University
of Stavanger

**DANIEL DIRDAL AND ERLEND BYGDÅS**

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

# Ethereum 2.0: Deploying a Private Devnet using Proof-of-Stake

Master's Thesis - Computer Science - June 2024

I, **Daniel Dirdal and Erlend Bygdås**, declare that this thesis titled, "Ethereum 2.0: Deploying a Private Devnet using Proof-of-Stake" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master's degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

*”All roads lead to me.”*

– Lee ”Faker” Sang-hyeok

# Abstract

In 2022, Ethereum, one of the largest and most utilized blockchains, underwent a major upgrade called *the merge*. This transition from proof-of-work to proof-of-stake aimed to enhance the network's scalability, security, and energy efficiency. Researchers have since published numerous papers on Ethereum, examining its security and identifying potential vulnerabilities.

This thesis focuses on deploying a private development network for Ethereum 2.0 to facilitate detailed simulations and evaluations of these proposed security concerns. The primary objective is to create a robust and flexible testing environment that allows for studying various attack scenarios and the effectiveness of consensus protocols.

The research involves developing tools for managing Ethereum nodes, such as simplifying the node deployment processes and enabling features for validator management. Key contributions include creating scripts for easy setup and maintenance of Ethereum nodes, user-friendly monitoring systems, and ensuring client diversity to enhance network resilience. It also lays the groundwork for enabling Byzantine behavior in validators, enabling researchers to perform simulations to test their attack scenarios even more easily.

Experimental conducted offers valuable guidelines for researchers and developers who wish to deploy their own nodes. The findings highlight the differences in hardware usage between the private development network and mainnet, providing a comprehensive analysis of resource requirements under different configurations.

Not only does the work address the technical challenges associated with deploying and managing Ethereum 2.0 nodes, but it also contributes to the Ethereum community by enhancing its usability and accessibility.

# Acknowledgements

We want to express our deepest gratitude to our supervisors, Leander Jehl and Arian Baloochestani Asl, for their invaluable guidance and support throughout our research. Their expertise was instrumental in the successful completion of this thesis.

We would also like to thank Chong-He from the Lighthouse team. His assistance and insights were crucial in overcoming several technical challenges we encountered.

# Contents

# Chapter 1

# Introduction

Since its inception, Ethereum has undergone continuous evolution, leading to the development of Ethereum 2.0, a major upgrade aimed at enhancing the network's security and sustainability. This upgrade introduces significant changes, including transitioning from a proof-of-work to a proof-of-stake consensus mechanism, which necessitates deploying consensus and validator clients. Large amounts of theoretical research have focused on identifying potential vulnerabilities and improving the security of Ethereum 2.0. Multiple research papers [1, 2, 3] provide valuable insights into potential attack vectors, weaknesses, and solutions. However, these studies lack practical testing, leaving a gap in the validation of their findings.

A private network is a controlled environment where developers can experiment with new features, test network upgrades, and simulate transactions without impacting the main network. These networks are known as *devnets*, short for development networks. By leveraging devnets, researchers can replicate theoretical attack scenarios to test their impacts on the network and verify the correctness of protocols. For example, they can test Ethereum 2.0 mechanisms, such as the inactivity leak or the proposer boost. Devnets are essential for assessing a network's resilience against challenging scenarios and validating the correctness of the protocols. These capabilities help understand network behavior under stress and devise strategies to enhance robustness and reliability.

As the Ethereum ecosystem constantly evolves with updates and improvements, documentation is often minimal and lacks comprehensive coverage of the full functionality required to effectively deploy and manage Ethereum nodes. This gap in documentation makes setting up a devnet a non-trivial task that requires a deep understanding of the underlying protocols.

We do not aim to perform the simulations described in the papers referenced. However, we aim to provide an application for researchers to test such theories in a controlled environment, thereby enabling research validation.

## 1.1 Objectives

The primary objective of this thesis is to create a robust and flexible testing environment for Ethereum 2.0, which uses the proof-of-stake consensus mechanism. This application can be a viable tool for researchers performing detailed simulations and evaluating various attack scenarios. By providing a practical platform for these activities, we aim to support the validation of theoretical research and enhance the overall understanding of Ethereum 2.0's security and functionality. To enable these capabilities, we have outlined several specific goals:

- **Research and Evaluation of Tools:** Conduct a comprehensive review of the existing tools for managing Ethereum nodes. This involves evaluating their functionalities, ease of use, and integration capabilities to identify the best options for our application.
- **Simplification of Node Deployment:** Develop streamlined processes for deploying Ethereum nodes, including execution, consensus, and validator clients. This objective focuses on simplifying the setup process to make it accessible to researchers and others who want to experiment with Ethereum.
- **Validator Management:** Study and utilize existing tools for managing validators and develop scripts to streamline operations such as setup and maintenance.
- **Node Monitoring:** Set up user-friendly monitoring that enables researchers to observe the status and performance of nodes easily.
- **Client Diversity:** Ensure the application supports multiple Ethereum clients to promote client diversity. This helps assess the network's resilience and performance across different software implementations.
- **Measuring Hardware Usage:** Experiment with different devnet configurations to examine the hardware usage of the clients, comparing findings to the minimum and required resources stated by the clients utilized.
- **Enable Byzantine Clients:** Implement byzantine functionality, laying the groundwork for allowing researchers to study and perform possible attacks on the chain.

These objectives together create a strong and flexible environment for Ethereum 2.0 research, enabling researchers to explore and make valuable contributions to Ethereum.

## 1.2 Approach and Contributions

The approach adopted in this thesis involved a comprehensive investigation into Ethereum 2.0's core functionalities and requirements, focusing on the intricacies of node deployment and validator management. Through this exploration, the thesis identifies and addresses several gaps in the existing methodologies for setting up and managing Ethereum nodes. Based on this study, we developed an application that provides the end user with a suite of scripts designed to achieve all the previously defined objectives. These scripts streamline the processes of node deployment, validator management, and node monitoring, facilitating easier experimentation and research within the Ethereum 2.0 ecosystem.

### Contributions

This thesis makes several significant contributions to Ethereum's research and development. Our work enhances the usability and accessibility of Ethereum 2.0 by addressing key challenges and introducing innovative solutions. The following points summarize the primary contributions of this thesis:

- **Detailed Study of Ethereum 2.0:** This thesis offers an in-depth examination of Ethereum, focusing on its critical components and underlying principles that are essential for understanding Ethereum 2.0. Delving deeply into various technical aspects. By providing a thorough technical overview, the thesis prepares readers with the necessary knowledge to understand the key components of Ethereum 2.0.
- **Comprehensive Ethereum 2.0 Development Network and Tools:** This thesis contributes to the field by developing a method for deploying a private Ethereum 2.0 development network. It also provides tools to cover comprehensive functionalities, including setting up validators and making deposits, withdrawing funds, and exiting validators. Compared to previous attempts by others to create similar environments, this thesis presents a more complete and feature-rich solution. Earlier efforts, discussed later in Chapter 4, have fallen short, primarily due to a lack of comprehensive features and user-centered design.
- **Resource Comparison Between Devnet and Mainnet:** We analyze the hardware requirements for running Ethereum clients on a private network and compare them to the mainnet requirements. By doing so, we provide users with the necessary information to make informed decisions about the resources they need for their simulations and testing environments.

In conclusion, this thesis not only addresses the technical challenges associated with deploying and managing Ethereum 2.0 nodes but also significantly contributes to the broader Ethereum ecosystem by enhancing its usability and accessibility. This work, through detailed research and development, paves the way for robust and user-friendly interactions with Ethereum.

## 1.3 Outline

This section provides a structured overview of the thesis, detailing the contents and focus of each chapter, guiding the reader through the progression of research and analysis undertaken.

- **Chapter 2** lays the foundational concepts necessary for understanding this thesis. It introduces the technical background and theoretical foundation of Ethereum and blockchain technologies.
- **Chapter 3** thoroughly examines Ethereum 2.0, focusing on its key upgrades and innovations over previous versions. It gives an in-depth explanation of Ethereum.
- **Chapter 4** presents a review of existing literature and the various Ethereum 2.0 clients currently available. It presents previous attempts at solving similar problems.
- **Chapter 5** details the research methods and the approach adopted for developing the proposed solution. It outlines the steps taken in the design and implementation phases of the application.
- **Chapter 6**, we perform experiments using the devnet. We analyze the results of the experiments.
- **Chapter 7** critically reflects on the application and the choices made during the implementation phase. We discuss the selection of tools used, the challenges encountered, and how these issues were addressed. Additionally, we compare the proposed solutions with related work, highlighting the advancements and contributions made.
- **Chapter 8** concludes the thesis.

# Chapter 2

# Background

This chapter will provide the foundational concepts necessary for understanding the remainder of this thesis. We start by examining Merkle trees and their usage. Then, we look at consensus and its importance in distributed systems and blockchains. Next, we delve into the workings of blockchains and explore various consensus protocols used in blockchain networks, such as proof-of-work and proof-of-stake. We also look at how blockchains can be updated through soft and hard forks. Then, we provide a concise overview of the Ethereum Virtual Machine and its enabling of smart contracts. Finally, we conclude with digital signatures essential for ensuring blockchain security.

## 2.1 Merkle Tree

A *Merkle tree* is a tree data structure where leaf nodes are labeled with the hash of the data, and non-leaf nodes (intermediate nodes) are the result of the hash of their children [4]. Merkle trees provide an efficient way to compare data by exchanging the root hash. It also provides an efficient way to verify the membership of a leaf without requiring the verifier to know all the leaves by using *Merkle proofs*.

When dealing with large Merkle trees, adding a leaf entails recomputing all intermediate nodes up to the new node, which could be impractical for production use. To address this issue, an optimization called an *incremental Merkle tree* initializes the tree with empty nodes (filled with zeroes) and maintains a partial Merkle tree for updates. In an incremental Merkle tree, leaf nodes are incrementally added from left to right, updating the partial Merkle tree by replacing an empty leaf. This approach ensures that only the hashes of the nodes on the path from the new leaf to the root need to be recomputed.

## 2.2 Consensus

Consensus is fundamental to distributed systems and blockchains, enabling multiple participants (often called nodes) to agree on a common, unchangeable value. Given the unreliable infrastructure, which can suffer from crashes, corrupted messages, and other failures, a robust protocol is essential to manage these challenges. In distributed systems, we examine two key properties: *safety* and *liveness* [5].

- Safety ensures that nothing bad will happen. More formally, if a safety property is violated at any time $t$, it cannot be satisfied again after that point.
- Liveness guarantees that something good will eventually happen. More formally, for any time $t$, there is a possibility that the property will be satisfied at some future time $t' \geq t$.

For consensus, a safety property is that every participant learns the same value, while a liveness property is that a value will eventually be chosen.

When selecting a consensus protocol, one must consider the failure model in which one operates. The two most common failure models are *crash-stop* and *Byzantine* [5].

In the crash-stop model, a node follows the protocol correctly until it crashes, stopping executing steps and sending messages. In contrast, the Byzantine model, commonly used in blockchains, allows nodes to exhibit arbitrary behavior. This means a node could be down, non-communicative, following a different protocol version, attempting to mislead other nodes, publishing contradictory messages, or displaying other faults.

## 2.3 Blockchain

In its simplest form, a blockchain is an immutable data structure consisting of blocks, each cryptographically linked to its predecessor. This linkage ensures the integrity of the blockchain, as it becomes impossible to remove or alter a block once it has been added. A basic example of a blockchain is shown in Figure 2.1.

Each block in a blockchain consists of two parts: the block header and the block body. The block header contains metadata about the current block, including the hash value of the parent block (each block is linked to its parent block through the hash value of the parent block), timestamp, Merkle Tree Root, and other information. The Merkle Tree is a binary tree where each leaf node represents a transaction record. It summarizes all transactions in a block. The Merkle Tree provides a digital fingerprint of the entire transaction set, allowing for efficient verification of specific transactions within the block.

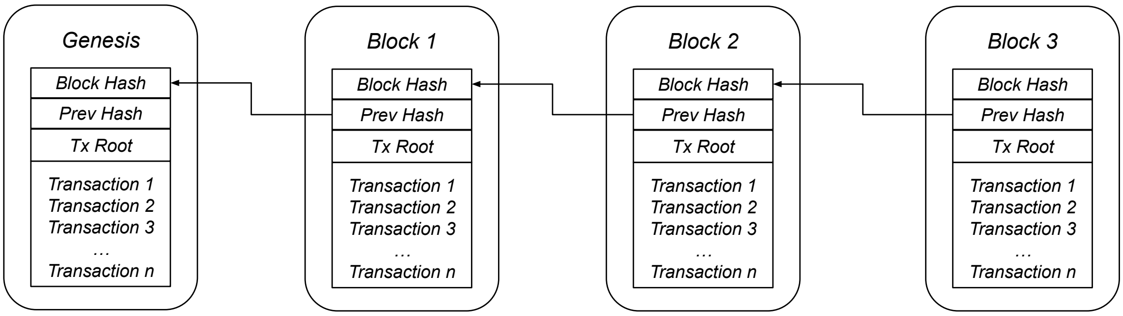

Figure 2.1: Each block points to its parent blocks through the *Prev Hash*, creating an immutable and tamper-evident chain.

Any modification to a block would result in a different hash value, making such changes easily detectable. For example, in Figure 2.1, changing block 1 would affect its child, block 2, because block 2's reference to block 1's hash would no longer match. This discrepancy would cascade through the blockchain, affecting block 3, block 4, and all subsequent blocks. The first block in the chain, which has no predecessor, is typically called the *genesis* block.

Another perspective on a blockchain is as a replicated state machine, which all participants maintain by applying blocks in a specific agreed-upon sequence. Each block causes a state transition, determined by a function that inputs the previous state and the new block to produce an updated state. Since all nodes start from a predefined genesis block and process the same sequence of blocks, they ultimately converge to the same state. Since each participant maintains their own local copy of the state machine, it is crucial to employ a consensus protocol to ensure all participants have the correct data and decide who will add the next block. When choosing a consensus protocol, it is also important to consider whether the blockchain operates in a *permissionless* or *permissioned* environment [6].

**Permissioned Blockchains** Permissioned blockchains restrict participation to authorized readers and writers. In these systems, a central entity controls and grants the rights to participate in read and write operations. Such blockchains typically rely on Byzantine fault tolerance protocols, which assume a known set of participants to generate and reach consensus on blocks.

**Permissionless Blockchains** Bitcoin [7] and Ethereum [8] are the most popular permissionless blockchains known for their openness and decentralization. In these systems, anyone can join or leave the network as a reader and/or writer. There is no central authority to manage membership or to ban illegitimate participants. As a result, the content on these blockchains is publicly accessible to anyone.

The two most common consensus protocols for permissionless blockchains are *proof-of-work* and *proof-of-stake* [9]. While often referred to as consensus protocols, they are actually *Sybil* resistance mechanisms. Both impose a cost on participation to prevent attackers from overwhelming the protocol at little or no cost.

### 2.3.1 Proof-of-Work

Proof-of-work, also called Nakamoto consensus [7], is the first consensus protocol for permissionless blockchains, where the cost of participation is in the form of computational resources.

Nodes operating within a proof-of-work system engage in a process known as *mining*, wherein they seek a nonce to solve a mathematical puzzle. This puzzle is solved if a nonce, when hashed with the block's content, produces a hash value lower than a predetermined target value, referred to as the target *difficulty*. The difficulty parameter dictates the average number of nonces a miner must iterate through. The miner can extend the blockchain by adding a new block upon solving the puzzle.

If an attacker seeks to alter a previous block, they must redo the proof-of-work for the target block and all subsequent blocks. This is because any change to an earlier block alters the hash value of all subsequent blocks built upon it. This renders an attack on proof-of-work blockchains impractical unless the attacker controls at least 51% of all mining power.

### 2.3.2 Fork-Choice Rule

In proof-of-work systems, it is possible for two or more blocks to be generated simultaneously, both pointing to the same predecessor. When this happens, it creates a *fork*, where participants must choose which block to follow, as each block represents a different version of the blockchain's state. To resolve this, blockchains implement a *fork-choice rule* to find a single leaf block, often referred to as the *head block*, upon which honest participants should follow and build upon. The chain extending from the genesis block to the head block is called the *canonical chain*. Any fork not included in the canonical chain is reorganized out, implying that it is not recognized within the blockchain's worldview.

The most prevalent fork-choice rule is the *heaviest chain rule*, used by Bitcoin and Ethereum before transitioning to proof-of-stake [7]. This rule selects the chain with the highest accumulated work. People often assume that the heaviest chain rule is the same as the longest chain rule, which selects the chain with the greatest number of blocks, but they are not identical. However, in about 99% of cases, they produce the same result, as the chain with the highest accumulated work usually also has the greatest number of blocks.

Another prevalent fork-choice rule is the *Greedy Heaviest Observed SubTree* (GHOST) [10], which prioritizes the heaviest subtree when identifying the head block. GHOST acknowledges that a published block serves as a vote for all its ancestors, thereby attributing a notion of *weight* to blocks rather than solely considering their height. In the event of a fork, GHOST follows the block with the heaviest subtree until it finds a single-head block.

Figure 2.2 highlights how the two fork-choice rules find a head block for honest participants. Under the longest chain rule, participants would build upon the blue block and discard alternative chains, as the head block has the highest height of 6. In contrast, blockchains using GHOST would prioritize the light red block as the head block due to its consideration of following the chain with the heaviest subtrees.

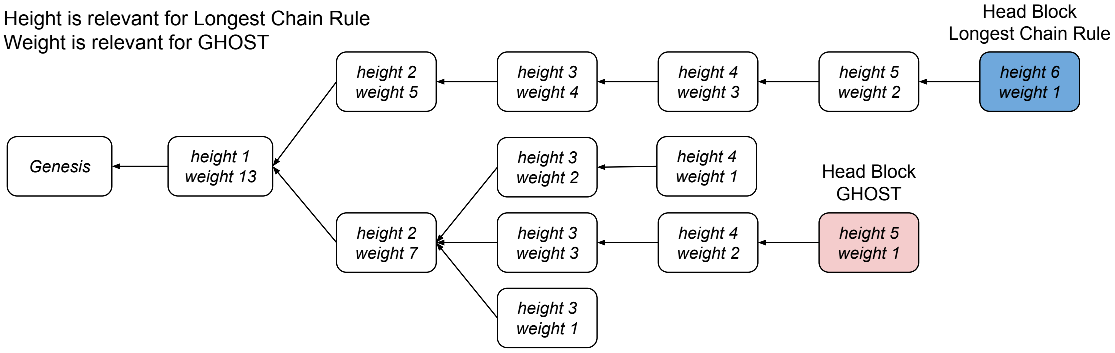

Figure 2.2: The height represents the distance from the genesis block, with the genesis block having a height of 0. The weight denotes the weight of each subtree.

### 2.3.3 Proof-of-Stake

Proof-of-stake is another Sybil resistance protocol employed in permissionless blockchains, which addresses the high resource consumption associated with proof-of-work. Instead of participating through computational resources, participants must stake a certain amount of the blockchain's cryptocurrency to engage in the protocol. In recent times, proof-of-stake adoption has significantly increased due to its reduced environmental footprint. Blockchains such as Ethereum, *Cardano* [11], *Solana* [12], *Algorand* [13], *Polkadot* [14] and many more utilize proof-of-stake as their consensus protocol. However, early proof-of-stake protocols have been susceptible to attack vectors such as *nothing at stake* and *long-range attack* [15].

Nothing at stake refers to when participants attempt to build on two or more blocks when forks occur. This issue is typically addressed by introducing some form of punishment for malicious behavior by making participants sign their published messages.

On the other hand, long-range attacks involve participants withdrawing their stake from the honest chain and then building a new chain from a block preceding the stake withdrawal. Subsequently, they publish the new chain, which diverges from the history of the honest chain such that the honest participants can't discern which chain to follow.

Long-range attacks are effectively tackled by integrating *finality* into the blockchain. This mechanism selects checkpoints that participants agree about, so everything before the checkpoint is finalized and will never be reverted, as shown in Figure 2.3. When a chain emerges from a block predating this checkpoint, honest participants unanimously disregard it. Subsequently, the protocol relies on a fork-choice rule to determine the validity of any post-checkpoint forks.

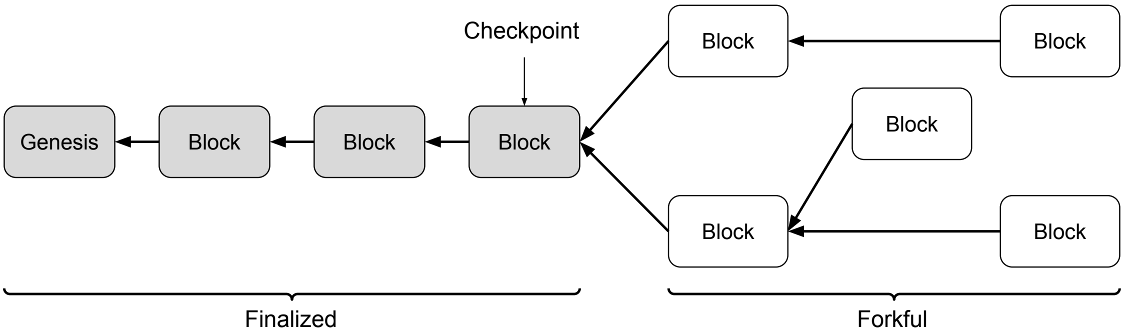

Figure 2.3: The consensus among honest participants confirms that the checkpoint block and all its ancestors are finalized and immune to reversal. Consequently, all forks before this checkpoint are reorganized to make a single chain. However, the descendants of the checkpoint block are susceptible to forks, allowing forking to occur. This characteristic renders long-range attacks unfeasible.

### 2.3.4 Updating a Blockchain

Permissionless blockchains involve participants who each maintain their localized perspective of the chain. These participants operate diverse software versions with no established mechanism to track updates. When a blockchain seeks to modify its functionality, such as altering block size or updating consensus rules, two primary methods are typically employed: *soft fork* and *hard fork* [16].

**Definition 2.1.** A soft fork is an update that introduces changes, enabling certain transactions or blocks to remain valid under both the new and prior versions, ensuring backward compatibility. However, blocks or transactions from the previous version are invalid under the new version.

The outcome of a soft fork hinges on the adoption rate among users. If less than $50\%$ of users update, the blockchain splits into two forks, each having a distinct view of the canonical chain. Conversely, if more than $50\%$ adopt the new version, users operating the old version will produce blocks that become orphaned.

**Example 2.1.** Reducing the block size from $1$ MB → $0.5$ MB can be done as a soft fork because nodes that don't upgrade their software will still accept the new smaller blocks. However, the updated nodes are not guaranteed to accept blocks from non-upgraded nodes, as these blocks can exceed the size limit set by the updated protocol.

**Definition 2.2.** A hard fork is an update that creates blocks and transactions incompatible with the existing protocol, making them invalid under the previous rules. However, blocks and transactions adhering to the rules of the prior protocol remain valid under the new protocol.

The success of a hard fork, like a soft fork, relies on user adoption rates. In contrast, if less than $50\%$ of users adopt the new protocol, the new blocks become orphaned. Conversely, if more than $50\%$ adopt the new rule, two chains emerge, each with a distinct canonical chain.

**Example 2.2.** Increasing the block size from $1$ MB → $2$ MB requires a hard fork because nodes that do not upgrade their software will not accept the new blocks, as they will, in most cases, exceed the size limit set by their protocol.

Table 2.1 presents a comparative overview of these methods.

| **Soft Fork** | **Hard Fork** |
|---|---|
| Tightening the rules (e.g., $1$ MB → $0.5$ MB) | Expanding the rules (e.g., $1$ MB → $2$ MB) |
| Backwards compatible | Not backward compatible |
| Old nodes accept new blocks | Old nodes don't accept new blocks |

Table 2.1: Summarized difference between a soft fork and a hard fork.

Ethereum's update process is overseen by the *Ethereum Foundation*, which typically implements changes via hard forks due to their practicality [17]. A hard fork in Ethereum typically comprises one or more *Ethereum Improvement Proposals* (EIPs) [18]. Each EIP undergoes several phases, during which people can discuss and propose enhancements before it is either accepted for inclusion in a hard fork or rejected as Ethereum's direction evolves.

## 2.4 Ethereum Virtual Machine

Compared to many blockchains that only support a distributed ledger, Ethereum operates as a distributed state machine through the *Ethereum Virtual Machine* (EVM) [19]. The EVM supports powerful functionality through *smart contracts* [20].

A smart contract is a program governed by code that runs on the EVM. It comprises two parts: *code* and *data*. The code defines the logic and rules of the smart contract, and once deployed to the EVM, it cannot be altered. The data represents the contract's state, which can be modified through user interactions, such as updating a variable from $5$ to $10$.

This structure allows smart contracts to automate complex processes and transactions, ensuring they execute exactly as programmed without the need for intermediaries. The code's immutability and the data's dynamic nature make smart contracts a robust tool for building decentralized applications.

Smart contracts on the EVM are written in Turing-complete programming languages, with the most popular being Solidity [21], a language specifically designed for developing smart contracts on Ethereum. Once written, smart contracts are compiled into EVM bytecode and deployed to the Ethereum blockchain, where they can be interacted with by users and other contracts.

The state of the EVM, which includes account balances, contract storage, and other data, is stored in a modified Merkle tree known as the *Merkle Patricia Trie* [22]. This data structure provides efficient $O(\log(n))$ complexity for lookups, inserts, and deletes, ensuring that the state can be quickly accessed and updated as needed.

### 2.4.1 Gas

In the EVM, every operation necessitates computational effort from each participating node in the Ethereum network. Consequently, a potential vulnerability exists where a smart contract could execute an infinite loop, consuming network resources indefinitely. To mitigate this issue and ensure that nodes are incentivized to execute commands on the EVM, every transaction incurs a cost known as *gas* [23]. Each interaction with the EVM carries a fixed gas cost multiplied by the current gas price. This is called the *gas fee* and must be paid regardless of whether the transaction succeeds or fails.

The gas fee comprises two components: the `base fee` and the `priority fee`. The `base fee` is dynamically determined by the protocol and adjusts according to block sizes. In Ethereum, the target for each block is approximately $15$ million gas, with a maximum limit of $30$ million gas. When the gas usage surpasses the target, the `base fee` increases; conversely, when it falls below the target, it decreases. The `base fee` is burned, effectively removing it from circulation. On the other hand, the `priority fee` serves as an incentive for nodes to include transactions in their blocks. Gas fees are denominated in *Ether* and are typically quoted in Gwei or Wei, with $1$ Gwei equal to $10^{-9}$ Ether and $1$ Wei equal to $10^{-18}$ Ether, respectively. The node initiating the transaction bears the responsibility of paying the gas fee.

## 2.5 Digital Signatures

In the Byzantine fault-tolerant model, which is prevalent in blockchain systems, there must be a reliable way to verify that transactions originate from the correct participant. Digital signature schemes provide this verification by ensuring several key properties [24]:

- **Authenticity:** A valid signature confirms that the signer intentionally signed the associated message.
- **Unforgeability:** Only the signer can produce a valid signature for the associated message.
- **Non-reusability:** The signature of one message cannot be reused for another message.
- **Non-repudiation:** The signer cannot deny having signed a message that has a valid signature.
- **Integrity:** Ensures that the message's contents have not been altered.

Digital signatures use cryptographic techniques to bind a participant's identity to the transactions they authorize. When a participant signs a transaction, they use their private key to create a signature that others can verify using the corresponding public key. This cryptographic linkage ensures that only the participant with the correct private key could have created the signature.

Ethereum and Bitcoin utilize the *Elliptic Curve Digital Signature Algorithm* (ECDSA) [25] for signing transactions. ECDSA offers robust security by leveraging the mathematical properties of elliptic curves. This makes it computationally infeasible for attackers to derive private keys from public keys or signatures.

### 2.5.1 BLS Signatures

For blockchains that require storing large amounts of digital signatures in blocks, using ECDSA poses challenges due to the fixed size of each signature. As participant counts increase, accommodating these signatures becomes impractical. In such cases, blockchains, especially those utilizing proof-of-stake, often opt for the *Boneh-Lynn-Shacham* (BLS) digital signature scheme [26] due to its support for signature aggregation. This enables a set of signatures to be condensed into a single aggregated signature without any increase in size compared to an individual signature.

The BLS digital signature scheme utilizes two elliptic curve groups, $G_1$ and $G_2$, defined over finite fields, each with an order of $r$ and leverages a unique property of elliptic curves known as *pairing* [27]. Notably, $G_1$ offers faster operations and a more compact representation than $G_2$ due to its definition over a smaller field. The generators for $G_1$ and $G_2$ are denoted as $g_1$ and $g_2$ respectively.

1. The secret key ($sk$) is a number between $1$ and $r$, with a length of $32$ bytes as an unsigned integer.

2. The public key ($pk$) is a point on the $G_1$ curve obtained by scalar multiplication of the secret key and the generator $g_1$, denoted as $pk \times g_1$. Its compressed serialized form occupies $48$ bytes.
3. During the signing process, the message ($m$) is mapped to a point on $G_2$, denoted as $H(m)$, where $H()$ is a hash function that maps bytes to $G_2$.
4. The signature $\sigma$ is a point on the $G_2$ curve represented as $sk \times H(m)$. Its compressed serialized form occupies $96$ bytes.

The BLS signature scheme offers the following API functionalities [26]:

- $Sign(sk, m) \rightarrow \alpha$: The signing function generates a deterministic signature given a secret key $sk$ and message $m$.
- $Verify(pk, m, \alpha) \rightarrow True/False$: The verification function determines whether the signature $\alpha$ is valid for the given public key $pk$ and message $m$. It outputs $True$ if the signature is valid and $False$ otherwise.
- $Aggregate([\alpha_1, ..., \alpha_n]) \rightarrow \alpha_{agg}$: The aggregation algorithm combines a list of signatures $[\alpha_1, ..., \alpha_n]$ into a single aggregated signature $\alpha_{agg}$.
- $AggregateVerify([pk_1, ..., pk_n], [m_1, ..., m_n], \alpha_{agg}) \rightarrow True/False$: The aggregate verification function verifies whether the aggregated signature $\alpha_{agg}$ is valid for the given public keys $[pk_1, ..., pk_n]$ and messages $[m_1, ..., m_n]$. It outputs $True$ if the aggregated signature is valid for all the public keys and messages and $False$ otherwise.

The *Aggregate* function is simply group multiplication on the $G_2$ group, resulting in the aggregated signature occupying $96$ bytes the same as a single signature. Similarly, public keys can undergo aggregation using the same principle applied in signature aggregation, albeit within the $G_1$ group, resulting in an aggregated public key of $48$ bytes.

The *AggregateVerify* function poses inherent challenges as it necessitates information about every message and its corresponding public key for verifying the correctness of the aggregate signature. This entails allocating space for each included public key and its corresponding message. Furthermore, it demands $n + 1$ pairing operations to validate the aggregated signature, rendering it prohibitively expensive.

A solution proposed in [28, 29] addresses these challenges by reducing the pairings required to verify an aggregated signature over the same message $m$ from $n + 1$ to just $2$ by utilizing the *Verify* function. However, instead of processing a single signature $\alpha$ and public key $pk$, it handles an aggregated signature $\alpha_{agg}$ and an aggregated public key $pk_{agg}$. To implement this type of signature aggregation, the system must be capable of tracking the included public keys in $pk_{agg}$ since distinguishing between a single public key and an aggregated public key is not feasible. If the membership status of all participants is known, one bit per participant can be used to track them. In cases with duplicate participant signatures, additional bits are necessary, as a single bit can only indicate inclusion or exclusion. An end-to-end example is shown in Figure 2.4.

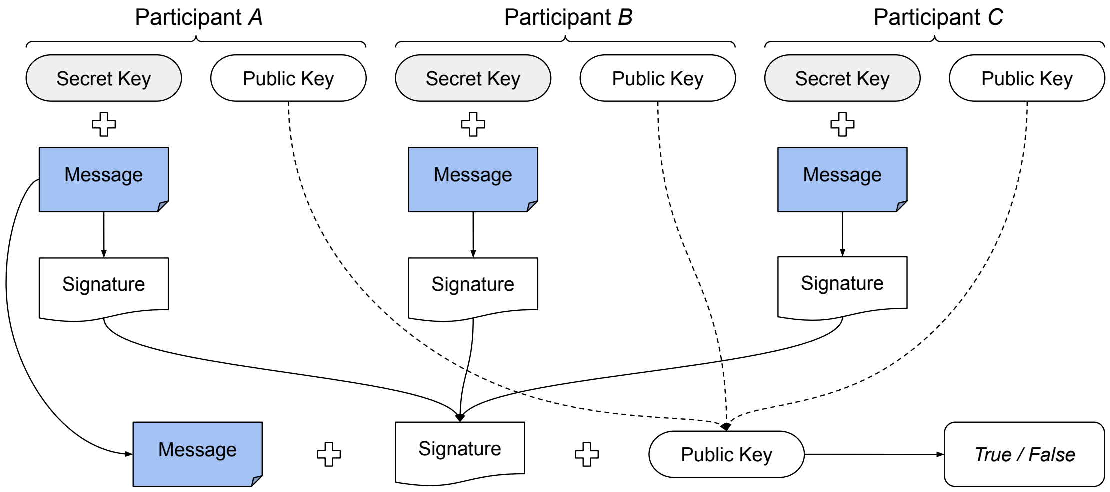

Figure 2.4: End-to-end example.

**Note 2.1.** In the context described above, public keys reside on the $G_1$ curve, while signatures are on the $G_2$ curve. Conversely, it is also feasible to invert this arrangement, with public keys in $G_2$ and signatures in $G_1$, resulting in public keys of $96$ bytes and signatures of $48$ bytes [26]. Implementers can choose the appropriate group based on their specific requirements. For instance, if fast signature aggregation is essential, it is advisable to have signatures in $G_1$. Conversely, if there is a higher frequency of public key aggregation, it is advisable to have public keys in $G_1$. In essence, operations on the $G_1$ group are notably faster than those on $G_2$.

# Chapter 3

# Technical Overview of Ethereum 2.0

Since its initial release, the Ethereum proof-of-stake protocol has undergone four hard forks, each building upon the previous version. All the releases are documented on GitHub through *specifications* [30]. The original release, known as *Phase 0* [31], contains the most comprehensive information regarding the proof-of-stake protocol. Subsequent hard forks, including *Altair* [32], *Bellatrix* [33], *Capella* [34], and *Deneb* [35], introduce specific changes and improvements. To fully understand the evolution of the proof-of-stake protocol up to the latest hard fork, it is necessary to review each specification document chronologically.

This chapter aims to provide technical insight into the proof-of-stake protocol that secures Ethereum's blockchain, commonly referred to as Ethereum 2.0. The content and structure of this chapter draw inspiration from the specifications [30], Vitalik Buterin's annotated specifcations [36], and Ben Edgington's forthcoming book, *"Upgrading Ethereum"* [37].

For readers already acquainted with Ethereum 2.0, we recommend skipping sections covering familiar topics. However, we strongly encourage reading the following sections as they are particularly relevant for understanding Chapter 5.

- In Section 3.1.2, we outline the necessary setup to participate in Ethereum, focusing on client architecture. This overview is essential for understanding the client requirements in our private deployment setup.
- In Section 3.3, provides a detailed exploration of a validator's lifecycle, covering all the steps involved in onboarding validators, a feature our devnet support.
- In Sections 3.4.4 and 3.4.5, we delve into attestation aggregation and how specific validators are selected for this task in Ethereum through an aggregator selection process. Understanding this process is crucial since we test an attack related to aggregator selection.
- In Section 3.5, we examine the networking stack for the consensus layer, including peer discovery, gossiping, and one-to-one communication.

The remaining sections provide a comprehensive overview of Ethereum 2.0, helping readers understand the various components and concepts that enable Ethereum to use proof-of-stake as its consensus protocol. These sections are particularly beneficial for those who do not already have an in-depth understanding of Ethereum 2.0.

## 3.1 The Beacon Chain

In this section, we start by explaining the beacon chain and presenting a timeline of the various hard forks it has undergone. Next, we delve into the client architecture powering Ethereum 2.0. We then explore how the beacon chain measures time using slots and epochs. Then, we discuss the beacon state and its state transition function. Finally, we provide a brief overview of the serialization method for the beacon chain, known as simple serialize, and how objects can be represented using merkleization with a single hash.

### 3.1.1 Timeline and The Merge

On December 1, 2020, Ethereum launched the *beacon chain* [38], which uses proof-of-stake as its consensus protocol. The beacon chain operated alongside the original proof-of-work chain, known as the execution chain, as illustrated in Figure 3.1. Subsequently, on September 15, 2022, Ethereum underwent a significant event known as *the merge*, during which the proof-of-work mechanism was disabled, and proof-of-stake took over as the primary consensus protocol [39]. Before the merge, beacon blocks did not include transactions; they contained only consensus-related data. However, following the merge, blocks from the execution chain were merged into the beacon blocks. These blocks from the execution chain, integrated into the beacon blocks, are commonly referred to as `execution_payload`, containing transactions and header information [40].

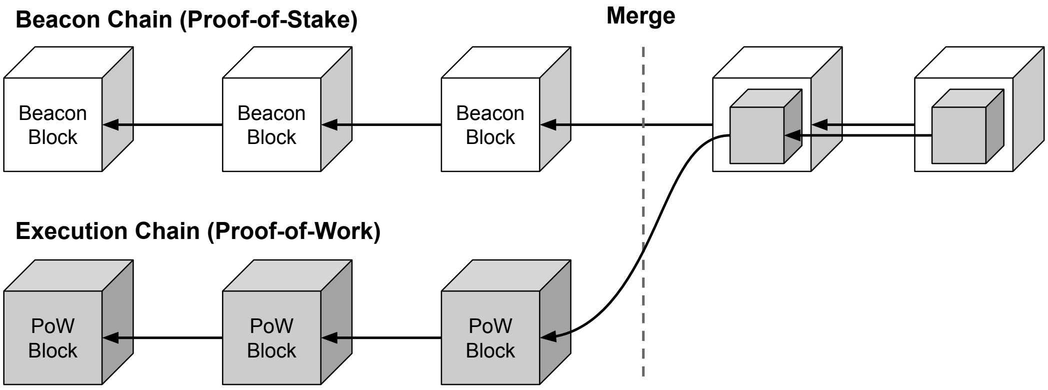

Figure 3.1: An illustration of the merge event that changed the consensus protocol for Ethereum from proof-of-work to proof-of-stake.

Figure 3.2 provides a comprehensive timeline, covering the initial releases of the beacon chain and all subsequent hard forks up to the present date. Notably, the Bellatrix hard fork marks the countdown of the merge event, which occurred nine days after the hard fork.

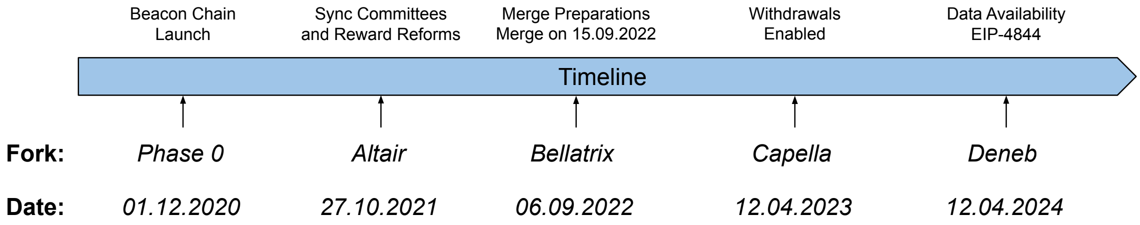

Figure 3.2: The timeline of hard forks related to Ethereum's proof-of-stake protocol.

### 3.1.2 Client Architecture

After transitioning to proof-of-stake, active participation in the Ethereum blockchain involves running two clients: the execution client and the consensus client [41]. The collaboration of these two clients forms a node, empowering users to contribute actively to the security and maintenance of the blockchain. Additionally, users can operate a third client known as the validator client, responsible for managing *validators* who have deposited the requisite amount. The validator's role is to propose blocks and vote on blocks to reach a consensus on the chain. Figure 3.3 visually illustrates the distinction between a standard node and a node operating as a validator.

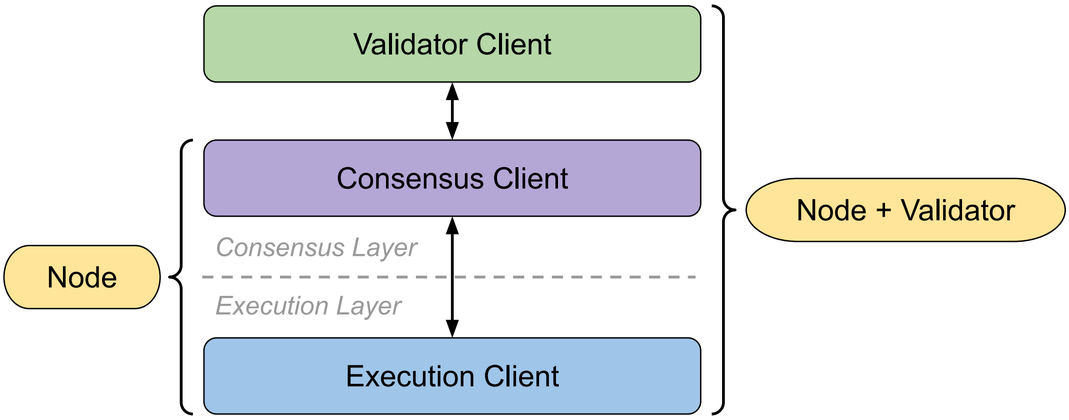

Figure 3.3: Client stack for a node. The validator client is only relevant for nodes that have deposited the required stake.

The two clients operate on different layers: the consensus client operates within the *consensus layer*, while the execution client operates within the *execution layer*. This setup was designed to simplify the transition to proof-of-stake and ensure a clear separation of responsibilities for each client. The clients communicate through a local connection called the *Engine API* [42] to facilitate interaction between the layers. They operate in a leader-follower setup, where the consensus client takes on the role of the leader, actively initiating actions, while the execution client assumes the follower role, responding to requests initiated by the consensus client. Figure 3.4 depicts this interaction, with solid lines denoting requests and dashed lines representing responses. The consensus and execution layers have separate peer-to-peer networks. Users can communicate with an Ethereum node using the

*Beacon API* [43] for the consensus layer and the *Execution API* [44] for the execution layer. The validator client is an optional plugin that is added externally to the Ethereum node and isn't essential for system interaction.

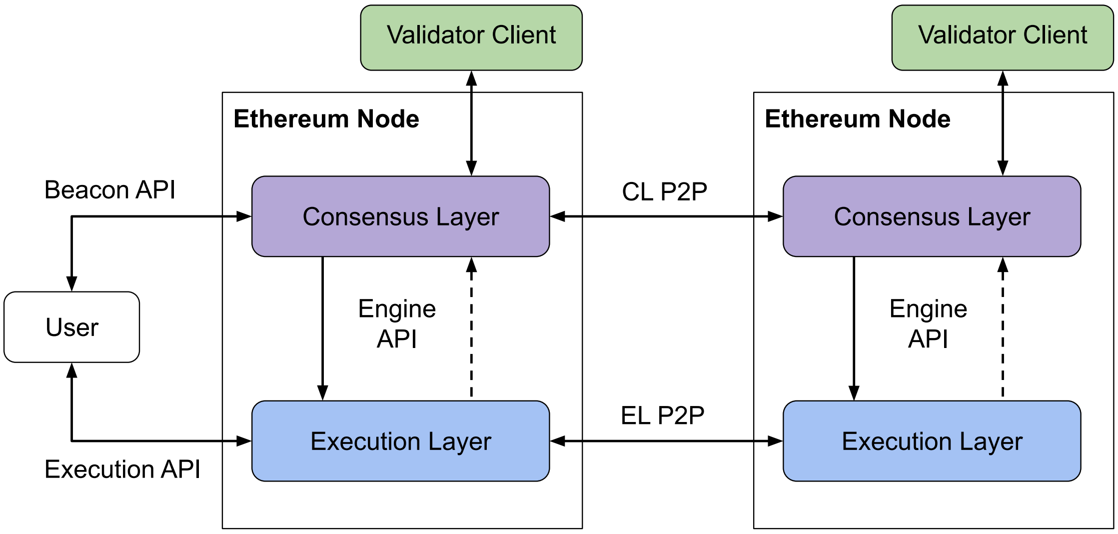

Figure 3.4: Simplified diagram of a coupled execution and consensus client with separate peer-to-peer networks. A user interaction with an Ethereum node occurs through their respective APIs.

