NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RICARDO FILIPE MENDES LOUREIRO

Bachelor in Computer Science and Engineering

REASONING ABOUT CONSENSUS PROTOCOLS:

SIMULATION AND VALIDATION ENVIRONMENT

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Jully, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

REASONING ABOUT CONSENSUS PROTOCOLS:

SIMULATION AND VALIDATION ENVIRONMENT

RICARDO FILIPE MENDES LOUREIRO

Bachelor in Computer Science and Engineering

Adviser: António Ravara
Associate Professor, NOVA University Lisbon

Co-adviser: Simão Melo de Sousa
Associate Professor, University of Beira Interior

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Jully, 2023

Abstract

The number of services and applications that require and rely on transactional, replicated and verifiable data to function is increasing with each passing day, from banking and financial applications to online voting. With these requirements also come challenges, like availability, consistency, and security mechanisms that allow for integrity, non-repudiation and encryption of messages.

A common solution that these applications use to satisfy these requirements are blockchain protocols, usually defined as distributed ledgers with a growing list of records (blocks), linked together by cryptographic hashes. Records are permanent and usually distributed across a peer-to-peer computer network where participants adhere to the consensus protocols to validate and add new transactions.

There are a wide range of different blockchain protocols and some of them are not set in stone, Like Ethereum which moved from proof of work into a proof of stake solution or Tezos where stakeholders are capable of proposing and agreeing on changes to the consensus protocol allowing it to evolve over time. Because of this a number of tools focused on assisting with the development of these protocols have emerged.

One of these tools is MOBS `https://github.com/mce-alves/MOBS`, Modular Blockchain Simulator, a simulator built with extensibility and modularity in mind, allowing users to simulate any family of protocols as well as parametrize multiple scenarios for their study. These parameterizations include bandwidth limits, byzantine behaviour of the participant nodes and adversarial behaviour. After the execution the desired statistics and information needed to validate the execution of the protocols can also be parametrized.

The goal of this dissertation is twofold we propose the implementation of tools that will allow us to validate the correctness of the implemented consensus protocols through the logs of the simulator, allowing for better qualitative results to be extracted and evaluated. Secondly also propose an extension to this tool to better allow the simulator to provide different sets of execution environments by allowing parameterization of the network layer of the simulator we propose Chord, HyParView and X-BOT since they are diverse protocols and very different from each other.

Keywords: Blockchain, Networking, Consensus, Simulation, Validation of protocol properties

Contents

List of Figures

List of Tables

1 Introduction

This thesis aims to address the practical challenges in designing, implementing and maintaining blockchain and consensus protocols by providing a simulation environment to analyse their behaviour and provide empirical metrics of their runtime in different environments and conditions. To achieve this we intend to take MOBs and expand the previously done work to help better test and validate these protocols by providing a wider range of networks, a more dynamic and independent network layer and improve the logging of the application to extract data to validate the protocols' properties.

1.1 Context

With each passing day the amount of distributed applications increases, and a subset of these are applications deal with data that requires validation, transactional operations by validated users and simultaneous access for updating and consulting the records. For this specific subset there are a group of protocols that have been created to meet and ensure these requirements. Blockchain protocols allow for simultaneous access, validation and update of records across a distributed database, each node has its own copy of the ledger that it uses to validate information and reach a consensus about its accuracy.

Around 2008 blockchain protocols appeared with the motivation to serve as a distributed ledger for cryptocurrency transactions. These protocols are not static, being because vulnerabilities or flaws that need to be correct or due to the very nature of the protocol itself. One of these dynamic protocols is Tezos [2], which relies on the stakeholders that participate in the system to propose and agree on changes and upgrades to the protocol. Another example is Ethereum [4] that started by using a blockchain protocol based on proof of work and in 2022 migrated towards an implementation based solely on proof of stake for Ethereum2. This opens a necessity for tools that aid in support these evolutions in a faster, more seamless and secure fashion.

Blockchain protocols operate on top of membership layers that dictate how the topology of the network is configured. Different membership environments come with different properties and trade-offs. Structured membership overlays allow for faster lookups for

specific nodes and a pre-defined and predictable structure to the network. Non-structured membership offer a more resilient overlay when new nodes are introduced or existing ones leave, albeit by choice or failure. And overlays that operate by building a partially structured overlays allow for the benefits of a non-structured overlay at the cost of slower re-structuration since optimization procedures are regularly executed to improve routing and lookup operations. The trade-offs and some of these protocols will be further explained in Chapter 2.

The challenges in developing blockchain protocols motivated the development of MOBs, a modular and extensible simulator that provides the ability to simulate different families of blockchain and consensus protocols. MOBs provides a parametrizable execution of the selected protocols, exhibiting a modular and extensible structure and offers detailed logs for the qualitative evaluation for the study of implemented protocols. However, this simulator can be further improved by incorporating conditions closer to real life, such as a dynamic and parametrizable network layer and more detailed logs with specific configurations for consensus protocols. These enhancements would enable the study and validation of the behaviour of these protocols in various scenarios, abstracted from the intricacies of the arrangement of the participants and the validation of the correctness of a wider array or protocols.

1.2 Problem

Consensus protocols are not trivial to define or understand correctly and in blockchain systems, where the behaviour can be dynamic and mostly financial transactions are dealt with, their correctness is crucial and errors can be costly. One example of this is Ethereum moving from Proof of Work to a Proof of Stake consensus, this opened the protocol to vulnerabilities to bouncing attacks on liveness [15]. There is also Solana [21], a new blockchain protocol that relies on Proof-of-History to build his chain, where after repeated testing results showed that the protocol does not fully achieve consensus and a single malicious validator can halt the Solana blockchain [17]. These tests also showed that there are inconsistencies in the behaviour between what is described in the documentation and what the protocol showed since Solana's implementation has deviated in undocumented ways from the available protocol design descriptions.

To help with this MOBs was developed giving the ability to test and validate these protocols under different conditions and settings by changing the execution parameters, aiming to catch these vulnerabilities or even logic errors before deploying changes to these protocols.

Right now the parameters to the network are set before the execution of the simulation, and regarding the network behaviour we can set the network topology, how many nodes will fail and when. In the real world a network's topology is dynamic, new nodes can enter as new participants, existing ones can leave, either by choice or by failure, and even

when the participants are static the network can suffer changes to its topology as a result of optimizations performed by this layer.

The simulation of how this layer behaves and the parametrization of the different protocols that can be used as its basis are the problems that we aim to address with this thesis.

1.3 Goal

With the different properties that different network overlays provide besides the parametrizations that are already provided by MOBs, we can further provide an even more modular simulator that allows for the study of the behaviour in different membership protocols. This will allow us to leverage the modularity of this layer to better simulate the behaviour of new participants coming into the system, existing ones leaving and how changes to the network topology of the network affects their execution.

Another aspect we want to improve is the logging module of MOBs, by providing a template for logging of consensus protocols and an after execution analysis tool to verify their correction, validate their execution and extract metrics for comparison between executions in different environments.

The goal of this thesis is to improve the networking layer of MOBs by making it more modular and therefore giving the following advantages:

- Different environments for more diverse testing scenarios.
- Stronger parametrization regarding the behaviour of the network.
- Better qualitative data by providing scenarios that better mimic real world execution.

1.4 Document Organization

The remainder of this document is organized in the following manner:

Chapter 2 studies related work: in particular we will cover several membership protocols and some implementations; Some simple consensus protocols; Overview of the types of blockchain consensus;

Chapter 3 we show Babel which is a framework for implemented distributed system protocols; And NetworkX, a package in Python that provides network analysis tools.

Chapter 4 studies the MOBs implementation.

Chapter 5 we will talk about the proposed work to address the problem introduced in this chapter.

2

Background

In this chapter we will discuss relevant work considering the goals of the work to be conducted in this thesis. In particular, we will focus on the following topics:

In section 2.1 we discuss the differences between the different kinds of membership systems as well as some implementations.

In section 2.2 we will discuss some consensus protocols.

