TÉCNICO LISBOA

Integrating asynchronous BFT with Ethereum validator networks

Matheus Guilherme Leça da Silva Franco

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Rodrigo Seromenho Miragaia Rodrigues
Prof. Henrique Moniz

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues
Member of the Committee: Prof. João Pedro Faria Mendonça Barreto

November 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I want to express my deepest gratitude to my supervisors, Rodrigo and Henrique, for giving me this opportunity and providing unwavering support throughout this research. Their expertise and willingness to guide me have been invaluable. Their mentorship not only shaped the academic aspects of this thesis but also fostered my personal and professional growth. This thesis stands as a testament to their exceptional mentorship, and I am honored to have had the opportunity to learn under their guidance.

Moreover, I want to thank my family for their encouragement and commitment to my education. Their emotional support, along with their dedication and hard work, made all of this possible for me. To my parents, sister, girlfriend, and extended family, thanks for your endless love and support.

Lastly, I want to express my gratefulness to my colleague and friend, Daniel Porto, who dedicated countless hours to helping me by discussing complex concepts, reviewing lines of code, and solving errors. However, his contribution extended far beyond the academic realm, supporting me and offering a friendly shoulder during difficult moments. His dedication was crucial for the completion of this journey. Daniel, you will always be a source of inspiration and a warm memory of friendship.

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 and PTDC/CCI-INF/6762/2020.

Abstract

The transition from Proof-of-Work to Proof-of-Stake by some blockchains, such as Ethereum, raised the bar for users to become validators and get the corresponding rewards. Recently, the concept of secret shared validators was introduced to overcome these hurdles by creating a network to run a validator in a decentralized and secure way. The consensus layer of such a network must run an efficient and robust Byzantine fault tolerance protocol in order to coordinate the network operators. This work aims to design and implement Alea-BFT, a robust, leaderless (making it inherently more resilient to node faults), and efficient asynchronous BFT protocol, with quadratic communication complexity, in the context of the SSV network. In addition to the design and implementation of this integration of the new protocol into SSV, an in-depth analysis and experimental evaluation shall be conducted in order to compare Alea-BFT's performance against other protocols, such as the ones that are currently adopted by the Ethereum foundation. Finally, we expect that such experimental evaluation will result in upgrades to Alea-BFT, making it more resilient to network or node faults.

Keywords

Alea-BFT; Asynchronous BFT protocols; Secret Shared Validators; Ethereum; Proof-of-Stake.

Resumo

A transição entre Proof-of-Work e Proof-of-Stake por algumas blockchains, como a Ethereum, elevou os requerimentos para usuários tornarem-se validadores e receberem recompensas. Recentemente, o conceito de Secret Shared Validators foi introduzido para solucionar estes problemas por meio da criação de uma rede com o objetivo de executar um validador de uma forma decentralizada e segura. A camada de consenso desta rede deve correr um protocol tolerante a falhas bizantinas eficiente e seguro com o propósito de coordenar os operadores da rede. Esse trabalho tem por objetivo desenhar e implementar Alea-BFT, um protocol robusto, indepedente de um líder (tornando-o inerentemente mais resistente a falhas de nós), e eficiente com complexidade de comunicação quadrática, no contexto da SSV network. Além do desenho e implementação da integração desse novo protocolo no contexto da SSV, uma análise profunda e avaliações experimentais serão conduzidas com o objetivo de comparar a performance do Alea-BFT contra outros protocolos, como o protocolo atualmente recomendado pela Fundação Ethereum. Finalmente, esperamos que a avaliação experimental tenha como resultado melhorias ao Alea-BFT, tornando-o mais resiliente a falhas de rede e de nós.

Palavras Chave

Alea-BFT; Protocolos BFT assíncronos; Secret Shared Validators; Ethereum; Proof-of-Stake.

Contents

List of Figures

List of Tables

Listings

1

Introduction

Contents

Ethereum is a decentralized, open-source blockchain that enables the creation of smart contracts and decentralized applications (dApps)[1]. It was initially released in 2015 and conceived by the programmer Vitalik Buterin. Nowadays, Ethereum is one of the most important blockchains in terms of market capitalization. At the beginning of 2022, it was approximately 450 billion dollars[2], losing only to Bitcoin.

Ethereum began by using Proof-of-Work (PoW) as its consensus mechanism, in which miners would compete to solve mathematical tasks in order to validate transactions and add them to the blockchain. PoW, however, has led to concerns such as its environmental impact due to its high energy consumption and the tendency to concentrate computing resources in locations where electricity is cheap.

This context gave rise to new consensus mechanisms such as Proof-of-Stake (PoS), which would later be adopted by Ethereum, promising to save up to 99,95% of Ethereum's energy use. In a PoS system, validators (also known as "stakers") are selected to validate transactions based on their stake in the network [1–3]. To this end, in the context of Ethereum, a user must, first, deposit 32 ETH into a

[1] V. Buterin, "Ethereum white paper: A next generation smart contract & decentralized application platform," 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[2] (2023, October) Ethereum's Market Capitalization History (2015 – 2023, $ Billion. Accessed 20-October-2023. [Online]. Available: https://www.globaldata.com/data-insights/financial-services/ethereums-market-capitalization-history/

deposit smart contract in order to become a validator. After that, it will be selected to validate blocks but its behavior will affect whether it will receive rewards for its work or be penalized for being offline or behaving maliciously.

Thus, the financial barrier and the computational resources necessary to run a validator can be challenging. Some validators use Staking-as-a-Service providers in order to run their duties on their behalf. This, however, represents a single point of failure in addition to requiring the provider to access the user's private key and, still, is susceptible to downtime.

To address these problems, the Ethereum Foundation proposed the idea of a Distributed Validator Technology (DVT). Then, a company named Secret Shared Validators (SSV)[3] came up with a secure and robust way to split an Ethereum validator key between non-trusting operators, decentralizing the validator's execution. SSV allows multiple parties to share the responsibilities and rewards of validating transactions while maintaining the secrecy of their identities. The operators representing the validator, however, should agree on what to send to the blockchain network, a problem solvable with Byzantine Fault Tolerant consensus protocols.

The BFT consensus (or agreement) problem was first introduced by Lamport, Shostak, and Pease in 1982 with the famous Byzantine Generals problem [4]. Byzantine fault tolerance is a property of a system described as its ability to function correctly even when some of its components may be faulty [4]. Since Lamport's original proposal, many BFT protocols have been developed, most of them only providing both safety and liveness in synchronous or partially synchronous settings [5]. Synchronous systems denote systems in which there is a known upper bound to communication time. Partially synchronous systems are those in which the upper time bound is not known or is only valid after an unknown interval, e.g. the IBFT [6] which is currently adopted by SSV. And, finally, in asynchronous systems, no assumption is made regarding communication time. The prioritization of synchronous and partially synchronous models was due to the FLP impossibility [7], which states that there is no deterministic solution for the consensus problem in an asynchronous system in which a single process can fail.

Recently, there has been a surge in research about asynchronous BFT consensus, notably since the proposal for a practical asynchronous BFT protocol called HoneyBadgerBFT [8], which was developed in 2017, with a $O(n^3)$ communication complexity. To overcome the FLP impossibility, randomization techniques were employed by which the protocol achieves consensus with a high probability as multiple rounds are performed. The advantage of asynchronous protocols relies on their resilience to the underlying network conditions, being robust to attacks or accidental faults in the network and nodes.

Following the interest in an efficient asynchronous BFT consensus protocol, Alea-BFT [5] emerged as the first practical protocol for real-world scenarios due to its quadratic communication complexity. It's

[3] A. Muroch. (2021, February) An Introduction to Secret Shared Validators (SSV) for Ethereum 2.0. Accessed 20-October-2023. [Online]. Available: https://blog.ssv.network/an-introduction-to-secret-shared-validators-ssv-for-ethereum-2-0-faf49efcabee

designed in a two-stage pipeline that combines a broadcasting and an agreement component that runs in parallel. It's built on top of common building blocks which allows for flexibility and improvement.

In summary, a key component of the SSV's validator network relies on a BFT protocol, which must be robust and efficient in relation to throughput and latency in order for the operators to be able to perform all validators' tasks in time. For that, we propose the integration of the Alea-BFT consensus protocol with SSV, along with a thorough experimental evaluation, comparing Alea-BFT to IBFT, the market standard protocol.

The biggest integration challenge faced was adapting Alea-BFT's state machine replication design to a problem requiring a single consensus execution. For instance, consensus instances are created whenever a validator duty needs to be solved. Nonetheless, it would be easier to fit Alea-BFT and performance would improve if the network was built as a state machine replication. To address this issue, we proposed several optimizations and analyzed their performance improvement both in normal scenarios and under unexpected behavior from both the nodes and the network.

First of all, we implemented an optimization to the agreement protocol (that can be extended to any context) which allows it to terminate faster in case all first binary votes are equal. Secondly, we created a delay before starting the agreement component in order for it to have a higher probability of success in the first run. Furthermore, we used the fact that most of the time the values proposed by operators are the same, allowing us to terminate the consensus in case we have equal operators' proposals. Lastly, we replaced the asymmetric signature scheme with message authentication codes, targeting the main performance bottleneck and improving the protocol performance.

Finally, we evaluated the prototype of the Alea-based SSV network and compared it to the current version of SSV. Throughout this process, we kept in touch with the engineering team at SSV and are now taking steps towards the integration of Alea into their code base. Our results show that Alea-BFT provides similar latency and throughput compared to QBFT (the IBFT implementation), for different system loads and network sizes. Also, Alea-BFT takes a step further showing great resilience to network instability and faulty nodes, being able to better maintain its latency and throughput under such scenarios. Therefore, Alea shows great potential as a practical asynchronous protocol, showing similar performance to QBFT and more resilience on faulty scenarios in a real-world application.

1.1 Organization of the Document

This thesis is organized into 6 chapters. In chapter 1, we present a brief overview of the root motivations for this thesis, namely the emergence of the validator networks, its necessity for a robust and efficient BFT protocol, and the evolving class of asynchronous BFT protocols. Chapter 2 provides background in several relevant topics for the understanding of the problem. It starts with a deep explanation of

Ethereum's consensus intricacies and elaborates on the functioning of an Ethereum validator. Then it explores the BFT consensus problem, explaining the important FLP impossibility theorem, and reviewing the asynchronous class of consensus protocols. Finally, it reviews in detail the Alea-BFT asynchronous protocol, explaining its components and asymptotical complexity.

In chapter 3, a global overview of the SSV architecture is presented. We highlight how the consensus module was designed and how Alea-BFT should be adapted to it. Then, we propose a list of protocol and cryptography optimizations that we incorporated into our design. Then, in chapter 4, we elaborate on the actual implementation of the SSV module, on Alea-BFT's implementation components, and on the private Ethereum test network required for the experimental setup.

Chapter 5 illustrates all the experiments performed and interprets each result obtained. It evaluates Alea-BFT against QBFT for different protocol optimizations, providing a breakdown of their performance, for cryptography optimizations, wide area setups, different network sizes, and fault scenarios. Finally, chapter 6 summarizes the document content and concludes.

2

Background & Related Work

Contents

This chapter has the goal of presenting the necessary knowledge required to understand the composition and functioning of Ethereum validator networks, as well as the evolution of asynchronous Byzantine Fault Tolerance protocols. The first section, 2.1, presents a brief introduction to the Ethereum blockchain which is followed by a thorough analysis of its first consensus mechanism, Proof-of-Work (PoW), and the rationale for the later adoption of Proof-of-Stake (PoS).

In section 2.2, we detail the functioning of a validator in the PoS mechanism, shedding light on its challenges and on the Secret Shared Validators' innovative solution. This solution is built upon three main components: a secret sharing protocol, an additive signature cryptographic scheme, and a BFT consensus protocol. We explain how secret sharing and aggregating signatures make it possible for decentralized validators to exist. Lastly, we go through each validator duty and how it can be executed in the SSV context.

Section 2.3 goes through the evolution of the BFT consensus protocols, detailing the possible communication models and the famed FLP impossibility result. We provide an overview of the Istanbul BFT protocol, currently recommended by Ethereum for the validator network. Then, we go through the realm of the asynchronous BFT protocols, delving into techniques used to overcome limitations posed by the FLP theorem and exploring its main protocols. Finally, in section 2.4, Alea-BFT is carefully explained, along with its broadcast and agreement components.

2.1 Ethereum

Ethereum is one of the most famous and important blockchains today. Though Bitcoin has the highest market capitalization among all cryptocurrencies, the design of Ethereum represented a major step in blockchain technology. While Bitcoin was responsible for uncovering a way to decentralize a currency without any central counterparty, Vitalik Buterin, in the 2014 Ethereum's white paper[1], took a step beyond the vision of Bitcoin by delivering apps and transaction protocols into the blockchain world. More than its token, ether, Ethereum, which became known as the "computer of the world", is a way to deploy decentralized applications (dApps) and smart contracts on the web. It is fast, secure, and does not rely on any authority.

2.1.1 Proof-of-Work

Diving into its mechanisms, Ethereum and most blockchains from the same time period used Proof-of-Work (PoW) [9] as its consensus model. PoW relies on finding a value for a field of the block, the nonce, such that the block's hash begins with a certain number of zero bits. The number of bits required to be zero represents the difficulty level and it can be adjusted in order to ensure an average time for the nonce to be found.

So, in order to add a block to the blockchain, one must keep trying different values for the nonce until the block's hash meets this requirement. This is difficult due to the randomness of the hash function. A node that spends computational power to solve this PoW puzzle is called a miner. As soon as miners receive a block, they compete with each other to solve the task first and get a reward by adding the block to the blockchain. It's important to note that the safety of the PoW mechanism is sustained on the assumption that the majority of miners act honestly and do not collaborate to manipulate the network. If this assumption does not hold, i.e. if a single entity or a coalition of miners controls more than 50% of the network's computational power (51% attack), attackers would be able to control the blockchain and prevent new transactions from being confirmed [10].

[1] V. Buterin, "Ethereum white paper: A next generation smart contract decentralized application platform," 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

This hash validity property allows for the blockchain to be highly sensitive to any modification in a block. Altering any piece of data in a block implies altering its hash, and, thus, turning it invalid. Moreover, each block in the blockchain contains the hash of the previous block, so any changes to a block B_X would require recomputing all the nonces for the blocks $B_{X+1}, B_{X+2}, ...$ that come after it. This is the machinery that makes it so difficult for a malicious user to try to modify the blockchain. If one wants to modify a single block, it needs to compete against all other miners in the network. In other words, it would need at least 51% of the total computational power of the network.

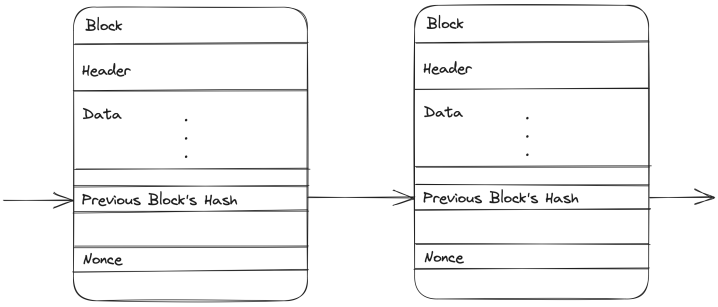


Figure 2.1: Block relation to the previous block in a Blockchain.

In this architecture, the expected time for a block to be added to the blockchain is dictated by the mining time. The blockchain increases as soon as a miner creates a block, finds the nonce, and broadcasts this valid block. Note that any miner can be a block proposer at any given time. Whether it will succeed or not depends on whether it creates a valid block faster than other miners.

However, this architecture may give rise to ambiguous situations. For example, when two different miners propose different valid blocks to be included at the same time. This constitutes a temporary fork. In such cases, other nodes need to decide which fork path they will choose in order to continue adding new blocks. This is accomplished with the fork-choice rule, which selects the set of blocks that have had the most work done to mine them.

Since this temporary fork situation may happen, a transaction cannot be considered completed only because it was added to a block that was proposed by a miner. If the network decides on a forked path in which the transaction wasn't included, then the transaction may be reversed. This leads to the definition of a property named "finality". A transaction has "finality" if it's part of a block of the blockchain that can no longer be altered. In PoW, finality is probabilistic. As more blocks are added to a block that contains the transaction, the probability of this transaction not being reversed becomes higher.