### Consensus Client

The consensus client, also called a *beacon node*, actively manages all aspects of consensus logic and the exchange of blocks and consensus votes across its dedicated peer-to-peer network, represented as CL P2P in Figure 3.4. The consensus client undertakes block verification upon receiving blocks on the CL P2P network. During this verification process, it inspects header information and forwards the `execution_payload` (bundle of transactions) to the execution client via the Engine API, awaiting a response. The block is appended to the chain if the header and the `execution_payload` are valid.

### Execution Client

The execution client manages transaction-related tasks such as handling transactions, gossiping transaction information, and managing the EVM. It is not responsible for block building and consensus logic, as the consensus client handles these tasks. The peer-to-peer network associated with the execution client, depicted as EL P2P in Figure 3.4, exclusively gossip transactions and nothing else.

If a user wishes to execute a transaction, they must send it to an execution client. The execution client checks its validity, places it in the mempool if legitimate, and broadcasts the transaction on the EL P2P network.

When the execution client receives a request from the consensus client, it involves either receiving an `execution_payload` or generating one. Upon receiving an `execution_payload`, the execution client validates and executes all the transactions bundled in the payload, sending a status message back to the consensus client indicating whether the payload was valid. The other request type process involves taking transactions from its mempool, executing them, updating the EVM state, and returning the newly generated `exectuion_payload` to the consensus client.

### Validator Client

Nodes can run a validator client alongside the beacon client, enabling them to manage one or multiple validators based on the staked amount. The validator's role is to grow and secure the chain. They receive rewards for active participation in the network but may face penalties if they act maliciously or fail to fulfill their duties adequately. The lifecycle of a validator will be detailed in Section 3.3.

## 3.1.3 Time

Time is inherently imprecise in proof-of-work protocols as blocks emerge when a valid block is successfully mined. However, the proof-of-stake consensus protocol in Ethereum $2.0$ operates differently. The beacon chain uses a notion of time through the use of *slots* and *epochs*. A slot is `SECONDS_PER_SLOT` $(12)$ seconds long, while an epoch consists of `SLOTS_PER_EPOCH` $(32)$ slots, which is equivalent to $6.4$ minutes [31]. Figure 3.5 visually represents slots and epochs.

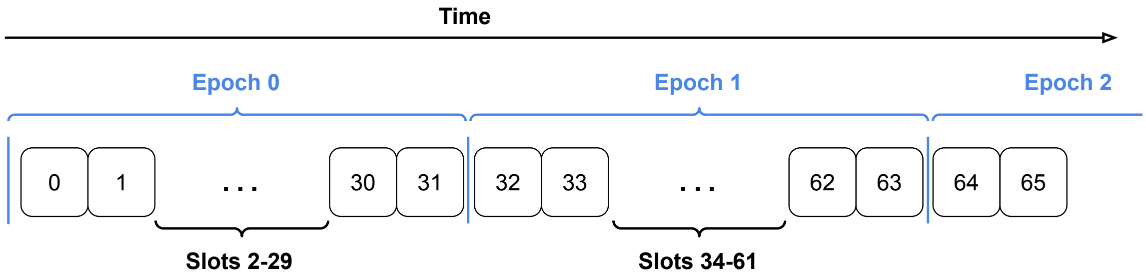

Figure 3.5: Time moves from left to right through slots and epochs. Each slot has the potential to contain a block.

Each slot involves the selection of a single validator, known as the *proposer*, to propose a block. The proposed block is gossiped throughout the peer-to-peer network of the consensus layer. While it's anticipated that there will be a block in each slot, occasional factors like asynchrony or invalid proposals might cause its absence. Each slot can only have one block associated with it, which typically arrives at intervals of `SECONDS_PER_SLOT` ($12$ seconds). If a slot passes without a block proposal, it remains indefinitely devoid of any associated block. Figure 3.6 demonstrates an example where there is a missing block proposer for slot $1$, and the block proposer for slot $3$ experiences some asynchrony issues, which makes the canonical chain $(0,0) \leftarrow (2,1) \leftarrow (4,2) \leftarrow (5,3)$ where the first element is the slot and second element is the block number. The concept and usage of epochs will be discussed in Section 3.2.

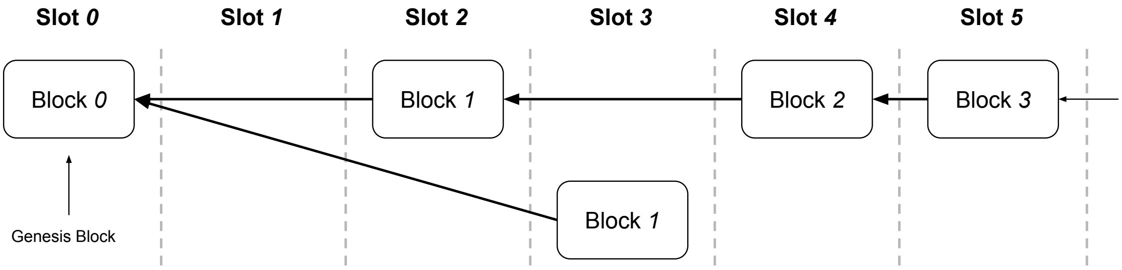

Figure 3.6: Slot $0$ consists of the genesis block (block number $0$). In slot $1$, the block proposer is absent, so block number $1$ is linked with slot $2$ instead. Since the block proposer for slot $3$ has some asynchrony problem, he does not see the block for slot $2$, so he mistakenly thinks his block should be number $1$. Then, the block proposers for slots $4$ and $5$ continue building on the block from slot $2$, resulting in a chain of four blocks over six slots. This sequence of events involves the absence of one block and the reorganization of another block.

**Note 3.1.** To calculate the epoch $j$ of a given slot $i$, the formula `epoch(i)` $= j = \lfloor \frac{i}{C} \rfloor$ is employed, where $C$ is `SLOTS_PER_EPOCH` ($32$). Beacon blocks in epoch $j$ have slot numbers denoted as $jC + k$, where $k$ traverses the set $\{0, 1, ..., C-1\}$. Consequently, the genesis block $B_{genesis}$ holds the slot number $0$ and marks the initial slot of epoch $0$.

### 3.1.4 Beacon State

The goal of every node participating in the beacon chain is to agree on a common object called the *beacon state*, which is a monolithic object containing all the necessary information about the beacon chain [34]. Not everything associated with the beacon state will be relevant to this thesis, so we will introduce parts of it when necessary. For a complete view of the beacon state, refer to Listing D.1 in Appendix D.

To ensure uniformity across nodes regarding the beacon state, block proposers incorporate their updated beacon state into their proposed beacon blocks after running a state transition function. This ensures that all nodes share a consistent view. The beacon state is condensed into a single $32$-byte hash for a minimal increase in block size. To see everything included in a beacon block, check Listings D.3 and D.4 in Appendix D.

#### State Transition

The typical method for updating a blockchain is through a state transition function that runs for each new block added to the chain, as was the case with Ethereum before it transitioned to proof-of-stake. In the beacon chain, however, state transitions are slot-driven rather than block-driven. This means the state transition function is executed for every slot, irrespective of block inclusion. In instances with no associated block for a slot, minimal changes occur to the state.

In addition to slots and blocks, the state transition function also oversees epoch transitions, occurring every `SLOTS_PER_EPOCH` ($32$) slots. Listing D.2 outlines all the tasks carried out during epoch processing, which takes place during the last slot of an epoch [34]. Epoch processing will be frequently discussed in this thesis, and each instance will pertain to one of the tasks listed.

In summary, the beacon chain state transition function comprises three phases.

1. **Slot processing**: It processes the slot regardless of whether a block is associated.
2. **Epoch processing**: It processes the epoch, occurring every `EVERY_PER_EPOCH` (32) slots.
3. **Block processing**: This occurs only for slots with an associated beacon block.

### 3.1.5 Simple Serialize and Merkleization

*Simple Serialize* (SSZ) serves as the serialization method for the beacon chain [45]. It can represent objects of varying complexity, like the beacon state, as strings of bytes. Given its dual usage in communication and consensus protocols on the beacon chain, SSZ must adhere to two key properties. Firstly, when serializing an object of a certain type, the deserialized result should match the original object; this is crucial for the communication protocol. Secondly, two objects of the same type with identical values should serialize to the same SSZ object, ensuring consistency for the consensus protocol. One aspect of SSZ is that it lacks self-description, meaning the expected deserialized object must be known beforehand [46]. This section provides a brief overview of SSZ, covering only the essential aspects. While a detailed understanding of SSZ is unnecessary for this thesis, this overview will help understand various components within the beacon chain.

SSZ supports only three basic types: `unsigned integers`, `bytes`, and `booleans` [47]. Unsigned integers are denoted as `uintN`, where `N` represents the bit count, with `N ∈ [8, 16, 32, 64, 128, 256]`. Meanwhile, `byte` comprises 8 bits, and `boolean` values are either `True` or `False`. Given the simplicity and limited utility of the basic types alone, SSZ introduces composite types that combine multiples of smaller types to accommodate a wider range of use cases. The following composite types exist [47].

- **Vector**: Represents an ordered fixed-length collection of a specific `type` with `N` elements, denoted as `Vector[type, N]` (e.g., `Vector[uint32, N]`).
- **List**: Represents an ordered variable-length collection of a specific `type` up to `N` elements, denoted as `List[type, N]` (e.g., `List[uint32, N]`).
- **Bitvector**: Represents an ordered fixed-length collection of the `boolean` type with `N` bits, denoted as `Bitvector[N]`.
- **Bitlist**: Represents an ordered variable-length collection of the `boolean` type up to `N` bits, denoted as `Bitlist[N]`.
- **Union**: This composite type is not currently utilized in Ethereum.
- **Container**: A container is a collection of values arranged in a specific order and can be of different types. Essentially, it can hold any mix of types, including other containers.

An example of a container is demonstrated in Listing 3.1, which showcases a Python data class containing key-value pairs.

```
class ContainerExample(Container):
    foo: Foo
    bar: Bar
    indices: List[uint32, 64]
```

Listing 3.1: ContainerExample class as defined by the consensus specifications [47].

The types `Foo` and `Bar` serve as type annotations but are not explicitly defined; instead, they represent underlying types such as `uint64` or `Bytes32`. These Python data class containers are frequently used in the consensus specifications [30] and will be used often in this thesis to make things easier to understand.

*Merkleization* is the process of representing an SSZ object as a 32-byte hash, known as a *hash tree root* [48]. This allows nodes in the network to compare beacon blocks, beacon states, and other relevant objects in the consensus layer, ensuring consistency across the network. The process involves taking a list of 32-byte chunks as inputs, which serve as the leaves, and applying the same process as generating a Merkle tree, as discussed in Section 2.1. No restrictions exist on the minimum or maximum number of chunks (leaves) provided. However, Merkleization utilizes zero-padded chunks to ensure that the total number of chunks is rounded to the next whole power of two, creating a complete binary tree.

For basic types or collections of basic types (such as lists and vectors), Merkleization proceeds straightforwardly. However, for containers and collections of composite types, one must recursively calculate the components' hash tree roots before Merkleizing the final hash tree root. Figure 3.7 illustrates the Merkleization process of the `ContainerExample` class from Listing 3.1.

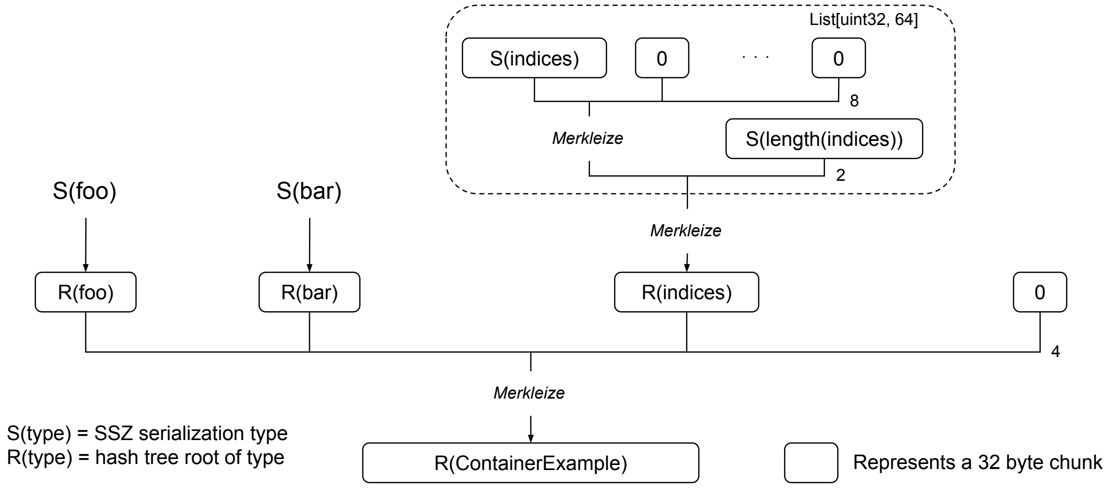

Figure 3.7: Illustrating the steps required to calculate the hash tree root of `ContainerExample`. Here, `S(type)` denotes SSZ serialization of `type`, while `R(type)` indicates Merkleization of `type`. The small digits indicate the number of chunks (leaves).

## 3.2 Gasper the Consensus Protocol

This section explores Ethereum's proof-of-stake consensus protocol, Gasper [49]. We start by exploring how validators convey their worldview through attestations. Subsequently, we investigate the progression of a slot and the timing assumptions that Ethereum relies on. Then, our focus shifts to Casper FFG, the mechanism utilized to achieve finality. Following this, we analyze the fork-choice rule LMD GHOST. Finally, we discuss Gasper, the protocol that integrates Casper FFG and LMD GHOST.

### 3.2.1 Attestations

To come to an agreement on the canonical chain, every validator actively participates in the consensus process by casting a vote for their interpretation of the beacon chain through a mechanism known as an *attestation* [50]. This involves expressing preferences for the validator's selected head of the chain, determined by a fork-choice rule, and casting votes for specific *checkpoints* that play a crucial role in achieving finality.

With the active validator set in Ethereum standing at $985,000$ as of the writing of this thesis [51], broadcasting an attestation for each slot by every validator would result in significant network congestion and processing overhead. Instead of voting for every slot, each validator votes for just one slot within an epoch, while all active validators vote together during each epoch. The definition of an active validator will be explained in Section 3.3.

Validators are effectively distributed across `SLOTS_PER_EPOCH` $(32)$ slots, with each slot corresponding to $\frac{1}{\texttt{SLOTS\_PER\_EPOCH}}$ of the active validators. So, over an epoch, every active validator is expected to have made an attestation declaring their view of the canonical chain. See Figure 3.8 for an illustrative example.

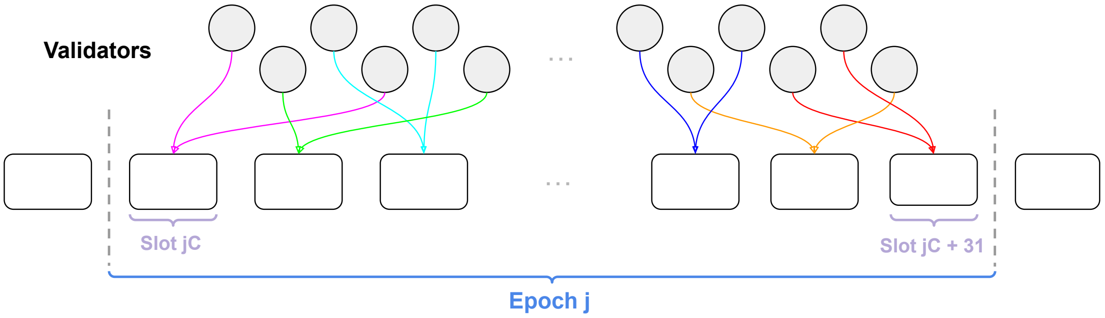

Figure 3.8: All validators vote for an epoch but only $\frac{1}{C}$ of active validators vote for a slot. $C$ is `SLOTS_PER_EPOCH` $(32)$ slots.

Achieving finality typically demands at least two-thirds agreement from the active validator set regarding a shared perspective. Since only a subset of validators cast votes for each slot, establishing a common voting reference becomes crucial. This is where the checkpoint votes come into play. A checkpoint vote marks an *epoch boundary*, coinciding with the first slot of each epoch. For instance, in Figure 3.5, these boundaries correspond to slot $0$, slot $32$, and slot 64. Checkpoint votes are directed at a block linked with an epoch, represented by a `Checkpoint` class, shown in Listing 3.2 [31].

```
class Checkpoint(Container):
    epoch: Epoch
    root: Root
```

Listing 3.2: Checkpoint class as defined by the consensus specifications [31].

The `epoch` field specifies a particular epoch, while the `root` signifies a specific block hash usually associated with the block of the first slot of that epoch. If a block is not found at the epoch boundary, the validator must backtrack in the chain to find the most recent block it observed before the boundary slot. This process is depicted in Figure 3.9, where `LEBB(Slot)` identifies the latest epoch boundary block.

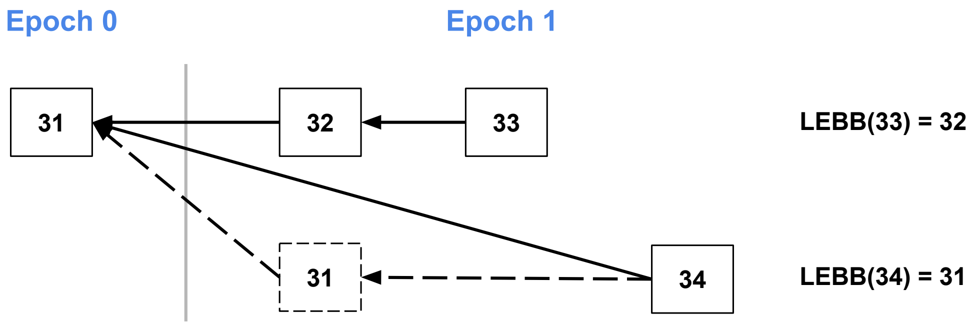

Figure 3.9: Illustration of epoch boundary blocks, with blocks labeled by their corresponding slot numbers. The block labeled 31 in epoch 1 serves as an illustration, emphasizing that block 34 endeavors to locate an epoch boundary block for slot 32 but encounters a challenge in finding one. Consequently, it pulls up block 31 from epoch 0 as an alternative.

The content of an attestation is detailed in the `AttestationData` class, outlined in Listing 3.3.

```
class AttestationData(Container):
    slot: Slot
    index: CommitteeIndex  # the committee the validator belongs to
    # LMD GHOST vote
    beacon_block_root: Root
    # FFG vote
    source: Checkpoint
    target: Checkpoint  # target.epoch == epoch(slot)
```

Listing 3.3: AttestationData class as defined by the consensus specifications [31].

The `slot` indicates the validator's assigned slot, while `index` denotes the committee it belongs to; the usage of a `CommitteeIndex` will be explored in Section 3.4.3. The `beacon_block_root` field indicates the block that the validator sees as the head of the chain after running a fork-choice rule. The `source` and `target` fields are checkpoint votes utilized in calculating finality. Section 3.2.2 will explore the last two fields.

### Progression of a Slot

During a single slot, the time is divided into `INTERVALS_PER_SLOT` (3) segments, each with its designated objective [52]. In the initial segment ($0-4$ seconds), the assigned block proposer generates and disseminates a block across the network. Subsequently, during the second segment ($4-8$ seconds), a committee of validators creates attestations for the current slot, typically by voting on the newly proposed block. If no block is proposed within the first 4 seconds of a slot, validators instead cast their votes for a previous block they deem the head of the chain. In the final interval ($8-12$ seconds), a designated aggregator consolidates attestations and broadcasts the aggregated attestation, ready for inclusion in an upcoming block, typically assigned to slot $i+1$. This aggregation task will be looked at in Section 3.4.4. Figure 3.10 provides a visual representation of the progression of a slot.

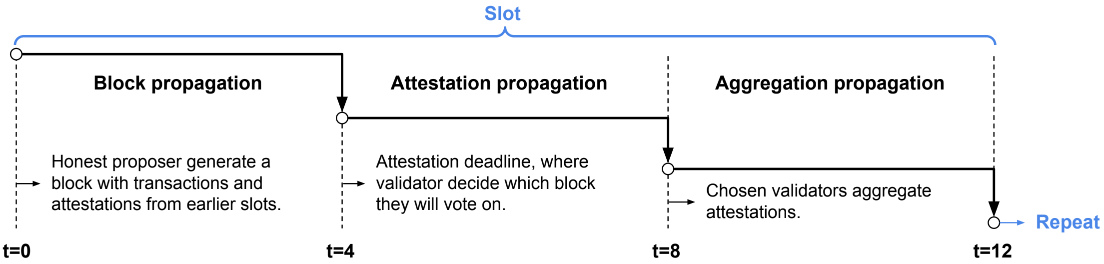

Figure 3.10: The progression of a slot is divided into 3 distinct parts. This process is repeated for every slot.

### Timing Assumption

Ethereum does not assume that validators see the same messages simultaneously or possess an identical view of the chain. Therefore, Ethereum relies on timing assumptions within its consensus mechanics to ensure safety and liveness. The timing assumptions commonly employed in blockchains are *synchronous*, *partial synchronous*, and *asynchronous* [5, 49].

Ethereum relies on the assumption that the internal clocks of nodes are synchronized within the timeframe of a slot (12 seconds) [52]. This synchronization ensures that validators can construct and attest to the latest block without issue.

Regarding the fork-choice rule, nodes ideally should not be more than (`SLOTS_PER_SECOND/INTERVALS_PER_SLOT`) (4) seconds apart. Otherwise, they may experience degraded performance and will not vote for the correct head.

Timing assumptions are not considered in terms of safety properties. This means the blockchain won't confirm conflicting checkpoints, ensuring safety regardless of timing variations. However, for continuous finalization of checkpoints in the blockchain, nodes should be synchronized within a 12-second timeframe.

## 3.2.2 Casper FFG

The protocol responsible for achieving finality (safety) in Ethereum is known as *Casper the Friendly Finality Gadget* (Casper FFG) [49]. Casper FFG operates in two phases: *justifi-*

*cation* and *finalization*, which are inspired by Practical Byzantine Fault Tolerance (PBFT) concepts such as *prepare* and *commit* [53].

Casper FFG functions atop blockchains that can consistently produce new blocks and employ their own fork-choice rule. However, these blockchains lack inherent finality, a gap that Casper FFG aims to address. In alignment with conventional PBFT protocols, Casper FFG necessitates a supermajority vote to achieve finality, corresponding to a quorum of $2/3$. Given that the beacon chain maintains knowledge of the active validator set, as active participation necessitates a stake, it becomes feasible to determine when $2/3$ of the validators have voted for the same checkpoints.

### Justification and Finalization

Casper FFG achieves finality through two rounds of all-to-all communication. In the first round, validators cast their votes to justify a particular checkpoint $C$, corresponding to the `target` field in Listing 3.3. If the validators observe a supermajority of votes for checkpoint $C$ as the `target`, they update checkpoint $C$ to justified. As there is no assurance that other validators have achieved a supermajority for checkpoint $C$, this justification process is a local property.

The second round aims to finalize checkpoint $C$, corresponding to the `source` field in Listing 3.3. Validators voting for checkpoint $C$ as the `source` indicate that they have received a supermajority of votes in favor of checkpoint $C$ as the `target` during the first round and wish to verify this view across the network in the second round. Upon receiving a supermajority of votes designating checkpoint $C$ as the `source`, validators finalize checkpoint $C$ and commit never to revert this decision. Finalization represents a global property, ensuring that once a checkpoint is finalized, honest validators will never revert it.

In summary, justifying a checkpoint entails a personal commitment never to revert it, while finalizing a checkpoint signifies a global commitment never to revert it [54].

Each round lasts an epoch (32 slots). Therefore, the protocol requires two epochs to finalize a checkpoint, corresponding to $12.8$ minutes. Every validator includes a `source` and `target` checkpoint in their `AttestationData` class, indicating the aim of finalizing the `source` checkpoint and justifying the `target` checkpoint. This vote is typically represented as a link in the form of $C_s \xrightarrow{V} C_t$, where $C_s$ represents the source checkpoint and $C_t$ represents the target checkpoint. The `source` checkpoint vote is always the latest justified (`LJ`) checkpoint that the validator has seen, while the `target` checkpoint is the latest epoch (`LE`) boundary. Combining the `source` and `target` votes into a single message, a checkpoint can be finalized every $6.4$ minutes if a supermajority is achieved. For a link $C_s \xrightarrow{V} C_t$ to be valid, $C_s$ must be an ancestor of $C_t$. Otherwise, the vote would contradict the validator's personal commitment to never revert a justified checkpoint.

Even though a round spans $32$ slots (one epoch), validators may receive a supermajority vote after slot $22$, which is $2/3$ of the way through an epoch. However, they won't update their justified and finalized statuses until the epoch is complete, as these calculations are carried out during epoch processing. Only attestations included in blocks are considered when calculating if a supermajority is achieved.

A link is termed a *supermajority link* when it receives a supermajority vote, signifying agreement from $2/3$ of validators. Such a link is represented as $C_s \xrightarrow{J} C_t$ indicating that the target checkpoint $C_t$ should be justified. Figure 3.11 illustrates an instance of a supermajority link, where $C_n$ represented the latest justified checkpoint. Since $C_{n+1}$ failed to garner a supermajority link during epoch $j+1$, it remained ineligible for an update to the justified status. However, during epoch $j+2$, a supermajority link $C_n \xrightarrow{J} C_{n+2}$ was established, consequently allowing $C_{n+2}$ to be updated to justified.

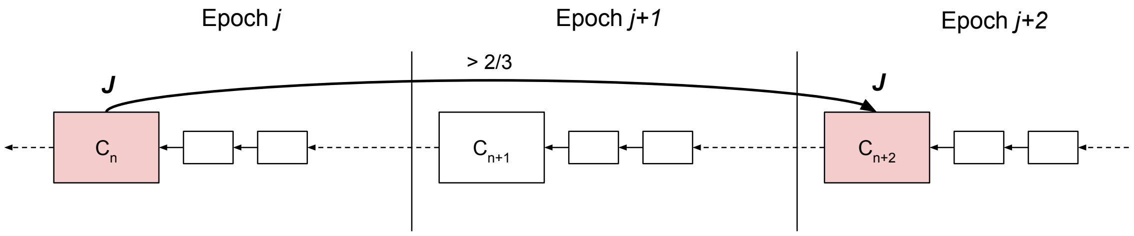

Figure 3.11: During epoch $j+2$, a supermajority link $C_n \xrightarrow{J} C_{n+2}$ justifies $C_{n+2}$. However, as $C_{n+1}$ did not receive a supermajority link during epoch $j+1$, it does not update to justified.

The rationale behind not finalizing $C_n$ in Figure 3.11, despite the presence of a supermajority link $C_n \xrightarrow{J} C_{n+2}$, lies in the requirement that to finalize a checkpoint, the `target` checkpoint must be a direct child to the `source` checkpoint, which is not the scenario here [49]. However, $C_{n+2}$ attains justification simply because of a supermajority link.

In contrast, Figure 3.12 showcases a scenario where a supermajority link $C_n \xrightarrow{J} C_{n+1}$ exists, with the `target` checkpoint $C_{n+1}$ being a direct child of the `source` checkpoint $C_n$. In this case, $C_n$ transitions to a finalized state, and $C_{n+1}$ becomes justified.

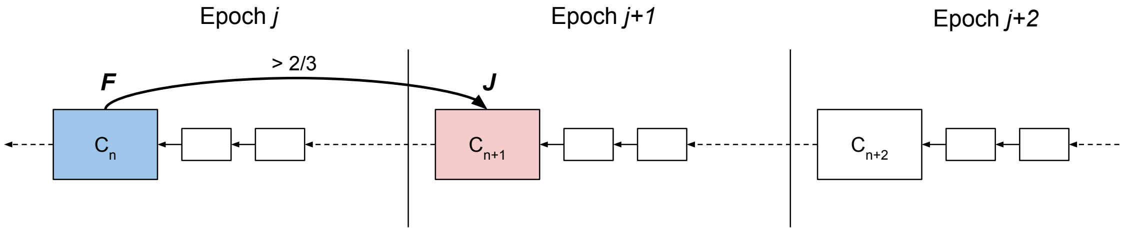

Figure 3.12: During epoch $j+1$, a supermajority link $C_n \xrightarrow{J} C_{n+1}$ results in the finalization of $C_n$, while $C_{n+1}$ is justified.

The scenario depicted in Figure 3.12 is referred to as *1-finality*, indicating that the `source` checkpoint and the `target` checkpoint are precisely one epoch apart. However, this concept can be extended to what is known as *k-finality* if there is an integer $k \geq 1$ and there exist adjacent checkpoints $C_j, ..., C_{j+k}$ such that checkpoints $C_j, C_{j+1}, ..., C_{j+k-1}$ are all justified and there exists a supermajority link $C_j \xrightarrow{J} C_{j+k}$, then checkpoint $C_j$ will be finalized [49].

**Note 3.2.** One reason for voting on checkpoints (block, epoch) rather than individual blocks is to accommodate missing blocks within epochs. For example, in epoch $j$, the only proposed block was for slot $jC$, while no other blocks were observed for the remaining slots. Consequently, the `target` checkpoint for epoch $j$ would be $(B_{jC}, j)$, where $B_{jC}$ represents the block for slot $jC$. Since no additional blocks were observed during epoch $j$, the checkpoint $(B_{jC}, j)$ would not be justified.

Suppose the block for the initial slot of epoch $j + 1$ is also missing. In this scenario, all attestations made during epoch $j + 1$ would fail to locate an epoch boundary block for slot $(j + 1) \times C$. Thus, the protocol would backtrack along the chain until it finds a block, which would be the block for slot $jC$. Consequently, the `target` checkpoint for epochs $j$ and $j + 1$ would share the same block $B_{jC}$ but differ in their epochs. This differentiation provides clarity regarding during which epoch a block is justified.

### The Four-Cases in Ethereum

The Ethereum $2.0$ protocol employs a *2-finality* approach, meaning the beacon state monitors the justification status of the four most recent epochs. This design choice is due to Ethereum $2.0$ only recognizing attestations for up to two epochs; attestations older than this are considered invalid. Consequently, checkpoints up to two epochs in the past can become newly justified (and subsequently finalized), resulting in four possible cases as outlined below and illustrated in Figure 3.13 [31, 49].

1. If checkpoints $C_{n-3}, C_{n-2}$ are justified and there is a supermajority link $C_{n-3} \xrightarrow{J} C_{n-1}$, then $C_{n-1}$ will be justified and $C_{n-3}$ will be finalized.
2. If checkpoint $C_{n-2}$ is justified and there is a supermajority link $C_{n-2} \xrightarrow{J} C_{n-1}$, then $C_{n-1}$ will be justified and $C_{n-2}$ will be finalized.
3. If checkpoints $C_{n-2}, C_{n-1}$ are justified and there is a supermajority link $C_{n-2} \xrightarrow{J} C_{n}$, then $C_n$ will be justified and $C_{n-2}$ will be finalized.
4. If checkpoint $C_{n-1}$ is justified and there is a supermajority link $C_{n-1} \xrightarrow{J} C_{n}$, then $C_n$ will be justified and $C_{n-1}$ will be finalized.

Under normal conditions, characterized by minimal asynchrony and high participation, primarily $1$-finality cases (cases $2$ and $4$) are anticipated, with case $4$ occurring most frequently. Instances of $2$-finality (cases $1$ and $3$) arise under special conditions, such as delayed attestations or when the protocol is close to the two-thirds threshold required for checkpoint finalization, necessitating a wait for the next epoch processing to determine if a supermajority has been achieved [49].

**Note 3.3.** The *2-finality* cases are checked sequentially, starting with case $1$, followed by case $2$, and so on. For example, case $2$ might finalize $C_{n-2}$, and then case $4$ could immediately finalize $C_{n-1}$.

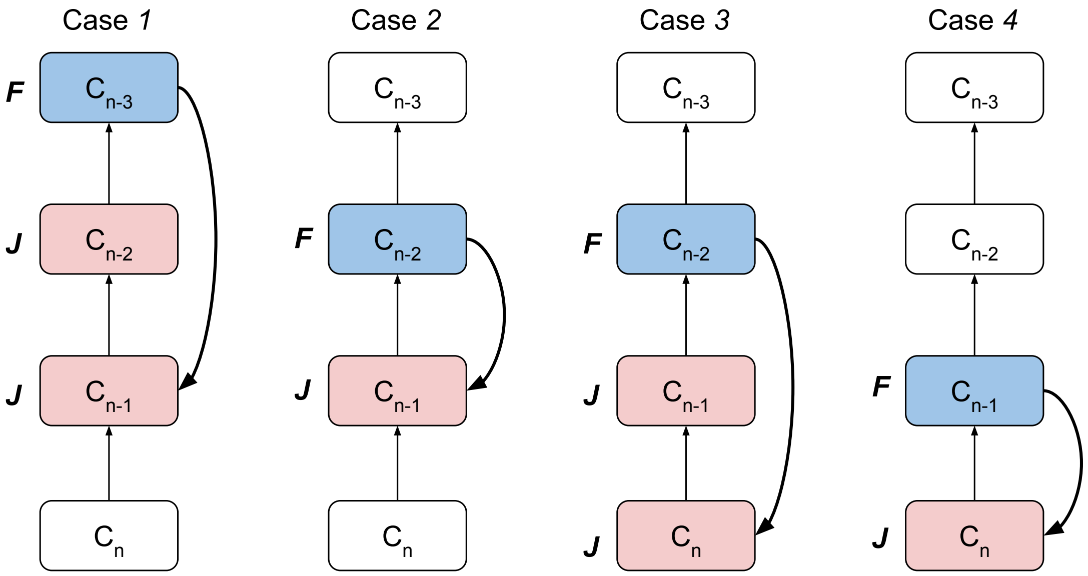

Figure 3.13: The four cases of 2-finality. In each case, the supermajority link finalizes the source checkpoint and justifies the target checkpoint. Inspired by [49].

### 3.2.3 LMD GHOST

The fork-choice rule implemented in Ethereum's proof-of-stake protocol is called *Latest Message Driven Greedy Heaviest Observed SubTree* (LMD GHOST) [49]. LMD GHOST is an adaptation of the GHOST protocol; however, it utilizes validators' attestations instead of blocks to calculate subtrees and find the head block. LMD GHOST only considers the latest attestation from each validator, hence the term *latest message driven*.

In Ethereum, LMD GHOST and Casper FFG work together to ensure a secure blockchain. LMD GHOST is responsible for growing the chain by handling block proposals and attestations, while Casper FFG, built on top of LMD GHOST, provides finality by periodically finalizing blocks to protect against chain reorganization and ensure consistency.

Unlike Casper FFG, which exclusively considers attestations included in blocks, LMD GHOST also considers attestations received directly through the gossip protocol. Consequently, the head discovered after executing LMD GHOST relies on each node's local view of the chain, as there is no assurance that nodes have observed the same blocks and attestations.

Each node records the latest attestation from every validator using a `Store` object [52]. Once a valid attestation from a validator is recorded, it is stored indefinitely in `Store` and continuously contributes to calculating subtrees in LMD GHOST. Only when a validator submits a newer attestation will his record be updated. To be deemed valid for inclusion in the `Store`, an attestation must originate from the previous or current epoch and pertain to a block that actually exists (i.e., the node has observed it).

To illustrate how LMD GHOST operates, refer to Figure 3.14, where the blue blocks represent the canonical chain after running LMD GHOST. In this instance, each attestation has a vote of 1, but in the actual implementation, it depends on the stake size. Nonetheless, the fundamental principle remains the same. In this scenario, validators are expected to attest to the leaf block in the canonical chain, while a block proposer should build upon it. The leaf block identified after executing LMD GHOST will serve as the `beacon_block_root` vote in `AttestationData`.

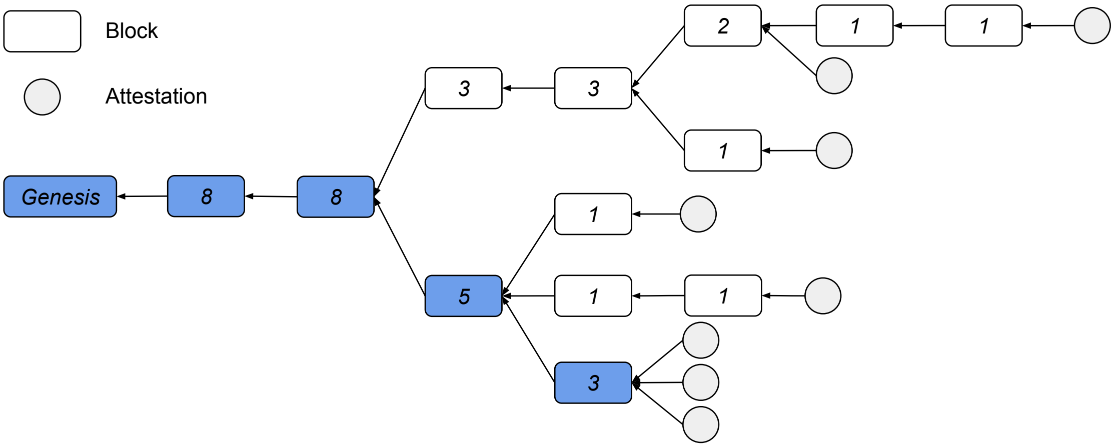

Figure 3.14: An example of the LMD GHOST fork-choice rule. In this example, the blue blocks will be the canonical chain. Inspired by [49].

### 3.2.4 Gasper

The Ethereum consensus protocol, known as *Gasper*, combines Casper FFG and LMD GHOST. Casper FFG is designed to guarantee the protocol's safety, whereas LMD GHOST is primarily focused on ensuring liveness. Validators vote simultaneously for each of these consensus mechanisms in their attestation, which was highlighted in Listing 3.3 where `beacon_block_root` is the LMD GHOST vote and the `source` and `target` checkpoint votes are for Casper FFG.

In Gasper, the most significant changes pertain to LMD GHOST, particularly in terms of which blocks it deems viable for the head. Unlike the traditional LMD GHOST algorithm, which considers all blocks from genesis onwards, Gasper's LMD GHOST algorithm adjusts its considerations. With Casper FFG providing finality to the blockchain, there is no necessity to consider blocks preceding the last finalized checkpoint, given the global commitment to never revert it. As the last justified checkpoint represents a personal commitment to avoid reversion, nodes initiate from this point and only concern themselves with branches ascending from this juncture onward. For an illustrative example, see Figure 3.15.

For further insights into Gasper and its implementation specifics within Ethereum, refer to [55, 56].

**Note 3.4.** During the genesis epoch, which spans slots $[0, 31]$, votes for Casper FFG do not directly influence protocol decisions but still impact the fork-choice rule. Validators active during this period receive full rewards and are incentivized to attest [50].

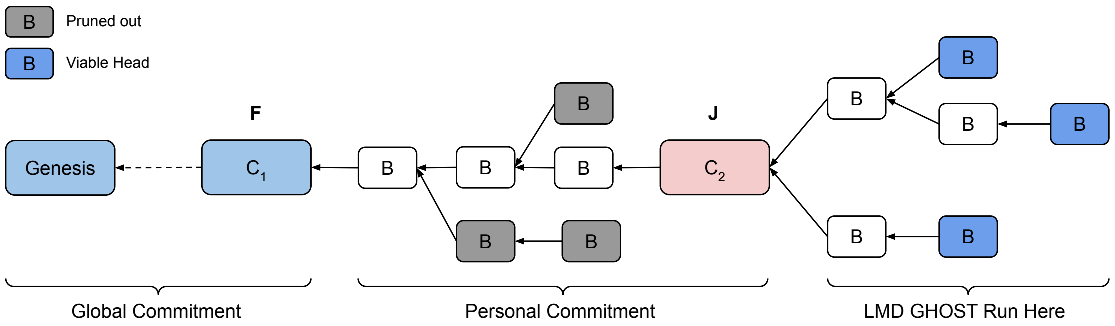

Figure 3.15: The global commitment applies to the finalized checkpoints, which include all blocks from the genesis block up to checkpoint $C_1$. The personal commitment refers to a validator's pledge not to reverse the justified checkpoint $C_2$. The black-grey blocks are pruned and will not be part of the main chain. The blue blocks are the potential head blocks that the LMD GHOST algorithm can choose.

### Slashing

As each validator generates precisely one attestation per epoch and has its stake locked up, Ethereum 2.0 defines slashing conditions to penalize malicious behavior. A slashed validator will have some or all of its stake removed.

**Definition 3.1.** A validator is slashed if one the following conditions is violated [49].

(S1) No validator makes two distinct attestations $\alpha_1, \alpha_2$ with $\texttt{epoch}(\alpha_1) = \texttt{epoch}(\alpha_2)$.

(S2) No validator makes two distinct attestation $\alpha_1, \alpha_2$ with

$$\texttt{epoch}(\texttt{LJ}(\alpha_1)) < \texttt{epoch}(\texttt{LJ}(\alpha_2)) < \texttt{epoch}(\texttt{LE}(\alpha_2)) < \texttt{epoch}(\texttt{LE}(\alpha_1)).$$

(S3) No validator proposes two distinct blocks $b_1, b_2$ with $\texttt{slot}(b_1) = \texttt{slot}(b_2)$.

The first condition (S1) prohibits validators from issuing multiple attestations with differing `AttestationData`. The second condition (S2) is pertinent to Casper FFG and prevents validators from attesting to a `source` and `target` that surrounds another attestation from the same validator. The final condition (S3) prohibits block proposers from engaging in equivocation.

### Inactivity Leak

Given Ethereum 2.0's emphasis on liveness over safety, the protocol continues to generate blocks even when checkpoints remain unfinalized. In cases where there hasn't been a finalized checkpoint for `MIN_EPOCHS_TO_INACTIVITY_PENALTY` (4) epochs [31], an emergency measure known as the *inactivity leak* is triggered to restore finality to the protocol.

The inactivity leak uses a scoring mechanism called the *inactivity score* [32, 57] to penalize inactive validators. This score is maintained individually for each validator in the beacon state and is updated during epoch processing.

Figure 3.16 illustrates how the inactivity score is adjusted for validators, with *active* denoting those who submitted a timely attestation. During epoch processing, validators may have a portion of their stake deducted due to their inactivity score. These penalties increase exponentially as their inactivity score increases. Upon re-finalizing a checkpoint, the inactivity leak is turned off, indicating that the active validators now possess $2/3$ of the total stake.

Even after deactivating the inactivity leak, validators who remained offline throughout the period will continue to face penalties in subsequent epochs until their inactivity scores reach $0$, representing the minimum score a validator can have.

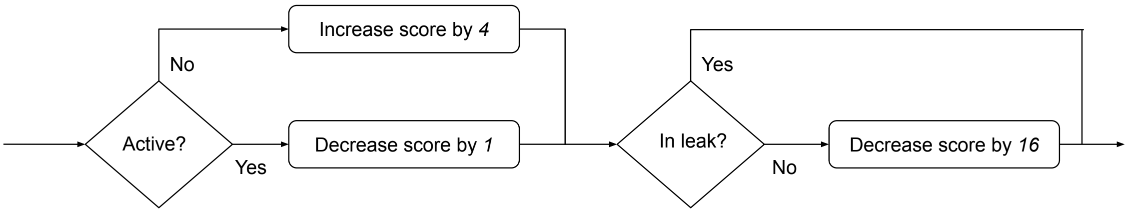

Figure 3.16: Flow diagram of how the `inactivity_score` is updated for each validator during epoch processing. Inspired by [57].