In section 2.3 we will discuss some different types of blockchain protocols.

2.1 Membership Protocols

This section introduces some membership protocols that have been considered to part of the work that will be developed.

2.1.1 Membership

A protocol where each node knows every other node in the network might work well for small networks, but it is not a scalable solution since each node would need to have *n - 1* communications channels open at all time, being n the amount of nodes in the system, and each node needs to be following any and all changes to the system. This is not feasible since the number of links between nodes would rise quadratically and networks can easily reach thousands of participants. In order to overcome the challenges that arise from large networks, we usually use partial view membership protocols. In these protocols each node only knows and maintains information about a small selection of nodes in the systems, making it a more scalable strategy than a total view protocol, since the number of connections tend to grow at a logarithmic rate instead.

When maintaining a partial view, protocols usually follow one of two strategies when managing their memberships:

- **Reactive:** Using this strategy, the partial view undergoes alteration solely when there are changes in membership, typically occurring when a node joins or leaves the membership.

- **Cyclic:** With this approach, the partial view changes periodically. Every t seconds nodes exchange information that may lead to modifications to the partial view.

This membership protocols are usually represented by a graph where the vertices represent the participants of the network and the edges represent the communication links. Depending on how the graph ends up we can classify the membership as structured, unstructured or partially structured. In the next sections we will go over these types of memberships and some implementations.

2.1.2 Structured Overlays

Structured overlay memberships follow a pre-defined structure by having the nodes routed to a specific logical position in the network. This known structure allows improvement on search primitives, enabling efficient discovery of data and process. The node's position is usually based on a unique identifier for each node of the network and different protocols organize the nodes by their identifier by

However, having a predefined topology come at the cost of a more costly re-structuration of the network every time a there is a change to the membership. This process is usually slow and costly since a change of one node may affect the network at a global level. This drawback becomes more relevant in high churn rate scenarios.

2.1.2.1 Chord

Consistent Hashing and Random Trees, or Chord [18] is a Distributed Hash Table based algorithm where each node is assigned a unique identifier based on the output of a hash function. Usually this has is based on the IP address of the node since this is unique for every node of the network, and making it so that once a node joins the network their identifier is set in stone. These identifiers are also so that each node knows which keys of the distributed hash table are assigned to them, since the range of keys that belong to a certain node is represented by the range defined by their identifier up to the identifier of the next node.

The graph generated based on the membership overlay will be a cyclic graph with a ring shape, with the nodes ordered by their generated identifier. Each node also maintains a routing table with around O(log n) distinct entries called a finger table. This routing table is used in order to navigate the overlay in a more efficient manner.

The tables maintained by these nodes are automatically updated when a new node joins or leaves the system making it always possible to find a node responsible to a given key. However, simultaneous failures may break the overlay since the correctness of the membership depends on each node knowing their correct neighbours. In order to increase fault tolerance, a periodical stabilization protocol is run in background, making sure the neighbours are still active and restructuring the overlay accordingly as well as updating the entries on the finger table.

2.1.2.2 Pastry

Pastry [16], similarly to Chord is a structured overlay protocol that defines a distributed hash table. A Pastry system is a self-organizing overlay network of nodes where every node has a 128-bit identifier. This identifier is assigned randomly when a node joins the system and is used to indicate a node's position in a circular space that ranges from 0 to $2^{128} - 1$. It is assumed that the hash function that gives the node's identifier generated a uniform distribution of identifiers around the 128-bit space.

Each Pastry node maintains a *routing table*, a *neighbourhood set* and a *leafing set*. The routing table contains the entries based on substrings of the own node's identifier. The neighbourhood set contains the node identifiers and the corresponding IP address of the M the closest nodes to the local node, M being parametrizable. The neighbourhood set contains the nodes that are used to maintain local properties. The leafing set contains the $L/2$ closest large node identifiers and $L/2$ smallest node identifiers, L also being a parametrized value. The leaf set is used in message routing.

Pastry makes use these tables in the following fashion: when a message is received for another node in the network we check the leaf set, if the node is in range of the leaf set we route directly to the destination node, otherwise we check the routing table for a node that shares a common prefix with the key.

2.1.3 Unstructured Overlays

Unstructured overlays place few constrains as on how the nodes chose their neighbours. This results in a random graph that is hard to predict and describe.

By not having a structured overlay these memberships end up being more fault-tolerant in high churn rate scenarios due to the cost of the network restructuring itself is fairly low.

2.1.3.1 SWIM

SWIM [7] stands for **S**calable **W**eakly-consistent **I**nfection-style Process Group **M**embership Protocol. This protocol is composed of two distinct components, a failure detector and a dissemination component.

Unlike traditional gossip based overlays that rely on a heartbeat strategy, the failure detector in SWIM is independent of the rest of the protocol. The failure detector is fully decentralized and is executed in a randomized probe-based fashion, the authors later suggest an optimization via a round-robin fashion instead. Every t seconds, parametrizable value, a random node that belongs to the membership is checked to see if it's still alive, if no awnser is received the request is forwarded to k other members to execute an indirect probe. If the nodes responds to at least one of the indirect requests the suspicion is removed and an *alive* message is spread to fasten the process in case other nodes suspect from it. This mechanism reduces The number of false positives in failure detection, which can usually be due to a temporary unavailability from a network partition or from a

slow-down of a node due to a high load of requests. As a drawback failure detection is slower.

The other component of this protocol is the gossip dissemination system which maintains a partial view of the network. This component updates whenever a member joins or leaves the system by an infection style dissemination protocol. In order to make this more efficient, the updates piggyback the messages that are sent during the failure detection procedure.

2.1.3.2 HyParView

HyParView [13] is a gossip based membership protocol that offers high resilience and high delivery reliability of messages while being highly scalable. It relies on a hybrid approach by maintaining, besides the active view that is used by the nodes to maintain the communication overlay, a passive view, in case the networks needs to be restructured.

HyParView uses different approaches when it comes to maintaining each of the views, for active view a reactive strategy is used, nodes react to events that require the network to be restructured such as new nodes joining the membership or existing ones leaving, either by failing or by choice.

The nodes in the active view are the ones with which each node maintains a communication link. In case the active view needs to be changed the nodes in the passive view may be promoted to active nodes the same way nodes in the active view may be demoted to the passive view. All the nodes in the active view of a given node have that node in their active view, making the connection graph that represents the overlay be a bidirectional graph.

For the passive view, HyParView uses a cyclic strategy by periodically exchanging information with other nodes regarding the contents of its views. This information is exchanged by means of a shuffle operation with another node from its active view. After the exchange, the nodes integrate the new received members in its passive views, even if for that, some older members have to be dropped. This mechanism helps to maintain a homogeneous overlay in the number of connections each node has.

The gossip strategy relies on the use of TCP as a reliable transport protocol and fail detector. Since all members are tested at each step of the gossip protocol, failed nodes do not stay unnoticed for long. This way, HyParView provides with fast self-healing properties that are characteristic of gossip protocols.

Tests done in [13] show that the algorithm is able to recover from as much as 80% node failure, as long as the overlay stays connected. By being able quickly react to failures in the system, the protocol was shown to be able to maintain 100% reliability for message dissemination.

2.1.4 Partially Structured Overlays

Partially structured overlays aim to get the best of both strategies. We can leverage the easy to maintain and fault-tolerant unstructured overlays and by applying some optimization procedure to the network we can achieve a more efficient search and application level routing.

2.1.4.1 T-MAN

T-MAN [11] was created with the motivation to give the ability of taking some random overlay, and *evolve* it into another one.

The logic behind the algorithm is giving each node a ranking value that every node can use to apply a function to determine how desirable a node is as a neighbour. Each node maintains a partial view that contains the addresses of nodes that are not its immediate neighbours, much like the HyParView overlay described in 2.1.3.2. Periodically each node exchanges its partial view with the first node in its active view, according to the ranking values which depends on the target overlay. The receiver will execute the same procedure as the sender, so they can later merge their local views and apply the ranking function. Using their peers' views to improve their own the overlay will gradually become closer the desired overlay.

