Miguel Cerdeira Alves

Bachelor in Computer Science and Engineering

Consensus Protocols Environments and Specifications

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
⟨**Computer Science and Engineering**⟩

Adviser:	António Ravara, Associate Professor, NOVA School of Science and Technology
Co-adviser:	Marco Giunti, Researcher, NOVA School of Science and Technology

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

⟨**February**⟩, ⟨**2021**⟩

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 often presented in high-level descriptions or "pseudo-code", which interferes with our ability to reason about them and their implementations. Moreover, these protocols are not written in stone, they are constantly evolving. For example, Ethereum uses a blockchain protocol based on proof of work but is progressively migrating towards an implementation based on proof of stake. Another example is Tezos, where stakeholders that participate in the system are capable of proposing and agreeing on changes to the core protocol itself, allowing it to evolve over time. It is, therefore, necessary to develop tools to assist in this evolution.

This thesis aims to address the lack of specification and experimentation environments to develop consensus and blockchain protocols, and thus help study their behavior.

The contribution will be a simulation environment that can aid researchers in making informed decisions about the effects of different design choices in the development of these protocols. Simulation of distributed, consensus, and blockchain protocols is an active field, however, a majority of these simulators are focused on performance evaluation, rather than a qualitative evaluation. The proposed simulation environment will follow a functional Domain Specific Language for abstracting blockchain protocols to enhance code readability and reusability and enable researchers to reason about the implementation of the protocols. The simulator should be modular and extensible, providing the ability to make changes to the protocols and thus simulate different families of protocols. Finally, the proposed simulator should be parameterizable to allow researchers to learn the effects of different parameters in the overall behavior of the protocol.

Keywords: Consensus protocols, blockchain, blockchain simulation, extensible simulator, blockchain abstractions

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 são regularmente apresentados através de descrições de alto nível ou em "pseudo-código", limitando a nossa capacidade de raciocinar sobre os mesmos. Para além disso, estes protocolos evoluem constantemente. Por exemplo, Ethereum usa um protocolo de blockchain baseado em proof of work, no entanto tem vindo a adotar gradualmente uma implementação baseada em proof of stake. Um outro exemplo é o caso de Tezos, onde os participantes do sistema possuem a capacidade de propôr e aceitar alterações ao protocolo do sistema em si, permitindo que este evolua ao longo do tempo. É assim importante desenvolver ferramentas que assistam esta evolução.

Esta tese tem como objetivo combater a falta de ferramentas de especificação e experimentação para desenvolver protocolos de consenso e de blockchain, ajudando assim a estudar as suas propriedades. O contributo será um ambiente de simulação que assista investigadores nas decisões que necessitam de realizar no desenho e desenvolvimento destes protocolos. A simulação de protocolos distribuídos, de consenso e de blockchain é uma área ativa, no entanto, a maioria dos simuladores possuem um foco na realização de análises de desempenho dos sistemas, e não nos princípios do seu desenho. O ambiente de simulação usará uma Linguagem de Domínio Específico (DSL) funcional de forma a abstrair a implementação de protocolos de blockchain, com o objetivo de melhorar a interpretação e reutilização do código escrito, bem como facilitar o raciocínio sobre a implementação dos mesmos. O simulador deve ser modular e extensível, permitindo que o utilizador possa fazer alterações de forma a simular diferentes famílias de protocolos. Por fim, o simulador proposto deve ser parameterizável de forma a permitir o estudo dos efeitos causados por diferentes parameterizações dos protocolos.

Palavras-chave: Protocolos de consenso, blockchain, simulação de blockchain, simulador extensível, abstração de blockchain

Contents

Chapter 1

Introduction

1.1 Context

Consensus protocols have crucial roles among distributed systems, such as blockchain technologies. They enable 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 [16]. 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], that possesses a key novel of *self-ammendment*. 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

which is the problem this thesis aims to address.

Hence, there is a need to divide these protocols into essential components, to achieve better abstraction between different protocols, and extract their similarities and key differences. These components can then be used as building blocks to structure the presentation and implementation of the protocols, enhancing their readability and code reusability, and thus enabling the development of an extensible simulation environment to study these protocols.

1.3 Goal

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.
- parameterizable, to allow researchers to learn the effects of different parameters in the overall behavior of the protocol.
- structured according to an abstraction framework, such as the five component framework presented in *"Survey Of Distributed Consensus Protocols for Blockchain Networks"* [31], enabling modularity, code reusability, and readability.
- focused on providing a qualitative evaluation of the simulated protocols.

The goal is **not** to simulate the protocol's behavior when compared to reality as a means for testing its efficiency, but rather to provide researchers results that allow them to reason about certain properties of the implementations, such as: How many adversaries does the system support? Is the system fair when handling stakeholders with different stakes?

This thesis is being developed in the context of the NOVA LINCS research unit, in parallel with other projects. The goal of this group of projects is to develop tools to support rapid prototyping, simulations, reasoning, and benchmarking of consensus algorithms and protocols. This will be achieved through the development of a Domain Specific Language (DSL) paired with static and dynamic analysis tools. The DSL should support defining executable specifications of protocols, which can then be linked with different tools - for example, the simulation environment that will be developed in this thesis - to allow for qualitative and quantitative analysis.

Chapter 2

Background

This chapter presents core concepts regarding consensus protocols, blockchain protocols, and simulation. It also describes three consensus and 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.

It is expected that a consensus protocol ensures the following properties [12]:

1. Termination - eventually, every correct process accepts a value.

2. Agreement - all correct processes accept the same value.

3. Integrity - if all correct processes proposed the same value, then all correct processes accept that same value. A stronger integrity constraint for Byzantine fault-tolerant consensus is: if a correct process accepts a value v, then v must have been previously proposed by some correct process.

We will now present Single-Decree Paxos [21], which allows different machines to agree on a single value out of several proposed values. Then we will present Raft [20][26] and Multi-Decree Paxos [20][27][10], which introduce the concept of a replicated log of operations and ensure that all correct processes will execute these operations in the same order on the same state.

Chapter 4 further elaborates on aspects related to the implementation of these protocols.

