NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

MIGUEL CERDEIRA ALVES

Bachelor in Science and Computer Engineering

A FRAMEWORK FOR SPECIFICATION AND SIMULATION OF CONSENSUS PROTOCOLS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
⟨November⟩, ⟨2021⟩

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

A FRAMEWORK FOR SPECIFICATION AND SIMULATION OF CONSENSUS PROTOCOLS

MIGUEL CERDEIRA ALVES

Bachelor in Science and Computer Engineering

Adviser: António Ravara
Associate Professor, NOVA School of Science and Technology

Co-adviser: Marco Giunti
Researcher, NOVA School of Science and Technology

Examination Committee

Chair: Luís Manuel Marques da Costa Caires
Full Professor, NOVA School of Science and Technology

Rapporteur: Miguel Nuno Dias Alves Pupo Correia
Full Professor, Instituto Superior Técnico da Universidade Nova de Lisboa

Member: António Maria Lobo César Alarcão Ravara
Associate Professor, NOVA School of Science and Technology

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
⟨November⟩, ⟨2021⟩

A Framework for Specification and Simulation of Consensus Protocols

Acknowledgements

First and foremost I would like to express my gratitude towards my adviser, Prof. António Ravara, for his invaluable insight, availability, and guidance. I would also like to thank my co-adviser Marco Giunti and Prof. Simão Melo de Sousa for sharing their vast knowledge of OCaml, and Prof. João Leitão for sharing his Distributed Systems knowledge as well as providing numerous feedback and suggestions.

I must also show my appreciation to my friends Afonso, Serhiy, and David, for all the jokes and laughter, even through stressful times.

Last, but most certainly not least, I would not have gotten this far without the overwhelming support shown by my parents, at every step of my personal and academic life.

“ Try again. Fail again. Fail better. ” (Samuel Beckett)

Abstract

Consensus protocols enable the coordination between multiple machines, allowing them to maintain a replicated state, and are therefore crucial in building reliable, large-scale distributed systems such as blockchain technologies, which have seen a rise in popularity in recent years.

However, these protocols are not written in stone, they are constantly evolving. For example, Ethereum is progressively migrating away from proof of work and towards a solution based on proof of stake. Another example is Tezos, where stakeholders are capable of proposing and agreeing on changes to the consensus protocol itself, allowing it to evolve over time. It is, therefore, necessary to develop tools to assist in this evolution.

This thesis presents a framework to address the lack of specification and experimentation environments to develop consensus and blockchain protocols, and thus help researchers make informed decisions about the effects that certain design choices have on the behavior of these protocols.

The framework is divided into two components, a simulator, and a graphical user interface. The simulator was built with a focus on extensibility and modularity, and allows users to simulate any family of protocols as well as parameterize multiple scenarios to study these protocols. For example, users can parameterize bandwidth limits, byzantine conditions such as message loss, periods of time when nodes are offline as well as adversarial behavior, and extract any number of statistics.

The graphical user interface offers a more intuitive way to perform parameterizations of the protocol, topology, and simulated nodes. Furthermore, it processes the output of the simulator, enabling the analysis of the messages exchanged between nodes and state changes, and it generates graphs for all user-defined statistics produced by the simulator.

The framework was validated by implementing a variety of protocols, such as Bitcoin-like, Algorand, Tenderbake, and Ouroboros. We demonstrate that the framework allows us to obtain, for example, realistic values for block interval and block propagation time for Bitcoin-like protocols. Furthermore, we show that the results obtained from studying the impacts caused by the block size on the duration of an Algorand round are equivalent to the results the authors of the protocol obtained through emulation.

Resumo

Os protocolos de consenso permitem a sincronização entre múltiplas máquinas de forma a que estas armazenem um estado replicado, sendo assim cruciais no desenvolvimento de sistemas distribuídos de confiança e de grande escala, como é o caso das tecnologias de blockchain que têm ganho popularidade nos últimos anos.

No entanto, estes protocolos evoluem constantemente. Por exemplo, o Ethereum tem vindo a adotar gradualmente uma solução baseada em proof of stake. Um outro exemplo é o caso da Tezos, onde os stakeholders podem propôr e votar em alterações ao protocolo de consenso em si, permitindo que este evolua ao longo do tempo. É assim importante desenvolver ferramentas que assistam esta evolução.

Esta tese apresenta uma framework para combater a falta de ferramentas de especificação e experimentação para desenvolver protocolos de consenso e de blockchain, de forma a assistir investigadores nas decisões que necessitam de realizar no desenho e desenvolvimento destes protocolos.

A framework encontra-se dividida em duas componentes, um simulador e uma interface gráfica. O simulador é modular e extensível, permitindo que utilizadores simulem qualquer família de protocolos, e permite o estudo dos mesmos através da parameterização de diferentes cenários. Por exemplo, um utilizador pode definir limites de largura de banda, condições bizantinas como perda de mensagens, períodos de tempo em que certos nós estão offline, existência de nós maliciosos, bem como extração de diversas estatísticas.

A interface gráfica disponibiliza uma forma mais intuitiva de parameterizar todos os componentes da simulação. Para além disso, processa os ficheiros produzidos pelo simulador, de forma a possibilitar a análise das mensagens trocadas entre nós e das mudanças de estado, e gera gráficos para todas as estatísticas produzidas pelo simulador.

A framework foi validada implementando diversos protocolos, nomeadamente Bitcoin-like, Algorand, Tenderbake e Ouroboros. Foi demonstrado que a framework permite obter valores realistas para o intervalo entre blocos, e tempo de propagação de blocos. Foi também demonstrado que os valores obtidos no estudo do impacto causado pelo tamanho dos blocos na duração de uma ronda de Algorand são equivalentes aos resultados que os autores do protocolo obteram por emulação.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

1.1 Context

Consensus protocols have crucial roles among distributed systems, such as blockchain technologies. They enable the coordination between multiple machines, allowing them to maintain a replicated state, and are therefore crucial in building reliable, large-scale distributed systems.

In recent years, blockchain technologies have been rising in popularity leading to the emergence of several different protocols, algorithms, and parameterizations.

Blockchain protocols are a composition of steps executed by each node in a network to maintain a distributed ledger among multiple nodes, and also enable the development of smart contracts. These protocols possess several underlying components, or algorithms, and each of which may have several parameters that can influence the performance and behavior of the overall protocol.

It should be highlighted that these protocols are not written in stone, they are constantly evolving. For example, Ethereum [8] uses a blockchain protocol based on proof of work. Their current goal is to migrate towards an implementation based solely on proof of stake for Ethereum2 [14]. At the current stage of development, Ethereum2 is implemented as a hybrid proof of work and proof of stake protocol, utilizing a proof of work block proposal mechanism, and a proof of stake checkpoint mechanism for block finalization.

Another example is Tezos [2], which possesses a key novel of self-amendment. Stakeholders participating in the system propose and agree on changes and upgrades to the core protocol itself, allowing the protocol to evolve over time. It is, therefore, necessary to develop tools that can aid in supporting these evolutions.

1.2 Problem

Consensus protocols are notoriously difficult to understand and even harder to define correctly. In the context of blockchain systems, mastering their dynamical behavior and

trust in their correctness is crucial, since the protocols control financial transactions. Moreover, understanding them is essential to enable evolution. **However**:

- there is a lack of specification and experimentation environments to develop consensus and blockchain protocols and thus study their behavior.
- existing simulators focus strictly on performance evaluations, and there aren't, to the best of our knowledge, tools that allow developers to study the behavioral changes caused by different parameterizations of the protocols.
- there are no tools that offer a programmatically well-defined, compositional, and extensible way to define these protocols, with distinct examples.

which are the problems this thesis aims to address.

1.3 Goal

To address the identified problems, the goal of this thesis is to develop a simulation environment that allows researchers to make informed decisions about the effects of different parameterizations of the underlying components involved in blockchain protocols.

The simulation environment should:

- be modular and extensible, providing the ability to make changes to the protocols, and simulate different families of protocols.
- be parameterizable, to allow researchers to learn the effects of different parameters in the overall behavior of the protocol.
- provide a well-defined structure for implementing new protocols.
- focus on providing a qualitative evaluation of the simulated protocols.

The goal of the simulation environment is to aid researchers in the entire development process of a new protocol. In the earlier stages of development, the main concern is whether or not the protocol behaves as expected - for example if the messages exchanged between nodes contain the expected contents - and if it is not behaving as expected the simulator should help researchers understand why. Performance becomes more important at the later stages of development.

Henceforth, although the simulator should produce realistic results, the goal is **not** strictly to simulate the protocol's behavior when compared to reality as a means for testing its efficiency, but also to provide researchers tools and results that allow them to reason about certain properties of their implementations, such as: Are nodes able to reach consensus in a reasonable time? Is the system fair when handling stakeholders with different stakes? Can the system reach a consensus when in the presence of adversaries?

1.4 Contributions

The contribution of this thesis is a simulator paired with a graphical user interface, of which we highlight the following functionalities that separate it from the state of the art (further detailed in Chapter 3):

- the simulator is capable of simulating very distinct protocols, providing implementations for Algorand, Tenderbake, Ouroboros-BFT, Ouroboros-Praos, and Bitcoin-like protocols.
- users can define periods of time where a given node is offline.
- the simulator models byzantine behavior such as loss and reordering of messages, and users can, not only specify malicious behavior for nodes, but they can also specify that nodes only begin acting maliciously at a specific point in time during the simulation.
- the underlying network topology can be pseudo-randomly generated based on input parameters, but users can choose to leverage the graphical user interface to create specific network topologies and individually parameterize each node's characteristics to study specific scenarios.
- the simulator is capable of modeling both mining pools and stake pools.
- the graphical user interface is decoupled from the specific protocols that are simulated, allowing it to display any protocol as well as any user-defined statistics.
- the simulations are parameterizable via the graphical user interface, and users can assign intervals of values to given parameters, automatically producing graphs showing the impact caused by each combination of parameters on the produced statistics.

In the following chapters, we will begin by presenting relevant concepts regarding consensus protocols, blockchain protocols, and simulation. Then, we will explain the inner workings of the simulator, namely how it is structured, the logic behind each component, how the different components interact, and how each functionality was validated. Finally, the graphical user interface will be presented, and we will conclude with a summary and proposals for future work.

2

Background

This chapter presents core concepts regarding consensus protocols, blockchain protocols, and simulation. It also describes several blockchain protocols that will be further addressed in chapter 4.

2.1 Consensus Protocols

Consensus protocols enable coordination between multiple machines, allowing them to maintain a replicated state and are therefore crucial in building reliable, large-scale distributed systems.

These protocols are widely used in the context of state machine replication, where different processes execute the same operations on the same state.

Each participant of the consensus protocol is referred to as a *process*, which *decide* on a value that gets locked as the result of the consensus.

It is expected that a consensus protocol ensures the following properties [10]:

1. **Termination** - eventually, every correct process decides on a value.
2. **Agreement** - all correct processes decide on the same value.
3. **Validity** - if all correct processes have the same initial value, then the agreed upon value by all correct processes must be that same value.
4. **Integrity** - no process decides more than one value.

The following section will elaborate on blockchain protocols, whose key components are the underlying consensus protocols.

2.2 Blockchain Protocols

The term blockchain is often used in two different contexts.

Firstly, blockchain can refer to an append-only data structure composed of a chain of blocks. At the very least, each block contains a cryptographic hash that identifies the previous block in the chain.

We will focus on the second use case, which is blockchain in the context of distributed systems. A blockchain protocol is the composition of steps executed by each node in a network to maintain a replicated copy of a blockchain among multiple nodes. These protocols often function under the assumption that nodes do not necessarily trust each other, however, they still need to reach a consensus on the ordering of the blocks in the blockchain. There are two main types of blockchain networks, permissioned and permissionless.

A permissioned blockchain, also referred to as a private blockchain, is a closed network where only authorized nodes can join and participate in the blockchain protocol. This does however require a centralized authority that regulates who can join the network. An example of a permissioned blockchain is Hyperledger Fabric [4].

A permissionless blockchain, also referred to as a public blockchain, allows any node to join the network and execute the blockchain protocol, and is, therefore, a fully decentralized environment. It is common for these blockchains to employ some type of incentive mechanism to strengthen their security. Some examples of permissionless blockchains are Bitcoin [21], Ethereum [8] and Algorand [17].

Furthermore, most permissionless blockchains can also execute on permissioned environments, often more efficiently and securely. However, the opposite is not true.

2.2.1 Five-Component Framework

To further explain blockchain protocols, we will first describe the five-component framework presented in *"A Survey of Distributed Consensus Protocols for Blockchain Networks"* [23], as it is the abstraction mechanism that we adopted to specify these protocols.

The authors identify five core components of blockchain protocols: block proposal, information propagation, block validation, block finalization, and incentive mechanism.

Block Proposal

The block proposal component encompasses how each node generates a new block and the corresponding proof of generation.

Examples of block proposal mechanisms include proof of work (PoW), proof of stake (PoS), and proof of authority (PoA), among others.

Information Propagation

The information propagation component relates to how nodes communicate with each other in the network for sending and receiving transactions, blocks, etc.

Gossiping, broadcast, and flooding are examples of information propagation mechanisms.

Block Validation

Block validation includes all operations related to verifying the validity of a propagated block, by checking the corresponding generation proof and the validity of the block's content.

For example, in proof of work protocols, verifying the generation proof of a block usually consists of verifying the block's hash against a certain threshold value, whereas in proof of stake protocols, verifying the generation proof consists in checking if the node that proposed the block is, in fact, eligible to do so.

Block Finalization

The block finalization component encompasses how the nodes in the network reach an agreement on the acceptance of validated blocks.

Some protocols use block finalization mechanisms that involve communication between nodes, such as Practical Byzantine Fault Tolerant consensus algorithms and checkpointing, while other use local finalization mechanisms such as the longest-chain rule and the Greedy Heaviest Observed Subtree (GHOST) rule.

Incentive Mechanism

The incentive mechanism includes the protocol's functionalities that promote the honest participation of nodes in the network, through block creation rewards and transaction fees, for example.

2.2.2 Bitcoin

Bitcoin [21] is a permissionless blockchain network that manages a decentralized digital currency (which is also referred to as bitcoin), allowing online payments to be sent directly between two entities, without the need for a third party central authority. Honest peers work together to make the system trustworthy by validating transactions, creating blocks through proof of work, and appending them to the blockchain.

The following is a description, divided according to the five component framework [23], of the protocol executed in the bitcoin network [21][12].

Block Proposal

Transactions are broadcasted through the network. Nodes running the mining software validate these transactions, pack them into a block and proceed to compute a nonce that, when hashed together with the block's header, produces a hash that begins with a predefined number of zero bits (the number of zero bits is periodically adjusted to allow an average of 6 new blocks per hour). This nonce is the proof object of this protocol and it is hard to compute, but rather simple to verify.

The previously described process is referred to as proof of work.

Information Propagation

Each block or transaction has an origin node where it is first created. After creation, this node immediately sends the corresponding data to its neighbors which is then propagated to the entire network via the gossip protocol.

To avoid sending transactions and blocks to nodes that have already received them from others, they are not forwarded directly. Once a node verifies a received block or transaction, it sends an inv message to neighbor nodes. This message contains a set of transactions and block hashes that the node has verified, and upon receiving an inv message other nodes can request transactions and blocks that they do not have, by responding with a getdata message.

Block Validation

Upon receiving a block, a node validates the transactions and the proof associated with the block. If the transactions are valid, and if the nonce indeed produces a value with the required number of zero bits when hashed together with the block's header then the block is accepted and added to that node's local chain.

Block Finalization

Nodes consider the longest chain in the block tree to be the correct one and will work towards extending it. Forks may occur if two nodes simultaneously broadcast different blocks that extend the same chain, leading different subsets of nodes to extend different chains. Eventually, this fork will be solved when one chain becomes longer than the other and all nodes will extend a common chain once again.

When a block is sufficiently deep in the chain, it can be considered final with a high probability, since the more blocks are built on the chain that extends it, the more work would be required to alter the chain.

Incentive Mechanism

A node is incentivized to mine a valid block, as he not only receives transaction fees from the transactions that get included in the block but also a fixed amount of bitcoins

as a reward for the creation of the block itself which is included in the block as a special transaction that creates those bitcoins.

