San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Spring 2023

Proof-of-Stake for SpartanGold

Nimesh Ashok Doolani
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

 Part of the Information Security Commons, and the Other Computer Sciences Commons

Recommended Citation

Doolani, Nimesh Ashok, "Proof-of-Stake for SpartanGold" (2023). *Master's Projects*. 1222.
DOI: https://doi.org/10.31979/etd.tp78-7dey
https://scholarworks.sjsu.edu/etd_projects/1222

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Proof-of-Stake for SpartanGold

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Nimesh Ashok Doolani

May 2023

© 2023

Nimesh Ashok Doolani

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Proof-of-Stake for SpartanGold

by

Nimesh Ashok Doolani

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2023

Dr. Thomas Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Katerina Potika	Department of Computer Science

ABSTRACT

Proof-of-Stake for SpartanGold

by Nimesh Ashok Doolani

Consensus protocols are critical for any blockchain technology, and Proof-of-Stake (PoS) protocols have gained popularity due to their advantages over Proof-of-Work (PoW) protocols in terms of scalability and efficiency. However, existing PoS mechanisms, such as delegated and bonded PoS, suffer from security and usability issues. Pure PoS (PPoS) protocols provide a stronger decentralization and offer a potential solution to these problems. Algorand, a well-known cryptocurrency, employs a PPoS protocol that utilizes a new Byzantine Agreement (BA) mechanism for consensus and Verifiable Random Functions (VRFs) to securely scale the protocol to accommodate many participants, making it possible to handle a growing number of clients with ease. In this research, we explore, implement, and document all the essential steps of the algorithm for any given round that leads to publishing a block, and we evaluate the performance and stability of Algorand using various numbers of users, their stakes, and network settings. To simulate the protocol, we extend the SpartanGold blockchain framework, which currently uses a PoW protocol, and convert it into a PoS model. Our results show that the PPoS protocol developed by Algorand is highly scalable, achieving consensus quickly and efficiently, even in the presence of malicious users or network partitions and offers higher security and Byzantine fault tolerance compared to traditional PoW and other PoS-based protocols.

KeyTerms: Blockchain consensus, Proof-of-Stake (PoS), Proof-of-Work (PoW), Algorand, Verifiable Random Functions (VRFs), Byzantine Agreement, SpartanGold.

ACKNOWLEDGEMENTS

I would like to convey my gratitude to my advisor Dr. Thomas Austin for his assistance and guidance during the course of this project. I would also like to express my appreciation towards the committee members Dr. Chris Pollett and Dr. Katerina Potika, for their inputs and feedback on this project.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Blockchain and cryptocurrencies are currently among the most trending technologies. Almost everyone has heard of Bitcoin, Ethereum, or NFTs. These technologies and protocols utilize blockchain as their core infrastructure. However, the concept of blockchain may not be clear to those who are new to it. This research paper explores various blockchain protocols and delves into Algorand's algorithms for achieving consensus. Therefore, it is essential to cover the basic concepts that lead up to the main concept.

1.1 What is a blockchain?

A blockchain, as the name suggests, is precisely a series of blocks or records appended one after another to form a serial link. It is a ledger that stores irrefutable data. These blocks are special data structures designed to store transactions or data in a cryptographically secure manner using hash trees, allowing them to be verified by everyone. These blocks specifically store the timestamp, hash of their previous block and transactional data, as mentioned before. The timestamp acts as a proof of existence of data at the time of creation of the block. The data on the blockchain cannot be modified because the inherent design links all blocks to each other right up to the genesis block. And to forge transactions in a given block would require the attacker to recompute the entire blockchain, which statistically is close to impossible.

Figure 1 describes a basic skeleton of a typical blockchain. Blocks are linked to each other, and each block contains relevant information about transactions, nonce, timestamp, proof, and the previous block hash.

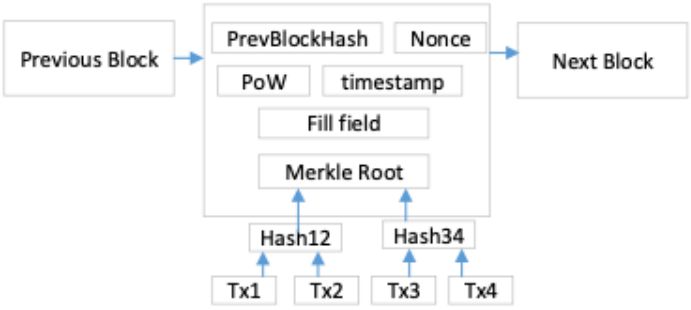

Figure 1: A basic blockchain skeleton. Copied from [1]

1. 2 Architecture

A blockchain is a decentralized peer-to-peer distributed architecture that relies heavily on the network majority to reach consensus to publish blocks. The network is setup with users as clients or nodes who vote, verify, and certify these blocks. The rationale behind this decentralization is to avoid an all-powerful central authority managing every aspect of the network. The problems with having power concentrated in a small group or individual are:

1. **Security risks:** It becomes easier to target and compromise a single entity that has sole control over everything.
2. **Lack of accountability:** Centralized control reduces the level of transparency which can affect fairness and accountability.
3. **Censorship:** The central authority may also have the power to censor or deny services to some users.

Although the blockchains are designed to operate in primarily one specific setting, there are other types as well:

1. **Public:** These are the main and the most sought-after types. There are no restrictions on their access. Any user is free to join and participate in the network.

The network is capable of handling bad actors and therefore does not require special guarding against them.

2. **Private:** The participants and other users experience restrictions in the blockchain and there are access controls set up by owners or administrators.
3. **Hybrid:** A flavor of blockchain that employs features of both public and private blockchain.

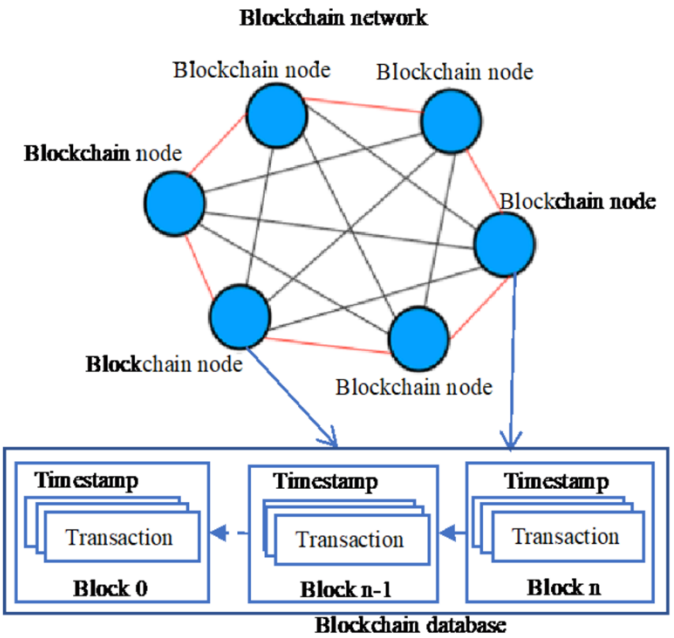

Figure 2: Overview of blockchain architecture. Copied from [2]

1.3 Consensus Protocols

Every distributed or peer-to-peer (P2P) architecture requires agreement between nodes. A lot of applications like cloud computing, clock synchronization, etc. are designed in a P2P fashion, which must deal with consistency problems arising due to concurrency and replication. In blockchain, similarly, consensus protocols are used by the nodes to come to an agreement on the blocks that get published to the chain. These protocols have important requirements that need to be fulfilled:

1. **Safety**: the protocols should ensure that transactions written to the blockchain cannot be reversed.
2. **Liveness**: the protocols ensure the network eventually progresses and publishes blocks.
3. **Fault tolerant**: the protocols should be tolerant of faulty nodes without affecting the system performance too much.
4. **Scalability**: the protocols should be able to scale to a large number of participating nodes.
5. **Security**: the protocols should be resilient against attacks like denial of service (DoS) or Sybil attacks.
6. **Energy efficient**: converging to an agreement should require efficient and sustainable use of energy.

Ismail and Materwala [3] divide consensus protocol into categories such as Compute-Intensive Based, Capability-Based, and Voting-Based. There are a lot of consensus protocols used in blockchains and cryptocurrencies, but the main ones can be categorized into these types:

1.3.1 Proof of Work

Proof-of-Work (PoW) [4] is probably the oldest consensus mechanism used in blockchains to achieve consensus and mine cryptocurrencies. In PoW, **miners** or participating nodes compete to solve complex mathematical problems using their computational power. The miner who solves the problem first gets to publish the block and transactions and in return is awarded with new cryptocurrency. The solution to the problem is called the proof of work.

After solving the problem, the miners share the solution with other miners who can easily validate the proof to the problem. The rationale behind this is to ensure that miners publish only valid transitions to the blockchain, and this can only be achieved by successfully expending computational resources to solve the problem. This also ensures that any wrongdoing or forging would require miners to recompute every block in the blockchain to rewrite history.

Advantage - This feature offers one prime advantage to the system - security. Mining a block is challenging and that makes it difficult for bad actors to cheat the network. This also prevents against Sybil attack [5] as creating multiple identities doesn't provide any advantage unless they all can provide computational resources to solve the problem. The complexity of the problems increases per round as only a limited number of coins can be mined. However, the difficulty is also tweaked periodically based on the competition to ensure blocks are published at a steady pace.

Disadvantage - Spending computational resources directly translates into high energy consumption and that is the biggest drawback of this protocol. The requirement increases substantially as the problem complexity increases. Another criticism of PoW is that miners pooling their resources together to achieve a high combined mining power can introduce a level of centralization into the network. Large mining pools can dominate the network and control the consensus.

1.3.2 Proof of Stake

This consensus protocol works differently than PoW protocols which uses computational power. As the name suggests, the participation depends on the individual stake of the users. PoS [6] systems have the concept of **validators** that are chosen randomly based on the amount of cryptocurrency they stake in the network. Validator nodes validate

all transactions that appear in the blockchain, and any malpractice would result into them losing all the stake as penalty.

Instead of solving complex mathematical problems or puzzles, validators are randomly elected based on their stake in the system. In exchange for honestly validating transactions to publish blocks, validators are rewarded with transaction fees proportional to the total cryptocurrency they have staked.

Advantage - PoS is energy efficient. Since it requires less than a fraction of computational power of PoW, this protocol requires minimal energy consumption. This is quite possibly the biggest advantage PoS systems have over other protocols. Transaction processing and validation and publishing blocks is extremely fast since there is no puzzle-solving involved.

Disadvantage - PoS systems are somtimes considered to be less secure than PoW systems due to greater potential for centralization resulting from the lower cost of acquiring a stake in the network. Additionally, PoS is theoretically susceptible to bad actors who could influence validators to vote in their favor. Wealthy actors who own a significant stake in the network may have greater decision-making power over blocks.

One specific problem with PoS is the "nothing-at-stake" problem [7], where validators could potentially validate blocks on multiple branches of a fork without penalty. This could lead to a situation where the blockchain fails to progress in a single direction while validators continue to receive rewards. The solution involves imposing penalties for this behavior; some PoS systems require participants to set aside some cryptocurrency (the stake) as a surety bond.

Figure 3 compares the leader election process to publish blocks in PoW and PoS system. PoW uses computational contribution from clients and PoS uses clients' stake as a direct alternative.

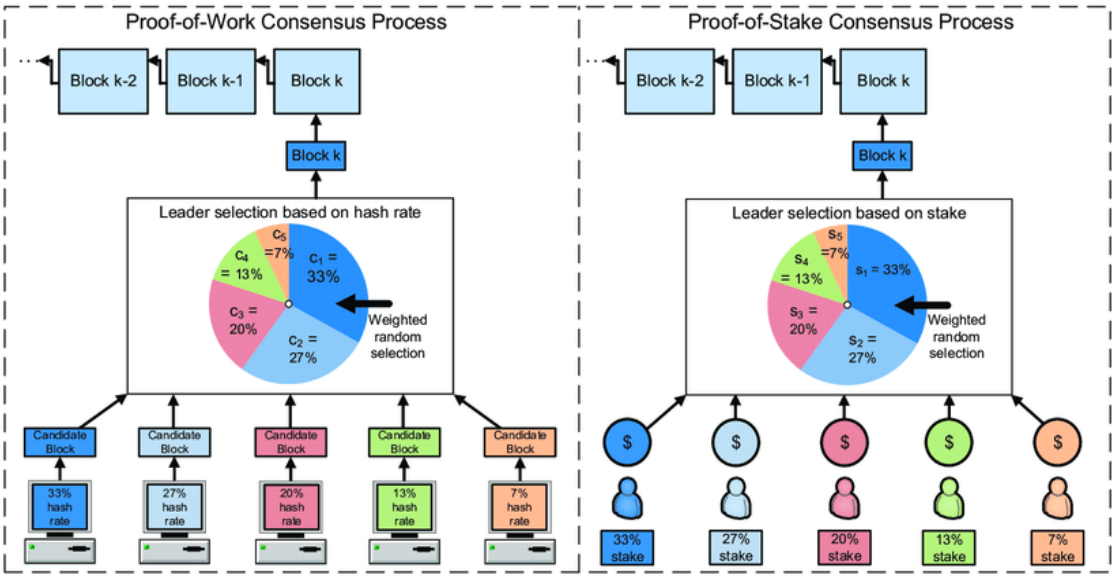

Figure 3: Contrasting PoW and PoS Mechanisms. Copied from [8]

1.3.3 Delegated Proof of Stake

Delegated Proof of Stake (DPoS) [9] is a variant of PoS where validators vote for an election committee or delegates who they trust to act in the best interest of the network. The votes are proportional to their stake in the network. This committee is of a fixed size and is responsible for proposing and validating blocks. This way, validators delegate their duties to a special set of validators. After successfully publishing blocks, the committee members can distribute the block rewards among the clients who voted for them.