## 3.3 Validator Lifecycle

This section provides an overview of the lifecycle of a validator in Ethereum 2.0. We start by defining a validator per the consensus specifications and explaining how it is stored in the beacon state. Then, we discuss the rate-limiting mechanisms for activations and exits of validators. Subsequently, we explore both the deposit and withdrawal processes. Finally, we summarize all the aspects of a validator.

### 3.3.1 Validator

At the heart of Ethereum 2.0 are the validators who are responsible for tasks like proposing blocks and making attestations. The core details about validators in Ethereum are encapsulated within a specific `Validator` class, shown in Listing 3.4 [31].

```
class Validator(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Bytes32  # Commitment to pubkey for withdrawals
    effective_balance: Gwei
    slashed: boolean
    # Status epochs
    activation_eligibility_epoch: Epoch  # When criteria for activation were met
    activation_epoch: Epoch
    exit_epoch: Epoch
    withdrawable_epoch: Epoch  # When validator can withdraw funds
```

Listing 3.4: Validator class as defined by the consensus specifications [31].

The `pubkey` field serves as a unique identifier for validators and is used to verify their signature. It contains a BLS public key, the digital signature scheme used for the consensus layer. The `withdrawal_credentials` field is the withdrawal address, which indicates the destination for a validator's rewards and remaining balance upon exiting. The `effective_balance` denotes the effective balance of a validator. The `slashed` field is a boolean value indicating whether the validator has been slashed. The remaining four fields are utilized in the activation and exiting processes by specifying the epochs for these events. When a validator is initialized, all status epochs are set to `FAR_FUTURE_EPOCH` ($2^{64} - 1$), a predefined constant in Ethereum. This is the default value for validators' status epochs until explicitly defined [31]. A simplified overview of a validator's lifecycle is depicted in Figure 3.17.

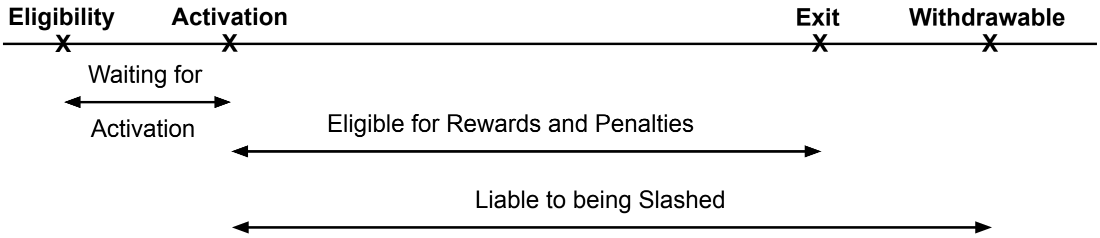

Figure 3.17: An overview of the status epochs in the `Validator` class 3.4.

An *active validator*, introduced briefly in Section 3.2.1, is formally defined as a validator whose `activation_epoch` $\leq$ `current_epoch` $<$ `exit_epoch` [31] (eligible for rewards and penalties in Figure 3.17). Validators who meet this criterion are eligible to participate as validators, generate attestations, and qualify for selection as block proposers.

### Registry

In Ethereum 2.0, the beacon state maintains a registry of all validators [31]. Each validator is assigned a unique `ValidatorIndex` (`uint64`) directly linking them to an `Validator` class within the `validators` field in the `BeaconState` class, as illustrated in Listing 3.5.

```
class BeaconState(Container):
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
    # Inactivity [New in Altair]
    inactivity_scores: List[uint64, VALIDATOR_REGISTRY_LIMIT]
    # Participation [Modified in Altair]
    prev_epoch_participation: List[ParticipationFlags, VALIDATOR_REGISTRY_LIMIT]
    cur_epoch_participation: List[ParticipationFlags, VALIDATOR_REGISTRY_LIMIT]
```

Listing 3.5: BeaconState class as defined by the consensus specifications [31, 32]. Only the relevant fields are included.

This approach enables easy access to a specific validator's state information by simply knowing its `ValidatorIndex`. Including the `ValidatorIndex` rather than the entire `Validator` class in messages significantly minimizes the message size. The `VALIDATOR_REGISTRY_LIMIT` ($2^{40}$) [31] establishes the maximum allowable number of validators. In the current design, validators are only added to the registry without any removal, regardless of whether they are active. As a result, the `VALIDATOR_REGISTRY_LIMIT` sets a cap of $1.1$ trillion validators. Given that generating a validator requires a minimum deposit of `MIN_DEPOSIT_AMOUNT` ($1$ ETH) [31], reaching this limit of $1.1$ trillion validators is not anticipated to be an issue in the foreseeable future.

Within the `BeaconState` class, the fields `balances` and `inactivity_score` are likewise associated with a distinct `ValidatorIndex`, albeit stored in separate registries. This separation stems from these fields being updated every epoch, unlike the `Validator` class, which undergoes less frequent updates. When computing the beacon state root, only the modified portions need to be recalculated, while the data of unchanged parts can be cached. Therefore, having data almost guaranteed to be updated on a per-epoch basis in their registries reduces the necessary calculation required. The `inactivity_score` is linked to the inactivity leak explored in Section 3.2.4.

The participation fields indicate which validators have provided timely attestations. These fields use flags to signal whether validators have submitted timely votes for the head, source, and target in `AttestationData`. Such information is crucial for calculating their rewards and confirming if a supermajority link was reached for an epoch. Storing data for two epochs is necessary because validators can have their attestations for epoch $j$ included up to the final slot of epoch $j + 1$.

### Balances

The beacon chain tracks two balance records for each validator: their *actual balance* and *effective balance* [58]. The actual balance corresponds to the `balances` field in the `BeaconState` class, while the effective balance is directly stored inside the `Validator` class. The actual balance for a validator is straightforwardly calculated using Equation 3.1.

$$\text{actual balance} = \text{deposits} + \text{rewards} - \text{penalties} - \text{withdrawals} \tag{3.1}$$

The actual balance of validators is updated during epoch processing and is measured with high precision, calculated in Gwei, which represents $10^{-9}$ ETH and can be any amount of Gwei. On the other hand, the effective balance is also calculated in Gwei but is limited to whole multiples of `EFFECTIVE_BALANCE_INCREMENT` ($1$ ETH) [31], derived from the actual balance. The effective balance underpins nearly all validator calculations. When determining if a validator can be activated, the effective balance is the focal point of consideration. When voting through attestations, the effective balance is considered for the LMD GHOST vote and is utilized in the Casper FFG calculations. The Ethereum protocol sets a limit on the maximum number of increments, `MAX_EFFECTIVE_BALANCE` ($32$ ETH) [31], that a validator can possess.

**Example 3.1.** For instance, if a node desires to stake $N$ ETH, where $N$ is a multiple of

`MAX_EFFECTIVE_BALANCE` and $N > 1$, the node must have $N$ validators to attain an effective balance of $N \times$ `MAX_EFFECTIVE_BALANCE`.

#### Updating Effective Balance

At the conclusion of each epoch (epoch processing), a function assesses whether validators' effective balances require adjustment based on their actual balances. This process incorporates hysteresis to decrease the frequency of adjustments resulting from minor fluctuations in the actual balance, with the goal of minimizing the need to recalculate the hash tree root of the `Validator` class. Specifically, a validator's effective balance remains unchanged until their actual balance changes by at least $0.5$ ETH [58]. If the actual balance is less than the effective balance minus $0.25$ ETH, the effective balance is decremented. Conversely, if the actual balance exceeds the effective balance plus $1.25$ ETH, the effective balance is incremented.

**Example 3.2.** Two validators, $A$ and $B$, have effective balances of $31$ and $32$, respectively. For validator $A$ to increase its effective balance to $32$, its actual balance must be $\geq 32.25$. On the other hand, for validator $B$ to decrease its effective balance to $31$, its actual balance must be $\leq 31.75$.

#### Ejection

If a validator's effective balance drops to or below `EJECTION_BALANCE` ($16$ ETH), the validator is automatically ejected from the system [31, 57]. This serves a dual purpose: it protects an offline validator that may have lost its key from losing all its ETH, and it aids in restoring $2/3$ of the total effective stake back to active validators. Regarding effective balance calculation, a validator is queued for ejection as soon as its actual balance drops below $16.75$ ETH.

### 3.3.2 Rate-Limiting Activations and Exits

Validators can leave the system, while new validators can join anytime. To maintain stability within the validator set between two checkpoints used to calculate if a supermajority link has been achieved, Ethereum enforces restrictions on the number of validators allowed to join or exit the system through a rate-limiting mechanism. In principle, Ethereum does not want the active validator set to change rapidly.

We will look at the process for joining the system, which is analogous to exiting. Validators with an effective balance of `MAX_EFFECTIVE_BALANCE` that have not yet been activated are added to an activation queue. During epoch processing, a limited number of validators, determined by a *churn limit*, are dequeued from the activation queue following a first-in, first-out (FIFO) order. The churn limit controls the rate at which validators are activated per epoch [59] and is determined by Equation 3.2 and 3.3.

$$\texttt{churn\_limit} = \texttt{max(MIN\_PER\_EPOCH\_CHURN\_LIMIT,}\ \left\lfloor \frac{|V_{active}|}{\texttt{CHURN\_LIMIT\_QUOTIENT}} \right\rfloor) \quad (3.2)$$

$$\texttt{churn\_limit} = \texttt{min(churn\_limit, MAX\_PER\_EPOCH\_ACTIVATION\_CHURN\_LIMIT} \quad (3.3)$$

For Equation 3.2, `MIN_PER_EPOCH_CHURN_LIMIT` is $4$, `CHURN_LIMIT_QUOTIENT` is $2^{16}$, and $|V_{active}|$ represents the number of active validators [31, 35]. The `MIN_PER_EPOCH_CHURN_LIMIT` ensures a minimum number of validators can join the system, particularly when the number of active validators is low. This mechanism guarantees that new validators will eventually be able to join the system. For Equation 3.3, the `MAX_PER_EPOCH_ACTIVATION_CHURN_LIMIT` ($8$) determines the maximum amount of validators eligible to join or exit the system on a per-epoch basis [35].

### 3.3.3 Lifecycle

We will now explore the lifecycle of a validator in a step-by-step manner based on the information provided in [50, 59], with Figure 3.18 showing a visual presentation of the steps. All the validator fields that will be checked and updated below are stored in the `Validator` class from Listing 3.4.

1. **Deposited:** The validator has initiated a transaction of $32$ ETH to a specific smart contract on the execution layer. Subsequently, the validator has been added to the beacon state after undergoing a deposit process, which will be discussed in Section 3.3.5.

2. **Eligiblity:** During epoch processing, validators are evaluated to see if they can join the activation queue. To qualify, a validator must have an `effective_balance` equal to `MAX_EFFECTIVE_BALANCE` ($32$ ETH) and their `activation_eligibility_epoch` must be set to `FAR_FUTURE_EPOCH`. If they meet both conditions, their `activation_eligibility_epoch` is updated to `current_epoch` $+ 1$, and they are added to the end of the activation queue.

3. **Activation Queue:** During epoch processing, validators in the activation queue undergo checks to determine if they can be activated. To be eligible for activation, a validator must have its `activation_eligibility_epoch` finalized and the `activation_epoch` set to `FAR_FUTURE_EPOCH`. If both of these conditions are met, validators up to the churn limit will have their `activation_epoch` set to `current_epoch` + `MAX_SEED_LOOKAHEAD` $+ 1$ in a FIFO order. The `MAX_SEED_LOOKAHEAD` ($4$ epochs $\approx 25.6$ mins) [31] is related to an attack on the randomness accumulator and will be looked at in Section 3.4.2.

4. **Activated:** After activation, the validator will provide attestations on a per-epoch basis and is eligible for selection as a block proposer. Most validators will remain in the active state for an extended period to accumulate rewards.

5. **Exit:** There are three ways a validator can exit the active validator set: voluntary exit, ejection, and being slashed. Regardless of the method, all validators must undergo an *initiate exit* process, which unfolds as follows: The validator is placed in an exit queue akin to the activation queue. Validators dequeued from the exit queue, up to the churn limit, have their `exit_epoch` updated to `current_epoch + MAX_SEED_-LOOKAHEAD`. Subsequently, the `withdrawable_epoch` is set to `exit_epoch + MIN_VAL-IDATOR_WITHDRAWABILITY_DELAY` ($2^8$ epochs $\approx$ 27 hours) [31]. The rationale behind the withdrawable delay is to ensure that validators who engage in misconduct can still be identified and penalized even after they exit the system but still have their stake locked up.

    a) **Voluntary Exit:** This occurs when a validator initiates a voluntary exit, indicating that it wants to stop being a validator. Before a validator can initiate a voluntary exit, it must have been active for at least `SHARD_COMMITTEE_PERIOD` ($2^8$ epochs $\approx 27$ hours) [31].

    b) **Ejection (Insufficient Balance):** This happens when the `effective_balance` of an validator drops to or below `EJECTION_BALANCE` ($16$ ETH).

    c) **Slashed:** If a validator gets slashed, the `withdrawable_epoch` is set to `exit_epoch` + `EPOCHS_PER_SLASHINGS_VECTOR` ($2^{13}$ epochs $\approx$ 36 days) instead of the usual withdrawable epoch in the initiate exit procedure [31].

6. **Withdrawable:** The validator can now withdraw their stake, officially ending their status as a validator.

Appending an additional epoch to some of the validator's status fields is necessary due to the timing of epoch processing, which occurs at the last slot of an epoch. During this process, calculations are carried out for epoch $j + 1$ while still within epoch $j$. This step is unnecessary for the initiate exit process, as it is handled during block processing rather than epoch processing.

**Note 3.5.** All nodes in the network maintain the activation queue as a separate data structure. However, this isn't the case for the exit queue, which is dynamically recalculated from the exit epochs of all validators stored in the beacon state, ensuring a fixed number can exit per epoch. Upon a validator initiating the exit process, all associated fields are promptly updated, regardless of potential changes in the churn limit in subsequent epochs. For example, in epoch $j$, if the churn limit is $5$ and $200$ validators initiate the exit process during epoch $j$, all of these validators will have their exit epoch determined based on the current churn limit of $5$, even if the churn limit were to change to $4$ or $6$ in the subsequent epoch $j + 1$.

### 3.3.4 Deposit

In Ethereum $2.0$, individuals interested in becoming validators must deposit a stake of $32$ Ether onto the execution layer through a specified smart contract [50]. With Ethereum's dual-layer structure consisting of the execution and consensus layers, a distinction exists

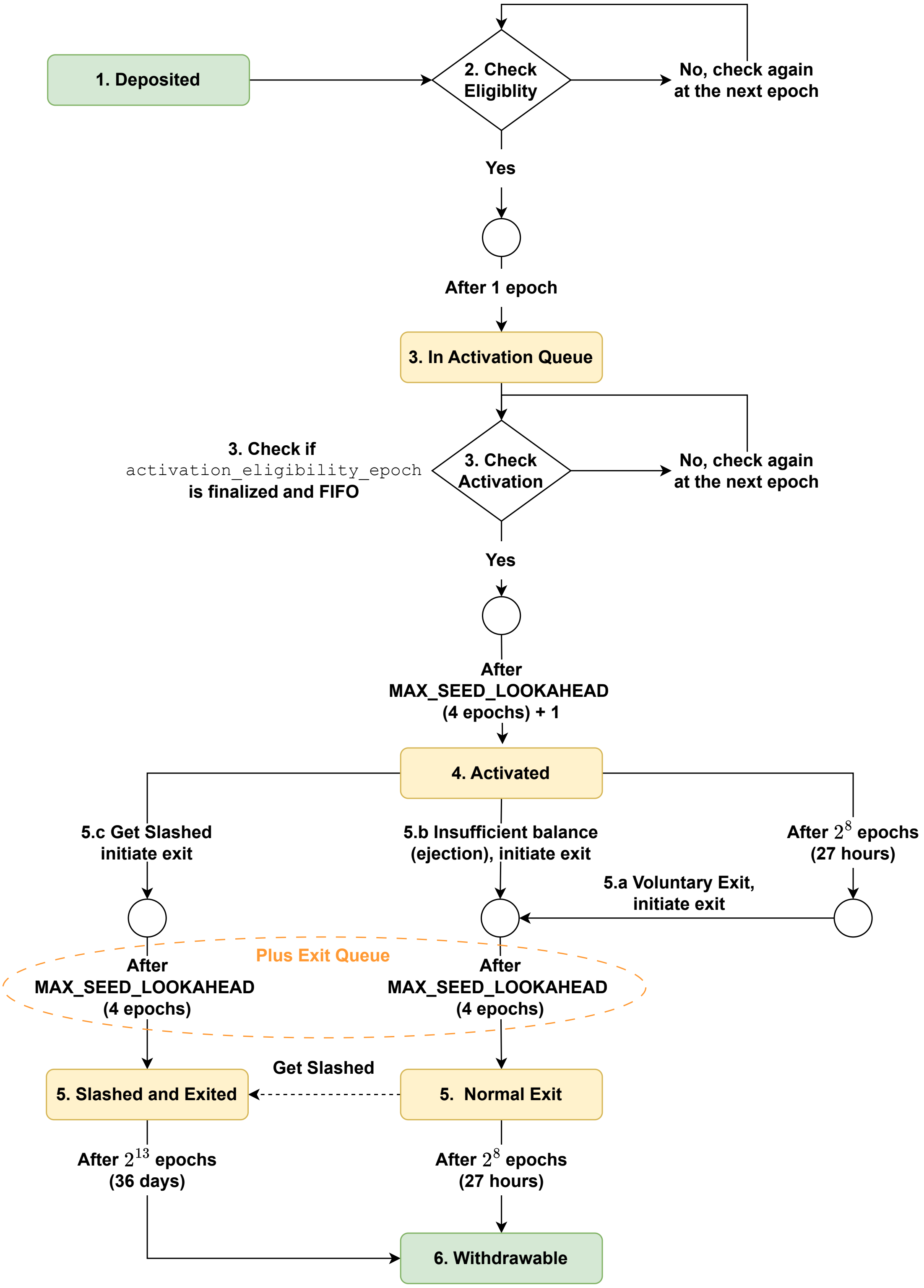

Figure 3.18: Lifecycle stages of a validator in Ethereum 2.0, including deposit, eligibility assessment, activation, operation, and exit processes. Inspired by [59].

between the Ether on these layers [60]. Ether on the execution layer is used for transaction execution and smart contract interactions, whereas Ether on the consensus layer is solely for the balance of validator accounts. As a result, users with Ether on the execution layer cannot directly utilize it on the consensus layer; instead, it must be explicitly transferred from the execution layer to the consensus layer.

### Deposit Contract

To become a validator in Ethereum 2.0, a user must initiate a standard Ethereum transaction to a specific smart contract known as the *deposit contract* [61]. A transaction amount of 32 ETH is required for activation, although it is also possible to become a validator by making multiple transactions of lower amounts that sum up to 32 ETH. The minimum deposit a user can make to the deposit contract is `MIN_DEPOSIT_AMOUNT` (1 ETH) [31]. There is no maximum deposit limit, but it is futile to deposit more than 32 ETH, as the effective balance of a validator caps at `MAX_EFFECTIVE_BALANCE` (32 ETH).

The deposit contract utilizes an incremental Merkle tree with a depth of `DEPOSIT_CONTRACT_TREE_DEPTH` (32), enabling it to accommodate up to $2^{32}$ leaves, thus facilitating a capacity for up to 4.3 billion deposits. This contract supports two primary operations: appending a leaf and calculating the root. The deposit contract and its corresponding Solidity source code are available on Ethereum's GitHub [62].

Listing 3.6 outlines a user's specific data required for the transaction. The `pubkey` will be the public key of the validator. Withdrawal information is stored in the `withdrawal_credentials` field. The `amount` indicates the deposited amount to the deposit contract and determines the increment in the validator's balance. The amount is measured in Gwei and must surpass `MIN_DEPOSIT_AMOUNT` (1 ETH).

```
class DepositData(Container):
    pubkey: BLSPubkey
    withdrawal_credentials: Bytes32
    amount: Gwei
    signature: BLSSignature  # Signing over DepositMessage
```

Listing 3.6: DepositData class as defined by the consensus specifications [31].

The field `signature` contains a BLS signature over a `DepositMessage`, encompassing the initial three fields: `pubkey`, `withdrawal_credentials`, and `amount`. A `DepositData` is essentially a signed version of a `DepositMessage`. The `signature` serves as evidence that the user possesses the secret key corresponding to the provided public key.

Once the deposit transaction executes on the EVM, it generates a receipt as an EVM log event containing the `DepositData`, provided the deposit contract validates its data. The consensus layer then retrieves this data and incorporates it into a beacon block for processing.

**Note 3.6.** The EVM does not validate the `signature` field due to its incapability to support BLS signature verification [62]. So, the responsibility of verifying the signature lies with the consensus layer. Should the signature fail to pass this validation, the associated validator will not be included in the `BeaconState`, resulting in the loss of the deposited amount. All interactions with the deposit contract burn Ether, effectively reducing its circulation. However, receipts from the deposit contract act as proof for the consensus layer to issue new Ether. This process explains the differentiation between Ether on the execution and consensus layers.

### 3.3.5 Deposit Processing

During the pre-merge phase, the beacon chain and the execution chain ran concurrently. Given that the beacon chain relies on the state of the deposit contract housed within the execution chain, participants need to establish a unified view of the deposit contract. This is accomplished through a voting mechanism, where each block proposer integrates their view of the deposit contract by including an `Eth1Data` class shown in Listing 3.7 in their beacon block. The `deposit_root` signifies the current hash tree root of the deposit contract's incremental Merkle tree, while the `deposit_count` indicates the total number of deposits recorded in the deposit contract. The last field `block_hash` refers to a particular block on the execution chain that reflects this view of the deposit contract.

```
class Eth1Data(Container):
    deposit_root: Root
    deposit_count: uint64
    block_hash: Hash32
```

Listing 3.7: Eth1Data class as defined by the consensus specifications [31].

To update the unified view of the deposit contract, participants must achieve consensus by having a majority of block proposers vote for the same `Eth1Data` over a period of `EPOCHS_PER_ETH1_VOTING_PERIOD` ($64$ epochs $= 2048$ slots $\approx 6.8$ hours) [31]. The voting period is often denoted as a cycle.

All information concerning the voting process and the latest deposit contract view is stored in the `BeaconState`, as depicted in Listing 3.8. The `eth1_data` represents the most recent unified view of the beacon chain regarding the deposit contract. The `eth1_data_votes` contain the `Eth1Data` votes from block proposers in the current cycle and are reset at the beginning of a new cycle. The last field, `eth1_deposit_index`, indicates the total number of deposits the beacon chain has processed.

```
class BeaconState(Container):
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, EPOCHS_PER_ETH1_VOTING_PERIOD*SLOTS_PER_EPOCH]
    eth1_deposit_index: uint64
```

Listing 3.8: BeaconState class as defined by the consensus specifications [31]. Only the relevant fields are included.

During block processing, a check is conducted to determine if a majority of the same `Eth1Data` is present in the `eth1_data_votes`. Usually, block proposers vote for the `Eth1Data` with the highest level of support. However, if the `eth1_data_votes` is empty, the block proposer votes for its latest view of the deposit contract.

Upon reaching a majority ($\geq$ 1025) in a cycle, the beacon state adopts this new perspective. If a majority is not achieved during a cycle, the deposit contract view remains unchanged, prompting validators to retry in the next cycle.

To cast a vote for a contract deposit view, the block containing the view must also have a timestamp that is `ETH1_FOLLOW_DISTANCE` ($2^{11} = 2048$) $\times$ `SECONDS_PER_ETH1_BLOCK` ($14$ seconds) old, amounting to roughly $8$ hours [31]. This measure was implemented to ensure high confidence that potential long reorganizations on the execution chain would not affect the beacon chain. Considering both the following distance and voting period, the minimum time a validator needs to wait after depositing to the deposit contract is $11.4$ hours, comprising $8$ hours for distance and $3.4$ hours for voting.

Upon updating the `eth1_data` with a new `Eth1Data` featuring a `deposit_count` of $n$, replacing the prior `deposit_count` of $m$, there may be $n - m$ new deposits included in forthcoming beacon blocks. The `deposit_root` in the new `Eth1Data` represents the root of the incremental Merkle tree after $n$ deposits. Therefore, block proposers must construct Merkle proofs for deposits $m+1, m+2, ..., n$ such that the final tree root for each proof matches the `deposit_root` stored in the beacon state. Each validator maintains its own deposit Merkle tree based on the deposit receipts (EVM log events) it has encountered from its execution client. As validators come across deposit receipts, they update their Merkle trees accordingly. Over time, this results in a deposit Merkle tree root that matches the `deposit_root` of the `Eth1Data` already present in the `BeaconState`. Validators can then easily generate Merkle proofs for deposits $m+1, m+2, ..., n$ since they have already constructed the Merkle tree with $n$ deposits. Therefore, if a validator becomes a block proposer, it can generate up to `MAX_DEPOSITS` ($16$) `Deposits` to include in its beacon block.

The `Deposit` class, as delineated in Listing 3.9, encompasses two components: a `data` object, which takes the form of a `DepositData` from Listing 3.6, and a `proof`, which constitutes a Merkle proof. This proof enables other validators to verify that the included `DepositData` has indeed been deposited to the deposit contract.

```python
class Deposit(Container):
    # Merkle path to deposit root
    proof: Vector[Bytes32, DEPOSIT_CONTRACT_TREE_DEPTH + 1]
    data: DepositData
```

Listing 3.9: Deposit class as defined by the consensus specifications [31].

A block proposer is required to incorporate all available `Deposits` into the beacon block in sequential order, up to the maximum of `MAX_DEPOSITS` ($16$). However, if the difference between `deposit_count` and `eth1_deposit_index` is less than $16$, only the deposit disparity should be included. If this difference is zero, no deposits are expected to be included.

Maintaining the sequential order of `Deposits` is crucial because the `eth1_deposit_index` in the beacon state is used to verify that the included Merkle proof accurately proves the deposit's position in the deposit tree created by the deposit contract. The entire beacon block is considered invalid if a block proposer fails to include any outstanding `Deposits` or any `proofs` are found invalid.

During block processing, all the `Deposits` included in the beacon block undergo processing. If all the `Deposits` are deemed valid, new validators are incorporated into the `BeaconState`. However, if a `Deposit` is made with a specific `pubkey` that already exists in the `BeaconState`, it adjusts the balance of the corresponding `Validator` instead of appending a new entry to the `BeaconState`. New validators are only added to the `BeaconState` if their `pubkey` has not been previously added.

**Note 3.7.** Following the merge, the execution chain became part of the beacon chain through an `execution_payload`, ensuring that all nodes tracking the canonical beacon chain share a unified view of the execution chain, which includes the deposit contract. In the post-merge stage, consensus on the deposit contract is automatically achieved through agreement on the beacon chain, but the voting process is still used. An upcoming hard fork is expected to fix the voting mechanism [63].

### 3.3.6 Withdrawals

The last remaining part of a fully functional proof-of-stake system is enabling participants to withdraw their stake. In earlier phases, including Phase 0, Altair, and Bellatrix, validators could not withdraw their balances, locking all deposited Ether on the consensus layer. The capability to withdraw rewards and stakes from the consensus layer to the execution layer was introduced in the Capella hard fork, as illustrated in Figure 3.2.

Every validator holds two BLS keys: one for *signing* and one for *withdrawal*. The signing key is used for all activities associated with being an active validator, while the withdrawal key proves ownership of the validator's balance and allows for modifying the `withdrawal_credentials` [64]. In the initial stages of Ethereum's proof-of-stake system, the `withdrawal_credentials` were included as a commitment to enable future withdrawals, as the integration of all system components was not yet fully understood [34].

#### Withdrawal Credentials

During the phases when withdrawals were not possible, and the methods for conducting withdrawals were not yet defined, each validator retained a withdrawal key. Through the `withdrawal_credentials`, this key allowed validators to prove ownership of their balance.

Separating the withdrawal key from the signing key divides the ownership and management responsibilities of the stake. The withdrawal key controls ownership of the stake, while the signing key handles the day-to-day operations of managing the stake. This separation allows third-party staking services to use the signing key for routine activities without having ownership of the stake or rewards, as the individual staker retains control through the withdrawal key [64].

With the introduction of Capella, withdrawals became possible (the process of how withdrawals occur will be examined in Section 3.3.6). Validators wishing to have their stake withdrawable from the consensus layer to the execution layer had to upgrade their `withdrawal_credentials` to a new format introduced in Capella. Currently, Ethereum employs two styles of `withdrawal_credentials`, distinguished by their prefix (the first byte) [34].

In the old style, a hash of the withdrawal public key is stored, with the first byte substituted by `BLS_WITHDRAWAL_PREFIX` (`0x00`). An example of the old style can be seen in Listing 3.10.

```
0x0089bd80690958ec4cd3a98d426c227dbf90238c599d0a3b6e5f1c76267cf07d
```

Listing 3.10: Old withdrawal credential using the `BLS_WITHDRAWAL_PREFIX` (`0x00`) prefix.

The new style uses a new prefix, `ETH1_ADDRESS_WITHDRAWAL_PREFIX` (`0x01`), followed by 11 zero bytes and then 20 bytes representing a standard Ethereum address (execution layer address). An example of the new style can be seen in Listing 3.11.

```
0x010000000000000000000000003804bd29e8b6140ae020cb14061dfa2f34bf1a9f
```

Listing 3.11: New withdrawal credential using the `ETH1_ADDRESS_WITHDRAWAL_PREFIX` (`0x01`) prefix.

Validators whose `withdrawal_credentials` begin with `0x00` and wish to make their balance withdrawable can update their credentials to the new style `0x01`. To update their credentials, a validator must submit a `BLSToExecutionChange` message, as defined in Listing 3.12 [34].

```
class BLSToExecutionChange(Container):
    validator_index: ValidatorIndex
    from_bls_pubkey: BLSPubkey
    to_execution_address: ExecutionAddress
```

Listing 3.12: BLSToExecutionChange class as defined by the consensus specifications [34].

The `validator_index` indicates a specific validator in the beacon state. The `from_bls_pubkey` represents the withdrawal public key of the requesting validator. The `to_execution_address` field indicates the execution layer address where withdrawals will be sent.

For the message to be accepted by nodes, it must be signed by the withdrawal secret key (ownership key) and then broadcast to the consensus layer's peer-to-peer network. The broadcasted message can be seen in Listing 3.13.

```
class SignedBLSToExecutionChange(Container):
    message: BLSToExecutionChange
    signature: BLSSignature
```

Listing 3.13: SignedBLSToExecutionChange class as defined by the consensus specifications [34].

After a period of time, a block proposer should include the `SignedBLSToExecutionChange` in its proposed beacon block. The block proposer can include up to `MAX_BLS_TO_EXECTUION_CHANGE` (16) of these changes in its beacon block.

During block processing, each included `SignedBLSToExectuionChange` must meet the following requirements [64]:

1. The associated `validator_index` within the `BeaconState` is prefixed with `BLS_WITHDRAWAL_PREFIX` (`0x00`) for its `withdrawal_credentials`.

2. The hash of `from_bls_pubkey` matches the `withdrawal_credentials` except for the first byte.

3. The signature over the `message` is verified against `from_bls_pubkey`.

If these requirements are met, a validator's `withdrawal_credentials` are permanently updated to Eth1 withdrawal credentials, allowing the validator to receive withdrawals.

Once the `withdrawal_credentials` have been updated to the new style, the withdrawal key becomes obsolete and serves no further purpose as changing `withdrawal_credentials` is a one-time operation. The execution layer address then assumes ownership of the stake. However, if a validator loses its withdrawal key before updating the `withdrawal_credentials`, the Ether associated with that validator will remain indefinitely locked in the consensus layer.

### Voluntary Exit

The typical method for validators to exit the active validator set is to issue a voluntary exit message indicating their intention to leave the system and have their stake made withdrawable. This involves creating a `VoluntaryExit` and encapsulating it within a `SignedVoluntaryExit`, which are shown in Listings 3.14 and 3.15 respectively.

```
class VoluntaryExit(Container):
    epoch: Epoch  # Earliest epoch when voluntary exit can be processed
    validator_index: ValidatorIndex
```

Listing 3.14: VoluntaryExit class as defined by the consensus specifications [31].

The `epoch` field in the `VoluntaryExit` class indicates the earliest epoch when the voluntary exit can be processed. If the `epoch` field is later than the current epoch, nodes can choose to buffer the message or ignore it. Additionally, both the included epoch and the current epoch must be greater than or equal to a validator's `activation_epoch` + `SHARD_COMMITTEE_PERIOD` (256), as depicted in Figure 3.18. The `validator_index` identifies the specific validator requesting to exit the active validator set.

For a `VoluntaryExit` message to be considered valid, it must be encapsulated within a `SignedVoluntaryExit`, signed by the exiting validator's signing key (not the withdrawal key). This mechanism prevents unauthorized parties from broadcasting voluntary exit messages on behalf of other validators.

```
class SignedVoluntaryExit(Container):
    message: VoluntaryExit
    signature: BLSSignature
```

Listing 3.15: SignedVoluntaryExit class as defined by the consensus specifications [31].

The `SignedVoluntaryExit` is broadcast to the consensus layer peer-to-peer network, enabling an upcoming block proposer to include it in their beacon block. Unlike the deposit process, which involves the execution layer, the exiting process is solely managed within the consensus layer.

**Note 3.8.** A beacon block can accommodate a maximum of `MAX_VOLUNTARY_EXITS` (16) instances of a `SignedVoluntaryExit` [31]. This allows up to 512 validators to signal their intent to cease being active validators per epoch.

### Withdrawal Process

Withdrawals for validators occur automatically, with block proposers sweeping through the validator set to identify withdrawals for inclusion in their beacon block [34]. The withdrawal process follows a round-robin pattern, commencing with `ValidatorIndex` 0. Each block proposer iterates through the validator set until they discover `MAX_WITHDRAWALS_PER_PAYLOAD` (16) withdrawals for inclusion. The next block proposer resumes the search from where the previous one left off.

To limit the computational effort involved in identifying withdrawals, a block proposer examines a maximum of `MAX_VALIDATORS_PER_WITHDRAWALS_SWEEP` ($2^{14}$) validators [34]. The search stops if it hasn't found 16 withdrawals during the sweep. This limit is necessary when many validators don't meet the requirements for a withdrawal.

There are two types of withdrawals: *partial withdrawal* and *full withdrawal* [64]. A partial withdrawal occurs when a validator's actual balance exceeds 32 ETH, and the effective balance is `MAX_EFFECTIVE_BALANCE` (32 ETH). The requirement for the effective balance ensures that validators do not remain stuck at an effective balance of 31 ETH due to the hysteresis threshold for updating the effective balance. Partial withdrawals are performed because an actual balance greater than 32 ETH is not needed, as the effective balance is capped at that amount. This process ensures that instead of all rewards being locked on the consensus layer, they become available on the execution layer.

On the other hand, a full withdrawal is only possible after a validator exits the validator set. To be eligible for a full withdrawal, the `current_epoch` must equal or surpass the validator's `withdrawable_epoch`, and the actual balance exceeds zero.

When processing the validator set, a block proposer determines whether a validator should undergo a full or partial withdrawal, with priority given to full withdrawals. To qualify for withdrawals, the `withdrawal_credentials` must adhere to the new style, starting with the prefix `0x01`.

Each withdrawal in a beacon block follows the format specified by the `Withdrawal` class, as depicted in Listing 3.16.

```
class Withdrawal(Container):
    index: WithdrawalIndex
    validator_index: ValidatorIndex
    address: ExecutionAddress
    amount: Gwei
```

Listing 3.16: Withdrawal class as defined by the consensus specifications [34].

The `index` field represents the cumulative number of withdrawals up to the current block. The `amount` field determines the reduction in the actual balance of the validator associated with the `validator_index`, as well as the increase in the execution layer account specified in the `address`. The `address` field is set to the last 20 bytes of the `withdrawal_credentials`, provided it adheres to the new withdrawal credentials style.

All nodes track the total number of withdrawals up to the present block and determine the next eligible validator for withdrawal by retaining this data within the `BeaconState`, as depicted in Listing 3.17. This stored information in the `BeaconState` is leveraged by block proposers to generate a list of `Withdrawal` for their beacon block. At the same time, other nodes use it to validate the included list of `Withdrawal` in beacon blocks.

```
class BeaconState(Container):
    # Withdrawals [New in Capella]
    next_withdrawal_index: WithdrawalIndex
    next_withdrawal_validator_index: ValidatorIndex
```

Listing 3.17: BeaconState class as defined by the consensus specifications [34]. Only the relevant fields are included.

### 3.3.7 Validator Lifecycle Summarized

A higher-level overview of a validator's lifecycle can be observed in Figure 3.19. Withdrawals are transferred from the consensus layer to the execution layer via the Engine API. The *Pending* state represents the deposit process, during which validators express their view of the deposit contract. The queue, active, and exiting stages align with the phases illustrated in Figure 3.18.

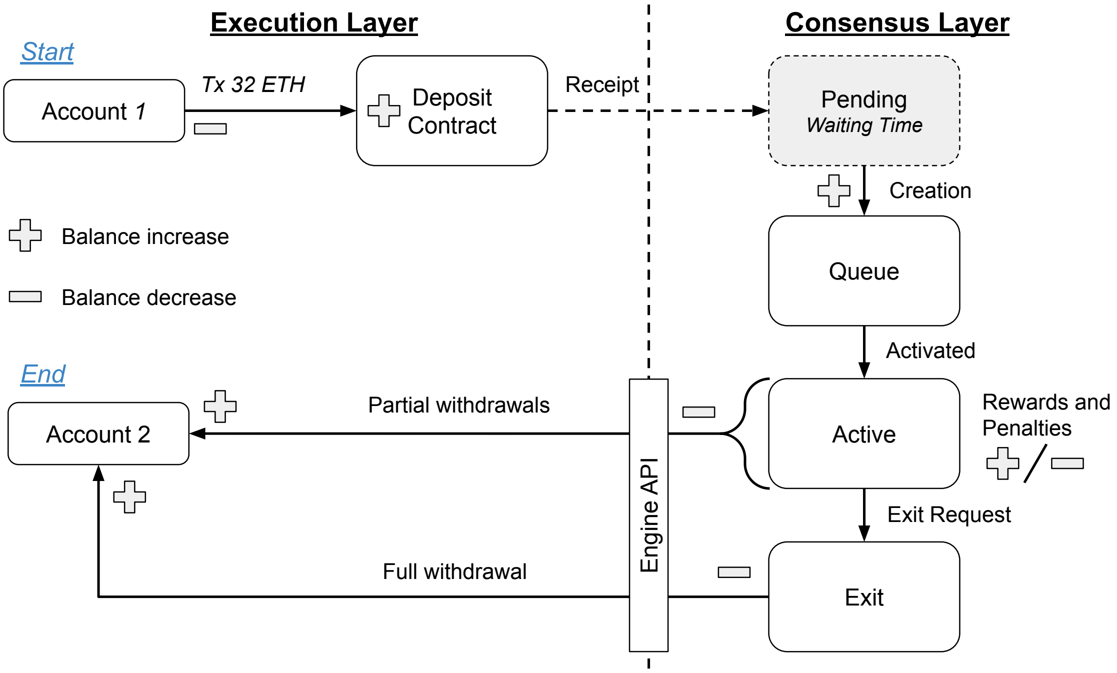

Figure 3.19: Time progresses clockwise, and a validator can remain in the active state indefinitely. Accounts 1 and 2 may be the same or different execution layer addresses. Account 2 serves as the withdrawal address.

The transfer type to the deposit contract in Figure 3.19 is a single transaction of 32 ETH.

However, users have the flexibility to achieve the same result gradually by making multiple transactions of 1 ETH or more, incrementing the validator's account balance until it reaches 32 ETH [61, 62]. The hysteresis threshold outlined in Section 3.3.1 also affects top-ups. In cases where a validator holds an actual balance of 31 ETH, a top-up of 1 ETH will not be enough to update the effective balance to 32 ETH since it does not exceed the hysteresis threshold.

## 3.4 Building Blocks

This section explores the key components that have enabled Ethereum 2.0. First, we introduce domains and explain how messages signed by different fork versions are handled during a hard fork. Next, we delve into how the beacon chain attains randomness. Following that, we analyze the composition of committees and the aggregation of attestations. Finally, we investigate the selection process for aggregators.

### 3.4.1 Domain

On the consensus layer, all BLS signatures incorporate an additional value known as a *domain*, which varies depending on the type of signing being performed [65]. Including a domain aims to prevent different signed messages intended for one function from inadvertently being valid in another function. After the Deneb fork, there are 11 distinct domain types. Table 3.1 includes 8 of these domain types, while the other 3 are excluded as they are irrelevant to this thesis.

Table 3.1: The consensus specifications define the domain types [31, 34]. The `DomainType` are represented as `Bytes4`. The column on the right indicates during which hard fork the domain type was introduced.

| **Name** | **Value** | **Fork** |
|---|---|---|
| `DOMAIN_BEACON_PROPOSER` | `DomainType('0x00000000')` | Phase 0 |
| `DOMAIN_BEACON_ATTESTER` | `DomainType('0x01000000')` | Phase 0 |
| `DOMAIN_RANDAO` | `DomainType('0x02000000')` | Phase 0 |
| `DOMAIN_DEPOSIT` | `DomainType('0x03000000')` | Phase 0 |
| `DOMAIN_VOLUNTARY_EXIT` | `DomainType('0x04000000')` | Phase 0 |
| `DOMAIN_SELECTION_PROOF` | `DomainType('0x05000000')` | Phase 0 |
| `DOMAIN_AGGREGATE_AND_PROOF` | `DomainType('0x06000000')` | Phase 0 |
| `DOMAIN_BLS_TO_EXECUTION_CHANGE` | `DomainType('0x0A000000')` | Capella |

Constructing a `Domain` (32 bytes) involves incorporating the appropriate `DomainType` from Table 3.1 corresponding to the signed message, along with a hashed tree root of a `ForkData` class, which is depicted in Listing 3.18. The initial four bytes of the `Domain` comprise of the `DomainType`, while the remaining 28 bytes constitute the first 28 bytes of the hashed tree root of the `ForkData` class. The `ForkData` consists of the current fork version and the genesis validator root. Including the genesis validator root serves to distinguish between different chains. For instance, if a validator uses the same signing key for two different chains, such as two testnets, a signature on one will not be valid on the other unless they share the same genesis validator root. Including the fork version also ensures that messages across different chain forks become invalid. This acts as a security measure to prompt users to upgrade their software for new hard forks.

```
class ForkData(Container):
    current_version: Version  # Fork Version
    genesis_validators_root: Root
```

Listing 3.18: ForkData class as defined by the consensus specifications [31].

However, there are three `Domains` that do not incorporate the current fork version: the `DOMAIN_DEPOSIT`, the `DOMAIN_BLS_TO_EXECTUION_CHANGE` and the `DOMAIN_VOLUNTARY_EXIT`. Instead, these three incorporate the genesis fork version, making them valid for all fork versions. User experience considerations primarily drive this design choice. Tools employed for these specific operations are relieved from the burden of monitoring the current fork version, thus enhancing user convenience.

Every BLS signature within the consensus layer is generated by signing the hash tree root of a `SigningData`, which includes an `object_root` and a `domain`, as illustrated in Listing 3.19. The `object_root` refers to the hash tree root of a particular object, such as an `Attestation`, while the `domain` represents a `Domain` linked with that object.

```
class SigningData(Container):
    object_root: Root
    domain: Domain
```

Listing 3.19: SigningData class as defined by the consensus specifications [31].

When a hard fork is planned, a special situation arises with `ForkData`. For instance, an attestation created during the final slot of epoch $j$ cannot be incorporated in a block before the first slot of epoch $j+1$. If a new fork version becomes active at epoch $j+1$, the attestation must have been signed using the previous fork version. Even though the attestation was signed with an outdated fork version, it should still be valid. To facilitate the validation of messages signed under a previous fork, nodes maintain a `Fork` class, as illustrated in Listing 3.20, within the beacon state. This enables the verification of messages associated with an older fork version. If the epoch associated with the message is less than the `epoch` field in the `Fork` class, the verification will use the `previous_version` instead of the `current_version`. Messages created during the `epoch` specified or later are verified against the `current_version` and cannot utilize an outdated fork version, encouraging users to update their software.

```
class Fork(Container):
    previous_version: Version  # Fork Version
    current_version: Version  # Fork Version
    epoch: Epoch  # Epoch of latest fork
```

Listing 3.20: Fork class as defined by the consensus specifications [31].