Proof-of-Work, though secure, has several concerns. First of all, a lot of energy is consumed as miners try to outdo each other. The energy consumption is so large that, in 2021, if Bitcoin was a country, it would be the top 30 energy consumer worldwide[2], ahead of Argentina and the Netherlands,

[2]C. Criddle. (2021, February) Bitcoin consumes 'more electricity than Argentina'. Accessed 20-October-2023. [Online]. Available: https://www.bbc.com/news/technology-56012952

for example. Additionally, PoW presents the potential for centralization due to the resource-intensive nature of mining. This can lead to a concentration of mining power in the hands of a small number of individuals or groups with access to abundant resources. Moreover, miners often seek out locations with low energy costs to maximize profitability, which can result in further centralization and negative environmental and economic consequences [11–13].

2.1.2 Proof-of-Stake

To address these issues, alternatives to Proof-of-Work began to appear in the blockchain community, namely the Proof-of-Stake class of protocols [1–3]. A major problem of PoW is the competition between miners, which causes redundant computations with tremendous energy consumption. Therefore, a better approach would be to elect a node as the block proposer. This isn't so easy, however.

Suppose that nodes register themselves in a shared list by broadcasting a certain type of message and that there is some mechanism for electing a node to be a proposer. In this design, several problems arise, for instance, a selected node may be temporarily unavailable, may maliciously propose an invalid block, not propose a block at all on purpose, or even propose multiple blocks. There's the risk of Sybil attacks since one can easily create many identities and attempt to control the network. Additionally, it would require the network to perform an extra consensus specifically for integrating a new node into our shared list and for electing a block proposer. The protocol should, moreover, be fair when electing the block proposer, i.e. it should not give preference for a node when there's no reason to do it.

Thus, when given the opportunity to propose a block, a malicious node has significant power to act against the network. To avoid this undesired behavior from happening, the node should be penalized. Therefore, in PoS, instead of only rewards, a node may also be penalized if its behavior is considered invalid. This represents a major change from the PoW protocol. Also, note that in order for a node to lose capital, it should have, at first, made a certain amount of it available for the network, or "staked" in the PoS vocabulary. The node that does this is called "staker". Only after staking a required amount of capital, a node is able to become a "validator" and perform validator tasks, such as block proposal. The burn of staked capital as penalization should force the staker to behave correctly. After a series of penalizations, the validator may even be slashed from the network, i.e. it's forcibly ejected from the network while continuously losing its stake. Note that this monetary commitment also serves as a defense against Sybil attacks by making the attack too costly.

An issue to be considered with this architecture is that, even though some nodes may take longer than others to produce a block, the network can't keep waiting forever for a proposer to propose a block. To deal with this, a time limit is assigned, namely a "slot" which takes 12 seconds to be completed. This represents another major change from the PoW architecture. In PoS, the blockchain increases as these time periods, or slots, are completed, instead of increasing as a function of the mining time duration.

After a slot is finished, the next proposer takes the last valid proposed block as the current head of the chain to create and propose its own block.

To simplify the overhead of the block proposer selection, these slots are organized in epochs. One epoch comprises 32 slots (6.4 minutes). At the beginning of each epoch, the block proposer for each slot in the epoch can be computed. This makes the process of selecting a proposer much simpler than having to select it for every slot. It's important that the selection process includes some randomness in order to prevent attacks that target the upcoming proposers. This randomness in selecting a proposer comes from the *RANDAO* protocol. In simple terms, entropy is created by mixing, with the *XOR* operation, the hash of the proposers' signatures over the current epoch number. This number is also combined with the previous epoch's *RANDAO* value and is used to calculate the block proposers for the next epoch.

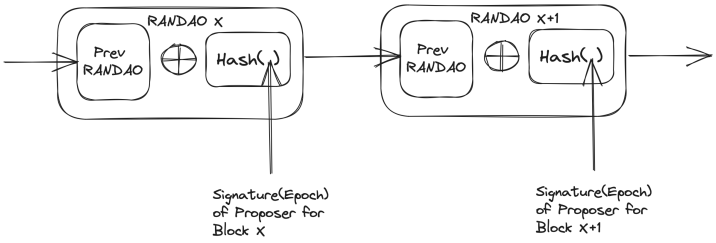


Figure 2.2: *RANDAO* calculation.

Since each proposed block needs to be verified, other validators should be in charge of validating it, or "attesting" it in the PoS vocabulary. It would be unreasonable to require all validators to perform attestation for every single block that is added to the blockchain and, therefore, a committee is elected to perform this task for a certain slot. However, to divide the load evenly among participants, every validator must perform one unique attestation every epoch.

Another important attribute that comes with this epoch feature regards the finality of a transaction. In PoW, the guarantee that a transaction wouldn't be reverted was probabilistic. In PoS, the finality of a transaction may be guaranteed using this epoch concept. The main idea is that the first block of an epoch may serve as a checkpoint. During the epoch, validators vote for the last pair of checkpoints that it considers to be valid. After a pair has received enough votes, which is represented by two-thirds of the amount of staked capital, the more recent checkpoint is tagged as "justified". The former one, which was also once "justified" in the previous epoch, becomes "finalized". Thus, we can guarantee that a transaction has finality when it's in a block of an epoch with a finalized checkpoint. This mechanism is known as Casper FFG and the Casper Vote is included in the attestation vote that the validator casts every epoch.

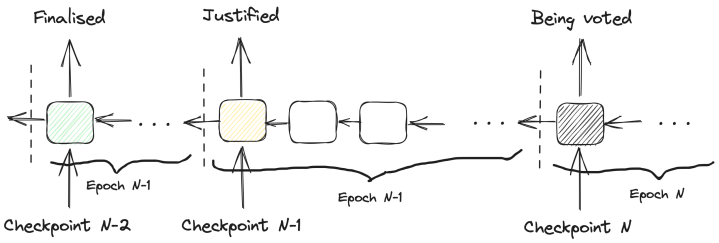


Figure 2.3: Checkpoints and Finality illustration.

A benefit of such architecture is that, in a normal scenario, the possibility of a temporary fork is mitigated. However, even if the proposer manages to create a fork, e.g., by broadcasting more than one block, a fork choice rule allows the nodes to identify the correct fork path. Differently from PoW, the chosen fork path is the one with the highest number of weighted attestations. The weight of an attestation is the amount of coins that the attestator has staked in the network. This is known as the LMD GHOST fork choice rule.

With this robust consensus mechanism in hand, which can replace and solve some of the problems with PoW, Ethereum planned to replace PoW with PoS and promised to save up to 99,95% of its energy consumption. This upgrade, called the ”Merge”, was performed on the Ethereum Mainnet on September 2022[3].

Since replacing the consensus mechanism was a substantial change to the network, susceptible to unexpected errors and problems, the Ethereum Foundation organized a plan to allow for a smoother transition. The strategy relied on the launch of a new Proof-of-Stake blockchain called Beacon Chain, totally separated from the Ethereum Mainnet at the time, but which would subsequently be merged with it. In December 2020, the Beacon Chain started by running in parallel to the Mainnet in an independent way. Then, upon the Merge event, the Beacon Chain started to accept transactions from the classic Ethereum network and the Proof-of-Work mechanism stopped working.

2.2 Running a Validator

2.2.1 Challenges

In Ethereum, to become a validator, a user must stake 32 ETH. Then, it becomes able to validate blocks and receive rewards for its job. Note that 32 ETH, at the beginning of 2022, was equivalent to 106k USD[4], which is a large sum of money and can represent an entrance barrier. Therefore, this contributes to the security of the blockchain but imposes limits on who can participate in it.

[3] P. Wackerow. (2022, August) Ethereum Development Documentation. Accessed 20-October-2023. [Online]. Available: https://ethereum.org/en/developers/docs/

[4] (2023, October) Ethereum USD Historical Data. Accessed 20-October-2023. [Online]. Available: https://finance.yahoo.com/quote/ETH-USD/history/

Moreover, being a validator can be demanding since the blockchain expects validators to maintain a high uptime level and to have the necessary hardware and infrastructure to support its tasks. And, as mentioned before, if the validator fails to act as expected, maybe due to inactivity, poor network, or malicious actions, it will be penalized by losing some amount of its staked capital.

Therefore, the old miner, who had no barrier to participate and earn money in the blockchain, now faces some technical requirements plus a substantial financial barrier. This made part of the community skeptical about the benefits of staking, which motivated solutions to mitigate this barrier.

2.2.2 Distributed Validator Technology

To solve these technical difficulties, some users looked for alternatives such as Validator-as-a-Service (VaaS)[5]. This consists of a service that a company offers to its users in which it provides the necessary hardware and infrastructure to support the duties of a validator. It abstracts, thus, the user from all the necessary resources and technical details of the validator process.

There are some problems with this solution, nonetheless. Every user on the blockchain has a private and a public key. The validator must sign blocks while voting for them to be integrated into the blockchain and this is only possible with the validator's private key. Thus, VaaS providers must also possess their users' private keys, which represents a security risk. Furthermore, since a VaaS is centralized, it is still susceptible to downtimes, pushing consumers to look for the top companies, and creating a centralization power in the hands of few entities.

It was in this scenario that the concept of Distributed Validator Technology (DVT) emerged as a new way to run a validator as a decentralized process, solving the centralization and security issues of VaaS. As a high-level summary, the DVT technology is composed of a Key splitting mechanism, an additive signature scheme, a BFT consensus protocol, and a contract network that associates the validator to a set of distributed operators.

In DVT, multiple parties jointly operate a validator node without revealing their identities to each other. This can be accomplished by splitting the validator's key between the mutually distributed operators, employing techniques such as the Shamir Secret Sharing algorithm [14] for example.

The decentralization is only possible because Ethereum's validator keys utilize the BLS12-381[6] additive signature scheme defined over elliptic curves. This means that if a predetermined number of parties sign the same data with their respective shares of a private key, a full signature can be obtained by combining the parties' signatures without ever using the validator's private key. As a result, multiple parties

[5]M. Schmiedt. (2020, June) Secret Shared Validators on Ethereum 2.0. Accessed 20-October-2023. [Online]. Available: https://medium.com/coinmonks/secret-shared-validators-on-ethereum-2-0-ea29ab380016

[6]A. Muroch. (2021, February) An Introduction to Secret Shared Validators (SSV) for Ethereum 2.0. Accessed 20-October-2023. [Online]. Available: https://blog.ssv.network/an-introduction-to-secret-shared-validators-ssv-for-ethereum-2-0-faf49efcabee

can run a single validator, and, even if some of them are faulty, the system can continue to operate, helping to maintain its liveness in validating blocks and avoiding the single point of failure problem.

The operators that run a validator must reach a consensus on the data to be sent to the blockchain network (e.g., a signed proposal block). This corresponds to the consensus layer of DVT. In the original proposal, the Ethereum foundation recommends the Istanbul BFT algorithm [6] for this purpose, which is a consensus protocol for partially synchronous settings (as will be detailed next) with $O(n^2)$ communication complexity defined as an upper bound on the total number of bits exchanged by the algorithm.

2.2.3 Shamir Key Sharing

One of the most important attributes of a validator is its private key, which is used to sign every network message. The validator's private key can not be given to any of the nodes that will run it in a decentralized way since the operators cannot be individually trusted, as it would expose the validator to security risks. The immediate problem that arises, thus, is: how is it possible to create a signature with the validator's private key in a decentralized setting and not expose the key to any node?

A similar problem was developed in a 1979 paper named "How to Share a Secret" by Adi Shamir [14]. The paper begins with the following combinatorial mathematics problem: "Eleven scientists are working on a secret project. They wish to lock up the documents in a cabinet so that the cabinet can be opened if and only if six or more of the scientists are present. What is the smallest number of locks needed? What is the smallest number of keys to the locks each scientist must carry?".

The answer to the problem is $C(11, 5) = 462$ locks and $C(10, 5) = 252$ keys. Note that the solution grows with the factorial of the number of scientists. On the other hand, Adi Shamir proposed a beautiful and simple solution that uses polynomial mathematics.

The key insight that he had is the following: Similar to the scientists' problem, a polynomial can only be reconstructed when a sufficient number of points is provided, depending on its degree. Thus he proposed the following construction:

1. using a bijective function, transform the data (i.e., the secret) into a number D
2. let q be a $k - 1$ degree polynomial, where k is the number of parties needed to uncover D
3. set $q(0) = D$ and let the other coefficients to be random
4. give for party i the share $D_i = q(i)$

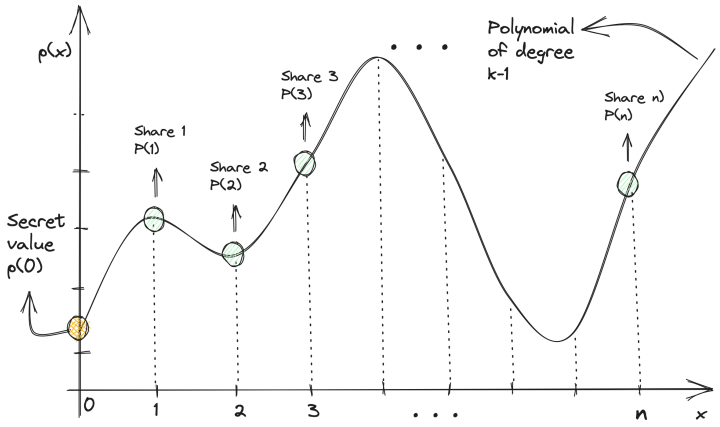


Figure 2.4: Shamir Secret Sharing illustration.

With such a scheme, the polynomial can be computed given any k points, or more, by polynomial interpolation, and, then, $D = q(0)$ can be uncovered. With less than k points, every value for D has equal probability. Note that we may even add new nodes without having to recalculate the polynomial (if we want the threshold number of parties to be the same). Also, this scheme is practical and efficient since there are $O(n \log^2 n)$ algorithms for polynomial evaluation and interpolation.

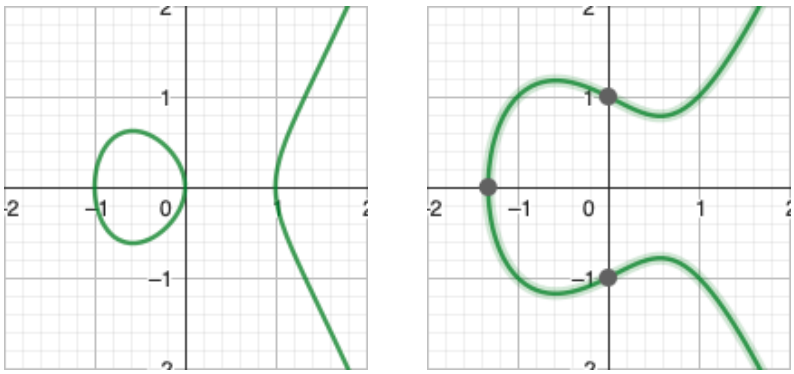
2.2.4 BLS Signatures

We already know how to split a key, but this would be pointless if the operators were supposed to calculate the validator's private key in order to produce a signature. It wouldn't solve the security problem since the key would be exposed and an operator could later pretend to be the validator.

That's where the BLS signature scheme is useful. It allows the operators to produce a signature as if it was created by the validator's private key without ever needing to explicitly compute it. Next, we briefly go through the BLS scheme in order to explain how it is possible to produce the signature in a decentralized way.

The BLS scheme uses elliptic curve cryptography. Equation (1) is the general equation for an elliptic curve, where a, $b \in \mathbb{R}$ and $4a^3 + 27b^2 \neq 0$ to avoid singular curves. Geometrically, there is a symmetry axis on $y = 0$. This symmetry serves as the foundation for the group structure that elliptic curves possess. The points on the curve, including a special point at infinity, form an additive Abelian group under the point addition operation. This operation takes two points P and Q, draws a line through them, and finds a third point R that lies in the curve. Reflecting R across the symmetry axis yields $P + Q$. Examples of such functions are shown in figure 2.5.

$$y^2 = x^3 + ax + b \tag{2.1}$$



(a) $y^2 = x^3 - x$ **(b)** $y^2 = x^3 - x + 1$

Figure 2.5: Graphs of elliptic curves

In cryptography, it's common to restrict the domain of such functions to finite cyclic groups of prime order. In simple words, the domain is restricted to integers and modular arithmetic is used with a prime order ((mod p) with p prime) such that there is an element, called the generator, which can derive every other element of the field by scalar multiplication of itself. An example of the curve $y^2 = x^3 + x$ restricted to the field F_{23} (integers modulus 23) is shown in figure 2.6.