Experimentally this algorithm is shown to be scalable and fast, with the convergence times growing approximately at a logarithmic rate in function of the number of nodes in the network. The problem that surges with this, is that the network only becomes as fault-tolerant as the desired overlay, since T-MAN doesn't aim to maintain a balanced degree between the nodes, this might create uneven load balance or even node isolation during or after the procedure

2.1.4.2 X-BOT

X-BOT [14] stands for **B**ias the **O**verlay **T**opology according to some targeting criteria **X**. This protocol is completely decentralized, and the nodes do not require any prior knowledge of where they will end up in the final topology. This protocol strives to preserve the degree of the nodes that participate in the 4-node coordinated optimization technique, described in greater detail below, this is essential to preserve the connectivity of the overlay. X-BOT is built in a way that every modification that is done by the protocol increases its efficiency and due to the dynamic nature of the model, its ensured that the overlay does not stabilize in a local minimum. These optimizations are done in a way that key features of the overlay, such as low clustering coefficient and low overlay diameter, are preserved. The protocol is highly flexible because it relies on a companion oracle to estimate the link cost and therefore bias the network according to different cost metrics.

The companion oracle is accessible by all nodes, and its sole purpose is to give the link cost from the node that invokes it to a given node.

The 4-node coordinated technique that has been referred above works as follows, a node i, the initiator, starts the optimization round selecting a node from its passive view. Node O is a node from i's active view that will be replaced. Node c, the candidate, is a node from i's passive view that is going to be upgraded to its active view. And finally node d is the node to be removed from the candidate's view so that i can be accepted. These nodes are always selected based upon the link cost values provided by the oracle, ensuring that every time the optimization procedure is called the network increases its efficiency

2.2 Consensus Protocols

Consensus protocols solve the problem of in a system of participants reaching an agreement on a given value. These algorithms must ensure an agreement is reached even in the presence of faulty nodes in the network. Consensus protocols must ensure they satisfy the following properties [6]:

- **Termination:** Eventually, every correct process decides some value.
- **Validity:** If all the correct processes propose the same value, then any correct process must agree on that value.
- **Integrity:** No process decides more than one value
- **Agreement:** Every correct process must agree on the same value.

We will now look at some protocols that solve consensus.

2.2.1 Chandra-Toueg

The Chandra-Toueg consensus protocol [5] for solving consensus in a network of unreliable processes equipped with an eventually strong failure detector. This failure detector is an abstract version of timeout signalling that each process preforms when other processes may have crashed. An eventually strong failure detector is defined with the two following properties:

- Eventually every process that crashes is permanently suspected by every correct process.
- There is a time after which correct processes are not suspected by analysis correct process, meaning that if they become suspects a one point but do not crash, the other correct processes will stop suspecting them eventually.

The algorithm assumes that the number of faulty processes is always less than half of the number of total nodes in the system. It operates in rounds using a rotating coordinator and each round the logic is as follows:

1. All processes send their preference to the coordinator.
2. The coordinator waits to receive messages from at least half of the processes, including itself and chooses the preference with most recent timestamp.
3. The coordinator sends the chosen preference to all nodes.
4. Each process waits to receive the chosen preference from the coordinator or wait for its failure detector to identify the coordinator as crashed.
    - In the first case it sets its preference to be the same as the coordinator and replies with an ack (acknowledge).
    - In the second case it responds with a *nack* (not acknowledge).
5. The coordinator waits to receive *acks* from a majority of processes, after it sends a decide message to all processes.
6. Any processes that received the decide message for the first time it broadcasts it to all processes.
7. The processes decide on the coordinator chosen preference and terminates.

2.2.2 Paxos

Paxos [12] is a family of protocols for solving consensus, for this example we will take a look at what is commonly referred as basic Paxos, that decides on a single value, and after take a look at MultiPaxos which gives a constant stream of agreed values. For this algorithm we make the following assumptions regarding processors:

- Operate at arbitrary seed.
- May experience failures.
- Have stable storage and may re-join the protocol after failures.
- Byzantine failures do not occur.
- The maximum number of failing processors is less than half of the total processors.

And the following assumptions regarding the network:

- Processors can send messages to any other processor.
- Messages are asynchronous and take an arbitrary time to deliver.
- Messages may be lost, re-ordered or duplicate.
- Messages when delivered are delivered without corruption.

Each participant can act as a Proposer, an Acceptor and a Learner. Each execution of basic Paxos decides on a single value and operates in two phases, each with two secondary phases.

- **Phase 1**

    1. **Prepare:** A Proposer creates a message which we call Prepare identified with a number n, this works as an identifier and has to be greater than any previous number used in previous Prepare messages. The Prepare message does not contain the proposed value. This message is sent to a quorum of acceptors, and a proposer shouldn't initiate Paxos if it cannot communicate with a quorum of acceptors.
    2. **Promise:** Acceptors wait for a Prepare message from any of the proposers, if they receive a message, depending on the value of n two flows can happen:
        a) If n is higher than every previous proposal received from any Proposer the Acceptor returns a Promise message and ignores all future proposals with a value less than n, if a proposal was accepted at some point in the past, the message includes the previous n value and the corresponding accepted value.
        b) Otherwise, the Acceptor can ignore the Prepare message or sending a not acknowledge to tell the Proposer to stop its attempt to create the Proposal.

- **Phase 2**

    1. **Accept:** If the Proposer receives Promises from a Quorum of Acceptors, it sets a value for its proposal. If any Acceptors that accepted a previous proposal they would have sent the previous accepted value, the Proposer will agree on the return value sent by the Acceptors with the highest n. In none of the acceptors had previously accepted a value then the Proposer chooses the value it initially wanted to propose. The proposer sends an Accept message with the chosen value and the n value.
    2. **Accept:** If an Acceptor receives and Accept message from a Proposer, it must accept if it hasn't proposed previously Proposals with an identifier grater than n. If the value is accepted, an Accepted message is sent to every proposer and Learner. Learners will only learn the decided value after receiving Accepted messages from a majority of Acceptors.

MultiPaxos is another algorithm in the Paxos family and is used when a continuous stream of agreed values is needed. If the leader is relatively stable phase 1 becomes unnecessary. To achieve this we include the round number along with each value which is incremented in each round by the same leader. We still need the phase 1 for the first round but in consequent rounds if the Leader does not change nor fails we can skip it, reducing the overhead for each round

2.3 Blockchain Protocols

Blockchain is the core technology of many cryptocurrencies as a distributed ledger technology. This technology combines cryptography, distributed system technology, peer-to-peer networking and many more. Different blockchain protocols use different consensus protocols to try and balance the trade-offs of each, and make sure we use the suitable consensus protocol for the blockchain type used by the system. The adopted consensus protocol needs to fit the demands of different application scenarios. In this section we will detail some popular blockchain consensus protocols that effectively address Byzantine behaviour in the system, taking basis in the work done in [22]:

PoW (Proof of Work): PoW is adopted by Bitcoin and Ethereum, this consensus protocol selects one node to create a new block each round of consensus by using a computational power competition. In this competition node nodes need to solve a cryptographic puzzle, the first node to solve this puzzle has the right to create a new block. Nodes need to constantly adjust the value of nonce in order to get the correct answer, which requires an enormous amount of computational power. PoW guarantees eventual consistency.