Advantage - DPoS is a reputation-based model that allows clients to remove bad-actor delegates by voting for new delegates to join the committee. Since the number of nodes that participate in the actual consensus is limited, DPoS can achieve extremely fast block publishing times and requires minimal energy consumption.

Disadvantage - Delegating the power to propose, validate, and publish blocks to a fixed set of clients does pose the potential for centralization. Power is concentrated and breaking the consensus would only require collusion among only a small number of nodes (delegate pool).

1.3.4 Pure Proof of Stake

Algorand's Pure Proof of Stake (PPoS) [10] protocol is an improved version of the traditional PoS model. Validators are selected to propose blocks based on the stake they have in the network, but unlike in traditional PoS, they do not have to set aside any cryptocurrency to participate in the consensus. The protocol uses the total wallet value of each validator to determine their influence in proposing blocks. Validators are deterministically and secretly selected to propose and validate blocks, and the protocol and supporting algorithms are designed to allow all nodes in the network to participate in this process.

Tendermint [11] also employs a PPoS where consensus depends only on the stake, but validators are required to bond coins to gain right to produce blocks.

Advantage - PPoS systems offer significantly more energy efficiency than PoW and provide more randomness, security, and decentralization than traditional PoS.

1.4 Incentives

Incentives play a huge role in keeping the blockchain network secure. In a distributed system like blockchain where every decision is taken by nodes in consensus, incentives act as motive for participants to contribute to the network by proposing and validating blocks and preventing against attacks. Incentives can be roughly classified into two groups:

1. **Monetary based:** these incentives are designed to engage participants from an economic standpoint, with PoW rewarding users with block rewards for finding the proof in the form of cryptocurrency and PoS rewarding the validators with transaction fees. The ultimate goal of monetary-based incentives is to ensure total gains from rewards outweigh the cost to act selfishly and jeopardize the network.

2. **Non-monetary based**: these incentives affect the reputation and credibility of clients in the network. These appeal to the emotional state of the participants and rewards and encourages them for good behavior.

Research in [12] proposed interesting requirements for an effective incentive mechanism and used them to evaluate existing blockchain versions. Some of the requirements include:

1. **Individual Rationality (IR)**: The system should reward rational behavior, as most users are not dishonest but are driven by rewards and benefits. Incentive mechanisms need to ensure that the benefits outweigh the cost of participating or any dishonest activity.
2. **Incentive Compatibility (IC)**: The system can function smoothly if the mechanism can ensure that individual interests are compatible with the general interests of the group or the whole network in general.
3. **Incentive Fairness (IF)**: This highlights that the mechanism should be fair in rewarding clients duly and that the rewards should be proportional to their contribution.

1.5 Guide to Paper

The rest of this paper is structured as follows: immediately following the blockchain basics, we present the problem statement that this paper aims to address, as well as the objectives and outcomes of this project. Chapter 2 provides the literature review for the background and related work. Chapter 3 is a comprehensive review of Algorand and its Byzantine consensus protocol. Chapters 4 and 5 cover the design and implementation of Algorand for SpartanGold. Chapter 6 evaluates various undesirable scenarios and discusses

how the algorithm handles them. Finally, Chapters 7 and 8 summarize the project and highlight areas for future work.

1.6 Problem Statement

SpartanGold currently uses a PoW consensus model, which suffers from scalability and high energy consumption. It also faces a 51% attack problem, where miners continue to mine in single-threaded mode before checking for any messages as long as their mining rounds or power. In SpartanGold semantics, mining power directly translates to miners' computational power. Therefore, there is a need to implement a PoS protocol using SpartanGold as the base.

Almost all PoS protocols have one major disadvantage: setting stake aside to participate in the network. Although the threat of losing assets works as a great incentive for participants to function honestly, it is undesirable for users to set aside a huge portion of their capital with no returns on investment just for the sake of the network and a slender chance of ever publishing a block. Algorand's pure PoS protocol solves this issue by enabling users to utilize their total stake without actually staking. Their Byzantine Agreement consensus protocol can be scaled to many users in the network using Verifiable Random Functions (VRFs), where users can check privately if they are chosen in the lottery. It also shows how the system can handle partitions either on the network side or in the presence of an adversary.

This research project aims to leverage the advantages of Algorand's novel features while also maintaining SpartanGold's cryptographic primitives and converting it into a PoS model.

1.7 Project Objectives

This primary objective of this research project is to implement and simulate Algorand's pure PoS model. There are some other specific objectives of this research and implementation:

1. To develop a novel consensus protocol for SpartanGold replacing its PoW protocol with a pure PoS model based on Algorand's Byzantine Agreement consensus protocol.
2. Implement Algorand's PoS model on top of SpartanGold by extending and overriding the clients, blocks and blockchain primitives while also preserving the base setup of SpartanGold.
3. Execute and demonstrate the working of Algorand for multiple rounds, highlighting how the algorithm progresses at every stage and reaches tentative or final consensus.
4. Provide a mechanism for publishing empty blocks in rounds where no block proposers can be obtained.
5. Experiment with different number of users and account balances, delays in messages, and network partition to how the algorithm progresses and detects possible forks.
6. Provide documentation and open-source code for the new consensus algorithm to enable others to replicate and extend the research.

CHAPTER 2

Related Work

2.1 Bitcoin

Bitcoin was the first protocol to implement a decentralized blockchain or ledger. It was invented in 2008 by an anonymous user or group called Satoshi Nakamoto and was officially introduced to the public in their whitepaper [13]. The unit of this cryptocurrency is called "bitcoin." The term with the capitalized first letter "B" denotes the protocol, while the lowercase is reserved for the cryptocurrency. The smallest unit is called a *satoshi*. One bitcoin is equal to 100 million *satoshis*. Bitcoin served as an inspiration for the invention and development of all other blockchain-based cryptocurrencies.

Bitcoin introduced a means of electronic transactions that does not require any trust-based model. It is composed of currency developed from digital signatures of transactions chained together, and these transactions are publicly recorded in blocks. Any transfer of currency requires digitally signing the hash of the previous transaction and the public key of the receiver and appending it to the coin. These blocks or transactions are almost impossible to manipulate by attackers if a significant portion of the network is controlled by honest nodes. Bitcoin uses a peer-to-peer system that allows members to join and leave at will. Any mining or validation of blocks requires CPU computation that involves solving mathematical problems and validating their proofs.

Figure 4 shows how the transactions are chained using the hash and signature of their previous transactions.

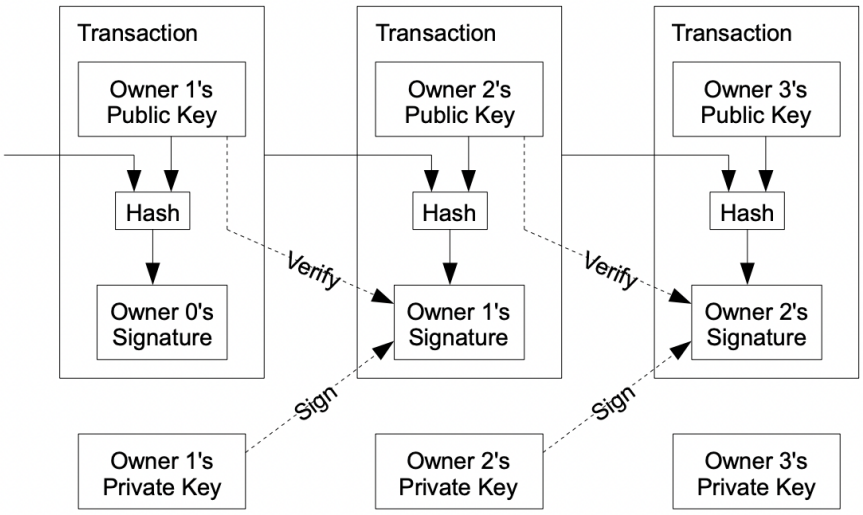

Figure 4: Chaining of transactions. Copied from [13]

Proof-of-work - Bitcoin uses a PoW protocol for consensus in its distributed protocol. The miners search for a value that, when hashed, would generate a hash with several leading zeros. A nonce is incremented in the block to produce a hash until a value is found that is less than the required difficulty target or has the required number of zero bits. The value of the target is so low that most of the hashes produce many leading zeros. For example, a hash could be:

`0000000000000000000ecf87bf07dcd21dc0d`

The difficulty is adjusted so that it would require a considerable amount of computation to find the proof, but verification would be inexpensive. Once blocks are published, an attacker would need to expend the same amount of computation that was required in the first place. This is designed to thwart any attempts to modify the blocks or transactions. If anyone had to change any past blocks, they would need to recalculate that block and all blocks after it. The amount of computation and energy required to do this is exponential.

2.2 Ethereum

Ethereum is a decentralized blockchain technology that became famous after Bitcoin. Currently it has the second highest market cap after Bitcoin. The cryptocurrency used on the Ethereum blockchain is called Ether. In 2013, Vitalik Buterin and other founders published a comprehensive whitepaper [14] describing Ethereum's design.

Ethereum provides a smart-contract feature, which allows users to design and deploy programs that perform transactions based on an agreement on behalf of clients. It also allows for the development of decentralized applications (DApps) on top of it. Ethereum created a special standard for non-fungible tokens (NFTs) [15], which are tokens that can be tied to any digital media on the blockchain and are considered a unique artifact that exists at that moment in time.

The intent behind Ethereum was to provide a blockchain technology that employs a Turing-complete programming language that enables users to create and deploy smart contracts on the fly to expand the use cases to DApps, digital assets, smart contracts, NFTs, etc.

One of the key components of Ethereum is the Ethereum Virtual Machine (EVM), which is a runtime environment. It allows developers to write smart contracts in a high-level language, mainly Solidity, and use Ethereum to execute them. EVM provides a stack, memory, gas, program counters, and more.

Gas is the unit of measurement for the computation required to execute anything on the EVM. Transaction fees are calculated using gas. The operations that can be performed in smart contracts are pre-defined with a gas price. Every transaction requires the sender to include the gas price and the gas limit to have that transaction included in the blockchain.

The transaction fees paid to the block proposer are specified by the sender's gas. The excess gas acts as a tip or an incentive for the block proposers.

Ethereum's mining under PoW works differently than Bitcoin's PoW. It involves random computation from the state, computing random transactions from the past N blocks, and calculating the hash. This is advantageous in two ways: first, smart contracts can require any level of computation, so there is no need for ASICs, and second, since mining requires every node to have access to the entire blockchain and validate transactions, there is no need for mining pools.

Ethereum initially used a PoW consensus model, but the movement to a PoS model, called The Merge, occurred on September 15, 2022. This was a significant advancement in the blockchain technology space in terms of moving toward an energy-efficient and environmentally friendly green consensus mechanism. The energy consumption required to run Ethereum dropped by nearly 99.95%, which is a remarkable achievement.

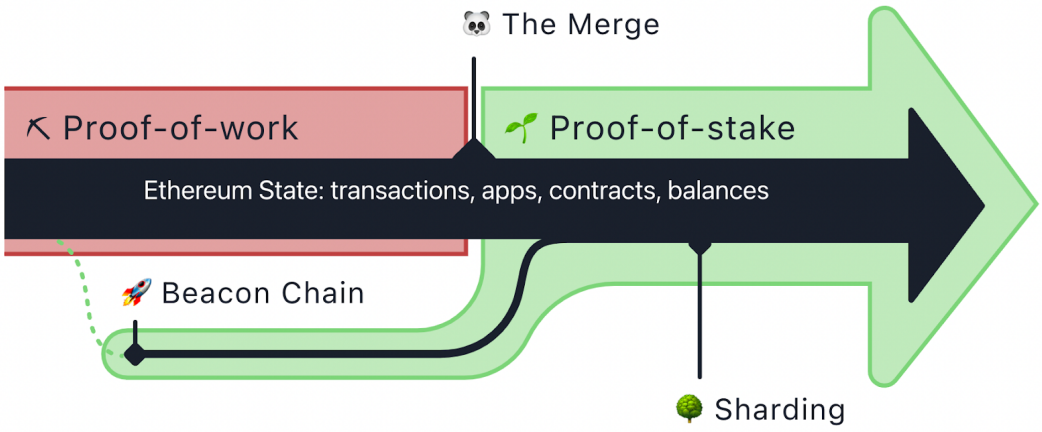

Figure 5: Ethereum’s The Merge. Copied from [16]

Figure 5 pictures Ethereum’s official movement from its PoW, Mainnet, to its new PoS layer, Beacon Chain. The Merge also paved the way for future updates like sharding.