2.1.1 Single-Decree Paxos

The Single-Decree Paxos protocol [21] is used to achieve consensus on a single value, and it has no notion of a replicated log. Each entity involved in the protocol is called a peer, and each peer can assume one, or multiple, of the following roles: *Proposer*, *Acceptor* and *Learner*. The protocol progresses through the exchange of messages between peers. The following description of the peer's behavior will be divided among the different roles, to improve readability.

When a peer receives a value from a client, it begins behaving like a Proposer. A Proposer executes the following steps [21]:

1. Sends a `PREPARE` message to all the other peers, which contains the proposal number that it intends to use in a future proposal.
2. Waits for the responses to its `PREPARE` message, which are called `PROMISES`. Upon receiving a `PROMISE` from a majority of peers, the Proposer proceeds to send a `PROPOSE` message to all the peers - note that if any of the received `PROMISES` contained the information of an already accepted value, this Proposer will have to propose that same accepted value.
3. Finally, if the Proposer receives an `ACCEPT` response from a majority of peers, then it has achieved consensus on its proposal and can inform the Learners (implementation dependent) and the client.

An Acceptor keeps a record of the received `PREPARE` and `PROPOSE` messages, and whether or not a value was already accepted. Its behavior consists of processing messages from Proposers and sending the appropriate responses.

Upon receiving a `PREPARE` message, an Acceptor will respond in one of three ways [21]:

1. If the proposal number in the `PREPARE` message is the largest the Acceptor has seen so far and it hasn't accepted values yet, the Acceptor will respond with a `PROMISE` message containing the received proposal number, and from that moment forward the Acceptor will ignore any messages that contain a smaller proposal number.
2. If the proposal number is not the largest the Acceptor has seen so far and it hasn't accepted any values yet, it will simply ignore the `PREPARE` message.
3. If the Acceptor has already accepted a value, it will respond with a `PROMISE` message containing the highest proposal number it has seen so far, and the value that it has accepted.

Upon receiving a PROPOSE message, an Acceptor will respond in one of two ways [21]:

1. If it hasn't accepted any proposals yet, and the proposal number is larger than or equal to the largest proposal number it has seen so far, the Acceptor will accept the value and will respond with an ACCEPT message.

2. If it has already accepted a value, or it has seen a larger proposal number, the Acceptor will respond with a REJECT message or simply ignore the proposal - depending on the implementation.

Finally, the Learner role represents the entity that will "learn" the value for which consensus was reached. In real systems, the Learners are, for example, databases. Therefore, in the articles that describe the protocol, there is very little information about the Learner's behavior.

It is important to note that, in the presence of concurrent proposals, the Single-Decree Paxos protocol may not terminate as different proposers might send consecutive PREPARE messages with higher proposal numbers for an indefinite amount of time. However, it does guarantee that if consensus is reached, all Learners will "learn" that same agreed-upon value.

2.1.2 Raft

The Raft protocol [26] is used in the context of log replication, ensuring consensus on the order in which the operations contained in the log entries will be applied. Each entity involved in the protocol is called a *peer*, and in any given instant a peer can have one of three roles: *Follower*, *Candidate* or *Leader*.

Each peer stores a *term* and a *log*. The term is used alongside the size of the log to keep track of how up-to-date each peer is.

At its core, a log is a sequence of entries. Each entry has an associated index in the log, term when it was added to the log and arbitrary data to be used by the specific system upon reaching consensus - this data can be, for example, the insertion of a tuple in a database that will be executed in all replicas after consensus is reach - which happens when the entry is marked as committed.

The protocol can be implemented using message exchange or RPCs (Remote Procedure Calls).

The following are the descriptions of each role's behavior, as presented in the official paper [26]:

When a peer has the Follower role, its behavior is mostly based on the execution of RPCs, with a few exceptions.

Upon receiving a client request, a Follower redirects that message to the current leader of the protocol.

When a Candidate calls a RequestVote RPC on a Follower, the result will correspond to the Follower's vote. This vote will be positive if the Candidate's term and log entries are up-to-date, and the Follower hasn't voted in favor of another candidate. Otherwise, the vote will be negative.

A Leader can execute AppendEntries RPCs on all Followers, not only to send them new log entries but also to inform them that there is an active Leader. The execution of this RPC on a Follower will succeed if the sent log entries are successfully appended to the Follower's log. The execution fails if the Follower is missing some previous log entries or if it detects that the Leader is outdated - if the Leader's term is smaller than the Follower's term. If the RPC's execution succeeds, the Follower will also compare its commit index to the Leader's, apply the newly committed operations, if there are any, and the RPC's result will include the index of the last entry in the Follower's log.

Finally, a Follower will change its role to Candidate when a *heartbeat timeout* occurs - when the Follower hasn't received a message from the valid Leader recently.

When a peer changes its role to Candidate, the election process begins. The Candidate will increment its current term and will execute RequestVote RPCs on every other peer.

Upon receiving a negative result from a RequestVote RPC, the Candidate will check if that follower's term is larger than his current term and if that is true, then the Candidate will revert its role to Follower since there is another peer more up-to-date than him.

If the Candidate receives a majority of positive results from the RequestVote RPCs, the Candidate will change its role to Leader.

The Leader is in charge of executing AppendEntries RPCs on all peers periodically, regardless of whether or not there are new entries in its log, because these messages also serve the purpose of informing the peers that the Leader did not fail.

When an AppendEntries RPC is unsuccessful, the Leader will decrement the index of the next message that needs to be sent to that peer, to try and solve the conflict. Worst case scenario, the conflict is at the start of the log, and therefore the request will only succeed when the Leader decrements the index to be sent to 0.

Every time an AppendEntries RPC is successful, the Leader will check what is the latest entry that it can commit, which will be the entry with the largest index that is replicated in the log of a majority of peers.

Upon receiving any message that contains a term larger than its own, the Leader will revert its role to Follower since that means there is another Leader more up-to-date.

The Raft protocol guarantees that peers will reach a consensus on the order of the entries in the log and execute those operations in the same order, however, it does not guarantee that all entries will be added to the log, as some might get lost during leader failures.

2.1.3 Multi-Decree Paxos

The Single-Decree Paxos protocol describes how to reach a consensus on a single value however, in a majority of scenarios, this is not enough to maintain a proper state of a system. Multi-Decree Paxos is essentially a sequential execution of multiple Single-Decree Paxos rounds, to achieve log replication.