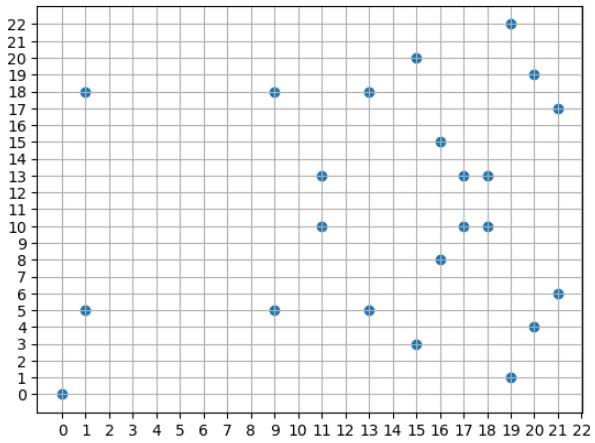


Figure 2.6: Elliptic curve $y^2 = x^3 + x$ restricted to F_{23}.

The generator, or base point, is usually denoted as G. The number of points that G generates is defined as the order of the curve. In the Elliptic curve scheme, a private key, sk, can be any number between 1 and order -1 while the relative public key, pk, is the private key multiplied by the base point, $sk \cdot G$. This is precisely the Elliptic Curve Discrete Logarithm Problem (ECDLP). It's computationally easy, given G and sk, to compute the public key, $pk = sk \cdot G$, but hard, given pk and G, to compute sk.

The BLS signature scheme also utilizes bilinear maps. A bilinear map is a function $e : G_1 \times G_2 \longrightarrow G_T$, where G_1, G_2, and G_T are multiplicative groups with the same prime order p. It satisfies two

properties (where g_1 and g_2 are the generators of G_1 and G_2 respectively) [15]:

- Bilinearity: $\forall v \in G_1, w \in G_2, a,\ b \in Z_p : e(a \cdot v, b \cdot w) = a \cdot b \cdot e(v, w)$
- Non-degeneracy: $e(g_1, g_2) \neq 1$

In the BLS signature scheme, the cryptographic primitives are defined as:

- Signature Generation: the scalar $s = sk \cdot H(m)$, where $H(m)$ is the hash of the message being signed.
- Signature Verification: a signature s over a message m for a public key pk is valid if $e(g_1, s) = e(pk, H(m))$. Notice that indeed $e(pk, H(m)) = E(sk \cdot g_1, H(m)) = E(g_1, sk \cdot H(m)) = e(g_1, s)$.

Going back to the secret key, if we define the polynomial in the finite field (equation 2.2) and give the shares to each party, each party will be able to produce a signature with its share (equation 2.3).

$$q(x) = a_0 + a_1 x^1 + ... + a_{k-1} x^{k-1} \text{mod}\, p \tag{2.2}$$

$$s_i = q(i) \cdot H(m) \tag{2.3}$$

But, to transform $q(i)$ to $q(i) \cdot H(m)$ is precisely the same thing as to transform q by $q \cdot H(m)$. Thus, the secret, $q(0)$ becomes $q(0) \cdot H(m)$ which is exactly the signature over m created by the validator's private key.

$$\begin{aligned} H(m) \cdot q(x) &= H(m) \cdot (a_0 + a_1 x^1 + ... + a_{k-1} x^{k-1}) \\ &= H(m) \cdot a_0 + H(m) \cdot a_1 x^1 + ... + H(m) \cdot a_{k-1} x^{k-1} \text{mod}\, p \end{aligned}$$

Therefore, using Shamir's secret sharing technique and the BLS signature scheme, it's possible to decentralize the validator's duties by producing its signature with an interpolation of the signature shares.

2.2.5 Validator duties

We have explored two fundamental components for validator decentralization: the key splitting mechanism and the additive signature scheme. When integrated with a Byzantine Fault Tolerance (BFT) consensus protocol, these elements form the essential building blocks for a fully decentralized validator. Now, we delve into the practical application of these components, detailing the operator steps for the five duties assigned to a validator: Block Proposal, Attestation, Attestation Aggregator, Sync Committee, and Sync Committee Aggregator.

Block proposal

For every slot, a validator is randomly selected (using the RANDAO value) to propose a block in a way that the probability of having a proposal duty for a certain slot is $1/N$, where N is the number of validators in Ethereum.

Once a validator is a proposer, it has to fetch the head of the chain using the fork choice rule, construct a block, sign it, and broadcast it to the network. The part of fetching the head of the chain and constructing a block can be abstracted by a Beacon client service, such as Prysm, Lighthouse, Lodestar, Nimbus, or Teku[7]. To sign the block, the signature shares need to be combined with the additive signature scheme, while consensus is used to decide on the block to be submitted.

To construct and return a block, the Beacon client requires the validator signature over the epoch number. Therefore, before doing consensus (pre-consensus phase), operators exchange partial signatures over the epoch number, reconstruct the validator signature, and, then, get the block from the client. After that, each operator, with a block from its client in hand, starts the consensus phase. After a block is decided, a post-consensus phase starts to collect partial signatures for the decided block, reconstruct the validator signature, and broadcast the signed block to the network.

Attestation

Every validator must produce one attestation every epoch. The slot in which it participates can be computed by the *RANDAO* of the previous epoch, giving extra time for the validator to know in advance its slot. The probability of having the attestation duty for a certain slot is simply $1/32$ since there are 32 slots in an epoch and the *RANDAO* value is a pseudo-random number. Particularly for the attestation duty, the validator may complete it in 32 slots, counted from the slot it was assigned, but with reduced rewards depending on the delay.

To create an attestation, the validator needs to get the block of the previous slot, create an Attestation object composed of the slot, its committee index, the beacon block root (the LMD GHOST vote), a source and target checkpoints (FFG Casper vote), and send it to its committee. To give time for the last block to be completely broadcasted, the operator awaits 4 seconds (one-third of the slot) before creating the Attestation object. Note that in this case, no pre-consensus phase is required, the object of consensus is the Attestation object and the post-consensus phase is necessary to construct the signature over the object.

Attestation Aggregator

For every slot, committees of validators are assigned to perform the attestation duty. At maximum, there are 64 committees per slot, each with a minimum of 128 and at most 2048 validators. In order not to flood the network with attestations, 16 members of the committee are selected to aggregate all

[7] C. Smith, "Nodes And Clients", 2023. [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/

attestations and broadcast the aggregation to the network, which is possible due to the aggregation property of BLS signatures. The probability of being an attestation aggregator for a certain slot is in the interval $[\frac{1}{32} \times \frac{C(15,2047)}{C(16,2048)}, \frac{1}{32} \times \frac{C(15,127)}{C(16,128)}]$ $[2.44 \times 10^{-4}, 3.9 \times 10^{-3}]$.

For the attestation aggregation duty, the validator needs to collect Attestation votes similar to its own vote and create an AggregateAndProof object, sign it, and broadcast it to the network. The AggregateAndProof object can be obtained by the Beacon client but one of its fields is the validator's signature over the slot value (to prove that it's an aggregator). Thus, this duty also requires a pre-consensus phase, followed by a consensus over the AggregateAndProof object and a post-consensus to construct the signature over the decided value. The operator waits 8 seconds (two-thirds of the slot) to give time for the attestations to be produced.

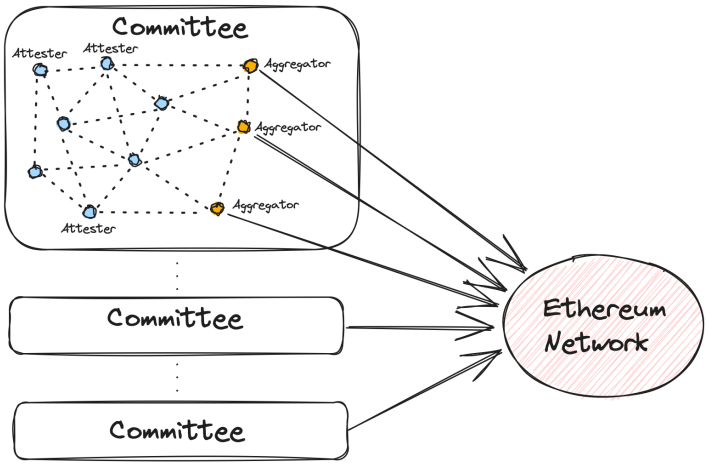


Figure 2.7: Attestation committees.

Sync Committee

Since running a full Ethereum node may be not feasible for everyone, Ethereum allows for the existence of "light nodes", which only keep track of headers of blocks in the chain. To help light clients keep track of the chain, the Sync Committee duty was created. It's similar to an attestation but a committee of 512 validators is selected to sign the block header that is the new head of the chain every slot during 256 epochs ($\approx$ 27 hours). The probability of participating in a sync committee is thus $\frac{C(511,N-1)}{C(512,N)} \approx 7.31 \times 10^{-4}$, for $N = 700,000$.

For this duty, the validator constructs a SyncCommitteeMessage object related to the previous slot and broadcasts it in the committee subnet. No pre-consensus is needed and the operator waits 4 seconds as in the attestation duty.

Sync Committee Aggregator

Similarly to attestation aggregation, members of the Sync Committee are selected to aggregate the messages. In particular, there are 16 aggregators, which makes the probability of being a sync

committee aggregator, for a certain slot, be $\frac{C(511,N-1)}{C(512,N)} \times \frac{C(15,511)}{C(16,512)} \approx 2.29 \times 10^{-5}$.

As in attestation aggregation, a pre-consensus phase is required to compute the validator's signature over a selection-proof object and verify if it's an aggregator or not. Also, the operator waits 8 seconds to give time for the sync committee messages to be broadcast.

2.3 BFT Consensus

The concept of Byzantine fault tolerance was first conceived by Lamport, Shostak, and Pease, in 1982, in a paper entitled "The Byzantine Generals Problem" [4]. The problem was to achieve consensus between generals who send messages to one another but don't necessarily trust each other. Byzantine fault tolerance was defined as the property of a system that allows it to continue to function correctly even when some of its components may crash or even work in a malicious way.

To compare the complexities of different BFT protocols, we rely on the message, communication, and time complexities. Here, message complexity is defined as the expected number of messages generated by correct replicas while communication complexity is the expected number of bits in messages generated by correct replicas. Time complexity can be defined in different ways depending on the protocol format, but we define it as the expected number of rounds of a protocol until it terminates.

2.3.1 Communication models

When the system is composed of processes trying to reach consensus, a key aspect is the type of communication they are submitted to. There are three ways to classify it: synchronous, partially synchronous, and asynchronous models. In synchronous systems, there is a known upper bound to communication time. Thus, timeout techniques can be applied to verify if a node is faulty or functional. Partially synchronous systems are those in which the upper time bound is not known or is only valid after an unknown interval. In asynchronous systems, no assumption is made regarding the communication time.

2.3.2 FLP Impossibility

One of the most important results regarding faults in the Byzantine fault tolerance problem is the Fischer-Lynch-Patterson impossibility [7]. Basically, it mathematically defines the problem as an automaton with a state space, transition functions, and deciding states, and demonstrates that if at least one process is faulty, defined in the paper as a process that can take a finite number of steps, in an asynchronous communication scenario, then there is always a possible sequence of steps that prevents

the system from terminating. This is accomplished by two lemmas. Firstly, it demonstrates that a protocol always has an initial configuration that can later decide on different outputs, defined in the paper as a bivalent configuration. Then, it shows that applying a step to the set of reachable states from a bivalent configuration produces a set that also contains a bivalent configuration. And, therefore, every configuration after any infinite schedule of transitions is bivalent.

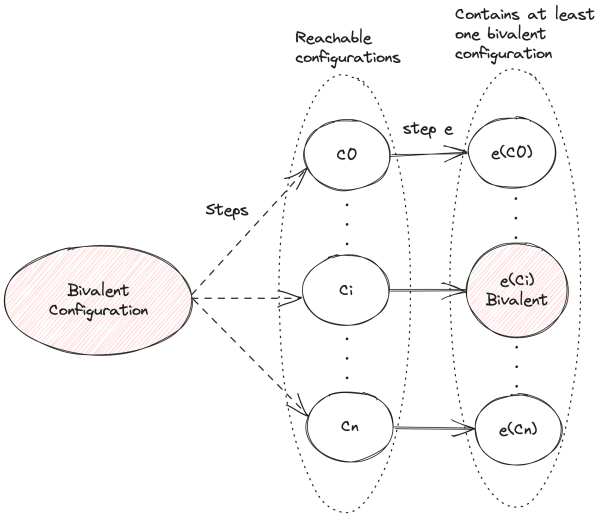


Figure 2.8: 2nd lemma to prove the FLP impossibility.

The paper, later, shows that it's possible, though, to reach a consensus with a partially synchronous setting if we restrict a majority of processes to be non-faulty. This result drove most solutions to BFT to use the assumption that the system is synchronous or partially synchronous.

One of the most famous partially synchronous protocols is the Practical Byzantine Fault Tolerance (PBFT) [16], introduced in 1999 by Castro and Liskov. PBFT has a message complexity of $O(n^2)$ where n is the number of nodes. It took a major step towards the practical adoption of these protocols, in showing that MACs (Message Authentication Codes) could be used instead of costly digital signatures to authenticate messages, providing much better performance.

The PBFT protocol is divided into consensus rounds which are divided into 4 phases. In the first one, the client sends a request to a leader node. In the second phase, the leader broadcasts the request to all other nodes. Subsequently, all nodes perform the task and reply to the client who, finally, accepts the response after $f + 1$ messages are received, where f is the maximum number of faulty nodes.

2.3.3 Istanbul BFT (IBFT)

Another partially synchronous protocol that is relevant for this thesis, since it's the current one adopted by SSV, is the Istanbul BFT (IBFT) [6], also referred to as QBFT, which was inspired by PBFT. It

was introduced to Ethereum on the Istanbul hard fork, released in 2019, and it's the main recommendation by the Ethereum foundation for the DVT technology. It has a quadratic message and communication complexity and proceeds in rounds (or views) in which there are four main phases: PRE-PREPARE, PREPARE, COMMIT, and ROUND-CHANGE.

Each round has a leader who proposes a value. The leader can be discovered by all parties, using a common function $LEADER$. The $LEADER$ function receives as input two variables: the round number and a variable called $height$. The $LEADER$ function can be any deterministic function that allows at least $f + 1$ nodes to eventually be the leader. The $height$ variable identifies an instance of the algorithm (in other words, it corresponds to the number of state machine commands that have been executed previously), never changing through the execution of a given consensus instance. Moreover, each replica maintains two other variables, $preparedRound$ and $preparedValue$, with the highest round and the corresponding value that was prepared.

On start-up, the nodes receive as input a value and the $height$ variable. If it's the leader for the current round, it broadcasts a $PRE-PREPARE$ message with the round number and the input value. Upon receiving this message, other nodes broadcast a $PREPARE$ message with the value and round number received. After receiving a quorum ($\lfloor \frac{n+f}{2} \rfloor + 1$) of $PREPARE$ messages from different parties, a node updates its $preparedRound$ and $preparedValue$ variables and broadcasts a message $COMMIT$ with the value. Finally, after receiving a quorum of $COMMIT$ messages, each node outputs the value and terminates the execution.

Since nodes wait until a quorum is reached, the liveness property must be assured with a timeout mechanism. If a timeout is reached, nodes send a $ROUND-CHANGE$ message to the new leader who, after receiving a quorum, starts the new round. The $ROUND-CHANGE$ message includes the values of $preparedRound$ and $preparedValue$, which will be used by the new leader who will broadcast the $preparedValue$ received, if there's one, or broadcast its own value, otherwise.

2.3.4 Asynchronous protocols techniques & overcoming FLP

Though partially synchronous protocols can be efficient, they need synchrony between replicas in order to ensure the liveness property. When the system suffers from interference from a malicious agent, the protocol can be prevented from making progress, or at least can have its performance degraded. The truth is that real-world communication has no upper time limit and, when network conditions are unstable, those protocols, including PBFT, are severely affected. In particular, in prior research [8], partially synchronous BFT protocols were tested against a malicious network scheduler and, by delaying messages at certain time points, it was possible to stop the protocol from making any progress.

Synchronous and partially synchronous protocols, nonetheless, might seem, at first glance, to be the

only possible solution due to the FLP impossibility. However, there are techniques, such as randomization, by which asynchronous protocols may circumvent this theoretical barrier. Without demanding any timing assumptions, they can guarantee the liveness property with a certain probability, which increases over time. The key idea is to run rounds, that try to reach a consensus, multiple times. As the number of rounds increases, the probability of reaching a consensus increases, and the probability of not having liveness becomes negligible.

In contrast to partially synchronous protocols, which may be severely affected by network conditions, asynchronous protocols are much more resilient to them. In particular, their performance is directly related to the actual network latency and these protocols can make progress as soon as messages are delivered [17]. Moreover, by not having to use timeout mechanisms, they may be simpler to implement, which translates into a lower engineering effort. Nonetheless, a common drawback of such protocols is associated with their complexity and execution time, which are usually higher than the ones of partially synchronous systems.