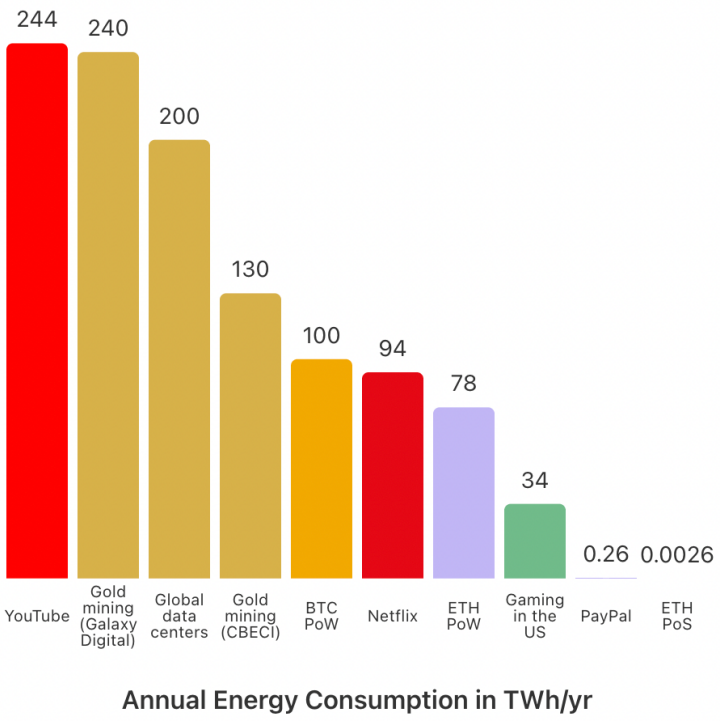

Figure 6: Annual energy utilization. Copied from [17]

Figure 5 shows the comparison of energy consumption between Ethereum PoS and the other biggest technologies in the world.

2.3 Proof of Activity

Proof-of-Activity (PoA) is a newer consensus protocol developed by Iddo Bentov et al. [18]. It is a hybrid model that combines the PoW and PoS models and consolidates their security levels. PoW gives power to those who expend computational resources, while PoS gives power to those who have a stake in the network. However, PoW suffers from mining pools and dedicated data centers set up to outcompete other miners, and many PoS systems suffer from centralization and power concentration in a few validators. PoA addresses these issues by combining the critical aspects of both these protocols.

In PoA, nodes are required to perform more complex mathematical computations than other PoW network nodes. One of the key principles in PoA is called **"follow-the-satoshi."**

This technique involves converting a pseudorandom value into a *satoshi* that is drawn from a pool of all the *satoshis* ever minted or generated. The basic idea is to choose a random value between zero and the total number of *satoshis*, find the block where this *satoshi* was generated, and then traverse all the transactions where this *satoshi* was transferred to ultimately find the current owner.

Mining and validation in PoA combine the best of both worlds of PoW and PoS. The basic flow can be described as follows:

1. Miners solve mathematical puzzles to find the proof that is smaller than the target.
2. The proof is broadcasted to all the nodes for verification.
3. All the nodes use follow-the-satoshi and the block hash to realize N stakeholders.
4. Every online node checks if they belong to the N stakeholders. If they do, they sign the block with the private key that holds the drawn satoshi and broadcast their signature.
5. After other nodes verify all N signatures, they add the block to the chain, and the rewards are distributed between the miner and the validators or stakeholders.

The PoS in PoA helps reduce the centralization risk of PoW by mitigating the formation of mining pools. It also reduces the energy requirement associated with PoW. Some PoS systems faced the problem where even non-participating high-stake nodes continued to receive rewards. PoA tackles this by rewarding only active nodes in the system who maintain an online node and take active participation in the consensus.

By transferring the power to validate transactions to the stakeholders, PoA addresses the centralization problem by not solely relying on miners' CPU power. It enables participants with stakes to make decisions in the interest of the network, promoting more decentralization and ensuring higher security.

2.4 Ouroborus

Ouroboros is a blockchain protocol that employs proof-of-stake and is used by the Cardano public blockchain. Kiayias et al [19] claim in their research proposal titled "A Provably Secure Proof-of-Stake Blockchain Protocol" that Ouroboros is the first blockchain protocol that ensures rigorous security. The protocol is designed to be more energy-efficient than proof-of-work protocols and introduces a new incentive mechanism to prevent attacks like selfish mining.

Ouroboros divides time into epochs and slots, with each slot corresponding to one block. A leader is elected for every slot, and to prevent adversarial attacks, each leader is required to consider the last few blocks in the received chain temporarily. The chain that is longer than the predetermined number of historical blocks is accepted. Participants have to synchronize with the global clock that manages the epochs for everyone in the network and register with the network and a global random oracle beacon that transmits random values to all nodes.

To participate in the staking procedure, stakeholders use their secret key to calculate the Verifiable Random Function (VRF) to generate a hash and a proof. If the hash is below a threshold, the stakeholder is chosen as a block proposer for that slot. The proposer packs all the transactions, the VRF proof, generates a new secret key, and broadcasts the block to the network. The rewards are collected at the end of each epoch and distributed to all the stakeholders involved.

Ouroboros requires clients to be online for the leader election process. Stakeholders who can't be online all the time can delegate their duties to stake pools, which can work on their behalf. At the end of each epoch, rewards are distributed to all stakeholders proportionally to their stake in the pool. The use of random oracles and VRFs provides

randomness and fairness. The hashes produced during each slot are stored in the blocks and are used later to generate the seed for a future round.

To prevent centralization, Ouroboros employs a neat solution on staking pools. Staking pools can only work as delegates if they represent a minimum threshold of stake of the entire network. This thwarts a fragmentation attack where attackers might increase the number of staking pools and reduce the stake in each pool. If the staking pools exceed a certain threshold of the total stake, the mining rewards remain constant, making it less attractive to profit-maximizing miners.

In conclusion, Ouroboros is a highly secure and efficient proof-of-stake protocol that uses innovative mechanisms to prevent attacks and centralization. It's a key component of the Cardano blockchain network.

2.5 SpartanGold

SpartanGold is a JavaScript implementation of a blockchain cryptocurrency developed and authored by Prof. Thomas Austin in his research [20], who also happens to be the project advisor of this research paper. SpartanGold is a blockchain cryptocurrency developed for research and academic purposes, as well as for rapid prototyping. It is inspired by Bitcoin and highlights the main features of the famous blockchain while simplifying or eliminating several complex features.

It employs a proof-of-work consensus mechanism where miners are compensated with transaction fees for their work in mining and publishing blocks. Originally, it started with an unspent-transaction output (UTXO) model for all users but later moved to a simple account-based approach for simplicity and ease of understanding. In this model, every client and miner own coins and maintains a map of the account balance of all other clients

and miners. Any rewards or transactions are simply added or subtracted from the balance of the parties involved.

Mining in SpartanGold is simplified for ease of understanding. It requires miners to calculate and increment the block hash value until it is just lower than the target set for the blockchain. At this point, a proof-of-work value is technically found, and miners broadcast this on the network. After verification, rewards are distributed, and a new mining round begins to find the next block.

Some of the classes used in SpartanGold are:

1. *Transaction*: stores the relevant information for any transaction like fees, sender and receiver addresses, signature and public keys of the sender, etc.
2. *Block*: stores the balances and transactions.
3. *Client*: manage the clients in the system who take care of storing the keys, managing transactions, and storing the blocks in the network.
4. *Miner*: an extension of the *Client* that is responsible for mining and broadcasting the proof-of-work to all others.
5. *Blockchain*: stores the general settings and constants required for the simulation.

SpartanGold works in two modes:

1. **single-threaded**: Due to JavaScript's single threaded nature and its run-to-completion configuration, only one miner can be active at a time. In reality, mining is strictly concurrent. To handle this, miners execute exclusively for the amount of mining power they own. Once that is exhausted, other miners are switched on the thread to start or continue their mining.

2. **multi-process**: This mode overcomes the shortcomings of the single-threaded mode because it involves miners running over separate processes or instances and they communicate with each other over TCP/IP.

2.6 Verifiable Random Function

A Verifiable Random Function (VRF) is a cryptographic primitive that combines hash functions and public-key infrastructure. It was introduced by Micali et al. [21] in their research. The VRF generates a hash and a proof value. The hash is a pseudorandom value generated on an input, while the proof binds the hash value to the identity of the user. By using the user's public key and the proof, anyone can verify that the hash was indeed generated by the same user.

VRFs typically include these primitives:

1. secret key of the user: sk_u
2. public key of the user: pk_u
3. cryptographic random value generated by the VRF: *hash*
4. proof of the cryptographic hash generated using sk_u: π
5. input data for the hash function: α

A VRF can be demonstrated as:

$$\langle \text{hash}, \pi \rangle \leftarrow \text{VRF}_{sk}(\alpha)$$

VRFs are deterministic algorithms, meaning that they produce the same hash value and proof for a given pair of (secret key, α). VRFs should fulfill some predefined requirements. Some of these include:

1. **Full uniqueness:** for a given pair of (public key, α) there will exist only one hash value which can be proved successfully. There will never be 2 valid proofs for the same pair.

2. **Full Collison Resistance:** for a given secret key, it should be impossible to have 2 different input values that generate the same hash value.
3. **Full Pseudorandomness:** if an adversary observes an output hash without its proof, then the hash value should be indistinguishable from any other random value.

VRFs have a wide range of applications in cryptography. They can be used to generate pseudorandom values in lotteries or choosing block proposers in various Proof of Stake (PoS) protocols like Ouroboros. VRFs are also heavily used in cryptographic sortition in multiple stages of the Algorand consensus protocol. They are also used to prevent certain attacks, as described previously.

CHAPTER 3

Algorand

3.1 Introduction

As described in the previous section, the main objective of this research project is to implement Algorand. Therefore, this section provides a comprehensive review of the cryptocurrency. Y. Gilad et al. [22] claim that Algorand can confirm transactions with a latency as short as a minute while scaling to many users. Additionally, the possibility of a fork is negligible, even when the network is partitioned, whereas certain cryptocurrencies require significant time to recover to a safe state. Algorand proposes a novel Byzantine Agreement (BA) consensus protocol called BA⋆ that uses Verifiable Random Functions (VRFs), described in section 2.6, to cover a wide range of users. The users preserve only their secret keys and do not maintain any private state to prevent any attack attempts once their identity is known to the network. Section 3.6 covers the BA⋆ protocol in detail. Y. Gilad et al. [22] conclude with experimental results involving 500,000 clients, claiming that Algorand's throughput is around 125 times that of Bitcoin and that it faces minimal penalties while scaling to additional users.

Algorand uses a pure proof-of-stake (PPoS) protocol [10]. PPoS was described in section 1.3.4. Compared to PoW, Algorand does not require any expensive computations for block creation, is highly decentralized, produces blocks within seconds, and never forks. Compared to delegated PoS, users don't have to delegate their power to other members to produce blocks, although there could be a version of this that allows participants to delegate their power to other individuals. However, this requires significant research to handle the centralization that comes with pooling and delegating. Every user is capable of participating - with odds proportional to their stake. Compared to other PoS protocols, Algorand users

reserve the right to spend their stake, as the protocol does not require them to set aside some for security. Users can participate in the consensus and enjoy financial freedom, which is a practical advantage compared to other PoS protocols like Tendermint [11].

3.2 Challenges and Solutions

Algorand seeks to overcome the shortcomings of other cryptocurrencies. The research highlights three specific challenges and how Algorand plans to tackle them.

1. The cryptocurrency must be resilient to Sybil attacks, where an attacker creates multiple identities to manipulate the Byzantine agreement protocol. To mitigate this, Algorand relies on clients' stakes as a measure of their contribution. A consensus can be reached as long as the majority of the total stake is honest, which is a constant somewhat greater than 2/3. Algorand can prevent double spending and forks as long as more than two-thirds of the total coins or money is owned by honest users.
2. The BA⋆ algorithm should scale to millions of users. To achieve this, Algorand introduces a concept of committees that participate in the consensus. These committees consist of a fixed number of members chosen randomly from the entire network based on their stake. These committees are responsible for executing all stages of the BA⋆ protocol. Randomness governed by individual stake ensures a high honest fraction. Other users can simply observe the communication and find out proposed blocks, votes, membership proofs, etc.
3. The network must prevent disruptive attacks and function even when attackers partition the network. When a committee member participates and shares their votes, their identity is revealed, and attackers could target them. To address this, Algorand replaces committee members in every stage of the protocol. As soon as

the members communicate, they are no longer required for the protocol. As mentioned before, BA⋆ does not require any clients to store any private state. For the next stage, the lottery is run again, which allows all users to participate and check if they are selected.

3.3 Overview

Algorand works similarly to other blockchains in terms of transactions, growing in multiple asynchronous rounds, and every block containing the hash of the previous block. Communication in Algorand occurs via a gossip protocol where new transactions are gossiped, and other users collect all transactions they receive through this gossip in case they are chosen as block proposers. Users choose a subset of other users to gossip to, and every message is signed by their private key, and its signature can be verified by all receivers. The same message is never forwarded twice.

To propose blocks, Algorand uses cryptographic sortition, or a lottery, which every user executes privately to check if they are selected. Cryptographic sortition is described in section 3.4. It works like a weighted average of users' stake, and there can be multiple winning tokens for a single user. There can be multiple block proposers in a round, but the priority of all those block values is compared to decide on a single value.

Receiving block proposals does not guarantee safety since, as described above, there can be multiple proposals. Algorand uses BA⋆ to reach consensus. Committee members cast their votes in their respective steps and rounds, and BA⋆ would execute until a step is found where a predefined majority of users have reached consensus.