There are several papers that present somewhat different descriptions of Multi-Decree Paxos, such as [27], [10] and [20]. For now, we will present the protocol as described in "Paxos Made Live - An Engineering Perspective" [10].

The state kept by each peer is very similar to the state kept in Single-Decree Paxos, with the addition of the log entries and the identity of the last known leader.

At the start of the protocol, one peer is elected to act as the *coordinator* (also known as *leader* or *distinguished proposer*). This election can be done through one round of Single-Decree Paxos, where a peer generates a new sequence number and sends a `PREPARE` message to the remaining peers. If he receives a majority of valid `PROMISES` then he can start acting as the coordinator. Note that to ensure no out-of-date peers, the `PREPARE` message must include the peer's log, which the leader then merges with its own log.

The coordinator can then receive requests from clients or from other peers, selecting one of those entries to be broadcasted. This is the `PROPOSE` message sent in Single-Decree Paxos. This message will now also include the coordinator's log, so that peers that failed in a previous iteration of the protocol may catch up to the current state of the system.

Once a majority of replicas acknowledge the coordinator - `ACCEPT` his proposal - consensus is reached, and the coordinator proceeds to send a `COMMIT` message to notify the replicas.

The previously mentioned steps are then repeated. In practice, the election step only needs to happen again if the current coordinator fails. This failure detection can be achieved in a similar way as in Raft, by leveraging timeouts and heartbeats.

Like Raft, Multi-Decree Paxos guarantees that peers will reach a consensus on the order of the entries in the log, but in the occasion of a coordinator failure, some client requests may not be added to the log.

2.2 Blockchain Protocols

Blockchain, as the name suggests, is 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.

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.

Blockchain 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.

The object of the consensus is a chain of blocks, as depicted in figure x. However, in practice, it is not uncommon for the data structure stored by each node to be a tree of blocks (simplified in figure 2.1) as forks may occur, which will be further explained in later sections.

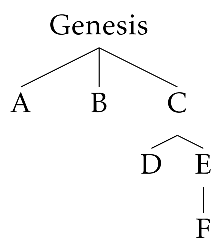

Figure 2.1: All chains must have the *Genesis* block as root. In this example, the block **F** is the head of the longest chain, and blocks **A**, **B** and **D** are referred to as stale or orphan blocks.

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 [6].

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 [24], Ethereum [8] and Algorand [19].

Several permissionless blockchains can also execute on permissioned environments, often more efficiently and securely.

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" [31], 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 and the chain it extends, 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 Byzantine Fault Tolerant agreement algorithms and checkpointing, while others 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 [24] 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 [31], of the protocol executed in the bitcoin network [24] [13].

Block Proposal

Transactions are broadcasted through the network. Nodes running 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 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 transaction and block hashes that the node has verified, and upon receiving an *inv* message other nodes can request transactions and blocks that they don't have by sending a *getdata* message to the sender of the *inv* 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 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 blocktree 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 [19] [11] is a blockchain network that manages a cryptocurrency, 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 [31], of the protocol executed in the algorand network [19] [11].

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.

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 selecting neighbor nodes based on how much money they hold and each node only relays messages after validating them.

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, the agreement protocol is executed.

Each node begins the agreement protocol with the block with the highest associated priority they have received. Each node then executes the following steps [11], in sequential rounds (also referred to as periods), until a consensus is reached:

1. Execute cryptographic sortition to check whether the node was selected to be part of the committee for that period. Note that a new period begins when a node receives a majority of next-votes for some block in the previous period.

2. If the node was indeed selected to be part of the committee, it broadcasts (to all nodes, not just the ones that are part of the committee) the block with the highest priority it has seen (if round=1) or the block that received the most next-votes in the previous round (if round>1), as well as the proof it belongs to the committee. This initially broadcasted block is known as the node's starting block.

3. If the node sees a majority of next-votes for an empty block, it sends a soft-vote for the non-empty block with the most votes that it has seen. Otherwise, if the node sees a majority of next-votes for one non-empty block, it sends a soft-vote for that same block.

4. If the node sees a majority of soft-votes for a non-empty block, then it sends a cert-vote for that block.

5. If the node has certified some value in the current period, it sends a next-vote for that same value. Else, if it has seen a majority of next-votes for an empty block, it also sends a next-vote for an empty block. Otherwise, it sends a next-vote for its starting block.

6. If the node sees a majority of soft-votes for a non-empty block, then it sends a next-vote for that block. Otherwise, if it sees a majority of next-votes for an empty block, and it has not certified a block in this period, it sends a next-vote for an empty block.

7. If a node sees a majority of cert-votes for one block for the same period, it sets that block as the output of its agreement protocol. The set of those cert-votes form a certificate for the block, meaning consensus was reached and it can be appended to the blockchain. Otherwise, a new round of the agreement protocol begins.

Note that regardless of whether or not a node belongs to the committee, it keeps track of the messages (votes) exchanged by the committee members. This is important because it allows committee members to be replaced every period while maintaining the progress achieved in previous periods which helps protect the network against targeted denial of service attacks.

Incentive Mechanism

Nodes receive a reward upon creating a block that gets successfully added to the blockchain.

2.2.4 Ethereum

Ethereum [8] [30] is a permissionless blockchain network that manages a cryptocurrency, ether, and supports a built-in Turing-complete programming language that users can leverage to create smart contracts and decentralized applications that can specify state transition functions. The execution of these pieces of code can be triggered by messages and transactions.

The following is a description, divided according to the five component framework [31], of the protocol executed in the Ethereum network [8] [30]. For clarification, this is a description of the Ethereum 1 protocol. An overview of the key functionalities introduced by the currently deployed version of Ethereum 2 [16] will be presented afterward.

Block Proposal

Miner nodes gather received transactions and pack them into a block. Since transactions may trigger state transition functions, it is necessary for the block to also include the most recent state - the state reached after the miner applied the transactions contained in the block, which may trigger contract code that changes the state.

The miner also includes zero or more stale blocks, also referred to as orphan or uncle blocks, into the created block. The included uncle blocks must be different from all uncles included in the previous blocks that belong to the chain that is being extended.