### 3.4.2 Randomness

In Ethereum 2.0, determining the roles of validators, like choosing the block proposer and organizing the active validator set into committees, represents a challenge within the consensus protocol. If every participant knew all the future block proposers and committee members for upcoming slots and epochs, the system would be susceptible to attacks like denial-of-service and censoring transactions. To mitigate these issues, the protocol requires a form of randomness, which is achieved through the use of *pseudo-randomness* that is continually updated as the beacon chain grows.

#### RANDAO

In Ethereum 2.0, the mechanism that accumulates randomness is called *Randao*. It acts as an accumulator, gradually collecting randomness from contributors via blocks in the beacon chain [66]. In each proposed beacon block, the block proposer contributes to Randao. This contribution is denoted by a `randao_reveal` object containing the BLS signature of the block proposer for their designated epoch. Thus, all block proposers for a given epoch $j$ will sign the same message (sign over epoch $j$), irrespective of their assigned slot within the epoch. The BLS signature scheme ensures precisely one valid signature for a given signing key for a message. This differs from ECDSA, where multiple possible signatures can be produced with the same secret key for the message. This restricts the block proposer's capacity to manipulate the Randao value, as they can only update it with a unique `randao_reveal`, minimizing the possibility of selecting a value advantageous to themselves.

The `randao_reveal` contributes to the update of the Randao through an `XOR` operation, ensuring that the Randao is regularly refreshed with each valid block. If $R_{n-1}$ represents the current Randao value and $r_n$ is the hash of `randao_reveal` for the newly proposed block, the new Randao value $R_n$ is calculated with Equation 3.4 and decpited in Figure 3.20.

$$R_n = r_n \oplus R_{n-1} \tag{3.4}$$

A block must include a `randao_reveal` to be valid; otherwise, it will be rejected. If a block is missing or invalid for a slot, the Randao value remains unchanged.

#### Lookahead

To give nodes in the network sufficient time to prepare for committee duties (the slot they should attest for), they are informed about the Randao value for the current and the next epoch in advance. How far in advance is determined by `MIN_SEED_LOOKAHEAD` (1 epoch) [31]. Therefore, the Randao value for epoch $j$ is based on the randomness from `MIN_SEED_LOOKAHEAD + 1` epochs ago ($j - 2$), while the Randao value for epoch $j + 1$ is derived from the randomness accumulated up to the end of epoch $j - 1$. However, predicting the Randao values for epochs beyond $j + 1$ is not feasible since they are contingent on the current epoch $j$, which is still accumulating randomness.

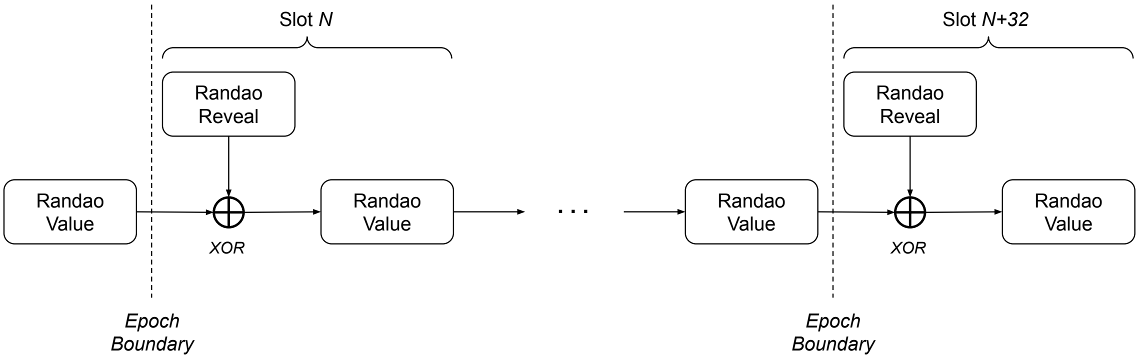

Figure 3.20: The Randao value is updated by proposed blocks containing a `randao_reveal`, which is hashed and XOR'd with the Randao value from the previous slot.

**Note 3.9.** The selection of block proposers is not influenced by the `MIN_SEED_LOOKAHEAD` parameter because their selection is based on their effective balance, which can be updated during epoch processing. Therefore, the election of all $32$ block proposers for an epoch occurs at the beginning of each epoch [50]. Since the same Randao value is employed for the entire epoch, the same block proposer would be selected for all $32$ slots. Therefore, the slot number is also incorporated into the seed to enable distinct block proposers for each slot in an epoch. For instance, during epoch $j$ to find the block proposer for each slot, the corresponding slot number $jC+k$, where $k = \{0, 1, ..., C-1\}$ would be included in the calculation.

Since the Randao value is utilized for selecting validators from the entire active validator set, there exists a potential vulnerability where attackers could exploit the network by strategically activating and exiting validators at specific times. To mitigate this risk, a maximum lookahead parameter known as `MAX_SEED_LOOKAHEAD` ($4$ epoch) is enforced, ensuring that all activation and exiting processes for epoch $j$ are delayed by a minimum of $4$ epochs [31, 50]. See Figure 3.21 for a visual example of the lookahead parameters.

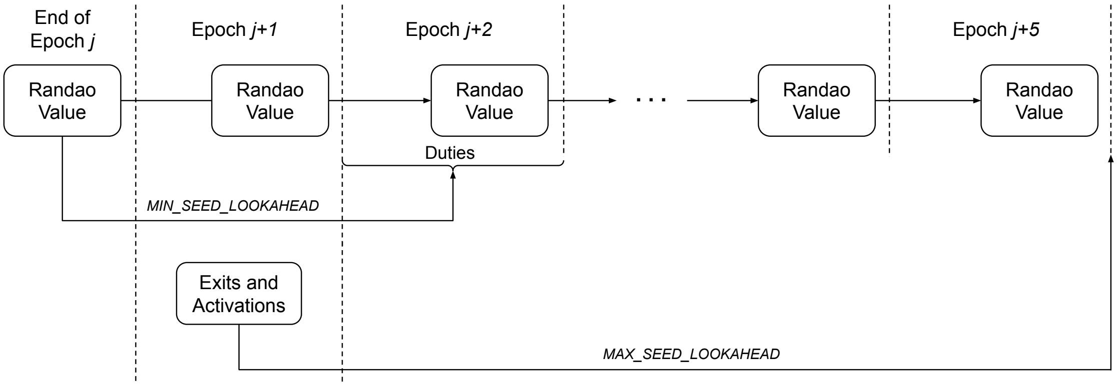

Figure 3.21: The Randao value accumulated until the end of epoch $j$ dictates committee duties for epoch $j+2$, as dictated by `MIN_SEED_LOOKAHEAD` ($1$ epoch). Validators exiting and joining during epoch $j+1$ will remain active until at least the end of epoch $j+5$ or join at the earliest at epoch $j+5$. Both exits and activations are determined by `MAX_SEED_LOOKAHEAD` ($4$ epochs). Inspired by [66].

**Note 3.10.** When computing the seed, validators incorporate a `DomainType` from Table 3.1 corresponding to its function. For example, the seed for determining block proposers includes `DOMAIN_BEACON_PROPOSER`, while the seed for determining committee duties includes `DOMAIN_BEACON_ATTESTER`. Additionally, validators factor in the epoch number during seed calculation. This is a precautionary measure in case no blocks are observed for over two epochs. If this happens, the Randao value will become stale, potentially locking the system (no blocks being proposed) in special cases.

#### Biasability

Each block proposer can modify the Randao value by proposing a valid block or maintain its current value by abstaining from proposing a block. In the latter scenario, the validator loses all rewards associated with the block proposal. Since the Randao value at the end of an epoch $j$ determines the committee duties for epoch $j + 2$ and the block proposers for epoch $j + 1$, the last slot of an epoch holds significant importance in influencing the duties. This influence is characterized as a binary decision, where the validator's action can either update the seed or not, resulting in a 1-bit influence [66]. If a validator possesses the last $k$ slots, they wield $k$ bits of influence, affording them $2^k$ possible ways to impact the seed value.

A validator can use $k$ bits of influence as an attack commonly referred to as *selfish mixing* [67], a strategy aimed at maximizing profits for a validator. The greater the number of $k$ bits of influence a validator possesses, the larger the opportunity to maximize profits. Users with a higher percentage of the overall stake are expected to receive more slots in each epoch. The attacker can also try to receive a majority in committees to influence the LMD GHOST algorithm. For instance, if a validator holds $25\%$ of the overall stake, it is anticipated to receive $1/4$ of all slots in an epoch, equating to $8$ slots. When validators obtain the last slots of an epoch, they can maximize their profits in upcoming epochs. However, they must consider that they forfeit this reward opportunity if they fail to propose a block. For a detailed analysis and examples of the biasability of the Randao, refer to sources [66, 67].

### 3.4.3 Committees

The objective of committees is to divide the validator set so that each validator participates in each epoch but only for a single slot. This division alleviates the burden on validators and keeps the message overhead associated with attestations within manageable limits. Each validator is assigned to precisely one committee, ensuring that all committees remain distinct throughout the epoch. Once the epoch concludes, all committees disband, and the active validator set is rearranged into new committees for the subsequent epoch. These committees are often denoted as *beacon committees* to differentiate them from *sync committees*, which is pertinent to light clients and falls outside the scope of this thesis. We'll refer to the beacon committees as committees in the remainder of this thesis.

### Shuffling

Ethereum utilizes a shuffling algorithm called *swap-or-not* [68] to randomly divide the active validator set into committees using the seed from Randao. This algorithm takes an array of any length and pseudorandomly shuffles it (e.g., $[0, 1, 2, 3, 4] \rightarrow [3, 0, 4, 2, 1]$), with the committees being consecutive slices of the output array. Unlike many shuffling algorithms that shuffle the entire array simultaneously, the swap-or-not algorithm shuffles only one index at a time. This feature allows validators to easily determine their responsibilities for a given epoch without performing the entire shuffling process. Additionally, it facilitates reverse lookup, identifying the original array index corresponding to a shuffled array index.

The swap-or-not algorithm is also utilized for selecting the block proposer, with each validator's effective balance determining their probability of being chosen. For further details about the shuffling algorithm and its implementation in Ethereum, refer to [69, 36].

### Composition and Sizes of Committees

So far, we have assumed that each slot would have a single committee, but this is not always the case. In any given slot, there can be a maximum of `MAX_COMMITTEES_PER_SLOT` ($64$) committees, each capable of accommodating up to `MAX_VALIDATORS_PER_COMMITTEE` ($2048$) validators. This results in a maximum of $32 \times 64 \times 2048 = 4,194,304$ active validators in Ethereum. However, achieving this limit would require approximately $134$ million Ether staked, surpassing the total amount of Ether, estimated to be around $120$ million [70].

To ensure a satisfactory level of security, each committee should consist of at least `TARGET_COMMITTEE_SIZE` ($128$) validators. This requirement makes it improbable for an attacker to randomly obtain a supermajority within a committee, even if they control many validators [71].

When generating committees for an epoch, the objective is to ensure the following [72]:

1. Each slot within an epoch should accommodate the same number of committees.
2. Maximize the number of committees while maintaining `TARGET_COMMITTEE_SIZE` ($128$) validators per committee.
3. When the number of committees per slot reaches `MAX_COMMITTEES_PER_SLOT` ($64$), the committee will increase up to a maximum of `MAX_VALIDATORS_PER_COMMITEE` ($2048$) validators.

The first condition hinges on having a minimum of `SLOTS_PER_EPOCH` ($32$) validators; otherwise, some committees will contain zero validators. Regarding requirement $2$, there must be at least $4096$ validators ($32 \times 128$), or else some committees may not achieve their minimum size requirement of `TARGET_COMMITTEE_SIZE` ($128$). Refer to Figure 3.22 for a graphical depiction illustrating the distribution of committees assigned to each slot throughout an epoch.

As shown in the `AttestationData` class from Listing 3.3, each attestation includes an `index` field containing a `CommitteeIndex`, which indicates the committee the validator is assigned in their designated slot. The `CommitteeIndex` ranges from $0$ to $N - 1$, where $N$

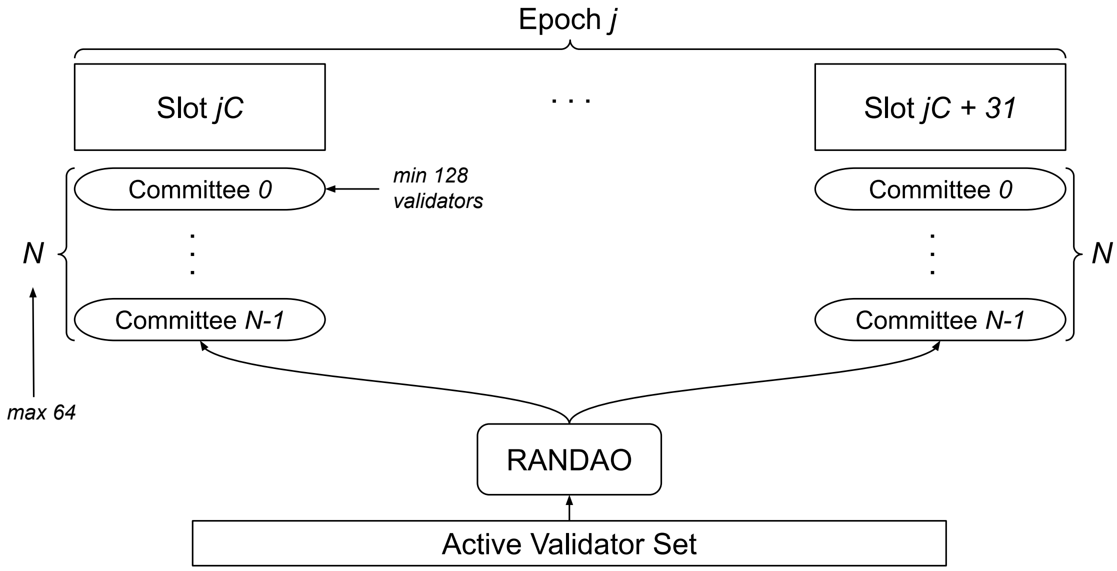

Figure 3.22: Each slot within an epoch accommodates the same number of committees, denoted as $N$, capped at a maximum of `MAX_COMMITTEES_PER_SLOT` $(64)$. Each committee aims for a minimum of `TARGET_COMMITTEE_SIZE` $(128)$ validators.

represents `MAX_COMMITTEES_PER_SLOT` $(64)$ as shown in Figure 3.22. The decision to have $64$ committees per slot originated from a sharding solution that is now deprecated [73].

### 3.4.4 Aggregate Attestations

Given the potential for up to $4$ million active validators (currently around $985,000$), storing all attestations made for a slot in a single block would be unfeasible due to the extraordinary size requirements. This underscores the primary rationale for adopting the BLS signature scheme within the consensus layer: its capability to aggregate multiple signatures into a single signature without compromising any inherent properties of a digital signature. Each validator generates an `AttestationData` class from Listing 3.3 in Section 3.2.1, with their view of the beacon chain (consensus votes), along with their assigned `Slot` and `CommitteeIndex`. This `AttestationData` class is then integrated into an `Attestation` class, as illustrated in Listing 3.21, where the `signature` field represents the validator's signature over the `AttestationData`. The validators also set a single bit to $1$ in the `aggregation_bits` to denote their membership in their assigned committee.

```
class Attestation(Container):
    aggregation_bits: Bitlist[MAX_VALIDATORS_PER_COMMITTEE]
    data: AttestationData
    signature: BLSSignature
```

Listing 3.21: Attestation class as defined by the consensus specifications [31].

When two validators share identical `AttestationData` and their `aggregation_bits` are disjointed, their signatures can be aggregated into a single signature using the BLS signature scheme. If a validator is included multiple times in the signature, the `aggregation_bits` will not account for it, highlighting the necessity for them to be disjointed.

To verify an aggregated attestation, nodes must reconstruct the aggregated public key of all included validators. This involves:

1. Identify the validators included in the aggregated attestation by their `ValidatorIndex`, using the `slot` and `index` from the `AttestationData` in conjunction with the `aggregation_bits` from the `Attestation` through the swap-or-not algorithm in reverse.

2. Retrieve and aggregate their public keys from the beacon state using their `ValidatorIndexs`.

3. Authenticate the aggregated signature using the reconstructed public key.

**Note 3.11.** The maximum number of aggregated attestations a block proposer can include is determined by `MAX_ATTESTATIONS` (128) [31]. To optimize block space, the block proposer is motivated to include aggregated attestations where the number of 1s in the `aggregation_bits` matches or closely aligns with the committee's size. As slots have a `MAX_COMMITTEES_PER_SLOT` (64), indicating that a block can contain up to two slots' worth of attestations. This provides capacity within blocks to account for missed attestations from previous slots and allows for including some imperfectly aggregated attestations.

Most `Attestations` within a single slot are likely to contain the same consensus votes (`beacon_block_root`, `source`, and `target`), while the `index` in their `AttestationData` will vary. Removing the `index` from `AttestationData` could facilitate further aggregation of `Attestations` beyond the current maximum of 2048.

The upcoming hard fork (Electra) [74] will include EIP-7549 [75], which relocates the `index` field from `AttestationData` to `Attestation`, allowing for even greater aggregation of attestations beyond the current limit of 2048. This modification's primary objective is to significantly reduce the number of BLS verifications necessary to achieve a supermajority vote ($2/3$ threshold).

### 3.4.5 Aggregator Selection

To prevent overwhelming the global Ethereum network with attestations, each committee operates within a subnet where attestations circulate before an aggregated attestation is broadcasted to the global network for block inclusion. These subnets will be looked at in Section 3.5. If every validator were to aggregate attestations and broadcast an aggregate attestation to the global network, it would be akin to each validator broadcasting its own attestation. To mitigate this issue, only a subset of each committee is chosen to act as aggregators. These selected aggregators aggregate attestations from their committee and broadcast the aggregated attestation to the global network during the final segment of a slot ($8 - 12$ seconds into the slot). The aggregation and selection process is shown in Figure 3.23.

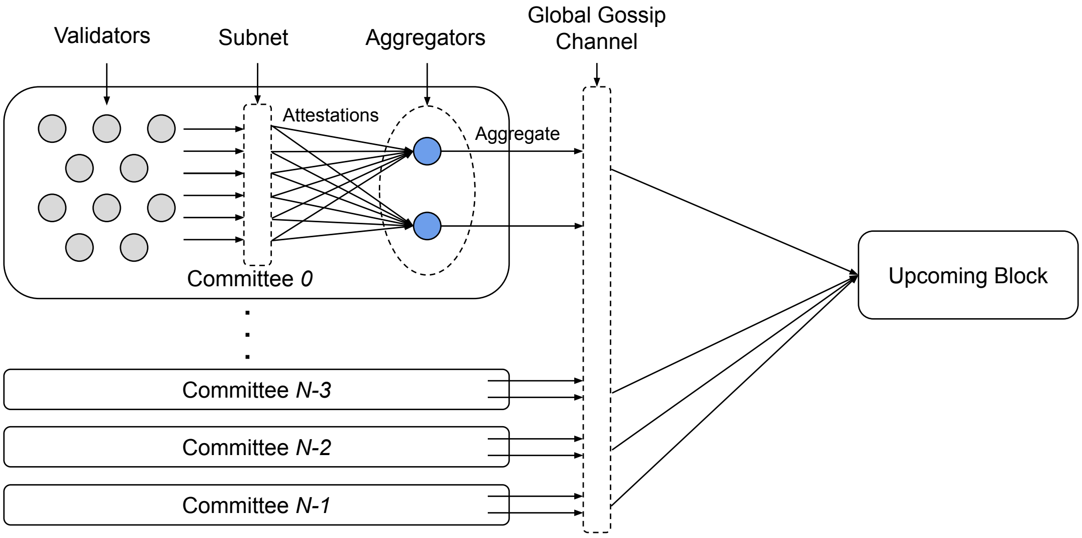

Figure 3.23: In each committee, validators sign their attestation and share it within a designated subnet. Aggregators are then selected to aggregate these attestations and broadcast the aggregated attestation to the global network. Inspired by [76].

### Selecting Aggregators

When choosing the subset of aggregators for a committee, it is advisable to include several aggregators to enhance the likelihood of having an honest aggregator with a strong network connection. Setting the number too low, such as 1 or 2, could pose risks if these aggregators experience asynchrony or choose not to fulfill their duties properly. Ethereum targets `TARGET_AGGREGATORS_PER_COMMITTEE` (16) aggregators per committee. However, to prevent attacks such as denial-of-service, the selection process for aggregators is random and known only to the chosen aggregators. Therefore, the chosen aggregators provide proof of their selection, which other validators can easily verify when the aggregated attestation is broadcasted to the global network [76].

The process for choosing aggregators for a committee is described in [50] and goes as follows.

1. Each validator in the committee creates a verifiable random number by making a BLS signature over the current slot number. The BLS signature is hashed using SHA256 to generate a uniform random number denoted as $x$.

2. Equation 3.5 determines whether a validator is an aggregator for its committee.

$$\text{Aggregator: } \begin{cases} \texttt{True}, \text{ if } x \bmod \frac{\texttt{len(committee)}}{\texttt{TARGET\_AGGREGATORS\_PER\_COMITTEE}} = 0 \\ \texttt{False}, \text{ otherwise} \end{cases} \tag{3.5}$$

   The `len(committee)` is how many validators are included in the given committee, which can be a value between 1 and `MAX_VALIDATORS_PER_COMMITTEE` (2048).

Validators use their signing key to sign the slot number, ensuring the confidentiality of aggregator selection. This signature is included as proof so other validators can verify their selection as an aggregator. In Listing 3.22 [31], the `selection_proof` represents the validator's signature over the slot number.

```
class AggregateAndProof(Container):
    aggregator_index: ValidatorIndex
    aggregate: Attestation
    selection_proof: BLSSignature
```

Listing 3.22: AggregateAndProof class as defined by the consensus specifications [31].

To prevent non-selected aggregators from altering the `aggregate` field while keeping the `selection_proof` valid, the `AggregateAndProof` must be encapsulated within a `SignedAggregateAndProof`. This wrapper includes the selected aggregator's `signature` over the `AggregateAndProof` [50], which is shown in Listing 3.23.

```
class SignedAggregateAndProof(Container):
    message: AggregateAndProof
    signature: BLSSignature
```

Listing 3.23: SignedAggregateAndProof class as defined by the consensus specifications [31].

Given the probabilistic nature of selection, the number of aggregators for a committee will vary. However, on average, there will be approximately $16$ aggregators for each committee. The worst-case scenario occurs when there are zero aggregators for a committee, resulting in all the validators assigned to the committee missing out on their rewards since their attestations will not be included in a block. The probability of having no aggregator in a committee is calculated as $(1 - \frac{16}{N})^N$, where $N$ represents the committee size and $16$ is the `TARGET_AGGREGATORS_PER_COMMITTEE` [76].

## 3.5 Networking

This section examines the networking stack used by the consensus layer. We begin by explaining what format nodes use for their peer-to-peer address. Next, we delve into the discovery process. We then explore how messages are gossiped across the network using gossip domains. Finally, we investigate how nodes request missing blocks using the Request/Response domain.

### 3.5.1 Ethereum Node Record

In Ethereum, there are three ways to convey a node's peer-to-peer address and identity: *multiaddr*, *enode*, and *Ethereum Node Record* (ENR) [77].

1. **Multiaddr:** A universal address format designed for peer-to-peer networks, representing addresses as key-value pairs with keys and values separated by a forward slash.
2. **Enode:** An Ethereum-specific URL scheme primarily used in the discovery process for the execution layer. This process closely mirrors the consensus layer discovery (discussed in Section 3.5.2) but utilizes *Discv4* [78] instead of *Discv5* [79].

3. **ENR:** The latest format, ENR, has replaced enode in most places due to its versatility. An ENR is a signed key-value record sorted by key and must be unique, meaning each key can only be present once. Some keys have predefined meanings as specified in EIP-778 [80], limiting their use to the stated purposes. ENR is used in the consensus client's discovery process and for storing information about other peers.

The ENR record contains the following information:

- *Sequence Number* - A number incremented whenever the record is updated, enabling nodes to track the most recent information effectively.
- *Signature* - A cryptographic signature of the record's content.
- *Additional key-value pairs* - Pairs that can contain arbitrary metadata.

The additional key-value pairs listed below are the fields used for the consensus layer, as specified in the consensus specifications [81].

- *id*: Specifies the name of the identity scheme (v4 is used). The identity scheme is responsible for deriving a node's address/ID.
- *secp256k1*: Represents a compressed `secp256k1` public key, consisting of 33 bytes.
- *ip*: Refers to the IPv4 address of the node, consisting of 4 bytes.
- *tcp*: Denotes the TCP port of the node's IPv4 address.
- *udp*: Indicates the UDP port of the node's IPv4 address.
- *ip6/tcp6/udp6*: Same as above, but for IPv6 instead.
- *eth2*: Consists of an SSZ encoded object `ENRForkID`.

```python
class ENRForkID(Container):
    fork_digest: ForkDigest
    next_fork_version: Version
    next_fork_epoch: Epoch
```

The `ForkDigest` is the first 4 bytes of the hash tree root of `ForkData` (32 bytes) from Listing 3.18 in Section 3.4.1. The `next_fork_version` is a `Bytes4` value representing the version of the upcoming planned hard fork, with `next_fork_epoch` indicating the epoch when the hard fork is scheduled to occur. If there is no hard fork planned, the `next_fork_version` mirrors the current fork version, and `next_fork_epoch` is set to `FAR_FUTURE_EPOCH`.

For seamless network operations, clients should connect to peers with matching `ENRForkID`. Clients can also connect to peers with a different `next_fork_version` and `next_fork_epoch` but must make sure they match before the hard fork occurs. Peers should never connect if their `fork_digest` values differ, as this indicates they do not share the same view of the current fork and/or the hash root of the genesis validator set.

- *attnets*: A SSZ object consisting of `Bitvector[ATTESTATION_SUBNET_COUNT]`, which indicates what attestation subnets the node is subscribed to. There are `ATTESTATION_SUBNET_COUNT` (64) subnets, matching the maximum committee size per slot of `MAX_COMMITTEES_PER_SLOT` (64) [81]. These subnets are used for aggregating attestations before broadcasting them to the global network, as detailed in Section 3.4.4.

  As there is no backbone for the subnets, each node is subscribed to `SUBNETS_PER_NODE` (2) subnets for a given period of `EPOCHS_PER_SUBNET_SUBSCRIPTION` (256 epochs $\approx$ 27 hours) [81]. This allows committee members to find their assigned subnet through the *attnets* field in the ENR of other nodes. The node's ID and the current epoch determine which subnets it should be subscribed to, ensuring that each node's selection is deterministic and shuffled.

### 3.5.2 Consensus Layer Network Stack

The networking stack of the consensus layer comprises three domains: *discovery*, *gossipsub*, and *request and response*, which are shown in Figure 3.24. Each domain has a distinct responsibility, which will be explored in the following sections. Additionally, the right column in Figure 3.24 indicates each domain's networking protocol/framework.

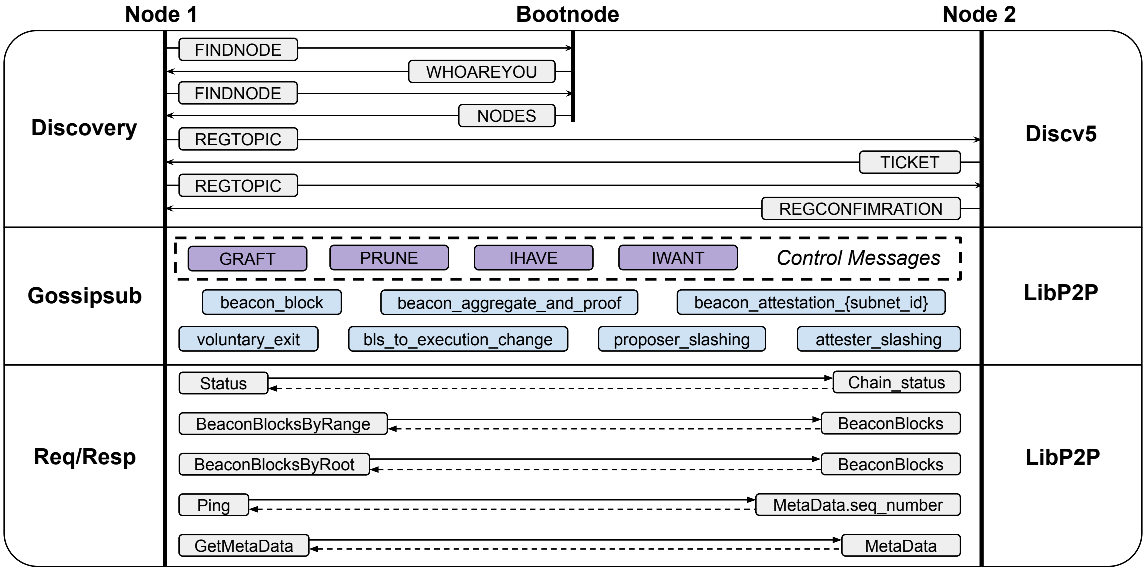

Figure 3.24: The networking stack of the consensus layer.

#### Discovery Domain

The initial step in joining the peer-to-peer network of the consensus layer involves discovery, as depicted in Figure 3.24, which uses the *Discv5* discovery protocol [79]. This protocol draws inspiration from *Kademlia Distributed Hash Table* (DHT) [82], albeit focusing on relaying and storing ENRs rather than arbitrary keys and values.

Since locating nodes without prior information is not feasible, the consensus client incorporates hardcoded bootnode addresses (ENR addresses). These addresses typically be-long to various client teams or the Ethereum foundation. From a high-level perspective, the system bootstraps by contacting a predefined set of known bootnodes. It then performs iterative lookups across the network to discover additional peers, populating a routing table organized into *k-buckets*.

In the routing table, nodes are placed into a *k-bucket* according to their *XOR* distance from the local node's ID. Given that a node ID consists of $256$ bits, each node maintains a *k-bucket* for nodes where $\log(self, n) == i$ and $0 \leq i < 256$, resulting in up to $256$ *k-buckets*. The discovery protocol uses $k = 16$, meaning each *k-bucket* can contain up to $16$ nodes [83].

The first part for a new node related to the discovery process involves establishing a handshake with the bootnode by exchanging `FINDNODE` and `WHOAREYOU` messages. Once the handshake is complete, the new node can request additional nodes from the bootnode by sending a `FINDNODE` request for nodes near a specific target distance. The bootnode responds with a `NODES` message containing nodes at the queried distance. If the new node requires more nodes, it can adjust the distance and retrieve additional nodes from adjacent *k-buckets* from the bootnode.

After receiving a sufficient number of nodes from the bootnode, the new node initiates the handshake procedure with each of these nodes and stores them in its DHT. The new node may request more nodes from these newly discovered nodes or decide that its *k-buckets* are adequately populated.

### Gossip Domain

The gossip domain, a component of the *libp2p framework* [84], operates using the gossipsub-protocol. This domain facilitates gossiping (one-to-many communication) messages throughout the network. Each message type in Ethereum has a distinct topic associated with as shown in Table 3.2. Most topics are global, meaning that every node in the network subscribes to them. Some topics, such as `beacon_attestation_{subnet_id}`, are subscribed to only by a subset of the validator set, as indicated by the *attnets* field in the ENR.

Table 3.2: Some topics as defined by the consensus specification [81, 85]. Five fields are excluded as they are outside the scope of this thesis.

| **Name** | **Message Type** |
|---|---|
| `beacon_block` | `SignedBeaconBlock` |
| `beacon_aggregate_and_proof` | `SignedAggregateAndProof` |
| `beacon_attestation_{subnet_id}` | `Attestation` |
| `voluntary_exit` | `SignedVoluntaryExit` |
| `bls_to_execution_change` | `SignedBLSToExecutionChange` |

To join and leave topics, nodes send subscribe and unsubscribe messages. Topic discovery occurs through a process involving the exchange of `REGTOPIC`, `TOPIC`, and `REGCONFIRMATION` messages [83], as highlighted in Figure 3.24. This process is necessary because libp2p does not inherently support discovering other peers and establishing connections with them [86].

Gossipsub manages topics by creating an overlay of peers for each topic, where each peer forwards messages to a subset of its peers rather than to all known peers. There are two different types of peering in a topic: *full-message* peering and *metadata-only* peering [84, 86].

Full-message peering involves transferring the entire content of a message. For example, in the topic `beacon_block`, a full-message corresponds to a `SignedBeaconBlock`. Each node in a topic is only fully peered with a small subset of all the peers, forming a sparsely connected network known as a *mesh*, with each member being a *mesh member*. Every node aims for a specific number of mesh members, with Ethereum targeting $8$ [81].

Additionally, a densely connected graph of peers exists, referred to as *remote peers*. These are peers connected to a node but not full-message peering; instead, they have metadata-only peering. The metadata-only network gossips about available messages and performs functions to help maintain the network of full-message peerings.

Peers are *bidirectional*, meaning each peer considers the other as either a full-message peering or a metadata-only peering. The process of upgrading a peer to full-message or demoting it to metadata-only is facilitated through two control messages: `GRAFT` and `PRUNE`. `GRAFT` is utilized to elevate a peering to full-message status, while `PRUNE` is employed to downgrade it to metadata-only. These updates occur at regular intervals of $0.7$ seconds, referred to as heartbeats [81].

The last two control messages, `IHAVE` and `IWANT`, ensure that remote peers have received the content of full-messages. A node sends an `IHAVE` message containing the IDs of messages it has seen in the last few seconds to a subset of its remote peers at regular intervals. If a remote peer has not encountered one of the message IDs included in an `IHAVE` message, it can request it through an `IWANT` message.

In summary, peers construct and maintain their topic mesh over time by exchanging control messages [84, 86]. These messages are essential for keeping each topic healthy and making sure messages can reach every participating peer.

**Note 3.12.** In the consensus layer, peers store the ENRs of other peers, while libp2p uses multiaddr to identify nodes. This is resolved by deriving a node's multiaddr from its ENR.

### Request/Response Domain

The Request/Response (Req/Res) domain handles one-to-one communication between nodes, allowing them to request specific information, such as missing blocks. All request and response types are depicted in Figure 3.24 and described in the consensus specifications [81].

The `Status` request and `Chain_Status` response contain identical fields: `ForkDigest`, `Finalized_Root`, `Finalized_Epoch`, `Head_Root`, and `Head_Slot`. This message exchange is crucial for establishing connections and ensuring nodes are aligned with the latest head slot. If the `ForkDigest` or the pair (`Finalized_Root`, `Finalized_Epoch`) provided by the peer does not match the client's chain at the expected epoch, the client should immediately disconnect, as this discrepancy indicates that the nodes are on different chains.

After establishing a peer connection, nodes can request missing beacon blocks using either `BeaconBlocksByRange` or `BeaconBlocksByRoot` requests. When a node is significantly behind the current head of the chain, it can utilize the `BeaconBlocksByRange` request. This request includes a `start_slot` and a `count`, indicating that the node wants blocks starting from `start_slot` up to `start_slot + count`. The responding node will return a list of beacon blocks, with a maximum limit of `MAX_REQUEST_BLOCKS` (1024). This request type is primarily used to sync new peers to the network. On the other hand, the `BeaconBlocksByRoot` request is typically used for recovering recent blocks. For example, this request is used when a node receives a block or attestation whose parent block is unknown. This allows the node to request specific blocks by providing a list of block roots it wants to obtain.

The `Ping` request checks the liveness of connected peers by including its own `MetaData.seq_number`. The responding peer replies with its own `MetaData.seq_number`. This exchange allows both peers to verify if they have the most current ENR record for each other. If the responding peer's `MetaData.seq_number` is newer than the one stored in the ENR, the node requests the latest metadata using the `GetMetaData` request. Conversely, if the requesting peer's `MetaData.seq_number` is newer, the responder may also request an update. This mutual verification ensures both nodes maintain up-to-date ENRs in their DHT.

# Chapter 4

# Related Works

In this chapter, we provide a list and brief description of available execution and consensus clients. Moving on, we present other projects that aim to achieve similar goals to ours.

## 4.1 Available Clients

The functionality and performance of a node are significantly influenced by the specific client software it operates. With a variety of clients available, each developed and maintained by different teams, the ecosystem benefits from increased client diversity [87]. This diversity enhances the system's overall resilience, providing a robust defense against vulnerabilities, bugs, and potential attacks, therefore contributing to the network's stability and security.

We will briefly introduce a list of execution and consensus clients available and highlight their usage percentage among Ethereum nodes in Figure 4.1. We start by introducing the execution clients [88]:

- **Geth** [89], short for Go-Ethereum, is the oldest and most utilized execution client. It offers the broadest range of tools and resources for both users and developers.
- **Besu** [90] is written in Java. Besu provides all Ethereum Mainnet functionality and extensive monitoring.
- **Erigon** [91], initially a fork of Geth, has since developed into a client that prioritizes speed and disk space efficiency. Its primary goal is to deliver a faster, more modular, and optimized execution layer implementation.
- **Nethermind** [92] is built on .NET, and focuses on high-performance and vast configuration options.

The following consensus clients are available for running Ethereum nodes [93]:

- **Prysm** [94] is the most used consensus client. It is written in Go and focuses on user experience, configurability, and documentation.

- **Teku** [95] is developed by the same team behind Besu, meaning they are often used in combination. Teku is a fully-fledged consensus and validator client written in Java, enabling participation in proof-of-stake consensus.
- **Nimbus** [96] is open source and written in Nim. Nimbus focuses on being as lightweight as possible regarding resources used. A lighter client means a greater margin of safety when the network is under stress.
- **Lodestar** [97], written in Typescript enables development for the JavaScript ecosystem.
- **Lighthouse** [98] is one of the earliest implementations of Ethereum and is written in Rust. Lighthouse has been available since the beacon chain genesis.
- **Grandine** [99] was recently released in 2024, and is written in Rust. Grandine aims to be high-performance and lightweight, enabling stakers to run on a Raspberry Pi [100].

The client diversity of each client in percentage among Ethereum nodes is shown in Figure 4.1 with data from [101]. Ethereum's goal is to have multiple clients with an equal share of the nodes, making it resilient to bugs and attacks and ensuring the chain can be finalized.

Client diversity in Ethereum, particularly concerning the consensus client, is crucial. If a single client controls over 33% of the nodes and experiences a bug or attack, the chain might be unable to finalize without triggering the inactivity leak. The risk escalates if a client exceeds 66%, potentially causing the chain to finalize an incorrect checkpoint. Figure 4.1 illustrates that Prysm exceeds the 33% threshold. To mitigate this risk, users should migrate to other clients. Those running Geth should also transition to other clients to dilute the pool.

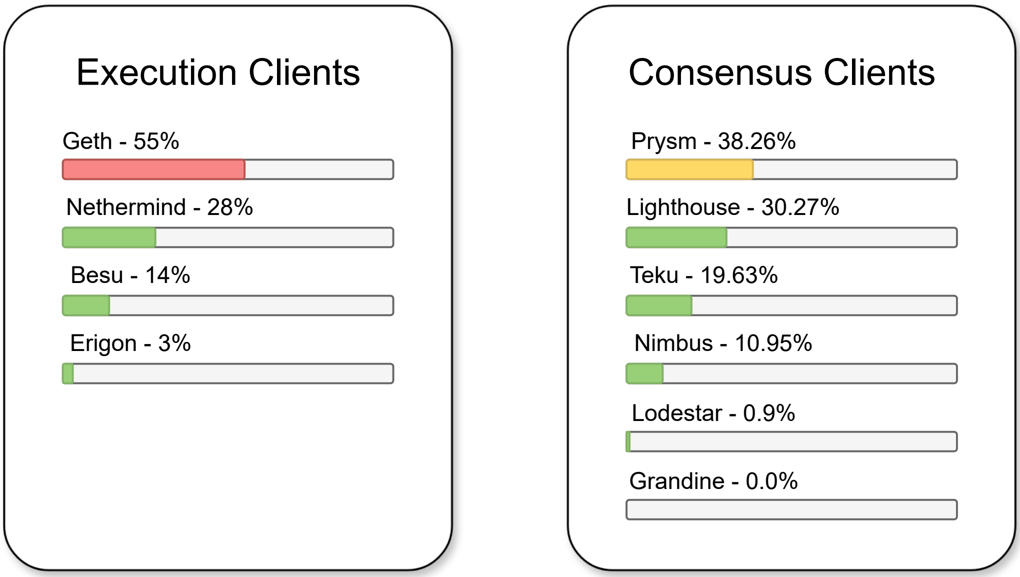

Figure 4.1: Client diversity of Ethereum execution and consensus clients as of June 12th, 2024 [101].

**Note 4.1.** The data sourced from [101] and depicted in Figure 4.1 is not completely accurate, as there is no inherent way to determine exactly which clients a node is running. The methodology used to gather this data is *supermajority* for the execution client and *blockprint* for the consensus client. These methods are explained in detail in [102].

## 4.2 Transitioning into Proof-of-Stake

Historically, one method for initiating a proof-of-stake development network involved starting the chain with a proof-of-work protocol. This initial phase continued until the mining difficulty reached a specific threshold, at which point the network transitioned to proof-of-stake [103].

However, the contemporary approach for launching a proof-of-stake devnet involves initiating the chain directly with proof-of-stake from slot 0. To progress the blockchain, validators must be initially injected into the beacon state. Without it, the blockchain would be unable to produce blocks, thereby preventing the addition of more validators later on.

This shift towards starting directly with proof-of-stake stems from significant updates in blockchain infrastructure, notably the release of Geth `v1.12.0`. This version introduced a breaking change by discontinuing support for proof-of-work, effectively moving away from the Ethash algorithm [104].

## 4.3 Existing Approaches

Since Ethereum's move to proof-of-stake is somewhat recent, there is a lack of documentation on how to set up a proper devnet for private testing. While we were in the process of setting up a development network, the documentation for the client software we were utilizing was being updated concurrently by its developers. This underscores the dynamic nature of the technologies we are working with and underscores the challenges inherent in setting up a development network.

We review 3 existing approaches for creating a private devnet:

1. Prysm's *How to Set Up an Ethereum Proof-of-Stake Devnet in Minutes* [105].
2. Lighthouse's *Simple Local Testnet* [106].
3. Zoraiz Mahmood's project *Deploy your own Local Ethereum PoS Testnet* [107].

### How to Set Up an Ethereum Proof-of-Stake Devnet in Minutes

This guide, developed by the official Prysm team, offers an introduction to launching an Ethereum node. It provides basic instructions for setting up a Geth execution client, along with Prysm's consensus and validator clients. The guide is designed as a follow-along approach, where the user copies the commands provided to configure the system. Key elements such as secret keys, JWT secrets, and configurations are hardcoded within the setup, which strictly defines the operational parameters. Additionally, the guide includes instructions on how to execute a simple transaction through the Geth console.

Another option is to start the devnet through a Docker setup, where users can easily initiate a basic network using only `docker compose`. While this method provides exceptional simplicity, it has limitations regarding configurability.

### Simple Local Testnet by Lighthouse

The Lighthouse team presents a different approach by offering a set of scripts that assist in deploying Ethereum nodes. A central script is provided to deploy multiple nodes, initiating a blockchain that starts from a genesis state post the Ethereum Merge. This method allows users to specify the number of nodes they wish to launch, providing greater adaptability for various testing needs. Lighthouse also outlines a manual process for creating a testnet, which involves a series of smaller scripts that the user can execute as needed. This setup facilitates increased validator counts across nodes, supporting scalable test environments.

### Deploy your own Local Ethereum PoS Testnet

This project builds upon Prysm's own guide, but it adds the capability to add multiple nodes. Like Lighthouse, Zoraiz Mahmood, provides a script rather than a follow-along tutorial with lines of code and instructions. Needless to say, this project uses Prysm as the consensus and validator client.

We discuss and compare these guides and approaches to our application later in Chapter 7.