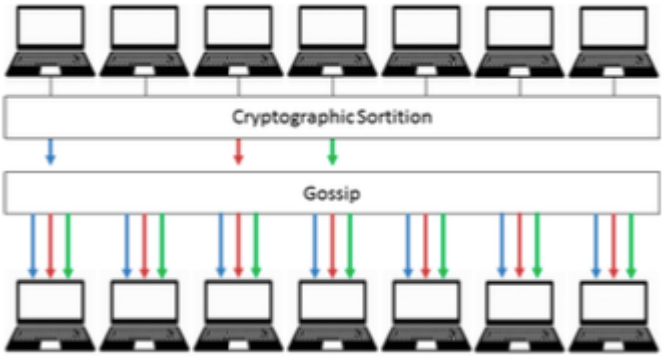

Figure 7: Basic message flow in BA⋆. Copied from [22]

Figure 7 describes a simple flow of messages between clients in a single step of BA⋆. The colored arrow indicates the message from a specific client.

Algorand works in **steps** to reach consensus before publishing a block. The BA⋆ results into two types of consensuses:

1. **FINAL**: A user reaching final consensus would mean that any other user who reaches final or tentative consensus in the same round would do so on the same block hash. This ensures safety since future blocks will be linked to this final block. A transaction would be successfully confirmed if it belongs to a block that has reached final consensus.
2. **TENTATIVE**: A tentative consensus means users have reached tentative consensus on different block values, and since no block hash has clear majority, a final consensus cannot be reached. A transaction belonging to a tentative block will only be confirmed once its successor blocks reach final consensus. A tentative consensus can be achieved in 2 cases:
    1. If the network is strongly synchronous, and an adversary has manipulated BA⋆ into reaching tentative consensus on a block. This does not reflect that BA⋆ ended up with two block values but only

its inability to reach majority consensus. Once following blocks are finalized, earlier blocks will be confirmed.

2. If the network is weakly synchronous, and an adversary has succeeded in partitioning the network, it can now end up with two different blocks. This results in a fork which Algorand resolves by periodically executing BA⋆ under fork resolution parameters and then informing which fork the users should switch to.

In a strongly synchronous network, Algorand [22] claims to reach consensus in a minimum of four steps and a maximum of 13 steps.

3.4 Sortition

The cryptographic sortition is the lottery mentioned in the previous section. It is an algorithm that outputs a subset of users randomly based on their weights. In this research, stakes and weights are used interchangeably. For a given user with weight w_i and the total weight of all users in the system $W = \sum_i w_i$, the probability of selection of a user i is proportional to w_i/W. The pseudorandomness is derived from the *seed*. Everyone in the network is aware of this seed value. The other prerequisite for executing sortition is a pair of public/private keys. Sortition uses verifiable random functions (VRFs), as described in section 2.6. To recap, a proof and a hash value are generated when a VRF is provided with an input. The hash remains indistinguishable from a random value if the secret key *sk* is unknown. The proof π can help anyone determine if the hash is derived from an input x using the public key *pk*, without ever needing the secret key.

procedure Sortition($sk, seed, \tau, role, w, W$):

$\langle hash, \pi \rangle \leftarrow \text{VRF}_{sk}(seed||role)$

$p \leftarrow \frac{\tau}{W}$

$j \leftarrow 0$

while $\frac{hash}{2^{hashlen}} \notin \left[\sum_{k=0}^{j} B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p)\right)$ **do**

  j++

return $\langle hash, \pi, j \rangle$

Figure 8: The cryptographic sortition algorithm. Copied from [22]

The selection process: Figure 8 highlights the cryptographic sortition algorithm used by Algorand. *sk* is the secret key of the user executing the algorithm. τ is the threshold that indicates the expected number of users for the given role. The role parameter takes in the different roles used by the user, such as proposing a block or being a member of a committee of BA⋆. To prevent against Sybil attacks, Algorand outputs a parameter *j* that indicates the number of times user *u* was selected based on their weight. *j* denotes the total number of sub-users of user *u*. The hash determines the number of selected sub-users using a binomial distribution. The probability that exactly *k* sub-users were chosen out of w_i, or simply *w,* follows the equation:

$$\mathrm{B(k; w, p)} = \binom{w}{k} p^k (1-p)^{w-k} \text{ where } \sum_{k=0}^{w} \mathrm{B(k; w, p)} = 1$$

To determine the final number of winning sub-users for any *w*, the hash value is normalized to consecutive intervals in the interval [0, 1), where any consecutive interval range is denoted by:

$$\left[\sum_{k=0}^{j} \mathrm{B(k; w, p)}, \sum_{k=0}^{j+1} \mathrm{B(k; w, p)}\right) \text{ where } j \in \{0, 1, \ldots, w\}$$

To find the winning number of sub-users, the interval in which $hash/2^{hashlen}$ belongs is checked. To verify the number of winning sub-users, the proof is used, and the binomial interval retrieval process is executed again.

Figure 9: Intervals in the range [0, 1). Copied from [23]

In [23], the author explains this pictographically as shown in Figure 9, where the total weight is assumed to be 2. The exercise is about randomly choosing a point on the line and selecting the interval it belongs to. The probability of finding the winning interval is given by B(k;w,p). It can be observed that the probability decreases with an increase in the number of winning sub-users k, and so users are more likely to realize k as 0 or a very small value. The verification algorithm which is run by all other users is described in Figure 10.

procedure VerifySort($pk, hash, \pi, seed, \tau, role, w, W$):
if $\neg VerifyVRF_{pk}(hash, \pi, seed||role)$ **then return** 0;
$p \leftarrow \frac{\tau}{W}$
$j \leftarrow 0$
while $\frac{hash}{2^{hashlen}} \notin \left[\sum_{k=0}^{j} B(k;w,p), \sum_{k=0}^{j+1} B(k;w,p)\right)$ **do**
  j++
return j

Figure 10: Verify sortition for a user. Copied from [22]

One of the key components of cryptographic sortition is the seed. Every block round requires a new seed that is known to everyone and should be chosen to prevent attackers from manipulating it. The seed for round *n* is determined in round *n-1*. Every block proposer chosen for round *n-1* also calculates the seed for round *n* using the same VRF function, where the input is the concatenation of the seed of the current round and the next round number. The *seed* is included so that when consensus is reached, every user has the knowledge of the seed to be used. If an invalid seed is found, an empty block is published

for that round. The seed is also refreshed every *R* rounds to prevent any manipulation by the attacker. At round *n*, the sortition is passed $seed_{n-1-(n \% R)}$.

3.5 Block Proposal

Algorand uses a threshold value $\tau_{proposer}$ in the sortition algorithm to ensure that a block is produced every round. Setting $\tau_{proposer}$ to 26 ensures that at least 1 and at most 70 block proposers are selected for a given round with a probability as high as $1 - 10^{-11}$. However, users may obtain multiple winning sub-users after running sortition, which can result in a lot of communication cost if all are gossiped. Therefore, the priority of each sub-user *j* needs to be determined by hashing the VRF output with all values of *j* and finding the highest priority hash value, which is then gossiped to every user. The other users only need to consider the highest priority blocks that they observe and discard all other irrelevant values. The priority block values and sortition proofs are one set of messages, while the block is sent separately as another message.

Each user should wait for an appropriate amount of time to gather block proposals. A shorter wait time would mean no block proposals are received, and the user starts BA⋆ with an empty block, which may lead to an empty block being reached by consensus if all users do the same. On the other hand, a longer wait time would introduce unnecessary latency. Algorand identifies three types of timeouts with respect to users waiting:

a. λ_{block}: the time any user would wait for receiving the block after receiving the highest block hash and proof, after which it will choose an empty block instead.

b. $\lambda_{stepvar}$: average time it takes the user to finish the last step of BA⋆. This is used usually when a user has already reached consensus and is waiting for other users who are still in the previous round.

c. $\lambda_{priority}$: time required to broadcast the block hash and its proof.

In case some of the block proposers are malicious, the network can be tricked into running BA⋆ with different block values, resulting in Algorand reaching consensus on empty blocks. This scenario is plausible but unlikely since an honest proposer with the highest priority block will broadcast the same block to all users, leading to a consensus being reached.

3.6 BA⋆

The most talked about protocol for Algorand is their BA⋆ protocol. It works in two phases:

1. **First phase**: the consensus on a block needs to be reduced to one of two options.
2. **Second phase**: BA⋆ agrees on either agreement on a proposed block or an empty block.

The stages of the protocol are divided into steps. The first phase involves strictly two steps, while the second phase takes 2 steps in the best case or 11 in the worst case. In every step, committee members cast their vote on some block, and everyone receives and counts these votes. The value that receives a threshold number of votes will be used by the users in the next step, provided they are selected as committee members. If they don't receive enough votes, they have to *TIMEOUT*, and the current step number determines which value will be used next.

If BA⋆ receives enough votes, it declares final consensus in its final step to confirm that there wouldn't be any other block value that also received enough votes. If this is not achieved, a tentative consensus is declared. An overview of the BA⋆ algorithm is described in Figure 11.

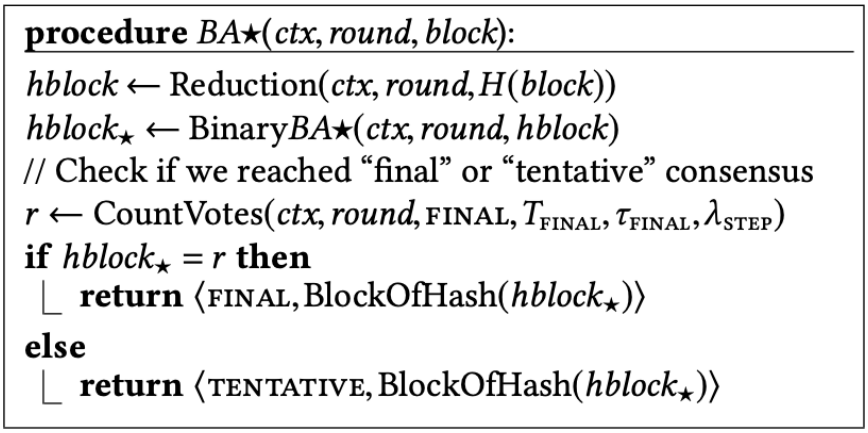

Figure 11: BA⋆ main algorithm. Copied from [22]

The procedure inputs a context *ctx* which is the current state of the blockchain, a *round* number denoting the current round for which blocks are being proposed, and a highest-priority block received. The context *ctx* stores specifically these values:

a. seed for the round

b. last confirmed block

c. all user weights or stakes

The consensus decision is also determined by BA⋆. All the components of this algorithm are covered in the following sections of this research report.

3.7 Voting

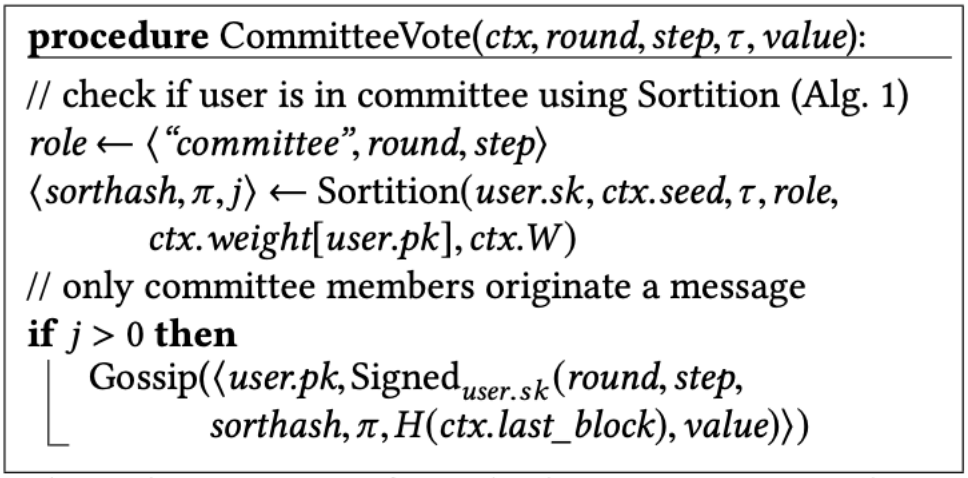

Figure 12: Algorithm for voting. Copied from [22]

The voting used by Algorand's committee members is described in Figure 12. *First*, it checks if the user can vote in the given round and step. This is achieved by calling sortition, described in section 3.4. *Second*, if the user is selected, they broadcast the block hash value passed to the function. The message also contains the previous block hash, derived from the context *ctx*, and is signed with their secret key. τ is the threshold for sortition.