2.2.3 Algorand

Algorand [17][9] is a blockchain network that manages a cryptocurrency, and is designed to confirm transactions on the order of one minute. At its core, Algorand uses a Byzantine Fault Tolerant agreement protocol that is not only scalable to many users but also allows the network to reach consensus on a new block with low latency and with a low risk of creating forks. The protocol also satisfies user replaceability, allowing for a different subset of users to participate in different steps of the agreement protocol, providing tolerance against targeted denial of service attacks.

The following is a description, divided according to the five component framework [23], of the protocol executed in the algorand network [17][9].

Block Proposal

Nodes collect pending transactions that they learn about into a block, in case they are chosen to propose the next block in the chain.

The nodes then proceed to execute cryptographic sortition that allows them to privately check if they are selected to propose a block. Cryptographic sortition ensures that only a small set of nodes are selected at random, with consideration to their weight (amount of money the node holds in the system). If a node is selected to propose a block, cryptographic sortition also produces a priority (used as a tie-breaker when several nodes are selected) and proof of the node's ability to propose and of its priority.

Upon being selected, a node will propagate two separate messages: a message to inform other nodes of its priority, and another message containing its proposal (the created block).

The described process of selecting a proposer randomly, based on its weight (stake), is referred to as proof of stake.

Information Propagation

Similar to Bitcoin, new transactions are propagated through the network via the gossip protocol, as well as the messages and blocks sent during the agreement protocol.

Algorand avoids forward loops by not allowing nodes to forward the same message twice, and mitigates pollution attacks by only relaying messages after validating their contents and the sender's credentials.

Block Validation

Upon receiving a block, a node will validate the transactions contained in the block and will perform a proposer eligibility check to ensure that the node that sent the block was

indeed selected to do so.

Block Finalization

Cryptographic sortition may select multiple nodes to propose a block in a given round. To reach a consensus on a single block to be appended to the chain, Algorand uses a Byzantine Fault Tolerant agreement algorithm, consisting of the following steps:

1. After performing the Block Proposal step, regardless of whether or node the node was selected to propose a block, it will wait a predefined amount of time before running the next step of the agreement algorithm, to allow time for the *Priority* messages to be propagated along the network.

2. The node will then check if it has already received the *Proposal* created by the node with the highest *Priority*. If so, it will immediately run the next step, otherwise, it will wait a predefined amount of time before timing out, or until the *Proposal* is received.

3. The node will *SoftVote* for the *Proposal* with the highest associated *Priority*, if it exists. Otherwise, it will *SoftVote* for the empty block. The next step will be executed after the node times out, or when it receives a majority of *SoftVotes* from other committee members.

4. If the node has seen a majority of SoftVotes for a non-empty block, it will *CertVote* for that same block. The next step (5) will be executed only if the node times out - if the node sees a majority of *CertVotes* it adds the block to the blockchain and advances to the next round (consensus has been reached).

5. If a node has *CertVoted* for a block during the round, it will *NextVote* for that same block. Otherwise, it will *NextVote* for the empty block. The next step is executed when the node times out.

6. The node will continue to execute step 5 until it sees:

    a) a majority of CertVotes, meaning consensus was reached and it can advance to the next round.

    b) a majority of NextVotes, meaning that nodes were unable to reach consensus. If this occurs, nodes increment the period and restart the round, from step 1.

Note that if a node does not belong to the committee for a given round, it participates as a spectator, keeping track of the messages exchanged by the other peers. This is important because it allows committee members to be replaced every round or period while maintaining the progress achieved by the previous committees, and helps protect the network against targeted denial of service attacks.

Incentive Mechanism

Nodes receive a reward upon creating a block for which consensus is reached and is successfully added to the blockchain.

2.2.4 Tenderbake

Tenderbake [6] is a byzantine-fault tolerant consensus protocol designed to bring deterministic finality to the Tezos [2] blockchain. The following is a description, divided according to the five component framework [23], of the Tenderbake protocol [6].

Block Proposal

The protocol progresses in levels and rounds. The level is given by the current height of the chain. For a given round, nodes will execute a pseudo-random number generator to see if they will participate in that round and if so obtain their role, which is either baker or committee member (validator).

Once a node sees that it has been assigned the baker role, it will send a proposal, containing a block. This block can either be a new block, created by the node, or it can be an endorsable block from a previous round. If a node sees a quorum of preendorsements for a given round, it marks the corresponding block as locked, and it must propose and/or vote for that block in subsequent rounds (further explained under **Block Finalization**).

Furthermore, when creating a new block, the node will include an *endorsement quorum certificate* of the parent block, as proof that it is extending a block for which consensus was reached.

Information Propagation

Propagation of transactions, messages, and blocks is performed via the gossip protocol. To mitigate pollution attacks, a given message is only forwarded once, and only valid messages are propagated.

Block Validation

A proposed block will be considered valid if it is composed of valid transactions, was created by the baker assigned to that round, and extends the longest valid chain that the node has seen.

Block Finalization

Each round has an associated duration, and nodes have access to loosely synchronized clocks. If nodes fail to agree on a block for a given round, the duration of the next round will increase.

In each round, nodes will attempt to agree on the contents of a block for the current level. To this end, a node will repeat the following steps each round:

1. Upon receiving a proposal **p**, the node will proceed to validate it. If the node is locked on a proposal different from **p** of a previous round, it will respond with a *preendorsements* message, containing the observed preendorsements for the locked proposal, to justify its decision. Otherwise, if the proposal is valid, the node will proceed to preendorse that proposal, by propagating a *preendorsement* message.

2. If a node receives a majority of *preendorsement* messages for **p** (preendorsement quorum certificate), it will lock on **p** and endorse it, by propagating an *endorse* message.

3. At the end of the round (computed by the round duration), if a node has received a majority of *endorse* messages for **p** (endorsement quorum certificate) then nodes have successfully agreed that **p** is the block for that level of the chain. Otherwise, nodes were unable to agree on a block, and will therefore begin a new round for the same level.

Incentive Mechanism

The baker (proposer) of a block will receive transaction fees as well as baking rewards. Validators will also be rewarded for endorsing a block that gets added to the chain.

2.2.5 Ouroboros BFT

Ouroboros [19] is a peer-reviewed, verifiably secure blockchain protocol used in the Cardano blockchain. Ouroboros-BFT was an initial version of the protocol, designed to provide consistency and liveness under circumstances where up to one third of the participants could act maliciously, under byzantine conditions.

The following is a description, divided according to the five component framework [23], of the Ouroboros-BFT protocol [19].

Block Proposal

Time is divided into slots, and in each slot, a node is selected to propose a block. Nodes take turns proposing blocks in a predetermined round-robin fashion.

If a node is selected to propose a block for the current slot, it will always attempt to extend its longest valid chain.

Information Propagation

Nodes propagate received blocks via gossip, if the block is valid.

Block Validation

A block is valid if its contents are valid and the signature associated with the block is the signature of the node that is assigned to the corresponding slot.

Block Finalization

When a new slot begins, the selected node will propose a new block to extend its longest valid chain.

Upon receiving a new block, a node will consider it as the head of its chain if the block is valid and extends the longest chain of valid blocks it has seen so far.

Incentive Mechanism

Stakeholders that operate stake pools, or delegate their stake to existing stake pools are rewarded with transaction fees. The protocol produces optimal rewards for all participants when the stake is delegated uniformly among a predefined number of stake pools.

2.2.6 Ouroboros Praos

Ouroboros-Praos [11] is a newer version of the protocol, designed to provide security against fully-adaptive corruption, where an adversary can corrupt any node at any moment, as long as the stakeholders maintain an honest majority of the stake.

The following is a description, divided according to the five component framework [23], of the Ouroboros-Praos protocol [11]. Note that we only present the Block Proposal and Block Validation components, as the remaining components are the same as in Ouroboros-BFT.

Block Proposal

There can now be multiple leaders per slot. Slot leaders are selected through the use of cryptographic sortition and verifiable random functions. As a result, an adversary cannot know which nodes will be selected as leaders in future slots.

If a node, upon executing the verifiable random function, sees that it is a leader for the current slot, it will create a new block that extends the longest valid chain it has observed, it will set the newly created block to be the head of its local chain and it will propagate the created block along with its credentials for the current slot.

Block Validation

A given block is valid if its contents are valid, the associated credentials belong to a leader of the current slot, and it extends a chain composed of valid blocks.

2.3 Simulation

"Simulation is the imitation of the operations of a real-world process or system over time" [7]. It is a crucial tool for solving problems and studying the behavior of real systems. Simulation offers several benefits [7] such as providing a controlled environment to test specific scenarios, without the influence of external factors. It also provides users the

ability to test the effects of system changes, before deploying those changes to production where any mistakes can have severe impacts, which is of utmost importance in blockchain systems since they often control financial transactions.

The following sections will present the fundamental simulation concepts, compare simulation to emulation, and present discrete-event simulation, as it is the chosen simulation model for our project. Unless explicitly stated, the presented information is based on *Simulation Modeling and Analysis* [20] by Averill Law.

2.3.1 Concepts

The advantage of simulation when compared to emulation is that it gives users more control over the system and the environment in which it operates, whereas in emulation some outside factors may impact its results.

In simulation, a system is a collection of entities that interact with each other and with the environment towards achieving a goal. A simulation may include all, or a portion of, the entities that exist in the real system.

The state of a system is a collection of variables that are needed to describe the system at a particular point in time, and are directly related to the results that the simulation should produce.

A system is often simulated through a model. A model is a representation of the real system and is often a simplified version of it, however, it must be detailed enough to allow for valid conclusions to be drawn about the real systems.

A simulation model can be classified along three different axes:

1. **Static or Dynamic**
    a) A model is classified as *static* if it represents a system at a specific point in time.
    b) A *dynamic* model represents the system as it evolves over time.
2. **Deterministic or Stochastic**
    a) A model is *deterministic* if the behavior of its entities does not rely on probabilistic components, such as randomizations.
    b) In a *stochastic* model, the simulation relies on random inputs.
3. **Continuous or Discrete**
    a) In a *continuous* model, the state of the simulation varies continuously over time.
    b) In a *discrete* model, the state only changes at discrete points in time.

A model that is *discrete*, *dynamic* and *stochastic* is referred to as a **discrete-event simulation model**.

2.3.2 Discrete-Event Simulation

Discrete-Event Simulation is used to model systems that evolve over time, using a set of state variables that change at discrete points in time - points where events occur. An *event* is seen as an instantaneous occurrence that can change the state of the system.

In discrete-event simulation, there is a distinction between *real time* and the *simulation time*. Real-time is the amount of time a simulation takes to execute. The simulation time is the view of time inside the simulation. This distinction is important because it enables the contraction or dilation of time, allowing users to study the long-term behavior of the system in a shorter amount of time, for example.

The existence of simulation time requires a mechanism for managing this clock. The most commonly used is the next-event time advance mechanism, which consists of computing the times when future events will occur and advancing the simulation clock to match the time of occurrence of the first of those events. An advantage of this mechanism is that it ignores periods of inactivity (where no events occur) by jumping the clock from event to event.

Although simulation can be used in a large variety of areas, some core components are present in most discrete-event simulation models [20]:

1. **System state** - the collection of all state variables that are necessary to represent the system at a specific point in time.
2. **Simulation clock** - a variable representing the current value of the simulation time, as well as the mechanism for advancing the clock.
3. **Event list** - a list containing the next time each event will occur.
4. **Statistical counters** - set of variables used to track statistical information required for calculating metrics regarding the simulation.
5. **Initialization routine** - responsible for initializing every component of the simulator at time 0.
6. **Event routine** - responsible for updating the state of the system when an event occurs.
7. **Library routines** - responsible for abstracting the execution of certain tasks in the system.
8. **Report generator** - produces a report with the results of the simulation.
9. **Main program** - the top-level simulation loop.

Furthermore, from a user's point-of-view, modelling discrete-event simulations can be done via two different approaches: *event-scheduling* or *process*. The event-scheduling approach consists of modeling a system by defining its key events and the corresponding

event-handler routines, that give a detailed description of the state changes that take place when each event occurs. On the other hand, the process approach consists of defining a single routine for an entity that describes its behavior as it progresses in the system.

It is worth highlighting that internally these two approaches are very similar, as they both require the components described above. However, from a user's perspective, the process approach usually feels more natural, as the code produced to implement the entity's routine is similar to the code produced for the real system.

3

State of the Art

The rise in popularity of blockchain technologies has led to an increasing effort of improving prototyping and testing platforms for distributed systems.

In regards to blockchain simulation, it isn't necessarily a new concept, however, it is still rather unexplored, especially simulators designed to be flexible enough to model different protocols, and provide a qualitative analysis of the systems, rather than just testing their performance.

Although there have been several simulators developed to test and validate the performance of specific blockchain systems, such as Blockchain Simulator [16], Bitcoin Network Simulator [3] and eVIBES [13], only a small number of simulators have been developed with the goal of providing extensibility to simulate several families of blockchain protocols.

We will describe extensible blockchain simulators, namely BlockSim [15], BlockSim [1], VIBES [22] and SimBlock [5], since their scope is similar to this work.

3.1 BlockSim: Blockchain Simulator

BlockSim [15] is a simulation framework developed in Python that assists in the design, and evaluation of blockchain protocols. It provides insights regarding how a blockchain system operates and allows the examination of certain assumptions on the chosen simulation models, without incurring the overhead of developing and deploying a real network.

BlockSim uses probabilistic distributions to model random phenomena, such as the time taken to validate a block and network latency to deliver a message, among others. These probability distributions can be specified by the user, in configuration files.

The core of the simulator is its Discrete Event Simulation Engine, providing primitives for event generation, scheduling and execution, and allowing different processes to communicate with each other through events. The Transaction and Node Factories are responsible for creating batches of random transactions and instantiating nodes, respectively. Finally, there is the monitor, whose purpose is to capture metrics during the simulation.

BlockSim uses different detached layers to provide the needed abstraction level to support different blockchain protocols, namely:

- Node layer - specifies the responsibilities and behavior of a node.
- Consensus layer - specifies the algorithms and rules for a given consensus protocol.
- Ledger layer - defines how a ledger is structured and stored.
- Transaction and block layer - specify how information is represented and transmitted. Network layer - establishes how nodes communicate with each other.
- Cryptographic layer - defines what cryptographic functions will be used and how.

For each of these layers, a base model is provided which the user can extend, and BlockSim also provides examples of how these models were extended to simulate Bitcoin and Ethereum1.

3.2 BlockSim: An Extensible Simulation Tool for Blockchain Systems

Although similar in name and language of choice, BlockSim [1] is a different discrete-event simulation framework also developed in Python, with the purpose of exploring the effects of parameterization and design decisions on the blockchain systems, by providing intuitive simulation constructs.

BlockSim has three main modules: the Simulation Module, the Base Module, and the Configuration Module. The Simulation Module is composed of four classes - Event, Scheduler, Statistics, and Main - and is in charge of setting up the simulation, scheduling events, and computing simulation statistics. The Configuration Module acts as the main user interface, where users can select and parameterize the models used in the simulation. Finally, the Base Module consists of the base implementation of the blockchain protocol that will be simulated.

To support a variety of different blockchain protocols, the Base Module is divided according to the following abstraction layers:

- Network layer - defines the blockchain's nodes, their behavior, and the underlying peer-to-peer protocol to exchange data between them.
- Consensus layer - defines the algorithms and rules adopted to reach an agreement about the current state of the blockchain ledger.
- Incentives layer - defines the economic incentives mechanisms adopted by the blockchain to issue and distribute rewards among the participating nodes.

For each of these abstraction layers, an extendable implementation is provided.

3.3 VIBES: Fast Blockchain Simulations for Large-Scale Peer-to-Peer Networks

VIBES [22] is a message-driven blockchain simulator developed in Scala, with the goal of enabling fast, scalable and configurable blockchain network simulations on a single computer, to the performance and security of blockchain systems.

Architecturally, VIBES is composed of Actors. These Actors can have one of three main types: Node, Reducer, or Coordinator. A Node follows a simple protocol to replicate the behavior of a blockchain network, whether it is a full node or a miner node. The Reducer, once the simulation ends, gathers the state of the network and produces an output with the simulation results that the user of the simulator can process.

The Coordinator is essential for providing scalability and speed. This is done by acting as an application-level scheduler that fast-forwards computing time. In practice, each Node estimates how long it will take to complete a certain task, such as mining a block and asks the Coordinator to fast-forward the entire network to that point in time. Once the Coordinator has gathered fast-forward requests from all Nodes, it will fast-forward the entire network to the time referred by the request with the earliest timestamp thus guaranteeing a correct order of execution of tasks.