# Chapter 5

# Approach

This chapter outlines our methodology for deploying a private Ethereum network, providing a comprehensive guide from initialization to monitoring. We start with a high-level overview of the application's architecture and workflow. Subsequently, we delve into initializing the blockchain's state, distributing the genesis state, and connecting peers to enable node interaction. We also detail a step-by-step procedure for activating each component of a node. Moreover, we discuss the management of validators and various validator operations, including generating deposits, changing withdrawal credentials, and making voluntary exits. Additionally, we explore the behavior of nodes acting Byzantine by intentionally skipping their aggregation selection duties. Finally, we investigate methods for users to monitor their nodes and inspect the state of the blockchain.

## 5.1 Introducing the Application

As mentioned, we begin by presenting and briefly explaining the application. The application provides the user with scripts that enable the launching of individual nodes. Together, these nodes run a private development Ethereum blockchain. Review Appendix A for a full list of all scripts included in the application. We also offer a detailed step-by-step guide on launching the full system using the provided scripts in Appendix B. In later sections, we will delve into each part of the application, providing more detail.

In Ethereum, anyone can participate and serve as a node by running both an execution client and a consensus client together. While there is no cost to running a node, earning rewards requires staking and running a validator client. In the remainder of this thesis, we use the term *node* to refer to a node that also runs a validator client.

Figure 5.1 provides a high-level overview of the application post-deployment. In this setup, $n$ nodes work together to maintain the blockchain. One node serves as the bootnode, which carries additional responsibilities during the initial launch of the blockchain.

Figure 5.2 provides an in-depth look at the architecture of a bootnode and a regular node within the application. The user executes the main script, `start_fullnode`, to initiate the deployment. The figure outlines the sequence of actions triggered by this script. The first node to be launched should always be a bootnode initiated by the user using the `--server`

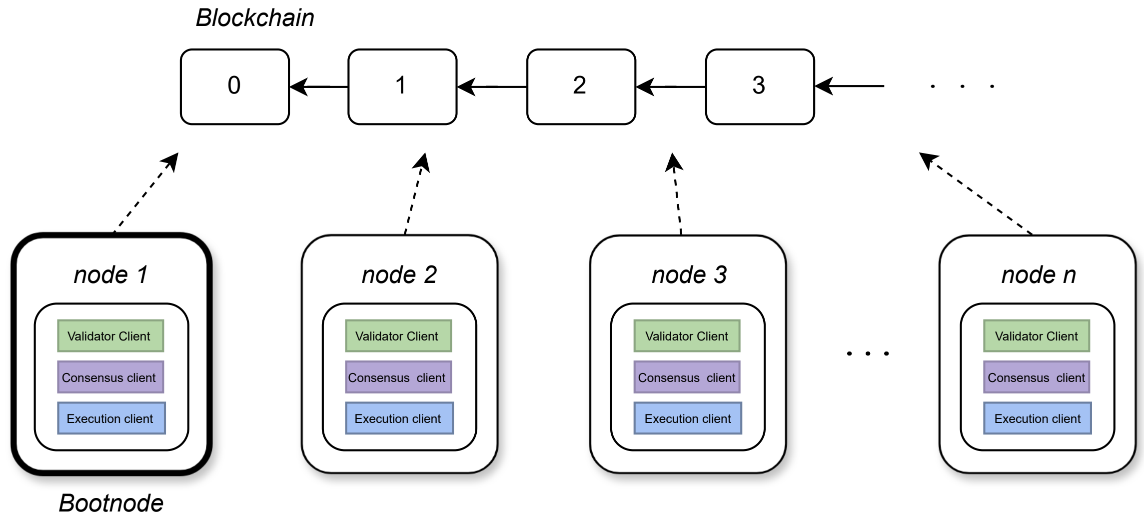

Figure 5.1: A brief overview of the deployed setup. Node 1 acts as the bootnode, enabling peer discovery for the consensus and execution layers of distinct peer-to-peer networks.

flag. This flag assigns the bootnode additional responsibilities, such as generating and publishing the initial state on a public server. Finally, both regular nodes and the bootnode launch their own instances of the execution, consensus, and validator clients. The following sections elaborate on the detailed steps involved in this process.

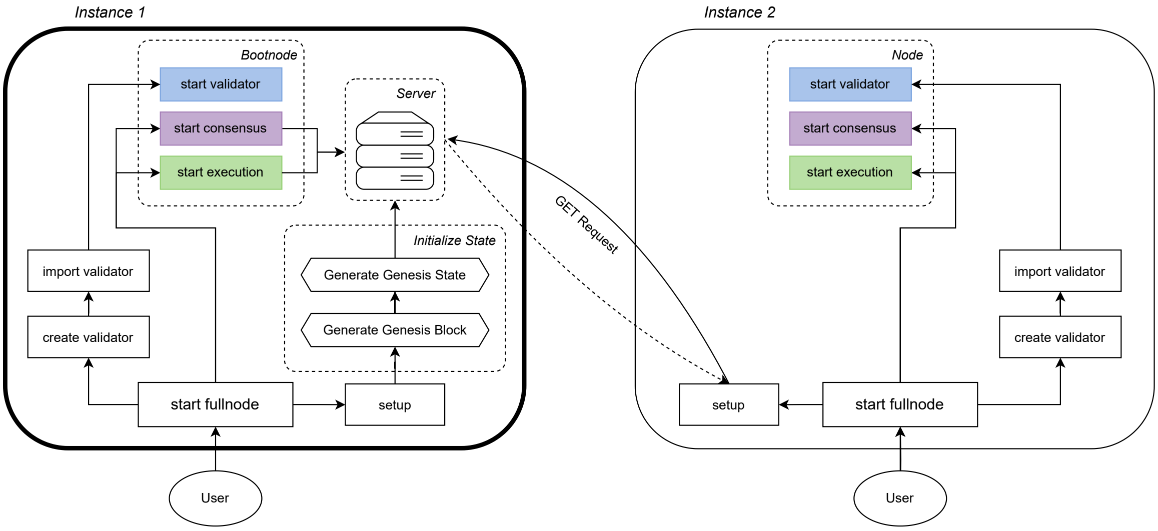

Figure 5.2: The architecture of each instance running a node. Starting with the user calling the `start_fullnode` script.

We primarily worked with 30 machines during the project, so the setup is configured by default for a maximum of 30 nodes. When deploying nodes, the user must specify two parameters to the `start_fullnode` script.

The first parameter is the node ID, which must be within the interval [1, ..., 30] and is defined through the `--node` flag. The second parameter is either the `--server` flag to indicate if the node should be a bootnode or the `--ip` flag to specify the IP address of the bootnode so the node can join the system.

**Example 5.1.** In Figure 5.1, the bootnode (`node-1`) was started with the `--server` flag and has an IP address of 152.94.162.11. To deploy a second node, like `node-2`, one must supply this IP address to the `start_fullnode` script using the `--ip` flag.

Additional changes are required to run the devnet with more than 30 nodes. In Appendix C, we outline the modifications required to run a system with a node count greater than 30.

## 5.2 Client Support

Our application uses Geth as the execution client and provides options to run either Prysm or Lighthouse as the consensus client. When choosing Prysm or Lighthouse as the consensus client, the validator client will automatically be selected from the same software origin as the consensus client. Therefore, running a Prysm consensus client will always be accompanied by a Prysm validator client, while a Lighthouse consensus client will always be accompanied by a Lighthouse validator client. To choose between Prysm and Lighthouse, the user can include either the `--prysm` or `--lh` flag in the `start_fullnode` script. If neither flag is provided, it defaults to Prysm.

When running multiple nodes, for example, `node-1`, `node-2`, and `node-3`, which are responsible for operating the blockchain, the choice of consensus client does not matter. `node-1` and `node-2` can run Prysm, while `node-3` runs Lighthouse.

## 5.3 Preparing the Chain

This section outlines the steps necessary to prepare the blockchain before its launch. We begin by introducing the tool `eth2-testnet-genesis`, which is used to define and generate the genesis state. Next, we examine the tool's input parameters and explain how we customized them to generate a genesis state that meets our specific requirements.

### 5.3.1 Genesis State

Setting up a private Ethereum proof-of-stake chain begins with defining and generating the genesis state. For this purpose, we use a tool called `eth2-testnet-genesis` [108]. This tool is specifically designed to create a genesis state with bootstrapped validators, enabling the chain to start directly with proof-of-stake.

To generate the genesis state, the tool requires three input parameters (files):

- **Genesis Block:** An Eth1 block described in a `genesis.json` file, containing information for the execution layer genesis block.
- **Beacon Chain Parameters:** Configuration parameters for the beacon chain, specified in a `config.yml` file.

- **Bootstrap Validators:** A `mnemonics.yml` file specifying the number of validators to be included in the beacon state at genesis time.

The tool generates a `genesis.ssz` file, which consensus clients use to create their beacon state at genesis time. Figure 5.3 demonstrates the operational process of the `eth2-testnet-genesis` tool. The following sections will examine these files closely, explaining the adjustments made to generate a genesis state that fulfills our specific requirements.

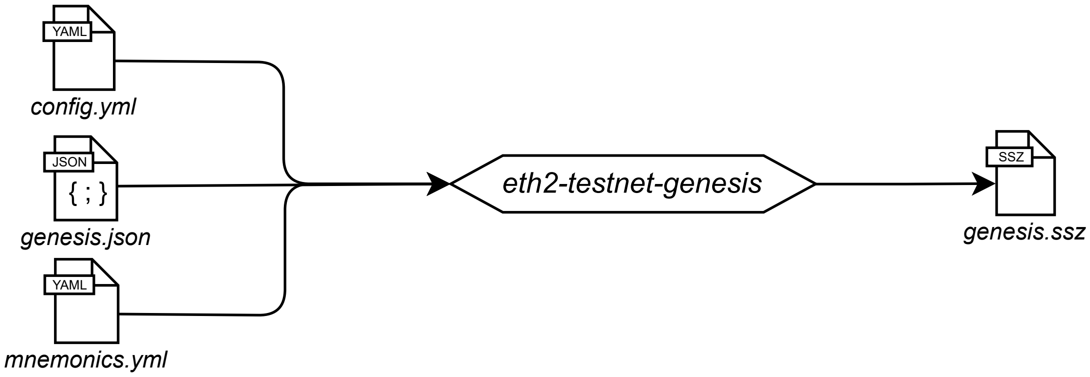

Figure 5.3: Schematic representation of the `eth2-testnet-genesis` tool processing flow.

### Genesis Block

The execution layer genesis block is specified in a `genesis.json` file, a snippet of which is provided in Listing 5.1. This file has two main sections: `config`, which includes parameters like `chainId` and `timestamp`, and `alloc`, which contains data relevant to the genesis block.

```
"config": {
  "chainId": 32382,
    ...
},
"alloc": {
    "0x27b160c1d49dfe290be0b1b10be650cd6c6f70a9": {
        "balance": "0x3635C9ADC5DEA00000"
    },
    "0x3804bd29e8b6140ae020cb14061dfa2f34bf1a9f": {
        "balance": "0x3635C9ADC5DEA00000"
    },
    ...
    "4242424242424242424242424242424242424242": {
        "code": "0x60806040526004361061003f5760003560e...",
        "balance": "0x0"
    }
}
```

Listing 5.1: Genesis block configuration file.

All the fields under `alloc` specify what should exist in the EVM from genesis time. The last field, featuring the address `424242...4242`, corresponds to the deposit contract discussed in Section 3.3.4. The `code` field contains the deposit contract in its bytecode format. The deposit contract must be included with a deposit count of $0$. This means that all $2^{32}$ leaves in the incremental Merkle tree must be the hash of $0$. If the deposit contract includes any deposit, the `eth2-testnet-genesis` tool will fail.

We manually populated the remaining fields with *externally owned accounts* (EOAs) [109] intended for use by deployed nodes to conduct transactions. Each EOA is initialized with a balance of $10,000,000$ ETH, ensuring that nodes have ample Ether reserves during experiments and testing. We included $30$ EOAs for the same reasons described in Section 5.1, assigning each node with an ID in the range $[1, \ldots, 30]$ a distinct EOA.

The $30$ EOAs are created using Geth with the command shown in Listing 5.2.

```
geth account new
```

Listing 5.2: The Geth command used for generating EOAs for the execution layer.

The generated EOAs are stored in distinct subfolders under a directory called `keys`, with each subfolder following the naming convention `node-i`, where $i$ ranges from $1$ to $30$. For example, `node-1` should use the EOA in the directory `keys/node-1`. This process ensures that each node controls a unique EOA specified in the `genesis.json` file, preventing any overlap where nodes control the same EOA. These folders are safeguarded against deletion, even during application resets. Each EOA is encrypted with an empty string as its password.

### Beacon Chain Parameters

The Beacon chain parameters are outlined in a `config.yml` file, which comprises two types of parameters: *presets* [110] and *configurations* [111]. Presets are fixed settings established during compile-time and are generally not changeable without recompiling the application. Conversely, configurations are dynamic variables that can be loaded during the runtime.

Listing 5.3 showcases the `config.yml` we utilized for testing the validator lifecycle. The `PRESET_BASE` field specifies that any parameters not explicitly listed will default to values from a designated specification. Two standards are supported by all clients: *mainnet* and *minimal*. Since our `PRESET_BASE` is set to use the mainnet settings, any unspecified parameters will inherit the values from the mainnet configuration.

```
# Free-form short name of the network that this configuration applies to
CONFIG_NAME: interop
# Extends the mainnet preset
PRESET_BASE: mainnet

# Genesis
MIN_GENESIS_TIME: 1714042445
GENESIS_DELAY: 120

# Time parameters
SLOTS_PER_EPOCH: 32
SECONDS_PER_SLOT: 12
SECONDS_PER_ETH1_BLOCK: 14
ETH1_FOLLOW_DISTANCE: 4
```

```
EPOCHS_PER_ETH1_VOTING_PERIOD: 2
MIN_VALIDATOR_WITHDRAWABILITY_DELAY: 2
SHARD_COMMITTEE_PERIOD: 2

# Validator Cycle
MIN_PER_EPOCH_CHURN_LIMIT: 512
MAX_PER_EPOCH_ACTIVATION_CHURN_LIMIT: 512

# Deposit Contract
DEPOSIT_CHAIN_ID: 32382
DEPOSIT_NETWORK_ID: 32382
DEPOSIT_CONTRACT_ADDRESS: 0x4242424242424242424242424242424242424242
```

Listing 5.3: The `config.yml` file used for defining the parameters of the beacon chain.

If one wants to accelerate the blockchain for testing purposes, it is advisable to reduce the parameters `SECONDS_PER_SLOT` and/or `SLOTS_PER_EPOCH`, as these changes have a cascading impact on the overall protocol. However, for our experiments, we maintained these two parameters at their default values to closely simulate the conditions of mainnet.

Mainnet configurations pose challenges for onboarding new validators in a devnet due to the lengthy deposit process outlined in Section 3.3.5 and the rate-limiting measures described in Section 3.3.2. To mitigate this, we've adjusted the `ETH1_FOLLOW_DISTANCE` and `EPOCHS_PER_ETH1_VOTING_PERIOD` values from $2048$ and $64$ to $4$ and $2$, respectively. This adjustment reduces the minimum time required for the beacon chain to update its deposit contract view from $11.4$ hours to just $7.4$ minutes, facilitating a more practical timeframe for adding new validators to the beacon state in a devnet environment.

To address the issue of numerous validators getting stuck in the activation queue due to the `MIN_PER_EPOCH_CHURN_LIMIT` and `MAX_PER_EPOCH_ACTIVATION_CHURN_LIMIT` being set to $4$ and $8$, respectively, we've increased both limits to $512$. Since the beacon chain can process up to $512$ deposits per epoch, it is possible to have $512$ validators joining the activation queue per epoch. This adjustment effectively removes the rate-limiting constraint.

To initiate a voluntary exit, as outlined in Section 3.3.6 a validator must have been active for $256$ epochs ($\approx 27$ hours). Additionally, to qualify for a full withdrawal after exiting, as described in Section 3.3.3, the validator must wait for an additional $256$ epochs ($\approx 27$ hours). By reducing the values of `SHARD_COMMITTEE_PERIOD` and `MIN_VALIDATOR_WITHDRAWABILITY_DELAY` from their original $256$ epochs to just $2$ epochs, we can expedite testing these functionalities within a reasonable timeframe suitable for a devnet. Instead of enduring a $54$-hour waiting period, we can now observe the entire process unfold in just $25.6$ minutes.

These adjustments significantly improve the practicality of testing functionalities related to the validator lifecycle, as detailed in Section 3.3. However, our deployed devnet faced compatibility issues when running both Prysm and Lighthouse nodes due to differences in how they interact with the `config.yml` file. These compatibility issues will be elaborated upon in Section 7.4.2.

**Note 5.1.** Adjusting some of these parameters carries a significant risk of compromising Gasper's safety properties. This risk becomes particularly apparent when altering the churn limit, as Casper FFG relies on maintaining extremely low variance in the validator set between checkpoint calculations.

When deploying the blockchain, it may be desirable for the genesis time to be the current time, meaning the chain starts at the genesis slot (slot $0$). To achieve this, the `MIN_GENESIS_-TIME` parameter in the `config.yml` file must match the current Unix time of the computer when running the `eth2-testnet-genesis` tool. To give ourselves some time when connecting other nodes and setting everything up correctly, we incorporate a `GENESIS_DELAY` which means that the blockchain won't start before the computer Unix time reaches `MIN_GENSIS_-TIME` + `GENESIS_DELAY`.

The final fields pertain to the deposit contract and should match the information provided in the `genesis.json` file, such as the chain ID and the deposit contract address.

### Bootstrapping Validators

Since the blockchain starts directly with proof-of-stake, we must include some validators in the beacon state at genesis time. Otherwise, the protocol won't be able to select block proposers for slots beyond slot $0$ as there are none to choose from. To prevent this, we bootstrap validators directly into the beacon state during the generation of the genesis state.

All validators that should be included from the genesis must be stated in a `mnemonics.yml` file. This file contains multiple *mnemonic strings* and their respective counts. The counts indicate how many validators should be derived from each mnemonic and included in the beacon state. The validators included in the genesis state are referred to as *genesis validators*. For example, in Listing 5.4, the first mnemonic contributes $64$ validators, while the second contributes $128$, resulting in $192$ genesis validators.

```yaml
- mnemonic: "nature expand bone never make where chalk autumn chicken present
    elegant face trouble giggle wrong stick brave strike child rocket sand try
    ask dinosaur"
  count: 64
- mnemonic: "mouse anchor daughter original holiday alpha expose brain garden
    access random shrug captain circle endless question plate vapor visa rival
    merge harvest frame donate"
  count: 128
...
```

Listing 5.4: The yaml file defines the number of validators to generate per mnemonic string.

We included $30$ mnemonics for the same reasons described in Section 5.1, assigning each node with an ID in the range $[1, ..., 30]$ a unique mnemonic. Using distinct mnemonics is essential to ensure that nodes do not control the same validators, as identical mnemonics would generate identical validator keys (signing and withdrawal). The mnemonics are arranged chronologically, with `node-1` assigned the first mnemonic entry, `node-2` assigned the second entry, and so forth. Nodes not participating in the devnet must have a count of $0$; otherwise, there will be validators in the genesis state that are not operated by anyone.

The process by which nodes assume control of their assigned genesis validators and how their keys are generated will be explored in Section 5.6.1.

**Example 5.2.** For setting up a devnet with three nodes—`node-1` with 64 validators, `node-2` with 0 validators, and `node-3` with 64 validators—their mnemonic counts must be set accordingly. The mnemonic counts for the first and third mnemonics must be set to 64, while the count for the second mnemonic should be 0. For mnemonics numbered 4 through 30, the count should be set to 0. This setup will generate a genesis state with 128 active validators.

## 5.4 Distributing State and Peer Discovery

This section begins by detailing the distribution of the genesis state via the bootnode. Next, we explore the peer discovery mechanisms for both the execution and consensus layers.

### 5.4.1 Distributing State

In our deployed devnet, we aim for the blockchain to start with the genesis slot (slot 0) corresponding to the Unix time when the `setup` script is executed within the `start_fullnode` script. If each node generates its own genesis state based on its individual Unix time, discrepancies will arise because the `MIN_GENESIS_TIME` will vary. This will lead to differences in the genesis state, particularly the `genesis_time` within their beacon state shown in Listing D.1. As a result, nodes will have different perspectives of the genesis time, preventing them from establishing a connection with each other. Additionally, if each node were to generate its own genesis state, coordinating and ensuring that all input files required by `eth2-testnet-genesis` are consistently up-to-date across each node would be impractical and error-prone. This could result in further inconsistencies and complicate the deployment process.

To circumvent these issues, we employ a single node, referred to as the bootnode, depicted in Figures 5.1 and 5.2, to define the genesis state and distribute it to other participating nodes. The bootnode is a standard node running an execution, consensus, and validator client. However, it carries the additional responsibility of generating and distributing the state and serves as the entry point for other nodes to join the network. The generation part corresponds to the process described in Section 5.3. To designate a node as a bootnode, the `--server` flag must be included when executing the `start_fullnode` script.

The bootnode starts a basic Python server on port 8000, which hosts all essential genesis state files, including `genesis.json`, `config.yml`, `genesis.ssz`, and `mnemonics.yml`. Normal nodes (those not started with the `--server` flag) must retrieve this information from the bootnode. This retrieval is facilitated by specifying an `--ip` flag in the `start_fullnode` script, allowing nodes to fetch the genesis state. This process is depicted in Figure 5.4 and takes place during the *setup* phase, which corresponds to the setup box in Figure 5.2. Furthermore, the two remaining fields hosted on the server, `enode` and `enr`, serve the purpose of peer discovery for the execution layer and consensus layer, respectively.

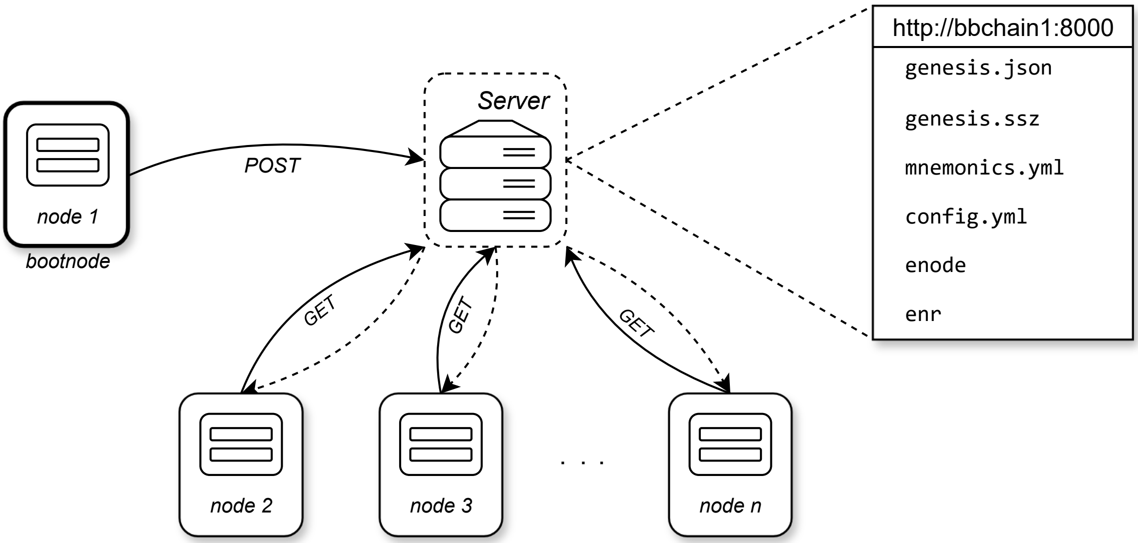

Figure 5.4: The bootnode operates a server that uploads the genesis state and addresses necessary for peer discovery. Other nodes fetch this information to synchronize their genesis state and engage in peer discovery.

### 5.4.2 Peer Discovery

In Section 3.5, we discussed how nodes rely on specific entry point addresses—enode for the execution layer and ENR for the consensus layer—to initiate their respective peer discovery protocols. Typically, these values are hardcoded into clients. However, this approach is not viable for us because the addresses change depending on the machine used, and no single machine will always be available as an entry point. Therefore, we distribute enode and ENR addresses along with the genesis state, as illustrated in Figure 5.4. The selected bootnode, which uses the `--server` flag, hosts its enode and ENR addresses. This setup allows other nodes to retrieve these addresses, initiate peer discovery processes for both layers, and establish connections with peers.

In summary, when nodes run the `start_fullnode` script with the `--ip` flag, they retrieve the necessary genesis state and peer discovery information from the bootnode, ensuring a seamless process for new nodes to join the network. Additionally, nodes connected to the bootnode will establish connections with other peers, as the bootnode provides a list of nodes it is connected to.

**Enode** During the bootnode's setup, we generate a nodekey to derive the enode address. The execution client automatically generates the nodekey during its initialization phase. The enode address is formed by combining the nodekey (hexadecimally encoded node ID) with the bootnode's IP address and a port number ($30303$ for Geth). This enode address is then posted to the hosted server so that regular nodes can retrieve the bootnode's enode address and initiate their peer discovery process for the execution layer's peer-to-peer network. Listing 5.5 illustrates an enode address structure: the part before the @ symbol denotes the nodekey, while the part after denotes the node's IP address and port number.

```
enode://bad2da161572060461ba977eacff67d6cc3bacd0b9577334e17db73c3e48e2bc902316
0355f9cfd5cf55e80ce5afd2f58d5d1da12d460fda89dbef6cf05731cf@152.94.162.11:30303
```

Listing 5.5: An example enode address of the execution client.

**ENR** The process of posting and fetching the ENR for the consensus layer is similar to that of the execution layer, with a notable distinction in how the bootnode obtains its ENR. The ENR address of the consensus client is not known until after the consensus client has been started. Once the consensus client is up and running (the process for starting the consensus client will be explored in Section 5.5.2), one can fetch the ENR by querying the endpoint `127.0.0.1:8080/p2p`. This phase is specific to Prysm, as we do not support having Lighthouse as the bootnode. Once obtained, the ENR is posted to the server, enabling other nodes to fetch it and start their peer discovery process for the consensus layer peer-to-peer network. The data contained in an ENR was discussed in Section 3.5.1 with an example of an ENR address in its encoded format shown in Listing 5.6.

```
enr:-MK4QJoAiA_o-s8Gn7u9c00s0KHC4aj0ACzMc4TFLgDK_X66dJcN0Z71xb02dlait-
    y5uqCT9QAJ7VvuA4a2DXx0ndCGAY8pKLXhh2F0dG5ldH0IAAAAAAAAACEZXRoMpD0K
    kgBIAAAk___________gmlkgnY0gmlwhJheog-Jc2VjcDI1NmsxoQII5FVvpH7k88yC0W4y
    rFjhvTKfBiuNlxB0za2NxAT0Z4hzeW5jbmV0cwCDdGNwgjLIg3VkcIIu4A
```

Listing 5.6: An example ENR address of the consensus client.

## 5.5 Starting a Node

After the main script `start_fullnode` finishes the setup phase, it calls three other scripts individually, responsible for each client: execution, consensus, and validator. The subsequent sections will delve into how each client is executed. We use `node-1` as the example node for the provided listings, featuring data actually used in the devnet.

### 5.5.1 Execution Client

The script responsible for launching the execution client begins by initializing the blockchain's genesis state (EVM state) for the node with a genesis block. This is accomplished using the command in Listing 5.7. The command requires the `genesis.json` file, created during the setup phase, to initialize the state.

```
./geth init --datadir="./execution" genesis.json
```

Listing 5.7: Geth command for initializing the blockchain for a node.

Next, we launch the execution client by running the Geth binary with specific command-line flags. These flags configure settings such as the network, API access, and metrics. They ensure the client operates according to the desired requirements and is set up for peer discovery.

The command for launching the execution client is shown in Listing 5.8, which contains a subset of the flags used with sample values for easier understanding. Flags related to logging and port numbers are not included in the listing.

The `--bootnodes` argument is required only if the `start_fullnode` script is initiated with the `--ip` flag. When the `--server` flag is used, the `--bootnodes` argument should be set to an empty string or can be omitted altogether. The `--nodekey` argument involves how the execution client generates its enode.

The three arguments with comments containing *EOA* ensure that the client takes control of its assigned EOA, one of the 30 listed in `genesis.json`. These arguments specify which EOA to unlock and the password used for encryption.

For the execution and consensus clients to communicate through the Engine API, they need a JWT token. The command in Listing 5.7 automatically creates this token, and its path is specified with the `--authrpc.jwtsecret` argument.

```
./geth
    --networkid="32832"
    --http                                                // Execution API
    --metrics                                             // Enable metrics
    --authrpc.jwtsecret="./jwtsecret"                     // Engine API
    --datadir="./execution"
    --bootnodes="enode://bad2...1cf@152.94.162.11:30303"  // Bootnode Address
    --keystore="./keys/node-1/keystore"                   // EOA keystore
    --unlock="27b160c6f70a9..."                           // Execution Layer EOA
    --password="./keys/geth_password.txt"                 // Password for EOA
    --nodekey="./bootnode/nodekey"                        // Enode
```

Listing 5.8: Command used to start the execution client using Geth.

The script `start_geth_execution_client` is responsible for generating a Geth database for the blockchain and launching the client with the necessary flags and values. This script is called by the `start_fullnode` script, as illustrated in Figure 5.2.

### 5.5.2 Consensus Client

Once the execution client is launched, the consensus client is started using either `start_prysm_consensus_client` or `start_lighthouse_consensus_client`, depending on the user's preference for the client choice. The initialization process for the consensus client involves specifying particular flags. These flags are utilized for tasks such as establishing the genesis state based on the `genesis.ssz` and `config.yml` files, defining the data directory, and setting the bootstrap address (bootnode address). For the same reason as described in Section 5.5.1, only nodes starting with the `--ip` flag require a bootnode address.

To enable communication with the execution client via the Engine API, the consensus client needs to utilize the same JWT token generated by the execution client, specified by `jwt-secret` and `execution-jwt` for Prysm and Lighthouse, respectively. This connection is established through the address `127.0.0.1:8551`, which is the default authentication port for the execution client (Geth).

To utilize the Beacon API, specific flags must be set (indicated with Beacon API comments), making the Beacon API accessible at the address `127.0.0.1:3501`.

### Prysm

Listing 5.9 shows the command to launch the consensus client using Prysm.

```
./beacon-chain
    --datadir="./consensus/beacondata"
    --genesis-state="./consensus/genesis.ssz"
    --chain-config-file="./consensus/config.yml"
    --jwt-secret="./execution/jwtsecret"                // Engine API
    --http                                              // Beacon API
    --grpc-gateway-port="3501"                          // Beacon API
    --bootstrap-node="enr:-MK4QIY9Ssd55tmxl2Gb7U..."    // Bootnode Address
```

Listing 5.9: Command used to start the consensus client, using Prysm.

### Lighthouse

Unlike Prysm, we do not directly input the bootnode's ENR, `genesis.ssz`, and `config.yml` into the Lighthouse binary. Instead, we organize them within a designated testnet directory, which Lighthouse accesses using the `--testnet-dir` argument for setting up its genesis state and using the ENR for peer discovery. Listing 5.10 shows the command for launching a Lighthouse consensus client.

```
./lighthouse bn
    --datadir="./consensus/beacondata"
    --testnet-dir="./config/lighthouse_testnet"
    --execution-jwt="./execution/jwtsecret"             // Engine API
    --http                                              // Beacon API
    --http-port="3501"                                  // Beacon API
```

Listing 5.10: Command used to start the consensus client, using Lighthouse.

## 5.5.3 Validator Client

As the final step, the validator client is launched by the `start_fullnode` script. Depending on the user's selection of the consensus client, a validator client from the same software origin is invoked using either `start_prysm_validator_client` or `start_lighthouse_validator_client`.

Two preliminary steps are required: ensuring the validator client controls its assigned genesis validators from the `mnemonics.yml` file and converting its EOA address into a checksummed version so that block rewards (the `priority fee` in transactions) can be sent to the correct execution layer address. Lighthouse and Prysm use the `--suggested-fee-recipient` flag to specify where block rewards should be directed. The process by which the validator client assumes control of its genesis validators will be looked at in Section 5.6.1.

We utilize the `--graffiti` argument to tag blocks with arbitrary data, allowing us to identify which client and node number generated each block.

#### Prysm

For Prysm, we specify the wallet directory and passphrase so that the client can identify the validators belonging to the client. Listing 5.11 lists the flags passed to the client.

```
./validator
    --datadir="./consensus/validatordata"
    --chain-config-file="./consensus/config.yml"
    --grpc-gateway-port="3501"
    --beacon-rpc-gateway-provider="3501"
    --suggested-fee-recipient="0x3804bD29E8b614..."       // EOA in checksum format
    --graffiti="Prysm node-1"
    --wallet-dir="./validator/wallet-1"
    --wallet-password-file="./validator/passphrase"
```

Listing 5.11: Command used to start the validator client, using Prysm.

#### Lighthouse

Lighthouse does not require a wallet path. Instead, it automatically searches the default validator directory created when importing Lighthouse validators. Listing 5.12 shows the command for launching a Lighthouse validator client.

```
./lighthouse vc
    --datadir="./consensus/beacondata"
    --testnet-dir="./config/lighthouse_testnet"
    --beacon-nodes="http://localhost:3501"
    --suggested-fee-recipient="0x3804bD29E8b614..."      // EOA in checksum format
    --http \
    --metrics-port="8081"
    --graffiti="Lighthouse node-1"
```

Listing 5.12: Command used to start the validator client, using Lighthouse.

### 5.5.4 Logging

All clients support different levels of logging verbosity. For instance, Prysm offers seven levels: `trace`, `debug`, `info`, `warn`, `error`, `fatal`, and `panic`, with the default set to `info`. Users can manually adjust the verbosity level using specific flags: `--verbosity` for Prysm, `--debug-level` for Lighthouse, and `--verbosity` for Geth. Typically, we ran each client with `debug` to obtain a comprehensive overview of their operations.

Usually, clients are launched either in separate terminal windows (totaling three) or combined into one. However, each approach has its own challenges, such as managing multiple windows or dealing with excessive logging in a single window. Instead, we direct all logging into separate log files and provide a script enabling users to monitor each client's output.

Users can specify which client to monitor, e.g., `./monitor <execution|beacon|validator>`. This script reads from the log files and displays the output in the terminal.

We created a script `kill_clients`, which orderly shutdowns the blockchain. This script terminates the running blockchain clients and offers an optional feature to preserve operational logs. Using the `--log` flag, users can ensure that all logs from a session are saved to a designated directory. This directory retains its contents across multiple runs, preventing the automatic deletion of logs on each execution. This functionality is particularly useful for troubleshooting and historical analysis of simulations. It allows users to review past operations without the risk of data loss between sessions.

## 5.6 Managing Validators

In this section, we outline the preparations required for managing validators. First, we explain the key generation process and how to import them into a wallet that a validator client can control. Then, we discuss the various operations a node can perform related to its validators once the chain has started, including making and sending deposits, setting up withdrawals, and exiting validators.

### 5.6.1 Pre-deployment Validator Preparations

Since validators are already included in the genesis state, we need a deterministic way to ensure that the deployed nodes control the correct validators based on the count of their assigned mnemonic in the `mnemonics.yml` file. To achieve this, we utilize a *Hierarchical Deterministic* (HD) wallet [112] for validator key generation. Using an HD wallet, ensure that validator keys are deterministically generated based on a given mnemonic string. This setup guarantees that the same validators are consistently generated and match those included in the genesis state, enabling nodes to control the correct validators.

HD wallets typically generate validator keys (signing and withdrawal) using the *BLS12-381 Key Generation* standard specified in EIP-2333 [113] and organize them in a hierarchical structure following the *BLS12-381 Deterministic Account Hierarchy* standard specified in EIP-2334 [114].

Each key is generated with a *derivation path*, a string that specifies indices used to navigate the tree of keys created with EIP-2333. Withdrawal and signing keys follow predefined paths: `m/12381/3600/i/0` and `m/12381/3600/i/0/0`, respectively, where $i$ represents the validator number. For example, generating keys for $100$ validators involves creating $100$ derivation paths for withdrawal keys and $100$ derivation paths for signing keys, resulting in $i$ ranging from $0$ to $99$.

Once the validator keys are generated, they must be imported into a node's validator wallet so that the node's validator client can control them. Only the signing keys must be imported into the wallet, as the withdrawal keys are only relevant for changing withdrawal credentials, as discussed in Section 3.3.6. The withdrawal keys are kept in cold storage for security.

Generating validator keys (both signing and withdrawal) takes approximately $1$ second per key, and importing a signing key into a wallet takes around $0.13$ seconds. For example, if a node controls $128$ genesis validators, the total time for key generation and import would be $128 \times (1 + 0.13) = 144$ seconds. While this duration is manageable, it becomes impractically long for a devnet environment when a node controls thousands of validators, making the key generation process excessively time-consuming.

To avoid the time-consuming process of generating new validator keys each time, we have pre-generated $4096$ keys (signing and withdrawal) for all $30$ mnemonics included in the `mnemonics.yml` file. Each node (`node-i`) has its signing and withdrawal keys in the `keys/node-i` folder alongside its EOA. This key generation is a one-time operation and does not need to be repeated unless additional validators or more than $30$ mnemonics are required; as mentioned earlier, this is discussed in Appendix C.

After setting up the execution and consensus client for a node, we copy its signing keys from `keys/node-i` into a distinct node wallet directory (`validator/wallet-i`) created during the startup phase. These keys are then imported into a wallet the validator client uses to take control of the signing keys, enabling it to perform validator duties. This process ensures that when the validator client is launched, it already contains the appropriate number of genesis validators specified in the `mnemonics.yml` file. Figure 5.5 illustrates this workflow.

**Example 5.3.** Consider the first entry of Listing 5.4, featuring a mnemonic string paired with a count of $64$. This indicates that `node-1` should start with $64$ genesis validators. We follow these steps:

1. Transfer the initial $64$ signing keys from a pool of $4096$ signing keys stored in `keys/node-1` to a wallet directory specific to the node (`validator/wallet-i`).
2. Import these signing keys into the wallet.
3. Start the validator client.

Once the genesis time occurs and the blockchain starts, `node-1` can begin performing validator duties for its $64$ genesis validators.

In Section 5.3.1, we discussed the `GENESIS_DELAY` added to the `MIN_GENESIS_TIME` to provide additional time for node setup and peer discovery. While we typically use a base `GENESIS_DELAY` of $120$ seconds, importing $4096$ validators takes approximately $9$ minutes, making it impossible for nodes to be ready within $120$ seconds if they have to import more than $923$ validators. To accommodate this, we introduce an additional delay to `GENESIS_DELAY`, calculated based on the highest validator count in Listing 5.4. This results in Equation 5.1, where `highest_validator_count` represents the highest count, and $120$ serves as the base delay.

$$\texttt{GENESIS\_DELAY} = \frac{9 \times 60}{4096} \times \texttt{highest\_validator\_count} + 120 \tag{5.1}$$

Since only the bootnode generates the genesis state and sets the `GENESIS_DELAY`, the other nodes that must be active from the genesis time must be started within the base delay. The base delay must be manually adjusted if starting the nodes requires more time.

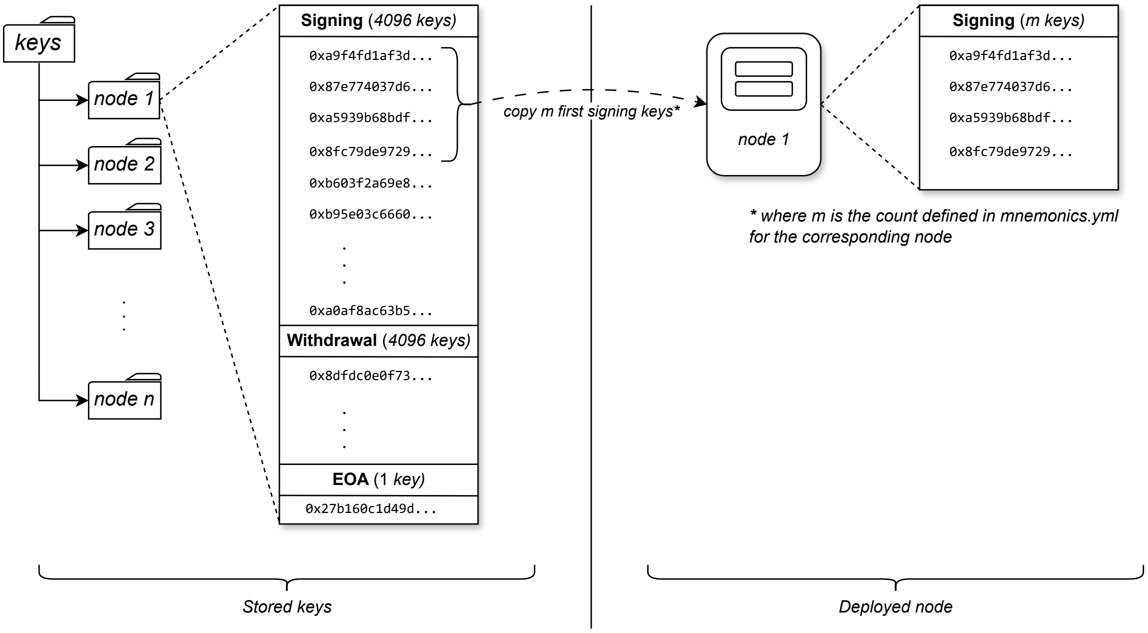

Figure 5.5: An illustration of how `node-1`'s signing keys from the `keys/node-1` folder are copied to `node-1`'s wallet folder (`validator/wallet-1`). This process prepares the keys for import into the wallet, which the validator client will manage.

**Note 5.2.** The tool `eth2-testnet-genesis` can generate signing and withdrawal keys in milliseconds, significantly faster than the standard method, which takes a whole second. This increased speed is due to several optimizations within `eth2-testnet-genesis`, such as bulk key generation using concurrency. Additionally, the tool bypasses certain security measures, such as encrypting each key, which are typically necessary for secure key generation. Since `eth2-testnet-genesis` is designed to create genesis states in a devnet environment, the primary focus is speed and efficiency rather than security.

### 5.6.2 Creating Validators

The script `create_validator` is responsible for generating validator keys (signing and withdrawal). This script offers two approaches to creating validators, each emphasizing either security or speed:

1. *Secure:* Generates the keys from scratch using the node's assigned mnemonic.
2. *Speed:* Copies existing keys, as described in Section 5.6.1.

One can choose between the two methods using the `--insecure` flag. Including this flag enables the speed method, while omitting it enables the secure method. The overall workflow of the `create_validator` script is depicted in Figure 5.6.

When initializing a node with the `start_fullnode` script, the `create_validator` script is indirectly invoked. Since the wallet has not yet been created, the `wallet exists` check will always return `No`. Consequently, the script will create the wallet and copy the signing

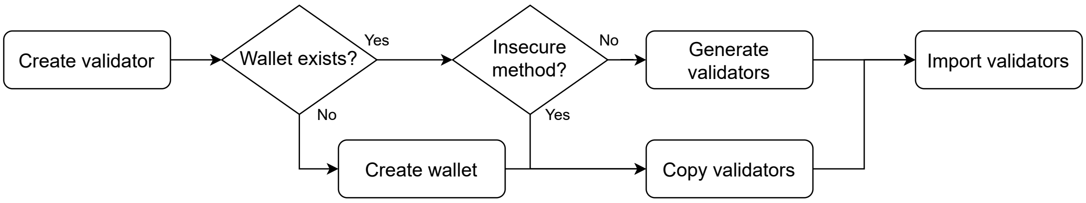

Figure 5.6: Flow diagram illustrating the logic of creating validators with the `create_validator` script.

keys from `keys/node-i` into it, as shown in Figure 5.5 and described in Section 5.6.1. Finally, the validators are imported into the wallet, completing the process.

Once the node is fully set up and operational, with its assigned genesis validators imported into the node's wallet and managed by the node's validator client, users can create additional validators using their preferred method. To accomplish this, one needs to generate or copy additional signing keys securely. These are then imported into the node's wallet, enabling the validator client to manage them effectively.