3.8 Count Votes and Process Messages

```
procedure CountVotes(ctx, round, step, T, τ, λ):
start ← Time()
counts ← {}     // hash table, new keys mapped to 0
voters ← {}
msgs ← incomingMsgs[round, step].iterator()
while TRUE do
    m ← msgs.next()
    if m = ⊥ then
        if Time() > start + λ then return TIMEOUT;
    else
        ⟨votes, value, sorthash⟩ ← ProcessMsg(ctx, τ, m)
        if pk ∈ voters or votes = 0 then continue;
        voters ∪ = {pk}
        counts[value] + = votes
        // if we got enough votes, then output this value
        if counts[value] > T · τ then
            return value
```

Figure 13: Algorithm for counting votes. Copied from [22]

The votes cast by the committee members are counted by an algorithm depicted in Figure 13. The message buffer *incomingMsgs* collects votes for any round and step. Every vote is processed, described in Figure 14, which validates them. *ProcessMsg()* returns 0

votes if the vote is not valid. It runs the sortition verification, described in section 3.4, to determine the number of sub-users *j* linked to the vote. *T* is the committee size that acts as a threshold for BA⋆ and τ is the expected number of committee members in sortition. Once any block value receives more than $T * \tau$ votes, the *CountVotes()* algorithm immediately returns it. If not enough votes are received within λ time then the function returns a *TIMEOUT*. Note that *ProcessMsg()* ignores duplicate votes.

```
procedure ProcessMsg(ctx, τ, m):
⟨pk, signed_m⟩ ← m
if VerifySignature(pk, signed_m) ≠ OK then
  return ⟨0, ⊥, ⊥⟩
⟨round, step, sorthash, π, hprev, value⟩ ← signed_m
// discard messages that do not extend this chain
if hprev ≠ H(ctx.last_block) then return ⟨0, ⊥, ⊥⟩;
votes ← VerifySort(pk, sorthash, π, ctx.seed, τ,
        ⟨"committee", round, step⟩, ctx.weight[pk], ctx.W)
return ⟨votes, value, sorthash⟩
```

Figure 14: Validate votes. Copied from [22]

3.9 Reduction

In the first phase of BA⋆, the agreement is reduced to one of two options: either a block hash or an empty hash. This is done by the Reduction algorithm presented in Figure 15. It ensures liveness and outputs either a block hash value or an empty block hash value. The first step is where committee members vote for the block hash provided by BA⋆. The second step requires committee members to vote for the block hash that received the required $T * \tau$ threshold votes or vote for an empty block hash. When the network is strongly synchronous and the highest priority block proposer is not malicious, almost all users will input the same *hblock* value to Reduction and Reduction will return the same *hblock* to the

next stage as well. In the opposite case, if the highest priority block user was indeed malicious, users will start Reduction with different *hblock* values and no particular *hblock* will win threshold amount of votes. Thus, Reduction will end with *empty_hash.*

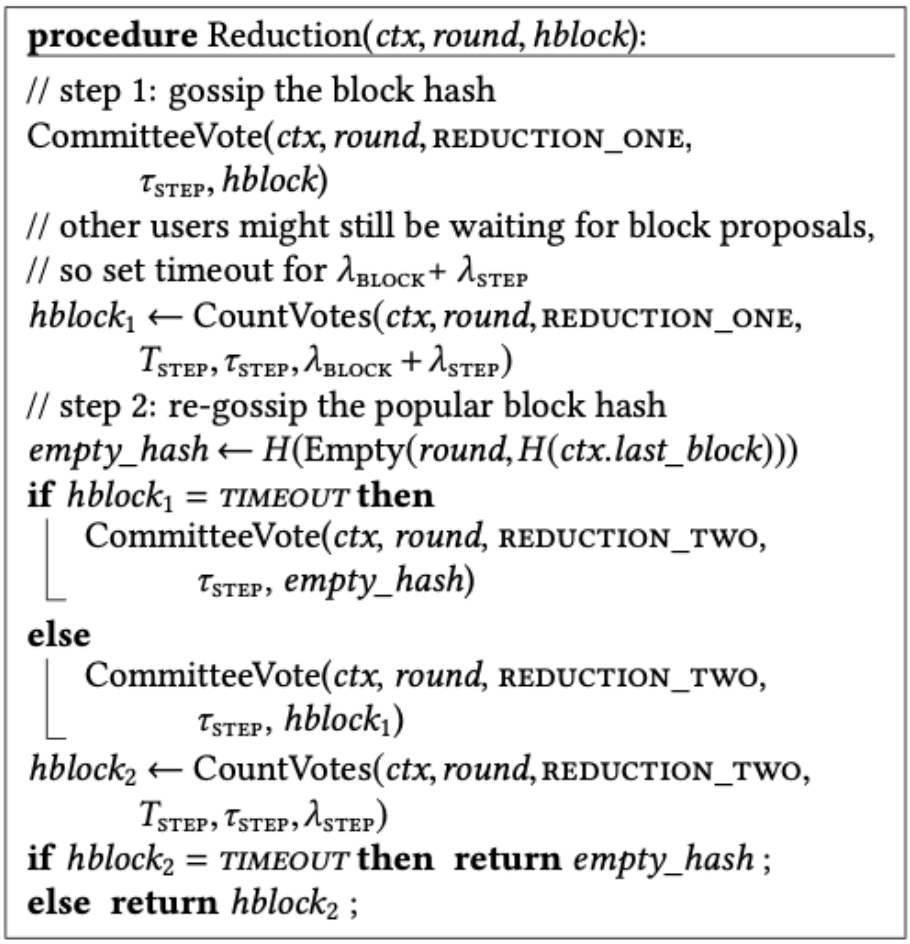

Figure 15: Reduction algorithm. Copied from [22]

3.10 Binary BA⋆

The Binary BA⋆ algorithm used in the second phase of BA⋆ is presented in Figure 16. It describes reaching agreement either on a block hash or an empty block hash. Algorand's research [22] explains how the protocol achieves safety in strong and weak synchrony.

3.10.1 Strong synchrony

In this scenario, it is assumed that the network is strongly connected, and a majority of users are honest. If a user who receives more than $T * \tau$ votes for any value, they vote for the same value in the following step. If no value has a majority, then the next vote should be selected in a way that should guarantee consensus in this strongly synchronous network state. If a user A observes enough votes for a value, they do not immediately return the value. It may so happen that an adversary has sent A votes that pushes its count past the threshold, but the other users do not see these votes and therefore, *TIMEOUT*. User A would return consensus on the block while others are still stuck. In this critical situation, the users returning from a timeout should continue to work on the value that could have potentially been returned by A. If more users returned consensus in the first step like A, there wouldn't be enough users to *CountVotes()* for the next step. Therefore, if a user has reached consensus, they vote for the same value in the following 3 *steps*, namely *step+1*, *step+2*, and *step+3*. This is so that enough votes are pushed for the value so that users who are lagging do not miss out on them. These users can count these votes in their following three steps and then return consensus in the 4th step which would be the second iteration of the while loop.

The common and ideal scenario is when the network is strongly synchronous. Most users will observe the same starting *block_hash* and reach consensus in the first step for the same *block_hash* value. This is where everyone votes again for the *'final'* step before returning. The final step has a larger committee size and to decide the consensus on a block value, its votes for the *'final'* step are counted.

```
procedure BinaryBA⋆(ctx, round, block_hash):
step ← 1
r ← block_hash
empty_hash ← H(Empty(round, H(ctx.last_block)))
while step < MaxSteps do
    CommitteeVote(ctx, round, step, τ_STEP, r)
    r ← CountVotes(ctx, round, step, T_STEP, τ_STEP, λ_STEP)
    if r = TIMEOUT then
        r ← block_hash
    else if r ≠ empty_hash then
        for step < s' ≤ step+3 do
            CommitteeVote(ctx, round, s', τ_STEP, r)
        if step = 1 then
            CommitteeVote(ctx, round, FINAL, τ_FINAL, r)
        return r
    step++

    CommitteeVote(ctx, round, step, τ_STEP, r)
    r ← CountVotes(ctx, round, step, T_STEP, τ_STEP, λ_STEP)
    if r = TIMEOUT then
        r ← empty_hash
    else if r = empty_hash then
        for step < s' ≤ step+3 do
            CommitteeVote(ctx, round, s', τ_STEP, r)
        return r
    step++

    CommitteeVote(ctx, round, step, τ_STEP, r)
    r ← CountVotes(ctx, round, step, T_STEP, τ_STEP, λ_STEP)
    if r = TIMEOUT then
        if CommonCoin(ctx, round, step, τ_STEP) = 0 then
            r ← block_hash
        else
            r ← empty_hash
    step++
// No consensus after MaxSteps; assume network
// problem
HangForever()
```

Figure 16: BinaryBA⋆ algorithm. Copied from [22]

3.10.2 Weak synchrony

This scenario arises when the network has a partition, and there is a risk that BinaryBA⋆ may end up with consensus on two block values. Let's consider a case where only user A receives votes for a block, but the messages are dropped for all others. In this case, A has enough votes to reach consensus, but other users move to the next step where they vote for the block hash again. These messages are dropped again, due to which the users now vote for empty hash, and this is when the messages are delivered. Seeing enough votes, a consensus will be reached for empty hash, but now BinaryBA⋆ has resolved to two separate block values, resulting in a fork. This is where the concepts of final and tentative consensus come into play. Final consensus is only decided when enough users have voted for the same value and no other value for that round. Tentative simply means that safety couldn't be ensured. The final consensus depends on the votes designated for the final step. This is a special vote indicating that a strong agreement is reached, and the block can be published. In the described scenario above, when A returns consensus, it will never achieve final consensus because it will never have enough users who voted final on that particular block value.

The issue where users are divided into two groups remains unresolved. Assume a group of users is voting for empty hash and combined with adversary's votes can cross the threshold. And there is another group who are voting for block hash. Locally, neither of the groups has enough votes. Since the procedure is known to the adversary, they can manipulate and control what everyone votes for in the next step. To ensure users vote for empty hash, the adversary sends its votes for empty hash, which gives them enough majority. To make users vote for block hash, the adversary simply lets their *CountVotes()* return a timeout, after which the users reset the block value to block hash for the next step, as described in the

algorithm depicted in Figure 16. This can be performed indefinitely and halt the progress forever. The solution to this is a simple algorithm called *CommonCoin* described in Figure 17.

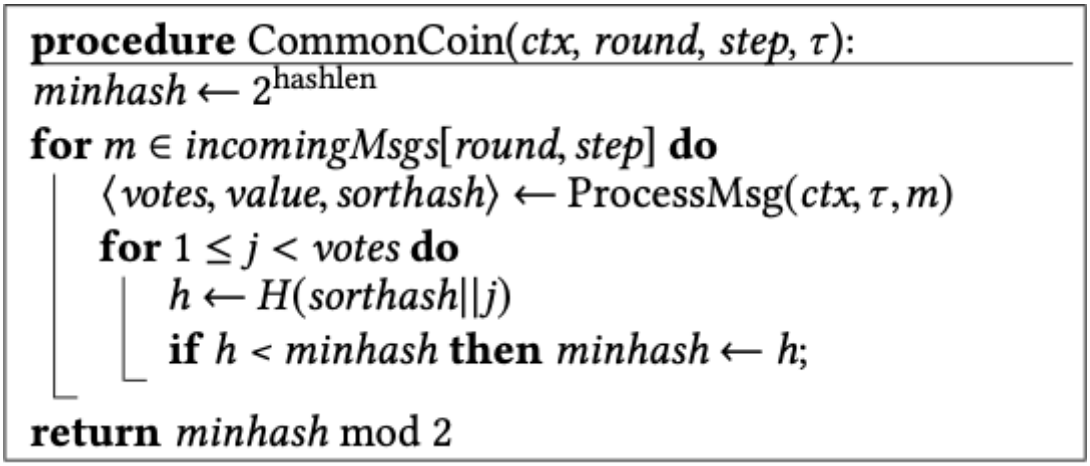

Figure 17: Calculating a common coin. Copied from [22]

CommonCoin acts like a coin flip between block hash or empty hash. If enough users realize the same coin bit without the adversary knowing it, consensus can be reached in 50% of the cases since it's the probability that the adversary guessed it wrong. The implementation takes use of the VRF sortition hashes and calculates the lowest sortition hash users observe in the current step. If a committee member had multiple sub-users selected, the sortition hash is concatenated with all those sub-users' values, and the minimum is found of those. The function returns with the least significant bit (LSB) of the minimum hash, which can either be 0 or 1. Since the VRF output was random, the hashes are random, and so will be their LSB. If the user with the lowest hash was honest, then all the users that received their message will also realize the same coin. If the adversary controls the lowest hash, they can send it only to certain users, again splitting the users who observe different coins. But given the fact that there are more than 2/3rd honest users who can control the lowest hash and with a 50% probability in the *CommonCoin* process, the consensus can be reached with more than 1/3 probability in a single iteration of the while

loop. Since the loop contains a sequence of 3 steps, and as per the calculation, consensus can be reached with probability (1/3), BinaryBA⋆ requires 2 + 3 * 3 = 11 steps to reach consensus in the worst case, where the 2 extra steps account for the Reduction steps that led to BinaryBA⋆.

3.11 Fork resolution

If the network is not synchronous, forks can occur, as explained in previous sections. The liveness property is still maintained because BA⋆ will result in tentative consensus, but it impacts safety because users on different forks will not consider each other's votes since their values for *ctx.last_block* will be different (see figure 14). One of the forks will grow longer, and the other will not have any users to put the votes past the threshold, causing BA⋆ to never reach consensus there. The solution is fork resolution. Algorand periodically executes fork resolution where a fork is proposed that everyone should agree to, and BA⋆ is used again to reach consensus. Users monitor even those votes that do not belong to their fork, for future use. The recovery protocol is similar to block proposal. They all run sortition to be selected as the *"fork proposer"* where they propose an empty block that belongs to the longest fork they have observed. Every user verifies the block and its previous hashes, and if it is the longest chain the user has seen. The fork should include all final blocks. In the end, they all execute BA⋆ to reach consensus on the block with the round number in it.