2.3.5 Initial asynchronous BFT protocols

The first asynchronous BFT protocols were proposed by Rabin [18] and by Ben-Or [19], in 1983. Both protocols applied randomization techniques, using a coin-tossing mechanism, to solve the binary agreement problem, where nodes can vote with 0 or 1. Rabin's protocol requires $n \geq 3f + 1$ nodes, where f is the number of faulty nodes. It uses a voting system in which nodes send their votes to all other nodes. If at least $2f + 1$ votes received are equal, a process would decide on this quorum's vote. Otherwise, a shared coin tossing would determine the decided value.

Ben-Or protocol, on the other hand, required $n \geq 5f + 1$ processes and relied on local coin tossing, which means that the n-th coin tossed may have different values for different processes. It also starts with a broadcast by each node, sending their proposals to all other processes. After $n - f$ messages were received, if at least $(n + f)/2$ messages proposed the same value v, then v would be the new proposal and would be, again, broadcasted. Otherwise, the node would broadcast a message with an empty proposal. After the second broadcast, after $n - f$ messages are received if at least $(n + f)/2$ messages contained the same value, this value would be decided on. Otherwise, a coin would be tossed locally in order to determine the value to be decided as 0 or 1. Though this initial generation of protocols isn't particularly efficient, their models would, later, serve as inspiration for more practical asynchronous protocols.

2.3.6 HoneyBadgerBFT

Following this model, HoneyBadgerBFT (HBBFT) arose in 2017 [8] claiming to be the first practical

asynchronous BFT protocol, with cubic communication complexity. The protocol consists of 3 phases. In the first one, each node randomly selects B transactions (where B is the batch size, used for scalability efficiency) from its buffer (with client transaction requests) and encrypts them with a common public key. Next, each node passes its encrypted batch as input to an Asynchronous Common Subset (ACS) round which will return a set with the approved encrypted transactions. Lastly, each node, for each transaction that is outputted by ACS, decrypts a share of it, multicasts it, waits until $f + 1$ shares are received, allowing it to decrypt the whole transaction, and removes the approved ones from its buffer. The protocol proceeds with these 3 phases (or a round) repeatedly, delivering transactions at the end of each round.

The batch picked by each replica is composed of randomly selected transactions in order to minimize a common subset of transactions proposed by different nodes. The protocol makes use of two subprotocols: ACS and Threshold encryption. Threshold encryption is a technique that allows any party to encrypt a message to a master public key, and reconstruct the message with $f + 1$ decrypted shares, each with a different key share, similar to threshold signatures explained in section 2.2. Threshold encryption is fundamental in order not to reveal a transaction that is being proposed in the ACS until it's finally accepted, otherwise, an adversarial network scheduler could prevent a targetted transaction from ever being performed.

ACS is composed of two stages. The first is a reliable broadcast, in which the nodes exchange their proposals. After a node receives $N - f$ proposals, it inputs 1 for the asynchronous binary agreement (ABA) protocol instances related to the nodes for which it received proposals and 0 for the others. ABA is a subprotocol by which the nodes agree on the value of a single bit. Once every ABA has been completed, a node gets the indexes of the ABAs which delivered 1 as the answer and outputs them. ABA uses the threshold signature scheme with a common coin. The common coin protocol is similar to the signature reconstruction of the distributed Ethereum validator. Every node signs a common coin sid, or name, with its key share and multicasts it. When a node receives at least $f + 1$ shares, it tries to combine them and form a valid signature, validating it with the common public key. After that, all nodes have common data that can be used to generate the common coin values.

2.4 Alea-BFT

To address the problem of cubic message and communication complexity, Alea-BFT [5] came out as the first protocol asynchronous protocol to achieve quadratic message and communication complexity showing great scalability potential for larger networks.

Alea-BFT has a two-stage pipeline design with an appropriate interface for state machine replication. The first is a broadcast phase in which nodes exchange received messages from clients, and the second,

occurring in parallel, is an agreement phase, composed of rounds during which leaders may add their values to the state. With this simple design, Alea-BFT overcomes several scalability hurdles of prior protocols, namely:

- Cubic message and communication complexity of ACS based protocols.
- The overhead of running multiple ABA instances until delivering a message.
- The unnecessary bandwidth usage when a value is broadcasted several times until it is delivered.
- Extra communication step for decrypting shares due to threshold encryption.

Alea-BFT provides optimal resilience for the byzantine model, tolerating up to $f = \lfloor \frac{n-1}{3} \rfloor$ byzantine or crash faults where n is the number of nodes. Safety is ensured as in the traditional Byzantine model, and the only requirements for liveness are that the messages exchanged will not be modified, which can be accomplished by message authentication, and are eventually delivered. Formally, Alea-BFT satisfies the following properties:

- **Validity:** If a correct process broadcasts m, then some process eventually delivers m.
- **Agreement:** If a correct process delivers m, then every correct process delivers m.
- **Integrity:** A message m appears at most once in the delivery sequence of any correct process.
- **Total Order:** If two correct processes deliver two messages m_1 and m_2, then both processes deliver m_1 and m_2 in the same order.

As said before, Alea-BFT consists of two stages that occur in parallel, as shown in algorithm **??**. The first stage is based on the Verifiable Consistent Broadcast Protocol (VCBC), a protocol that allows a sender to broadcast a message to all nodes with a proof that a quorum of processes received the value. The second stage is based on the Asynchronous Binary Agreement (ABA) protocol, also part of HoneyBadger's ACS, which allows correct processes to agree on the value of a single bit.

Algorithm 2.1: Alea-BFT Initialization (P_i) extracted from [5]

constants:
  N
  f
state variables:
  $S_i \leftarrow 0$
  $queues_i \leftarrow 0$
procedure START
  $queues_i[x] \leftarrow$ **new** $pQueue()$, $\forall x \in \{0, \ldots, N-1\}$
  async BC-START()
  async AC-START()

For the broadcast component (algorithm 2.2), each process starts with a queue, containing the VCBCs that were received from each node, and receives messages from clients. Those messages are stored in a buffer. When the buffer reaches a specific size – the batch size – the node broadcasts this batch with an incremental priority value through the VCBC protocol. Every other node, thus, will store this batch with its priority value in the queue allocated for the ID of the sender.

Algorithm 2.2: Alea-BFT Broadcast Component (P_i) extracted from [5]

constants:
  B
state variables:
  buf_i
  $priority_i$
procedure BC-START
  $buf_i \leftarrow \emptyset$
  $priority_i \leftarrow 0$
upon receiving a message m, from a client **do**
  if $m \notin S_i$ **then**
    $buf_i \leftarrow buf_i \cup \{m\}$
    if $|\text{buf}_i| = B$ **then**
      input buf_i to VCBC$(i, \text{priority}_i)$
      $\text{buf}_i \leftarrow \emptyset$
      $\text{priority}_i \leftarrow \text{priority}_i + 1$
upon outputting m for VCBC$(j, \text{priority}_j)$ **do**
    $Q_j \leftarrow queues_i[j]$
    Q_j.Enqueue$(\text{priority}_j, m)$
    If $m \in S_i$ **then**
      Q_j.Dequeue(m)

In the startup of each node, another thread is launched to run in parallel and to perform the agreement phase (algorithm 2.3). A variable will keep track of the round number, which indicates the leader, and, in each round, each node will select the batch with the lowest priority from the queue of the leader and accept it or not (if there exists the batch locally or not), inputting its proposal to the ABA. The node, then, waits until the ABA delivers a result b. If b is 1, the batch was accepted and it is removed from the queue. If b is 0, nothing is done and the next round is started. This batch will then later be revisited when its sender becomes the leader again.

Algorithm 2.3: Alea-BFT Main Agreement Component (P_i) extracted from [5]

state variables:
  r_i
procedure AC-START
  $r_i \leftarrow 0$
  while true do
    $Q \leftarrow \text{queues}_i[F(r_i)]$
    $value \leftarrow Q.\text{Peek}()$
    $proposal \leftarrow value \neq \perp \ ? \ 1 : 0$
    input $proposal$ to ABA(r_i)
    wait until ABA(r_i) **delivers** b **then**
      if $b = 1$ **then**
        if $Q.\text{Peek}() = \perp$ **then**
          broadcast $\langle$FILL-GAP, Q.id, Q.head$\rangle$
        wait until $(value \leftarrow Q.\text{Peek}()) \neq \perp$ **then**
          AC-DELIVER$(value)$
    $r_i \leftarrow r_i + 1$

It can occur that the ABA instance decides for 1 but a correct replica voted 0. In this scenario, the replica does not have the batch delivered. To solve this problem, there is a recovery mechanism that can be used to obtain the batch from another correct replica.

Next, we explain the subprotocols used by Alea-BFT.

2.4.1 Verifiable Consistent Broadcast Protocol (VCBC)

VCBC, depicted in figure 2.9, allows a node to send a message to all other nodes and, later, broadcast a proof that at least a quorum of nodes received the message by collecting signatures from these nodes. The protocol consists of three message types:

- A *VCBC Send* message from the sender to all other nodes with the message.
- A *VCBC Ready* message from all nodes to the author of the *VCBC Send* with a signature over the message's hash.
- A *VCBC Final* message from the sender to all nodes with a proof created after it received a quorum of signatures. These signatures can be aggregated into a single one if a cryptographic scheme such as BLS is used.

VCBC ensures the following properties:

- **Validity:** If a correct process broadcasts m, then all correct processes eventually deliver m.
- **Consistency:** If a correct process delivers m_1 and another process delivers m_2, then $m_1 = m_2$.
- **Integrity:** Every correct party delivers at most one message. Additionally, if the sender is correct, then the message was previously broadcasted by it.

- **Verifiability:** If a correct party delivers a message m, then it can produce a single message M that it may send to other parties such that any correct party that receives M can safely deliver m.
- **Succinctness:** The size of the proof is independent of the length of the message.

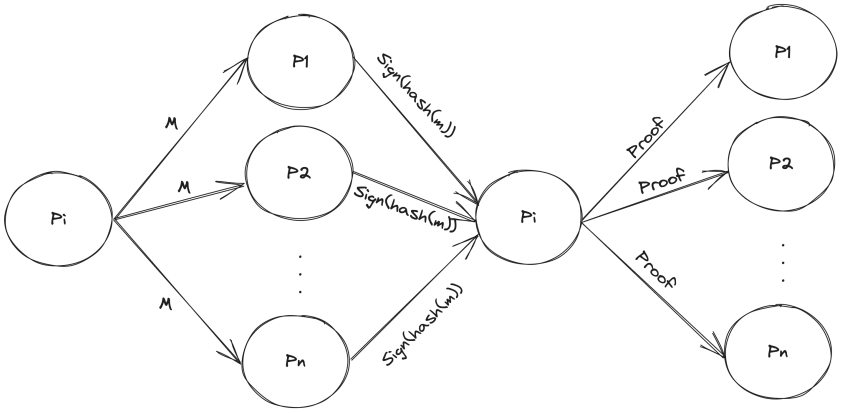


Figure 2.9: VCBC protocol.

2.4.2 Asynchronous Binary Agreement (ABA)

The ABA protocol allows for a set of correct processes to agree on a common bit value. The protocol runs in rounds until termination. In each round, every process proposes a bit. The protocol consists of four message types:

- An *ABA Init* message by which a process proposes a bit. If a node receives a weak support, $f + 1$ where f is the maximum number of faults, of *ABA Init* for the bit value b, it also sends a *ABA Init* with b if never sent.
- An *ABA Aux* message with a single bit value sent after a strong support, $\lfloor \frac{N+f}{2} \rfloor + 1$ where N is the number of replicas and f the maximum number of faults, of *ABA Init* is received. The weak support rule for *ABA Aux* is similar to the *ABA Init* rule.
- An *ABA Conf* message with 0, 1, or both after a strong support of *ABA Aux* is received with values for which a strong quorum of *ABA Init* was received.
- An *ABA Finish* message with 0 or 1. The weak support rule for *ABA Finish* is similar to the *ABA Init* rule. A node may send an *ABA Finish* if it receives a quorum of *ABA Conf* with values for which a strong quorum of *ABA Init* was received. If this list contains just a single bit and it's equal to the round's coin value, then it can send the *ABA Finish*. Otherwise, it jumps to the next round.

ABA ensures the following properties:

Table 2.1: Alea-BFT components complexity.

Component	Message	Communication	Time
Broadcast	$O(N)$	$O(N(\|m\| + \lambda))$	$O(1)$
Agreement	$O(\sigma N^2)$	$O(\sigma \lambda N^2)$	$O(\sigma)$

- **Agreement:** If a correct process decides b_1 and another correct process decides b_2, then $b_1 = b_2$.
- **Termination:** The probability of deciding after r rounds approaches zero as $r \to \infty$.
- **Validity:** If all correct processes propose b, then any correct process that decides must decide b.

2.4.3 Complexity

The message, communication, and time complexities for the Broadcast and Agreement components are shown in the table 2.1, where N is the number of processes, m is the application message size, λ is the signature size and σ is the expected number of ABA rounds. In the Alea-BFT paper [5], it's shown that $\sigma \to 1$. The recovery mechanism also has a quadratic message and communication complexity in N and constant time complexity.

3

Design

Contents

This chapter is dedicated to the specification of the validator network in which Alea-BFT will be implemented. In section 3.1, we will detail the layers that compose a specific implementation of the validator network, namely the Secret Shared Validators (SSV) implementation. In section 3.2, the consensus interface module will be highlighted and, in 3.3, we'll delve into the adaptations necessary to include Alea-BFT. Finally, section 3.4 will explore possible optimizations for the Alea-BFT protocol, taking the SSV context into consideration, and section 3.5 will discuss optimizations related to cryptography efficient usage.

3.1 SSV Architecture

The SSV architecture is built upon several modules necessary for the operator's functioning. First of all, it must discover peers that belong to the SSV network, which is done through a network discovery module. For communication, another P2P interface is defined with methods that allow the operator to subscribe and unsubscribe to subnets and broadcast messages in these subnets. The P2P communication layer is used to exchange messages for the SSV protocol. The messages can be related to the consensus protocol or to partial signature sharing, used in the pre-consensus and post-consensus phases. One important consideration very relevant to us is the fact that this communication layer doesn't allow a peer to send a message exclusively to another peer, it only allows broadcasting. This suits well the adopted QBFT protocol since all messages are broadcasted, but it's actually a negative factor for Alea-BFT since the *VCBC Ready* message could only be sent to the VCBC author. Thus, for the Alea-BFT case, this design represents unnecessary bandwidth usage and more congestion in the network.

Moreover, the operator needs a gateway to the Ethereum network. This is accomplished by the connection with a Beacon client service. From this client, the operator can keep track of the current slot and can get status information regarding the Ethereum blockchain, it can request data objects for validator duties, and submit signed objects to be broadcast to the entire blockchain network.

There's also a contract layer used to manage the relationship between the SSV network participants. The contract layer is implemented in Solidity which is used for implementing smart contracts in Ethereum. It allows operators to join the SSV network and Ethereum validators to hire operators.

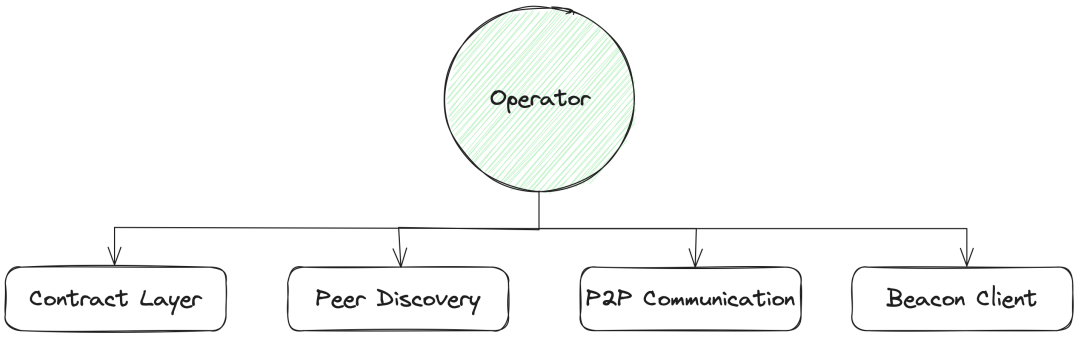


Figure 3.1: Operator modules.

3.2 The role and interface of consensus in SSV

A committee of operators, that runs a decentralized validator, does consensus when one of its validators has a duty. The path until the start of the consensus begins with the Beacon client which sends a signal to the operator whenever one of its validators has a duty for the current slot. Once this duty

starts, the committee of operators that represents the validator starts a pre-consensus phase, if required, and then launches a consensus instance. Notice how consensus is used as a standalone instance, instead of a replicated state machine which would be more aligned with the abstraction offered by Alea-BFT. This design was conceived in that way due to the fact that an operator may participate in several different duties in the same slot and each duty is executed by different committees. If a single consensus execution was performed for the entire network, the necessary number of participating nodes could slow down the protocol execution. Moreover, the probability of termination for each duty would be correlated, so a faulty leader or network instability in a fraction of the network could affect every duty termination.