To activate these additional validators, the user must make a deposit and wait for the process outlined in Section 3.3.5. The steps for making and sending a deposit will be explained in Section 5.6.3. Once the additional validators are activated, the node can begin performing validator duties for the newly added validators.

**Note 5.3.** We highly recommend using the insecure method when creating additional validators. This approach provides a significant speed advantage for testing purposes. Additionally, since there are no economic consequences if the keys are compromised, it is a suitable choice. However, it's crucial to note that the mnemonics used for generating the validator keys are openly available. Therefore, these included mnemonics should never be utilized in a mainnet scenario.

### 5.6.3 Validator Operations

While nodes encompass execution, consensus, and validator client functionalities, they lack certain features, such as generating deposit data for a deposit transaction and altering withdrawal credentials. To address these limitations, we use a third-party tool called `ethdo` [115]. `ethdo` is a command-line tool designed for managing common tasks in Ethereum 2.0 related to a beacon node. Most operations with `ethdo` require connecting it to a beacon node, a functionality supported by all consensus clients. For example, `ethdo` connects to Prysm using the localhost address and the port defined by the `--grpc-gateway-port` flag. We used `ethdo` for generating the validator keys (signing and withdrawal) in the `create_validator` script described in Section 5.6.2 and the pre-generation of keys described in Section 5.6.1.

#### Deposit

Since execution clients can generate and broadcast transactions, we utilize Geth to create and broadcast deposit transcations. The transaction generated with Geth for a deposit is shown in Listing 5.13.

```
./geth attach --exec "eth.sendTransaction({
    from: eth.accounts[0],                    // The node's EOA
    to:'0x4242424242...4242',                 // Deposit contract address
    value:'32000000000000000000',             // 32 ETH
    gas:'120000',
    gasPrice:'15000000000',
    data:'$RAW_DEPOSIT_DATA'})"               // DepositData in raw format
    geth.ipc                                  // Provide access to the API
```

Listing 5.13: The command used to send a deposit transaction from an EOA (execution layer address), to the deposit contract (`0x424242...42`).

All fields are generally straightforward, except for the `data` field, which has the dual function of either deploying a smart contract or triggering a function call on an existing smart contract. As the deposit contract has already been deployed with the execution layer address `0x424242...`, we use its public `deposit` function to make deposits [62].

In Section 3.3.4, we introduced `DepositData`, the data required by the deposit contract for processing new validators. For a transaction to be valid and interact with the deposit contract, the `DepositData` must be in raw format (concatenated and in hexadecimal). The raw `DepositData` is generated by `ethdo` and occurs automatically after generating a validator's keys (signing and withdrawal) when using the `create_validator` script. This data is stored in a *deposit storage* directory for easy access. It's worth noting that the raw `DepositData` isn't generated for the genesis validators, as they are already included and active in the beacon state from genesis time.

The user must run the `make_deposits` script to make a deposit transaction. This script has one prerequisite: the validator keys (signing and withdrawal) must already have been generated using the `create_validator` script, such that raw `DepositData` is available.

Once the `make_deposits` script is executed, it checks the deposit storage for raw `DepositData` that has not yet been spent (i.e., used in a deposit transaction). The script displays the number of unspent deposits and allows the user to select the number of deposit transactions to make.

If the total number of unspent deposits is $m$ and the user chooses $n$ where $n \leq m$, the script will generate and broadcast $n$ deposit transactions using the command shown in Listing 5.13. These deposits will then be marked as spent to prevent multiple uses. The overall flow of the `make_deposits` script is shown in Figure 5.7.

Additionally, the script can be called with a `--all-deposits` flag to skip the selection process and automatically process all unspent deposits.

### Withdrawal Credentials Change

All genesis validators and validators added through deposits use the old withdrawal credentials style starting with `0x00`. To enable partial and full withdrawals, we need to update these credentials to comply with the new `0x01` style, as discussed in Section 3.3.6.

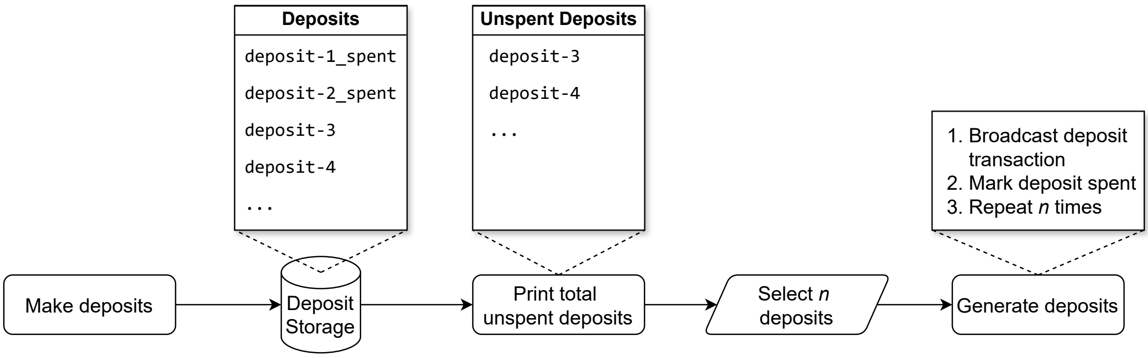

Figure 5.7: Flow diagram showing the process for making deposits. The deposit storage contains all the raw `DepositData` (`deposit-i`) for each validator generated beyond the genesis validators. Once a deposit transaction has been executed, the deposit status is updated to `deposit-i_spent`, indicating that a deposit transaction has been made for this validator. This update prevents the same `deposit-i` from being used again. The two tables, *Deposits* and *Unspent Deposits*, are included for visual clarity.

We provide a script, `convert_withdrawals_address`, to update the validators to the new style. The script uses `ethdo` to generate `SignedBLSToExecutionChange` messages [116]. These messages are then broadcast through the connected beacon node for inclusion in an upcoming beacon block. The command used in `ethdo` is shown in Listing 5.14, where `--mnemonic` parameter specifies the node's mnemonic, while the `--withdrawal-address` parameter provides the node's EOA in checksum format.

```
./ethdo validator credentials set
    --mnemonic="nature expand bone ... dinosaur"
    --withdrawal-address="0x3804bD29E8b614 ..."    // EOA in checksum format
```

Listing 5.14: The command used to create one or more `SignedBLSToExecutionChange` message(s). The mnemonic and EOA belongs to `node-1`.

When running `convert_withdrawals_address`, `ethdo` retrieves information from the consensus client about the validators managed by the node and verifies if they were generated with the provided mnemonic string. For all validators generated with this mnemonic and still using the old withdrawal prefix style `0x00`, `ethdo` will generate and broadcast a `SignedBLSToExecutionChange` through the consensus client for each one. After some time, these messages will be included in beacon blocks and processed, updating the validators' withdrawal credentials to the new style `0x01`, ensuring they become eligible for withdrawals.

### Exit

To enable the exiting of validators, we have developed the `exit_validator` script, which allows users to exit all validators associated with a specific mnemonic string or select individual validators for exit.

The exit script uses `ethdo` similarly to the withdrawal credentials change but creates `SignedVoluntaryExit` messages instead, which are also broadcasted through the connected beacon node [117]. Upon executing the `exit_validator` script, it lists all active validators (signing keys) associated with the connected node, displaying them by their derivation paths. Based on this list, the user can decide whether to exit all active validators or just a single validator. The process is illustrated in Figure 5.8.

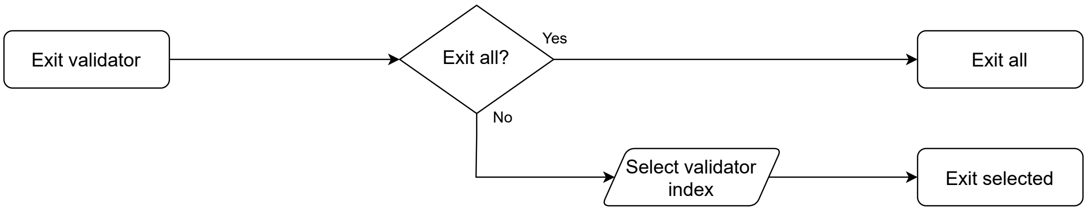

Figure 5.8: Flow diagram showing the process of exiting validators.

The `ethdo` command used for creating and broadcasting `SignedVoluntaryExit` is shown in Listing 5.15. To exit all active validators associated with the given mnemonic string, the `--path` argument is omitted. If the user wants to exit a specific validator, they must provide the derivation path for that validator's signing key to the `--path` argument. When the derivation paths are listed, each path has an associated index. Users only need to input this index into the terminal rather than typing out the entire derivation path.

```
./ethdo validator exit
    --mnemonic="nature expand bone ... dinosaur"
    --path="m/12381/3600/i/0/0"                    // Derivation Path i ∈ [0, ..., 4095]
```

Listing 5.15: The command used to create one or more `SignedVoluntaryExit` message(s). The mnemonic belongs to `node-1`.

## 5.7 Enabling Byzantine Behavior

To enable Byzantine behavior in our consensus client, we forked Prysm and introduced an additional flag, `--byzantine-behavior`. By incorporating this flag when starting Prysm, it is possible to check anywhere in Prysm's consensus client code if the flag is set through the conditional check demonstrated in Listing 5.16.

```
if flags.Get().Byzantine {
    ...  // Execute Byzantine behavior here
}
```

Listing 5.16: Check if the `--byzantine-behavior` flag is set.

We utilized this mechanism to allow nodes running validators to skip the aggregation selection process described in Section 3.4.5. When a validator controlled by a Byzantine node is selected to create and broadcast a `SignedAggregateAndProof`, the validator client requests an aggregated attestation from the consensus client. The consensus client returns an error instead of providing an aggregated attestation, preventing the validator from signing and broadcasting the aggregated attestation.

The selection process for creating and broadcasting a `SignedAggregateAndProof` is designed to be probabilistic, with the aim of selecting 16 validators for this task. The presence of at least one honest validator guarantees the proper completion of the process. However, this setup allows for the possibility of *free-riding*. In this context, free-riding refers to a validator's strategy of having their attestation included in a block (to earn rewards) without actively participating in the aggregation process.

The change we implemented in Prysm for enabling a node to skip selection validator aggregation responsibility can be seen in Listing 5.17.

```
// If validator is byzantine, it should not broadcast an aggregated attestation
if flags.Get().Byzantine {
     fmt.Printf("Byzantine: SubmitAggregateSelectionProof Byzantine Behaviour from
      Validator: %d, Slot: %d \n", validatorIndex, req.GetSlot())
     return nil, status.Errorf(codes.Internal, "Validator: %d is byzantine",
    validatorIndex)
}
```

Listing 5.17: The modified Prysm code in the `aggregator.go` file to allows Byzantine actors to bypass the aggregator selection process, enabling free-riding.

In Chapter 6, we perform an experiment to analyze free-riding behavior. Specifically, we compare the hardware usage of a client engaging in free-riding with that of an honest client.

## 5.8 Tracking Metrics

All three clients—execution, consensus, and validator—generate extensive logs during operation. These logs, while informative, often include a significant amount of data that may not be directly relevant. These logs are primarily utilized to verify the system's proper functioning. However, they do not offer statistical insights, necessitating alternative methods for data collection. To get more relevant data, we employ *Prometheus* [118] to gather the necessary data and *Grafana* [119] as the data visualization and presentation tool.

### 5.8.1 Grafana Dashboard

Grafana is configured with Prometheus as the data source, which is configured to scrape consensus client data from port 8080, validator client data from 8081, and execution client data from 6060 with a 15-second time interval. Figure 5.9 shows the dashboard for Geth.

This dashboard shows detailed information about the node, such as the blockchain state and transactions. It also shows network traffic and CPU usage. Geth provides a Grafana dashboard, which requires *InfluxDB* [120] as its data source. However, we use a similar dashboard designed for Prometheus.

The Grafana dashboard for the consensus client can be seen in Figure 5.10. In the dashboard for the consensus client, we see both clients' uptime, the number of validators they run, the peer count, and their earnings. More complex graphs show the total balance, individual validator balances and attestation, and the number of block proposals. Our dashboard is mainly built upon one provided by Prysm [121], with only minor modifications.

One of the main statistics we are focusing on is hardware usage, as we will be conducting some experiments on this later in Chapter 6.

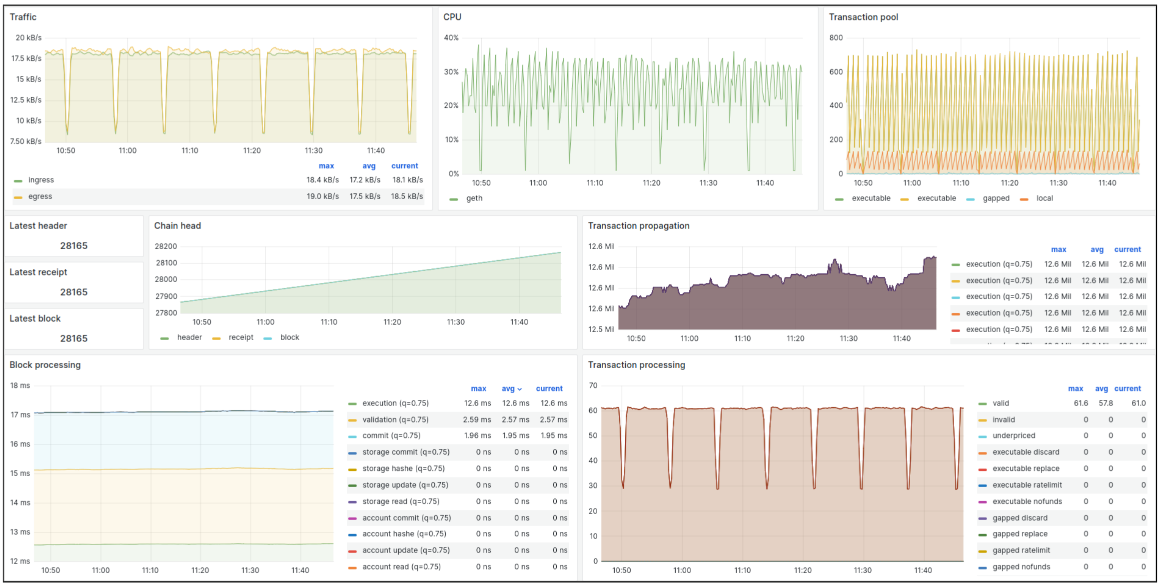

Figure 5.9: Grafana dashboard for monitoring the execution client.

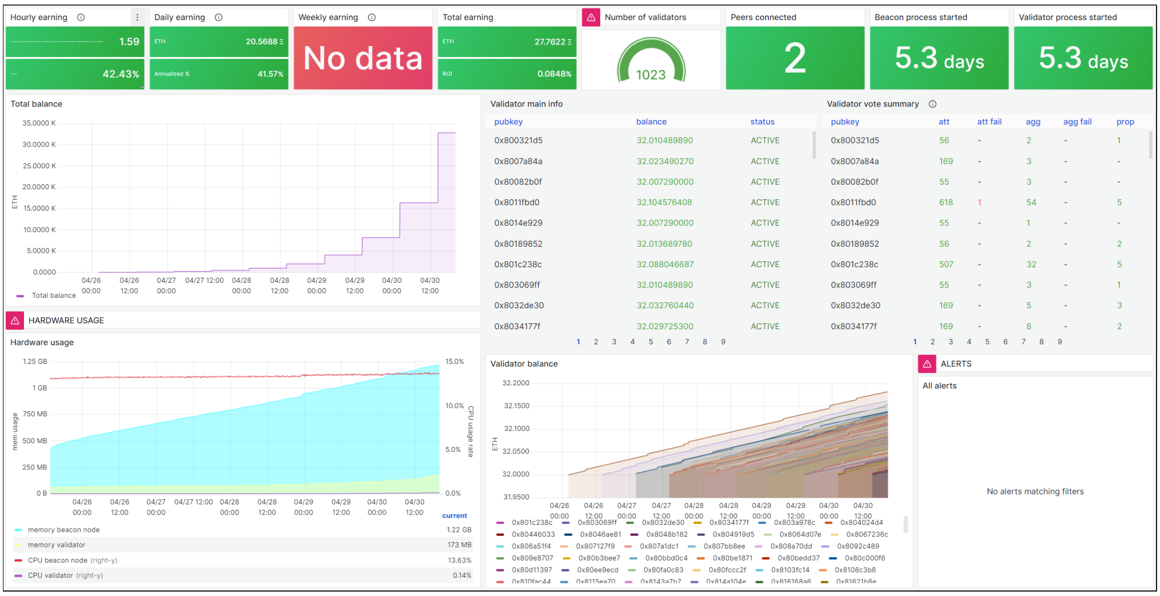

Figure 5.10: Grafana dashboard for monitoring the consensus and validator client.

### 5.8.2 Accessing the dashboard

Typically, the dashboard is accessed via `127.0.0.1:3000`. However, when running multiple nodes simultaneously and communicating with them exclusively through SSH, it is necessary to distinguish between each node. Hence, we increment the Grafana port by the node ID (`node-i` where $i \in [1, ..., 30]$). For instance, the dashboard for `node-1` will be hosted on port 3001, `node-2` on port 3002, and so forth. This approach is also done for Prometheus, where the base port is 9090.

**Note 5.4.** We adopt a similar approach to access the Execution API and Beacon API for the execution client and consensus client, respectively. Where we also incorporate the node ID into the port number.

# Chapter 6

# Experimental Evaluation

In this chapter, we outline the experiments conducted and their results. We start by briefly explaining what we aim to examine with the experiments, followed by a brief overview of the machines operating the application. Following this, we give a brief introduction to all the experiments performed. Subsequently, we follow a sequential approach, discussing each experiment and its result before proceeding to the next. Finally, we will conduct an analysis of all the experiments.

## 6.1 Goals

Our experiments aim to analyze the hardware usage of all clients, namely consensus, execution, and the validator, while focusing on two key metrics: CPU and memory usage. By examining the CPU and memory requirements, we aim to determine whether the hardware demands for running a node on a devnet differ from those on the mainnet. This comparison will help us determine the necessary hardware when running a devnet rather than participating in mainnet or one of the public testnets. By doing so, researchers can more easily determine if their equipment can handle running a devnet instead of relying on mainnet resource requirements, which may be excessive and unnecessary for their needs.

## 6.2 Setup

The application is deployed on the *BBChain cluster* at UiS. This cluster consists of 30 machines, each named *bbchain* and followed by an ID ranging from 1 to 30, i.e., *bbchain1*, *bbchain2*, and so forth. This simplifies manually entering an ID for each node we launch. Instead, we use the identifier already present in the hostname. Each of the 30 machines runs on Ubuntu 23.04, with an Intel(R) Xeon(R) E-2136 CPU at 3.30 GHz, with 6 cores and 12 threads. They have 32 GB of RAM and an INTEL SSD with 1.6 TB storage.

The machines are running Go version `1.21.6`. We utilize Prysm fork based on release `v5.0.30` [122] and Lighthouse binary version `v5.1.3`. Additionally, we use Geth version `v1.13.15`.

## 6.3 Experiments

The following experiments will be conducted in this section.

- The first Experiment 6.3.1 aims to establish a baseline for comparing hardware usage in subsequent experiments.
- The second Experiment 6.3.2 examines hardware usage when the execution client broadcasts transactions and compares it to the baseline experiment where no transactions were made.
- The third Experiment 6.3.3 investigates the impact of peer count on hardware usage by running the system with $5$, $10$, $15$, and $20$ peers, respectively.
- The fourth Experiment 6.3.4 analyzes hardware usage when dynamically increasing the number of validators over time.
- In the fifth and final Experiment 6.3.5, we use the Byzantine flag implemented in Section 5.7 to compare free-riding between an honest node and a Byzantine node.

Most experiments use both Lighthouse and Prysm as consensus clients to allow comparison.

In the experiments, we configure specific parameters for each node's consensus client by adjusting the number of peer connections they maintain. For Prysm, this parameter is set using the `--p2p-max-peers` flag, while for Lighthouse, it is managed through the `--target-peers` flag. These flags specify the maximum number of peers the clients should attempt to maintain. Any excess peers will be pruned to avoid unnecessary connections. Having an excessively high peer count can negatively impact the performance of a beacon node [123].

By default, Prysm is configured to maintain $70$ peers, whereas Lighthouse defaults to $100$. However, for the purposes of our experiments, we adjust these values to match the number of nodes in the experiments being conducted. We aim to minimize the overhead of discovering and maintaining additional peer connections, allowing us to focus on the experiment's core objectives with minimal external influences.

**Note 6.1.** Every slot had an associated block during the experiments, so the slot number always matched the block number. For instance, slot $i$ had block number $i$, and slot $i + 1$ had block number $i + 1$. This pattern continued until each experiment was concluded.

### 6.3.1 Establishing a Baseline

In this experiment, we aim to set a baseline for future reference. We accomplish this by deploying a devnet of $5$ nodes, each hosting $128$ validators. `Node-1` through `node-4` is running Prysm, while `node-5` is running Lighthouse. This setup allows us to compare hardware usage between the two clients and examine how running Prysm impacts the execution client Geth compared to Lighthouse.

### Result

Figure 6.1 and 6.2 show the hardware usage for a Prysm and Lighthouse node, respectively. Prysm's memory usage shows a steady increase in the beacon node (consensus client). This memory increase is because Prysm retains the last $n$ blocks (those since the last finalized checkpoint) in memory to handle potential reorgs. The beacon state also experiences some space growth when new blocks are proposed (multiple beacon states are also kept in memory in the case of reorgs). Similarly, the validator client's memory usage increases slightly when the node is selected for block proposing, making attestations, or serving as an aggregator. The most significant memory for the validator client increase occurs when the validator is selected as the block proposer, as it must generate the beacon block and send it to the beacon node. Additionally, since Prysm is written in Go, anything placed on the heap requires garbage collection.

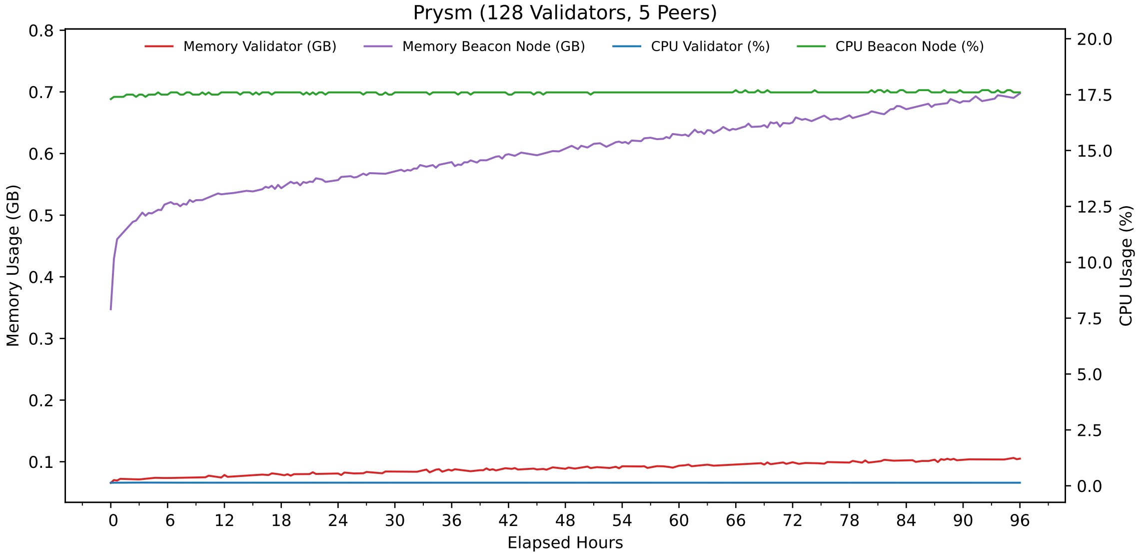

Figure 6.1: Hardware usage baseline for a Prysm node running $128$ validators.

Lighthouse follows a similar pattern in memory usage, increasing as the number of blocks grows and also keeps blocks and beacon states in memory in the case of reorgs. One thing that is especially evident for Lighthouse is how it handles memory compared to Prysm. It does not have a garbage collector but instead releases memory manually through Rust's ownership model, giving much more evident spikes in memory consumption than Prysm.

The most notable distinction between Prysm and Lighthouse in Figure 6.1 and Figure 6.2, respectively, is their CPU usage for the consensus client, with Prysm registering approximately $17.5\%$, and Lighthouse records about $1.0\%$.

While the CPU usage for the validator client remains exceptionally low for both Prysm and Lighthouse, Lighthouse's validator client appears to utilize slightly more CPU than Prysm's validator client, which consistently maintains a $0.1\%$ usage rate.

The CPU usage for both clients remained extremely stable throughout the $96$-hour running period, indicating that no outside factors affected the experiment.

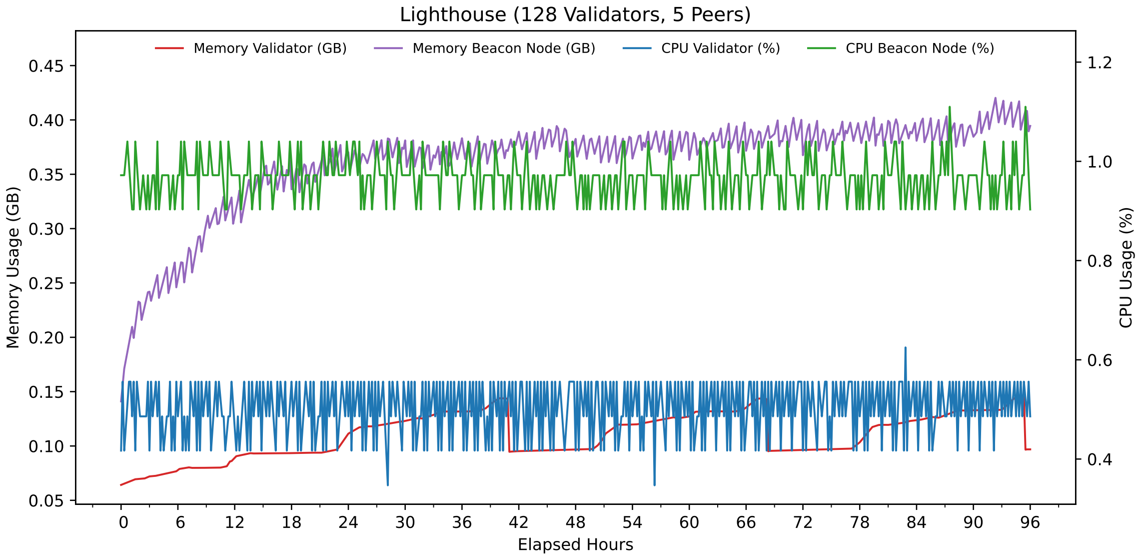

Figure 6.2: Hardware usage baseline for a Lighthouse node running 128 validators.

**Execution Client** The hardware usage for the execution client can be seen in Figure 6.3 with Prysm and Figure 6.4 with Lighthouse as their respective consensus client. Since all the beacon blocks contain `execution_payload` with zero transactions, the execution client Geth exhibits minimal resource consumption in both memory and CPU usage. The main difference is that Geth uses slightly more memory when running with Prysm than Lighthouse, while Geth's CPU usage is marginally higher when running with Lighthouse compared to Prysm.

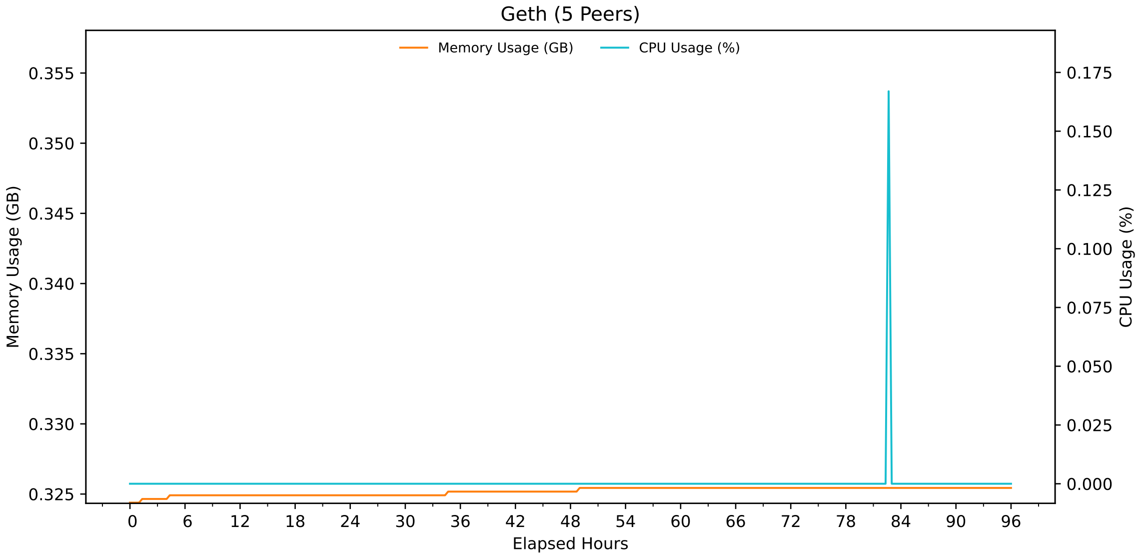

Figure 6.3: Execution client baseline usage. The corresponding consensus client is Prysm.

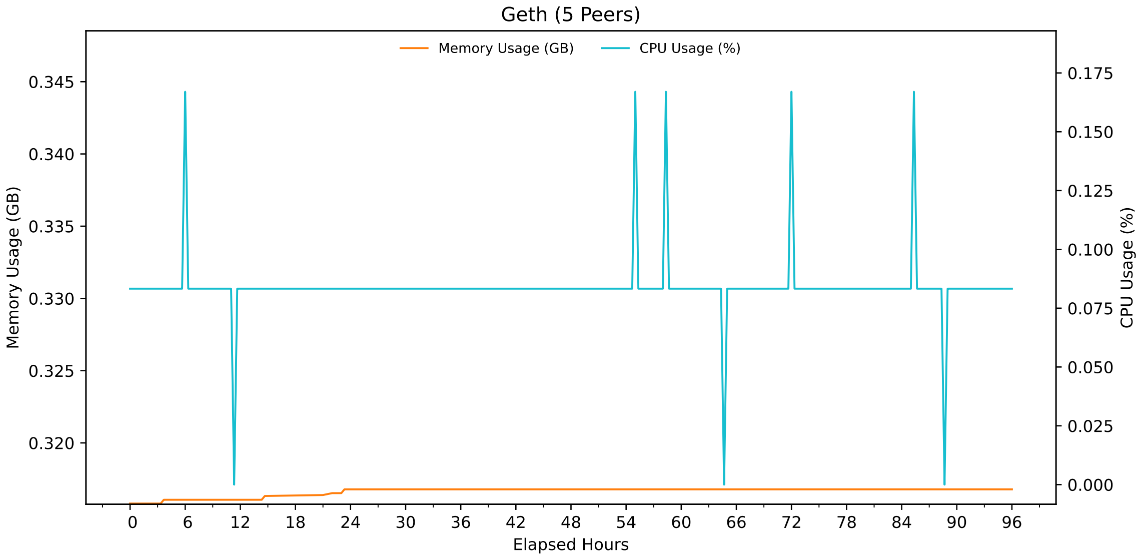

Figure 6.4: Execution client baseline usage. The corresponding consensus client is Lighthouse.

### 6.3.2 Generating Transaction Traffic

This experiment aims to simulate more network traffic by generating transactions to reach the target block size of 15 million gas as closely as possible. Since a single transaction has a gas cost of 21, 000, a block can contain up to 714 transactions before exceeding the target size. A single node can generate around 140 transactions per slot with our machines. With 5 nodes, we achieve approximately 700 transactions per slot. Maintaining around 700 transactions per block enables us to observe hardware usage under more realistic conditions. In this experiment, we broadcast transactions constantly during the full duration of the deployment.

#### Result

In this experiment, we utilize both Prysm and Lighthouse as consensus clients. Figure 6.5 shows the node running Prysm, while Figure 6.6 shows the node running Lighthouse. Both Prysm and Lighthouse follow the same pattern observed in the previous experiment. Prysm's CPU usage for the beacon client utilizes around 17%, while Lighthouse only utilizes around 1% of its CPU. One noticeable difference is the validator's CPU utilization, which is lower than the baseline recorded previously.

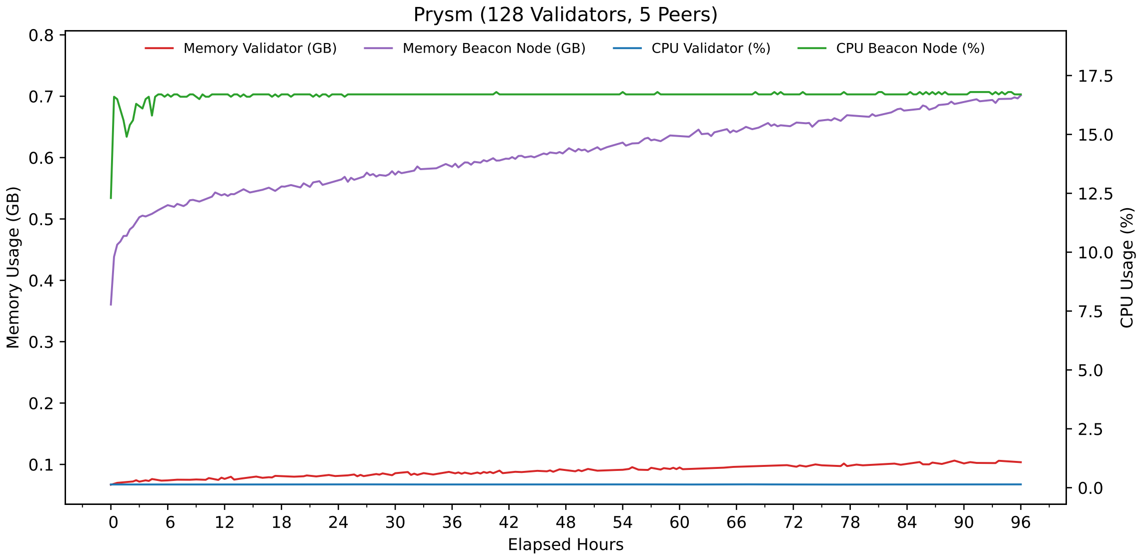

Figure 6.5: Baseline when transactions are generated and broadcast for a Prysm node.

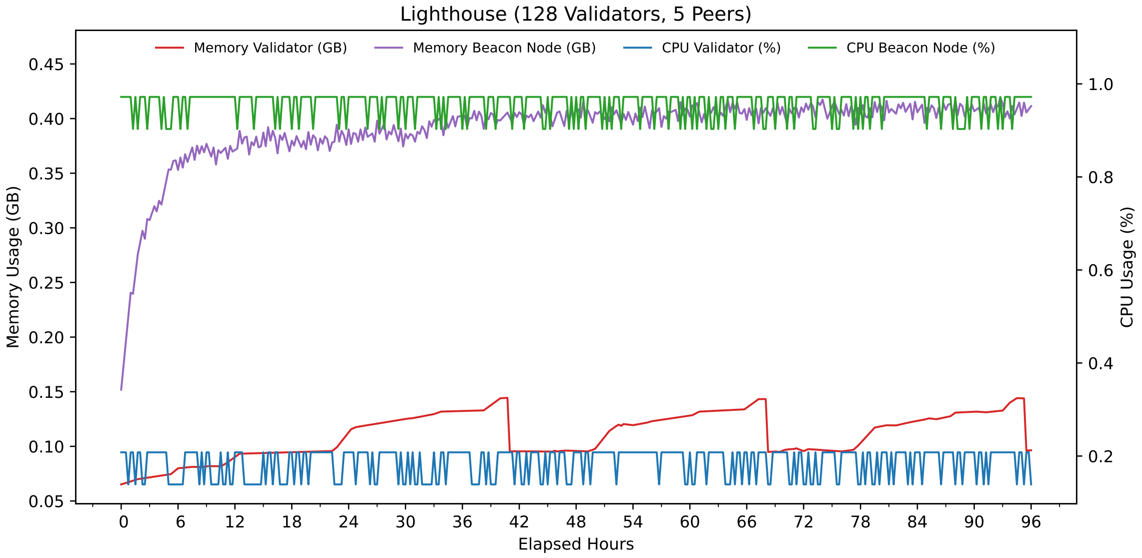

Figure 6.6: Baseline when transactions are generated and broadcast for a Prysm node.

To easily see the comparison between the hardware usage of this experiment and the baseline experiment, we introduce Figure 6.7 and 6.8. These figures show the results of a separate deployment conducted over three phases: an initial baseline for $24$ hours, followed by $24$ hours during which transactions were broadcast, and finally, another $24$ hours of normal baseline execution.

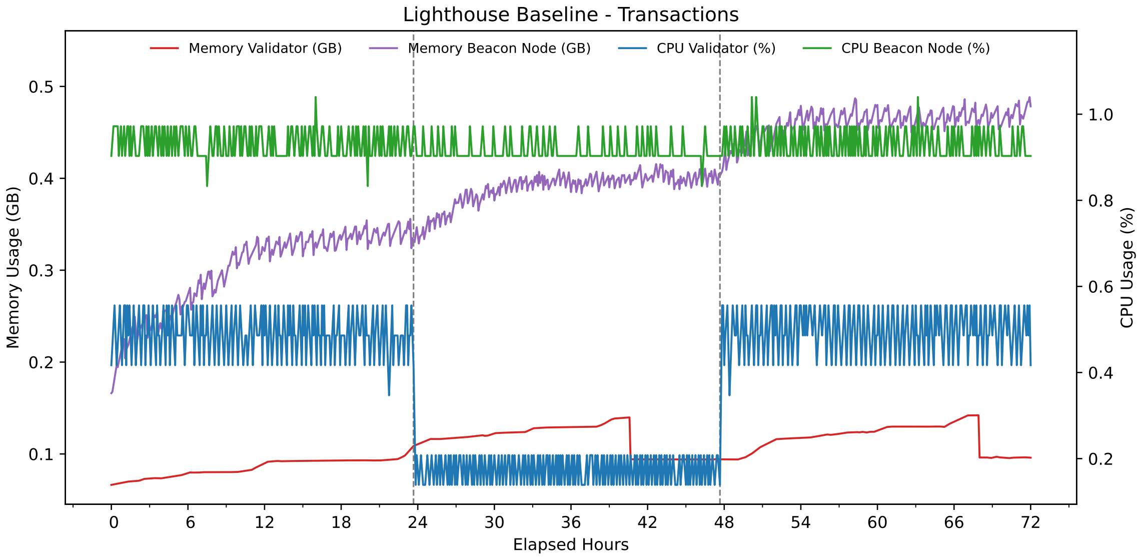

Figure 6.7: Lighthouse's hardware usage during a three-phase experiment consisting of toggling transactions on and off.

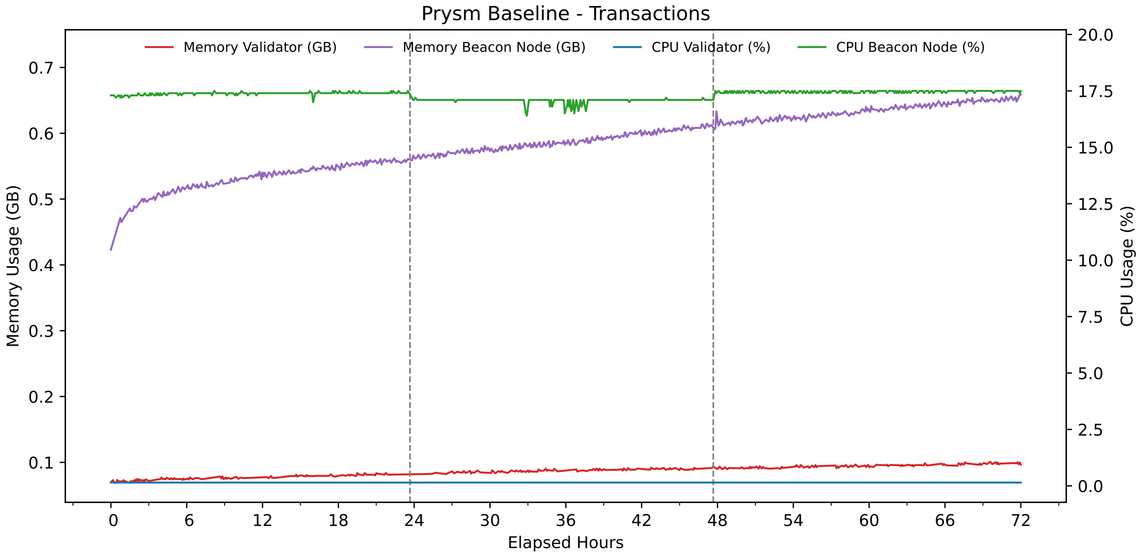

Figure 6.8: Prysm's hardware usage during a three-phase experiment consisting of toggling transactions on and off.

In Figure 6.7, we see a noticeable reduction in CPU usage for the validator client when transactions are broadcast in the background for the node running Lighthouse. Additionally, the memory usage of the beacon node stabilizes slightly, characterized by a lower frequency. In contrast, Figure 6.8 shows that the node running Prysm experiences only a minor decrease in memory usage for the beacon node.

As we know, blocks are proposed within a slot's first $4$ seconds. During this time, the proposer also waits for the `execution_payload` from its execution client. Broadcasting the block also takes time. When a peer receives this block, it must also send it to its execution client for validation, execute all transactions, update its EVM state, and ensure the state matches the state in the block. This extensive procedure occupies the beacon and validator client, preventing them from performing other tasks, and therefore, reduces their hardware utilization.

**Execution Client** Moving on, we examine the hardware usage for the execution clients. Figure 6.9 shows the hardware usage for the execution client on the node running Prysm. Similarly, Figure 6.10 presents the hardware usage for the node running Lighthouse as the consensus client. These figures represent the hardware usage during the initial experiment, during which transactions were continuously broadcast.

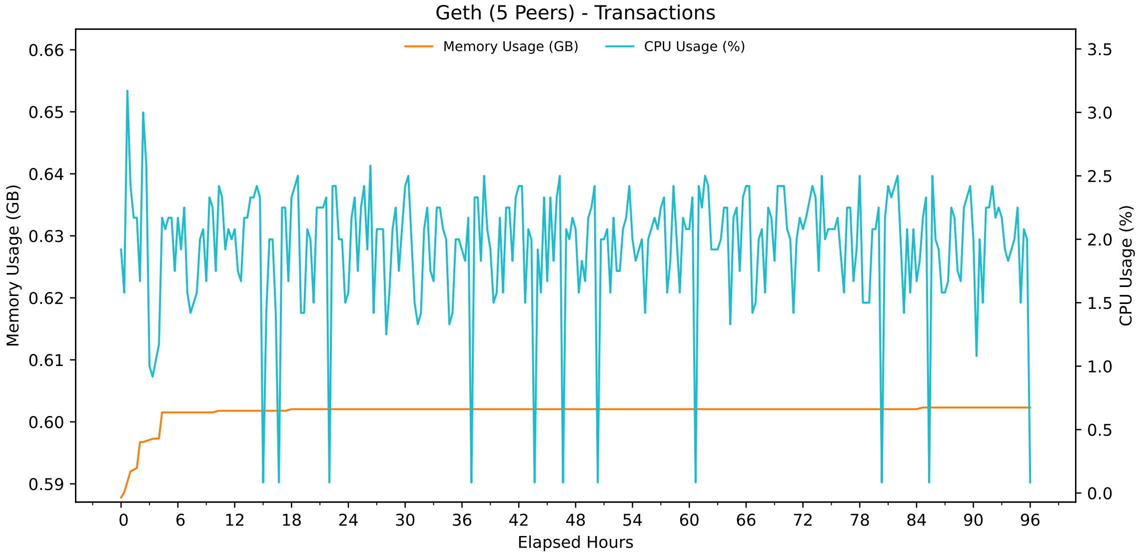

Figure 6.9: Baseline when transactions are generated and broadcast. The corresponding consensus client is Prysm.