CHAPTER 4

Design

This section of the project report covers the design decisions and the general flow of the Algorand protocol in the implementation[1]. It also discusses the reasoning for reworking some of the SpartanGold components to enhance usability and understanding. Since this project is developed using the SpartanGold framework, it is also developed in JavaScript.

Section 2.5 briefly explains the classes used in SpartanGold. However, in our implementation, these classes were removed:

1. *Transaction*: For simplicity, this class was not included in our initial implementation of the Algorand protocol, which focuses primarily on the Byzantine consensus protocol. It may be considered for future work.
2. *Miner*: This class was an extension of the *Client* class and dealt specifically with initiating a new search and mining for proof. However, since this project uses a PoS approach on SpartanGold, mining is no longer necessary.

4.1 Classes

The key files of our implementation include:

1. ***Driver***: This serves as the entry point for the application and is similar to the setup in SpartanGold. The driver sets up the blockchain, clients, network, and the genesis block. Each client is initialized with a starting balance, which corresponds to its weight or stake in the cryptographic sortition process for all stages. The driver also defines a special genesis seed to be used for the genesis block before the blockchain

[1] For more information about our Pure Proof of Stake implementation, please see my GitHub repository: https://github.com/nimesh13/pure-proof-of-stake

begins adding blocks. The actual genesis seed used in our implementation is described as:

```
let genesisSeed = "# THIS IS GENESIS BLOCK SEED FOR CS298 #"
```

The driver initializes all clients using *client.initialize()* and they all begin execution for a designated period of time. After this period, their balances at their stage are printed, and the process terminates.

2. ***StakeBlock*:** This the block class of SpartanGold-Algorand. It is an extension of the original Block class with a few added functionalities. The class declaration is described as:

```
class StakeBlock extends Block
```

StakeBlock sets the seed for block creation and initializes the parameters that deal with sortition proof, hash, number of sub-users or tokens, and the status of the block - whether final or tentative. The following methods are available within the StakeBlock class:

a. *getTotalCoins()* - calculates the total number of coins in the system concerning a block. The result is represented as W, which is used in the cryptographic sortition process.

b. *getContext()* - generates the context ctx that stores the current state of the blockchain. It consists mainly of these 4 values:

1. *seed* - of the current round
2. *lastBlock* - block hash of the previous block
3. *w* - balances of all the clients.
4. *W* - the total number of coins in the system, calculated using *getTotalCoins()*

c. *serialize()* - this represents the snapshot of a block. The StakeBlock is translated into a json object which is then serialized and returned. *serialize()* is mainly used to calculate the hash of a block. In SpartanGold, all the attributes are calculated in generating the hash but in our implementation, we use only a few attributes namely:

1. *prevBlockHash* - hash of the previous block
2. *rewardAddr* - the block proposer of the block
3. *seed* - seed used to propose the block
4. *chainLength* - the current length of the chain, also the round number in Algorand semantics.
5. *genesisBlockHash* - the hash of the genesis block
6. *balances* - an array of current balances of all the clients

The reason for using these specific attributes is that when clients generate an empty block, they do so locally. No empty block is transmitted over the network that everyone uses in their chain. Thus, when they propose blocks in the next round, they will all have the same *prevBlockHash* value, and their votes will not be discarded. On the contrary, if they all generate an empty block that results in a different block hash for everyone, their *prevBlockHash* will be different in the next round.

3. ***StakeBlockchain*:** This is the blockchain class of SpartanGold-Algorand. It is an extension of the original Blockchain class described as:

```
class StakeBlockchain extends Blockchain
```

Like SpartanGold, this stores the various configurations of the blockchain, such as the sortition threshold for intermediate steps and the *FINAL* step, the committee

sizes for the sortition, and the timeout value for the different stages of the consensus protocol. It also stores certain constants required during consensus.

4. ***StakeClient*:** The main class of SpartanGold-Algorand is *StakeClient*. It is derived from the original Client class as:

```
class StakeClient extends Client
```

Algorand's Byzantine consensus protocol is implemented in this class. *StakeClient* is responsible for keeping track of client keys, proposals, incoming messages, listeners for all emitted events, and participating in the protocol. Clients communicate with each other using broadcasting messages on certain channels that other clients listen to. The types of events used in our implementation include:

1. **PROPOSE_BLOCK**: after clients are initialized, an event is triggered to allow them to start proposing blocks.
2. **ANNOUNCE_PROOF**: this event is emitted when the clients are selected to propose blocks after executing the cryptographic sortition.
3. **GOSSIP_VOTE**: when clients need to broadcast their vote at any step or stage of the consensus, this event is used.
4. **ANNOUNCE_BLOCK**: this event is used by the block proposer to finally send their proposed blocks after a consensus is reached.

findWinningProposal() - This function is used by the clients after waiting for the allocated time interval for receiving proposals. They calculate the minimum hash from all received proposals and store it in *winningBlockhash*. If no proposal is received, this function creates and empty block and moves on to the next round.

4.2 Algorand's Byzantine consensus

Algorand's Byzantine consensus consists of two phases - reduction and binary agreement. Although the source code differs slightly in how these phases are constructed, we'll follow closely how they are presented in the Algorand whitepaper.

4.2.1 Reduction Phase

The methods that correspond to the *REDUCTION* phase are:

a. *reductionOne(...)* - the first step of reduction, *REDUCTION_ONE*, where the clients vote for the *block_hash* passed to the function after finding the minimum hash.

b. *countReduceOne(...)* - this method counts the votes accumulated for the REDUCTION_ONE step.

c. *reductionTwo(...)* - the second step of reduction, *REDUCTION_TWO*, where the clients vote either for *empty_hash* or the block hash obtained from *countReduceOne()*.

d. *countReduceTwo(...)* - this method counts the votes accumulated for the *REDUCTION_TWO* step and decides to forward either *empty_hash* or the block hash obtained from the votes.

4.2.2 BinaryBA⋆ Phase

The methods that correspond to the BinaryBA⋆ phase are:

a. *binaryBAStarStageOne(...)* - the first step of the algorithm where the clients vote for the hash passed on from the reduction phase.

b. *binaryBAStarCountStageOne(...)* - this method counts the votes for the first step and returns if a non-empty hash is found otherwise the hash value is reset the original hash.

c. *binaryBAStarStageTwo(...)* - clients vote for the block hash again.

d. *binaryBAStarCountStageTwo(...)* - this method counts the votes for the second step and returns the hash if enough votes are found. Otherwise, the hash is set to *empty_hash* value.

e. *binaryBAStarStageThree(...)* - clients vote for the *empty_hash* in the third step.

f. *binaryBAStarCountStageThree(...)* - this method counts the votes for the third step and calls the *commonCoin()* function to decide on a *block_hash* or an *empty_hash.*

g. *commonCoin(...)* - this method implements the coin flip that returns the least significant bit (LSB) of the minimum hash for that given round.

h. *BAStar(...)* - this method describes the BA⋆ algorithm described in section 3.6. When a block hash is returned from any step of the BinaryBA⋆, *BAStar()* calculates the votes for the *FINAL* step and compares with the value with which it is called. If they are equal, final consensus is reached else it announces only a tentative consensus.

Figure 18 provides an easier understanding of the flow of control in our implementation. These are the methods that implement the other algorithms mentioned in the Algorand whitepaper:

a. *committeeVote(...)* - implements the voting algorithm

b. *processMsg(...)* - implements the checks and validations for received votes

c. *countVotes(...)* - counts the votes for a given round and step

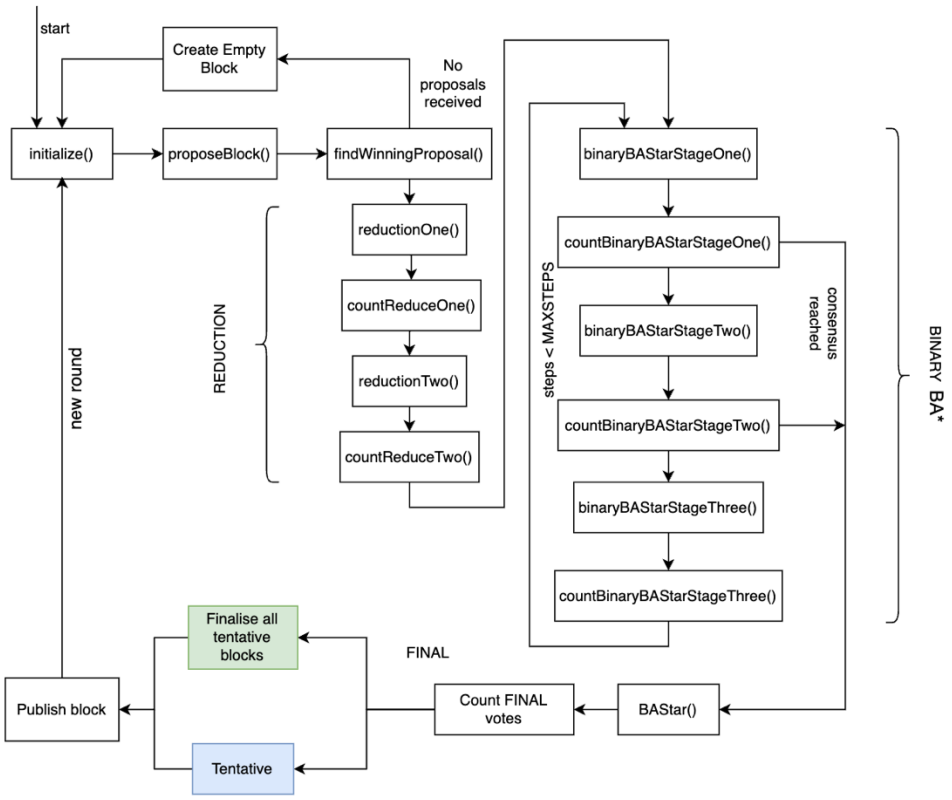

Figure 18: Overview of the entire flow.

4.3 Network and Communication

We reused SpartanGold's network communication driver *FakeNet* which implements broadcasting and sending messages to other users. The gossip protocol in SpartanGold-Algorand involves sending messages to every other client since this simulation requires only a few users. A more sophisticated gossip algorithm could be taken up as future work.

CHAPTER 5

Implementation

This section covers the actual simulation details in our implementation. Using screenshots, we demonstrate the different stages of the Algorand consensus protocol and document the configurations used for the simulation. In later sections of this chapter, we also cover the different Byzantine situations and discuss how the implementation progresses under them. We begin the demonstration by executing the protocol under normal conditions. The driver code executes for 3 minutes, after which it logs the final balances before terminating the process. The configurations used in the simulation are described in Table 1.

Table 1: Implementation parameters

Parameter	**Meaning**	**Value**
clientArray	The clients used in the simulation	Alice, Bob, and Charlie
balances	The starting balances of all clients	15, 10, 20 respectively
chanceMessageFails	Message drop percentage	0
messageDelayMax	Message delay in the network	0
coinbaseReward	Rewards for publishing a block	25
COMMITTEE_SIZE	Committee size for BA⋆	2
SORTITION_THRESHOLD	Number of block proposers and committee members	2
SORTITION_THRESHOLD_STEP	Threshold for SortitionThreshold step	68.5%
SORTITION_THRESHOLD_FINAL	Threshold for SortitionThreshold final	68.5%
timeout	Timeout range for all stages	3-6 seconds

5.1 *FINAL* Consensus

The common scenario when the network is strongly synchronous will publish blocks in the very first step and reach FINAL consensus, since there are many participants, and the majority of the network is honest. Let's look at how the application achieves this, covering all the stages of the consensus protocol:

5.1.1 Block Proposal

Figure 19 shows the block proposal stage where only Alice was able to publish blocks. Bob and Charlie can only wait for block proposal since they did not qualify in the block proposal selection stage.

```
nimesh@Nimeshs-MacBook-Air algorand-proof-of-stake % node driver.js
Starting simulation.  This may take a moment...
Initial balances:
[ Alice ] Balance: 15 gold.
[ Bob ] Balance: 10 gold.
[ Charlie ] Balance: 20 gold.

[ Bob ] [ BLOCK_PROPOSAL ] Listening for other proposals.
[ Charlie ] [ BLOCK_PROPOSAL ] Listening for other proposals.
```

Figure 19: The block proposal stage.

5.1.2 Receive Proof

Figure 20 describes *Alice's* proposal being broadcast to everyone in the network. A point to note here is that the broadcast in our implementation sends an event or message to the client who initiated it also. Since only *Alice* proposed a block, it automatically becomes the minimum *block_hash* the clients have observed.

```
[ Alice ] [ RECEIVE_PROOF ] Received a proposal.
[ Bob ] [ RECEIVE_PROOF ] Received a proposal.
[ Charlie ] [ RECEIVE_PROOF ] Received a proposal.
```

Figure 20: All clients receive block proposals.

5.1.3 Reduction-One Stage

Figure 21 describes the first Reduction stage where clients vote and then count the votes received from all other clients. The hash value observed in the figure is the *block_hash* that crossed the threshold amount of votes. All the clients are able to reduce this stage hash to the same value.

```
[ Alice ] [ REDUCTION_ONE ] Voting..
[ Bob ] [ REDUCTION_ONE ] Voting..
[ Charlie ] [ REDUCTION_ONE ] Voting..

[ Alice ] [ REDUCTION_ONE ] Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Bob ] [ REDUCTION_ONE ] Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Charlie ] [ REDUCTION_ONE ] Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
```

Figure 21: The Reduction One stage.