Finally, VIBES provides a graphical user interface that highlights some extracted metrics and allows users to view a timelapse of the events processed during simulation.

3.4 SimBlock: A Blockchain Network Simulator

SimBlock [5] is a blockchain network simulator, developed in Java, to perform experiments on blockchain protocols using a large number of nodes. SimBlock operates on a message level and allows users to easily change the behavior of nodes, to study their impact on the overall network. The simulator is divided into three components: Network, Node, and Consensus.

- Network component - creates the network topology, given a set of parameters including number of nodes, number of neighbors assigned to each node, and latency tables.
- Node component - defines the state of a node as well as message handling operations.
- Consensus component - defines the rules for creating and validating blocks, as well as extending the local blockchains.

Furthermore, the simulator is accompanied by a simple graphical user interface that allows users to load the output file of a simulation, thus observing the evolution of the local chain of each node.

3.5 Critical Analysis

	Adversarial Behavior	Offline Nodes	Bandwidth Limits	Network Topology	Proof of Stake	Proof of Work	GUI
VIBES [22]	yes	not modeled	not modeled	generated via parameters	not modeled	bitcoin	yes
BlockSim [15]	not modeled	not modeled	throughput calculated from distribution	fully linked	not modeled	bitcoin ethereum	no
BlockSim [1]	not modeled	not modeled	not modeled	fully linked	not modeled	bitcoin ethereum	no
SimBlock [5]	not modeled	not modeled	specify expected available bandwidth	generated via parameters	simple example	bitcoin dogecoin litecoin	yes

Table 3.1: Feature comparison between existing blockchain simulators.

VIBES [22] is scalable, fast, and is capable of simulating Bitcoin-like protocols, however, it requires significant changes to be made to the simulation engine itself [13]. It provides a lot of the functionalities that we consider to be of interest, including a rich graphical user interface. However, there is a severe lack of separation between the code that refers to the simulator and the protocol-specific code, which significantly hinders the extensibility and readability of the implemented protocols. Furthermore, the statistics displayed in the graphical user interface are directly tied to the protocols being simulated - if a user intends to implement a new protocol that extracts different metrics/statistics, changes would need to be made to the GUI in order to display them.

Neither BlockSim [15], BlockSim [1] nor SimBlock [5] model adversarial behavior, neither do they model the ability to define periods of time where a subset of nodes become offline.

VIBES [22], BlockSim [15] and BlockSim [1] do not model Proof of Stake protocols. This is a significant disadvantage to extensibility because as a result, these simulators do not offer abstractions for timers and alarms which are commonly used in proof of stake protocols. Simply extending the simulators with the corresponding events is not enough, since timers can be scheduled and then canceled, incurring significant performance decreases if these events are not handled carefully.

Despite providing a simple proof of stake example, SimBlock [5] still suffers from the aforementioned limitations, as it does not support the definition of timers and alarms, does not offer a clear separation between simulator and protocol-specific code, and does not model adversarial behavior nor the ability to specify intervals where a subset of nodes are offline.

Hence, the simulator and graphical user interface developed in this thesis intend to

address all the highlighted limitations by:

- focusing on modularity and extensibility from the beginning, for both the simulator and the graphical user interface
- allowing users to define two separate implementations for a node, one for a good node and one for a malicious node, and specify when a given node starts behaving maliciously
- providing flexible network topology customization, through both parametrization and specific network topology files
- allowing the user to specify bandwidth limits, enabling the study of the effects caused by, for example, flooding attacks
- providing concrete implementations for both proof of work and proof of stake protocols, as well as a library to abstract and facilitate the implementation of common operations performed by these protocols

4

The Simulator

To address the limitations of the state of the art highlighted in Chapter 3, we developed a framework for specification and simulation of consensus protocols. The framework is divided into two components, the simulator, and the graphical user interface.

Chapter 5 will present the graphical user interface. This chapter will explain, in detail, the simulator component.

We will begin by explaining how the simulator works from a high-level point of view, the different modules that compose the simulator, and how they interact with each other, and finally, we will explain in detail all the functionalities provided by each module and how they were validated.

4.1 Overview

The language chosen to implement the simulator was OCaml, as language features such as *modules* and *functors* provide significant modularity and extensibility, and allow us to define a structured workflow for implementing new protocols in the simulator.

The simulator adopted a *Discrete-Event Simulation Model* meaning, as mentioned in Section 2.3.2, that the state of the system only changes at discrete points in time, where events occur. These events are stored in a queue, ordered by the timestamp when they should be processed.

Entities, in this case, *nodes* create and react to events sequentially, throughout the simulation. The illusion of parallelism comes from the fact that the time when events are processed is based on an internal *clock*, managed by the simulator. This allows for multiple nodes to process and react to events in the same *simulation time*, but in different *real time*.

As mentioned, events are processed sequentially, one at a time. The simulator obtains the next event with the smallest timestamp from the queue and fast-forwards the *clock* to that timestamp. Each event also has a target, which is the entity that should process that event. The simulator then gives control over to the respective entity, that will process the event, change its state and potentially create new events that are added to the queue. The

process is then repeated until there are no more events in the queue, or some predefined stopping condition is reached.

Figure 4.1 illustrates the aforementioned process. At the beginning of the simulation (*simulation time 0*), there are two events in the queue. For the purpose of this example, let's assume these events represent the reception of a message. Each event has a target (a recipient) and a timestamp (when the message should be delivered).

At each iteration, the simulator will pop the event with the smallest timestamp from the queue and fast-forward the clock to match the event's timestamp. It will then let the event's target (the message's recipient) process that event. The handling of the event can originate new events, in this case, a message is sent to E4. Since the newly created event has a smaller timestamp, it will be processed before the event that was already in the queue.

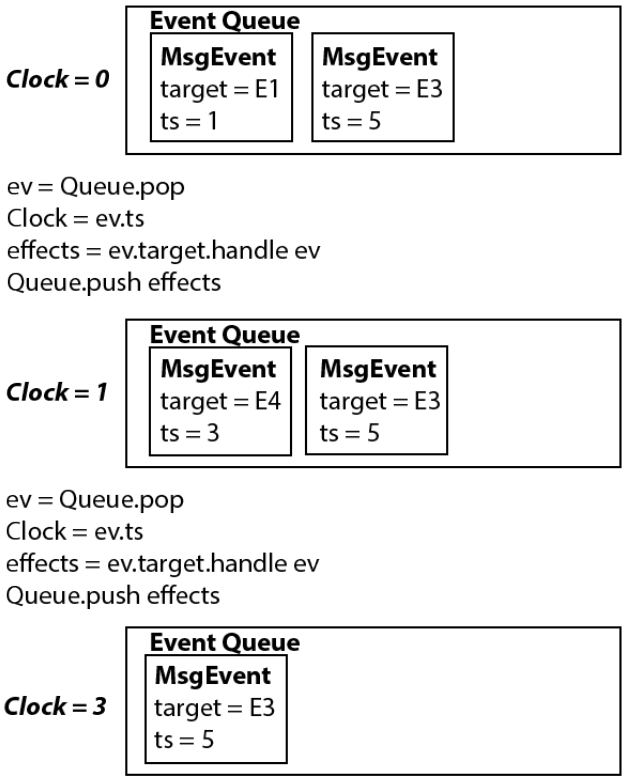

Figure 4.1: High-level illustration and pseudocode of a discrete-event simulation.

4.2 Module Based Architecture

The simulator is composed of different modules. Figure 4.2 illustrates the relations between the different top-level modules.

The *Main* module is the entry point for the simulator and manages the execution of the simulated protocols.

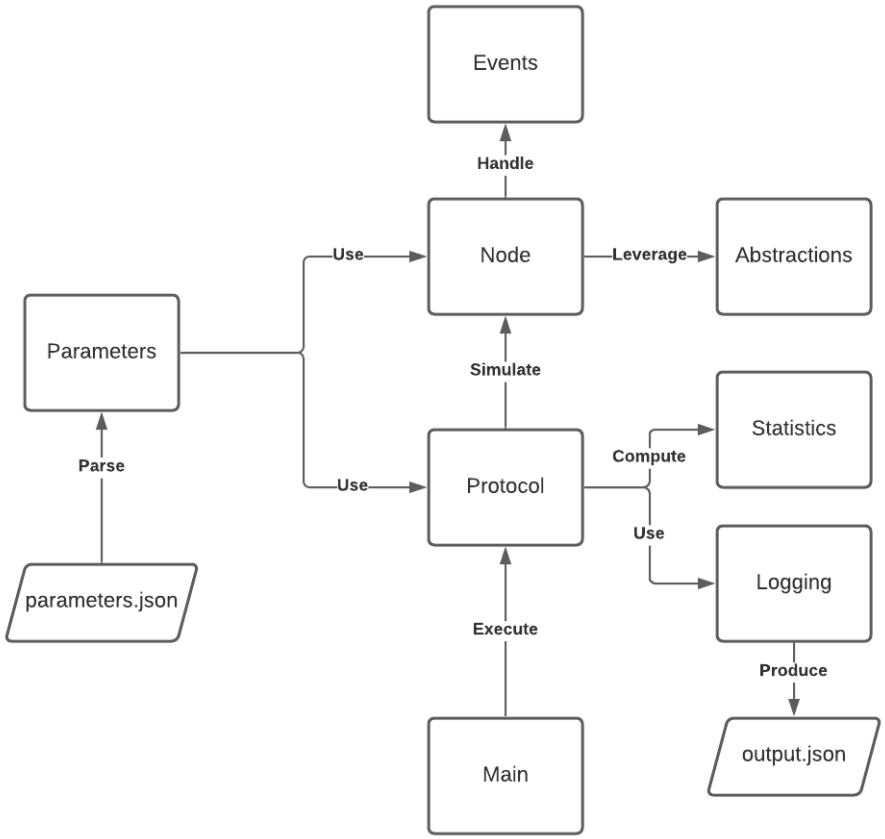

Figure 4.2: Illustration of top-level module interactions.

A *Protocol* is the top-level loop of the simulation. It begins by initializing the different nodes, network topology, event queue, and performs the event handling and delegation previously described in section 4.1.

The *Node* module is a user-defined module, encompassing the state and behavior of an entity in the simulated protocol. Its behavior can depend on a set of user-defined parameters, and it can leverage *Abstractions* in its implementation.

The *Abstractions* module provides several primitives to aid in the development of new simulations. For example, provides useful primitives for proof of stake sortition, proof of work mining, timers and alarms, and message exchanges.

Finally, a protocol leverages the *Statistics* and *Logging* modules to extract metrics and produce a rich output file that can later be processed by the graphical user interface.

The following sections will provide an in-depth explanation of the functionalities encompassed in all modules of the simulator, including lower-level modules.

4.3 Events

The *Events* module defines:

- The expected signature for user-defined *Message* modules.
- The *functor* that creates the *Event* type and module.
- The *functor* that creates the *EventQueue* type and module.

When implementing a new protocol, users will need to define an implementation for the *Message* module, which must have the signature presented in Listing 4.1. The operations contained in the module must behave as follows:

- **t** → the type of the messages that will be exchanged between nodes.
- **to_json** → a function that converts a message of type **t** to stringified JSON format. The implementation of this operation will enable the user to see the contents of the messages being exchanged in the graphical user interface (Chapter5).
- **get_size** → a function for obtaining the size of a message of type **t**. This information is leveraged by the network to compute message arrival times.
- **processing_time** → a function that returns the expected time a node spends processing a message of type **t**. This information is also leveraged by the network to compute message arrival times.
- **identifier** → a function that produces an integer that identifies the content of the message, for example, in a blockchain protocol the identifier of a message of type **t** can be the hash of the contained block. This information is used by the graphical user interface (Chapter 5) when displaying message exchanges.

Listing 4.1: Message module signature

```
module type Message = sig
    type t

    val to_json : t -> string
    val get_size : t -> Size.t
    val processing_time : t -> int
    val identifier : t -> int
end
```

In order to create the *Event* and *EventQueue* modules, the user simply needs to call the *MakeEvent* and *MakeQueue* functors (Listing 4.2).

Listing 4.2: MakeEvent and MakeQueue functor signatures

```
module MakeEvent(Msg : Message) :
            (Event with type msg = Msg.t)

module MakeQueue(Event : Event) :
            (EventQueue with type ev = Event.t)
```

MakeEvent receives the user-defined *Message* module as an argument, and produces an implementation for the *Event* module, whose signature is presented in Listing 4.3.

As indicated by type **t**, the simulator will always have three kinds of events: Message, MintBlock, and Timeout. However, there is a dependency between the Message event and the user-defined message type, hence the use of a functor.

The operations provided by the *Event* module encompass the following behavior:

- **timestamp** → obtain the timestamp in which the event should be processed. For each kind of event, the timestamp is:
  1. *message* → when the message should be received.
  2. *mintblock* → when the node finishes minting a block.
  3. *timeout* → when the node is notified that it timed out.
- **target** → obtain the node that should process the event. For each kind of event, the target is:
  1. *message* → the recipient of the message.
  2. *mintblock* → the minter of the block.
  3. *timeout* → the node that set the timer.
- **create_message** → given the sender, recipient, reception timestamp and message, creates the corresponding *message* event.
- **create_mint** → given a node's id and the timestamp when the minting process is completed, creates the corresponding *mintblock* event.
- **create_timeout** → given a node's id, the timestamp when the node should be notified of the timeout and a corresponding label, creates a *timeout* event.

Listing 4.3: Event module signature

```
module type Event = sig
    type msg

    type t =
    Message of int * int * Clock.t * msg
    | MintBlock of int * Clock.t
    | Timeout of int * Clock.t * timeout_label

    val timestamp : t -> Clock.t
    val target : t -> int option
    val create_message : int -> int -> Clock.t -> msg -> t
    val create_mint : int -> Clock.t -> t
    val create_timeout : int -> Clock.t -> timeout_label -> t
end
```

Finally, *MakeQueue* receives the resulting *Event* module as an argument, and produces an implementation for the *EventQueue*, whose signature is presented in Listing 4.4. Once again, the use of a functor derives from the inherent dependency between the queue and the contained events.

The resulting *EventQueue* will store tuples, containing the timestamp when each event should be processed, and the corresponding event. Insertions and removals will always maintain the queue ordered by the timestamp, from lowest to highest. The exposed operations behave as follows:

- **get_event** → removes the event at the head of the queue (smallest timestamp) and returns it. Raises an exception if the queue is empty.
- **add_event** → performs an ordered insertion of a new event into the queue.
- **has_event** → checks if there are any elements in the queue.
- **cancel_minting** → removes an existing *mintblock* event from the queue. We elaborate on the necessity for this function in Section 4.12.1.
- **clear** → empties the event queue. This operation is required when executing multiple batches of simulations (Section 4.9).

Listing 4.4: EventQueue module signature

```
module type EventQueue = sig
    type ev

    type elem = (Clock.t * ev)

    val get_event : unit -> elem
    val add_event : ev -> unit
    val has_event : unit -> bool
    val cancel_minting : int -> unit
    val clear : unit -> unit
end
```

4.4 Parameterization

The simulator receives a JSON file as input (`parameters.json`), containing the parameterizations for the simulation. The available parameters are divided into three categories: *General Parameters*, *Network Parameters* and *Protocol Parameters*. Optionally, a user can provide another JSON file (`topology.json`), that allows for a more fine-tuned configuration of the simulation.

4.4.1 General Parameters

As the name indicates, *General Parameters* are parameters that refer to the general definition of the simulation and the nodes being simulated. 19 parameters fall under this category:

1. **num-nodes** → the number of nodes that will be simulated.
2. **end-block-height** → a stopping condition for blockchain protocols. The simulation terminates when one node possesses a local chain with this height.
3. **timestamp-limit** → a general stopping condition for all protocols. The simulation terminates when the internal *simulation clock* reaches this timestamp.
4. **verbose-output** → a flag for producing complete logs, containing all processed events and the produced statistics. If this flag is set to false, the output of the simulation will be limited to statistics.
5. **use-topology-file** → a flag for using the parameterization provided in the topology file. The topology file allows for more fine-tuned parameterization of each node, individually (Section 4.4.4).