Since an operator may need to handle several different consensus for its validators, each validator is associated with a *ValidatorController* structure that manages the validator state and duties. For each duty type, the validator controller has a unique *DutyRunner* object that executes the duty properly, following the steps mentioned in section 2.2.5. Each *DutyRunner* holds a *QBFTController* which has an *Instance* object that represents the consensus instance and implements the protocol.

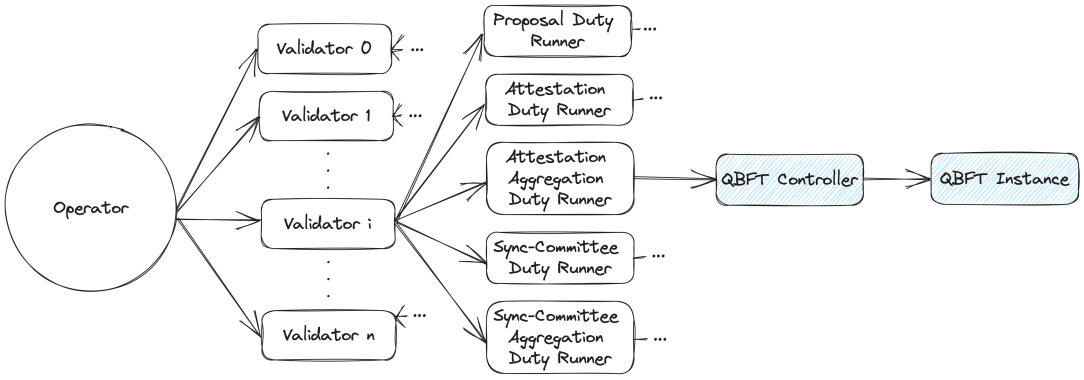


Figure 3.2: SSV structures.

The *Instance* module provides an interface that accounts for a start-up method and a method to process incoming messages. All other modules functionalities are abstracted from the *Instance* module which receives function objects to perform necessary steps such as broadcasting a message and creating and verifying signatures, for instance. The function for processing messages returns information regarding the consensus state such as if it has been decided or not.

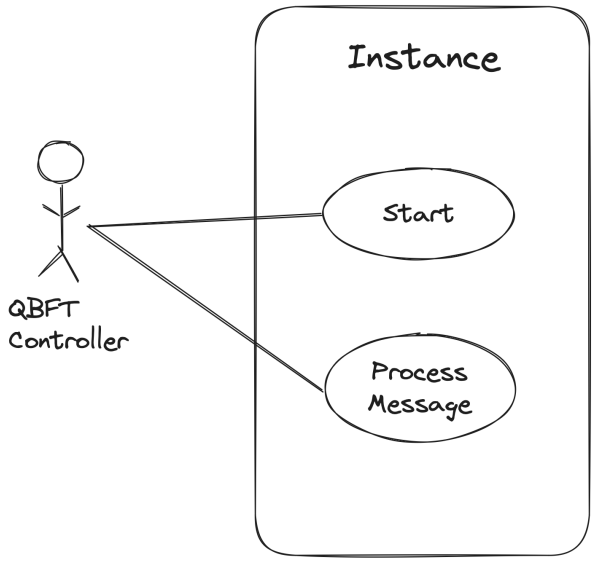


Figure 3.3: *Instance* use cases.

3.3 Adapting Alea-BFT to one-shot consensus

Due to the standalone consensus instance design, we have to launch Alea-BFT to decide only on a single value and then terminate. Further than that, the value to be decided is just a single object, e.g. a block to be proposed, and not a list of objects, and, thus, batching is no longer applicable. An advantage, however, is that each participant will only propose one value and thus we can drop the priority tag in VCBC messages, which will allow some optimization as we will see in section 3.4.

In the Alea-BFT specification, a replica could actively receive messages from a client. In the SSV context, the single unique value to be ever received is provided in the start-up procedure (algorithm 3.1). Therefore, once the protocol starts, we can initiate a VCBC for our input value and we don't need to launch any independent thread to handle client messages. Moreover, in Alea-BFT's specification, the agreement component runs in a separate thread. Since an operator may run numerous consensus instances at the same time, launching threads for each could cause a processing overhead. A better solution is to refactor the agreement loop as a procedure to be called in the instance's start-up and in the termination of an ABA. Thus, the instance's start-up method can call the procedure to launch the first agreement round and, once ABA for round r finishes, it calls a procedure to launch the agreement for round $r + 1$.

Even though the one-shot consensus design presents some obstacles and requires some adaptions in the Alea-BFT protocol, in the following section we propose some optimizations that can improve the protocol performance and balance with such obstacles.

Algorithm 3.1: Alea-BFT Initialization (P_i) adapted for one-shot consensus

procedure START
Input: $value$

$queues_i[x] \leftarrow$ **new** $pQueue()$, $\forall x \in \{0, \ldots, N-1\}$
input $value$ to VCBC(i,0)
AC-START()

3.4 Protocol optimizations for Alea-BFT

As stated in section 3.3, one of the major drawbacks of the SSV design is the fact that consensus is used as a standalone instance and not as a state machine replication. Even though this is not the ideal scenario for Alea-BFT, we can take some properties of this specific context to seek out new optimization and adjustment for the protocol. Next, we provide some context-free optimizations and some specially designed to better fit the protocol to the SSV case.

3.4.1 Fast ABA (FA)

Our first optimization is context-free, in other words, it can be applied to any project that uses Alea-BFT. Experimental results [5] showed that the number of ABA rounds tends to the optimal value of 1. This tells us that, most of the time, at least a quorum of participants agree on the first vote of the protocol. This is reasonable because ABA takes longer than VCBC and, thus, as the protocol runs, the VCBC queues will be filled and most participants will vote with 1 in the ABA, if the round leader has been proposing values, or 0, otherwise. This likely agreement on the first vote supplies us with a spot to investigate and look for optimizations.

One of the properties of the ABA is *validity*, in other words, if all correct processes propose b, then any correct process that decides must decide b. Therefore, if we receive the first vote of every process and if they are all the same, we can know in advance the value that will be decided because at least a quorum of correct processes voted equally. Indeed, note that even if f malicious participants broadcast 1 to some nodes and 0 to others, if all the correct nodes voted equally, then the malicious processes won't be able to reach $f+1$ *ABA Init* votes for the opposite bit and, thus, it will never be in the *ABA Conf* values to be decided.

Nonetheless, even if a process knows the output in advance, it needs to continue running the protocol until the protocol terminates because other honest peers may not have the same view of the system's state. But, to accelerate termination, the process that knows the output can already send an *ABA Finish* message with the decided bit in advance.

Sending the *ABA Finish* message at the beginning of the ABA protocol, allows us to explore another improvement. Usually, the coin-sharing protocol would be initialized jointly with an ABA instance, even

though the coin is only used during the *ABA Conf* phase. By sending early *ABA Finish* messages, there's the possibility that none of the nodes would need to process any *ABA Conf* message in case everyone receives a quorum of *ABA Finish* messages before reaching the *ABA Conf* phase. Therefore, we can postpone the coin-sharing protocol to only when it's indeed necessary. We can launch it at any ABA step that occurs before receiving a quorum of *ABA Conf*. If we reach a quorum of *ABA Conf* messages and the coin-sharing protocol hasn't concluded yet, we just wait until it does, as it was done previously.

3.4.2 First ABA Delay (AD)

As experimentally shown in [5], the number of ABA rounds tends to 1 as the number of agreement rounds increases. This tells us that the early ABAs have less probability of deciding in one round. This is expected because, on start-up, the VCBC queues are empty and, even though the VCBC protocol is faster than ABA, as the first ABAs start, some processes may have already received the first VCBC from the leader and some may not.

To address this issue, we delay the start of the first ABA to increase its probability of terminating in one round (algorithms 3.2 and 3.3). This is particularly important for our case because the total protocol time will be defined by the first ABA that terminates and decides 1, so we require that the chance of the deciding 1 in the first ABA is as high as possible. Instead of a time delay, which would be more natural and common, we can wait until a threshold number of VCBCs are completed (valid *VCBC Finals* received). Here we have a trade-off, the more VCBCs we wait, the higher the chance of terminating in one ABA but the longer the wait time. The value used was a quorum, $(\lfloor \frac{n+f}{2} \rfloor + 1)$, of VCBCs, which represents the highest possible threshold value. This is due to the fact that ABA is an expensive protocol and it's worth waiting if we are increasing the probability of delivering on the first instance.

Algorithm 3.2: Alea-BFT Initialization (P_i) adapted for ABA delay

procedure START
Input: $value$

$queues_i[x] \leftarrow$ **new** $pQueue()$, $\forall x \in \{0, \ldots, N-1\}$
input $value$ to VCBC(i,0)
~~AC-START()~~ ▷ delayed

Algorithm 3.3: Alea-BFT Broadcast Component (P_i) adapted for ABA delay

state variables:
  $ac\_started \leftarrow$ False
upon outputting m for VCBC(j, priority$_j$) **do**
  $Q_j \leftarrow queues_i[j]$
  Q_j.Enqueue(priority$_j, m$)
  ▷ Delay condition
  If $ac\_started \neq$ True **then**
    $number\_of\_vcbcs \leftarrow 0$
    For x $\in$ {0, ..., N-1}
      If $queues_i[x] \neq \emptyset$ **then**
        $number\_of\_vcbcs \leftarrow number\_of\_vcbcs + 1$
    EndFor
    If $number\_of\_vcbcs \geq quorum$ **then**
      async AC-START()
      $ac\_started \leftarrow$ True
  If $m \in S_i$ **then**
    Q_j.Dequeue(m)

3.4.3 Complete VCBC View (CV)

As stated previously, the standalone consensus instance is a drawback for Alea-BFT. However, this provides us with the following: each participant will start a VCBC only once (with its consensus proposal value). To handle the case in which a malicious participant launches multiple VCBCs, we can add a restriction rule by which a node will not answer to a *VCBC Send* from the process p_i if it has already received another *VCBC Send* from p_i with different data.

But that's not all. In the distributed validator context, it's very likely that the operators share the same view of the blockchain and will propose the same value for the consensus. It wouldn't be the case, for instance, if a fork or a re-org happens in the blockchain, but this is extremely rare. Thus we have in hand a situation similar to the ABA first vote agreement. Indeed, if we receive VCBCs from all processes and if each VCBC has the same data, we can know in advance the value for which the consensus will decide (algorithm 3.4). It's impossible for a malicious operator to perform an attack in which it sends different VCBC results to different nodes. For that, the malicious operator would need to collect at least a quorums of *VCBC Readys* for two different *VCBC Sends* created by it, which is impossible due to our restriction rule.

With this optimization, we may know the decided value in advance. However, a node can not stop processing messages because other nodes may have different views of the network. But, similarly to the Fast ABA optimization, the node can accelerate termination by sending *ABA Finish* messages in advance.

Algorithm 3.4: Alea-BFT Broadcast Component (P_i) adapted for Complete VCBC View

state variables:
upon outputting m for VCBC(j, priority$_j$) **do**
    $Q_j \leftarrow queues_i[j]$
    Q_j.Enqueue(priority$_j, m$)

    $number\_of\_vcbcs \leftarrow 0$
    For x ∈{0, ..., N-1}
      If $queues_i[x] \neq \emptyset$ **then**
        $number\_of\_vcbcs \leftarrow number\_of\_vcbcs + 1$
    EndFor
    If $ac\_started \neq$ True $\wedge\ number\_of\_vcbcs \geq quorum$ **then**
      async AC-START()
      $ac\_started \leftarrow$ True

    ▷ Complete VCBC View Check condition
    If $number\_of\_vcbcs = N$ **then**
      If $AllEqual(queues_i)$ **then**
        AC-DELIVER($queues_i[i]$.head)

    If $m \in S_i$ **then**
      Q_j.Dequeue(m)

3.5 Cryptographic optimizations

One of the major bottlenecks of consensus protocols lies on costly cryptography functions [16, 20, 21]. Taking this into consideration, this section explores optimization proposals that focus on more efficient usage of cryptography primitives.

3.5.1 BLS Aggreation

The BLS scheme is known for its signature aggregation features. That is precisely, for instance, why Ethereum decided to use it as its main cryptographic scheme. It can aggregate any amount of signatures or public keys in a single value. More than its common application of reducing message sizes, this can be used to rapidly verify a batch of signatures. BLS allows one to verify an aggregated signature over different messages (regular aggregation), but the real power comes from verifying an aggregated signature over equal messages (fast aggregation). Fast aggregation allows one to verify any amount of signatures in a constant time (basically the time to verify a single signature), while regular aggregation can provide, approximately, 50% speed up [15].

With this feature, we can, for instance, fast aggregate and verify all the *VCBC Ready* valid messages since they are signed over the same data. Similarly, we can apply it to the *ABA Init*, *ABA Aux*, *ABA Conf* and *ABA Finish* messages to fast aggregate and verify weak and strong support batches of messages.

3.5.2 Message Authentication Codes

This optimization can be applied to Alea-BFT and any other protocol that doesn't require non-repudiation, which is not the case for some of the messages in QBFT. The PBFT paper [16] showed that great performance improvement can be achieved by replacing costly asymmetric key digital signatures with message authentication codes (MACs). To use MACs, we need to define symmetric keys between peers. This can be easily accomplished with the Elliptic-curve Diffie-Hellman (ECDH) key agreement protocol since each operator already has a BLS key pair and stores the public keys of all other operators.

Note, however, that we will still need to use BLS due to the VCBC's threshold signature usage. So, when a node receives a quorum of *VCBC Ready*, it still needs to aggregate the signatures and verify the aggregated signature before sending the *VCBC Final* message. In the same way, when a node receives a *VCBC Final* message, it needs to verify the aggregated signature. For all other message types, we can use MACs for message authentication.

3.5.3 Other asymmetric schemes

Since BLS verification and signing primitives are way more expensive than other asymmetric cryptographic schemes' primitives, as it will be shown in section 5.4, we could replace the entire BLS usage with another scheme. For instance, we could implement RSA or, more modern, ECDSA and EdDSA. The only careful adaptation we would need to make would be to include, in the *VCBC Final* message, a buffer of signatures with their respective signers instead of the single aggregated BLS signature.

4

Implementation

Contents

In this chapter, we go over the implementation details of the integration of Alea-BFT into the SSV infrastructure. Section 4.1 explains how each SSV module works and section 4.2 goes into specific details of the Alea-BFT module implementation. Lastly, section 4.3 explains why a private Ethereum network was used and how it was created.

4.1 SSV modules implementation

The SSV code is written in the Go programming language and the implementation consists of 73,000 lines of code. The main structure is the *OperatorNode*. As seen in section 3.1, it has a discovery layer, a communication layer, and a connection with an Ethereum consensus client, the Beacon client. The Beacon client can use Prysm, Lighthouse, Lodestar, Nimbus, or Teku, or even an independent

implementation. Overall, it has to follow the API interface defined by Ethereum in the *go-eth2-client* repository[1].

For the peer discovery layer, the operators use the Discv5 protocol created by Ethereum, which uses the Kademlia distributed hash table (DHT). The DHT requires a bootstrap node which is responsible for handling connections with new peers and sharing with them information about all connected peers. SSV provides some public bootstrap nodes for different Ethereum chains, such as the Mainnet and the Goerli testing network.

As the communication layer, the operators use the PubSub protocol of Libp2p, which is also used by Ethereum for its p2p network. Basically, PubSub provides an interface for a node to subscribe and unsubscribe to different topics and to publish messages on a specific topic. By subscribing to a topic, the operator can listen to all messages being published in it. In PubSub's initial version, FloodSub, a peer would need to hold stable connections with all other peers for each subscribed topic and, when it wanted to publish a message, it would send it directly to each peer. This however required the peer to handle a significant amount of connections which would be impossible to do in a big network, such as the Ethereum Mainnet.

Then, in PubSub's later version, GossipSub, a peer would be connected to only a small fraction of the network, its *mesh* list. Now, to publish a message, it needs to send it only to the peers in its *mesh*. When receiving a message never seen before, a peer propagates it to other peers in its *mesh*, allowing the message to propagate through the whole network. There's also a gossiping mechanism by which peers periodically send metadata about messages they have received to random peers. If a received metadata was never seen before, it can request the message from the peer that gossiped the metadata. This mechanism allows a peer to recover a message in case it has missed it.

