn both Mutations*

Finally, after we get two models that learn the centralized mutation and the ring mutation respectively, we will move to merge both models and get a model that exchange between them when an environment parameter change. But there is a problem in that. These two models use different configurations, as the ring mutation is denormalized and the centralized mutation is not. Therefore, before building the model that merges both mutations, we need to adapt the centralized mutation to the same conditions as the ring mutation, since the ring mutation is the most limited model because of the denormalization.

Dataset - 1st iteration

The goal is the adapt the centralized model to the same conditions as the ring model. Therefore, first, we test the denormalization applied in ring model. If the centralized model does not learn anything, it will be necessary to find a set of factors that fit both models. However, if we take a look to the parameters that was denormalized in the ring mutation, we can see that the same parameters influence the centralized mutation. The *peerID* helps to identify the pattern which the coordinator broadcast a message, and the *votes* and *isMajority* in the patterns to wait until gather majority of votes and send message after that. Therefore, the denormalization of these parameters can work well.

The resulting dataset, after applying denormalization, is present in Table 17.

Input Parameters

PeerID1	...	PeerIDN	CoordID1	...	CoordIDN	isMajority	Peer1Vote	...	PeerNVote	DecisionValue
0		0	0		0	0	0		0	0
0		100	0		0	0	0		10	0
0		0	0		0	0	0		10	0
0		0	0		0	0	0		0	0
0		0	0		0	0	0		10	0
0		0	0		0	0	0		0	0

Target Parameters

DelayToPeer1	...	DelayToPeerN
0.00		0.00
-1.00		-1.00
-1.00		0.00
-1.00		-1.00
-1.00		-1.00
-1.00		-1.00

Table 17: Sample of the preprocessed dataset of the *centralized* mutation after denormalize the parameters that most influence.

Training - 1st iteration

The model trains an ANN with 122 nodes in the input layer, 100 nodes in each hidden layer and 40 nodes in the output layer, as the previous models. We increase the previous dataset of centralized mutation to, approximately, **11 MB** to improves the learning process. Finally, the ANN was configured with 150 *epochs*, as the previous models. The training process takes, approximately, **4.5 minutes**.

Results - 1st iteration

On the first try, we achieved a **model that learns the centralized mutation** using denormalization. Compared to this model with the previous that learns the centralized mutation, it **takes almost twice as long**. However, as the main goal is to merge two models, this is not a serious problem. Figure 30 shows the results of this model.

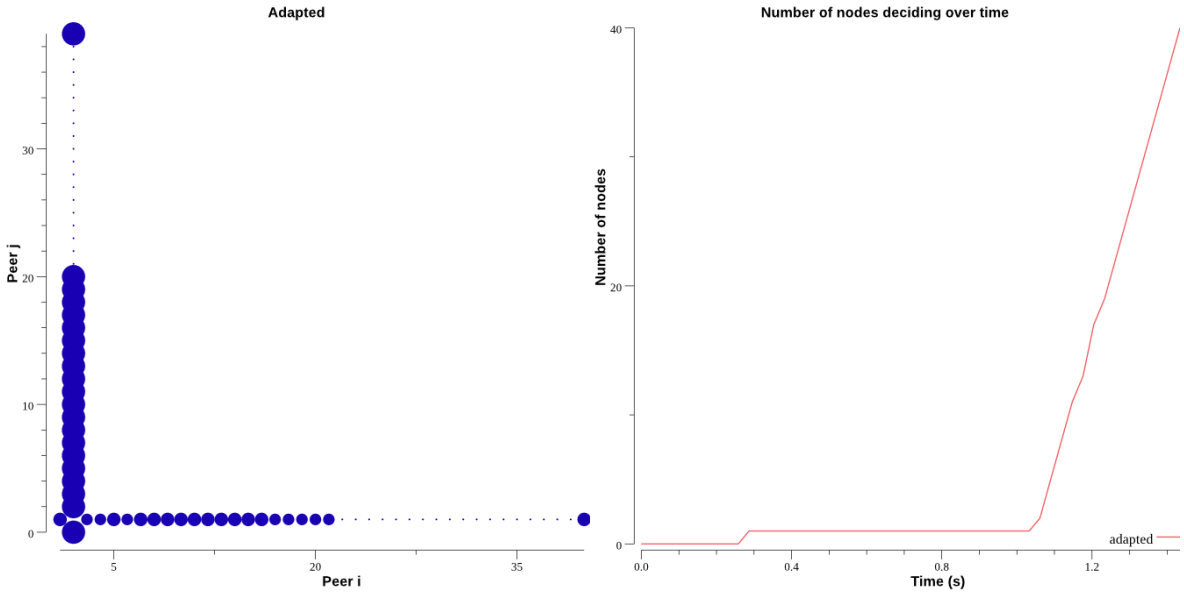


Figure 30: Message exchange pattern and respective time to reach consensus of *adapted* mutation derived from *centralized* mutation after denormalization.