6. **topology-file** → the path where the topology file is stored.
7. **number-of-batches** → how many times to repeat the simulation with different pseudo-random seeds.
8. **seed** → the base seed for all rng-based operations in the simulator. When running multiple batches, each batch will use a different seed, obtained through this seed. This allows for reproducibility of results.
9. **pow-target-interval** → the target interval between blocks for proof of work protocols. This parameters is used to compute the initial block difficulty.
10. **avg-mining-power** → the average mining power of a node, in hashes per millisecond.
11. **stdev-mining-power** → the standard deviation for a node's mining power, in hashes per millissecond.
12. **avg-coins** → the average coins of a node.
13. **stdev-coins** → the standard deviation for a node's coins.
14. **reward** → the reward for creating a blocks. Although the simulator allows the definition of a more complex reward function to be defined by the user.
15. **bad-nodes** → the number of malicious nodes in the simulation.
16. **become-bad-timestamp** → the timestamp when the previously mentioned nodes start acting maliciously.
17. **offline-nodes** → the number of nodes that can become temporarily offline during the simulation.
18. **become-offline-timestamp** → the timestamp when the previously mentioned nodes become offline.
19. **become-online-timestamp** → the timestamp when the previously mentioned nodes become online again.

4.4.2 Network Parameters

The *Network* category encompasses all parameters required to pseudo-randomly generate the network's topology, as well as computing message arrival times. The following are the parameters that belong to this category:

1. **num-regions** → the number of distinct regions. In the context of the simulator, a region has a set of network properties such as latencies and bandwidths, and each region has a different set of properties. Each node can only belong to a single region.

2. **num-links** → the number of neighbors that each node has a communication link to.

3. **latency-table** → a table with `num-lines=num-columns=num-regions`. For each line **x** and column **y**, `latency-table[x][y]` provides the average latency expected when sending a message from a node in region **x** to a node in region **y**.

4. **region-distribution** → a list with *num-regions* elements, specifying the fraction of nodes that are assigned to each region. The sum of all elements in the list must equal to 1.

5. **download-bandwidth** → a list with *num-regions* elements, specifying for each region what is the download bandwidth per link.

6. **upload-bandwidth** → a list with *num-regions* elements, specifying for each region what is the upload bandwidth per link.

7. **limited-bandwidth** → the value assigned to this flag has the following impact:

   a) **true** → the simulator handles the previously defined upload and bandwidth values as the maximum available bandwidth per link. A detailed explanation of this functionality is presented in Section 4.7.3.

   b) **false** → the previously defined upload and download bandwidths are assumed to always be available, and nodes can leverage the complete specified bandwidth values when sending any message.

4.4.3 Protocol Parameters

This category of parameters includes only user-defined parameters, that affect the protocol being simulated.

By default, there are no parameters in this category. However, common parameters to insert in this category are the size of a block, number of committee members, and required votes for consensus, for example.

4.4.4 Topology File

The topology file is an optional input for the simulator. It allows users to define specific network topologies and individually parameterize each node being simulated, as opposed to pseudo-randomly generating these values from higher-level parameters.

The contents of this file consist of a list of nodes, where each element in the list is a JSON object that parameterizes an individual node. An example a parameterization of a node is included in Listing 4.5, and it contains the following information:

- **id** → the identifier of the node.

- **region** → the region to which the node is assigned.

- **stake** → the initial stake of the node.
- **hPower** → the hashing power of the node.
- **links** → a list containing the identifiers of this node's neighbors.
- **offline** → the period of time where the node is offline.
- **malicious** → if the node is malicious, and the timestamp when it starts behaving maliciously.

Listing 4.5: Example parameterization of a node via the topology.json file

```
{
    "id":1,
    "region":1,
    "stake":100,
    "hPower":100,
    "links":[2,3,4],
    "offline": {
        "from":0,
        "to":500
    },
    "malicious": {
        "isbad":true,
        "start":23
    }
}
```

4.5 Statistics

A large feature of the simulator is producing metrics and statistics to be analyzed using the graphical user interface (Chapter 5) at a later time.

The simulator extracts a set of statistics, by default (Section 4.5.1), however it is extremely simple to define new statistics and metrics to be extracted (Section 4.5.2).

4.5.1 Default Statistics

By default, the simulator extracts four statistics, which we deemed relevant for any consensus protocol.

The simulator computes the **average time to reach consensus**. The simulator assumes consensus is reached when the *state* record of a Node's type changes.

The network module of the simulator keeps track of the **total number of messages exchanged** and the **total amount of megabytes exchanged**. However, these values are only produced if the protocol does **not** use the gossip abstraction provided by the simulator, as will be further explained in Section 4.7.4.

Finally, to give users an idea of workload distribution, the simulator produces a list with the **number of events processed per node**.

This data and any user-defined data that is produced by the simulator is automatically processed and converted into graphs when using the graphical user interface (Chapter 5).

4.5.2 Extracting more Statistics

There was a focus on making the process of extracting custom statistics and metrics as simple as possible. To that end, OCaml's functors were once again leveraged to provide an extensible and compositional process of adding more statistics.

Statistics are gathered and computed by modules. The signature for a *Stats* module is included in Listing 4.6. An implementation of this type should provide the following:

1. **t** → the type of the values being processed. In general, these are integers or floating-point numbers.
2. **process** → a function that receives the identifier of a node, and a value of type **t**. This function should perform any necessary state changes to store the received value.
3. **get** → uses the state to compute the final statistics, and returns them in JSON format.
4. **clear** → resets the state of the module. Used when we start a new batch (Section 4.9).

Listing 4.6: Stats module signature

```
module type Stats = sig
    type t

    val process : int -> t -> unit
    val get : unit -> string
    val clear : unit -> unit
end
```

Although a user can implement a module with the specified signature by hand, the simulator provides functors that create *Statistic* modules for common operations (Listing 4.7).

Each call to one of the functors produces a module with an internal state. This means that if we want to compute, for example, averages for two different metrics, we would need to call the *Average* functor twice in order to obtain two distinct modules.

The existing functors are described below:

- **Average** → compute the average of all values passed in calls to the *process* function.
- **Median** → compute the median of all values passed in calls to the *process* function.
- **Percentile95** → compute the 95th percentile of all values passed in calls to the *process* function.
- **Min** → keep track of the smallest value passed in a call to the *process* function.
- **Max** → keep track of the largest value passed in a call to the *process* function.
- **CountAll** → sum all values passed in calls to the *process* function.
- **CountPerNode** → sum all values passed in calls to the *process* function, seperating the different counters by the node that called the function. As opposed to the previously described modules that produce a single value, this module produces a list of values (one value per node).
- **Compose** → receives two *Statistic* modules as arguments, and produces a new module whose *get* function returns a JSON that combines the result of the *get* function defined in both modules provided in the arguments.

Listing 4.7: Functor module signatures

```
module Average(X:Arg)                : Stats
module Median(X:Arg)                 : Stats
module Percentile95(X:Arg)           : Stats
module Min(X:Arg)                    : Stats
module Max(X:Arg)                    : Stats
module CountAll(X:Arg)               : Stats
module CountPerNode(X:Arg)           : Stats
module Compose(A:Stats)(B:Stats)     : Stats
```

As it can be seen in Listing 4.7, all functors, with the exception of *Compose*, receive a module of type *Arg* as argument. This module contains the following information:

- **label** → the label that will be associated with the statistic/metric produced by the module. It will be used as in the legend of a graph in the graphical user interface.
- **use_intervals** → a boolean value which has the following impact on the produced module:
  - **true** → this option facilitates the computation of some time-based metrics. For example, creating an *Average* module with `use_intervals=true`, and calling the *process* function with the timestamp of the creation of each block would give us the average time elapsed between the creation of any two consecutive blocks.

  - **false** → the computations are performed on the values provided in the *process* calls.

- **format** → setting this flag to **true** causes the module to interpret the values as *millisseconds*, and converts the resulting value to *seconds*. Keeping this flag as **false** prevents the module from having any assumptions of the value's format.

Finally, to provide a better understanding of all functionalities and usage of the statistics module of the simulator, Listing 4.8 includes an example.

Listing 4.8: Usage example of the Statistic's functors

```
module Args1 = struct
    let label = "avg1"
    let use_intervals = false
    let format = true
end
module Args2 = struct
    let label = "avg2"
    let use_intervals = true
    let format = false
end
module Args3 = struct
    let label = "counter"
    let use_intervals = false
    let format = false
end

module Avg1    = Statistics.Make.Average(Args1);;
module Avg2    = Statistics.Make.Average(Args2);;
module Counter = Statistics.Make.CountPerNode(Args3);;
module Final   = Statistics.Compose
                 (Counter)
                 (Statistics.Compose(Avg1)(Avg2));;

Avg1.process 1 1000;;
Avg1.process 3 1100;;
Avg1.process 3 900;;
Avg2.process 1 0;;
Avg2.process 2 100;;
Avg2.process 1 150;;
Counter.process 1 10;;
Counter.process 1 32;;
```

```
Counter.process 3 1;;

Final.get ();;
(* Output: {avg1: 1, avg2: 75, counter: [42,1]} *)
```

4.6 Logging

The *Logging* module contains all the necessary operations required to produce the simulator's output file, in JSON format, which is then processed by the GUI (Chapter 5).

The structure of the log is an extended version of the format used by SimBlock [5] since a portion of the GUI is also built on top of SimBlock's graphical user interface.

These changes can be summarized as follows:

1. The log entries corresponding to message exchanges were extended to include the contents of the message. The content is obtained from the `to_json` function defined by the user in the *Message* module (Listing 4.1)
2. The simulation parameters are now included in the log.
3. When using the stake-based sortition abstraction provided by the simulator (Section 4.12.2), the log will include which nodes belong to the committee for each level of the chain, and which nodes are the leaders (or proposers) for each level.
4. The produced statistics/metrics are now included in the log.

We will now describe, in detail, the structure of the logs produced by the simulator. Each log entry has two top-level keys: **kind** and **content**.

The **kind** key can have one of 10 values, which include the content specified below:

1. **parameters** → log the parameters used in the simulation. Contains a key-value pair for each *Protocol Parameter* defined by the user.
2. **add-node** → log the creation of a node in the simulator. Contains the timestamp when the node was created, the id of the node, and the region to which the node is assigned.
3. **add-link** → log the creation of a link between two nodes (neighbors). Contains the timestamp when the link was created, and the id of both nodes.
4. **flow-message** → log the exchange of a message between two nodes. Contains the timestamp when the message was sent, the timestamp when it was received, the id of the sender, the id of the recipient, and the JSON representation of the message's contents.

5. **add-block** → log a change in the node's state. Contains the timestamp when the state change occurred, the id of the node whose state changed, the id of the block that is now the head of the node's local chain, and the id of the node that created the block.

6. **node-committee** → log that a node belongs to the committee for a given level. Contains the timestamp, the id of the node, and the corresponding level.

7. **node-proposer** → log that a node is a proposer (leader) for a given level. Contains the timestamp, the id of the node, and the corresponding level.

8. **create-block** → log the creation of a block. Contains the timestamp when the block was created, the id of the node that created the block, and the id assigned to the block.

9. **statistics** → log the obtained statistics. Contains a key-value pair for each statistic produced by the simulator.

10. **per-node-statistics** → log the obtained statistics. This kind of log entry differs from the previous one, as it should only be used for statistics that produce values assigned to individual nodes (for example, *CountPerNode* described in Section 4.5.2).

For reference, Appendix A contains an example of all kinds of log entries produced by the simulator.

Finally, it is worth noting that a user of the simulator is never required to explicitly log any operation, as all logging operations are automatically handled by the simulator.

4.7 Network

The *Network* module is responsible for all operations regarding topology generation and message exchanges. It models important behavior such as computing message arrival times and the usage of bandwidth limits and provides an abstraction for gossip.

4.7.1 Topology Generation

At the start of the simulation, the *Network* will assign each node to a region and assign a set of neighbors to each node, creating the respective communication links.

If the user does **not** use a topology file, the region, and neighbor assignments will be performed by the simulator, in a pseudo-random fashion, based on the specified network parameters (Section 4.4.2).

This pseudo-random assignment is performed similarly to the process described by the authors of SimBlock [5].

Regions are numbered from 0 to `(num-regions)-1`. The `region-distribution` parameter (Section 4.4.2) contains a list where index **i** indicates the fraction of nodes that

belong to region **i**. A list of length equal to the total number of nodes is then computed, where each index **j** contains the region assigned to node **j**.

Links are bidirectional, meaning that if a link is added from node **i** to node **j**, then a matching link is also added from node **j** to node **i**. **Links** are stored in a two-dimensional array, where row **i** contains the identifiers of all neighbors of node **i**. The process of computing the set of neighbors for a given node **n** is as follows:

1. Shuffle a list with the identifiers of all nodes, except for **n**. These are the potential neighbors.
2. Iterate through the list of potential neighbors, until the specified number of neighbors (links) is assigned to node **n**. A potential neighbor **k** is assigned to node **n** if both of the following conditions are met:
   a) There is no existing link from **n** to **k**, nor from **k** to **n**.
   b) Neither node has surpassed the parameterized number of links.

It is worth noting that every shuffle operation used is *seeded*, to ensure reproducibility.

4.7.2 Message Exchanges

One of the major functionalities provided by the *Network* module is the **send** function. This operation can be called by a node, providing its own id, the id of the node to whom they wish to send a message, and the respective message. The function will then compute the timestamp when the message will arrive at the destination, create the corresponding *event* and add it to the *EventQueue*.

The arrival timestamp for a message is computed as follows:

$$arrival\_time = current\_time + latency + upload\_time + pending\_upload\_time$$

Where:

1. **current_time** is the time when the node called the *send* function.
2. **latency**, or propagation delay, is the time taken for a packet to travel from the sender to the recipient. It is computed from the parameterized `latency-table` according to a Pareto Distribution [24].
3. **upload_time** is the amount of time required for the sender to upload the message to the link. It is obtained by computing $msg\_size/link\_bandwidth$, where $link\_bandwidth$ is the minimum between the upload bandwidth of the sender, and the download bandwidth of the recipient.
4. **pending_upload_time** depends on the value assigned to the *limited-bandwidth* parameters. If the value is `false`, **pending_upload_time** will be zero. Otherwise, it is computed as described in section 4.7.3.

4.7.3 Bandwidth Limits

At a high level, the *limited-bandwidth* parameter enables a per-link queueing effect, limiting the number of messages in transit to a single message per link.

At a lower level, as mentioned in the previous section, this affects the computation of the message arrival time performed by the **send** function, namely the **pending_upload_time**. The *Network* will then keep track of the timestamp when the outgoing message queue for each link between nodes becomes free, and use this information to compute the **pending_upload_time**, which refers to the time a node must wait until its outgoing queue is empty and a new message can be sent.

Finally, when calling the **send** function, the arrival timestamp is computed following what was previously described. Furthermore, the *Network* updates the time when the corresponding link becomes available, setting it to:

$$MAX(link\_available\_timestamp, current\_time) + upload\_time$$

4.7.4 Gossip

A common form of message propagation in these protocols is **gossip**. The basic version of gossip consists of each node forwarding a non-duplicate received message to all of its known neighbors, except for the neighbor that sent the message.

However, from a simulation point of view, gossip leads to a significant performance bottleneck.

Let us suppose we have a network with 100 nodes, each possessing 8 neighbors. When a node sends a message that gets propagated through the network via gossip, there can be up to $8 * 100 = 800$ messages exchanged. This leads to the generation of 800 message receival events when in reality only 100 of those events can lead to state changes (the first time each node receives the message).

The impact is more noticeable when the number of nodes and/or the number of neighbors increases. Furthermore, it is extremely noticeable in Practical Byzantine Fault Tolerant consensus protocols, since these usually involve the exchange of a larger amount of messages. For context, a single round of Algorand, with 100 nodes, 8 neighbors each, 50 committee members, and 1 proposer, with no offline nodes and no malicious nodes, is expected to generate up to $8 * 100 * 2 + 2 * 50 * 100 * 8 = 81600$ events, when only $100 * 2 + 2 * 50 * 100 = 10200$ of those events would lead to state changes.

For this reason, an attempt was made to provide a **gossip** abstraction. The abstraction provides significant performance increases, but it also has a significant amount of limitations in the scenarios where it can be used.