5.1.4 Reduction-Two Stage

Figure 22 describes the second Reduction phase where the *block_hash* is voted and votes counted for this stage again. The hash value observed in the figure amounts to the *block_hash* that crossed the majority threshold amount of votes. On close observation, the value in both the Reduction phases is the same.

```
[ Alice ] [ REDUCTION_TWO ] Voting..
[ Bob ] [ REDUCTION_TWO ] Voting..
[ Charlie ] [ REDUCTION_TWO ] Voting..

[ Alice ]  REDUCTION_TWO ] Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Bob ]  REDUCTION_TWO ] Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Charlie ]  REDUCTION_TWO ] Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
```

Figure 22: The Reduction Two stage.

5.1.5 BinaryBA⋆ First Stage

After the Reduction phases, the *block_hash* is passed on to the BinaryBA⋆ method. Figure 23 describes the first stage of BinaryBA⋆ where everyone votes for the block hash that was returned from Reduction. In the figure, the block hash earns the majority of votes, and every client reaches finality on it. They vote three times before voting for the same value in the FINAL step.

```
[ Alice ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Voting...
[ Bob ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Voting...
[ Charlie ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Voting...

[ Alice ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Alice ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Quorum reached. Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Bob ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Bob ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Quorum reached. Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Charlie ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Charlie ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Quorum reached. Hash: 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
```

Figure 23: The BinaryBA⋆ first stage.

5.1.6 BA⋆

After the value is returned from BinaryBA⋆, BA⋆ counts the votes again for the *FINAL* step before deciding the consensus - **final** or **tentative**. As observed in figure 24, a final consensus is observed and block proposer, Alice, announces the block to the whole network. When a final consensus is reached for any block, all prior tentative blocks are also finalized.

```
[ Alice ] [ BA* ] FINAL Consensus reached! 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Alice ] Announcing block!
[ Bob ] [ BA* ] FINAL Consensus reached! 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41
[ Charlie ] [ BA* ] FINAL Consensus reached! 99171f38318adf54ca2633df34fb0c0034a35e4cfda9a4d60388e60e1e75ad41

[ Alice ] Recevied a FINAL block. Finalizing all TENTATIVE blocks!
[ Bob ] Recevied a FINAL block. Finalizing all TENTATIVE blocks!
[ Charlie ] Recevied a FINAL block. Finalizing all TENTATIVE blocks!
```

Figure 24: BA⋆ signals consensus.

5.1.7 Balance update

The final balances after the end of the process reflects the new balances with Alice's balance increased by the *coinbaseReward* which is 25 gold. Her initial balance was set to 15.

```
Final balances:
[ Alice ] Balance: 40 gold.
[ Bob ] Balance: 10 gold.
[ Charlie ] Balance: 20 gold.
```

Figure 25: Final balances after first round.

Safety is guaranteed as the block receives a clear majority even in the *FINAL* step, meaning it can never be reverted.

Liveness is also clearly guaranteed as the blockchain is able to publish blocks.

5.2 *TENTATIVE* Consensus

5.2.1 Consensus on a block

There is also a possible scenario where everyone is the network achieves quorum and returns the *block_hash* but during BA⋆ vote counting for the *FINAL* step, they don't receive a majority and simply time out. This leads to a tentative consensus because the clients *were* able to observe a lot of participation for the *block_hash* but not for the *FINAL* step. Let's look at how the application progresses in such a scenario.

```
[ Alice ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Hash: b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
[ Alice ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Quorum reached. Hash: b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
[ Bob ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Hash: b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
[ Bob ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Quorum reached. Hash: b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
[ Charlie ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Hash: b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
[ Charlie ] [ BINARYBA* ] [ STAGE-1 ] Step: 1 Quorum reached. Hash: b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
|

[ Alice ] [ BA* ] TENTATIVE Consensus reached! Hash: b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
[ Alice ] [ BA* ] TENTATIVE Consensus reached! FINAL votes: TIMEOUT
[ Alice ] Announcing block!
[ Bob ] [ BA* ] TENTATIVE Consensus reached! b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
[ Bob ] [ BA* ] TENTATIVE Consensus reached! FINAL votes: TIMEOUT
[ Charlie ] [ BA* ] TENTATIVE Consensus reached! b2ab77ef5c11d85faa92bc2200822be48aa77fa34bae98f66dfda8157c7fe8ff
[ Charlie ] [ BA* ] TENTATIVE Consensus reached! FINAL votes: TIMEOUT
```

Figure 26: Tentative consensus on a *block_hash*.

Figure 26 describes the scenario where the clients timeout during counting the votes for the FINAL step. A tentative consensus is declared for the *block_hash* and the block proposer, Alice, broadcasts the block. The block is added tentatively and can only be finalized when a future block generated from this block achieves final consensus.

Safety and **liveness** are both provided here since a consensus was reached on a block and the blockchain can progress even without the block being finalized. Safety here is, however, weaker than final consensus.

5.2.2 Consensus on an empty block

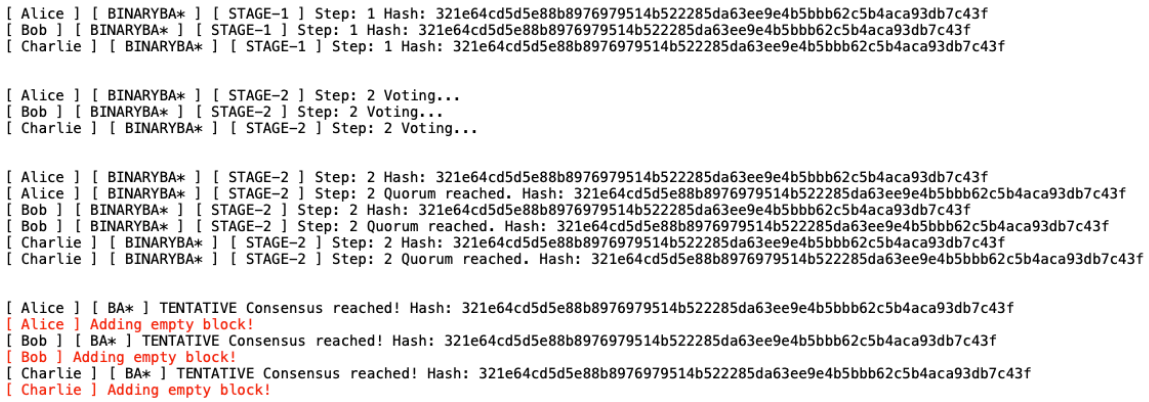

Figure 27: Tentative consensus on *empty_hash*.

A weakly synchronized network can also lead to clients agreeing on an empty block. Figure 26 demonstrates that scenario. It can be observed that the second stage of BinaryBA⋆ deals with agreement on *empty_hash* and receives quorum for that value. Since BA⋆ cannot finalize this consensus, only a tentative consensus is declared, and the clients add an empty block to their blockchain.

Safety and **liveness** both are provided here since consensus was obtained, even though it is an empty block, and cannot be reverted and the blockchain is able to progress through to the next round but they are weaker than final and tentative consensus on a block as the transactions are yet to be confirmed by the network.

CHAPTER 6

Evaluation

This section covers various undesirable scenarios that can befall a blockchain network and discusses how our implementation of the algorithm progresses. We also comment on the safety and liveness properties of those scenarios with respect to our implementation.

6.1 No Consensus after *MAXSTEPS*

As discussed in section 3.10.2, the algorithm runs for a combined 11 steps, with 2 in the reduction phase and 9 steps in the binary phase to reach consensus in the worst-case scenario. But to simulate a weak network where messages are dropped and no block value ever receives a majority of votes, we can increase the threshold required to count the votes for the *FINAL* step. This is equivalent to a weak network where clients cannot collect enough votes. For this, we set the *SORTITION_THRESHOLD_FINAL* to 74%.

```
[ Alice ] [ BINARYBA* ] [ STAGE-2 ] Step: 5 Voting...
[ Bob ] [ BINARYBA* ] [ STAGE-2 ] Step: 5 Voting...
[ Charlie ] [ BINARYBA* ] [ STAGE-2 ] Step: 5 Voting...

[ Alice ] [ BINARYBA* ] [ STAGE-2 ] Step: 5 Hash: f0353cc81ca55d9d15c095126b9e8af3c107a92ad715bf82601c463b2b270908
[ Bob ] [ BINARYBA* ] [ STAGE-2 ] Step: 5 Hash: f0353cc81ca55d9d15c095126b9e8af3c107a92ad715bf82601c463b2b270908
[ Charlie ] [ BINARYBA* ] [ STAGE-2 ] Step: 5 Hash: f0353cc81ca55d9d15c095126b9e8af3c107a92ad715bf82601c463b2b270908

[ Alice ] [ BINARYBA* ] [ STAGE-3 ] Step: 6 Voting...
[ Bob ] [ BINARYBA* ] [ STAGE-3 ] Step: 6 Voting...
[ Charlie ] [ BINARYBA* ] [ STAGE-3 ] Step: 6 Voting...

[ Alice ] [ BINARYBA* ] [ STAGE-3 ] Step: 6 Hash: TIMEOUT
[ Bob ] [ BINARYBA* ] [ STAGE-3 ] Step: 6 Hash: TIMEOUT
[ Charlie ] [ BINARYBA* ] [ STAGE-3 ] Step: 6 Hash: TIMEOUT
```

Figure 28: BinaryBA⋆ intermediate steps.

Figure 28 displays the intermediate steps of the BinaryBA⋆ algorithm. Step 5 in the figure corresponds to the second stage of BinaryBA⋆ where the clients only return consensus if *empty_hash* receives a majority, which does not happen. Counting votes for step 6 or the third stage results in a timeout after which the clients implement the *CommonCoin()* functionality.

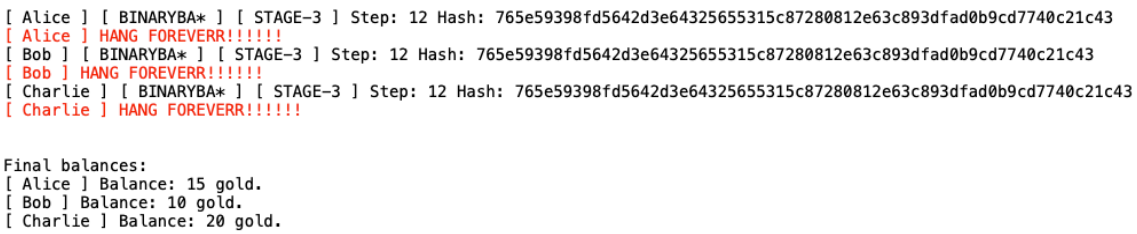

Figure 29: *MAXSTEPS* is reached.

The other intermediate steps are skipped to focus on the issue at hand. Figure 29 shows clients reaching *MAXSTEPS* number of iterations without ever reaching consensus. In such a case, Algorand recovers liveness by implementing fork resolution, but since fork resolution is beyond the scope of this project, we display a warning and exit the execution.

Security is ensured here since the network reaches a safe state.

Liveness is not ensured as the blockchain is unable to proceed and halts all execution.

6.2 Byzantine Client

In this scenario, we introduce a Byzantine client, Trudy (intruder), into the network.

```
let trudy = new StakeByzantineClient({ name: "Trudy", net: fakeNet });
```

We start Trudy off with an initial balance of 30 coins. Given that Alice owns 15 coins, Bob owns 10, and Charlie owns 20, Trudy owns exactly 2/3 of the total coins. There can be two further scenarios in the network involving Trudy:

1. Trudy does not propose blocks.
2. Trudy controls the minimum block hash.

6.2.1 Byzantine Client is not a block proposer

When Trudy receives block proposals, she chooses to suppress the minimum block hash and promote a different block. This obviously depends on whether Trudy gets selected in the committee to vote on blocks. We assume a scenario where Trudy is able to vote and generates a large number of sub-users or winning tokens. When she votes on a block, the total number of votes can cross the threshold, and she can reach consensus on that block.

To reiterate, the block that wins the consensus is not the best block that everyone has observed, but the one that Trudy promoted using her high-stake power.

6.2.2 Byzantine Client is a block proposer

In this scenario, Trudy has the ability to propose blocks with double spending and other malicious transactions, coupled with her voting and suppressing advantage described in the previous scenario. This block can achieve consensus and can be added to the blockchain.

Both scenarios are common to all PoS protocols where a dishonest party controls the majority of the stake. Although hypothetical, this is a plausible scenario that can affect the network and all other clients.

Safety is seriously undermined in the second scenario.

Liveness is guaranteed since blocks can be added to the blockchain in a timely manner.

6.3 Network Partition

A more common scenario where the network exhibits weak synchrony is when the system is partitioned into smaller groups. Communication happens only within the group and they are unable to reach to the other clients in the network. This reduces the participation and the voting power and as a result no block will ever receive true majority to add blocks. But for the sake of the simulation, let's assume the parameters are set so as to allow at least empty blocks to be published as fallback. The network partition can be described as:

```
let fakeNet1 = new FakeNet();
fakeNet1.register(alice, bob, charlie);
```

Alice, *Bob*, and *Charlie* belong to a group which uses *fakeNet1* as their communication medium. The messaging and broadcasts will only reach to these clients only. We start their

consensus for a short period where they are able to add one empty block. Figure 30 briefly describes this scenario.

```
nimesh@Nimeshs-MacBook-Air delegated-proof-of-stake % node driver-partition.js
Starting simulation.  This may take a moment...
Starting clients set - 1
Alice has 15 gold.
Bob has 10 gold.
Charlie has 20 gold.

[ Alice ] Received a proposal.
[ Bob ] Received a proposal.
[ Charlie ] Received a proposal.

[ Alice ] Received Termination Request.
[ Bob ] Received Termination Request.
[ Charlie ] Received Termination Request.
```

Figure 30: fakeNet1 participants start the round.