Dataset - 2nd iteration

We already have two models that learned the respective mutations, so, the next step is to merge them. Since the models are similar, the merge process is simplified.

First, it is necessary to generate a new dataset that contains several snapshots of both mutations. Besides, this dataset will contain an extra parameter, which will be the differentiating parameter that distinguishes the two mutations. Since the goal for the model is to be able to change between two mutations when the environment change, we select the

bandwidth parameter. This environment parameter represents the number of messages that the peer can process in one second. For an environment with a bandwidth of 500, the best mutation will be the centralized mutation. On the other hand, for an environment with a smaller bandwidth like 100, the ring mutation will fit better. So, the new dataset will contain snapshots of the centralized mutation in an environment with a bandwidth of 500, while the ring mutation with a bandwidth of 100. The resulting dataset is present in Table 18.

Bandwidth	PeerID	CoordID	isMajority	Decision	Peer1Vote	...	PeerNVote	DelayToPeer1	...	DelayToPeerN
100	1	2	0	null	1		1	-1.00E-03		0.00
100	36	2	1	"value"	1		1	-1.00E-03		-1.00E-03
500	9	2	0	null	1		1	-1.00E-03		-1.00E-03
100	8	2	0	null	0		0	-1.00E-03		-1.00E-03
100	2	2	1	"value"	1		1	0.00		-1.00E-03
500	2	2	0	null	1		1	-1.00E-03		-1.00E-03

Table 18: Sample of the dataset created from the merge of two datasets created by both mutations.

As the model is denormalized, we need to keep the bandwidth parameter denormalized as well. If we normalized only this parameter, it will assume values smaller or equal than 1, which is too small. Since it is a differentiating parameter it needs to be a parameter with a big weight, because it should be one of the most influential in the model. Keeping the bandwidth denormalized, we have the following preprocessed dataset:

Input Parameters

Bandwidth	PeerID1	...	PeerIDN	CoordID1	...	CoordIDN	isMajority	Peer1Vote	...	PeerNVote	DecisionValue
100	100		0	0		0	0	10		10	0
100	0		0	0		0	5	10		10	1
500	0		0	0		0	0	10		10	0
100	0		0	0		0	0	0		0	0
100	0		0	0		0	5	10		10	1
500	0		0	0		0	0	10		10	0

Target Parameters

DelayToPeer1	...	DelayToPeerN
-1.00E-03		0.00
-1.00E-03		-1.00E-03
-1.00E-03		-1.00E-03
-1.00E-03		-1.00E-03
0.00		-1.00E-03
-1.00E-03		-1.00E-03

Table 19: Sample of merged dataset with the *bandwidth* normalized.

Training - 2nd iteration

The training process keep the previous ANN. Therefore, the ANN has 122 nodes in the input layer, 100 nodes in each hidden layer and 40 nodes in the output layer. However, the dataset used is a merge of two datasets, which makes it have, approximately, **34 MB**, which is a very big dataset. Finally, the ANN was configured with 150 *epochs*, as usual, and takes, approximately, **15 minutes** to train.

Results - 2nd iteration

After integrating the model with the simulator, we realized that the model almost forgot everything that it learns. In both environment, when the protocol starts, **there are only four peer exchanging messages**. However, in the environment that is expected the centralized mutation run, the protocol executes a message exchange pattern similar to it, but it does not reach to the consensus because of the coordinator did not learn all its patterns. On the other hand, the other environment execute a message exchange pattern completely different from the ring mutation. This causes the protocol only reach to the consensus when the protocol starts to converge to the early mutation. The results of the environment with a bandwidth of 500 are present in Figure 31.