The abstraction consists of computing the shortest path between each pair of nodes in the network, using an adapted version of the *Floyd Warshall's Algorithm*, presented in Listing 4.9. The result is a table containing, for each row **i** and column **j**:

1. The sum of the latencies of all links that will be traversed to send a message from **i** to **j**.

2. A list containing the link bandwidth for each link traversed to send a message from **i** to **j**.

3. The number of links traversed to send a message from **i** to **j**.

Listing 4.9: Pseudo code of the Floyd Warshall's Algorithm adapted for the simulator

```
shortest_paths = MakeTable num_nodes num_nodes
for i=0 to num_nodes-1 do
    for j=0 to num_nodes-1 do
        if i=j then
            shortest_paths[i][j] = (0,[],0)
        else
            shortest_paths[i][j] = (-1,[],-1)
    do
do
for i=0 to num_nodes-1 do
    foreach neighbor in neighbors[i] do
        l = get_latency i neighbor
        b = get_bandwidth i neighbor
        shortest_paths[i][neighbor] = (l,[b],1)
    done
done
for k=0 to num_nodes-1 do
    for i=0 to num_nodes-1 do
        for j=0 to num_nodes-1 do
            (l1,b1,n1) = shortest_paths[i][k]
            (l2,b2,n2) = shortest_paths[k][j]
            (l,b,n)    = shortest_paths[i][j]
            if l1>-1 and l2>-1 then
                if l > l1 + l2 or l=-1 then
                    shortest_paths[i][j] = (l1+l2,b1@b2,n1+n2)
        done
    done
done
return shortest_paths
```

The **gossip** function provided by the *Network* module will then use the computed `shortest_paths` table as follows:

1. A node calls the **gossip** function, providing its own id and the message to be propagated to the network.

2. For each node **i** in the network, the function will compute the earliest time when the message will arrive at node **i**, and add the corresponding event to the *EventQueue*. This computation consists in:

    a) Computing the *upload_time* for each link traversed, using the size of the message and the list of bandwidths stores in the *shortest_paths*.

    b) And adding the *current_timestamp* and the accumulated *latencies* (also stored in the *shortest_paths*).

As mentioned previously, the provided gossip abstraction can **only** be used under the following restrictive conditions, otherwise, it would lead to unrealistic results:

- The *limited-bandwidths* parameters is set to **false**.
- Nodes do not become offline at any point during the simulation.
- All regions share the same *upload-bandwidth* and *download-bandwidth* parameter values.

Although the conditions are very restrictive, we decided to include the abstraction as it significantly speeds up the simulation in use-cases where the conditions can be met. For example, when simulating Algorand with 500 nodes, 50 committee members, 8 neighbors, and a stopping condition of 100 blocks, while respecting the aforementioned restrictions, we obtained the results presented in Table 4.1.

	With Gossip Abstraction	Without Gossip Abstraction
Real-Time Elapsed	680s	12045s
Majority SoftVotes	10.2s	10.24s
Consensus Reached	10.39s	10.47s

Table 4.1: Results from simulating 500 nodes of Algorand with and without the gossip abstraction.

The resulting metrics for the average time to see a majority of softvotes and the average time to reach consensus are very similar, while the time required to run the simulation is significantly smaller.

4.8 Timers and Alarms

As mentioned in Chapter 3, a significant limitation of the state of the art is the lack of support for timers/alarms. This prevents the implementation of a majority of Practical Byzantine Fault Tolerant consensus protocols.

To address this limitation, this simulator supports *timeout* events. Nodes can have multiple timers, with each node having its own set of timers, uniquely identified by a *label*.

The *Timer* module provides a functor for creating a timer. The arguments of the functor are the *Event* and *EventQueue* modules, as can be seen in Listing 4.10. The resulting module has the signature presented in Listing 4.11.

Listing 4.10: Functor for creating a timer

```
module Make(Event : Simulator.Events.Event)
           (Queue : Simulator.Events.EventQueue
                    with type ev = Event.t)
           : Timer
```

Listing 4.11: Signature of the Timer module

```
module type Timer = sig
    val set : int -> Simulator.Clock.t -> string -> unit
    val cancel : int -> Simulator.Events.timeout_label -> unit
    val expired : int -> Simulator.Clock.t ->
                  Simulator.Events.timeout_label -> bool
    val clear : unit -> unit
end
```

The resulting module provides the following operations:

- **set** → creates and schedules a timeout event for a node, given a node's id, the timer's duration, and the label for the timer.
- **cancel** → cancels a previously set timer, given the id of a node and the timer's label.
- **expired** → given the id of a node, a timer's label and a timestamp, checks if the corresponding timer has already expired at the specified timestamp.
- **clear** → removes all existing timers.

In a discrete-event simulator, the direct approach to implement the specified **set** and **cancel** operations is to have the **set** operation immediately add the *timeout* event to the event queue, and then if the **cancel** operation is called before the event is processed, it would remove the *timeout* event from the queue. There is, however, a significant performance bottleneck in the described **cancel** operation. Since the event queue is an ordered

doubly-linked list, removing events from the queue would have a time complexity of O(n) in the worst case.

This bottleneck was very noticeable in the Algorand simulation, as most timers are expected to be canceled. To address this bottleneck, the following solution was implemented:

1. The **set** function is implemented as described, immediately adding the *timeout* event to the event queue.
2. The *Timer* module stores, in its state, an array of hash maps. Each index **i** of the array, contains a hash map where the keys are the labels of node **i**'s timers, and the values are the timestamps when each timer expires.
3. The **expired** function performs the verification using the previously described array.
4. The new **cancel** function simply sets the corresponding timestamp in the hash map to -1. The top-level simulation loop, when handling a timeout event, will only forward the event to the corresponding node if it has not expired, otherwise, the event will be discarded.

The impact of this optimization can be seen in table 4.2. When simulating the Algorand protocol with 500 nodes, until a stopping condition of 250 blocks, there was a reduction of approximately 6 minutes and 30 seconds in execution time, which translates to a 19.3% decrease.

	Before Optimization	After Optimization
Real-Time Elapsed	2010s (33m30s)	1621s (27m1s)

Table 4.2: Real time elapsed when simulating Algorand, before and after the optimization.

4.9 Batches

An important aspect of discrete-event simulation is that when the simulator possesses components that are based on pseudo-random variables, a single execution of a simulation isn't enough to allow users to extract conclusions about the results, as these can sometimes be directly tied to the inputs that were used.

To this end, the simulator utilizes batches, running multiple simulations using different pseudo-random seeds, to mitigate the impact that specific seeds might have on the outputs of the simulation. If a user parameterizes the simulator to run 5 batches (`number-of-batches=5`), then it will execute the same simulation 5 times, with different *seeds*.

In order to maintain reproducibility, which is a significant advantage of simulation, the *seed* specified in the parameters is used to extrapolate the individual *seeds* for each batch.

4.10 Node

The main component of the simulation is the *Node*, which will contain all the logic for the consensus algorithm executed by each node in the network.

The simulator defines a template for a node's type, presented in Listing 4.12. It must have 5 mandatory fields, two of which can have user-defined types.

The *state* field is in charge of storing the object of consensus. In the case of blockchain protocols, the type **'b** will be a *Block*. The *data* field can contain any amount of arbitrary data required by the node, in order to execute the desired protocol.

Listing 4.12: The template for a node's type.

```
type ('a, 'b) template = {
    id : int;
    region : Abstractions.Network.region;
    links  : Abstractions.Network.links;
    mutable state : 'b;
    mutable data : 'a;
}
```

To simulate a protocol, the user must implement a module with one of two signatures: *AbstractNode* or *BlockchainNode*.

An implementation of the *AbstractNode* signature must provide the following:

- **value** → the type of values for which consensus will be reached. It is the type that gets assigned to the *state* field.
- **node_data** → the protocol specific data stored by the node. It is the type assigned to the *data* field.
- **t** → the type of a node. It consists in applying the types **value** and **node_data** to the *template*.
- **init** → create the initial data of a node.
- **handle** → given the current data of the node and an event, returns the data that results from processing the event.
- **compare** → check if two nodes are equal.
- **state** → returns the contents of the node's *state* field.
- **state_id** → returns an integer that uniquely identifies the value stored in the node's *state* field.
- **parameters** → returns a JSON string containing the protocol's parameters.

A *BlockchainNode* is an extension of the *AbstractNode*. It restricts the type of the *state* field to a block, and must provide an additional operation - **chain_height** - to obtain the height of the node's local chain.

It is worth noting that, for either case, the **compare**, **state**, **state_id** and **parameters** functions don't need to be implemented if the user includes the result of *MakeBaseNode* in its node implementation.

Furthermore, an alternative way of implementing a node is presented in Section 4.13.3

4.11 Initializer

One of the modules the user must implement is the *Initializer*. The initializer is in charge of kickstarting the simulation, by creating the initial events that get added to the *EventQueue*.

The user must implement an initializer with the signature presented in Listing 4.13. It must contain the type of the nodes being simulated, the type of the events used in the simulation, and an **init** function, which receives a hash table with each node's initial data and returns a list containing the events required to start the simulation.

For example, to kickstart the Bitcoin simulation, a node is chosen at random to mint the genesis block. In Algorand, a *timeout* event is scheduled for each node to begin the first round of the protocol.

Listing 4.13: The signature of the Initializer module.

```
module type Initializer = sig
    type node
    type ev

    val init : (int, node) Hashtbl.t -> ev list
end
```

4.12 Key Functionalities and Abstractions

This section will present the most relevant functionalities and abstractions provided by the simulator.

4.12.1 Proof of Work Minting

The simulator offers an abstraction for proof of work minting, in the module *PoW*.

When implementing a proof of work protocol, users can instantiate a *PoW* module by calling the provided functor, with the *Event*, *EventQueue* and *Block* modules as arguments. The resulting module's signature is included in Listing 4.14, as provides the following operations:

- **init_mining_power** → if a topology file is being used, assigns the specified hashing power to each node. Otherwise, uses the parameterized average hashing power and standard deviation of hashing power to pseudo-randomly assign an hashing power to each node.
- **start_minting** → abstraction for the minting process. Computes an estimation of the time it will take to discover a valid *nonce* for a block, and adds the corresponding *MintBlock* event to the *EventQueue*.
- **stop_minting** → removes a previously created *MintBlock* from the *EventQueue*.
- **get_mining_power** → given the id of a node, returns its hashing power.
- **total_mining_power** → returns the combined hashing power of the entire network.

Listing 4.14: The signature of the PoW module.

```
module type PoW = sig
    val init_mining_power : unit -> unit
    val start_minting : int -> 'a Simulator.Block.t -> unit
    val stop_minting : int -> unit
    val get_mining_power : int -> int
    val total_mining_power : unit -> int
end
```

The estimation of the time it takes a node to find a valid *nonce* for a block is done similarly to SimBlock [5], drawing values from a geometric distribution. From the distribution, we can obtain a pseudo-random estimate of the number of attempts required to succeed in a trial. This can be directly translated to the number of hashes required to find the correct *nonce*, and diving the obtained value by the hashing power of a node will result in the time it needs to mine in order to find a valid *nonce*.

We then validated the abstraction with the following process:

1. Define a target block interval of 10000 milliseconds.
2. Create 10 nodes, having an average hashing power of 100 hashes per millisecond, and a standard deviation of 10 hashes per millisecond.
3. With different *seeds*, allow each node to produce an estimate of the time it will take to find a valid *nonce*, 1000 times.
4. Repeat the process with 100, 1000, and 5000 nodes.

The results are tabulated in Table 4.3. As we can see, regardless of the number of nodes, approximately one node will succeed in finding the valid *nonce* within the target interval. Nodes can take longer to find the *nonce*, but it is also possible for multiple nodes to find it before the target interval, as is in reality.

	100 nodes	1000 nodes	5000 nodes
Average successes within interval	1.01	1.04	1.02

Table 4.3: Average successes within the interval, after 1000 iterations, varying the number of nodes between 100, 1000 and 5000.

As a final remark regarding the abstraction, the usage of the geometric distribution is valid because the phenomenon being modeled is a sequence of independent trials, where each trial only has two outcomes (either we found the correct *nonce*, or not) and the probability of success is the same for every trial.

4.12.2 Proof of Stake Sortition

A common operation used in proof of stake protocols is stake-based sortition, where a node or subset of nodes is selected to perform a certain task based on their stake.

The simulator offers a functor that creates a *PoS* module, when provided with the *Logger* and *Block* modules. The resulting *PoS* module provides the following operations:

- **in_committee** → given the *id* of a node, the *head* of the node's chain, the expected *size of the committee* and a list of arbitrary user-defined parameters, produces a **credential**, if the node belongs to the committee. Otherwise, returns **None**.
- **is_proposer** → given the *id* of a node, the *head* of the node's chain, the expected *number of proposers/leaders* and a list of arbitrary user-defined parameters, produces a **credential**, if the node belongs to the committee. Otherwise, returns **None**.
- **valid_proposer** → given the *id* of a node, the *head* of the node's chain, the expected *number of proposers/leaders* and a list of arbitrary user-defined parameters and a *credential*, returns whether or not the credential is valid and its owner is in fact a proposer.
- **valid_committee_member** → given the *id* of a node, the *head* of the node's chain, the expected *size of the committee* and a list of arbitrary user-defined parameters and a *credential*, returns whether or not the credential is valid and its owner is in fact a member of the committee.
- **proposer_priority** → this operations receives the same arguments as the **valid_proposer** function. It returns the priority associated with a proposer, which is used as tie-breaker in protocols where there can be multiple proposers/leaders in the same round (for example, Algorand).

The list of user-defined parameters received by each function enables the creation of different committees and leaders for different protocols, as well as for different phases of the same protocol. For example, in Algorand, every *round-period-step* combination has a different committee.

Internally, the module uses a **sortition** function to perform the verifications and generate the credentials. This function has the following behavior:

1. Receive as arguments a block, the number of selections to be performed, and a list of arbitrary parameters.

2. The list of parameters is used as a *seed* for the operation, whereas the block is used to obtain the balances of all nodes in the network.

3. Until *number of selections* have been made, repeat:

    a) Select a random coin from the total number of coins in the network.

    b) Produce a credential for the owner of the selected coin.

4. Returns the list of generated credentials.

A **credential** is an internal type, presented in Listing 4.15, used by the *PoS* module to perform assignments and validations of leaders and committee members.

Listing 4.15: The type of a PoS credential.

```
type credential = Credential of node_id * block_id *
                  num_selections * priority * params
```

4.12.3 Mining Pools and Stake Pools

The simulator allows for mining pools or stake pools to be simulated in a very simple way.

When creating a topology file (Section 4.4.4), a user can simply specify that a node's hashing power or stake is the combined value of all nodes in the pool. This allows for a very high-level abstraction of mining pools and stake pools.

If more detail is required, a user can increase the number of *regions*, and parameterize the region's properties to reflect the pool, by adding low latencies, high bandwidths, etc. Furthermore, there are no restrictions on the behavior implemented in a node, meaning that a user can implement the message exchanges that happen internally in any given pool.

4.12.4 Blocks and Block Rewards

As mentioned in the introductory chapter (Chapter 1), the focus of the simulator is on the consensus portion of the protocols. Hence, it was decided that modeling transactions was outside of the scope of the simulator.

The simulator offers an abstraction for blocks, namely block creation, genesis blocks, and block rewards. In the simulator, blocks have the type presented in Listing 4.16. The block's *header* contains information relevant to most blockchain protocols, as well as some information that is used internally by the simulator. A block also contains additional, arbitrary, user-defined contents, as some protocols require more information to be stored in the blocks.

The block's header contains the following information:

- **id** → the identifier of a block. A user can think of the **id** as the hash that uniquely identifies the block's contents.
- **height** → the distance from the block to the genesis block.
- **minter** → the identifier of the node that created the block.
- **parent** → the block's parent (the block that this block extends).
- **timestamp** → the timestamp when the block was created.
- **balances** → this is where we abstract the transactions. It is an array containing the coins owned by each node.
- **difficulty** → the difficulty of the block. This field is calculated by the simulator, derived from the total mining power of the network and the target interval between blocks. It is only used for performing proof of work operations, as described in Section 4.12.1.
- **total_difficulty** → the sum of the difficulties of the chain since the genesis block. This field is calculated by the simulator, and is only used for performing proof of work operations, as described in Section 4.12.1.

Listing 4.16: The type of an abstract block.

```
type block_header =
{
    id                  : block_id;
    height              : int;
    minter              : node_id;
    parent              : node_id option;
    timestamp           : Clock.t;
```

```
    balances          : balances;
    difficulty        : int;
    total_difficulty  : int;
}

type 'a t =
{
header    : block_header;
contents  : 'a;
}
```

Providing the *Logger*, *BlockContents* and *BlockRewards* modules as arguments to the functor presented in Listing 4.17, results in a *Block* module with the signature described in Listing 4.18.