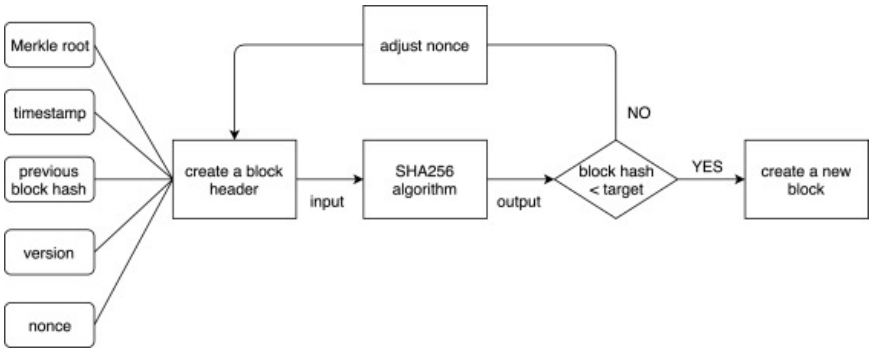

Figure 2.1: Flow of PoW

PoS (Proof of Stake): In PoS, selecting which node creates a new block depends on the held take rather than the computational power. Node till need to solve a SHA256 puzzle, but the nonce does not need to be adjusted, instead the key to solve the puzzle is the amount of stake. Nxt and Ouroboros adopt PoS and Ethereum plans to transition from PoW to PoS. This consensus protocol guarantees like PoW eventual consistency.

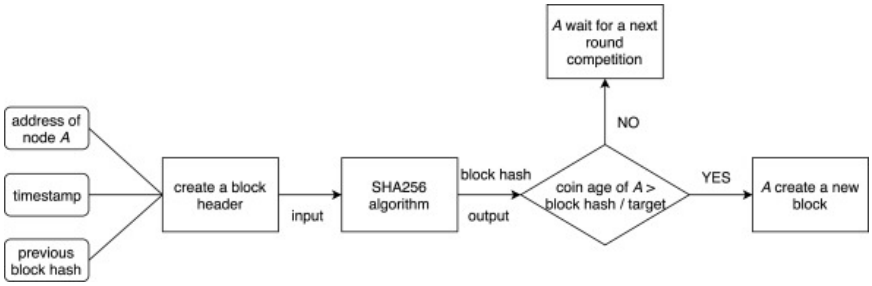

Figure 2.2: Flow of PoS

DPoS (Delegated Proof of Stake): The principle here is to let nodes who hold stake to

vote to elect block creators, this way delegating the job of creating the blocks and reducing computational consumption. This is adopted by BitShares and EOS.

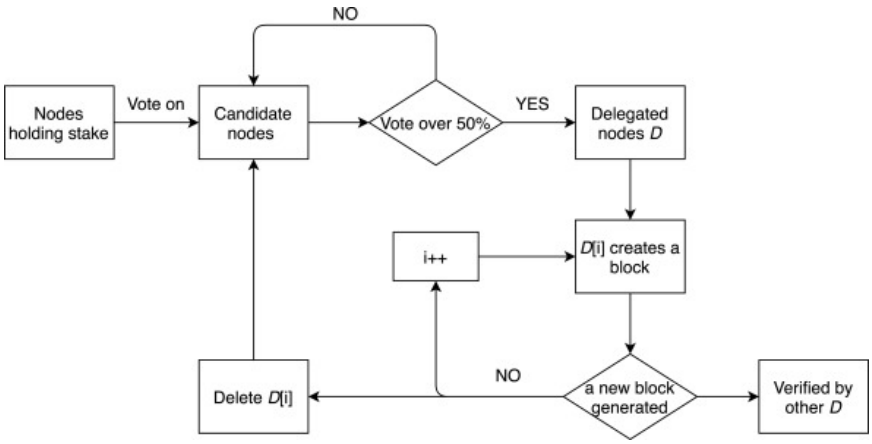

Figure 2.3: Flow of DPoS

PBFT (Practical Byzantine Fault Tolerance): PBFT is a Byzantine Fault Tolerance protocol with low algorithmic complexity, it contains five phases: request, pre-prepare, prepare, commit and reply. PBFT guarantees nodes maintain a common state and take a consistent action in each round, guarantees strong consistency and is adopted by EOS to validate blocks while using DPoS for creation.

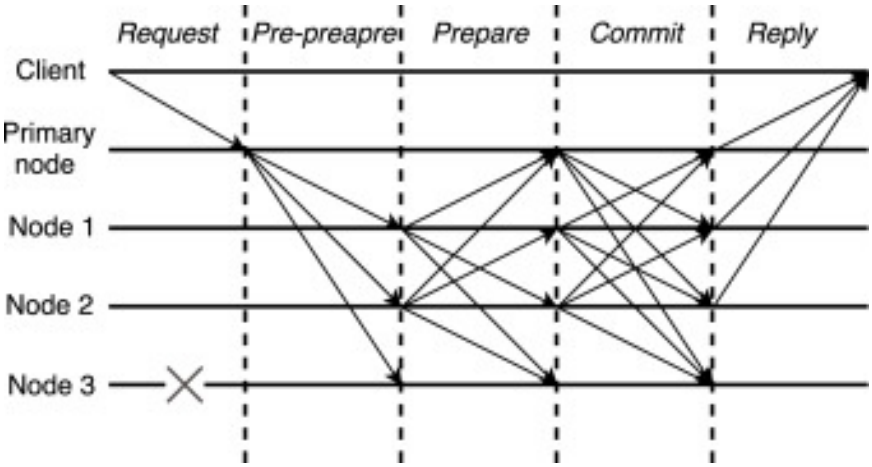

Figure 2.4: Flow of PBFT

3

Related work

In this chapter we will talk about work that has been developed and that we will use either as a comparison to what we want to develop or as a part of the solution that we are proposing.

In section 3.1 we will present Babel, a framework that simplifies the development of distributed protocols.

In section 3.2 we will discuss NetworkX, a Python library for graph analysis.

In section 3.3 we will discuss some blockchain simulators of note that have been developed.

3.1 Babel

In this section we will discuss Babel [9], a framework that simplifies the development of distributed protocols within a process that executes in real hardware. Babel simplifies the development process by abstracting the developer from the low level complexities associated with typical distributed systems implementations like interactions with other protocols, communication details, handling timers and concurrency control. The communication complexities are hidden behind abstractions called *channels* that can be extended by the developer. The Babel framework is composed by three main components:

- **Protocols:** This is the component the developer implements encoding the behaviour of the distributed system being designed. Protocols are modelled as state machines whose state changes by action of external events, for this purpose each protocol contains an event queue that keeps track of these events.

- **Core:** This is the central component that coordinates the execution of all protocols and mediates all interactions within the framework. Core also keeps track of timers setup by the protocols and delivers an event to the correct protocol when the timer is up.

- **Network:** This component handle the channel abstraction that was mentioned previously. Protocols can interact with channels by opening and closing connections and sending messages as well as receiving relevant events for the protocol.

Babel is provided as a Java library and the developer can implement a protocol by extending the abstract class *GenericProtocol* that provides the needed methods to generate events, register callbacks to process received events and the initial behaviour for a node for the protocol that is being implemented. The provided API can be split into three categories:

- **Timers:** This allows for the execution of periodic actions.
- **Inter-protocol communication:** Babel supports multiple protocols executing concurrently in the same process so mechanisms for these protocols to interact with each other exist. These mechanisms take the form of requests and notifications.
- **Networking:** Babel offers different network channels with different capabilities but keeping the same model of interactions between protocols and the different channel implementations.

Evaluation testing in [9] show that Babel can reduce the effort for programmers in creating prototypes of distributed protocols, comparing the number of lines programmed using Babel and standalone versions of the same protocols in Java. This is achieved mostly by the logic that is covered in the network and core layers, relieving the developer from having to worry about handling timers, message ordering and most communication besides events.

Regarding performance tests conducted by implementing MultiPaxos [12] with the framework when comparing to standalone implementations, the implementation using Babel showed comparable performance with less implementation overhead.

3.2 NetworkX

NetworkX [10] is a network analysis Python package that allows for the analysis of networks and network algorithms. This package provides data structures for representing several types of graphs such as simple graphs, directed graphs and graphs with parallel edges and self-loops. In addition to the basic data structures, there are also graph algorithms implemented for calculation network properties and structure measures such as node connectivity, clustering coefficient, average shortest path.

For NetworkX nodes can be any time of hashable object such as strings, numbers, files or even functions. Edges or links are represented as a tuple of nodes. The standard ways of representing graphs are edge lists, adjacency matrices and adjacency lists. Functions to manipulate graphs are also provided, simple functions come as class methods, like

adding or removing nodes or edges and complex statistics and algorithms are provided as package functions such as clustering, the shortest path and visualization.