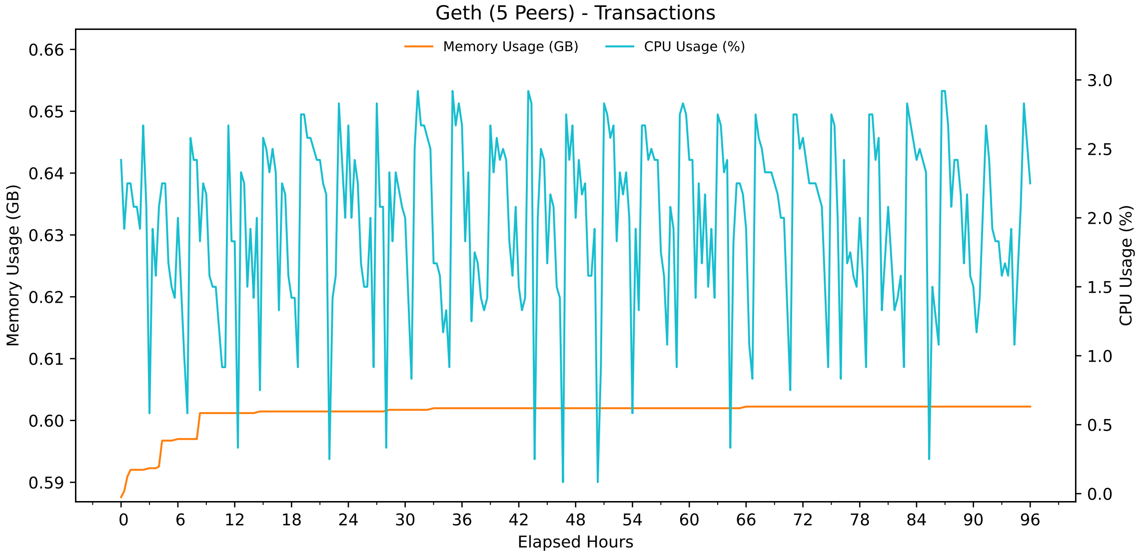

Figure 6.10: Baseline when transactions are generated and broadcast. The corresponding consensus client is Lighthouse.

Compared to baseline experiments, there is an obvious increase in the CPU and memory usage for the execution clients, as we observe in Figures 6.9 and 6.10. Comparing the execution client usage between nodes running Prysm and Lighthouse, the Prysm node utilizes slightly lower CPU usage overall. It hovers around 2% CPU usage, while the node running Lighthouse has higher spikes, nearing 3% utilization. Overall, the execution client's usage is barely influenced by which client the consensus layer operates.

### 6.3.3 Increasing the Peer Count

In this experiment, we analyze the hardware usage when launching a devnet with a varying number of peers. We observe the first 24 hours of deployments with different configurations: 5 nodes, 10 nodes, 15 nodes, and 20 nodes. In all these deployments, only the first 5 nodes are assigned 128 validators each, while the remaining nodes run without validators. By keeping the genesis validator set constant, we can avoid the increased hardware usage associated with a higher validator count. This approach helps manage the beacon state's growth and reduces the number of attestations that need to be broadcast. It allows us to focus solely on how hardware is affected by a higher peer count, minimizing external factors as much as possible.

This experiment only examines the consensus clients, as the baseline experiment covers the execution client's usage.

#### Result

When nodes report their peer count, they do not include themselves. We refer to peer count as the total number of nodes in the system, not the reported peer count of a specific node. For example, `node-1` may report 9 nodes, but we refer to the peer count as 10.

Figures 6.11, 6.12, 6.13, and 6.14 shows the Prysm and Lighthouse usage during the first 24 hours of deployment with the different peer counts. Figure 6.11 is from the same experiment as the baseline experiment but only shows the first 24 hours.

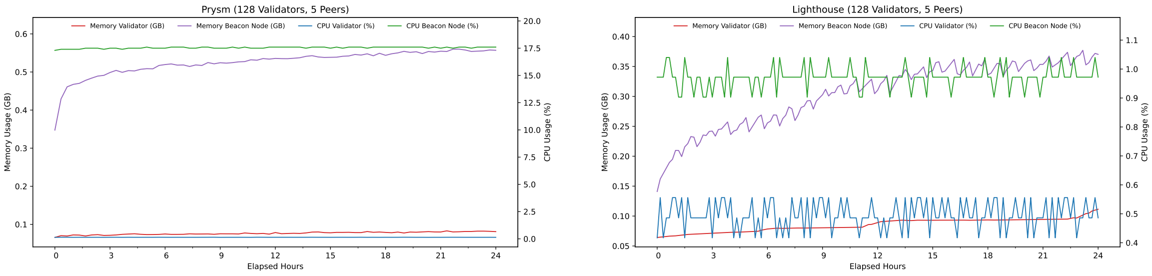

Figure 6.11: Hardware usage for Prysm and Lighthouse with number of peers equal 5.

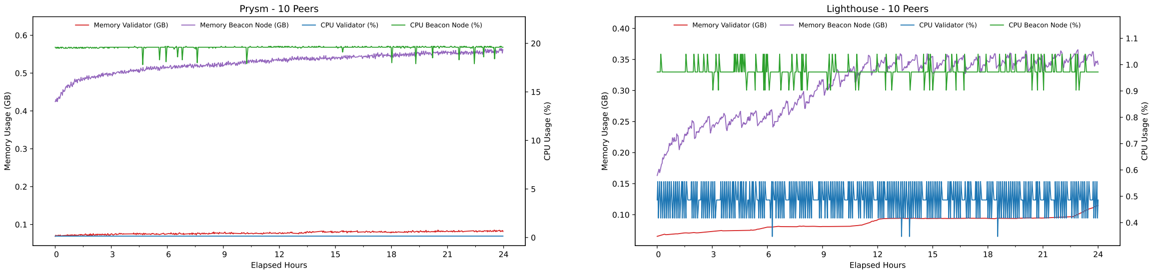

Figure 6.12: Hardware usage for Prysm and Lighthouse with number of peers equal 10.

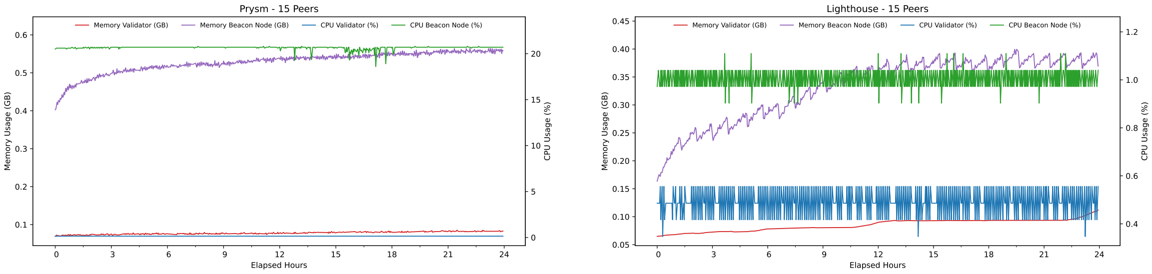

Figure 6.13: Hardware usage for Prysm and Lighthouse with number of peers equal 15.

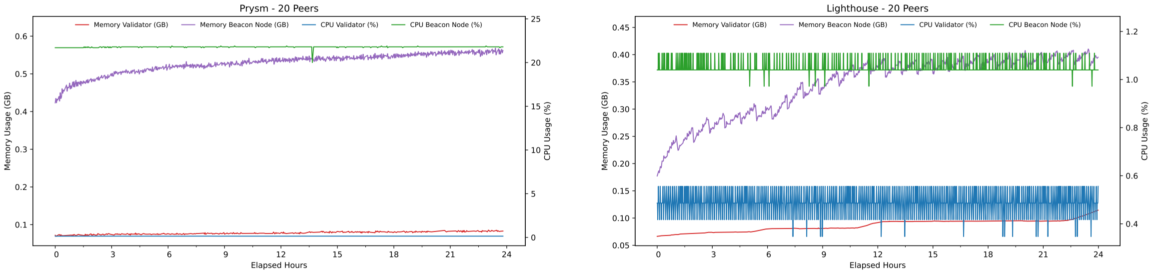

Figure 6.14: Hardware usage for Prysm and Lighthouse with number of peers equal 20.

For the Prysm nodes, the CPU usage fluctuates from the lowest utilization of 17.5% to the highest of 22.5% for the beacon node. Lighthouse remains more consistent throughout the different peer counts, with a minimal increase in CPU usage with a higher peer count.

### Peer Count for Nodes with Zero Validators

As mentioned, to keep the experiment as fair as possible, we do not run any validators on any of the nodes apart from the 5 first ones. This also allows us to see how the peer count affects hardware when we are not running validators. In Figures 6.15 and 6.16, we look at `node-10`, which does not run any validators. The experiment with 5 nodes is not included as all the participating nodes had a genesis validator count of 128.

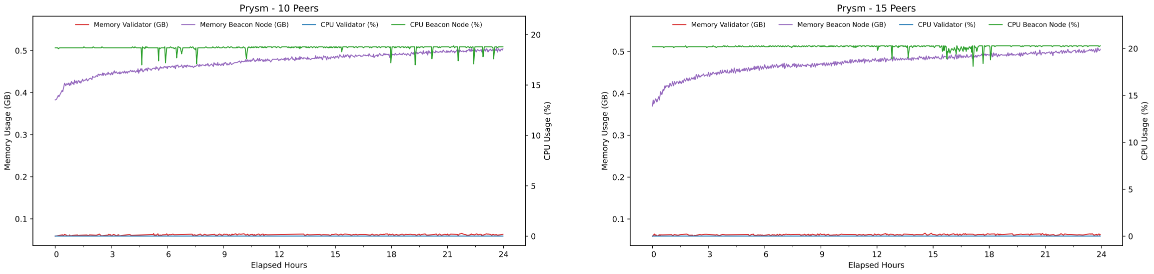

Figure 6.15: Hardware for nodes running 0 validators. Left: Peer count 10. Right: Peer count 15

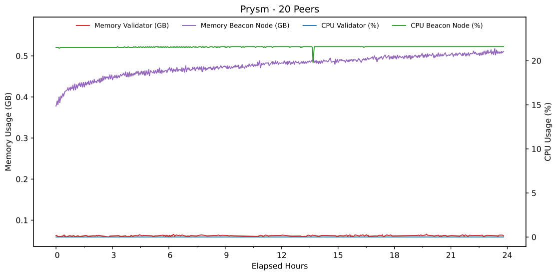

Figure 6.16: Hardware for nodes running 0 validators with peer count 20.

As expected, the hardware usage for the consensus client for the nodes running 0 validators remains almost identical to those with 128 validators. The only observable difference is in the CPU and memory usage of the validator client. Since we do not run any validators, the usage by the validator client is minimal. We study the effects of an increasing validator count in the following experiment.

### 6.3.4 Increasing the Validator Count

In this experiment, we aim to assess the impact of the number of validators on a node's performance. We deploy a total of 3 Prysm nodes: `node-1` is configured with 1 genesis validator, `node-3` runs with 4096 genesis validators, and `node-2` starts with 0 genesis validators and gradually increases its validator count over time. The validator count of `node-2` is incremented by $2^n$, where $n \in [0, ..., 11]$, every 12 hours. `Node-2` validator counts progresses as follows: 0, 1, 3, 7, 15, 31 and so forth, until it reaches a total of 4095 validators. After `node-2` reached 4095 validators, the experiment ran for approximately an additional 100 hours to better understand how the distribution of validators would affect the node's hardware usage.

For this experiment, we utilize the beacon chain parameters detailed in Listing 5.3 from Section 5.3.1 to speed up the process of adding validators.

#### Result

The three Figures 6.17, 6.18 and 6.19 are for `node-1`, `node-2` and `node-3`, respectively. Since these nodes ran much longer than the other experiments, we can clearly observe when the garbage collector frees memory. Additionally, the beacon node's memory usage increased noticeably after 96 hours, rising to 1 GB compared to 0.7 GB in the baseline. This increase is due to the greater number of validators (starting with 4097 compared to 640 in the baseline), which means more attestations need to be broadcast, stored in the beacon blocks, and the beacon state contains more validators. Consequently, epoch processing requires more work.

Although all the nodes require the same amount of beacon node memory and show minimal differences in CPU usage, there is a significant disparity in the memory consumption and CPU usage of the validator clients, especially for `node-3` and `node-2` after some time. The dashed vertical lines in Figure 6.18 mark the times when `node-2` made deposits for new validators.

The significant rise in memory consumption seen in Figure 6.19 for the validator client of `node-3` after just a few hours is due to its control over 4096/4097 genesis validators. This control enables `node-3` to produce nearly all blocks, broadcast numerous attestations, and frequently handle attestation aggregation duties. Consequently, it must retain more information in memory for its validator client than the other nodes.

As `node-2` acquires more validators and its percentage of all active validators increases, while `node-3`'s percentage decreases, both nodes must share the responsibility of proposing blocks. However, the increased number of attestations broadcasted by both nodes leads to a rapid rise in memory consumption for their validator clients.

An interesting observation when studying Figure 6.18 (`node-2`) and Figure 6.19 (`node-3`) is the CPU usage for their respective validator clients. Initially, `node-3` with 4096 active validators has the highest percentage of active validators, leading to increased CPU usage. This is because having the most active validators means the node proposes a higher percentage of blocks, thereby intensifying the CPU workload.

Over time, as `node-2` gradually increases its validator count, it acquires a larger percentage of the total active validators. Consequently, we observe a decrease in validator client CPU usage for `node-3` as its share of validators decreases. This shift reduces the number of blocks proposed by `node-3`'s validators, thus lowering its CPU usage.

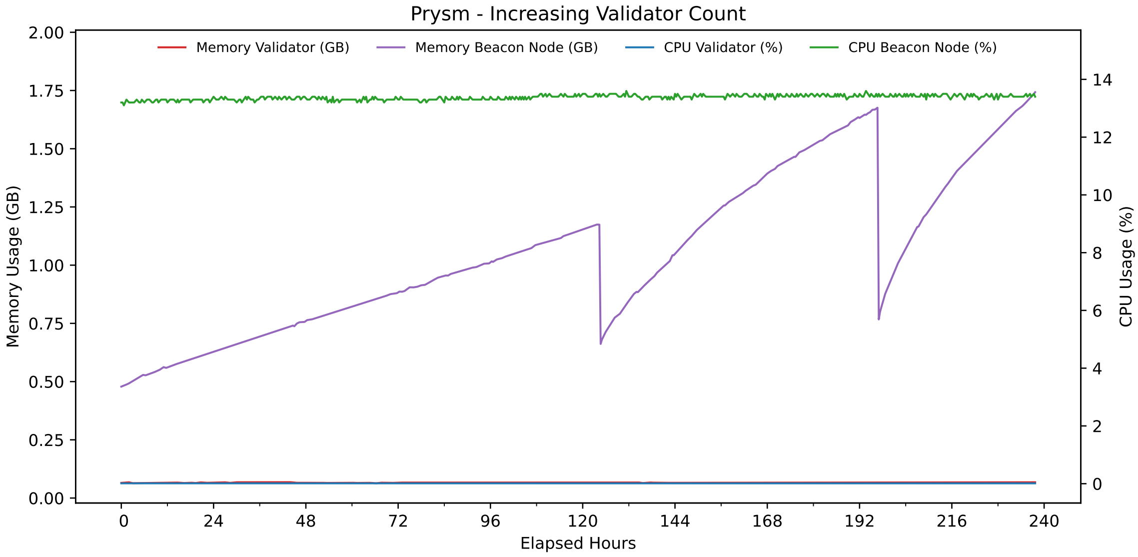

Figure 6.17: Node 1's hardware usage with a stable validator count of 1.

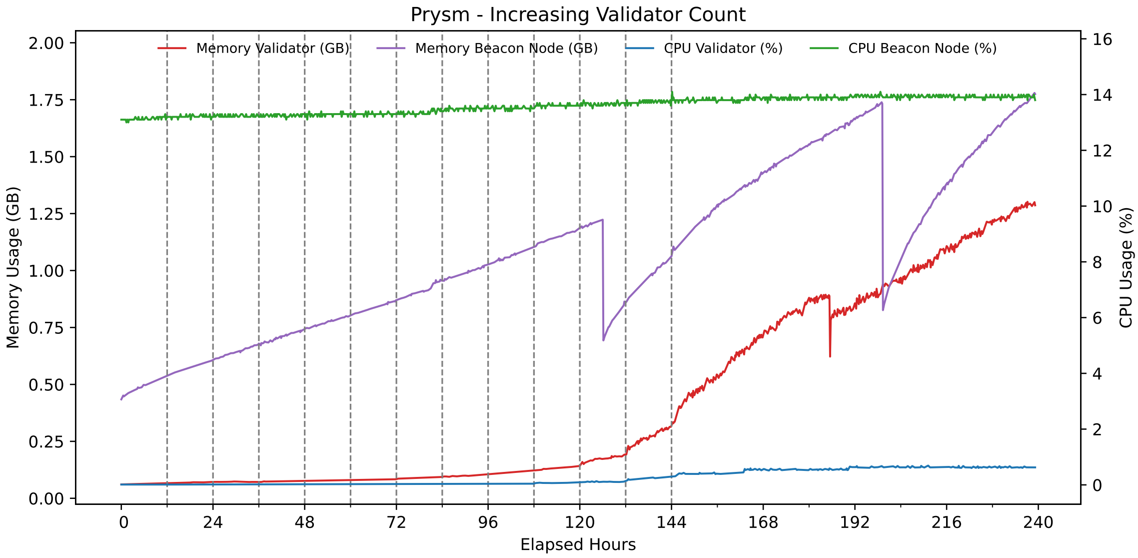

Figure 6.18: Node 2's hardware usage when increasing the validator count every 12 hours. It began with 0 validators and continued until it reached 4095 validators.

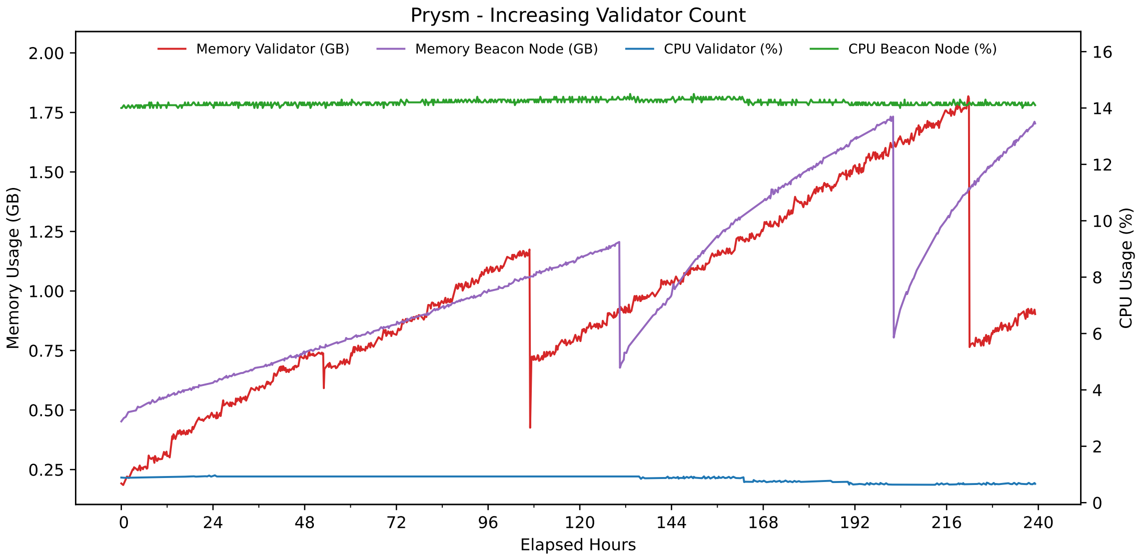

Figure 6.19: Node 3's hardware usage with a stable validator count of 4096.

### 6.3.5 Byzantine Node Performance

In this experiment, we compare the performance of a Byzantine Prysm node to that of a regular, honest Prysm node. Utilizing the `--byzantine-behavior` flag in the modified Prysm code, we can designate a single node as Byzantine while the remaining nodes operate as regular nodes by omitting this flag. The Byzantine node skips its duty when it is selected to aggregate attestations, as described in Section 5.7. The experiment includes five nodes, each with 128 genesis validators, all running Prysm. Our focus is solely on the hardware usage of the consensus client and the validator client, as the execution client's hardware is not affected and remains equal to that of the baseline experiment. The primary objective of this experiment is to evaluate the extent to which a Byzantine node can achieve resource savings through free-riding behavior.

### Result

Figures 6.20 and 6.21 show the non-Byzantine and Byzantine hardware usage, respectively. The CPU and memory usage for the consensus and validator clients is similar across both nodes. This suggests that the resources saved for free-riding are minimal or nonexistent. However, there is a higher risk of missing attestation rewards if one of the Byzantine node's validators is the sole validator selected to perform attestation aggregation for a particular committee.

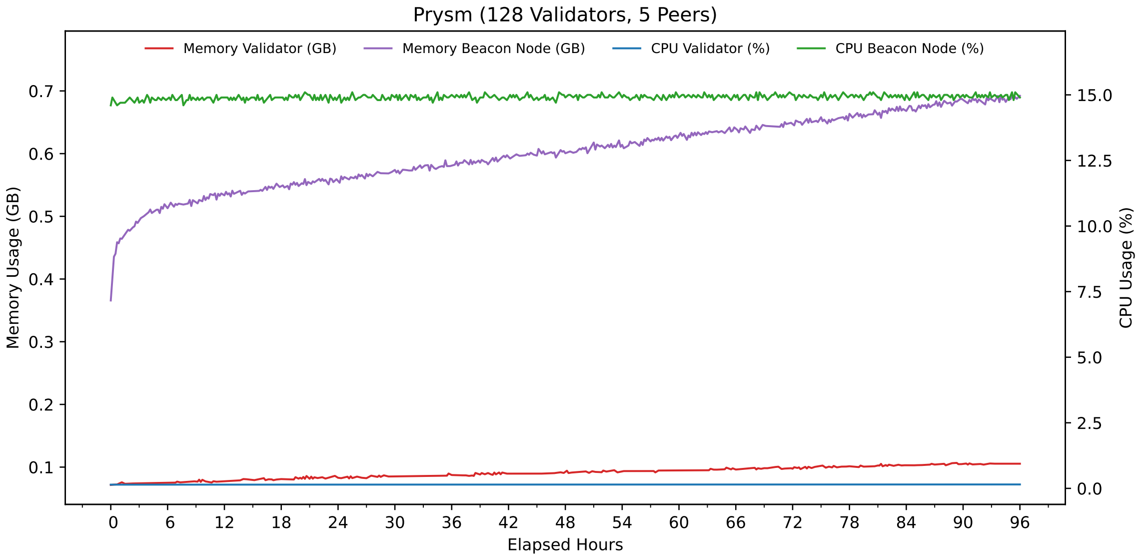

Figure 6.20: Non-Byzantine hardware usage with a stable validator count of 128.

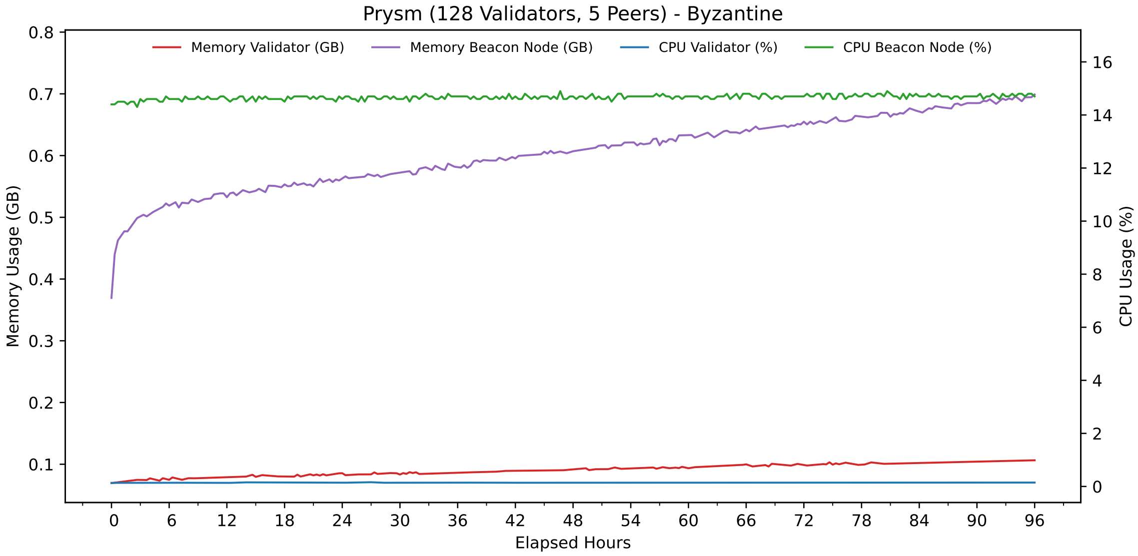

Figure 6.21: Byzantine hardware usage with a stable validator count of 128.

## 6.4 Experimental Analysis

We begin by presenting observations made during the experiments. Following this, we reflect on the goals set regarding hardware usage requirements for a devnet compared to the mainnet recommendations. Lastly, we briefly comment on some of the experiments we conducted.

### 6.4.1 Observations

Across all experiments, we see a consistent pattern for both beacon nodes. Prysm has stable CPU usage for both the consensus and validator clients, with the consensus client memory usage increasing steadily as blocks are kept in memory and as the beacon state grows to combat potential reorgs. In the longer experiment with increasing validator count, we observe sharp drops in the beacon memory upon reaching a certain threshold before gradually increasing just above the initial drop point, repeating this cycle continuously.

Lighthouse has a significantly lower CPU usage for the consensus client, generally hovering at around $1\%$. The validator memory also follows a distinct pattern across all experiments, releasing memory at a certain point. Interestingly, the first drop always happens after $40$ hours of deployment in all experiments.

Overall, when examining Geth's hardware usage, it is evident that its resources are minimal. The CPU remains at $0\%$ throughout all experiments, barely approaching $1\%$ at times. Memory is hovering at the $0.2-0.3$ GB range. Only during transaction generation and broadcasting do we observe spikes in both CPU and memory utilization. Although, the usage still remains relatively low, at $0.6$ GB memory and $2\%$ CPU.

For our experiments, Lighthouse performed better than Prysm regarding hardware usage. Lower memory usage for both consensus and validator clients and lower CPU utilization for the consensus client. Prysm's validator client's CPU usage barely utilizes less than Lighthouse's.

We couldn't perform certain experiments, like increasing the validator count, using Lighthouse due to reasons discussed later in Section 7.4.2. It would have been interesting to conduct an increasing validator count experiment with Lighthouse to understand how Lighthouse nodes handle a larger active validator count and control a larger percentage of all active validators.

### 6.4.2 Mainnet Comparison

Our objective was to determine whether the required resources for a devnet are equivalent to those for the mainnet. Table 6.1 presents the recommended hardware requirements for Geth, Prysm, and Lighthouse.

Note that for Lighthouse, the requirements assume an execution client is also running. Geth's disk space recommendation is somewhat outdated, as it was written in late $2022$. At that time, they recommended $2$ TB of SSD, including a consensus client. However, an updated, unofficial community agreement is that $4$ TB SSD will be required mid-$2024$ [124].

Table 6.1: Recommended System Requirements for Geth [125], Prysm [126] and Lighthouse [127] for running a node on mainnet.

| | **Geth** | **Prysm** | **Lighthouse** |
|---|---|---|---|
| **CPU** | Quad-core CPU | Intel Core i7-4770 or AMD FX-8310 or better | Quad-core AMD Ryzen Intel Broadwell, ARMv8 or newer |
| **Memory** | 16 GB | 16 GB | 32 GB |
| **Storage** | 2 TB SSD | 100 GB+ of SSD | 2 TB SSD |

The table shows that 32 GB of memory is recommended when running an execution client (Geth) and a consensus client (Prysm or Lighthouse) for mainnet.

Based on the results of all experiments, it appears feasible to run a devnet on lower-resource machines. A machine with 8 GB of memory dedicated solely to the full node should be sufficient for normal devnet operations. Additionally, a somewhat modern CPU with at least 4 cores is adequate for a devnet environment.

However, it's important to note that experiments with many validators can significantly increase memory requirements. This was evident in the experiment with an increasing validator count, where the consensus client of all the participating nodes requires more memory. The nodes with many validators will also see high memory usage for their validator client.

Performing additional experiments with a greater variation in the total number of validators and distribution of validators among the nodes following a normal distribution would likely provide valuable insights. Combining some of the experiments, such as increasing validator with more peers, could also be highly beneficial.

### 6.4.3 Runtime and Transaction Load

It became evident that many experiments would have benefited from a longer runtime. For instance, only in the experiment involving an increasing validator count (Section 6.3.4) did we observe periodic drops in the memory usage of consensus and validator clients. Unlike the other experiments, this occurred because it ran long enough to trigger these drops.

The transaction load experiment would also benefit from a more realistic transaction load. One issue with our generated transaction load was that each client used its own assigned EOA and sent a small amount of Wei to itself. This had minimal effect on the EVM state, as only the already included accounts needed minimal updates (changing balance and updating nonce).

Incorporating real transactions from the Ethereum mainnet or a testnet would have allowed us to achieve a more accurate comparison of hardware usage for the execution client. Real transactions would have caused the state to grow by including more EOAs, deploying smart contracts, and executing smart contract operations. This approach could have provided a more realistic assessment of hardware usage related to an execution client.

# Chapter 7

# Discussion

In this chapter, we begin by reflecting on the application we developed and critically analyzing its shortcomings. We will delve into the design decisions made throughout the implementation process. Additionally, we will review the tools available for interacting with Ethereum, comparing the approaches and solutions previously discussed for deploying a private devnet. We will also address the challenges and issues encountered during the application's development and the solutions implemented to overcome them. Finally, we propose directions for future work, highlighting potential enhancements to improve the application further.

## 7.1 Critical Reflection

Reflecting on the objectives set at the beginning of this thesis, we have successfully achieved our primary goal of creating an application that simplifies the setup of an Ethereum private network environment. However, several aspects warrant critical reflection to identify limitations and areas for improvement.

We start by discussing the initial phase of learning and understanding Ethereum and how it helped develop the application. Next, we reflect on our initial approach with the clients we used and consider how we could have approached it differently. We also address the potential benefits of dedicating more time to gathering a broader range of deployed node metrics. Additionally, we present a paper we initially intended to use to compare our findings and discuss the challenges and insights gained from this comparison. Finally, we highlight our experiences in receiving feedback and assisting others in deploying their own Ethereum nodes within various communities.

### 7.1.1 Learning from Documentation and Specifications

One thing that greatly helped us understand Ethereum's proof-of-stake protocol was reading specific documents, such as the consensus specifications, Vitalik Buterin's annotated specifications, and Ben Edington's book, *Upgrading Ethereum*. These resources were incredibly helpful in deepening our understanding of the protocol. This knowledge significantly aided our troubleshooting and decision-making processes, such as the process for dynamically adding validators, exiting validators, enabling withdrawals, and knowing what beacon chain parameters to adjust to meet our specific demands.

Gaining an understanding of Ethereum's proof-of-stake protocol through the specifications was particularly beneficial when working with Prysm's source code to implement Byzantine behavior. Given the vast and complex nature of Prysm's codebase, having a solid theoretical foundation was invaluable for identifying which files to modify and comprehending various components and their functions.

Overall, reading documentation and specifications proved essential, highlighting the importance of a solid theoretical foundation before making design choices and starting practical implementations.

### 7.1.2 Client Diversity

In the early phases of the project, we concentrated primarily on Prysm, the most widely used consensus client. This focus was mainly because Prysm is written in Go, the programming language we are most comfortable with. As described in Section 4.3, the Prysm team has their own guide for setting up a private devnet. Although this guide was somewhat lacking, it provided a useful starting point.

Later in the project, we integrated support for Lighthouse into our application. The Lighthouse team has also made a guide for setting up a private devnet, which offered valuable tips and a different perspective on setting up a devnet compared to Prysm.

We regret not exploring other consensus clients, such as Teku, Nimbus, and Lodestar, and not investigating Lighthouse earlier. Doing so would have been extremely beneficial, providing us with deeper insights into deploying a devnet and understanding the various available options.

Most clients have their own guides for setting up a devnet, focusing solely on their respective clients. These guides would have helped us understand different clients' operations and deployment requirements in a devnet environment, providing a broader perspective on setting up and deploying a devnet.

Including a broader range of clients can benefit researchers who want to use the project for their experiments, providing a more realistic client distribution similar to mainnet. Greater client diversity would also have been advantageous in our evaluation of hardware usage associated with the clients, offering a more comprehensive overview of their performance.

In summary, thoroughly exploring all the different consensus clients and their configurations early on would likely have enabled our application to support the client diversity that Ethereum aims to achieve.

### 7.1.3 Metrics and Monitoring Tools

Regarding hardware metrics, we primarily focused on memory and CPU usage. However, a thorough evaluation should include disk space, read/write operations, and network utilization. These metrics are crucial for understanding the full scope of resource consumption and system performance.

Finally, another area for improvement was the utilization of monitoring tools. We relied on existing Grafana dashboards but did not customize them extensively to fit our needs. Developing a fully customized dashboard that incorporates all the clients (execution, consensus, and validator) could have provided more valuable insights and better visualized the interactions between these components.

Overall, while the thesis has achieved most of its main objectives, these reflections highlight areas where additional focus and improvement could further enhance the application.

### 7.1.4 Experimental Evaluation Comparison

One goal of our study was to compare our findings with those of other research on hardware utilization of different clients. A notable paper by Cortes-Goicoechea et al. [128] provides a comprehensive comparison of hardware utilization across various consensus clients. However, this paper has two inherent flaws that make it difficult to compare their findings with ours.

First, the paper was published in late 2020, so it is relatively outdated and was conducted on the Medalla testnet. This was before the beacon chain was even deployed on Ethereum's mainnet, which occurred on December 1, 2020, as shown in Figure 3.2. Second, the study's objective was different from ours: *"The objective of this study is to monitor specific metrics in order to understand the behaviour and performance of the clients when initialized to sync to the Eth2 network."* [128]. In contrast, our focus did not include syncing new nodes to an existing chain by requesting blocks from other nodes. Instead, we initiated a blockchain where every node participated from the genesis time, determined by the machine's local Unix time.

Because of these two differences, especially the second one, comparing our findings to those in the paper didn't make sense. We measured the same metrics but under two vastly different conditions. However, one consistent observation was the memory consumption pattern for Prysm. It reached a threshold, triggered garbage collection, and then increased its threshold for the next cycle until it stabilized. We also observed this pattern clearly in the increasing validator count experiment.

### 7.1.5 Community Contributions

Throughout the project, we actively participated in communities related to the different consensus clients, such as the official Discord servers for Prysm and Lighthouse. This was beneficial, as it was helpful to receive assistance from developers within these communities to set up the devnet.

Moreover, we were also able to contribute by aiding others in setting up their devnets, addressing their questions, and offering practical tips. We provided support on various issues on GitHub, including validator management and modifying the source code for certain tools.

Additionally, users have expressed notable interest by requesting access to our project on GitHub, highlighting the relevance of the thesis.

## 7.2 Exploring Available Tools

Not only is the choice of clients themselves vast, but there are also many tools used to interact with Ethereum. We will now discuss some of the tools we previously used but ended up selecting another instead. We also discuss several validator management tools that are available. We start by discussing the tool used for generating the Genesis state.

### 7.2.1 Genesis State Tools

Generating the Genesis state is a crucial step in setting up a devnet. Both Prysm and Lighthouse offer tools tailored for this purpose, named `prysmctl` and `lcli`, respectively.

Initially, we utilized `prysmctl`. However, by recommendation from a Prysm developer and Lighthouse developer, Chong-He, we transitioned to `eth2-testnet-genesis`. However, by recommendation from both Prysm and Lighthouse developers, we transitioned to `eth2-testnet-genesis`. One of this tool's biggest advantages is that it allows us to import the bootstrapped validators into a validator client. This is because `eth2-testnet-genesis` allows us to specify the mnemonics it should use to generate the validators. We can then use `ethdo` to recreate the keys (signing and withdrawal) by generating them through `ethdo` and importing the signing keys into the validator client, as previously detailed in Section 5.6.1. This was impossible when using `prysmctl`, as the mnemonic is unknown for the generated validators. Because of the transition from `prysmctl` to `eth2-testnet-genesis`, we can initiate withdrawals and exits for validators generated at genesis time. We can also use Lighthouse as the beacon node since the validators can be imported into a Lighthouse validator client, which was impossible when they were created by `prysmctl`.

Adopting `eth2-testnet-genesis` thus enhanced the flexibility and functionality of our system, allowing for more robust and versatile devnet configurations.

### 7.2.2 Deposit and Validator Management Tools

In our exploration of tools for deposit generation and validator management, we assessed the capabilities of several options:

- `staking-deposit-cli` [129]
- `eth2-val-tools` [130]
- `ethdo`

`staking-deposit-cli` is the most utilized tool for deposit data generation, particularly on the mainnet. However, its utility is limited to mainnet and a select few testnets. This renders it impractical for our purposes without resorting to performing source code modifications.

`eth2-val-tools`, developed by the same team responsible for `eth2-testnet-genesis`, provides an experimental tool for managing validators. Beyond facilitating deposit data generation, it offers a variety of tools for validator management.

`ethdo`, previously discussed in Chapter 5 is similar, providing various management operations for validators.

### Modifying the Source Code

One of the earlier approaches to dynamically adding validators to the system was to utilize the `staking-deposit-cli` tool in combination with `ethdo`. To do this, we first have to perform the required code modifications.

The reason why `staking-deposit-cli` is not supporting a private devnet is due to the settings of the devnet being unknown to it. Unlike a public network, a devnet is often launched under different settings and conditions. This means the initial genesis settings for a devnet vary, as opposed to the constant settings for public testnets. The `staking-deposit-cli` tool utilizes a `BaseChainSetting`, which consists of the network's genesis fork version and genesis validator root, see Listing 7.1.

```
class BaseChainSetting(NamedTuple):
    NETWORK_NAME: str
    GENESIS_FORK_VERSION: bytes
    GENESIS_VALIDATORS_ROOT: bytes
```

Listing 7.1: BaseChainSetting specification.

Therefore, we defined our own `BaseChainSetting`. To do this, we implemented a function that fetches the required parameters from the running devnet. This was done by querying the beacon chain endpoint to retrieve the `GENESIS_FORK_VERSION` and `GENESIS_VALIDATORS_ROOT`. Fetching from the API rather than hard-coding specific values ensures we do not have to modify the code each run. With these modifications to `staking-deposit-cli`, we were able to use it to create keystores and deposit data for the private devnet.

### The Process

After generating the keystores with the altered version of `staking-deposit-cli`, we employed `ethdo` to generate the raw deposit data since it is not supported by `staking-deposit-cli`. Generating validators using both tools was done concurrently, ensuring the signing and withdrawal keys match up at every step. The process is based on a guide by Potuz, a Prysm developer [131]:

1. Generate the signing keys using `staking-deposit-cli`.
2. Using `ethdo`, recreate the signing keys using the same mnemonic.
3. Compare public keys of the created signing keys.
4. After ensuring the signing keys match, generate the withdrawal keys using `staking-deposit-cli`.

5. Recreate the withdrawal keys using `ethdo`.
6. Compare withdrawal keys, ensuring they match.
7. Generate raw deposit data with `ethdo`.
8. Make and broadcast the deposit transaction.

Ideally, this process should be performed on an offline, air-gapped machine to ensure the keys are not compromised. However, ensuring the security of the validator creation is not a priority for our deployment.

Using this method, creating validators requires two times as many operations, which is not optimal for our use case, as we are generating a large number of validators. Instead, we opted to use `ethdo` alone, abandoning the code modifications to `staking-deposit-cli`. The process of using `ethdo` alone was outlined earlier in Section 5.6.2.

## 7.3 Comparing Solutions

Existing solutions, such as the Prysm devnet setup guide, Lighthouse's local testnet, and Mahmood's framework presented in Section 4.3, can provide valuable approaches for node deployment. However, these approaches also come with inherent limitations. For instance, Prysm's method involves a follow-along, hardcoded configuration process that lacks flexibility for dynamic setups. Similarly, while Lighthouse offers more dynamic deployment scripts, it still requires significant manual effort to manage and scale. In contrast, our solution seeks to streamline these processes while offering a more feature-rich, scalable, and user-friendly approach.

One of the perhaps main features we provide is the ability to add validators after the chain has started. None of the other three solutions we've examined provide a way to streamline the process of creating validators, generating deposit data, making and broadcasting deposits, and importing the validators into the validator client of choice between Prysm and Lighthouse. Along with adding validators, we lay the groundwork for analyzing incentives by enabling withdrawals and exits. None of the other approaches support either of these features.

The projects utilizing Prysm as the consensus client use `prysmctl` for generating the state. Similarly, the Lighthouse setup also uses its own tool, `lcli`. Our approach uses the third-party tool, `eth2-testnet-genesis`. Along with the advantages previously presented in Section 7.2.1, using a third-party tool for state generation ensures that our environment can easily adapt to include additional clients in the future without substantial modifications.

Table 7.1 compares our application to the other approaches discussed.

Prysm is denoted as partial support for deploying multiple nodes on a single machine because their guide only briefly mentions the next steps to launch a second node. However, this is a lackluster and incomplete guide.

Table 7.1: The table summarizes the available features for each discussed application. ✓ indicates full support, and ~ indicates partial support.

| | **Our application** | **Prysm's Guide** | **Lighthouse's Testnet** | **Zoraiz Mahmood** |
|---|---|---|---|---|
| Deploy single node | ✓ | ✓ | ✓ | ✓ |
| Deploy multiple nodes (single machine) | | ~ | ✓ | ✓ |
| Deploy multiple nodes (multiple machine) | ✓ | | | |
| Dynamically adding validators | ✓ | | | |
| Withdrawals and Exits | ✓ | | | |
| Multiple clients supported | ✓ | | | |
| Support Byzantine behavior | ✓ | | | |
| Enable metric collection | ✓ | | | |

## 7.4 Challenges

During the development of the application, there were multiple challenges and issues to overcome. This section will outline some of the challenges and the solutions we implemented.

### 7.4.1 Cluster Restrictions

Setting up a bare-bone deployment of a single Ethereum node on localhost was simple; however, migrating to the cluster and launching multiple nodes presented several challenges. These cluster challenges impacted our workflow and required various adaptations to be made.

#### Alloted Timeslots

One of the primary challenges was the limited timeslots available for using the machines. This constraint necessitated reducing the length of some experiments. Instead of running experiments for extended periods, we had to shorten their duration to fit within the given timeslots. This adjustment ensured we could still conduct the necessary experiments within the given constraints.

### Software Compatibility

Another significant issue was the incompatibility of the cluster's software versions with the requirements of Prysm and Lighthouse. The cluster had an older version of Go, while Prysm required Go version `1.21`. We installed local Go versions on each machine in the cluster to avoid disrupting other users by updating Go globally. This solution allowed us to meet the software requirements of Prysm without affecting the overall cluster environment.

Similarly, Lighthouse, written in Rust, typically relies on Rustup for installation. However, Rustup was not available on the cluster. Fortunately, Lighthouse provides pre-built binaries, which we were able to utilize. These binaries enabled us to run Lighthouse but did not include the `lcli` tool, preventing us from testing it.

### Missing Utilities

The absence of certain utilities on the cluster also posed challenges. Initially, we intended to use *web3py* for creating transactions, as it facilitates easier deployment of smart contracts. However, the cluster lacks pip, making it impossible to install web3py. As a result, we opted to use simple transactions alternative methods for transaction creation that did not rely on web3py.