Finally, nodes execute a process similar to Bitcoin's proof of work. The difference is that the resulting proof is a nonce that when hashed together with the block header and a dataset obtained by downloading the full blockchain, must be lower than a target value.

Information Propagation

Ethereum nodes communicate using a gossip protocol. It is worth highlighting that the messages are exchanged using the RLPx Transport protocol, which allows nodes to send encrypted messages.

Block Validation

When a node receives a block, it will start by verifying if it extends a valid chain. It will then validate the contained transactions, gas limit, and block header, and finally will verify if the nonce generated is a valid proof of work.

Then it will apply the transactions contained in the block. If any transaction returns an error or the total gas consumed exceeds the limit, the validation fails. Otherwise, if all transactions successfully terminate, it will check whether or not the state reached after their execution is the same as the one contained in the block.

Block Finalization

In the Ethereum network, nodes follow the Greedy Heaviest Observed Subtree, or GHOST rule. This is similar to the longest chain rule, where nodes work to extend the longest chain, but it also considers the uncle blocks in the calculation of which chain is the "longest".

As with the longest chain rule, when a block is sufficiently deep in the chain, it can be considered final with a high probability.

Block Finalization in Ethereum 2

The current state of Ethereum 2 is a hybrid proof of stake and proof of work protocol. It builds on top of the Ethereum 1 proof of work implementation, adding a checkpointing mechanism to block finalization, using the CasperFFG protocol [9].

CasperFFG [9] allows nodes to maintain a checkpoint tree and vote on a chain of checkpoints, resolving forks and finalizing the blocks on the agreed chain.

Every time a node appends a block to the chain, with that block's height being a multiple of 100, that block is considered a valid checkpoint block.

Nodes assume the role of validators. Each validator has an associated deposit of ether, its stake. Validators broadcast vote messages, containing two checkpoints and their respective heights.

Each node's checkpoint tree also includes a vote count, where the votes are weighted according to the voter's stake. Consensus on a chain is reached as follows:

1. If an ordered pair of checkpoints (a,b) has received 2/3 of weighted votes, it is called a supermajority link.

2. A checkpoint a is justified if it is the checkpoint tree's root, or if there is a supermajority link (b,a) such that b is justified.

3. A checkpoint a becomes finalized if it is the checkpoint tree's root, or if it is justified and there is a supermajority link (a,b) where b is a direct child of a.

4. Honest validators always cast votes to extend the chain with the highest justified checkpoint.

Once a checkpoint becomes finalized, all the blocks it extends not only in the checkpoint tree but also in the overall block tree, can be considered final.

Incentive Mechanism

Nodes get rewarded for successfully creating a block that gets appended to the main chain and for the associated transaction fees. Besides that, a node is also rewarded for the creation of stale blocks that get included as uncles in blocks that extend the main chain.

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.

Discrete event simulation is a widely used simulation model where the state of the system is only updated at discrete points in time where the events occur. In this model it is common for a central mechanism to manage the system, progressively updating the simulated time and scheduling the execution of these events.

Overall, 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.

CHAPTER 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.

"Protocol Combinators for Modeling, Testing, and Execution of Distributed Systems" [5] define `Distributed Protocol Combinators` (DPC) which are a set of high-order programming primitives, extracted from common interaction patterns present in distributed systems. The DPC was implemented in Haskell, along with tools to test protocols implemented with the DPC. To validate their work, some well-known protocols such as `Single-Decree Paxos` and `Two-Phase Commit` were implemented. However, it is unclear whether or not the system would be able to specify and simulate more complex protocols such as Bitcoin, for example.

There have also been efforts to define simpler, intuitive, and executable specifications of existing protocols such as Multi-Decree Paxos [22], leveraging *DistAlgo* [23] which is a language that aims at simplifying the specification and execution of distributed algorithms, which in turn enables users to better reason about the implemented algorithms.

In terms of blockchain simulation, it isn't necessarily a new concept, however, it is still rather unexplored especially simulators designed to provide a qualitative analysis of the systems, rather than testing their performance. Although there have been several simulators developed to test and validate the performance of specific blockchain systems, such as Blockchain Simulator [18], Bitcoin Network Simulator [3] and eVIBES [14], 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 existing extensible blockchain simulators, namely BlockSim [17], BlockSim [1] and VIBES [29], since their scope is similar to this work.

3.1 BlockSim: Blockchain Simulator

BlockSim [17] is a simulation framework developed in Python that assists in the design, implementation, 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 in 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

BlockSim [1] is a discrete-event simulation framework developed in Python, with the purpose of exploring the effects of configuration, parameterization and design decisions on the behavior of 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 in 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 a 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 [29] 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.

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.

3.4 Critical Analysis

VIBES [29] is scalable, fast, and is capable of being extended to blockchain protocols other than bitcoin, although requiring some changes to be made to the simulation engine [14].

	Adversarial Behaviour	Abstraction Layers	Proof of Stake	Proof of Work
VIBES [29]	not modeled	none	not modeled	bitcoin
BlockSim [17]	not modeled	node layer consensus layer ledger layer transaction layer block layers network layer cryptographic layer	not modeled	bitcoin ethereum
BlockSim [1]	not modeled	network layer consensus layer incentives layer	not modeled	bitcoin ethereum

Table 3.1: Overview of the provided functionalities of the existing simulators, in regard to properties we consider relevant. If adversarial behaviour is modeled, what are the adopted abstraction layers for extending the simulator to other protocols, and what proof of stake and proof of work protocols have been modeled.

However, it does not follow any abstraction mechanisms to specify the protocols to be simulated, which would increase its extensibility.

Neither BlockSim [17] nor BlockSim [1] model adversarial behavior, nor do they provide a concrete foundation for simulating families of blockchain protocols based on proof of stake.

Both BlockSim [17] and BlockSim [1] define different abstraction layers to facilitate the simulation of different blockchain protocols. However, we believe that using finer-grained abstraction layers is advantageous.

Hence, this thesis intends to fill this gap by providing a simulator that models adversarial behavior, models more families of protocols besides proof of work, and defines concrete, finer-grained abstraction layers for modeling blockchain protocols, thus achieving better code structure and reusability, and enhancing the readability and understandability of the simulated protocols.