Another of the implemented algorithms NetworkX provides is the ability to draw a given graph, will be a helpful visual aid to understand the layout of the network that a protocol generates.

3.3 Simulators

3.3.1 VIBES

Vibes [19] is a message driven blockchain simulator developed with the goal of providing configurable, scalable and fast network simulations. It also provides a GUI for visualization of some extracted metrics and allows users to view a time-lapse of the processed events.

Vibes is developed in Scala and as such inherits the Actor language paradigm, the actors in Vibes can have one of three roles:

- **Node:** Follows the protocol to replicate the behaviour of the blockchain network.
- **Coordinator:** This actor acts as an application-level schedule. It receives requests from all nodes to fast-forward the network to the point in time when each node has completed his current task and once it receives a request from all nodes it moves the entire network to the earliest timestamp, guaranteeing a correct execution order of all tasks.
- **Reducer:** Once the simulation ends the reducer gathers the state of the network and produces the simulation results to be processed by the user.

3.3.2 BlockSim: Blockchain Simulator

BlockSim [8] is a simaulation framework that assists in the design and Evaluation of blockchain protocols. Developed in Python it uses a probablistic distributions model that can be specified by the user to model rnadom phenomena such as time taken to validate a block and network latency.

BlockSim's architecture, as shown in the figure, is made up by the following components:

- **DESE:** Discrete Event Simulation Engine, based on SimPy

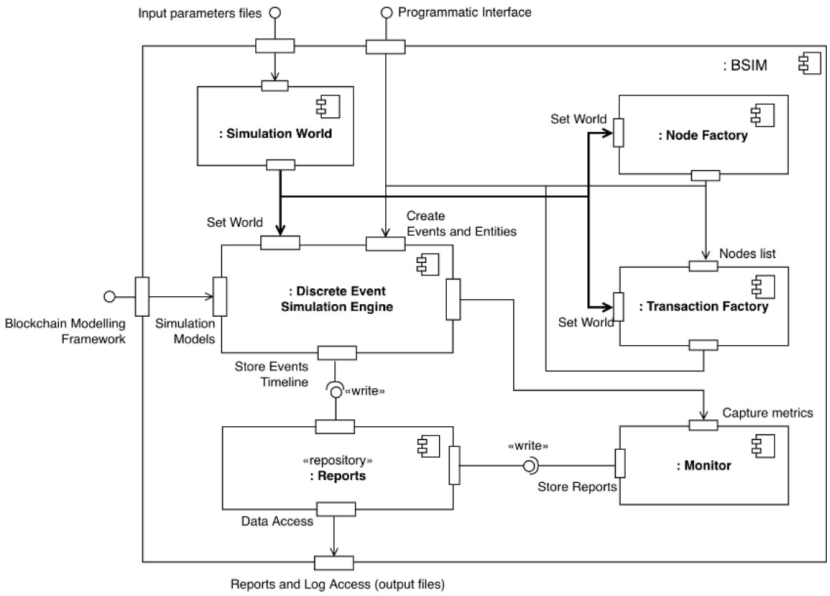

Figure 3.1: BlockSim Architecture

- **Simulation World:** Responsible for handling input/configuration parameters of the simulations.
- **Transaction and Node Factory:** Responsible for creating batches of transactions modelled as random phenomena. Node Factory creates nodes that are used during the simulation
- **Programatic interface and Simulation Example:** Main interface available to the user
- **Monitor and reports:** Monitor captures metrics during the simulation, i.e. number of transactions broadcasted or received, transactions added to queue. These metrics are stored in the reports component
- **Blockchain modelling Framework:** Has several layers like the Node layer, the Consensus, the Ledger, Transaction and block, Network and Cryptgraphic.

3.3.3 BlockSim: Extensible simulation tool for blockchain systems

Although with a similar name of the previously presented simulator, BlockSim is a different framework designed to build and simulate discrete-event dynamic systems models for blockchain systems.

BlockSim has three main modules:

- **Simulation:** Is in charge of setting up the scheduling of events and compute the simulation statistics.
- **Base:** Consists of the layers for the implementation of the blockchain protocol that will be simulated

- **Configuration:** Is the main user interface where the simulation can be selected and parameterized.

The base module has three layers that can be extended by the user:

- **Network:** represents blockchain nodes and the underlying peer-to-peer protocol to exchange data.
- **Consensus:** encompasses algorithms and rules adopted to reach an agreement about the current state of the blockchain ledger.
- **Incentives:** contains the economic incentive mechanisms adopted by a blockchain to issue and distribute rewards among participating nodes.

3.3.4 SimBlock:A Blockchain Network Simulator

SimBlock was developed in Java to preform experiments on blockchain protocols with a large amount of nodes. It allows users to easily change the behaviour of nodes and sutdy their overall impact on the network. It also provides users with a simple GUI for users to load the output file of a simulation and observe evolution of the simulated protocol. Simblock is divided into three components:

- **Network:** Creates the network topology which is configurable in the number of nodes, number of neighbours for each node and latency for each node to its neighbours.
- **Node:** Defines the behaviour of each node in accordance to the simulated protocol.
- **Node:** Defines the rules for creating and validating blocks.

3.3.5 JABS

JABS [20] was developed in JAVA and is aimed at researching large-scale blockchain consensus algorithms, with a main focus on simulating consensus, network, and ledger-data layers. The simulator is designed to be modular and extensible, optimized for performance and scalability.

JABS is developed to be used as a discrete-event simulation tool for benchmarking, evaluating, adjusting, and comparing consensus algorithms, especially for global-size public blockchains

JABS is composed of five main components:

- **Scenario:** Serves as a template for designing and adjusting simulation parameters.
- **Network:** Describes the connections between nodes and their bandwidth.

- **Simulator:** Responsible for processing each node's events in the correct order.
- **Node:** Is where the logic for the protocol is implemented and defines each node's behaviour.
- **Logger:** Handles all the outputs in the simulation into either a standard output or a CSV file.

3.3.6 Critical Analysis

	Adversarial Behavior	Offline Nodes	Bandwidth Limits	Network Topology	Proof of Stake	Proof of Work	GUI
VIBES [19]	yes	not modeled	not modeled	generated via parameters	not modeled	bitcoin	yes
BlockSim [8]	not modeled	not modeled	Throughput calculated from distributionn	fully linked	not modeled	bitcoin ethereum	no
BlockSim [1]	not modeled	not modeled	not modeled	fully linked	not modeled	bitcoin ethereum	no
SimBlock [3]	not modeled	not modeled	specify expected available bandwidth	generated via params	simple example	bitcoin dogecoin litecoin	yes
JABS [20]	not modeled	not modeled	configurable	generated via network layer	simple example	bitcoin DAGsper	no
MOBs	yes	yes	parameterizable	generated via params	tenderbake	bitcoin algorand ouroboros	yes

Table 3.1: Feature comparison between existing blockchain simulators and MOBS.

In Vibes [19] there is a lack of separation between the code that defines the simulator and the codes that defines the protocols, which makes implementing new protocols difficult and time costly. On the GUI provided this lack of separation also exists, having the statistics that are displayed tied to the protocols being simulated, which means that if the user wants to implement a new protocol that has different metrics/statistics, changes would need to be done to the GUI to support them.

Neither BlockSim [8], BlockSim [1], SimBlock [3] nor JABS support adversarial or Byzantine behaviour of the nodes, making it impossible to test the protocols when the network is not in perfect conditions.

VIBEs, BlockSim and BlockSim do not model Proof of Stake protocols which hinders their extensibility, since as a result, these simulators don't offer abstractions for timers and alarms commonly used in proof of stake.

JABS designed to implement simple blockchain protocols, and is not ready to implement pure consensus protocols. Furthermore, it lacks the means to leverage the already implemented logic in other protocols to be re-used in new implementations, making the development of new protocols costly and time-consuming.