The *BlockContents* module defined by the user includes the definition of a type **t**, representing the protocol-specific block contents. In case there is no additional information required, the type is an empty record.

The *BlockRewards* module is also a user-defined module, that includes a single function, that receives the creator of a block, the current list of balances, and returns the updated list of balances.

Listing 4.17: The functor for creating a Block module.

```
module Make(Logger : Logging.Logger)
           (BlockContent:BlockContent)
           (Rewards:BlockRewards)
           : (BlockSig with
              type block_contents = BlockContent.t
              and type block = BlockContent.t t)
```

Listing 4.18: The signature of a Block module (with getters ommitted).

```
module type BlockSig = sig
    type block_contents

    type block = block_contents t

    val create : node_id -> block -> block_contents -> block
    val null : block_contents -> block
    val genesis_pow : node_id -> int -> block_contents -> block
    val genesis_pos : node_id -> block_contents -> block
    val equals : 'a t -> 'a t -> bool
end
```

The Block module provides the following operations, besides getters:

- **create** → creates a new block given the id of the node that is creating the block, the previous block in the chain, and the suder-defined block *contents* to be included in the block.
- **null** → some protocols require a block to represent the *null* or *empty* block, which is what this function returns.
- **genesis_pow** → obtain the genesis block, for proof of work protocols.
- **genesis_pos** → obtain the genesis block, for proof of stake protocols. There is a need for a seperate function, because the internal fields *difficulty* and *total_difficulty* are initialized with different values depending on the family of the protocol.
- **equals** → compares two blocks, returning whether or not they are equal.

4.12.5 Offline Nodes

As mentioned in previous sections, the simulator allows the user to define periods of time where some nodes are offline. Using a topology file, as described in Section 4.4.4, users have more control, as they can specify the period of time where each individual node is offline. When topology files are not being used, it is still possible to parameterize, in a more general way, that **x** nodes will be offline from timestamp **t1** until timestamp **t2**.

As expected, during the period that a node is offline, all the events directed to that node will be discarded.

4.12.6 Malicious Nodes

The simulator allows users to define malicious nodes and the timestamp when those nodes start acting maliciously.

The definition of a malicious node, consists in the implementation of a separate *Node* module, as described in Section 4.13. The simulator will then use the "good"*Node* module to handle the node's events up until the defined timestamp. From that point onwards, the "bad"*Node* module will be used.

The architecture of the simulator does impose some limitations to the definition of the malicious *Node* module, namely:

1. The internal state stored by a malicious node must be the same as the state stored by a "good"node.
2. The type of the messages exchanged by malicious nodes must also be the same as the ones exchanged by "good"nodes.

4.13 Defining a new Protocol to be simulated

The *Protocol* module contains functors that create a protocol, which is the top-level simulation loop. A user can opt between creating an abstract protocol or a blockchain protocol.

4.13.1 Abstract Consensus Protocol

Although the simulator was developed with a focus on consensus protocols in the context of blockchain systems, it is possible to implement classical consensus protocols.

The process of implementing a new abstract protocol in the simulator is the following:

1. Define the type of the messages that will be exchanged between nodes, and implement the *Message* module as described in Section 4.3.
2. By calling multiple functors in succession as demonstrated in Listing 4.19, the user can obtain the majority of modules required.
3. Create the *Statistic* modules to be used, either by leveraging the provided functors or implementing a new one. Both alternatives are described in Section 4.5.
4. An implementation of an *AbstractNode* and an *Initializer* is then required, as described in Sections 4.10 and 4.11, respectively.
5. Finally, the functor to create a protocol can be called. The signature of the functor is presented in Listing 4.20.

Listing 4.19: Sequence of functor calls after implementing the Message module.

```
module MyEvent   = Simulator.Events.MakeEvent(MyMsg);;
module MyQueue   = Simulator.Events.MakeQueue(MyEvent);;
module MyTimer   = Abstractions.Timer.Make(MyEvent)(MyQueue);;
module MyNetwork = Abstractions.Network.Make(MyEvent)
                                            (MyQueue)(MyMsg);;
module MyLogger  = Simulator.Logging.Make(MyMsg)(MyEvent);;
```

Listing 4.20: Abstract protocol functor.

```
module Abstract(Event : Simulator.Events.Event)
    (Queue : Simulator.Events.EventQueue with type ev = Event.t)
    (Timer : Abstractions.Timer.Timer)
    (GoodNode : AbstractNode with type ev = Event.t)
    (BadNode : AbstractNode with type ev = Event.t and
               type value = GoodNode.value and
               type node_data = GoodNode.node_data)
    (Initializer : Initializer with type node = GoodNode.t and
```

```
                type ev = Event.t)
  (Logger : Simulator.Logging.Logger with type ev = Event.t)
  (Statistics : Simulator.Statistics.Stats)
  (Network : Abstractions.Network.Network)
  : Protocol
```

As mentioned, the resulting **protocol** provides a *run* function containing the top-level simulation loop. The following is the sequence of operations performed by the run function:

1. Initialize the logger, creating the log file and logging the protocol's parameters.
2. Create and initialize the state of every node.
3. Populate internal data structures with information regarding offline and malicious nodes.
4. Call the *Initializer*, to add the initial events to the *EventQueue*.
5. While the queue has pending events, and the stopping conditions have not been reached, execute the following loop:
   a) Extract the event with the smallest timestamp from the *EventQueue*.
   b) If the node that should handle the event is not offline, let it process the event. Note that if the node is malicious, the event is processed using the handler defined in *BadNode*, otherwise it uses the definition in *GoodNode*.
   c) If the node's *state* field changed, logs the change and records the time taken to reach consensus.
6. When the program exits the loop, it logs all statistics produced by the protocol, the network, and the nodes.

4.13.2 Blockchain Protocol

The process of defining a blockchain protocol is almost identical to the process that was described for implementing a classical consensus protocol. The differences are:

1. Instead of implementing an *AbstractNode*, the user must implement a *BlockchainNode*.
2. The resulting **protocol** can handle stopping conditions that rely on the height of the chain.

4.13.3 Five-Component Framework

In an attempt to provide more code reusability, users can implement blockchain protocols structured according to the Five Component Framework, described in Chapter 2, by following the described process:

1. Define the type of messages exchanged between nodes, and implement the *Message* module.
2. Call the same series of *functors* described in Listing 4.19.
3. Implement 5 modules, one with each of the following signatures: *Proposal*, *Validation*, *Propagation*, *Finalization* and *FCFNode* (the corresponding signatures can be found in Appendix B). Note that the incentives component is defined elsewhere, in the block rewards, as specified in Section 4.12.4.
    a) **Proposal** → provides a function that returns one of two things:
        i. A **message**, containing the node's proposal, if it meets the conditions to create a proposal in its current state.
        ii. **None** otherwise.
    b) **Validation** → contains a function that checks the validity of any message received by the node.
    c) **Propagation** → contains two operations:
        i. **receive** → iterates the node's message queue, and returns a list of messages that the node can process in its current state.
        ii. **propagate** → propagates a message through the network.
    d) **Finalization** → contains a function to process received messages. Contains the majority of the behavior encompassed by the consensus algorithm.
    e) **FCFNode** → defines the state of a node, as well as a function to compute the node's initial state.
4. Call a functor using the implemented modules as arguments, which will produce a *BlockchainNode*.

The key differences between this approach and the ones described in the previous sections can be summarized as follows:

1. The behavior of the consensus algorithm executed by a node is divided into 5 modules.
2. The functor uses these modules to produce the equivalent of the main loop that each node will execute. Behind the scenes, this loop gets converted to an event handler, that executes one iteration of the loop every time there is an event for a

given node. Listing 4.21 presents a pseudo-code description of one iteration of the loop.

Listing 4.21: Pseudo-code of one iteration of the loop of a Five Component Framework's Node.

```
var proposal = Proposal.create_proposal()
if proposal != None then
    Propagation.propagate(proposal)
    Finalization.process(proposal)
var pending_messages = Propagation.receive(msg_queue)
for msg in pending_messages do
    Finalization.process(msg)
```

This approach proves useful when defining nodes with malicious behavior. For example, to implement a flooding attacker, the only module that would need to be reimplemented is *Propagation*, and the remaining modules could be reused. We also validate the usefulness of this approach when implementing different versions of the same protocol, as presented in Section 6.4.

5 The Graphical User Interface

As mentioned, the developed framework is divided into two components. Chapter 4 presented the simulator component. This chapter will present the Graphical User Interface.

5.1 Overview

The usability of a simulator is just as important as the functionalities it provides. Hence, to assist in the parameterization of the simulations, as well as the analysis of its results, we developed a Graphical User Interface.

The graphical user interface was implemented using NodeJS, Vue3, and ElectronJS. The usage of ElectronJS allows the creation of native executables for every platform. Furthermore, the fact that the underlying implementation consists of web technologies enables the future deployment of the framework as a web application (Section 7.2).

5.2 Extensibility

The graphical user interface was developed with the intention of allowing users to use it with their own custom simulators. To that end, the presented GUI can be used with any simulator, assuming the following conditions are met:

1. The simulator uses a *parameters.json* file as input, with three categories of parameters: *General*, *Network* and *Protocol*. The actual parameters included in each category are all user-defined.

2. The simulator produces an *output.json* file, with the format presented in Section 4.6.

If those conditions are met, the only requirement to use the graphical user interface with a different simulator is the `simulator_path` variable in the source code.

5.3 Features

The interface is composed of four pages, whose functionalities and inner workings are described in the following sections.

5.3.1 Parameterization

One of the functionalities provided by the graphical user interface is the ability to define all parameters of the simulation.

To this end, the *Parameters* window parses the `parameters.json` file, whose contents were described in Section 4.4, and produces a form where the user can customize the values of every parameter. This window can be seen in Figure 5.1.

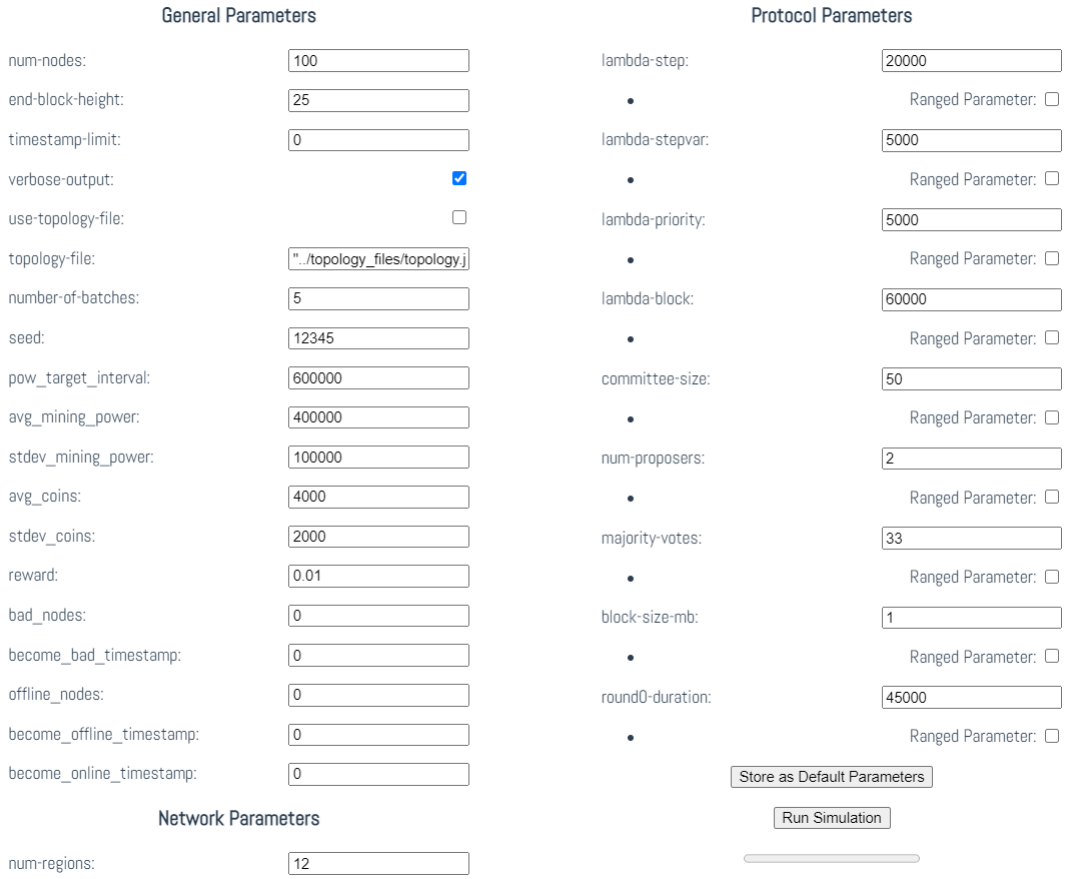

Figure 5.1: The Parameters window.

Then, selecting the *Run Simulation* button will execute the simulation with the specified parameters.

Furthermore, besides specifying a value for a parameter, users can also specify an interval of values to be assigned to the parameters. The simulator will then run multiple simulations, one for each different combination of values assigned to the different parameters.

Figure 5.2: Example of specifying a range for the block-size and round-duration parameters.

Internally, for the example provided in Figure 5.2, the GUI will:

1. Produce one *parameters.json* file, for each combination of parameters. In this case:
   a) **parameters1.json** → `block-size-mb=1` and `round0-duration=1000`
   b) **parameters2.json** → `block-size-mb=1` and `round0-duration=2000`
   c) **parameters3.json** → `block-size-mb=2` and `round0-duration=1000`
   d) **parameters4.json** → `block-size-mb=2` and `round0-duration=2000`
2. Execute the simulation using each of the generated parameter files, producing at least one *output.json* file per parameter file.
3. Assuming we are using 2 batches, the files in Listing 5.1 would be produced, where *outputX-Y.json* means output of batch **Y** with parameterization **X**.

Listing 5.1: Example of output files produced when pressing Run Simulation.

```
output1-1.json
output1-2.json
output2-1.json
output2-2.json
output3-1.json
output3-2.json
output4-1.json
output4-2.json
```

5.3.2 Topology Specification

The graphical user interface enables users to specify topology files (Section 4.4.4) without the need to manually write the JSON file.

The *Topology* window offers users a canvas to construct a network topology, as well as individually parameterize each node. The canvas can be seen in Figure 5.3, and the parameterization options available when a node is selected are displayed in Figure 5.4.

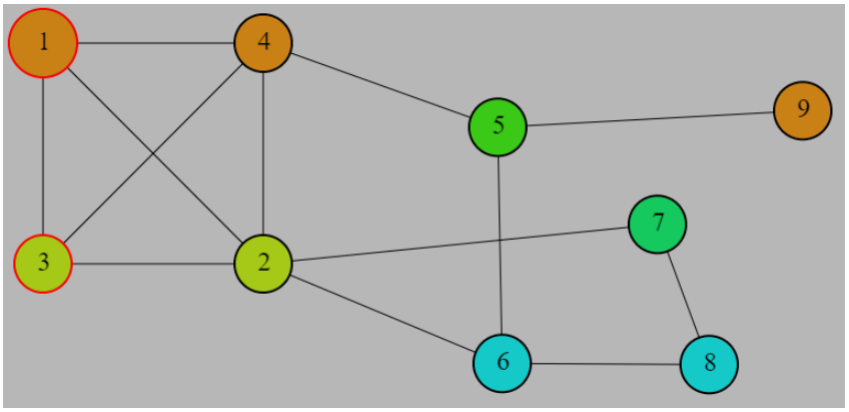

Figure 5.3: The Topology window.

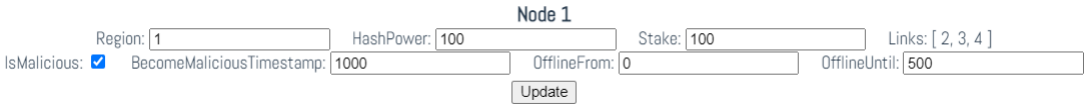

Figure 5.4: The possible parameterizations for each individual node.