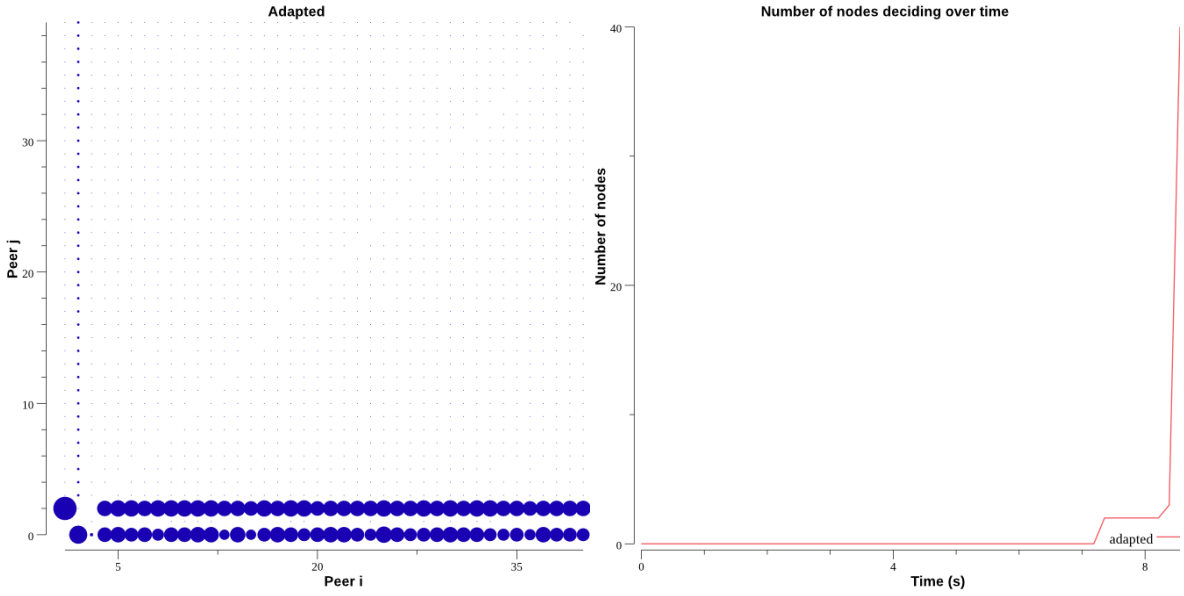


Figure 31: Message exchange pattern and respective time to reach consensus for the *adaptive* mutation in a environment with a bandwidth of 500.

On the other hand, the results of the environment with a bandwidth of 100 are present in the Figure 32.

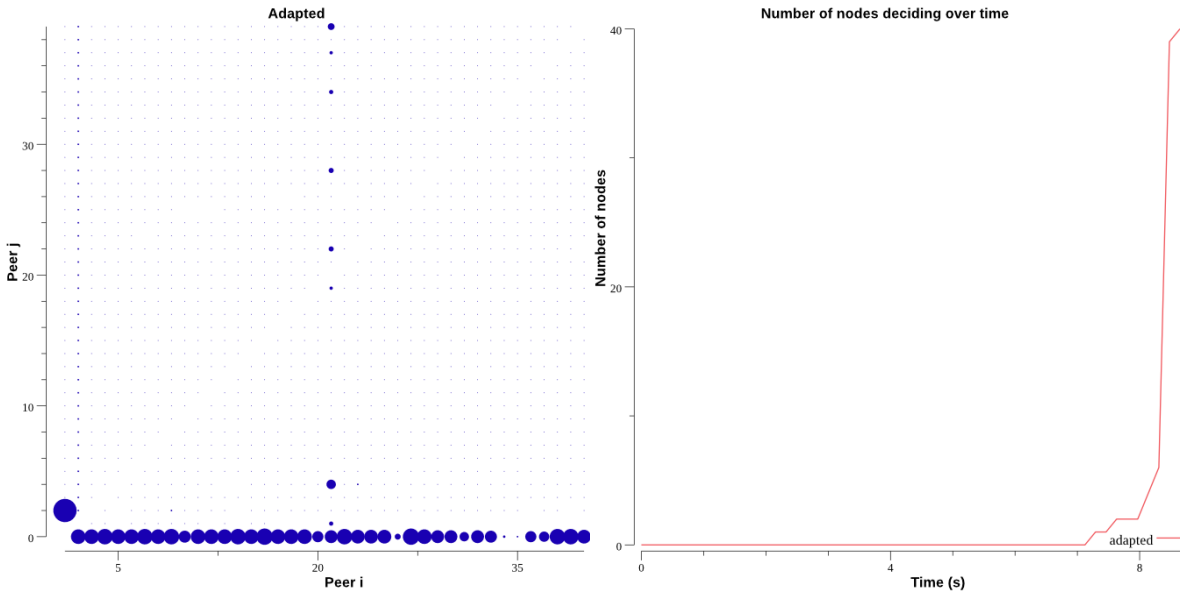


Figure 32: Message exchange pattern and respective time to reach consensus for the *adaptive* mutation in a environment with a bandwidth of 100.