The GossipSub protocol also provides defenses against Sybil and Eclipse attacks by using mechanisms such as a peer scoring system and the maintenance of a *mesh* with good-scoring peers. As stated in section 3.1, note that the interface provided by PubSub allows only the broadcasting of a message. In other words, it doesn't allow a peer to send a message exclusively to another peer, which would be beneficial for the VCBC protocol.

4.2 Alea-BFT module in the SSV code

As stated in section 3.2, the operator holds a *ValidatorController* structure for each of its validators. Each *ValidatorController* holds a *DutyRunner* for each duty type, which handles a *QBFTController* that manages an *Instance* object that represents the consensus instance.

Given this design, we need only to adapt the *QBFTController* and the *Instance* structures. We

[1](2023, October) Attestantio: go-eth2-client. GitHub repository. Accessed 20-October-2023. [Online]. Available: https://github.com/attestantio/go-eth2-client

decided to keep the QBFT structures untouched and we defined totally new controller and consensus instance structures for Alea-BFT, which comprised 5,000 lines of Go code. To allow Alea-BFT to run, we needed to change just a single line of the SSV code, namely an *import* line in the *DutyRunner* implementation. With this import redirection, we can switch between consensus protocols with minimal change. If we wanted to swap between protocols during live execution, we could simply add a new *import* line, set a flag option to indicate which protocol to use, and create an API to modify the flag in real-time. This, however, could be more complex on the protocol level due to the necessity of a recovery mechanism.

For the Alea-BFT's *Instance* structure, we defined a *VCBCState* structure to manage VCBCs received from other peers and *VCBC Ready* messages received for its own VCBC. An agreement component state structure, *ACState*, was created to manage the execution of the ABA instances. To select the leader of each ABA, a round-robin function was implemented taking as input the agreement round number and the duty's slot value. Also, a *CommonCoin* structure was defined for the ABA's shared coin feature. This structure handles the creation of the partial signature for the coin's seed and generates the coin value for agreement and ABA round pair.

At last, we had to define new message types to support the VCBC and ABA protocols, as shown in code 4.1. The messages defined are nested into a *SignedMessage* structure which contains fields such as the signature, the operator identification, the validator identification, and the duty type.

Listing 4.1: Alea-BFT messages definition

```
type VCBCSendData struct {
  // There's no priority because an operator can perform only one VCBC
  Data    []byte
}

type VCBCReadyData struct {
  Hash    []byte
  Author  types.OperatorID
}

type VCBCFinalData struct {
  Hash         []byte
  AggregatedMessage *SignedMessage
}

type ABAInitData struct {
```

```
17   ACRound alea.ACRound // Agreement component round
18   Round   alea.Round // ABA round
19   Vote    byte
20 }
21
22 type ABAAuxData struct {
23   ACRound alea.ACRound
24   Round   alea.Round
25   Vote    byte
26 }
27
28 type ABAConfData struct {
29   ACRound alea.ACRound
30   Round   alea.Round
31   Votes   []byte
32 }
33
34 type ABAFinishData struct {
35   ACRound alea.ACRound
36   Vote    byte
37 }
```

4.3 Private Ethereum Testnet

To perform experiments, one could use an existing Ethereum testing network, such as the Goerli network. However, to participate in the network, one must first fully synchronize to the network's current state, which can be very costly due to the data sizes of these networks. For instance, for the Ethereum Mainnet, it would be necessary to synchronize over 1286 GB[2]. Moreover, we wouldn't have any type of control regarding the type and amount of duties to be executed, which makes the setup of the experiment very difficult.

A better solution is to create a private Ethereum testing network and its associated Beacon client. First of all, this removes the obstacle regarding data synchronization since the blockchain can be bootstrapped as many times as necessary. Moreover, the network parameters can be customized in order to create the required environment setup for each experimental test. For instance, by controlling the number of validators in the network, we can force all validators to perform sync-committee duties for

[2](2023, October) Ethereum Chain Full Sync Data Size. Accessed 20-October-2023. [Online]. Available: https://ycharts.com/indicators/ethereumchainfullsyncdatasize

every slot. By fixing a single committee of SSV operators and controlling the number of SSV validators, we can set the desired system load for the experimental test.

Therefore, we followed the second approach, creating a private customizable Ethereum network. As an initial setup, 241 validators were defined, out of which 1 belonged to the SSV network. Since Ethereum's PoS attempts to request 512 validators to perform a sync committee duty for every slot, by this setup we could force all validators to have at least one duty in every slot. To accomplish more duties per slot, we can either increase the number of SSV validators or hard code the committee of operators to do mock sync committee duties, if necessary.

5

Evaluation

Contents

This chapter provides an analysis of the performance of the Alea-BFT protocol in the SSV network context. Our focus was to answer the following questions:

- How does Alea-BFT compare to QBFT, in local networks, both in latency and throughput, for different system loads?
- How do optimizations improve Alea-BFT's performance?
- What are the most important execution bottlenecks?
- How does Alea-BFT compare to QBFT in wide-area experiments?

- When the number of replicas increases, do the implementation of both protocols follow the expected quadratic behavior?
- How each protocol is affected in the presence of node faults or network problems?

The chapter begins with section 5.1 describing the experimental setup used in the experiments. In section 5.2, Alea-BFT is compared against QBFT in a local area network considering the different protocol optimizations discussed in section 3.4. Then, in section 5.3, we analyze and dissect in detail the results for the performance of Alea-BFT. Section 5.4 highlights the biggest processing bottleneck of the implementation of the Alea-BFT protocol and explores improvements to overcome this performance hurdle. Then, we compare Alea-BFT with QBFT for wide area simulations in section 5.5. Section 5.6 analyzes how the protocol latency changes for different network sizes and, lastly, section 5.7 compares the resilience of the Alea-BFT protocol against QBFT under faulty scenarios.

5.1 Experimental setup

Except for the experiment with different network sizes, overall, all tests deployed 4 SSV operators, the current standard value used by the company. Each operator was launched in a virtual machine running Ubuntu 20.04 with 2 Intel(R) Xeon(R) Gold 5320 CPU @ 2.2GHz processors with 26 cores and 64GB of RAM, though each virtual machine was limited to 4 vCPUs. The different machines could communicate through a local network. For the wide area experiment, to simulate a higher transmission delay, we used the *tc qdisc* command available in Linux. The private Ethereum beacon clients were also deployed in similar machines, as well as an independent bootstrap node used for the discovery layer.

The metrics used to compare the protocols were latency and throughput. Base latency is the time required by the consensus protocol to terminate one instance, in our case to agree on one duty. For experiments in which a higher system load was used, where we used 2 or more duties per slot, the latency metric was calculated as the average of the latencies for each duty.

The throughput was calculated as the number of duties that could be decided on a single slot, instead of a rate of executions per second. This decision was made because validators are expected to broadcast their duty outputs during the slot duration interval. For example, for the attestation duty, if the validator broadcasts its attestation only after the first slot, its reward is significantly reduced. For other duties, the validator's output won't even be processed and it would be penalized.

Since our private Ethereum network allows us to impose a Sync Committee duty for every validator in every slot, this duty was the one used for all experiments. Moreover, a benefit of this duty is that its steps resemble those of the attestation duty, which is the most common duty type on real Ethereum networks. In the SSV implementation, the object to be decided on consensus, *ConsensusData*, for the Sync Committee duty, has a maximum size of 256KB. Also, for this duty, operators need to wait 4

seconds after the start of the slot, in order to receive the block of the previous slot. Thus, a committee has only 8 seconds to perform the duty.

For each experimental configuration, we conducted 20 observation samples and considered their mean value as the result. This approach was adopted to enhance the reliability and accuracy of the experiments' outputs. In some charts, data points may appear with an error bar which represents the standard deviation of the data samples.

5.2 Alea-BFT and QBFT performance comparison

The first basic question we attempted to answer was how the base latency performance of Alea-BFT compares to QBFT on a local network with 4 operators, and also the contribution of individual optimizations to the performance of Alea-BFT. For that, we launched an unchanged version of Alea-BFT. Then we added the Fast ABA (FA) optimization. To this already optimized version of Alea-BFT, we added the First ABA Delay (AD) and Complete VCBC View (CV) optimizations, respectively. Then we compared their base latency performance against the unchanged QBFT implementation from SSV. The base latency comparison can be seen in figure 5.1.

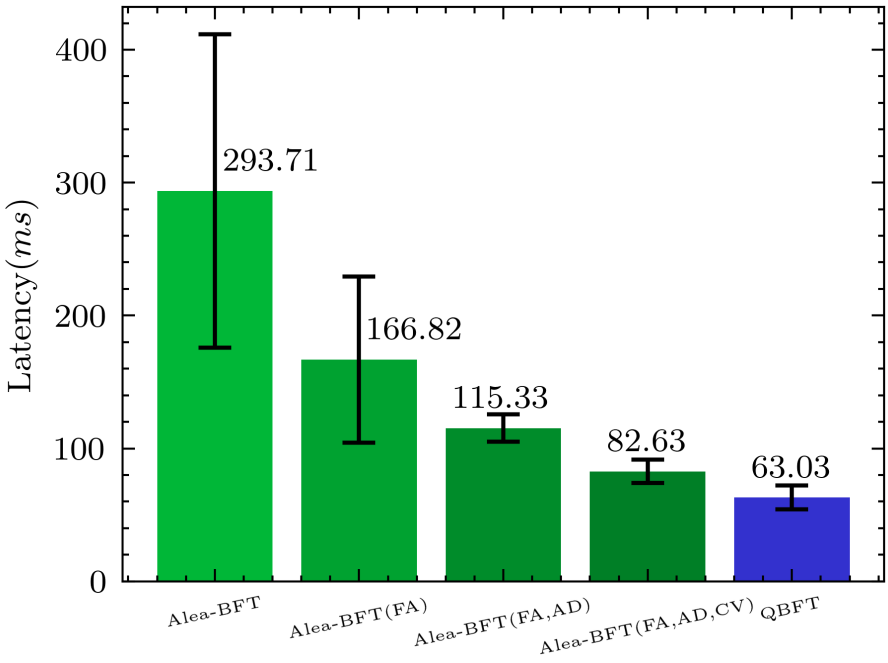


Figure 5.1: Base latency for QBFT and Alea-BFT with protocol optimizations.

The Fast ABA optimization (FA) provided, approximately, 43% speed-up against the original version. Then, the First ABA Delay (AD) optimization with the FA optimization provided a 31% speed-up against

using solely FA. Finally, adding the Complete VCBC View (CV) optimization added a speed-up of 29%. These results are summarized in table 5.1.

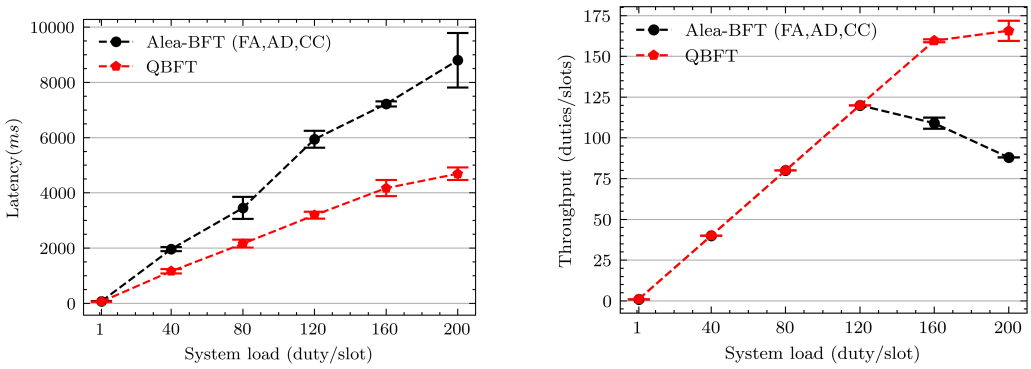
Table 5.1: Protocol optimizations speed-up.

	FA	**FA, AD**	**FA, AD, CV**
Speed-up against previous version	43%	31%	29%
Speed-up against original version	43%	61%	72%

The QBFT performance was superior to Alea-BFT with the FA, AD, and CV optimizations by 23%. Even though both algorithms have quadratic communication and message complexities, Alea-BFT exchanges more messages until termination, which can justify the longer latency. Notice that, since the experiment was launched in a local network, the message processing time dominates the message transmission time and, thus, the number of messages to be processed has a bigger weight in the final total base latency. This can be better observed in the breakdown section 5.3.

Figure 5.1 also shows a large standard deviation for the Alea-BFT unchanged version and FA version. This variation is directly related to the number of ABAs performed until termination. If the first leader's queue is empty upon the start of the first ABA, the operators will have to complete the whole ABA protocol in order to try to decide on a value in the next agreement round. In a lucky scenario, the first leader's VCBC propagates through the network before the first ABA starts allowing the operators to terminate the consensus in the first agreement round. The big time difference between these two scenarios and the unpredictability of a lucky or an unlucky scenario creates this big standard deviation. Note that accelerating the ABA execution with the Fast ABA optimization reduces the base latency but still carries a great standard deviation.

We, then, proceeded to compare the average latency between QBFT and Alea-BFT with all optimizations for increasing system load. Figure 5.2(a) shows the results for experiments with 1, 40, 80, 120, 160, and 200 duties per slot. The latency grows almost linearly as the system load increases due to the fact that the processing time dominates the message transmission delay and is proportional to the system load since there's no batching optimization. As expected by the base latency results, QBFT could keep its latency smaller than Alea-BFT due to the difference in the number of messages.



(a) Latency

(b) Throughput

Figure 5.2: Performance per system load for QBFT and Alea-BFT optimized.

Looking at the same set of experiments through the throughput metric perspective, presented in figure 5.2(b), we see that QBFT was able to decide more duties per slot for higher system loads. This was expected due to its smaller base latency and due to the higher number of messages that Alea-BFT has to process. The throughput of Alea-BFT decreases after the number of duties per slot reaches 120. This drop occurs because all duties run simultaneously, and the introduction of new duties overloads the CPU. Consequently, all duties take longer to complete, leading to the decline in throughput, instead of remaining stable at a certain value.

Even though this Alea-BFT version with protocol optimizations could not terminate 200 duties for the current slot duration in Ethereum, a load of 200 duties per slot is already very big in terms of necessary validators. Namely, from section 2.2.5, we can compute the expected number of duties per slot for a single validator, $E(d|V=1)$, by

$$E(d|V=1) = \sum_{i=1}^{5} E(d_i|V=1) = \sum_{i=1}^{5} P(d_i)$$

where d_i with $i = 1, ..., 5$ are the duty types. Considering a total of 700000 Ethereum validators, and committees of size 128, a validator has 3.598×10^{-2} expected duties per slot. If the events (duty assignments) were independent, the expected number of duties per slot for V validators would simply be $E(V)_{i.i.d} = V \times 3.598 \times 10^{-2}$. However, some events are independent and some are negatively correlated, and, thus, although $E(V)_{i.i.d}$ is not an actual estimation, it serves as an upper bound for the actual estimation.

So, for instance, to have an expectation of 200 duties per slot, the committee of operators would need to be responsible for at least $V \geq \frac{200}{3.598 \times 10^{-2}} \approx 5558$ validators, as can be seen in figure 5.3. Indeed, we conducted a Monte Carlo simulation of duty assignments through 1024 epochs and a control group of

5558 validators had, on average, 195 duties per slot.

Furthermore, regarding the current throughput peak of 120 duties per slot, in section 5.4, we will discuss more optimizations that will allow Alea-BFT to overcome this barrier.

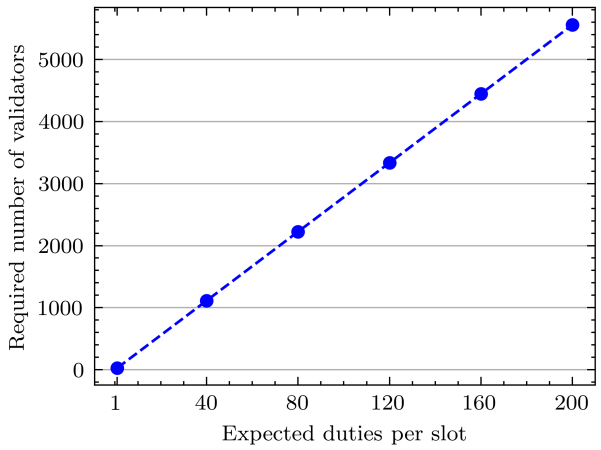


Figure 5.3: Required validators for duty per slot expectation.