Chapter 4

Use Cases

This chapter will introduce the work done in preparation for the thesis.

4.1 Consensus Protocols in Rust

Single-Decree Paxos [21], Raft [26] [20] and Multi-Decree Paxos [10] [20] were implemented in Rust, with the goal of achieving a better understanding of consensus protocols and what is required to transform their pseudo-code descriptions into working implementations, as well as extracting similarities from the implementations of the different protocols. These toy implementations can be accessed at the Github repository [4].

The three implementations possess the same foundation:

1. Each node is simulated by one thread and nodes can communicate with each other via Rust's `mpsc` crate [15].

2. Artificial delays were added to some operations, such as sending a message.

3. Messages can be "lost", therefore some nodes never received them.

4. Nodes can temporarily fail. During that period of time, they do not receive nor respond to any messages.

5. Membership is static. There is no notion of membership management as nodes do not leave the network, neither do new nodes join it.

6. The execution of the protocols produces verbose output files to aid in understanding the behavior and progress of the protocols.

4.1.1 Single-Decree Paxos

The implementation of the Single-Decree Paxos protocol following its description [21] was straightforward, however, the resulting implementation has over 500 lines of code, whereas the protocol is usually presented in less than one page of pseudo-code. Most of this overhead is due to having to define the structure of every message sent during the protocol's execution, defining the operations to send, receive and validate each message, and defining error handling and logging operations.

The main aspect of the protocol that must be highlighted is the fact that different peers must use different proposal numbers, otherwise the *Agreement* property, mentioned in Chapter 2, would not be ensured. In the toy implementation, a simple solution was achieved using floating-point proposal numbers. Each peer starts with a proposal number, where the fractional part of the number is equal to its peer identifier and the peer simply increments this number by 1 as needed.

Another possible solution would be for peers to increment the proposal number multiple times, and only use a proposal number **i** if **i % n = p**, where **n** is the total number of peers in the network and **p** is the peer's identifier.

4.1.2 Comparison between Multi-Decree Paxos and Raft

The implementation of the Raft protocol resulted from an almost direct translation of the protocol's pseudo-code [26] and *TLA+* specification [25].

The main difference between the implementation and the original paper [26] is that communication between peers was done through *message-passing* instead of *remote procedure calls*. This decision was done with the purpose of reusing the *message-passing* code from the Single-Decree Paxos implementation, as well as reusing its overall project structure.

As highlighted in Chapter 2, the Single-Decree Paxos protocol achieves consensus on a single value, whereas Raft and Multi-Decree Paxos are used to maintain a replicated log. Because of this difference, finding similarities between Raft and Single-Decree Paxos proved difficult, as the main similarities weren't specific to the protocols themselves, but rather specific to how the implementations were structured and how messages were exchanged.

So with the implementation of Multi-Decree Paxos, the main goal was to reuse as much code as possible from the Raft implementation. To this end, the paper *"Paxos vs Raft: Have we reached consensus on distributed consensus?"* [20] was used as a reference, since the authors presented the Multi-Decree Paxos protocol using Raft's terminology, making it easier to reason about the similarities between them. For example, proposal numbers are equivalent to Raft's terms and the distinguished proposer is equivalent to Raft's leader.

The key differences between the protocols are related to leader election and how log entries are committed. These differences are summarized below, as highlighted in the paper [20]:

1. How does the protocol ensure that each term has at most one leader?
   a) `Raft` → a follower can become a candidate in any term. Each follower will only vote for a single candidate per term, therefore only one candidate will receive a majority of votes and thus become the leader.
   b) `Multi-Decree Paxos` → a peer **p** can only become a candidate in term **t** if **t % n = p**, where **n** is the total number of peers in the network. This means that there will only be one candidate per term, and therefore there will only be one leader per term.
2. How does the protocol ensure that a new leader's log contains all committed log entries?
   a) `Raft` → a follower votes for a candidate only if the candidate's log is at least as up-to-date as the follower's log.
   b) `Multi-Decree Paxos` → each vote from a follower to a candidate also includes the follower's log entries. Upon receiving a majority of votes, the candidate will merge the entries with the highest term to its own log.
3. How does the protocol ensure that the leaders safely commit log entries from previous terms?
   a) `Raft` → the leader replicates the old uncommitted log entries to the other peers without changing them, but these entries will only be committed when the leader commits at least one entry from its current term.
   b) `Multi-Decree Paxos` → uncommitted log entries from previous terms are added to the leader's log with the leader's new term. The leader then replicates these entries as if they were added in his term.

These differences translate to roughly 250 non-reusable lines of code for each protocol, and close to 400 lines of code that are common to both protocols.

The section of Multi-Decree Paxos that may require further explanation is how the logs are merged when a leader is elected. As mentioned, each peer keeps track of the last entry that has been committed, and when voting for a candidate the peers will include their log in the vote message.

Upon being elected, the leader will then merge these logs with its own. Already committed entries will not change, however, each index after the last committed entry will contain the log entry with the highest term that is included in the log of a peer at that same index - in summary, each index will contain the most up-to-date log entry among the logs of all peers in the network.

After the logs have been merged, the term of all uncommitted entries will be updated to the leader's current term.

4.2 Blockchain Protocols in OCaml

To begin defining the abstractions for the blockchain protocols that will be simulated, the signatures of high-level functions used in the different components of Bitcoin, Ethereum and Algorand were defined and divided into modules, and a template for a modular main function that can be used by the three protocols was also defined.

OCaml is the language of choice because it provides the right amount of abstraction we require, however, there are some questions regarding whether or not the language's multithreading capabilities will be able to satisfy our needs.

In the following sections, the module signatures of essential functions identified in the Bitcoin, Algorand and Ethereum protocols will be presented, as well as the reusable module-based top-level function and its intended usage.

4.2.1 Function Signatures

The goal of this section was to identify essential functions that are used by different protocols and divide them into modules. When implemented, these modules should serve as a library that provides functions and structures commonly used by blockchain protocols, facilitating the development of the top-level modules used in the modular main function, which will be presented in the following section.