4

MOBs

4.1 Overview

MOBs standing for Modular Blockchain Simulator is divided into two main components, the simulator a and the graphical user interface, we will look into them separately.

4.1.1 Simulator

The simulator makes use of OCaml's *modules* and *functors* to provide modularity and extensibility. MOBs adopts a *Discrete-Event Simulation Model* making the state of the system only change in discrete points in time when events occur. These events are stored in a queue ordered with two main values:

- **Timestamp:** This value dictates the order in which the events are stored in the queue and is based on the simulator's internal clock. When getting a new event the simulator will fetch from the queue the one with the smallest timestamp and move its internal clock to match that event's timestamp.

- **Target:** The entity that should process this event.

Events are fetched from the queue until no more events remain or a predefined stopping condition is reached.

The simulator is built in a module based architecture, Figure 4.1 illustrates how the different modules interact. These modules are:

- **Main:** Entry point for the simulator, manages the execution of protocols.

- **Protocol:** Top-level loop of the simulation, initializes the different nodes, network topology, event queue and performs event handling and delegation.

- **Node:** This is a user defined module that describes the behaviour of an entity in the simulated protocol.

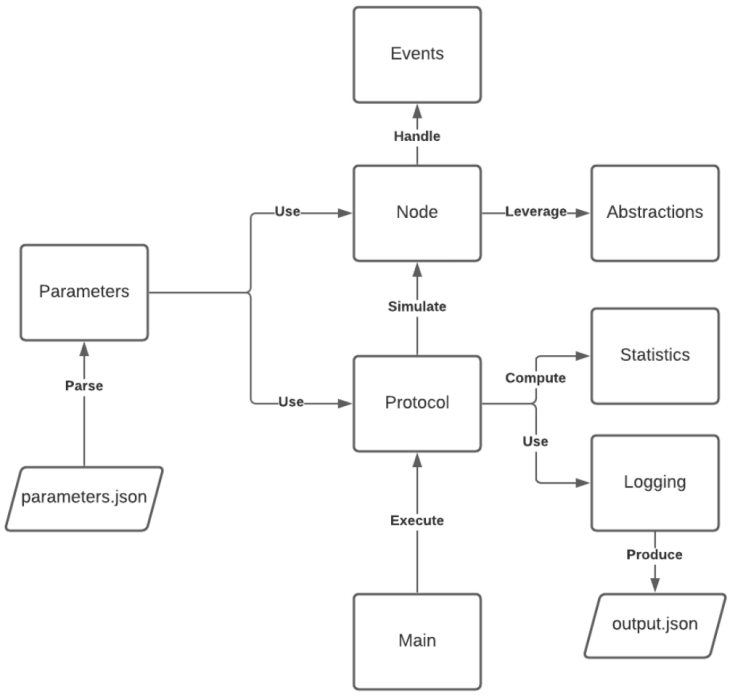

Figure 4.1: Illustration of top-level module interactions

- **Abstractions:** This module provides primitives to aid in the development of new simulation such as proof of stake sortation, proof of work mining, timers, alarms and message exchanges.
- **Statistics and Logging:** Extract metrics and values from the execution to be processed by the GUI.

4.1.2 Graphical User Interface

The graphical user interface was implemented in NodeJS, Vue3 and ElectronJS. The choice for web technologies enables the future deployment of simulator as a web application. The GUI was developed with the goal of allowing the users to use it with their own custom simulators as long as the following conditions are met:

1. The simulator uses a parameters.json as an input with three categories, General, Network and Protocol, the actual parameters inside each category are user defined.
2. The output of the simulator produces log with two top-level entries, kind and content. Kind can be one of ten values, each with their specific content, *parameters, add-node, add-link, flow-message, add-block, node-committee, node-proposer, create-block, statistics* and *per-node-statistics*.

The GUI is composed of 4 pages, described in the following sections.

4.1.2.1 Parametrization

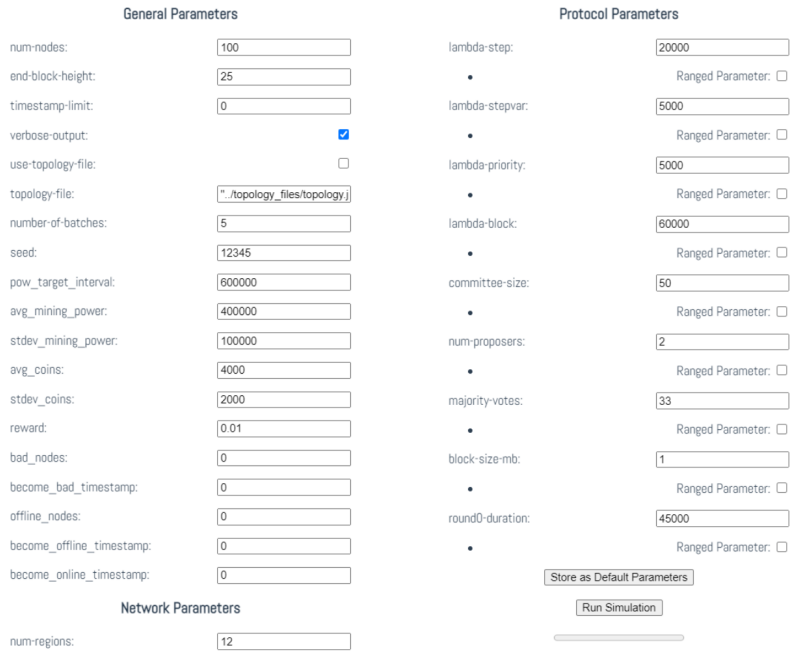

Figure 4.2: Parameters window

The Parameters window parses the parameters.json file and produces a form where the user can customize the values or ranges of values for every parameter.

4.1.2.2 Topology Specification

The GUI also allows user to specify the topology of the network without needing to manually write the JSON file. The Topology window offers a canvas to construct a network topology as well as set individual parameters for each node.

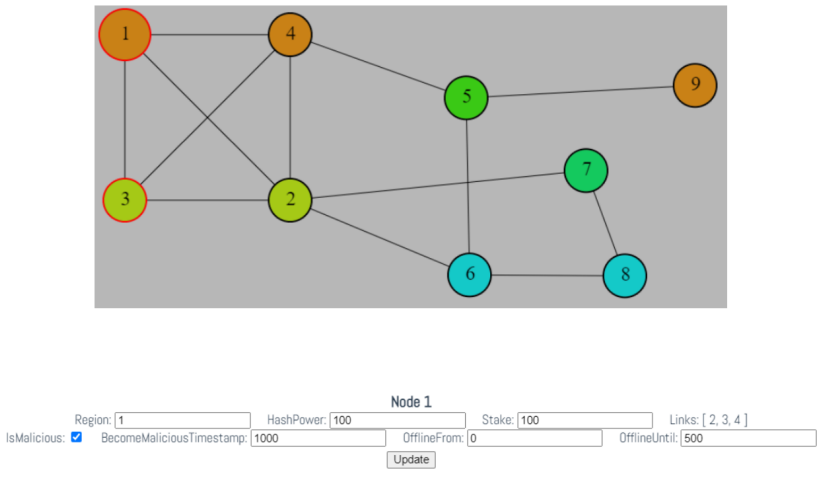

Figure 4.3: Topology window and possible parametrizations for each individual node

4.1.2.3 Visualizer

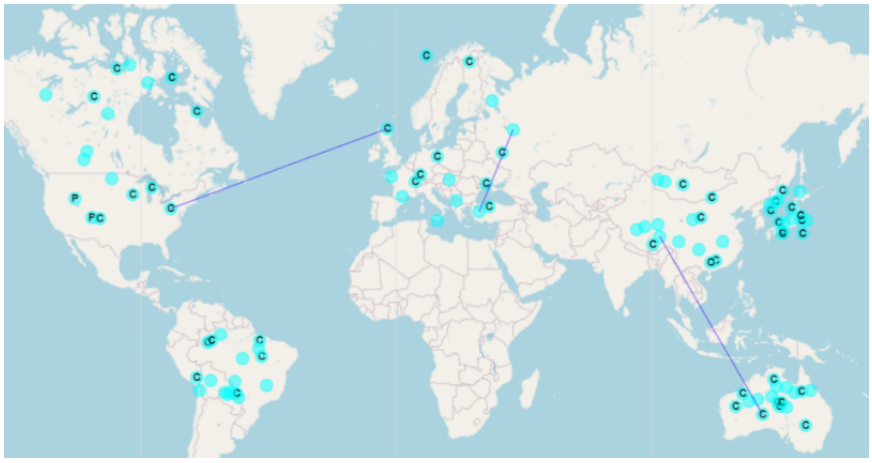

Figure 4.4: Time-lapse in the Visualizer window

The Visualizer window allows user to play the state of each node in a time-lapse manner and visualize exchanged messages.