5.3 Execution breakdown

Now, we attempt to provide a deep explanation of the latency breakdown of the optimized Alea-BFT. First of all, the Complete VCBC View optimization may cut out the ABA steps of the protocol, the start-up procedure and the processing of *VCBC Send*, *VCBC Ready*, and *VCBC Final* messages are left untouched and do always occur. Let's analyze each step one by one to understand where is the current bottleneck of the current implementation.

For each protocol step, we performed tests with the go benchmark framework. This framework attempts to provide a reliable benchmark for the processing time of a function by running it hundreds of times and tracking down the processing time profile. Note that the test function is executed in isolation and, thus, it doesn't dispute CPU with other operator modules. The results are presented in table 5.2. How these steps are stacked together such that the total time can be visualised is presented in figure 5.5(a) and will be discussed next.

Table 5.2: Benchmark latency of relevant Alea-BFT steps, considering the Complete VCBC View optimization, in milliseconds.

Start-up	*VCBC Send*	*VCBC Ready*	*VCBC Ready Quorum*	*VCBC Final*
0.93	3.18	2.32	3.5	2.7

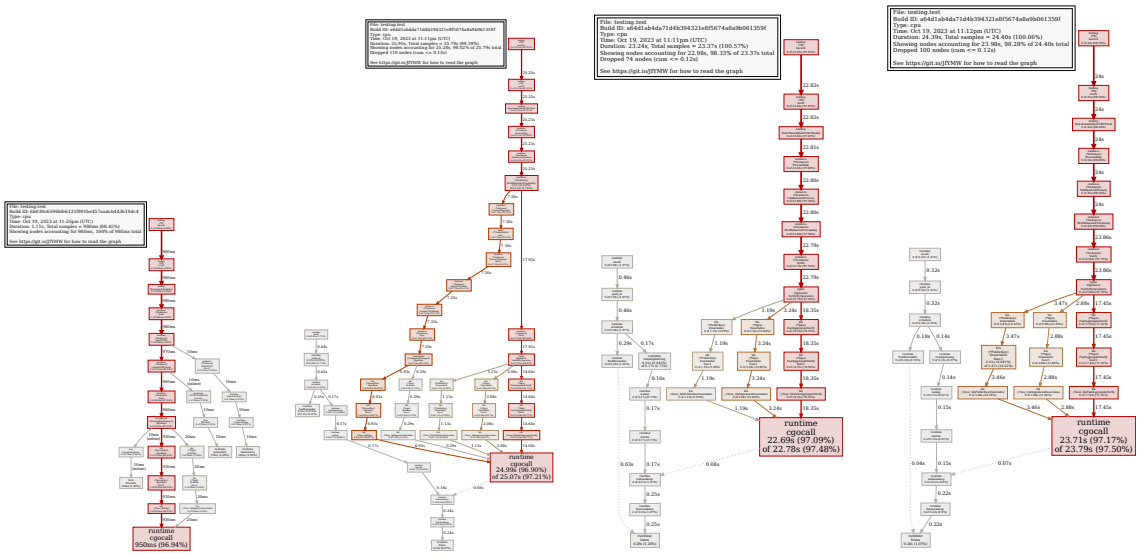
The start-up procedure takes approximately 0.93 milliseconds. This step consists only of some allocation of variables and the creation of a *VCBC Send* message. We produced a CPU profiling map of

this step using the *pprof* tool of Go (figure 5.4(a)). It can be seen that the signing process with the BLS key takes 97.96% of the total processing time.

Similarly to the start-up case, the *VCBC Send* processing step uses 96.90% of its time for BLS cryptography (figure 5.4(b)). This step, on the other hand, performs two cryptographic tasks by verifying the signature of the incoming message and signing a *VCBC Ready* message as a response. This explains why it takes, on average, 3.18 milliseconds.

The *VCBC Ready* step performs a signature verification and, if it's the $(\frac{(N+f)}{2}+1)^{\text{th}}$ (the quorum value for N replicas and f faults) ready to be received, it also performs a signature aggregation of all received *VCBC Ready* messages. That's why the *VCBC Ready* that completes the quorum takes 3.5 milliseconds while an earlier *VCBC Ready* takes, on average, 2.32 milliseconds. It can be seen in figure 5.4(c) that, for a non-completing quorum *VCBC Ready*, BLS verification takes 97.52% of the processing time.

The *VCBC Final* step performs a signature verification over the aggregated signature and, if it's the quorum completing the message, it will also call the ABA start-up, which will sign an *ABA Init* message. A non-completing quorum *VCBC Ready* takes 2.7 milliseconds from which 97.79% is due to BLS verification, according to figure 5.4(d).



(a) Start-up **(b)** *VCBC Send* **(c)** *VCBC Ready* **(d)** *VCBC Final*

Figure 5.4: Step profiling.

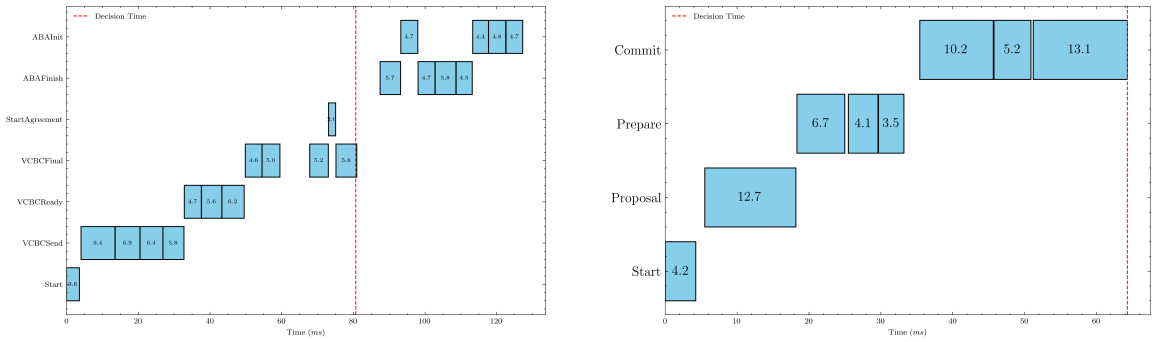
With the Complete VCBC View optimization, in a scenario of N honest nodes with good network connection, in order to terminate the consensus an operator needs to process at least 1 start-up procedure, N *VCBC Send*, a quorum of *VCBC Ready* and N *VCBC Final* messages. For $N = 4$ operators,

this represents a total of

$$0.93 + (0.93) + (4 * 3.18) + (2 * 2.32 + 3.5) + (4 * 2.7) = 32.59$$

milliseconds at minimum. Therefore, for this current implementation, independently of the effectiveness of the communication layer, $32.59\ ms$ is a lower bound for the base latency for 4 operators. Of course, these are just boundary values and they are not reached due to the parallel execution of other SSV layers and to other modules of the operator. In section 5.4, improvement to these boundary values is reached by using efficient cryptographic optimizations.

Another useful visualization of the base latency is presented as a Gantt chart in figure 5.5. The processing times of Alea-BFT's steps are different from the benchmark results which is expected since the benchmark test isolates the function processing while, during real execution, other operator modules work in parallel to the Alea-BFT protocol, inevitably slowing it down. Comparing it to the QBFT breakdown, shown in figure 5.5, we can see that, in fact, Alea-BFT needs to process more messages until termination, even having to process messages after knowing the consensus decision value in advance.



(a) Optimized Alea-BFT **(b)** QBFT

Figure 5.5: Execution breakdown.

5.4 Improving the cryptography bottleneck

As discussed in section 3.5 and noticed by the experimental results shown in section 5.3, cryptography is a significant bottleneck in terms of message processing, taking more than 90% of the processing time. BLS, particularly, has very expensive primitives when compared to other schemes, which can be visualized in figure 5.6. For instance, according to figure 5.6, other asymmetric schemes such as RSA and EDDSA can verify signatures 4519% and 3172% faster than BLS, respectively. The ideal scenario would be to change all asymmetric message authentication to symmetric message authentication, using

Message Authentication Codes (MAC) which can be as fast as $1.1\mu s$, for the HMAC implementation, as can be seen in figure 5.6. However, as discussed in section 3.5, VCBC requires asymmetric schemes, and thus our best option is to combine MACs with BLS as described in section 3.5.

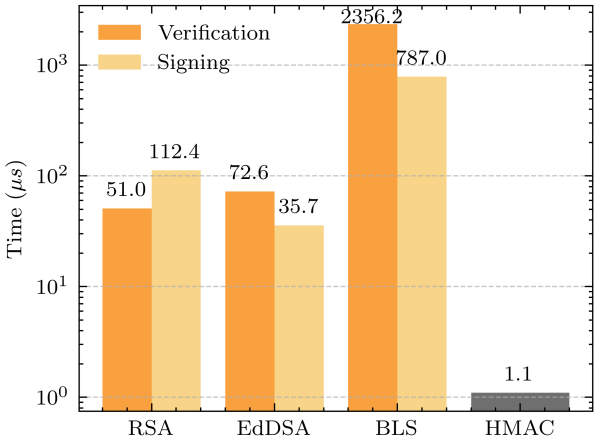


Figure 5.6: Cryptograhpy functions benchmark.

We implemented each cryptography option discussed above and compared their base latency which be seen in figure 5.7. Note that performing BLS aggregation already makes Alea-BFT have a better base latency result than QBFT. Adopting HMAC, RSA, and EdDSA makes it even faster by 152%, 251%, and 381%, respectively. Here, it's worth noting that QBFT could also make use of the MAC optimization for the proposal and commit messages, but the prepare and round-change messages would still require BLS signatures.

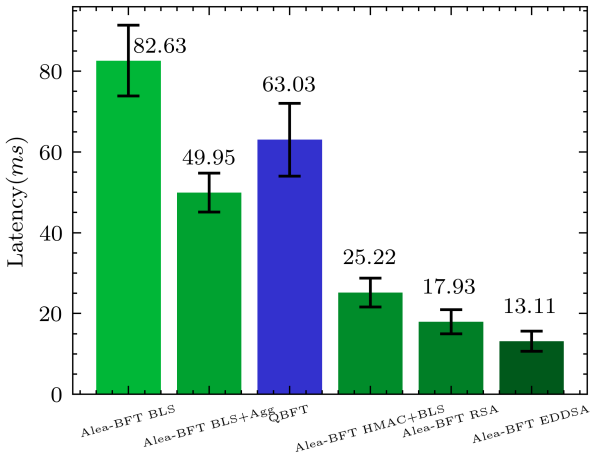
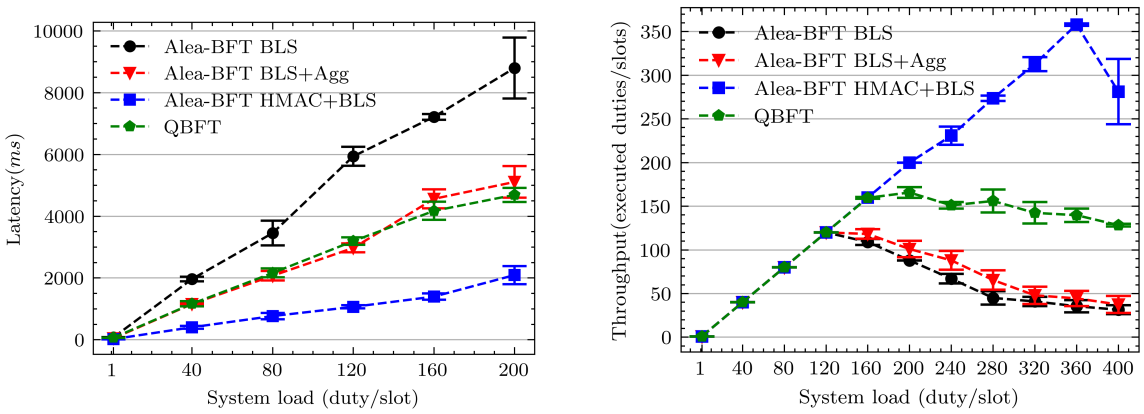


Figure 5.7: Base latency with different cryptography optimizations.

To retain the usage of BLS, we continue for the next experiments with the Alea-BFT version with normal BLS (without performing signature aggregation), with BLS with aggregation, and with BLS and

HMAC.

To verify how different loads affect the protocols, we proceeded to compare the average latency between QBFT and Alea-BFT, for these different versions, for increasing system load. Figure 5.8(a) shows the results for experiments with 1, 40, 80, 120, 160, and 200 duties per slot. It can be seen that Alea-BFT with aggregation was able to keep up with the average latency of QBFT, while Alea-BFT with HMAC and BLS was able to maintain lower latencies for higher system loads.



(a) Latency

(b) Throughput

Figure 5.8: Performance by system load for different cryptography optimizations.

Regarding the throughput metrics, we also performed experiments with 240, 280, 320, 360, and 400 duties per slot in order to completely capture the throughput potential of the HMAC and BLS combination. By looking at figures 5.8(b) and 5.9, we can see that Alea-BFT with HMAC can outperform the QBFT results. BLS with aggregation, on the other hand, continued to present a smaller throughput peak. A reason for this is that, even though Alea-BFT with BLS aggregation has comparable latencies to QBFT, QBFT doesn't have to process any more messages after it knows the consensus result, which is not true for Alea-BFT which has to keep processing more messages due to possible divergent views of the network.

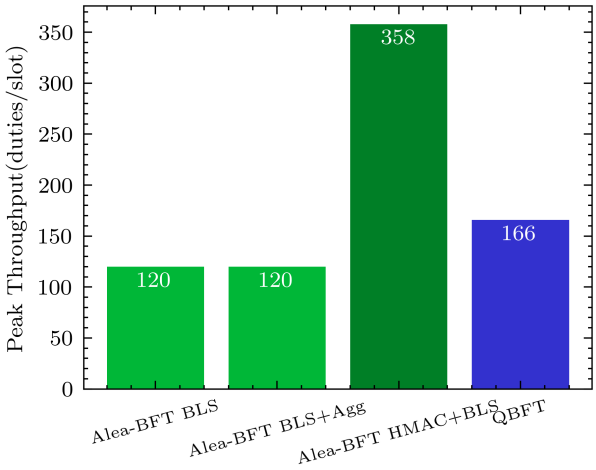


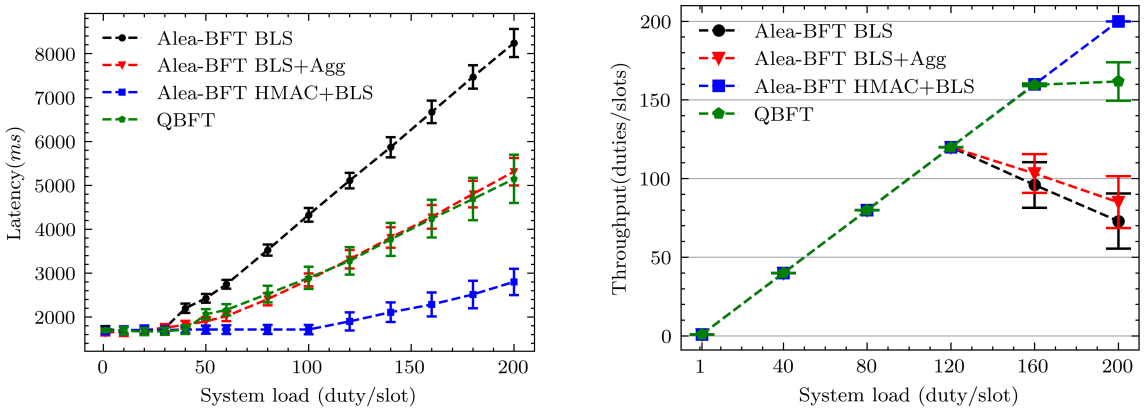
Figure 5.9: Peak throughput for different cryptography optimizations.

This section showed that the use of efficient cryptography schemes can have a great impact on the performance of the Alea-BFT protocol in local area networks since processing time dominates message transmission delay. However, when the network nodes are far apart, with big transmission delays, the most important factor is the number of steps that the protocol has to make until termination, which will be explored in section 5.5.

5.5 Results on wide area networks

The total execution time of a decentralized protocol can be seen as a function of the processing time and the message transmission delay. Until now, we have been considering local area setups, in which the transmission delay is minimal and, thus, processing time takes the most important role. Nonetheless, the SSV network is permissionless and open for anyone to join. Therefore, committee nodes may be spread throughout the whole world and, thus, it's important to understand how Alea-BFT behaves in a wide area setup.

For that, we used the *tc qdisc* command, available in Linux, to simulate transmission delay between machines. The delay is calculated as a normally distributed variable for a given mean and standard deviation values. We initially used 500 milliseconds as the mean value and 100 milliseconds as the standard deviation, as suggested by data scrapped from the company's scenario. The results are shown in figure 5.10.