To this end, the following modules were defined, using as reference the Bitcoin, Algorand and Ethereum protocols: `Network`, `Blocktree`, `ProofOfStake` and `ProofOfWork`.

The `Blocktree` module, contains operations that should be applied to a blocktree data structure. For example, retrieving the best chain in the tree, checking if a block is already in the tree, among other operations.

The `ProofOfWork` module, defines operations commonly used in proof of work protocols, such as computing the hash of a block and a *nonce*, as well as verifying if the produced hash is valid, through the execution of an acceptance function.

The `ProofOfStake` module, defines operations to select committee members and proposers for a given round, as well as operations to validate their credentials/capabilities.

Finally, the `Network` module defines operations to send and receive messages, as well as compute network delays.

The full module signatures can be found in the Appendix A. All types are defined as abstract types in each module, to improve the readability of the signatures in this phase. Most of the types are intuitive, however, it is worth delving deeper into the intended usage of the communication channels in the `Network` module.

The `communication_channels` type should be a set of multiple message channels, one for each type of message being exchanged in the network - one can expect, at the very least, one channel for blocks and another for transactions. The receive operations will then check the respective channel for new messages, and the `send_message` operation can receive as an argument any message type, and through pattern matching the correct communication channel will be used.

The implementation of these communication channels could leverage, for example, *camlboxes* from the *Netcamlbox* module provided in the *Ocamlnet* package [28]. *Caml-boxes* provide an inter-process communication mechanism through what are essentially mailboxes, supporting a *multiple producer and single consumer* scenario.

4.2.2 Modular Main Function

Using the paper *"A Survey of Distributed Consensus Protocols for Blockchain Networks"* [31] as reference, the *functor* presented in Listing 4.1 was defined.

In OCaml, *functors* are functions that receive one or more modules as input arguments and produce a new module as output. In this scenario, it allows us to define a single top-level function that can be used in all protocols by using different combinations of input modules, which we defined as being the modules that directly map the behavior of a protocol according to the five component framework.

It is necessary that the modules must operate over the same data types, as specified in lines 2-8, however the behavior encapsulated in each module should be independent, allowing them to be easily exchanged.

Listing 4.1: Main Functor

```
1  module Main
2    (BlockProposal : Proposal)
3    (BlockValidation : Validation with type block = BlockProposal.block and type proof =
       ↪ BlockProposal.proof)
4    (BlockPropagation : Propagation with type block = BlockProposal.block and type proof =
       ↪ BlockProposal.proof)
5    (BlockFinalization : Finalization with type block = BlockProposal.block and type
       ↪ blocktree = BlockValidation.blocktree and type transaction = BlockProposal.
       ↪ transaction)
6    (IncentiveMechanism : Incentive)
7    (NetworkOperations : Network with type block = BlockProposal.block and type transaction
       ↪  = BlockProposal.transaction and type proof = BlockProposal.proof)
8    (CheckpointFinalization : Checkpoint with type blocktree = BlockValidation.blocktree) =
       ↪  struct
9
10   let run bc txs checkpointTree receiveChannel networkChannels = begin
11     let tmpBlockSet : (BlockProposal.block list ref) = ref [] in
12     let rec main () =
13       match (BlockProposal.propose_block txs) with
14         | Some (blk, proof) ->
15           tmpBlockSet := blk::!tmpBlockSet;
```

```
16          BlockPropagation.propagate_block (blk, proof) networkChannels
17        | None -> ();
18      match (NetworkOperations.receive_transaction receiveChannel) with
19        | Some tx ->
20          txs := tx::!txs
21        | None -> ();
22      match (NetworkOperations.receive_block receiveChannel) with
23        | Some (blk, proof) ->
24          begin
25            match (BlockValidation.validate_block (blk, proof) !bc) with
26            | true ->
27              tmpBlockSet := blk::!tmpBlockSet;
28              BlockPropagation.propagate_block (blk, proof) networkChannels
29            | false -> ()
30          end
31        | None -> ();
32      bc := BlockFinalization.finalize_block tmpBlockSet !bc txs;
33      checkpointTree := CheckpointFinalization.finalize_checkpoint !bc !checkpointTree;
34      main () in
35    main ()
36  end
37
38 end
```

The arguments of the *run* function are the state stored by each node. For the moment, that state is the following:

1. bc - a reference to a blockchain or blocktree data structure.
2. txs - a reference to a set of transactions, where the node will store the transactions that haven't yet been included in a block.
3. checkPointTree - a reference to the checkpoint tree data structure.
4. receiveChannel - the communication channels that the node can use to receive messages.
5. networkChannels - the communication channels of the known peers, to whom the node can send messages to.

The recursive *main* function is the main loop executed by each peer in the network.

It starts by running one iteration of its block proposal mechanism (lines 13-17). If that iteration succeeds, it must produce a block and an associated proof. The block gets added to the set of temporary blocks, and the (block, proof) pair is propagated to the peers.

Next, it attempts to receive a transaction (lines 18-21), if there is one to be received, and adds it to the set of transactions.

It then attempts to receive a block, if there is one to receive, and proceeds to validate it (lines 22-31). If the block is valid, it gets added to the set of temporary blocks and propagated to the peers.

Finally, it executes block finalization (line 32) which receives the blocktree and the set of temporary blocks, and returns an updated blocktree.

A similar procedure is done for checkpoint finalization (line 33), which receives a blocktree and a checkpoint tree, and produces an updated checkpoint tree.

Note that this *functor* is a good foundation, however, it must be further developed. At the moment it has some limitations, such as:

1. some parameters must be abstracted as different protocols may receive different inputs. For example, the `propose_block` operation must receive a *nonce* in proof of work, and a *round* in proof of stake. A possible solution might be including an abstract type that will contain the necessary arguments for each operation, and store those arguments in the node's state.

2. transactions and blocks could be received continuously until there are none left in the channels to be received.

3. the incentive mechanism module is currently not being used.

As mentioned, to use the presented *functor*, one must implement the required input modules, which were defined according to the Five Component Framework. The Five Component Framework was presented in more detail in Chapter 2, as well as an explanation of the behavior each of the five components encompasses in the Bitcoin, Algorand and Ethereum protocols.