4.1.2.4 Statistical Analysis

The Statistics window aids in the analysis of the metrics produced by the simulator. The GUI will parse the output.json file and display it in an easy-to-read format. These formats can come as graphs, displaying minimum and maximum values observed for each metric that was produced and a graph with per node statistics.

4.2 Initial Network Exercise

As an initial exercise we chose to implement Chord in MOBs, this served twofold:

- To understand how MOBs worked and how to use it to implement protocols
- To see the capabilities of MOBs in providing qualitative metrics to evaluate a membership protocol.

To achieve this implementation we made use of MOBs Node module and implemented the following events:

- **Join:** An event that a node receives when another node first joins the network. The receiving node will evaluate if the identifier of the new node makes him a candidate to be that node's predecessor in the ring.
- **JoinResponse:** A node that has previously received a join will reply with this event if the new node has been accepted as their predecessor. This contains the identifier of the old predecessor so that the new node can make it its own.

- **Uft:** This event stands for update finger table, this event shares an identifier known to a node, being from their finger table, their own, or one of its neighbors with a random node. The receiver node will add this identifier to their finger table.
- **UpdateSuccessor:** This event is only used when a node updates their successor this event is sent to itself as a way to log the new link.
- **UpdatePredecessor:** This event works the same way as UpdateSuccessor but for the predecessor node.
- **RebalanceNetwork:** When a node receives this event it comes with an identifier, the receiver node will see if the identifier is a better candidate for their successor. This serves as a cyclic membership managing strategy.
- **SendRebalanceNetwork:** Periodically a node will propagate this message to neighboring nodes to trigger a RebalanceNetwork event with their identifier.

The implementation of this protocol at the Node level turned out to not be successful.

One of the reasons is that when implementing this at the Node module, the network module implements a network with a random graph, and as an initial operation, one node would trigger a RebalanceNetwork with the identifier of another node and through an epidemic broadcast fashion trigger all nodes into trying to join to other nodes. This resulted in one of two scenarios:

- Several Chord rings would be created instead of one single ring with all nodes, making further communications since at the Chord protocol level, nodes from different rings would be unknown from each other.
- After making nodes send periodic SendRebalanceNetwork events to neighboring nodes at the Network module level, if the first couple of events made them unable to integrate the ring they would become dormant, since the trigger to send new SendRebalanceNetwork events is triggered by receiving events from other nodes

Another reason this protocol failed was that even leveraging overlay created at the Network layer to send SendRebalanceNetwork events to know nodes at that level to ensure that all nodes would converge to the same ring, this would create an enormous amount of events that would slow down the simulator. This showed some promising results, and we were able to see signs of a converging network, but the search for better neighbors at the Chord level turned out to be a blind search, and the closer the network became to converge the longer it took for new updates to take place.

To solve these issues we proposed move the implementation to the network module level. This will make possible to initiate one node at a time and reduce blind lookups, solving all three problems at once:

- Only one ring will be created

- Since nodes would always ping a known member of the ring to join, the protocol would ensure they would be placed on the correct place of the network and not be left out isolated

- Join operations in a formed chord ring would be less costly in the amount of messages generated since we don't have to broadcast events and can instead target to specific nodes.

We can also leverage this implementation at the network level to make the network module even more modular and extensible, making it easier to implement new network protocols.

4.3 Consensus protocols implemented

One of the problems we want to tackle with this thesis is getting better qualitative information out of blockchain and consensus protocols logs so that we can evaluate their correctness at the end of the simulation. To achieve this we implemented two simple and well known consensus protocols so that we can extract the runtime information and use them as a simple example to see MOBs' limitations and if further work needs to be done in Statistics and Logging module.

The first of these protocols was Paxos. This was implemented as referenced in 2.2.2, with only main changes to the protocol made taking an optimistic approach and as a first phase not worrying about scenarios where malicious nodes are present or messages are lost. The second was having only 1 proposer through the whole simulation, making this an implementation closer to MultiPaxos than regular Paxos, this was done for the sake of simplicity an ease of implementation and testing.

With and implementation of Paxos done and validated, a script in Python was done that allowed us To scrub the logs and extract runtime metrics:

```
Value: 6816668 Reached at: 96928ms nodes agreed before: 6 nodes agreed after:3
Value: 1261359 Reached at: 97595ms nodes agreed before: 7 nodes agreed after:2
Value: 2731005 Reached at: 97984ms nodes agreed before: 6 nodes agreed after:3
Value: 4043237 Reached at: 98564ms nodes agreed before: 6 nodes agreed after:3
```

Figure 4.5: Output from the first iteration of the log analyser script

This was the first iteration of this script which allowed us to see what value was accepted, at what tie it was accepted, how many nodes accepted the value and how many accept messages reached the proposer after the value was accepted. On the second iteration of this script we added a section with condensed global data, where we can more easily observe metrics like number of values accepted, total simulation runtime, average time per consensus and the average acceptance percentage of consensus.

The implementation of both the protocol and the script can be consulted at this pull request: https://github.com/RMLoureiro/MOBS/pull/2.

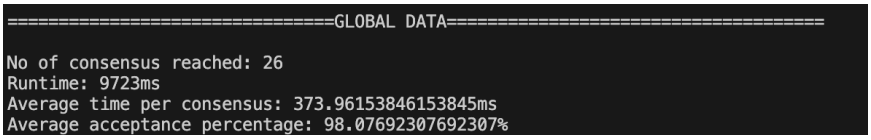

Figure 4.6: Output from the second iteration of the log analyser script

With this work done we implemented Chandra-Tueg as described in 2.2.1, we chose this protocol for the ease of implementation and again focused on an optimistic implementation where we didn't take into account malicious nodes and lost messages. We also took into account and implemented the logs of this protocol so that the same script developed for Paxos could be used without changes to analyse this protocol. The implementation can be consulted at this pull request: https://github.com/RMLoureiro/MOBS/pull/4.

5

Work to be developed

This chapter will describe the work that will be developed to address the problems presented in 1.2 making use of the tools and information presented in 3 and 4.

5.1 Proposed work

The work will be divided in two sections, enhancing MOBs' network layer, and the extending the logs and statistics module to allow for better qualitative information about the implemented protocols at the node level.

5.1.1 Network Layer Enhancement

In this section we will present the proposed enhancements to the Network layer of MOBs. For this, we propose the following work to be conducted:

- **Modular Network Layer:** The goal is to make the network layer modular and allow for the implementation of diverse membership protocols. This will allow for a more diverse set of parametrizations and network environments when testing the already implemented.
- **Network Module Parametrization:** Here we want to expand the already provided parametrizations of this module, allowing for the parametrization of events such as, allowing nodes to join the network after the simulation already started, allowing a random amount of nodes to fail at a random point during execution and simulate periodic bandwidth slowdowns between certain nodes.

These enhancements to the Network module will allow not only for the advantages described above, but also allow for the network itself to restructure in case of node failures and a closer replication of real-life conditions, since right now the overlay layout is static, besides being able to parametrize nodes to be offline after a certain amount of time. In the real world if this happens the network will restructure and initiate protocols to keep its integrity and correctness, which depends on the protocol that was implemented, for

Chord we would try to re-structure back to a cyclic ring formation, for HyParView we would populate each node's active vies to go back to the parametrized size.