Dataset - 3rd iteration

In the previous results, we can see that all that the models learn was undone. The only parameter that can cause this is the bandwidth because before we add it to the dataset, the models work well. Therefore, we analyze the parameter in more detail and we found the problem.

Problem The problem is that the bandwidth parameter denormalized has a huge weight compared to the other parameters. This causes the model to be greatly influenced by the bandwidth parameter and ignore the other parameters.

Solution The solution is to increase the weight of the other parameters to stand out and thus the model realize their importance. Therefore, we test the model with others factors and after several tests, we got the following factors:

Parameter	Factor
peerID	200
votes	20
isMajority	10
coordID	1
decision	1

Table 20: The factors applied to each parameter to avoid these parameters to be ignored.

The resulting dataset with the new factors is present in Figure 21. The resulting factors are double of the previous factors. Increasing the values to double causes the model not to ignore the other parameters.

Input Parameters

Bandwidth	PeerID1	...	PeerIDN	CoordID1	...	CoordIDN	isMajority	Peer1Vote	...	PeerNVote	DecisionValue
100	200		0	0		0	0	20		20	0
100	0		0	0		0	10	20		20	1
500	0		0	0		0	0	20		20	0
100	0		0	0		0	0	0		0	0
100	0		0	0		0	10	20		20	1
500	0		0	0		0	0	20		20	0

Target Parameters

DelayToPeer1	...	DelayToPeerN
-1.00E-03		0.00
-1.00E-03		-1.00E-03
-1.00E-03		-1.00E-03
-1.00E-03		-1.00E-03
0.00		-1.00E-03
-1.00E-03		-1.00E-03

Table 21: Sample of merged dataset with the *bandwidth* denormalized.

Training - 3rd iteration

The ANN is the same as the previous models, which has 122 nodes in the input layer, 100 nodes in each hidden layer and 40 nodes in the output layer. Besides, the dataset used is the same as the previous, which has **34 MB**, and the ANN was configured with 150 *epochs* as well. Finally, the training process takes, approximately, **20 minutes**, because of the dimension of the dataset and the denormalization of the input parameters, which causes the increase of training time.

Results - 3rd iteration

In the end, we achieved a model that was able to learn both mutations in a single dataset, distinguishing them by the bandwidth parameter. The problem of the bandwidth to have a high weight made the model ignore all the other parameters and for that not following any mutation. After readjusting the weights of the parameters, the model learned again all the patterns. Therefore, when the simulator is configured with an environment that has a **bandwidth of 500**, the model uses the **centralized** mutation. On the other hand, if the simulator configures an environment with a **bandwidth of 100**, the model uses the **ring** mutation. This feature makes the protocol adaptable to the change of environment by automatically configuring itself, which makes it an **adaptive consensus protocol**.

Figure 33 shows the results got when exchanging the bandwidth in the simulated environment. The first environment has a bandwidth equal to 500 and the second equal to 100. Besides, interesting observations, is that the performance of both mutations is very good, in spite of this mutation to have a model that loads the protocol.