4.3 Results

The previous two sections presented an overview of the practical work that was done in preparation for the thesis.

The implementation of the different consensus protocols in Rust lead to a better understanding of consensus protocols and what is required to transform their pseudo-code descriptions into working implementations. A large portion of each implementation wasn't specific to the behavior of the protocols themselves but rather the underlying mechanisms that are used to enable communication between peers. It was also seen that, although Multi-Decree Paxos and Raft may seem very different in their textual descriptions, their implementations share a lot of similarities.

The definition of the module signatures for essential auxiliary functions in blockchain protocols, as well as the main *functor* in OCaml, set an extensible foundation upon which the simulation environment will be built, even though some limitations are still present.

Chapter 5

Work Plan

The following chapter presents a tentative plan for the work that will be developed to achieve the goal of the thesis.

5.1 Overview

The project is divided into six major development phases. The timeframes to complete each phase are summarized in figure 5.1.

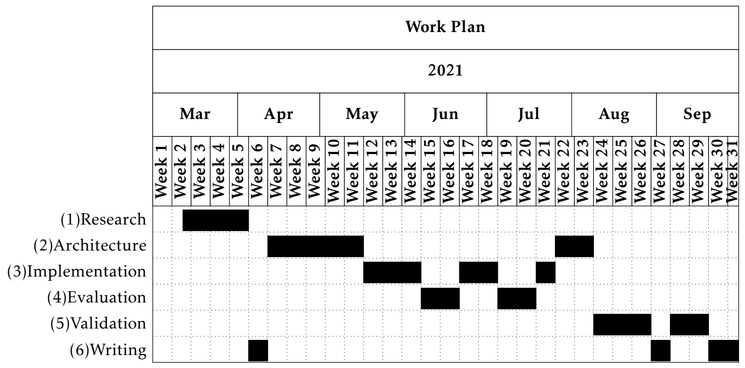

Figure 5.1: Visualization of the allocated timeframes for each phase of development.

5.2 Phases of Development

The following is a more detailed explanation of what will be done in each phase of development.

Phase 1 - Further Research into State of the Art - 3 Weeks

The first phase will consist of performing further research into the state of the art, building on top of the knowledge already acquired in the preparation phase.

An emphasis will be placed on exploring different simulation techniques and best practices since this will be crucial for phase two.

Three weeks should be an appropriate amount of time, considering that some level of experimentation will be required to not only learn different simulation techniques but also to better understand which will be the most appropriate for our use case. This phase may also lead to an overhaul of the function signatures that have been defined.

Phase 2 - Simulator Architecture - 6 Weeks

The second phase will consist of developing the simulator's architecture as well as implementing its core functionalities, building a solid foundation for phase 3.

More precisely, the result of this phase should be a simulation environment where it is possible to simulate different nodes that can communicate among themselves in a network, as well as a mechanism for extracting and aggregating relevant data from each node.

Phase 3 - Implementation of Blockchain Protocolos - 6 Weeks

Phase three will involve implementing blockchain-specific functionalities.

The data structures and data structure operations will be implemented, as well as essential operations required to simulate the Bitcoin, Algorand and Ethereum protocols - the necessary block proposal, block validation, information propagation, block finalization, and incentive mechanism modules for each of the protocols.

Phase 4 - Evaluation of the Implementation - 5 Weeks

The fourth phase will consist of evaluating the implementations made in the previous phases, through simulation.

The Bitcoin, Algorand and Ethereum protocols will be simulated, and an objective analysis of the results will be performed.

The timeframe of five weeks should be appropriate, to account for possible bug fixing, optimizations, and adding new functionalities if needed.

Phase 5 - Validation of the Simulation Environment - 5 Weeks

Phase five will be validating the developed simulation environment, through simulating some expected use cases.

The same protocols will be simulated, varying their parameterizations and analyzing their results.

Phase 6 - Writing the thesis - 5 Weeks

Finally, phase six will consist of writing the thesis document and preparing the final presentation.

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.

[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.

[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.

[4] M. Alves. *MSc Dissertation Preparation*. online. 2021. url: https://github.com/mce-alves/MSc-Dissertation-Preparation/tree/master/preparation/prototypes.

[5] K. J. A. ANDERSEN and I. SERGEY. “Protocol combinators for modeling, testing, and execution of distributed systems”. In: *Journal of Functional Programming* 31 (2021), e3. doi: 10.1017/S095679682000026X.

[6] 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.

[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.

[8] V. Buterin. *Ethereum Whitepaper*. online. 2013. url: https://ethereum.org/en/whitepaper/.

[9] V. Buterin and V. Griffith. "Casper the Friendly Finality Gadget". In: *CoRR* abs/1710.09437 (2017). arXiv: 1710.09437. URL: http://arxiv.org/abs/1710.09437.

[10] T. D. Chandra, R. Griesemer, and J. Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*. Ed. by I. Gupta and R. Wattenhofer. ACM, 2007, pp. 398–407. DOI: 10.1145/1281100.1281103. URL: https://doi.org/10.1145/1281100.1281103.

[11] 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.

[12] 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.

[13] 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.

[14] A. S. Deshpande. "Design and Implementation of an Ethereum-like Blockchain Simulation Framework". MA thesis. Technical University of Munich, 2018.

[15] R. Documentation. *Module std::sync::mpsc*. online. accessed: Februrary 20th 2021. URL: https://doc.rust-lang.org/std/sync/mpsc/.

[16] Ethereum. *The Beacon Chain*. online. accessed: Februrary 15th 2021. URL: https://ethereum.org/en/eth2/beacon-chain/.

[17] C. S. F. Faria. "BlockSim: Blockchain Simulator". MA thesis. Instituto Superior Técnico, 2018.

[18] 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.

[19] 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.

[20] H. Howard and R. Mortier. “Paxos vs Raft: have we reached consensus on distributed consensus?” In: *7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2020, Heraklion, Greece, April 27, 2020*. Ed. by A. D. Fekete and M. Kleppmann. ACM, 2020, 8:1–8:9. DOI: 10.1145/3380787.3393681. URL: https://doi.org/10.1145/3380787.3393681.

[21] L. Lamport. *Paxos Made Simple*. online. 2001. URL: https://lamport.azurewebsites.net/pubs/paxos-simple.pdf.