Specifically, this page enables users to:

- Create nodes by *left clicking* on the canvas.
- Create links between nodes, by *right clicking* on two separate nodes.
- Remove a node or a link by *left clicking* on it and pressing the *D* key.
- Individually parameterizing each node, by *left clicking* a node and updating the values displayed in the dialog box.
- Store the topology file and/or load an existing one.
- Intuitively see properties of the nodes, such as which nodes are malicious (red border), and what nodes belong to the same region (nodes are colored according to their assigned region).

5.3.3 Visualizer

The *Visualizer* window was built on top of SimBlock's Visualizer [5]. Originally, it allowed users to upload the output from a simulation, and play it in the form of a timelapse, allowing users to observe the state of each node, as shown in Figure 5.5.

We converted SimBlock's visualizer to Vue3, and implemented the following extensions:

1. When a proof of work protocol is being simulated, the nodes that minted the block that is currently being propagated are marked with the letter **M**.

2. When a proof of stake protocol is being simulated, the nodes that are currently proposers or committee members are marked with a **P** or **C**, respectively (Figure 5.6).

3. Clicking on a node will show the block that is in the head of its local chain, and some statistics for proof of stake protocols such as the number of proposals created by the node and the number of committee participations (Figure 5.7).

4. Clicking on a link between two nodes will display the contents of the message that is currently being exchanged, formatted according to the `to_json` operation in the *Message* module (Figure 5.8).

5. Users can adjust the speed at which the simulation is displayed.

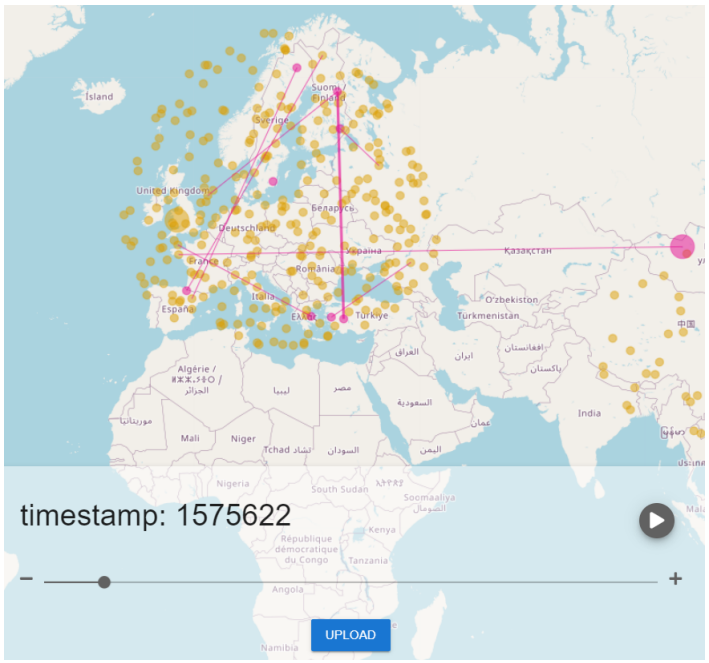

Figure 5.5: The timelapse in SimBlock's visualizer.

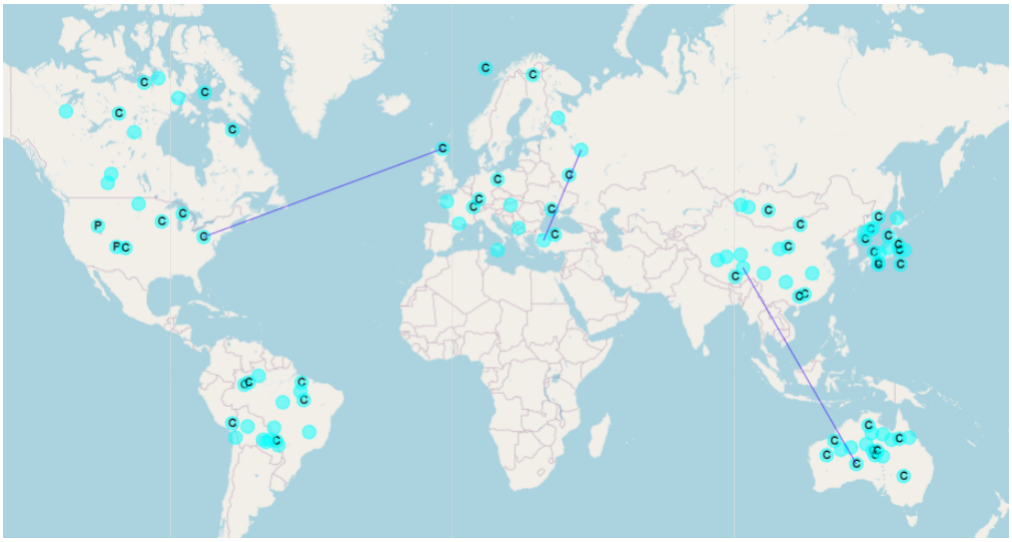

Figure 5.6: The timelapse in the Visualizer window.

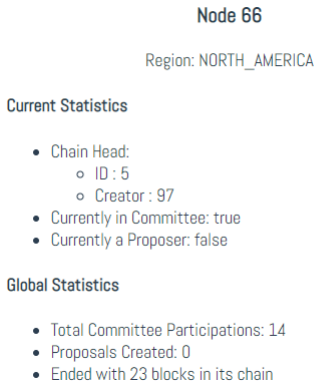

Figure 5.7: The information displayed when a user clicks on a node.

5.3.4 Statistical Analysis

Finally the *Statistics* window aids in the analysis of the metrics produced by the simulator, by parsing the *output.json* file and displaying it in an easy-to-read format. Specifically:

1. Generates graphs showing the effects the parameter changes have on the produced metrics (Figure 5.9). The GUI computes the averages of the metrics generated by the simulator in each batch, for each parameterization. The information displayed in the graphs automatically uses these averages.

2. Displays the minimum and maximum value observed for each metric produced, highlighting the parameters used in that specific simulation (Figure 5.10).

3. Generates a graph with the per-node statistics, where the X-axis is the id of the node, and the Y-axis is the value of the statistic (Figure 5.11).

Figure 5.8: The information displayed when a user clicks on a link between nodes.

It is worth highlighting that all the information displayed is dynamic, according to the contents of the *output.json* files. If a user produces more (or less) metrics in its simulation, the *Statistics* page will display it accordingly.

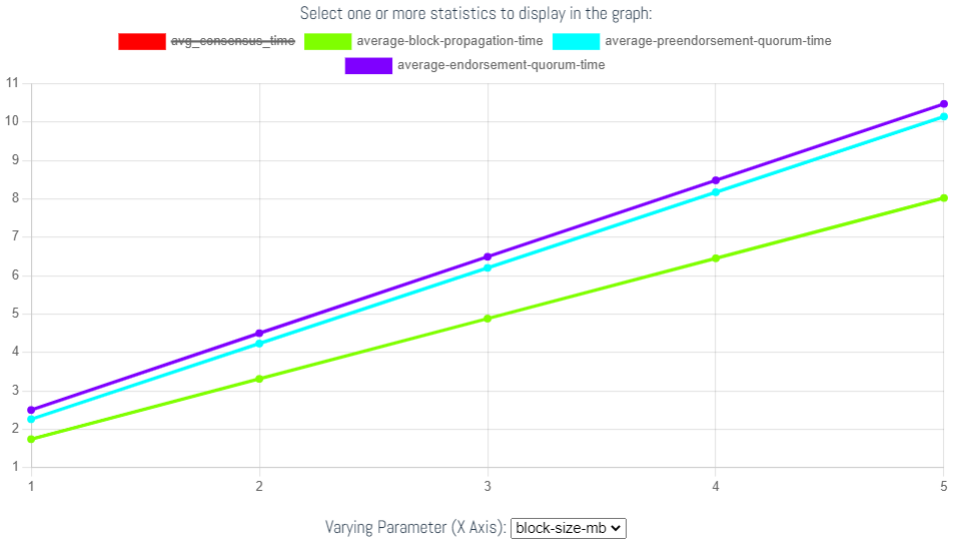

Figure 5.9: The graph generated for a Tenderbake simulation, varying the size of a block.

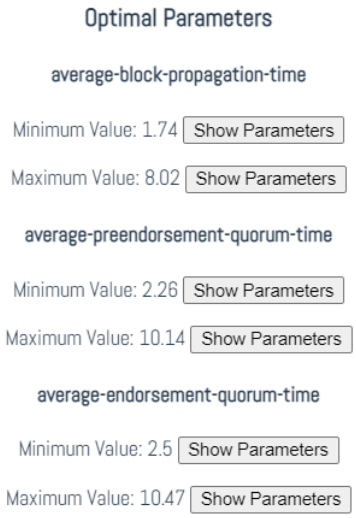

Figure 5.10: The minimum and maximum values observed for each output metric.

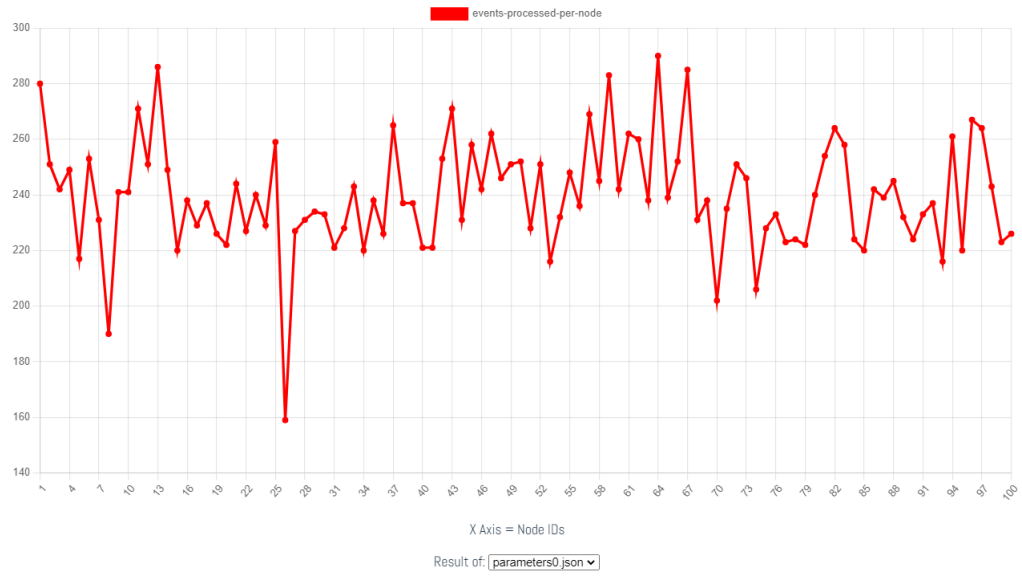

Figure 5.11: The graph generated for the number of events processed by each node, where one node is temporarily offline.

6

Use Cases

To validate different aspects of the simulator, several protocols were implemented. The following chapter will discuss the implemented protocols, their purpose, and the obtained validation results.

6.1 Bitcoin-like

The **Bitcoin-like** implementation is a general proof of work protocol, simulating the behavior described in Section 2.2.2. Using different parameterizations, this implementation can be used to simulate the Bitcoin blockchain, as well as Bitcoin forks such as Litecoin and Dogecoin.

This was the first protocol to be implemented in the simulator, with the main purpose of validating the proposed network model.

To that end, we followed a validation process similar to the authors of SimBlock [5], which consisted in comparing the median block propagation time produced by our simulator with the real values measured by Gervais et al. [16].

Since the realism of the simulation's output is dependant on how accurate the parameters are, we adapted the network parameters used by the authors to match the ones required by our simulator.

Table 6.1 contains the parameters used to simulate Bitcoin, Litecoin and Dogecoin, using the *Bitcoin-like* implementation.

	Bitcoin	Litecoin	Dogecoin
Number of Nodes	6000	800	600
Target Interval	10min	2min30sec	1min
Block Size	534 KiB	6.11 KiB	8 KiB

Table 6.1: Parameters used to simulate Bitcoin, Litecoin and Dogecoin, respectively.

	Bitcoin	Dogecoin	Litecoin
Measured MBPT	8.7s	0.98s	1.02s
Simulated MBPT	9.1s	0.86s	0.88s

Table 6.2: Comparison between of the measured median block propagation time measured with the value obtained with our simulator, for Bitcoin, Litecoin and Dogecoin.

The results obtained are listed in Table 6.2. We can see that our simulator is capable of capturing the median block propagation time similar to reality. This not only validates the proposed implementation of the *Bitcoin-like* protocol, but also the computations of message arrival times (described in Section 4.7).

6.2 Algorand

An important feature of our simulator, when compared to the state of the art, is the support for Practical Byzantine Fault Tolerant consensus protocols. Providing a concrete, validated implementation of such a protocol was, therefore, crucial to establish the value of the simulator.

To this end, the Algorand Agreement protocol (described in Section 2.2.3) was implemented in the simulator.

We validated our implementation by comparing our results with the results the authors presented in their paper [17]. Namely, we show that:

1. The simulated median latency for one round (average time to see a majority of certvotes) is close to the results presented by the authors in the paper.
2. Using the simulator to study the effects of the block's size in the different steps of the protocol leads to the same conclusions that the authors obtained through emulation.

The authors of the Algorand paper provided a detailed description of the conditions under which each node executed:

1. Nodes were deployed on virtual machines, using Amazon's EC2.
2. Each node was assigned a bandwidth limit of 20 Mbps.
3. Each node was assigned 8 neighbors.
4. The latency was modeled using inter-city latency measurements [18], for 20 major cities.
5. Nodes were equally assigned to each of those 20 cities.

We converted the described scenario to the format expected by the simulator.

Median Latency for one Round

Table 6.3 contain the relevant parameters used to simulate the median latency for a round of Algorand.

	Number of Nodes	Number of Committee Members	Number of Proposers	Block Size	Stopping Condition
Value	5000	2000	26	1 Mb	250 blocks

	λ priority	λ block	λ step	λ stepvar	Number of Batches
Value	5s	60s	20s	5s	10

Table 6.3: Parameters used in the simulation to compute the median time to complete a round.

As a result of the simulation, we obtained that the median latency for one round is **15.8s**. The result obtained by the authors, through emulation was **~18s**.

Effect of the size of a Block

One of the use cases analyzed by the authors, was the impact that the size of a block had on the time taken to complete each of the following steps of the protocol: block proposal, majority of softvotes and majority of certvotes.

Although the authors performed the study with 50000 nodes, which our simulator is not capable of doing within a reasonable amount time, we show that even using significantly less nodes, as detailed in Table 6.4, we can obtain results that lead to the same conclusions the authors reached through emulation.

	Number of Nodes	Number of Committee Members	Number of Proposers	Block Size	Stopping Condition
Value	500	100	5	1-8 Mb	250 blocks

	λ priority	λ block	λ step	λ stepvar	Number of Batches
Value	5s	60s	20s	5s	10

Table 6.4: Parameters used in the simulation, to study the impact of the size of a block in the duration of each step of the protocol.

Figure 6.1 shows the results of the simulation.

We can see that, with a block size ranging between *1Mb* and *4Mb*, there are only slight variations in the computed metrics. This means that, within this range of values, the time taken to propagate a proposal through the network is less than or equal to λblock + λstepvar, which is the mandatory time that a block waits for proposals at the begging of a round.

When the block size ranges between *5Mb* and *8Mb*, the block proposal step increases significantly, because the previously described *wait period* is no longer enough to allow for the proposal to be propagated through the network.

However, we can also see that the size of a block only impacts the duration of the block proposal step, whereas the *majority-softvotes* and *majority-certvotes* durations remain the same.

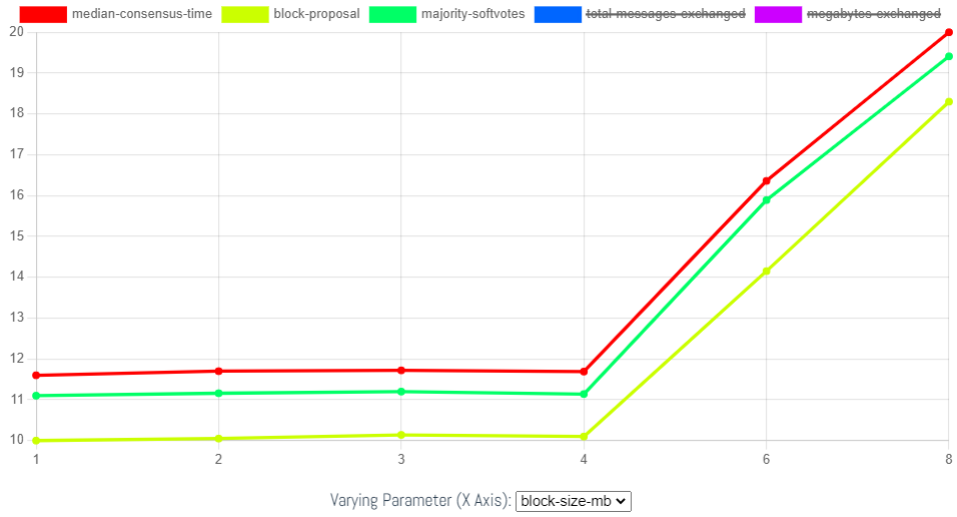

Figure 6.1: The duration of each step of an Algorand round, as a function of the block size.