We describe another group of clients as:

```
let fakeNet2 = new FakeNet();

fakeNet2.register(minnie, mickie, trudy);
```

Minnie, *Mickie*, and *Trudy* form a group which uses *fakeNet2* as their communication channel. All messages will only be shared among these three clients. Similar to group 1, they are allowed to run for some time before terminating. A point to note here is that Trudy is not a Byzantine client. We have reused the client but in an honest setting now.

```
Starting clients set - 2

Minnie has 20 gold.
Mickie has 10 gold.
Trudy has 30 gold.

[ Minnie ] Received a proposal.
[ Mickie ] Received a proposal.
[ Trudy ] Received a proposal.

[ Minnie ] Received Termination Request.
[ Mickie ] Received Termination Request.
[ Trudy ] Received Termination Request.
```

Figure 31: fakeNet2 participants start the round.

Let's say at this point, both these groups were only able to produce empty blocks since a clear majority could not be established and the network establishes full communication between them. That is, all 6 clients can now reach each other. We can describe this in terms of our implementation as:

```
fakeNet1.register(minnie, mickie, trudy);

fakeNet2.register(alice, bob, charlie);
```

They all start producing blocks together now as described in figure 32.

```
Starting all the clients together.
Alice has 15 gold.
Bob has 10 gold.
Charlie has 20 gold.
Minnie has 20 gold.
Mickie has 10 gold.
Trudy has 30 gold.

[ Bob ] [ BLOCK_PROPOSAL ] Listening for other proposals.
[ Charlie ] [ BLOCK_PROPOSAL ] Listening for other proposals.


[ Alice ] [ RECEIVE_PROOF ] Received a proposal.
[ Bob ] [ RECEIVE_PROOF ] Received a proposal.
[ Charlie ] [ RECEIVE_PROOF ] Received a proposal.
[ Minnie ] [ RECEIVE_PROOF ] Received a proposal.
[ Mickie ] [ RECEIVE_PROOF ] Received a proposal.
[ Trudy ] [ RECEIVE_PROOF ] Received a proposal.
[ Minnie ] [ RECEIVE_PROOF ] Received a proposal.
[ Mickie ] [ RECEIVE_PROOF ] Received a proposal.
```

Figure 32: All clients start working.

Alice, *Bob*, and *Charlie* have produced an empty block. *Minnie*, *Mickey*, and *Trudy* have also produced an empty block, but it resolves to a different block hash. Now, when they all start the next round together, their *ctx.lastBlock* values will be different from the other group members, and during voting, this mismatch is detected, which essentially means a fork. Figure 33 describes this in our implementation. The application throws an error for this case, which terminates the node process. Algorand's implementation allows the clients to ignore the votes for a round for which the last blocks are different. It

implements a dedicated fork resolution that will solve this anomaly in the future. However, the current scope does not entail fork resolution.

```
/Users/nimesh/Desktop/CS-297/algorand-proof-of-stake/client.js:530
            throw new Error(`
            ^

Error:
                Possible fork detected.
                Received block that doesn't extend the chain:
                received -> "7d0a415746c2e37a2d581ce951c1fc100028a4e26793a9fbfb6db6d65468f33d"
                actual -> "c7911f1e98e81ccfdf6312af640412074188d92442ccd9b669cded22f22cfb24"
    at StakeClient.processMsg (/Users/nimesh/Desktop/CS-297/algorand-proof-of-stake/client.js:530:19)
    at StakeClient.countVotes (/Users/nimesh/Desktop/CS-297/algorand-proof-of-stake/client.js:573:57)
    at StakeClient.countReduceOne (/Users/nimesh/Desktop/CS-297/algorand-proof-of-stake/client.js:166:28)
    at Timeout._onTimeout (/Users/nimesh/Desktop/CS-297/algorand-proof-of-stake/client.js:154:18)
    at listOnTimeout (internal/timers.js:557:17)
    at processTimers (internal/timers.js:500:7)
```

Figure 33: Different *lastBlock* values are detected.

In this scenario, safety is undermined because the network can no longer guarantee that transactions are recorded in a single shared blockchain. Liveness is also affected since fork resolution can cause delays. In our implementation, if a fork is detected, the application crashes and prevents any further progress in the blockchain.

Table 2 summarizes the safety and liveness of all different scenarios.

Table 2: Analysing safety and liveness.

Sr. No.	Scenario	Safety	Liveness
1	Final consensus on a block.	Strongest	Strongest
2	Tentative consensus on a block.	Strong	Strong
3	Tentative consensus on an empty block.	Strong	Weak
4	No consensus after MAXSTEPS	Weak	Weakest
5	Byzantine Client	Weakest	Strong
6	Network Partition	Weak	Weakest

CHAPTER 7

Future Work

This research project focused on mainly the implementation that included Algorand's Byzantine agreement protocol and all its other subroutines and converting SpartanGold to a PoS model. There are however several problems that have been addressed and remain open for the scope of this project:

1. Algorand's verifiable random function (VRF) outputs the exact number of block proposers or committee members on the input threshold and the key pairs, which is a critical requirement for scaling the cryptocurrency. The implementation of cryptographic sortition in our project utilizes a third-party VRF library [24], resulting in random number of selected users and their sub-users in every round. VRFs in Algorand enable the selection of a fixed number of clients every round. Since this is only a simulation project focused on the key components of the PoS mechanism, future work could include implementing VRF to target current design parameters that would provide more precise control over the selection lottery.
2. In a real setting, any blockchain networks and cryptocurrencies would be dynamic, allowing clients to join or leave the network at their will. However, our research implementation does not account for clients joining in the middle of a round. This is because updating their copy of the blockchain to the latest state and downloading all finalized blocks would take time and identifying the waiting time before they can participate in the protocol is a complicated task.
3. In our implementation, we utilize the *setTimeout()* function to ensure all clients can execute their part and occupy the main thread. When a *setTimeout()* is called on a client, the main thread becomes available for other miners to perform their own

computations. While SpartanGold allows the execution in a multi-process mode, which is a more accurate simulation of a concurrent network, this is beyond the scope of this research. Future work could include exploring Algorand's implementation in a different multi-threaded language or allowing clients to run independently on different processes and communicate over TCP/IP.

4. As described in section 3.4, the seed used to propose blocks is refreshed every R rounds. This is beyond the scope of the project. In our implementation, every seed is derived from the seed of the previous block only.
5. When the protocol cannot reach consensus under *MAXSTEPS* in the BinaryBA⋆ implementation, as described in section 3.10.1, the research [21] indicates that Algorand performs a fork resolution to recover the liveness. However, this recovery mechanism is not included in the implementation presented in this research, and the algorithm will halt once MAXSTEPS is reached.
6. In the current implementation of the research project, if a fork is detected, the algorithm terminates the process with an error. However, in a realistic scenario, clients would ignore votes that do not pertain to the same *last_block* value, and one of the forks would eventually gather enough votes to continue adding blocks, making it longer. Future work could focus on ensuring liveness even with forks.
7. Section 3.11 describes Algorand's periodic execution of fork resolution, which follows a procedure similar to block proposal. However, this aspect of Algorand's protocol is beyond the scope of the current project. As mentioned before, in the event of a fork, the application terminates the process with an error, but implementing Algorand's fork resolution could be an area for future research.

8. The core of any cryptocurrency is processing transactions, which involves collecting and validating exchanges between clients. However, this project omits transactions to focus on how Algorand progresses through stages. Transactions can be taken up in the future to provide more realistic blockchain features.
9. Since the number of calculations is relatively small and there are only a few clients, the timeout for all the different stages of our implementation is set the same to increase usability and focus on the algorithm. Future development could dictate increasing the load and utilizing different timeout values for multiple stages.

CHAPTER 8

Conclusion

In this paper, we implemented Algorand's Byzantine consensus protocol for SpartanGold framework. Our experiments have demonstrated how the algorithm behaves under different scenarios such as network partitions, tentative consensus on empty and non-empty blocks, and the presence of Byzantine clients. We have also been able to understand why safety and liveness are affected differently in each scenario.

In situations where the network is able to reach final consensus, safety and liveness guarantees are the strongest. On the contrary, safety and liveness only get weaker as the network establishes only a tentative consensus or no consensus at all. Moreover, the presence of Byzantine clients holding the majority stake in the network affects safety the worst while network partitions can weaken both safety and liveness.

In summary, while this research and implementation have successfully documented a novel consensus protocol that can efficiently scale to many users and effectively deal with various scenarios, there is still a lot of room for future work and research in implementing Algorand's Byzantine consensus for SpartanGold. The SpartanGold framework already provides a starting point for academic research and rapid prototyping, and our implementation can serve as a base for future research and development of blockchain networks based on the Algorand consensus algorithm.

REFERENCES

[1] X. Yang, J. Liu, and X. Li, "Research and Analysis of Blockchain Data," *J. Phys. Conf. Ser.*, vol. 1237, no. 2, p. 022084, Jun. 2019.

[2] T. Salman, R. Jain, and L. Gupta, "Probabilistic Blockchains: A Blockchain Paradigm for Collaborative Decision-Making," in *IEEE UEMCON 2018*, 2018.

[3] L. Ismail and H. Materwala, "A Review of Blockchain Architecture and Consensus Protocols: Use Cases, Challenges, and Solutions," Sep. 2019.

[4] C. Dwork and M. Naor, "Pricing via Processing or Combatting Junk Mail," in *Advances in Cryptology — CRYPTO' 92*, 1993, pp. 139–147.

[5] J. R. Douceur, "The Sybil Attack," in *Peer-to-Peer Systems*, 2002, pp. 251–260.

[6] S. King and S. Nadal, "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake," Aug. 2012.

[7] V. Buterin, "On Stake," *Ethereum Foundation Blog*, 05-Jul-2014. [Online]. Available: https://blog.ethereum.org/2014/07/05/stake. [Accessed: 24-Apr-2023].

[8] C. T. Nguyen, D. T. Hoang, D. N. Nguyen, D. Niyato, H. T. Nguyen, and E. Dutkiewicz, "Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities," *IEEE Access*, vol. 7, pp. 85727–85745, 2019.

[9] "Delegated proof of stake (DPOS)," *Delegated Proof of Stake (DPOS) - BitShares Documentation documentation*. [Online]. Available: https://how.bitshares.works/en/master/technology/dpos.html. [Accessed: 24-Apr-2023].

[10] "Pure proof-of-stake," *Algorand*. [Online]. Available: https://algorand.com/technology/pure-proof-of-stake. [Accessed: 24-Apr-2023].

[11] J. Kwon, “Tendermint: Consensus without mining.” 2014.

[12] R. Han, Z. Yan, X. Liang, and L. T. Yang, “How Can Incentive Mechanisms and Blockchain Benefit with Each Other? A Survey,” *ACM Comput. Surv.*, vol. 55, no. 7, pp. 1–38, Dec. 2022.

[13] S. Nakamoto and W. bitcoin.org, “Bitcoin: A Peer-to-Peer Electronic Cash System,” Oct. 2008.

[14] V. Buterin, “Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform,” Nov. 2013.

[15] W. Warren, “ERC-721 non-fungible token standard,” *ethereum.org*, Mar-2018. [Online]. Available: https://ethereum.org/en/developers/docs/standards/tokens/erc-721/. [Accessed: 24-Apr-2023].

[16] “The Merge,” *ethereum.org*, Sep-2022. [Online]. Available: https://ethereum.org/en/roadmap/merge/#merge-and-energy. [Accessed: 24-Apr-2023].

[17] “Ethereum Energy Consumption,” *ethereum.org*. [Online]. Available: https://ethereum.org/en/energy-consumption/. [Accessed: 24-Apr-2023].

[18] I. Bentov, C. Lee, A. Mizrahi, and M. Rosenfeld, “Proof of Activity: Extending Bitcoin’s Proof of Work via Proof of Stake.”

[19] A. Kiayias, A. Russell, B. David, and R. Oliynykov, “Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol.”

[20] T. H. Austin, “SpartanGold: A Blockchain for Education, Experimentation, and Rapid Prototyping,” in *Silicon Valley Cybersecurity Conference*, 2021, pp. 117–133.

[21] S. Micali, M. Rabin, and S. Vadhan, “Verifiable Random Functions.”

[22] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling Byzantine Agreements for Cryptocurrencies.”

[23] I. Hagopian, “The intuition behind Algorand’s cryptographic sortition,” *Ignacio Hagopian (@jsign) blog*, 27-Mar-2019. [Online]. Available: https://ihagopian.com/posts/the-intuition-behind-algorands-cryptographic-sortition. [Accessed: 24-Apr-2023].

[24] “VRF-js,” *npm*. [Online]. Available: https://www.npmjs.com/package/@idena/vrf-js. [Accessed: 24-Apr-2023].