[22] Y. A. Liu, S. Chand, and S. D. Stoller. “Moderately Complex Paxos Made Simple: High-Level Executable Specification of Distributed Algorithms”. In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by E. Komendantskaya. ACM, 2019, 15:1–15:15. DOI: 10.1145/3354166.3354180. URL: https://doi.org/10.1145/3354166.3354180.

[23] Y. A. Liu, S. D. Stoller, and B. Lin. “From Clarity to Efficiency for Distributed Algorithms”. In: *ACM Trans. Program. Lang. Syst.* 39.3 (2017), 12:1–12:41. DOI: 10.1145/2994595. URL: https://doi.org/10.1145/2994595.

[24] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. online. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[25] D. Ongaro. *Raft TLA Specification*. online. accessed: Februrary 19th 2021. URL: https://github.com/ongardie/raft.tla.

[26] D. Ongaro and J. K. Ousterhout. “In Search of an Understandable Consensus Algorithm”. In: *2014 USENIX Annual Technical Conference, USENIX ATC ’14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by G. Gibson and N. Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[27] R. van Renesse and D. Altinbuken. “Paxos Made Moderately Complex”. In: *ACM Comput. Surv.* 47.3 (2015), 42:1–42:36. DOI: 10.1145/2673577. URL: https://doi.org/10.1145/2673577.

[28] G. Stolpmann. *Ocamlnet 4 Reference Manual*. online. accessed: Februrary 20th 2021. URL: http://projects.camlcity.org/projects/dl/ocamlnet-4.1.8/doc/html-main/index.html.

[29] L. Stoykov. “VIBES: Fast Blockchain Simulations for Large-scale Peer-to-Peer Networks”. MA thesis. Technical University of Munich, 2018.

[30] G. Wood. *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*. online. 2021. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[31] 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.

Appendix A

OCaml Module Signatures

Listing A.1: Network Module

```
module type Network = sig
    (* the type of blocks included in the messages *)
    type block
    (* the type of proof that is associated with a block *)
    type proof
    (* the type of a transaction *)
    type transaction
    (* the type of the messages sent in the network *)
    type message
    (* the communications channels used for message exchange *)
    type communication_channels

    (* minimum delay when sending a message *)
    val min_delay : float
    (* maximum delay when sending a message *)
    val max_delay : float
    (* chance to fail when sending a message *)
    val fail_chance : float
    (* uses the min,max delay bounds, and returns a delay in that interval *)
    val compute_delay : float
    (* sends a message to the provided communication channels *)
    (* A message can be a block*proof, transaction, etc *)
    (* An appropriate channel for the message type will be selected via pattern matching
        ↪ *)
    val send_message : message ->(communication_channels list) ->unit
    (* try to receive a new transaction *)
    val receive_transaction : communication_channels ->(transaction option)
    (* try to receive a block from the network *)
    val receive_block : communication_channels ->((block * proof) option)
```

```
end
```

Listing A.2: Blocktree Module

```
    module type Blocktree = sig
    (* the type of the blocks in the block tree *)
    type block
    (* the type of the hash of a block *)
    type hash
    (* the type of the block tree's implementation *)
    type blocktree

    (* the genesis block *)
    val genesis : block
    (* the initial blocktree, containing only the genesis block *)
    val init : blocktree
    (* write a block to the tree *)
    val write_block : block ->blocktree ->blocktree
    (* get the head of the best (heaviest) chain in the blocktree *)
    val get_best_chain_head : blocktree ->block
    (* returns the current best chain *)
    val get_best_chain : blocktree ->(block list)
    (* given a blocktree and two blocks (heads of different chains) *)
    (* returns which should be considered the heaviest chain *)
    val fork_choice_rule : blocktree ->block ->block ->block
    (* ensures that adding the block to the tree won't cause cycles *)
    (* true==no cycles *)
    val no_cycles : blocktree ->block ->bool
    (* checks if the blocktree already contains a block with the same hash *)
    val contains_hash : blocktree ->hash ->bool

end
```

Listing A.3: Proof of Work Module

```
module type ProofOfWork = sig
    (* the type of the block's implementation *)
    type block
    (* the type of the hash *)
    type hash
    (* the type of the nonce *)
    type nonce
    (* the type of the acceptance threshold *)
    type threshold

    (* the initial nonce that should be used *)
    val init_nonce : nonce
    (* receives a nonce and returns the next one to be used *)
    val next_nonce : nonce ->nonce
    (* computes a hash, given a block and a nonce *)
    val compute_hash : block ->nonce ->hash
    (* accepts (true) or refuses (false) a hash by comparing it to a threshold *)
    val acceptance_function : hash ->threshold ->bool

end
```

Listing A.4: Proof of Stake Module

```
module type ProofOfStake = sig
    (* the type of the blocktree/blockchain *)
    type blocktree
    (* type of an asynchronous round (slot for which the block will be proposed) *)
    type round
    (* the type of the private and public seeds/keys *)
    type key
    (* the type of the hashes *)
    type hash
    (* type of the structure storing the system's users and their stake *)
    type stakes
    (* the type representing a step in the agreement protocol *)
    type step
    (* the type of the proof produced by the algorithm *)
    type proof

    (* returns the users and their corresponding initial stakes (weights) *)
    val weights : stakes
    (* given a peer's private key, a round and step in the protocol, generates a hash and
        ↪  proof if the peer belongs to the committee for that round and step *)
    val committee_sortition : key ->round ->step ->((hash * proof) option)
    (* given a peer's private key and a round, generates a hash and proof if the peer was
        ↪  selected to proposer a block in that round *)
    val proposer_sortition : key ->round ->((hash * proof) option)
    (* given a peer's private key, a round and step in the protocol, and a hash*proof
        ↪ credential, returns whether or not the credentials are valid and belong to the
        ↪  public key's owner *)
    val committee_validation: key ->round ->step ->(hash * proof) ->bool
    (* given a peer's private key, a round in the protocol and a hash*proof credential,
        ↪ returns whether or not the credentials are valid and belong to the public key'
        ↪ s owner *)
    val proposer_validation: key ->round ->(hash * proof) ->bool

end
```