6.3 Tenderbake

The included example use case for the Tenderbake (Section 2.2.4) simulation, consists in finding an appropriate value for the round duration parameter, for a given block size.

To this end, the network parameterization described in Section 6.2 was used.

We then assigned an arbitrarily large value to the round duration parameter and executed the simulation while varying the size of a block between *1Mb* and *5Mb*.

The results of the simulation can be found in Figure 6.2. From the results, we can conclude that, for example, when using blocks with a size of *3 Mb*, the round duration parameter should have a value higher than *5.65s*, otherwise it will often take multiple rounds to reach consensus on a block.

Furthermore, from the results we can conclude that, similarly to Algorand, the size of a block only impacts the duration of the initial step of the protocol.

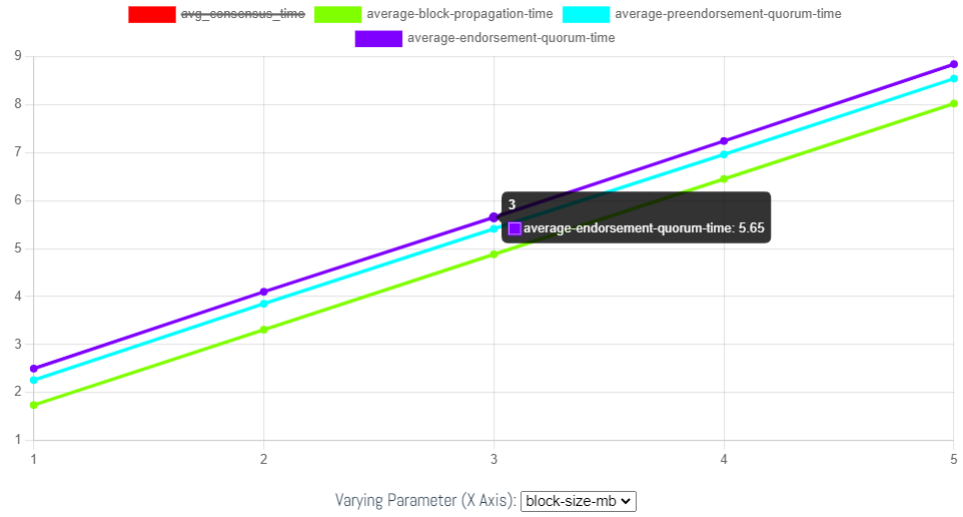

Figure 6.2: The average block propagation time, preendorsement quorum time, and endorsement quorum time observed when simulating 250 Tenderbake nodes, while varying the size of a block between 1Mb and 5Mb.

6.4 Ouroboros

The purpose of the implementation of the two versions of Ouroboros was to validate the proposed model to define protocols according to the Five-Component Framework.

To this end, both Ouroboros-Praos and Ouroboros-BFT were implemented according to the event-driven approach (Section 4.13) as well as the Five-Component Framework approach (Section 4.13.3).

Table 6.5 highlights the number of lines required for each implementation. In both approaches, Ouroboros-Praos was implemented first, and the Ouroboros-BFT attempts to reuse modules defined in Ouroboros-Praos.

	Event Driven	Five-Component Framework
Ouroboros-Praos	237 lines	275 lines
Ouroboros-BFT	163 lines	73 lines
Lines Reused	74 lines	202 lines

Table 6.5: Comparison of number of lines required for each implementation.

We can see that although the usage of the Five-Component Framework approach requires more lines of code for the first implementation, it allows us to reuse significantly more code when defining a new version of the protocol.

This is because we can reuse parts of the node's behavior, whereas in the event-driven approach the user has to reimplement the entire node module (even though a significant portion can be copied).

7

Conclusion

7.1 Summary

Blockchain technologies have seen a rise in popularity in recent years, leading to the emergence of numerous blockchain protocols and consensus algorithms. The development of such protocols involves a lot of decisions regarding their behavior and parameterizations, and reasoning about these issues is often not trivial.

Although blockchain simulation isn't a necessarily new concept, it is rather unexplored when it comes to simulators designed to support different families of protocols.

The proposed simulator is, to the best of our knowledge, the first simulator that provides concrete implementations of both proof of work and proof of stake protocols, without abstracting the messages exchanged in the underlying Practical Byzantine Fault Tolerant consensus algorithms. The simulator also provides great flexibility for specifying test scenarios, such as the existence of offline and malicious nodes.

We validated our work through a variety of use cases by implementing the Bitcoin-like, Algorand, Tenderbake, Ouroboros-BFT, and Ouroboros-Praos protocols. Namely, the simulator can accurately model median block propagation times, provide results that are equivalent to emulation, and that the proposed model for implementing protocols according to the Five-Component Framework allows for significant code reusability when simulating different versions of the same protocol.

Furthermore, the proposed graphical user interface significantly improves the usability of the simulator, by offering an interface for parameterizing the protocol, running the simulations, and analyzing the results, either by the dynamically generated graphs and statistics or by playing a timelapse of the simulation.

7.2 Future Work

The developed framework can be improved in different areas, which we outline as future work:

- **Web Application** → as mentioned in Chapter 5, the graphical user interface was

built using web technologies. It could be interesting to have the simulator and the GUI deployed in a server, and allow users to make requests to the server to execute simulations.

- **Predefined Metrics/Statistics** → by default, the simulator produces the following metrics: average time to reach consensus, the number of messages and megabytes exchanged, and the number of events processed by each node. It would be valuable to find and extract, by default, more metrics that are relevant for all blockchain protocols.

- **Gossip Abstraction** → as mentioned in Section 4.7.4, the gossip abstraction imposes several restrictions to the scenarios in which it is used. Further developing the abstraction in order to remove any restrictions, while still obtaining significant performance increases would be a valuable contribution.

- **Protocol Evolution** → the functionality for malicious nodes could be reused to model version changes of protocols. For example, we could implement two versions of the same protocol and specify the timestamp when nodes begin adopting the new version and study the effects caused on the state of the blockchain.

- **Generalize Five-Component Framework** → although proved useful when implementing multiple versions of the same protocol, the infrastructure provided to implement protocols according to the Five-Component Framework imposes restrictions on the types of nodes and messages exchanged (Section 4.13.3). Further generalizing these modules to a point where the *BlockProposal* component of two very distinct protocols could be reused would be a valuable contribution.

- **Validation** → although the Bitcoin-like and Algorand implementations were validated by comparison with real-world data, the Tenderbake and Ouroboros implementations were not. A solid contribution would be to gather the necessary information to accurately parameterize the network for real Tenderbake and Ouroboros deployments, and compare the simulated metrics with reality.

Bibliography

[1] M. Alharby and A. van Moorsel. "BlockSim: An Extensible Simulation Tool for Blockchain Systems". In: *Frontiers Blockchain* 3 (2020), p. 28. DOI: 10.3389/fbloc.2020.00028. URL: https://doi.org/10.3389/fbloc.2020.00028 (cit. on pp. 16, 17, 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 pp. 1, 10).

[3] L. Alsahan, N. Lasla, and M. Abdallah. "Local Bitcoin Network Simulator for Performance Evaluation using Lightweight Virtualization". In: *IEEE International Conference on Informatics, IoT, and Enabling Technologies, ICIoT 2020, Doha, Qatar, February 2-5, 2020*. IEEE, 2020, pp. 355–360. DOI: 10.1109/ICIoT48696.2020.9089630. URL: https://doi.org/10.1109/ICIoT48696.2020.9089630 (cit. on p. 16).

[4] E. Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. Ed. by R. Oliveira, P. Felber, and Y. C. Hu. ACM, 2018, 30:1–30:15. DOI: 10.1145/3190508.3190538. URL: https://doi.org/10.1145/3190508.3190538 (cit. on p. 5).

[5] Y. Aoki et al. "SimBlock: A Blockchain Network Simulator". In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops, INFOCOM Workshops 2019, Paris, France, April 29 - May 2, 2019*. IEEE, 2019, pp. 325–329. DOI: 10.1109/INFCOMW.2019.8845253. URL: https://doi.org/10.1109/INFCOMW.2019.8845253 (cit. on pp. 16, 18, 19, 34, 35, 44, 57, 62).

[6] L. Astefanoaei et al. "Tenderbake - Classical BFT Style Consensus for Public Blockchains". In: *CoRR* abs/2001.11965 (2020). arXiv: 2001.11965. URL: https://arxiv.org/abs/2001.11965 (cit. on p. 10).

[7] J. Banks. "Introduction to simulation". In: *Proceedings of the 31st conference on Winter simulation: Simulation - a bridge to the future, WSC 1999, Phoenix, AZ, USA, December 05-8, 1999, Volume 1*. Ed. by P. A. Farrington et al. WSC, 1999, pp. 7–13. DOI: 10.1109/WSC.1999.823046. URL: http://doi.ieeecomputersociety.org/10.1109/WSC.1999.823046 (cit. on p. 12).

[8] V. Buterin. *Ethereum Whitepaper*. online. 2013. URL: https://ethereum.org/en/whitepaper/ (cit. on pp. 1, 5).

[9] J. Chen et al. "ALGORAND AGREEMENT: Super Fast and Partition Resilient Byzantine Agreement". In: *IACR Cryptol. ePrint Arch.* 2018 (2018), p. 377. URL: https://eprint.iacr.org/2018/377 (cit. on p. 8).

[10] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems - concepts and designs (3. ed.)* International computer science series. Addison-Wesley-Longman, 2002. ISBN: 978-0-201-61918-8 (cit. on p. 4).

[11] B. David et al. "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain". In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by J. B. Nielsen and V. Rijmen. Cham: Springer International Publishing, 2018, pp. 66–98. ISBN: 978-3-319-78375-8 (cit. on p. 12).

[12] C. Decker and R. Wattenhofer. "Information propagation in the Bitcoin network". In: *13th IEEE International Conference on Peer-to-Peer Computing, IEEE P2P 2013, Trento, Italy, September 9-11, 2013, Proceedings*. IEEE, 2013, pp. 1–10. DOI: 10.1109/P2P.2013.6688704. URL: https://doi.org/10.1109/P2P.2013.6688704 (cit. on p. 6).

[13] A. S. Deshpande. "Design and Implementation of an Ethereum-like Blockchain Simulation Framework". MA thesis. Technical University of Munich, 2018 (cit. on pp. 16, 19).

[14] Ethereum. *The Beacon Chain*. online. accessed: Februrary 15th 2021. URL: https://ethereum.org/en/eth2/beacon-chain/ (cit. on p. 1).

[15] C. S. F. Faria. "BlockSim: Blockchain Simulator". MA thesis. Instituto Superior Técnico, 2018 (cit. on pp. 16, 19).

[16] A. Gervais et al. "On the Security and Performance of Proof of Work Blockchains". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by E. R. Weippl et al. ACM, 2016, pp. 3–16. DOI: 10.1145/2976749.2978341. URL: https://doi.org/10.1145/2976749.2978341 (cit. on pp. 16, 62).

[17] Y. Gilad et al. "Algorand: Scaling Byzantine Agreements for Cryptocurrencies". In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 51–68. DOI: 10.1145/3132747.3132757. URL: https://doi.org/10.1145/3132747.3132757 (cit. on pp. 5, 8, 63).

[18] *Global Ping Statistics*. URL: https://wondernetwork.com/pings (cit. on p. 63).

[19] A. Kiayias and A. Russell. "Ouroboros-BFT: A Simple Byzantine Fault Tolerant Consensus Protocol". In: *IACR Cryptol. ePrint Arch.* (2018), p. 1049. URL: https://eprint.iacr.org/2018/1049 (cit. on p. 11).

[20] A. Law. *Simulation Modeling and Analysis, 5th Edition*. McGraw-Hill Science/Engineering/Math, 2014. ISBN: 0072988436 (cit. on pp. 13, 14).

[21] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. online. 2008. URL: https://bitcoin.org/bitcoin.pdf (cit. on pp. 5, 6).

[22] L. Stoykov. "VIBES: Fast Blockchain Simulations for Large-scale Peer-to-Peer Networks". MA thesis. Technical University of Munich, 2018 (cit. on pp. 16, 18, 19).

[23] Y. Xiao et al. "A Survey of Distributed Consensus Protocols for Blockchain Networks". In: *IEEE Commun. Surv. Tutorials* 22.2 (2020), pp. 1432–1465. DOI: 10.1109/COMST.2020.2969706. URL: https://doi.org/10.1109/COMST.2020.2969706 (cit. on pp. 5, 6, 8, 10–12).

[24] W. Zhang and J. He. "Modeling End-to-End Delay Using Pareto Distribution". In: *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*. 2007, pp. 21–21. DOI: 10.1109/ICIMP.2007.26 (cit. on p. 36).

A

Appendix 1 - Log Format Example

Listing A.1: Example of each kind of log entry produced by the simulator

```
{"kind":"parameters","content":{
    "committee-size":50,
    "num-proposers":2,
    "majority-votes":33,
    "block-size-mb":1}
},
{"kind":"add-node","content":{
    "timestamp":0,
    "node-id":1,
    "region-id":2}
},
{"kind":"add-link","content":{
    "timestamp":0,
    "begin-node-id":1,
    "end-node-id":8}
},
{"kind":"flow-message","content":{
    "transmission-timestamp":1966,
    "reception-timestamp":1966,
    "begin-node-id":33,
    "end-node-id":91,
    "block-id":0,
    "msg-data":{
        "type":"INV",
        "block_id":"0"}
    }
},
```

```
{"kind":"add-block","content":{
    "timestamp":1995,
    "node-id":36,
    "block-id":0,
    "owner-id":97}
},
{"kind":"node-committee","content":{
    "timestamp":10000,
    "node-id":2,
    "round":2}
},
{"kind":"node-proposer","content":{
    "timestamp":10315,
    "node-id":48,
    "round":3}
},
{"kind":"create-block","content":{
    "timestamp":2257909,
    "node-id":56,
    "block-id":1}
},
{"kind":"statistics","content":{
    "avg-consensus-time":748.19,
    "total-messages-exchanged":24098,
    "megabytes-exchanged":2387.78}
},
{"kind":"per-node-statistics","content":{
    "events-processed-per-node":[279,243,231,
        253,231,252,233,204,254,233]}
}
```

B

Appendix 2 - FCF Module Signatures

Listing B.1: Signatures of Five Component Framework modules and functor that creates a Blockchain Node.

```
module type Proposal = sig
    type message
    type node

    val create_proposal : node -> message option
end

module type Validation = sig
    type message
    type node

    val validate : node -> message -> bool
end

module type Propagation = sig
    type message
    type node

    val receive : node -> message list ref -> message list
    val propagate : node -> message -> unit
end

module type Finalization = sig
    type message
    type node

    val process : node -> message -> node
```

```
end

module type FCFNode = sig
    type node_data
    type value
    type message
    type t = (node_data, value) Protocol.template
    type ev

    val init : int ->
            Abstractions.Network.links ->
            Abstractions.Network.region -> t
    val chain_height : t -> int
end

module Make
  (Event : Simulator.Events.Event)
  (Node         : FCFNode      with type ev    = Event.t and
        type message = Event.msg)
  (Proposal     : Proposal     with type node  = Node.t  and
        type message = Node.message)
  (Validation   : Validation   with type node  = Node.t  and
        type message = Node.message)
  (Propagation  : Propagation  with type node  = Node.t  and
        type message = Node.message)
  (Finalization : Finalization with type node  = Node.t  and
        type message = Node.message)
  : (Protocol.BlockchainNode   with type value = Node.value and
        type node_data = Node.node_data and
        type ev = Node.ev)
```