(a) Latency

(b) Throughput

Figure 5.10: Performance by system load for wide area with $500\ ms$ transmission delay.

Figure 5.10(a) indicates that, for lower system loads, both QBFT and Alea-BFT, with different cryptography models, produced similar latencies. This is reasonable since the transmission delay dominates the total time and both QBFT and the optimized version of Alea-BFT take 3 transmission steps to terminate. For instance, in a normal scenario, the optimized Alea-BFT starts with each node broadcasting a *VCBC Send* message. After the first transmission delay, nodes receive the *VCBC Send* messages and broadcast a *VCBC Ready* message. After a second transmission delay, nodes receive the *VCBC Ready* messages and broadcast a *VCBC Final* message. After the third transmission delay, nodes receive the *VCBC Final* messages and terminate in case all VCBCs contain the same proposed value. For QBFT, this is similar but with the *proposal*, *preapre*, and *commit* messages.

After a certain system load, latency starts to increase. This can be explained by the fact that a higher system load makes the protocols exchange more messages and the processing time starts to be comparable to the transmission delay. As the system load increases, the processing time dominates even more the transmission delay making the results look similar to the ones obtained in the local area experiment. These turning point values were: 30 for Alea-BFT with normal BLS; 40 for Alea-BFT with BLS aggregation and for QBFT; and 100 for Alea-BFT with HMAC and BLS.

Regarding the throughput metric, shown in figure 5.10(b), the results had little difference from the local area case. This is expected because, after 120 duties per slot (the first throughput degrading point), the behaviors of the protocols were similar to the behaviors seen in the local area experiment.

To show how high transmission delays dictate the total latency time for low system loads, we performed experiments with increasing system delays in which the operators had to decide only one duty. The standard deviation was kept at $100\ ms$. The results, shown in figure 5.11, confirm that all protocols followed precisely the transmission delay increment, as expected

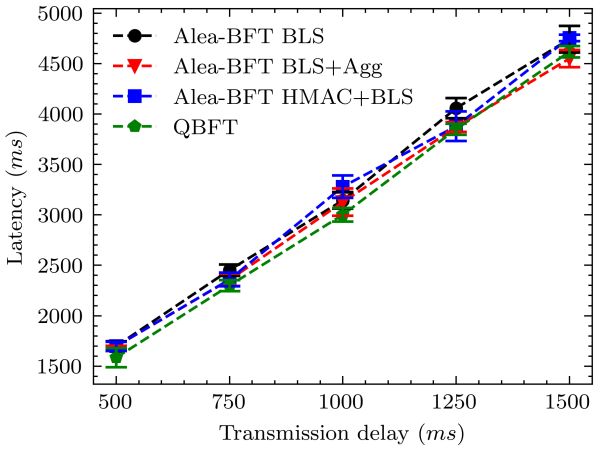


Figure 5.11: Latency per transmission delay.

5.6 Performance as a function of the network size

Next, in figure 5.12, we examine how the base latency changes for different system sizes, i.e. the number of operators. Currently, in SSV, a validator can only hire 4, 7, 10, or 13 operators, as defined in its smart contract. Even though this small set of possible group sizes restricts our view of the complexity pattern of the base latency, figure 5.12 suggests that the protocols follow a quadratic pattern. This is expected since the message and communication complexities of both algorithms are quadratic. Note that, even with Alea-BFT's Complete VCBC View optimization, which may prevent an execution from having to perform a whole agreement round, the complexity remains quadratic since $O(N)$ VCBCs are executed and each has a linear complexity.

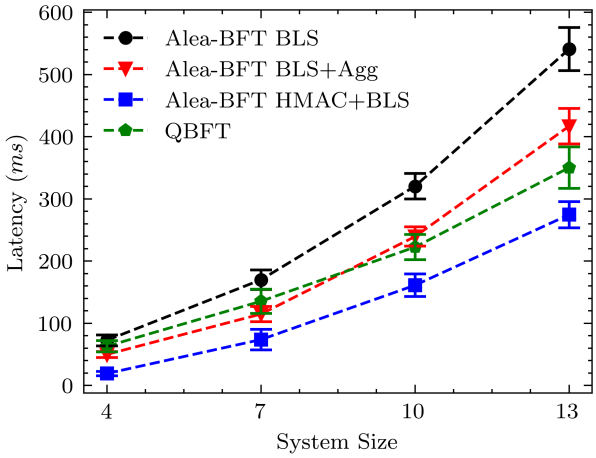


Figure 5.12: Base latency per system size.

Even though Alea-BFT with the BLS aggregation optimization is faster than QBFT for a smaller num-

ber of nodes (4 and 7), for a higher number of nodes its latency seems to cross the QBFT curve. This can be explained by the fact that the communication layer of SSV doesn't allow direct messages between peers and, thus, a peer actually receives several *VCBC Ready* messages that are not relevant to it. Indeed, at maximum (if all nodes work correctly), a node would receive N *VCBC Ready* messages for each of $N-1$ VCBCs of which it's not the author, and, thus, a total of $N \times (N-1)$ extra *VCBC Ready* messages. Even though it doesn't process these messages, they represent unnecessary bandwidth usage and overload the network. Also, this negative effect is quadratic, $O(N^2)$, making it more severe for higher system sizes. Notice that this affects every Alea-BFT version implemented using this communication layer interface, including the version with the simple BLS usage and the one with the HMAC+BLS optimization.

5.7 Performance under faulty scenarios

One of the most important features of asynchronous BFT protocols is their resilience to network instability problems and node faults. Thus, our next step aims to analyze what happens with the performance of each protocol under a faulty scenario.

Figure 5.13 shows the base latency comparison for the normal case, for a crashed node, and for a byzantine node scenario. The crash or byzantine fault occurs right at the beginning of the slot. In the QBFT experiment, the leader process was chosen to be faulty one, which has a $1/4$ probability of happening, while in the Alea-BFT experiment the faulty node was chosen at random. In this QBFT implementation, it was used a round-change timeout of 2 seconds, as defined by the SSV implementation.

For the Alea-BFT protocol, the byzantine node behavior was implemented in order to slow down the protocol execution by not answering properly to the protocol messages. So, for instance, it does not respond properly to a *VCBC Send* message, and it tracks the ongoing ABA state in order to send messages containing opposite votes to the majority. Also, it sends extra invalid messages that will be dropped by honest peers but will force them to waste resources with signature verification. For the QBFT protocol, the byzantine node was implemented in a similar way plus the addition that it won't broadcast valid *Proposal* messages for a round in which the faulty node is the leader.

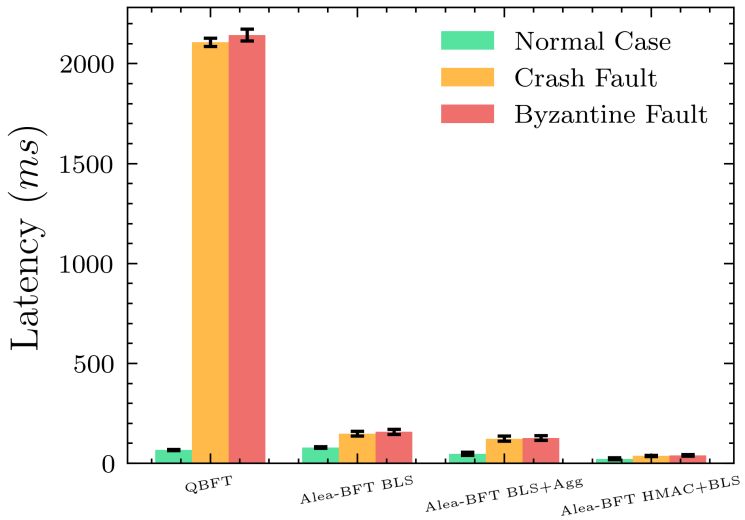


Figure 5.13: Base latency for different fault scenarios.

It can be seen that the fault effects on Alea-BFT are minimal compared to QBFT. This is due to the fact that, if the QBFT leader is faulty, the protocol will remain idle until a timeout occurs. On the other hand, Alea-BFT is able to make progress despite a faulty node. This occurs because the correct functioning of the protocol is not dependent on a single node but on a quorum of them. Notice that the Fast ABA and the Complete VCBC View protocol optimizations do not accelerate the protocol termination in these faulty scenarios because each of these needs all the nodes working correctly, which can explain why the Alea-BFT base latency under these faults is higher than under a normal case scenario. Actually, the results became similar to the base latency of the unoptimized version.

Another experiment was performed in order to analyze the effects of faulty scenarios in the protocol's throughput, which can be seen in figure 5.14. The figure shows the trace of the protocols' throughput for a sequence of slots for a system load of 100 duties per slot. During slots 1 to 10, all the nodes are honest and all protocols are able to reach the desired throughput. At the beginning of slot 11, a random node crashes and it persists in failure until slot 20. For slots 21 to 30, all nodes work correctly again.

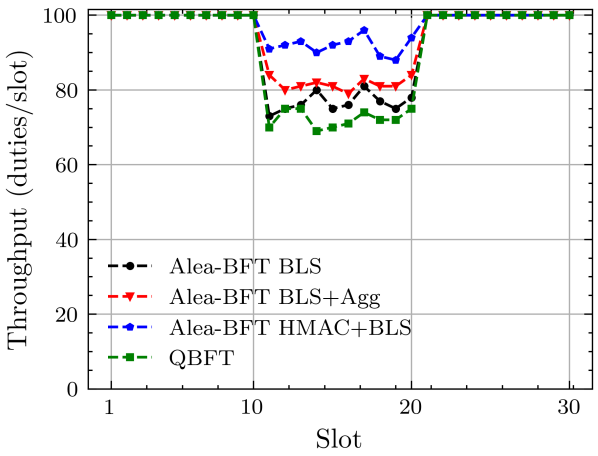


Figure 5.14: Throughput trace with crash fault.

The result shows that Alea-BFT is more resilient to these types of faults, being able to maintain a higher latency throughout the crash period. Again, this is explained by the design principles of Alea-BFT, which allow the protocol to make progress even with faulty nodes, while QBFT is severely dependent on the leader functioning well.

6

Conclusion

Contents

Among the components of a validator network, there's a consensus layer that allows operators to agree on some value. The objective of this work was to integrate Alea-BFT, an asynchronous BFT protocol, into the real-world application of validator networks. Currently, Ethereum recommends the QBFT protocol for the consensus layer of such networks. Our work, therefore, aimed to show that Alea-BFT can provide similar performances to partially synchronous protocols, such as QBFT, with the advantage of being more resilient to network instability and fault scenarios, and, thus, being able to be adopted in practice.

During the design of the integration of Alea-BFT into the SSV network code, the biggest obstacle was to adapt Alea to a one-shot consensus, as abstracted by the SSV network, instead of implementing it to be used as a state machine replication which would be more suitable to its design. However, throughout the process, we proposed many protocol optimizations that took advantage of the adaptations that we needed to make.

Regarding the experimental component, we verified how these protocol optimizations were able to make Alea-BFT have comparable results to QBFT. Then, we analyzed in detail the breakdown of

Alea-BFT's performance, providing profiling data and showing exactly the bottleneck component of our implementation which is cryptography. To diminish the hurdles of this bottleneck, we proposed some suitable alternative solutions and optimizations that don't require SSV to change its cryptographic scheme. These alternatives allowed Alea-BFT to even outperform QBFT.

We also analyzed the protocol behavior for wide area scenarios, to take into consideration the possibility of geographically spread operators. We showed that QBFT and the optimized version of Alea-BFT had similar results due to the similar number of steps they require until termination. Moreover, we investigated the impact of increasing the network size, by adding more operators, and confirmed our expectations of a quadratic behavior for both protocols. Lastly, we evaluated these protocols under different fault scenarios. This was an important experiment because SSV is permissionless and, as more people participate in the network, it can not guarantee that operators will behave honestly. Alea-BFT, as expected, showed much better performance due to its resilience to network instability and node faults. Therefore, even though QBFT may show good results under stable scenarios, its leader-driven design comes with the price of considerable performance impact under these fault conditions.

In conclusion, our implementation of Alea-BFT in validator networks provided similar results to the partially synchronous QBFT protocol. One of Alea-BFT's drawbacks was the higher number of exchanged messages. But this is a price associated with its leaderless design. On the other hand, one of the advantages of this design is the ability to keep making progress even under fault situations, which can become more frequent as the SSV network increases its size.

6.1 System Limitations and Future Work

The biggest implementation challenge regards the one-shot consensus design, specified by the SSV interface. Though it would require considerable changes, this interface could be modified in order to add a state machine replication interface to the network. Instead of executing different consensus instances with several different committees, the network as a whole would participate in the state maintenance. This would allow Alea-BFT to use batching, one of its powerful features used to increase throughput. More than that, since attestation is by far the most frequent duty, many different consensus instances that occur in parallel decide on the same data. This repeated work would be done in a single step of the state machine replication.

The communication layer design also posed some limitations to Alea-BFT's performance, since unnecessary *VCBC Ready* messages increased the network congestion. To minimize this preformance penalty, one could keep the GossipSub protocol, since it's widely adopted and tested against common network attacks, but add an extra communication layer dedicated exclusively to direct messages.

We suggested several cryptography optimizations, including even changing the used cryptographic

scheme. Indeed, although Ethereum uses the BLS scheme and partial signature messages are forced to use it, consensus messages could apply more efficient schemes, such as RSA and EdDSA. Regarding the VCBC threshold signature, since these schemes don't allow signature aggregation, one could send a buffer of signatures since these asymmetric schemes provide non-repudiation.

Bibliography

[1] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:261284440

[2] V. Buterin and V. Griffith, “Casper the Friendly Finality Gadget,” *arXiv e-prints*, p. arXiv:1710.09437, Oct. 2017.

[3] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling byzantine agreements for cryptocurrencies,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 51–68. [Online]. Available: https://doi.org/10.1145/3132747.3132757

[4] L. Lamport, R. E. Shostak, and M. C. Pease, “The byzantine generals problem.” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982. [Online]. Available: http://dblp.uni-trier.de/db/journals/toplas/toplas4.html#LamportSP82

[5] A. Oliveira, H. Moniz, and R. Rodrigues, “Alea-bft: Practical asynchronous byzantine fault tolerance,” 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2202.02071

[6] H. Moniz, “The istanbul bft consensus algorithm,” 2020.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *J. ACM*, vol. 32, no. 2, p. 374–382, apr 1985. [Online]. Available: https://doi.org/10.1145/3149.214121

[8] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The honey badger of bft protocols,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–42. [Online]. Available: https://doi.org/10.1145/2976749.2978399

[9] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” May 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[10] S. King and S. Nadal, “Ppcoin: Peer-to-peer crypto-currency with proof-of-stake,” 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:42319203

[11] P. Howson and A. de Vries, “Preying on the poor? opportunities and challenges for tackling the social and environmental threats of cryptocurrencies for vulnerable and low-income communities,” *Energy Research Social Science*, vol. 84, p. 102394, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214629621004813

[12] A. Beikverdi and J. Song, “Trend of centralization in bitcoin’s distributed network,” in *2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2015, pp. 1–6.

[13] K. Mohsin, “Cryptocurrency its impact on environment,” in *International Journal of Cryptocurrency Research*, 2021.

[14] A. Shamir, “How to share a secret,” *Commun. ACM*, vol. 22, no. 11, p. 612–613, nov 1979. [Online]. Available: https://doi.org/10.1145/359168.359176

[15] D. Boneh, M. Drijvers, and G. Neven, “Compact multi-signatures for smaller blockchains,” in *Advances in Cryptology – ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2018, p. 435–464. [Online]. Available: https://doi.org/10.1007/978-3-030-03329-3_15

[16] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI ’99. USA: USENIX Association, 1999, p. 173–186.

[17] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, “Dumbo: Faster asynchronous bft protocols,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 803–818. [Online]. Available: https://doi.org/10.1145/3372297.3417262

[18] M. O. Rabin, “Randomized byzantine generals,” in *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, 1983, pp. 403–409.

[19] M. Ben-Or, “Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols,” in *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’83. New York, NY, USA: Association for Computing Machinery, 1983, p. 27–30. [Online]. Available: https://doi.org/10.1145/800221.806707

[20] M. K. Reiter, “Secure agreement protocols: Reliable and atomic group multicast in rampart,” in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, ser. CCS ’94. New York, NY, USA: Association for Computing Machinery, 1994, p. 68–80. [Online]. Available: https://doi.org/10.1145/191177.191194

[21] D. Malki and M. Reiter, “A high-throughput secure reliable multicast protocol,” in *Proceedings of the 9th IEEE Workshop on Computer Security Foundations*, ser. CSFW ’96. USA: IEEE Computer Society, 1996, p. 9.