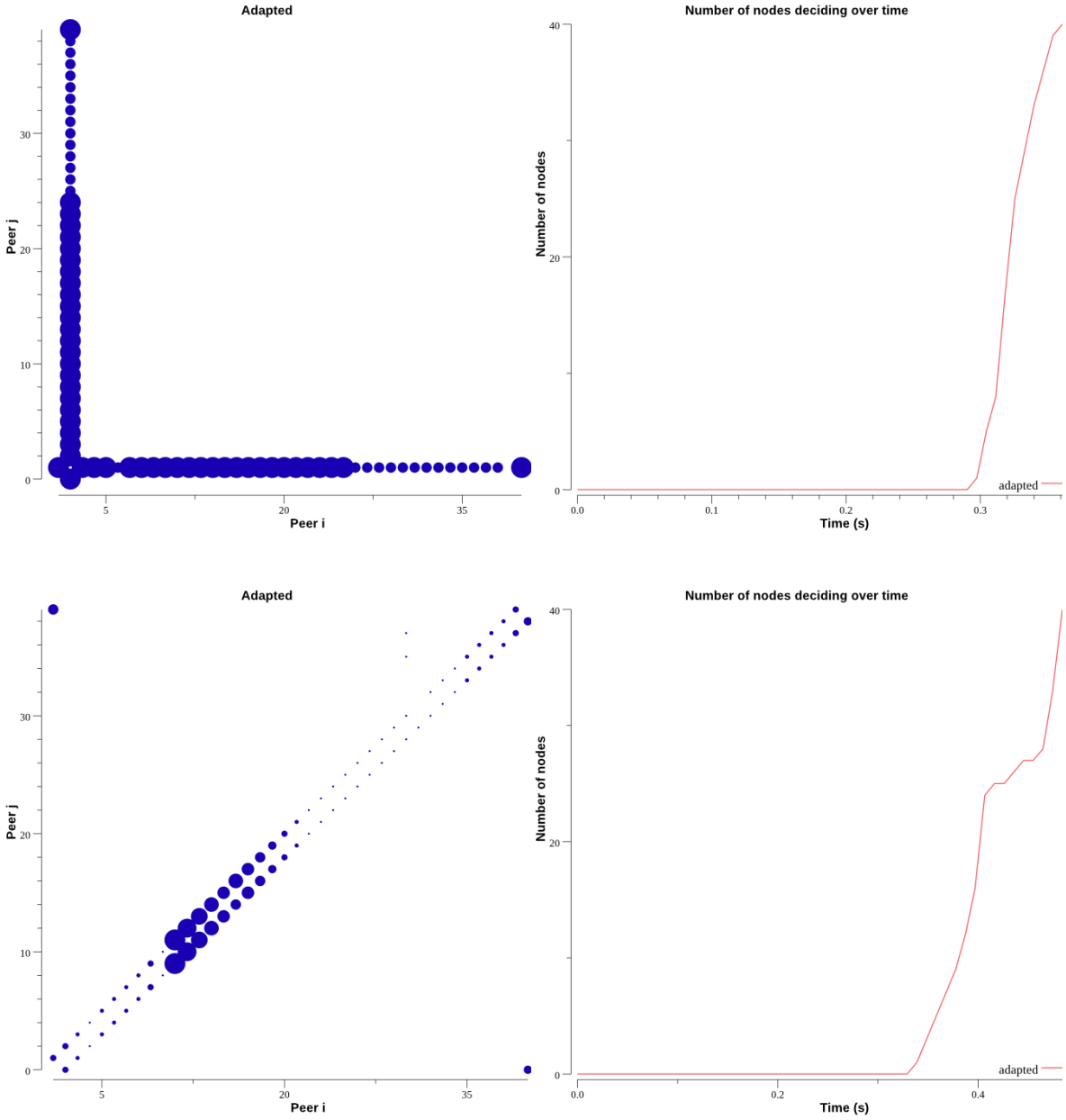


Figure 33: Message exchange pattern and respective time to reach consensus for *adaptive* mutation, alternating the bandwidth from 500 to 100 messages/second.

5

CONCLUSION AND FUTURE WORK

We propose a *Proof of concept (POC)* of a new consensus protocol, called **adaptive consensus protocol**, that can adjust to a given environment. It is implemented in *Go*, following a modular and extensible approach, focusing on possible integration with the HLF blockchain implementation. The protocol is based on the MCP which is able the majority of faults and offers a range of mutations which it can adopt: *ring*, *centralized*, *early*, and *gossip*. Besides, MCP presents a modular and extensible structure, which allow adding a new implementation of mutations. Thus, we reimplement the MCP protocol and integrate it into a simulator that allows us to test and analyze the protocol in several environments. The new protocol integrates the four mutations, with **an improvement in the ring mutation** to make it able to run in environments with faulty nodes. From this protocol, we developed an adaptive component to give the protocol the capacity to configure itself automatically through ML. For that, we collect metrics of the protocol from the simulator in each run to feed several ML models. The goal is to make these models learns the protocol characteristics or behaviors from the data collected. From this approach, several tests were performed, obtaining several models of ML and DL, standing out two: a model that based on a dataset with several examples of execution of different environments predict the best mutation for a given environment; a model with an ANN that based on the protocol parameters given by two mutations (*ring* and *centralized*) can replicate the learned mutations taking into account a differentiating environment parameter. However, the last model brings the most interesting result. Integrating the last model with the protocol in the simulator, it makes possible to-have a DL model to learn and decide to use different mutations depending on environment conditions, like the bandwidth parameter. Therefore we achieve a protocol that can adapt to a given environment, *i.e.*, an **adaptive consensus protocol**.