For this we propose the implementation of three membership overlay protocols, Chord, HyParView and X-BOT. These three protocols were chosen so that we can have a structured overlay protocol, an unstructured protocol and a partially-structured respectively.

We also propose a small upgrade to the GUI, allowing users to select the network protocol that they desire their simulation to run on alongside with the option for a validation script to be run after to check the network's correctness and integrity. These two extensions will allow users to either check the validation of a network protocol if they choose to implement a new one, or make sure of the integrity of the network overlay when studying the logs from the protocols that run on top of it.

We propose to evaluate this work in the following way:

- After implementation of these protocols a script in Python with the aid of the NetworkX package should be written in order to validate the overlay's properties. The script should be able to generate a graph from the logs provided by MOBs. The values of their properties and the metrics we can extract the overlay should be comparable to the ones presented in protocol's paper and other tests that were conducted on them.

- After validation of each of these protocols run the already implemented consensus and blockchain protocols, and we should see similar results to the ones obtained when testing with the current Network layer. We should expect some variance in the performance of the protocols implemented at the Node module since one of the goals of this enhancement is to study their performance using different overlays.

- The enhancements to the GUI should be seamless and work for existing scenarios while now giving the ability to select which network protocol to run, the validation script for said protocol and show the results given by that script.

5.1.2 Improve qualitative data

In a first step to better understand what information we can extract and how we can leverage the simulator to evaluate Blockchain protocols, we will first implement simpler consensus protocols, namely Chandra-Toueg, Paxos and MultiPaxos described in 2.2.

Implementing these protocols will allow us to better understand and see if changes need to be done to the current log and statistics modules in order to obtain better qualitative data, needed to evaluate these protocols. For this further study needs to be done in order to evaluate what data and properties we need.

After these implementations and validations we will move to blockchain protocols and see if we can qualitativly evaluate these protocols, specifically we will try to replicate the vulnerabilities found in Ethereum in [15].

5.2 Work Plan

In this section we present the work plan for the elaboration of this work. We will divide the work into six phases. The enhancement of the network layer, study of the values and parameters needed for the validation of consensus protocols, implementation of consensus protocols and their validation, implementation of blockchain protocols and their validation, writing an article detailing the work conducted and finally writing of the final thesis document.

Table 5.1: Schedule

Task	Start Date	End Date	Weeks
Enhancement of the network layer Implementation Testing and Validation	31 July	8 September	5
Consensus protocols Study of consensus protocols in MOBS Implementation Testing and Validation	11 September	20 October	6
Blockchain protocols Study of needed parameters for evaluation Implementation Testing and Validation	23 October	22 December	9
Thesis plus Article	2 January	9 February	6

On our planning we are leaving about 2 months as a buffer in case any of the phases of work becomes more troublesome than expected.

Bibliography

[1] M. Alharby and A. van Moorsel. "Blocksim: An extensible simulation tool for blockchain systems". In: *Frontiers in Blockchain* 3 (2020), p. 28 (cit. on p. 19).

[2] V. Allombert, M. Bourgoin, and J. Tesson. "Introduction to the Tezos Blockchain". In: *CoRR* abs/1909.08458 (2019). arXiv: 1909.08458. URL: http://arxiv.org/abs/1909.08458 (cit. on p. 1).

[3] Y. Aoki et al. "Simblock: A blockchain network simulator". In: *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2019, pp. 325–329 (cit. on p. 19).

[4] V. Buterin. "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform". In: (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper (cit. on p. 1).

[5] T. D. Chandra and S. Toueg. "Unreliable failure detectors for reliable distributed systems". In: *Journal of the ACM (JACM)* 43.2 (1996), pp. 225–267 (cit. on p. 9).

[6] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design (International Computer Science)*. 4th rev. ed. Addison-Wesley Longman, Amsterdam, 2005. ISBN: 0321263545 (cit. on p. 9).

[7] A. Das, I. Gupta, and A. Motivala. "SWIM: scalable weakly-consistent infection-style process group membership protocol". In: *Proceedings International Conference on Dependable Systems and Networks*. 2002, pp. 303–312. DOI: 10.1109/DSN.2002.1028914 (cit. on p. 6).

[8] C. Faria and M. Correia. "BlockSim: blockchain simulator". In: *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE. 2019, pp. 439–446 (cit. on pp. 16, 19).

[9] P. Fouto et al. *Babel: A Framework for Developing Performant and Dependable Distributed Protocols*. 2022. arXiv: 2205.02106 [cs.DC] (cit. on pp. 14, 15).

[10] A. A. Hagberg, D. A. Schult, and P. J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by G. Varoquaux, T. Vaught, and J. Millman. Pasadena, CA USA, 2008, pp. 11–15 (cit. on p. 15).

[11] M. Jelasity, A. Montresor, and O. Babaoglu. "T-Man: Gossip-based Fast Overlay Topology Construction". In: *Comput. Netw.* 53.13 (2009-08), pp. 2321–2339. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2009.03.013. URL: http://dx.doi.org/10.1016/j.comnet.2009.03.013 (cit. on p. 8).

[12] L. Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001-12), pp. 51–58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/ (cit. on pp. 10, 15).

[13] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 419–429. ISBN: 0-7695-2855-4. DOI: 10.1109/DSN.2007.56. URL: https://doi.org/10.1109/DSN.2007.56 (cit. on p. 7).

[14] J. Leitão et al. "X-BOT: A Protocol for Resilient Optimization of Unstructured Overlay Networks". In: *IEEE Transactions on Parallel and Distributed Systems* 99.PrePrints (2012). ISSN: 1045-9219. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.29 (cit. on p. 8).

[15] U. Pavloff, Y. Amoussou-Guenou, and S. Tucci-Piergiovanni. *Ethereum Proof-of-Stake under Scrutiny*. 2022. arXiv: 2210.16070 [cs.CR] (cit. on pp. 2, 28).

[16] A. Rowstron and P. Druschel. "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". In: *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. Vol. 2218. Springer Berlin Heidelberg, 2001-11. Chap. Middleware 2001, pp. 329–350. ISBN: 978-3-540-45518-9. URL: https://www.microsoft.com/en-us/research/publication/pastry-scalable-distributed-object-location-and-routing-for-large-scale-peer-to-peer-systems/ (cit. on p. 6).

[17] J. Sliwinski et al. "Halting the Solana Blockchain with Epsilon Stake". In: *Proceedings of the 25th International Conference on Distributed Computing and Networking*. ICDCN '24. <conf-loc>, <city>Chennai</city>, <country>India</country>, </conf-loc>: Association for Computing Machinery, 2024, pp. 45–54. ISBN: 9798400716737. DOI: 10.1145/3631461.3631553. URL: https://doi.org/10.1145/3631461.3631553 (cit. on p. 2).

[18] I. Stoica et al. "Chord: a scalable peer-to-peer lookup protocol for Internet applications". In: *IEEE/ACM Transactions on Networking* 11.1 (2003), pp. 17–32. DOI: 10.1109/TNET.2002.808407 (cit. on p. 5).

[19] L. Stoykov, K. Zhang, and H.-A. Jacobsen. "Vibes: fast blockchain simulations for large-scale peer-to-peer networks". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos*. 2017, pp. 19–20 (cit. on pp. 16, 19).

[20] H. Yajam, E. Ebadi, and M. A. Akhaee. "JABS: A Blockchain Simulator for Researching Consensus Algorithms". In: *IEEE Transactions on Network Science and Engineering* (2023) (cit. on pp. 18, 19).

[21] A. Yakovenko. "Solana: A new architecture for a high performance blockchain v0. 8.13". In: *Whitepaper* (2018) (cit. on p. 2).

[22] S. Zhang and J.-H. Lee. "Analysis of the main consensus protocols of blockchain". In: *ICT Express* 6.2 (2020), pp. 93–97. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2019.08.001. URL: https://www.sciencedirect.com/science/article/pii/S240595951930164X (cit. on p. 12).