However, the development of this protocol has brought many challenges, which we highlight two of them. At first, the process to integrate the ANN with the simulator takes some time because of the problem to run all peers, each one with a ANN, in the same machine overload the machine. Therefore, was necessary to implement an extra component that does not concern the dissertation directly. The other limitation is that all the models imple-

mented only can be used to a specific number of peers. If we change the number of peers, it is necessary to build a new model. Therefore, we were focus in a fixed number of peers.

Finally, this dissertation proposes a new protocol to HLF and proves that it is possible to learn from the mutations through ML strategies.

5.1 FUTURE WORK

All models implemented are a simple approach to the problem, which offers the possibility of improvements in the models or even new solutions. For example, the ANN that learns the *ring* mutation and *centralized* mutation can be extended to learn the rest of the mutations. Additionally, the ANN used has a simple structure that can be improved by changing the hidden layers, and find a better structure for them. The problem of a fixed number of peers, in the ML models can be solved by fixing a big number of columns to aggregate more than one possibility of values.

Once the new protocol presents a modular and extensible structure, another interesting approach would be to create new mutations by using ML models. In this case, it would be interesting to build ML that can learn different patterns from different mutations and create a new mutation by combining the different patterns learned. With this model, it would be possible to run the protocol and have a different subset of peer running a different mutation, preferably the best mutation for the case, in the same protocol. However, it would be also interesting to consider other strategies other than ML.

After the development of the consensus protocol, it will be necessary to make some benchmarks and compare it with other solutions.

Another topic is to improve the level of resiliency of the protocol. The current protocol is a CFT protocol, which can reach consensus if only components fail. Therefore, another goal will be to improve the protocol resilience to malicious peers, making it a BFT protocol.

Finally, another future goal is to make the protocol a possible reliable solution as a consensus protocol in the HLF.

BIBLIOGRAPHY

[1] JW MICHAEL, ALAN COHN, and JARED R BUTCHER. Blockchain technology. *The Journal*, 2018.

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL https://bitcoin.org/bitcoin.pdf.

[3] Evelyn Cheng and Fred Imbert. Bitcoin tops record $19,000, then plunges in wild 2-day ride, 2017. URL https://www.cnbc.com/2017/12/06/bitcoin-tops-13000-surging-1000-in-less-than-24-hours.

[4] Market capitalization, January 2019. URL https://www.blockchain.com/en/charts/market-cap.

[5] Jonathan Liebenau and Silvia Elaluf-Calderwood. Blockchain innovation beyond bitcoin and banking. Available at SSRN 2749890. 2016.

[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5584-1. doi: 10.1145/3190508.3190538. URL http://doi.acm.org/10.1145/3190508.3190538.

[7] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. Bridging paxos and blockchain consensus. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1545–1552. IEEE, 2018.

[8] Minh Quang Nguyen, Dumitrel Loghin, and Tien Tuan Anh Dinh. Understanding the scalability of hyperledger fabric.

[9] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[10] Qassim Nasir, Ilham A. Qasse, Manar Abu Talib, and Ali Nassif. Performance analysis of hyperledger fabric platforms. pages 1–14, 2018.

[11] Joao Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–58. IEEE, 2018.

[12] José Pereira and Rui Oliveira. The mutable consensus protocol. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 218–227. IEEE, 2004.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[14] Francisco Maia, Miguel Matos, Jose Pereira, and Rui Oliveira. Worldwide consensus. volume 6723, pages 257–269, 06 2011. doi: 10.1007/978-3-642-21387-8_21.

[15] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1982.

[16] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.

[17] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[18] Christian Cachin and Marko Vukolić. Blockchains consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

[19] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[20] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *Big Data (BigData Congress), 2017 IEEE International Congress on*, pages 557–564. IEEE, 2017.

[21] Pavel Vasin. Blackcoin's proof-of-stake protocol v2. *URL: https://blackcoin. co/blackcoin-pos-protocolv2-whitepaper. pdf*, 2014.

[22] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[23] Rachid Guerraoui, Rui Oliveira, and André Schiper. Stubborn communication channels. Technical report, 1998.

[24] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[25] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[26] M Swarna, S Ravi, and M Anand. Leaky bucket algorithm for congestion control. *International Journal of Applied Engineering Research*, 11(5):3155–3159, 2016.

[27] Google. Colaboratory: Frequently asked questions. URL `https://research.google.com/colaboratory/faq.html`. Accessed: March 16, 2019. [Online].

[28] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436, 2015.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project: UID/EEA/50014/2019.