## 7.4.2 Interoperability Issues

One of the biggest challenges was ensuring Prysm and Lighthouse established a connection. We encountered a recurring issue where clients failed to sync, eventually leading to Lighthouse disconnecting from Prysm and labeling it faulty. During these attempts to connect, Lighthouse generated warnings indicating invalid responses and issues with blocks from past failed chains:

```
Apr 09 10:36:44.262 WARN Peer sent invalid response to parent request., reason:
ExtraBlocksReturned, peer_id: 16Uiu2HAmFdHNjdaJD9Ef7GiqhMGztd..., service: sync

Apr 09 10:59:17.254 DEBG Block is from a past failed chain. Dropping, block_slot:
4, block_root: 0x63ebf5ecabd2c03a1f9880ba418279ec26..., service: sync
```

The root cause of these connectivity issues was traced back to the beacon chain parameters settings specified in the `config.yml` file. At the time, we were using a customized `config.yml` with certain values, such as `SLOTS_PER_EPOCH`, set lower than the mainnet specifications. Although `SLOTS_PER_EPOCH` is a preset value and should be fixed during compile-time, Prysm still allowed us to change this value without any issues when running exclusively Prysm nodes.

The reason Prysm accepted different preset values compared to Lighthouse lies in the programming languages they are written in. Prysm, developed in Go, benefits from a runtime environment that allows for dynamic parameter adjustment during execution. In contrast, Lighthouse is written in Rust, which lacks a runtime environment. This characteristic means Lighthouse cannot deviate from the predefined preset specification values [132], leading to synchronization challenges when interacting with a client like Prysm that can operate under different parameter settings.

Using the mainnet config preset solved our connectivity and syncing issues. However, using the mainnet configuration presents certain drawbacks, particularly regarding flexibility in experimental environments. For instance, in the mainnet configuration, the protocol only allows adding up to 8 validators per epoch. This limitation becomes a significant constraint during experiments with increasing validator counts, such as the experiment discussed in Section 6.3.4.

### 7.4.3 Bucket List

In Geth's implementation of Kademlia, each *k-bucket* has a `bucketSize` of 16, allowing it to contain a maximum of 16 peers. Additionally, the `bucketIPLimit` in Geth is set to 2, meaning that no more than two IP addresses from the same `/24` subnet can be included in any one bucket [133]. This setting is part of the node discovery protocol to maintain a healthy network by preventing overrepresenting nodes from the same network. This helps ensure a more diverse and resilient peer network.

Given that our setup involves deploying up to 30 nodes on the same subnet mask of `/24`, we exceed this limit. This means only two nodes from our subnet can be included in any single bucket. Despite this, the system's functionality remains unaffected as the nodes connect and communicate successfully. Only during debug mode is this limitation observed by outputting messages indicating an IP is exceeding the bucket limit:

```
DEBUG[05-22|14:51:48] IP exceeds bucket limit                    ip=152.94.162.12
DEBUG[05-22|14:52:01] IP exceeds bucket limit                    ip=152.94.162.13
```

## 7.5 System Monitoring

As discussed in Section 5.8.2, our decision to increment port numbers stems from the need to streamline accessing the dashboards for multiple nodes. This method significantly simplifies port forwarding management, which is essential for remotely monitoring each node's status and performance via SSH connections.

Traditionally, accessing a node's dashboard involves initiating an SSH connection that maps a local port to the corresponding port on the server hosting the node. The command typically used is:

```
ssh -L 3000:localhost:3000 username@bbchain1.ux.uis.no
```

This setup works well for single-node monitoring. However, when managing multiple nodes, this approach requires the user to terminate and re-establish the SSH tunnel for each node change, specifying a new hostname each time. For example, switching from *bbchain1* to *bbchain2* would necessitate canceling the current port forwarding and setting up a new one to a different host.

**Example 7.1.** For instance, in an experiment involving five nodes, each node can be accessed through a distinct port on the local machine, with port numbers ranging from $3001$ to $3005$, corresponding directly to the node's identifier. Each port directly reflects the node ID, making it intuitive for users to switch between dashboards of different nodes simply by altering the port number in the URL.

This approach of incrementing port numbers alleviates the otherwise cumbersome process. It allows simultaneous and seamless access to multiple nodes' dashboards without requiring manual termination and activation of SSH tunnels.

## 7.6 Future Works

For future improvements to the application, we suggest some of the following features:

- Deploy multiple nodes on a single machine.
- Implement a graphical user interface.
- Simplify and automate setup using Docker
- Perform Byzantine experiments.
- Include support for other consensus and execution clients.
- Measure hardware resources related to syncing.

Having support for deploying multiple nodes on a single machine is beneficial for researchers and developers who do not utilize a cluster but would like to use our features that other solutions don't provide. Implementing this is straightforward, requiring only the launch of multiple clients on a single machine with incremented ports to avoid interference. This was done during initial testing on localhost. However, we drifted away from it when we transitioned to the cluster, where we initially opted for default ports to simplify the setup.

A graphical user interface that utilizes our scripts could also benefit researchers who may not be technically proficient, facilitating easier setup, simulations, and experimentation on a private development network. A web-based interface could allow users to select the number of nodes, the validator count for each, and the specific client to launch through a graphical interface. However, this approach might reduce flexibility, limiting the ability to customize deployments fully.

To further simplify the deployment and use of our application, we propose utilizing Docker. Docker can simplify the process by allowing users to deploy nodes easily with minimal effort. With Docker, users can pull pre-configured container images and start multiple instances on a single machine, reducing the complexity and time required for setup. We did not utilize Docker in our experiments because it was unavailable on our cluster. However, integrating Docker in future work could significantly enhance the usability of our program, making it more accessible for researchers.

We also suggest conducting more simulations and specific experiments beyond studying hardware usage. Experiments focusing on the Byzantine fault tolerance implementation we provide could open up interesting opportunities for further research.

Our groundwork for genesis state generation and distribution makes the addition of support for other clients straightforward. Therefore, the potential for additional support for clients could be explored. Other consensus clients, such as Teku, Nimbus, and Grandine, could be integrated. Nimbus especially focuses on being lightweight, which would be an interesting comparison to Prysm and Lighthouse. Currently, we only utilize Geth as the execution client. Expanding to include support for another execution client could provide valuable research insights. For instance, Nethermind is an emerging client with significant migration from Geth.

Additionally, measuring the hardware and network usage during the protocols' syncing process would be an interesting area for experimentation. This could involve running multiple nodes for an extended period (e.g., 2-4 weeks), then introducing a new node and observing the time required to achieve full synchronization from the genesis state. Monitoring the hardware and network usage throughout this period would provide valuable insights into the performance and efficiency of different clients during the syncing process.

# Chapter 8

# Conclusion

In the rapidly evolving world of Ethereum, deploying a development network is essential for providing researchers with a proper testing environment. This setup allows them to perform simulations and test potential attacks, contributing significantly to the security and robustness of Ethereum. Our application facilitates an easy setup of a cluster consisting of multiple nodes, enabling a wide range of features, including validator management and flexible client configuration. Notably, none of the existing solutions offer a comparable set of features.

Using our applications, we conducted various experiments to conclude the necessary hardware requirements for a development network and compared these to the mainnet recommendations. We analyzed how hardware usage changes when nodes run with a small number of validators compared to a larger number. Additionally, we investigated how different levels of network load impact node performance and how the number of connected peers influences performance.

Our Application also lays a solid foundation for further development. Utilizing third-party genesis state generation tools that do not rely on a specific client enables easy expansion of consensus client support.

# Bibliography

[1] Ulysse Pavloff, Yackolley Amoussou-Genou, and Sara Tucci-Piergiovanni. *Byzantine Attacks Exploiting Penalties in Ethereum PoS*. 2024. arXiv: `2404.16363 [cs.CR]`.

[2] Michael Neuder et al. *Low-cost attacks on Ethereum 2.0 by sub-1/3 stakeholders*. 2021. arXiv: `2102.02247 [cs.CR]`.

[3] Do Hai Son, Tran Thi Thuy Quynh, and Le Quang Minh. *RANDAO-based RNG: Last Revealer Attacks in Ethereum 2.0 Randomness and a Potential Solution*. 2024. arXiv: `2403.09541 [cs.CR]`.

[4] Ralph C. Merkle. “Protocols for Public Key Cryptosystems”. In: *1980 IEEE Symposium on Security and Privacy*. 1980, pp. 122–122. DOI: `10.1109/SP.1980.10006`.

[5] Luís Rodrigues Christian Cachin Rachid Guerraoui. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer Berlin, Heidelberg, 2011.

[6] Karl Wüst and Arthur Gervais. “Do you Need a Blockchain?” In: *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. 2018, pp. 45–54. DOI: `10.1109/CVCBT.2018.00011`.

[7] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: `https://bitcoin.org/bitcoin.pdf`.

[8] Vitalik Buterin. *Ethereum: A next-generation smart contract and decentralized application platform*. 2014. URL: `https://github.com/ethereum/wiki/wiki/White-Paper`.

[9] Jie Xu, Cong Wang, and Xiaohua Jia. “A Survey of Blockchain Consensus Protocols”. In: *ACM Comput. Surv.* 55.13s (2023). ISSN: 0360-0300. DOI: `10.1145/3579845`.

[10] Yonatan Sompolinsky and Aviv Zohar. “Secure high-rate transaction processing in Bitcoin”. In: *Financial Cryptography and Data Security*. Springer, 2015, pp. 507–527. URL: `http://www.cs.huji.ac.il/~avivz/pubs/15/btc_ghost_full.pdf`.

[11] Cardano Team. *Cardano*. 2017. URL: `https://cardano.org` (Accessed: 9 June 2024).

[12] Solana Team. *Solana*. 2020. URL: `https://solana.com` (Accessed: 9 June 2024).

[13] Algorand Team. *Algorand*. 2019. URL: `https://algorandtechnologies.com` (Accessed: 9 June 2024).

[14] Cardano Team. *Polkadot*. 2020. URL: `https://polkadot.network/` (Accessed: 9 June 2024).

[15] Vitalik Buterin. *On Stake | Ethereum Foundation Blog*. 2014. URL: `https://blog.ethereum.org/2014/07/05/stake` (Accessed: 9 June 2024).

[16] Neo C. K. Yiu. *An Overview of Forks and Coordination in Blockchain Development*. 2021. arXiv: `2102.10006 [cs.CR]`.

[17] Vitalik Buterin. *Hard Forks, Soft Forks, Defaults and Coercion*. 2017. URL: `https://vitalik.eth.limo/general/2017/03/14/forks_and_markets.html` (Accessed: 9 June 2024).

[18] Ethereum Foundation. *Ethereum Improvement Proposals*. Ethereum Improvement Proposals (EIPs) describe standards for the Ethereum platform, including core protocol specifications, client APIs, and contract standards. Network upgrades are discussed separately in the Ethereum Project Management repository. URL: `https://eips.ethereum.org/`.

[19] Ethereum Foundation. *Ethereum Virtual Machine (EVM) | ethereum.org*. 2024. URL: `https://ethereum.org/en/developers/docs/evm/` (Accessed: 9 June 2024).

[20] Ethereum Foundation. *Introduction to Smart Contracts | ethereum.org*. 2024. URL: `https://ethereum.org/en/developers/docs/smart-contracts/` (Accessed: 9 June 2024).

[21] Solidity Team. *{solidity}*. A statically-typed curly-braces programming language designed for developing smart contracts that run on Ethereum. 2024. URL: `https://soliditylang.org/` (Accessed: 9 June 2024).

[22] Ethereum Foundation. *Merkle Patricia Trie | ethereum.org*. 2024. URL: `https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/` (Accessed: 9 June 2024).

[23] Ethereum Foundation. *Gas and Fees | ethereum.org*. 2024. URL: `https://ethereum.org/en/developers/docs/gas/` (Accessed: 9 June 2024).

[24] W. Stallings. *Cryptography and Network Security: Principles and Practice, Global Edition*. Pearson Education, 2022.

[25] Wikipedia contributors. *Elliptic Curve Digital Signature Algorithm — Wikipedia, The Free Encyclopedia*. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Elliptic_Curve_Digital_Signature_Algorithm&oldid=1186400923` (Accessed: 9 June 2024).

[26] Dan Boneh et al. *BLS Signatures*. Internet-Draft draft-irtf-cfrg-bls-signature-05. Work in Progress. Internet Engineering Task Force, 2022. 31 pp. URL: `https://datatracker.ietf.org/doc/draft-irtf-cfrg-bls-signature/05/`.

[27] Vitalik Buterin. *Exploring Elliptic Curve Pairings*. 2017. URL: `https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627` (Accessed: 9 June 2024).

[28] Dan Boneh, Manu Drijvers, and Gregory Neven. "Compact Multi-signatures for Smaller Blockchains". In: Brisbane, QLD, Australia: Springer-Verlag, 2018, pp. 435–464. ISBN: 978-3-030-03328-6. DOI: `10.1007/978-3-030-03329-3_15`. URL: `https://doi.org/10.1007/978-3-030-03329-3_15`.

[29] Justin Drake. *Pragmatic Signature Aggregation with BLS*. 2018. URL: `https://ethresear.ch/t/pragmatic-signature-aggregation-with-bls/2105?u=benjaminion` (Accessed: 9 June 2024).

[30] Etherem Foundation. *Ethereum Proof-of-Stake Consensus Specifications*. Deneb Edition. 2024. URL: `https://github.com/ethereum/consensus-specs/tree/v1.4.0` (Accessed: 9 June 2024).

[31] Ethereum Foundation. *Phase 0 – The Beacon Chain*. Deneb Edition. 2020. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/phase0/beacon-chain.md` (Accessed: 9 June 2024).

[32] Ethereum Foundation. *Altair – The Beacon Chain*. Deneb Edition. 2021. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/altair/beacon-chain.md` (Accessed: 9 June 2024).

[33] Ethereum Foundation. *Bellatrix – The Beacon Chain*. Deneb Edition. 2022. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/bellatrix/beacon-chain.md` (Accessed: 9 June 2024).

[34] Ethereum Foundation. *Capella – The Beacon Chain*. Deneb Edition. 2023. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/capella/beacon-chain.md` (Accessed: 9 June 2024).

[35] Ethereum Foundation. *Deneb – The Beacon Chain*. Deneb Edition. 2024. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/deneb/beacon-chain.md` (Accessed: 9 June 2024).

[36] Vitalik Buterin. *Vitalik's Annotated Ethereum 2.0 Spec*. This is an annotated version of the Phase 0 beacon chain spec. 2020. URL: `https://github.com/ethereum/annotated-spec/blob/master/phase0/beacon-chain.md` (Accessed: 9 June 2024).

[37] Ben Edgington. *Upgrading Ethereum*. Capella Edition. 2023. URL: `https://eth2book.info/capella`.

[38] Ethereum Foundation. *The Beacon Chain | ethereum.org*. 2024. URL: `https://ethereum.org/en/roadmap/beacon-chain/` (Accessed: 9 June 2024).

[39] Ethereum Foundation. *The Merge | ethereum.org*. 2024. URL: `https://ethereum.org/en/roadmap/merge` (Accessed: 9 June 2024).

[40] Ethereum Foundation. *Blocks | ethereum.org*. 2024. URL: `https://ethereum.org/en/developers/docs/blocks` (Accessed: 9 June 2024).

[41] Ethereum Foundation. *Node Architecture | ethereum.org*. 2024. URL: `https://ethereum.org/developers/docs/nodes-and-clients/node-architecture` (Accessed: 9 June 2024).

[42] Ethereum Foundation. *Engine JSON-RPC API*. 2024. URL: `https://github.com/ethereum/execution-apis/tree/main/src/engine` (Accessed: 9 June 2024).

[43] Ethereum Foundation. *Eth Beacon Node API*. Version 2.5.0. 2024. URL: `https://ethereum.github.io/beacon-APIs` (Accessed: 9 June 2024).

[44] Ethereum Foundation. *Ethereum JSON-RPC Specification*. 2024. URL: `https://ethereum.github.io/execution-apis/api-documentation` (Accessed: 9 June 2024).

[45] Ethereum Foundation. *Simple Serialize | ethereum.org*. 2024. URL: `https://ethereum.org/en/developers/docs/data-structures-and-encoding/ssz/` (Accessed: 9 June 2024).

[46] Ben Edgington. “SSZ: Simple Serialize”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/building_blocks/ssz/`.

[47] Etherem Foundation. *Simple Serialize (SSZ)*. Deneb Edition. 2020. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/ssz/simple-serialize.md` (Accessed: 9 June 2024).

[48] Ben Edgington. “Hash Tree Roots and Merkleization”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/building_blocks/merkleization/`.

[49] Vitalik Buterin et al. *Combining GHOST and Casper*. 2020. arXiv: `2003.03052 [cs.CR]`.

[50] Ethereum Foundation. *Phase 0 – Honest Validator*. Deneb Edition. 2020. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/phase0/validator.md` (Accessed: 9 June 2024).

[51] Bitfly. *Open Source Ethereum Explorer*. 2024. URL: `https://beaconcha.in/` (Accessed: 18 Apr. 2024).

[52] Ethereum Foundation. *Phase 0 – Beacon Chain Fork Choice*. Deneb Edition. 2020. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/phase0/fork-choice.md` (Accessed: 9 June 2024).

[53] Miguel Castro and Barbara Liskov. “Practical Byzantine fault tolerance”. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI ’99. USENIX Association, 1999, pp. 173–186. ISBN: 1880446391.

[54] Ben Edgington. “Casper FFG”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/consensus/casper_ffg/`.

[55] Ethereum Foundation. *Capella – Beacon Chain Fork Choice*. Deneb Edition. 2023. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/capella/fork-choice.md` (Accessed: 9 June 2024).

[56] Ben Edgington. “Fork Choice”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 3. URL: `https://eth2book.info/capella/part3/forkchoice/`.

[57] Ben Edgington. “Inactivity Leak”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/incentives/inactivity/`.

[58] Ben Edgington. “Balances”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/incentives/balances/`.

[59] Hsiao-Wei Wang. *A note on Ethereum 2.0 phase 0 validator lifecycle*. 2020. URL: `https://notes.ethereum.org/@hww/lifecycle` (Accessed: 9 June 2024).

[60] Ben Edgington. “Deposits and Withdrawals”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/deposits-withdrawals/`.

[61] Ben Edgington. “The Deposit Contract”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/deposits-withdrawals/contract/`.

[62] Ethereum Foundation. *Phase 0 – Deposit Contract*. Deneb Edition. 2020. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/phase0/deposit-contract.md` (Accessed: 9 June 2024).

[63] Peter Davies (@petertdavies) Mikhail Kalinin (@mkalinin) Danny Ryan (@djrtwo). *EIP-6110: Supply validator deposits on chain [DRAFT]*. [Online serial]. Ethereum Improvement Proposals, no. 6110. Dec. 2022. URL: `https://eips.ethereum.org/EIPS/eip-6110`.

[64] Ben Edgington. “Withdrawals”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/deposits-withdrawals/withdrawal-processing/`.

[65] Ben Edgington. “BLS Signatures”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/building_blocks/signatures/`.

[66] Ben Edgington. “Randomness”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/building_blocks/randomness/`.

[67] Toni Wahrstätter. *Selfish Mixing and RANDAO Manipulation*. 2023. URL: `https://ethresear.ch/t/selfish-mixing-and-randao-manipulation/16081` (Accessed: 9 June 2024).

[68] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. *An Enciphering Scheme Based on a Card Shuffle*. 2014. arXiv: `1208.1176 [cs.CR]`.

[69] Ben Edgington. “Shuffling”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/building_blocks/shuffling/`.

[70] YCHARTS. *Ethereum Supply*. 2024. URL: `https://ycharts.com/indicators/ethereum_supply` (Accessed: 9 June 2024).

[71] Chih-Cheng Liang. *Minimum Committee Size Explained*. 2019. URL: `https://medium.com/@chihchengliang/minimum-committee-size-explained-67047111fa20` (Accessed: 9 June 2024).

[72] Ben Edgington. “Committees”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/building_blocks/committees/`.

[73] Vitalik Buterin. *Ethereum 2.0 Phase 1 – The Beacon Chain with Shards*. This is an annotated version of the Phase 1 beacon chain spec. 2020. URL: `https://github.com/ethereum/annotated-spec/blob/master/phase1/beacon-chain.md` (Accessed: 9 June 2024).

[74] Tim Beiko (@timbeiko). *EIP-7600: Hardfork Meta - Pectra [DRAFT]*. [Online serial]. Ethereum Improvement Proposals, no. 7600. Jan. 2024. URL: `https://eips.ethereum.org/EIPS/eip-7600`.

[75] dapplion (@dapplion). *EIP-7549: Move committee index outside Attestation [DRAFT]*. [Online serial]. Ethereum Improvement Proposals, no. 7549. Nov. 2023. URL: `https://eips.ethereum.org/EIPS/eip-7549`.

[76] Ben Edgington. “Aggregator Selection”. In: *Upgrading Ethereum*. Capella Edition. 2023. Chap. 2. URL: `https://eth2book.info/capella/part2/building_blocks/aggregator/`.

[77] Ethereum Foundation. *Network Addresses | ethereum.org*. 2024. URL: `https://ethereum.org/en/developers/docs/networking-layer/network-addresses/` (Accessed: 9 June 2024).

[78] Ethereum Foundation. *Node Discovery Protocol*. 2018. URL: `https://github.com/ethereum/devp2p/blob/master/discv4.md` (Accessed: 9 June 2024).

[79] Ethereum Foundation. *Node Discovery Protocol v5*. 2020. URL: `https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md` (Accessed: 9 June 2024).

[80] Felix Lange <fjl@ethereum.org>. *EIP-778: Ethereum Node Records (ENR)*. [Online serial]. Ethereum Improvement Proposals, no. 778. Nov. 2017. URL: `https://eips.ethereum.org/EIPS/eip-778`.

[81] Ethereum Foundation. *Phase 0 – Networking*. Deneb Edition. 2020. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/phase0/p2p-interface.md` (Accessed: 9 June 2024).

[82] Petar Maymounkov and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric”. In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS ’01. 2002, pp. 53–65. ISBN: 3540441794.

[83] Ethereum Foundation. *Node Discovery Protocol v5 - Theory*. 2020. URL: `https://github.com/ethereum/devp2p/blob/master/discv5/discv5-theory.md` (Accessed: 9 June 2024).

[84] @vyzo. *gossipsub v1.0: An extensible baseline pubsub protocol*. 2020. URL: `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0.md` (Accessed: 9 June 2024).

[85] Ethereum Foundation. *Capella – Networking*. Deneb Edition. 2023. URL: `https://github.com/ethereum/consensus-specs/blob/v1.4.0/specs/capella/p2p-interface.md` (Accessed: 9 June 2024).

[86] Libp2p. *What is Publish/Subscribe*. 2024. URL: `https://docs.libp2p.io/concepts/pubsub/overview/` (Accessed: 9 June 2024).

[87] Ethereum Foundation. *Client Diversity | ethereum.org*. 2024. URL: `https://ethereum.org/developers/docs/nodes-and-clients/client-diversity` (Accessed: 9 June 2024).

[88] Ethereum Foundation. *Nodes and clients | ethereum.org*. 2024. URL: `https://ethereum.org/nb/developers/docs/nodes-and-clients/#execution-clients` (Accessed: 12 June 2024).

[89] go-ethereum. *go-ethereum*. 2024. URL: `https://geth.ethereum.org/` (Accessed: 12 June 2024).

[90] Hyperledger Besu. *Hyperledger Besu Ethereum client*. 2024. URL: `https://besu.hyperledger.org/` (Accessed: 12 June 2024).

[91] Erigon. *Erigon*. 2024. URL: `https://erigon.tech/` (Accessed: 12 June 2024).

[92] Nethermind Client. *A robust client for Ethereum node operators*. 2024. URL: `https://docs.nethermind.io/` (Accessed: 12 June 2024).

[93] Ethereum Foundation. *Nodes and clients | ethereum.org*. 2024. URL: `https://ethereum.org/nb/developers/docs/nodes-and-clients/#consensus-clients` (Accessed: 12 June 2024).

[94] PrysmaticLabs. *Table of Contents | Prysm*. 2024. URL: `https://docs.prylabs.network/docs/getting-started` (Accessed: 12 June 2024).

[95] Consensys. *Teku | Ethereum 2.0 Client for Institutional Staking*. 2024. URL: `https://consensys.io/teku` (Accessed: 12 June 2024).

[96] Nimbus. *Nimbus*. 2024. URL: `https://nimbus.team/index.html` (Accessed: 12 June 2024).

[97] Lodestar. *Ethereum meets JavaScript*. 2024. URL: `https://lodestar.chainsafe.io/` (Accessed: 12 June 2024).

[98] Lighthouse. *Lighthouse Book*. 2024. URL: `https://lighthouse-book.sigmaprime.io/` (Accessed: 12 June 2024).

[99] Grandinetech. *Grandine: A fast and lightweight Ethereum consensus client*. 2024. URL: `https://github.com/grandinetech/grandine` (Accessed: 9 June 2024).

[100] Grandine. *Grandine Ethereum consensus client is open-sourced!* 2024. URL: `https://medium.com/@grandine/grandine-is-open-sourced-b1815cf0ae39` (Accessed: 9 June 2024).

[101] Ether Alpha. *Client Diversity*. 2024. URL: `https://clientdiversity.org/#distribution` (Accessed: 15 Feb. 2024).

[102] Ether Alpha. *Data Methodology*. 2024. URL: `https://clientdiversity.org/methodology/` (Accessed: 12 June 2024).

[103] Raul Jordan. *How to Set Up an Ethereum Proof-of-Stake Devnet in Minutes*. 2023. URL: `https://github.com/prysmaticlabs/documentation/blob/master/website/docs/advanced/proof-of-stake-devnet.md` (Accessed: 12 Apr. 2024).

[104] Geth Team. *Krogam DMZ (v1.12.0)*. The v1.12 release family drops support for proof-of-work. 2023. URL: `https://github.com/ethereum/go-ethereum/releases/tag/v1.12.0` (Accessed: 9 June 2024).

[105] Prysmaticlabs. *How to Set Up an Ethereum Proof-of-Stake Devnet in Minutes*. 2023. URL: `https://docs.prylabs.network/docs/advanced/proof-of-stake-devnet` (Accessed: 8 May 2024).

[106] Lighthouse. *Simple Local Testnet*. 2023. URL: `https://github.com/sigp/lighthouse/tree/stable/scripts/local_testnet` (Accessed: 8 May 2024).

[107] Zoraiz Mahmood. *Deploy your own Local Ethereum PoS Testnet*. 2023. URL: `https://github.com/rzmahmood/ethereum-pos-testnet` (Accessed: 12 June 2024).

[108] protolambda. *Eth2 Testnet Genesis State Creator*. Deneb Support. 2024. URL: `https://github.com/protolambda/eth2-testnet-genesis` (Accessed: 9 June 2024).

[109] Ethereum Foundation. *Ethereum Accounts | ethereum.org*. 2024. URL: `https://ethereum.org/en/developers/docs/accounts/` (Accessed: 9 June 2024).

[110] Ethereum Foundation. *Presets*. Deneb Version. 2024. URL: `https://github.com/ethereum/consensus-specs/tree/v1.4.0/presets` (Accessed: 9 June 2024).

[111] Ethereum Foundation. *Configurations*. Deneb Version. 2024. URL: `https://github.com/ethereum/consensus-specs/tree/v1.4.0/configs` (Accessed: 9 June 2024).

[112] Prysm. *Keys, wallets, and accounts*. 2024. URL: `https://docs.prylabs.network/docs/wallet/introduction` (Accessed: 12 June 2024).

[113] Carl Beekhuizen (@CarlBeek) <carl@ethereum.org>. *ERC-2333: BLS12-381 Key Generation [DRAFT]*. [Online serial]. Ethereum Improvement Proposals, no. 2333. Sept. 2019. URL: `https://eips.ethereum.org/EIPS/eip-2333`.

[114] Carl Beekhuizen (@CarlBeek) <carl@ethereum.org>. *ERC-2334: BLS12-381 Deterministic Account Hierarchy [DRAFT]*. [Online serial]. Ethereum Improvement Proposals, no. 2334. Sept. 2019. URL: `https://eips.ethereum.org/EIPS/eip-2334`.

[115] wealdtech. *Ethdo*. Latest update as of 2024. 2020. URL: `https://github.com/wealdtech/ethdo` (Accessed: 9 June 2024).

[116] wealdtech. *Changing withdrawal credentials | Ethdo*. 2023. URL: `https://github.com/wealdtech/ethdo/blob/master/docs/changingwithdrawalcredentials.md` (Accessed: 9 June 2024).

[117] wealdtech. *Exiting validators | Ethdo*. 2023. URL: `https://github.com/wealdtech/ethdo/blob/master/docs/exitingvalidators.md` (Accessed: 9 June 2024).

[118] Prometheus. *Prometheus*. 2024. URL: `https://prometheus.io/` (Accessed: 12 June 2024).

[119] GrafanaLabs. *Grafana*. 2024. URL: `https://grafana.com/` (Accessed: 12 June 2024).

[120] influxdata. *InfluxDB. It's About Time*. 2024. URL: `https://www.influxdata.com/` (Accessed: 14 June 2024).

[121] Prysm Documentation. *Dashboard designed for more than 10 validator keys*. 2024. URL: `https://docs.prylabs.network/assets/grafana-dashboards/big_amount_validators.json` (Accessed: 3 May 2024).

[122] Prysmaticlabs. *v5.0.3 | Prysm Version*. 2024. URL: `https://github.com/prysmaticlabs/prysm/releases/tag/v5.0.3` (Accessed: 12 June 2024).

[123] SigmaPrime. *Advanced Networking | Lighthouse Book*. 2024. URL: `https://lighthouse-book.sigmaprime.io/advanced_networking.html#target-peers` (Accessed: 12 June 2024).

[124] yorickdowne. *Overview | Hall of Blame*. 2024. URL: `https://github.com/eth-educators/ethstaker-guides/blob/main/migrating-to-a-larger-disk.md` (Accessed: 12 June 2024).

[125] go-ethereum. *Hardware requirements*. 2022. URL: `https://geth.ethereum.org/docs/getting-started/hardware-requirements` (Accessed: 12 June 2024).

[126] Prysm. *Install Prysm with Docker*. 2024. URL: `https://docs.prylabs.network/docs/install/install-with-docker` (Accessed: 12 June 2024).

[127] SigmaPrime: Lighthouse Book. *Installation - Lighthouse Book*. 2024. URL: `https://lighthouse-book.sigmaprime.io/installation.html` (Accessed: 12 June 2024).

[128] Mikel Cortes-Goicoechea, Luca Franceschini, and Leonardo Bautista-Gomez. *Resource Analysis of Ethereum 2.0 Clients*. 2020. arXiv: `2012.14718 [cs.CR]`.

[129] Ethereum Foundation. *staking-deposit-cli*. 2020. URL: `https://github.com/ethereum/staking-deposit-cli` (Accessed: 15 June 2024).

[130] protolambda. *Validator management tools*. 2020. URL: `https://github.com/PRotolambda/eth2-val-tools` (Accessed: 15 June 2024).

[131] Potuz. *My checklist to create and verify accounts and then deposit*. 2024. URL: `https://www.reddit.com/r/ethstaker/comments/jrkbe2/my_checklist_to_create_and_verify_accounts_and/` (Accessed: 12 June 2024).

[132] SigmaPrime. *Lighthouse Eth Specs | GitHub*. 2024. URL: `https://github.com/sigp/lighthouse/blob/3058b96f2560f1da04ada4f9d8ba8e5651794ff6/consensus/types/src/eth_spec.rs#L291-L338` (Accessed: 12 June 2024).

[133] go-ethereum. *p2p/discover/table.go*. 2023. URL: `https://github.com/ethereum/go-ethereum/blob/9fd76e33af367752160ab0e33d1097e1e9aff6e4/p2p/discover/table.go#L53` (Accessed: 12 June 2024).

# Appendix A

# Overview of All Scripts

Table A.1 is an overview of all the scripts available in the project. Some of these are intended to be executed frequently, such as `start_fullnode`, while others, such as `start_prysm_consensus_client`, are not, as other scripts invoke them. For localhost deployment, all scripts should take a `--node` flag followed by an integer value, e.g., ”`--node 1`”. The scripts will automatically remind the user to supply the flag if it is missing.

Table A.1: An overview of all scripts in the application. Scripts marked with '*' are intended to be executed by the user; the remaining are invoked by other scripts or used as a one-time setup.

| Scripts | Description | Flags | Flag(s) Description |
|---|---|---|---|
| build_deps.sh* | Builds or installs dependencies required | `--all` | Install all |
| start_fullnode* | Launch a fullnode | `--server`<br>`--config` <value><br>`--ip` <value><br>`--prysm`<br>`--byzantine`<br>`--lh` | Launch a bootnode<br>mainnet (default), minimal, interop<br>IP of the bootnode<br>Uses Prysm as client (default)<br>Run Prysm node as Byzantine<br>Uses Lighthouse as client |
| create_validator* | Creates validators | `--num`-validators<br>`--insecure` | Number to create<br>Copies instead of generating |
| make_deposits* | Send deposits to the deposit contract | `--all-deposits` | Send all deposits available |
| convert_withdrawal_address* | Enable partial withdrawals for validators | | |
| exit_validator* | Used for exiting validators | | |

| | | | |
|---|---|---|---|
| monitor* | Used for monitoring a node. Example : './monitor beacon'. | execution | Monitor the execution client |
| | | beacon | Monitor the beacon client |
| | | validator | Monitor the validator client |
| kill_clients* | Shut down the chain | `--log` | Save all metrics and logs created by the clients |
| setup | Create genesis block and state. | `--server` | Prepare a bootnode |
| | | `--config` <value> | Defines the config.yml file to use |
| | | `--ip` | IP of bootnode |
| | | `--skip` | Skip deletion of ./network dir |
| copy_validators | Copies validator accounts from pre-generated keystore | | |
| start_geth_execution_client | Launch a Geth execution client | | |
| start_prysm_consensus_client | Launch a Prysm consensus client | `--byzantine` | Run Byzantine version |
| start_lighthouse_consensus_client | Launch a Lighthouse consensus client | | |
| import_prysm_validators | Import validators into a Prysm validator client | | |
| import_lighthouse_validators | Imports validators into a Lighthouse validator client | | |
| start_prysm_validator_client | Launch a Prysm validator client | | |
| start_lighthouse_validator_client | Launch a Lighthouse validator client | | |
| store_logs | Store logs of current run (does not stop the chain) | | |
| start_metrics | Start the Prometheus and Grafana servers | | |
| init_wallets | Initiates 30 wallets with 4096 accounts each | | |
| node_config | Defines variables for the node. All scripts source this file | | |

| | | |
|---|---|---|
| checksum | Compute the checksum address of provided value | <value> |
| copy_validators.py | Used by copy_validators to write the new index file | |
| find_enr.py | Get the enr of the beacon client. | |

# Appendix B

# Instructions to Compile and Run the System

Here, we will go through a detailed example of how to deploy the devnet on the BBChain cluster. First, we present the requirements for running the system.

## B.1 System Requirements

First, Prysm requires Go version 1.21.6 or newer. Scripts require `jq` and Python `3.11.x`. Prysm, Lighthouse, and other necessary binaries are automatically downloaded and installed by using the `build_deps.sh` script.

## B.2 Example Configuration Used

For this example, we will use a configuration consisting of 3 nodes on the BBChain cluster (*bbchain1*, *bbchain2*, and *bbchain3*). `node-1` will act as the main node running a Prysm consensus client; it will be responsible for generating the genesis and making it available for the other nodes. `node-1` will run 128 validators, `node-2` runs 64 validators using Prysm, and `node-3` will run 32 validators, and run Lighthouse.

## B.3 Step by Step Guide

Follow-along steps to set up the private devnet. Ensure the project is cloned on all machines you wish to deploy nodes.

**Node-1:**

1. SSH into *bbchain1*, and change working directory to `/home/<user>/deploy-ethereum-pos/devnet`.

2. Open `./config/mnemonics.yml`, and specify the number of validators for the first 3 mnemonics while the remaining 27 should have a count of 0. The beginning of the file should look like this:

```
- mnemonic: "nature expand bone never make where chalk autumn chicken
   present elegant face trouble giggle wrong stick brave strike child
   rocket sand try ask dinosaur"
  count: 128
- mnemonic: "mouse anchor daughter original holiday alpha expose brain
   garden access random shrug captain circle endless question plate vapor
   visa rival merge harvest frame donate"
  count: 64
- mnemonic: "liberty annual spread cry eye stereo used suit effort inmate
   hello kitten palm since owner comfort blood ginger dolphin soldier ridge
    cake direct clip"
  count: 32
```

3. Start the fullnode script: `./start_fullnode --server --config mainnet`

Now, the first node has been deployed. Since the max number of validators defined in `mnemonics.yml` is 128, we have a genesis delay of around 136 seconds. This means we need to launch the next nodes within 4 minutes to ensure they have a chance to propose the first couple of blocks.

**Node-2:**

4. SSH into *bbchain2*: `ssh bbchain2` and change working directory as previously: `/home/<user>/deploy-ethereum-pos/devnet`.

5. Execute the fullnode script, but specify the IP of the bootnode, here, `node-1`: `./start_fullnode --ip 152.94.64.11`.

Now, `node-2` should be performing the necessary steps of fetching the genesis information and importing its validators into the validator client.

**Node-3:**

6. SSH into *bbchain3*: `ssh bbchain3` and change working directory: `/home/<user>/deploy-ethereum-pos/devnet`.

7. Execute the fullnode script as earlier, but include `--lh` to launch a Lighthouse node: `./start_fullnode --ip 152.94.64.11 --lh`.

Now, the system should be up and running with 3 nodes, 128, 64, and 32 validators, respectively. Review logs by using the script: `./monitor execution` on each of the nodes.

## B.4 Creating Validators

If we want to increase the validator count on for example `node-3` after we have deployed the system, perform the following steps on `node-3`:

1. Run `./create_validator` (optionally, use the `--insecure` flag to speed up).
2. Answer prompts about how many you wish to create.
3. After the script finishes, make deposits with `./make_deposits`.
4. Answer prompts asking how many deposits you wish to make.
5. Wait until the chain picks up the deposits and they are processed. Refer to Section 3.3.5 for the deposit process.

Review `http://localhost:3501/eth/v1/beacon/states/head/validators` to see the status of all validators, including those we deposited.

## B.5 Enable withdrawals

If we wish to enable partial withdrawals on, say `node-1`, do the following on `node-1`:

1. Run the script: `./convert_withdrawal_address`

Review `http://localhost:3501/eth/v1/beacon/states/head/validators` to see the withdrawal credentials change for the validators belonging to `node-1`.

## B.6 Exit Validators

If we wish to exit one or more validators on, say `node-1`, do the following on `node-1`:

1. Run the script: `./exit_validator`
2. Answer "yes" to the prompt if you wish to exit all validators or "no" if you wish to further select individual ones.

Review `http://localhost:3501/eth/v1/beacon/states/head/validators`, to see the validator status change to "exited" and eventually "withdrawable" for the validators belonging to `node-1`.

## B.7 Stopping the System

The script `./kill_clients` can be used to stop the nodes running on that node. Supplying the flag `--log` stores the logs created by all clients, as well as the data gathered by Prometheus.

# Appendix C

# Additional Changes to Launch Over 30 Nodes

As mentioned in Chapter 5, some changes are necessary to use our application when deploying over 30 nodes.

## C.1 Required Changes

In `genesis.json`, there are 30 execution layer accounts (EOA) with 10 million ETH each. Creating more of these accounts is necessary to deploy more nodes. The command shown previously in Listing 5.2 should be used to create further accounts. After creating the accounts, include their public key in `genesis.json` file. Additionally, add more directories to the `keys` directory, such as `keys/node-31`, `keys/node-32`, and so on. The generated EOA for `node-i` should be put in `keys/node-i`.

The file `mnemonics.yml` has 30 mnemonic strings listed. To be able to generate validators for additional nodes above 30, the user has to add one mnemonic string (and count pair) for each node over 30. Tools such as `eth2-val-tools` and `staking-deposit-cli` can be used to generate a mnemonic. There are also online generators that produce valid mnemonics. Note that the mnemonic should have a length of 24 words. Entries can be added manually, or by using the following command:

```
mnemonic="new mnemonic here" && echo "- mnemonic: \"$mnemonic\"\n  count: 0" >>
    mnemonics.yml
```

There is a script `init_wallets`, which was used once to generate the initial 4096 keys (signing and withdrawal) for the 30 mnemonics included in the `mnemonics.yml` file. This script requires minimal modifications (change for loop from 30 to wanted number) and should be executed again to generate keys for the additional nodes. More than 4096 keys can also be generated by modifying the script. Currently, with 30 mnemonics, the script takes around 11 hours to finish execution.

The `keys/node-i` directories must exist for nodes beyond 30 if the `init_wallets` scripts is executed with more than 30 nodes.

# Appendix D

# Code From Consensus Specification

All the listings and their comments in this Appendix are sourced from the consensus specifications [30].

## D.1 Beacon State

```
class BeaconState(Container):
    # Versioning
    genesis_time: uint64
    genesis_validators_root: Root
    slot: Slot
    fork: Fork
    # History
    latest_block_header: BeaconBlockHeader
    block_roots: Vector[Root, SLOTS_PER_HISTORICAL_ROOT]
    state_roots: Vector[Root, SLOTS_PER_HISTORICAL_ROOT]
    historical_roots: List[Root, HISTORICAL_ROOTS_LIMIT]  # Frozen in Capella,
    replaced by historical_summaries
    # Eth1
    eth1_data: Eth1Data
    eth1_data_votes: List[Eth1Data, EPOCHS_PER_ETH1_VOTING_PERIOD *
    SLOTS_PER_EPOCH]
    eth1_deposit_index: uint64
    # Registry
    validators: List[Validator, VALIDATOR_REGISTRY_LIMIT]
    balances: List[Gwei, VALIDATOR_REGISTRY_LIMIT]
    # Randomness
    randao_mixes: Vector[Bytes32, EPOCHS_PER_HISTORICAL_VECTOR]
    # Slashings
    slashings: Vector[Gwei, EPOCHS_PER_SLASHINGS_VECTOR]  # Per-epoch sums of
    slashed effective balances
    # Participation [Modified in Altair]
    prev_epoch_participation: List[ParticipationFlags, VALIDATOR_REGISTRY_LIMIT]
    curr_epoch_participation: List[ParticipationFlags, VALIDATOR_REGISTRY_LIMIT]
```

```
    # Finality
    justification_bits: Bitvector[JUSTIFICATION_BITS_LENGTH]  # Bit set for every
     recent justified epoch
    previous_justified_checkpoint: Checkpoint
    current_justified_checkpoint: Checkpoint
    finalized_checkpoint: Checkpoint
    # Inactivity
    inactivity_scores: List[uint64, VALIDATOR_REGISTRY_LIMIT]  # [New in Altair]
    # Sync
    current_sync_committee: SyncCommittee  # [New in Altair]
    next_sync_committee: SyncCommittee  # [New in Altair]
    # Execution
    latest_execution_payload_header: ExecutionPayloadHeader  # [New in Bellatrix]
    # Withdrawals
    next_withdrawal_index: WithdrawalIndex  # [New in Capella]
    next_withdrawal_validator_index: ValidatorIndex  # [New in Capella]
    # Deep history valid from Capella onwards
    historical_summaries: List[HistoricalSummary, HISTORICAL_ROOTS_LIMIT]  # [New
     in Capella]
```

Listing D.1: BeaconState class as defined by the consensus specifications.

## D.2 Epoch Process

```
def process_epoch(state: BeaconState) -> None:
    process_justification_and_finalization(state)  # [Modified in Altair]
    process_inactivity_updates(state)  # [New in Altair]
    process_rewards_and_penalties(state)  # [Modified in Altair]
    process_registry_updates(state)
    process_slashings(state)  # [Modified in Altair]
    process_eth1_data_reset(state)
    process_effective_balance_updates(state)
    process_slashings_reset(state)
    process_randao_mixes_reset(state)
    process_historical_summaries_update(state)  # [Modified in Capella]
    process_participation_flag_updates(state)  # [New in Altair]
    process_sync_committee_updates(state)  # [New in Altair]
```

Listing D.2: process_epoch function as defined by the consensus specifications.

## D.3 Beacon Block

```
class BeaconBlock(Container):
    slot: Slot
    proposer_index: ValidatorIndex
    parent_root: Root
    state_root: Root
    body: BeaconBlockBody
```

Listing D.3: BeaconBlock class as defined by the consensus specifications.

```
class BeaconBlockBody(Container):
    randao_reveal: BLSSignature
    eth1_data: Eth1Data  # Eth1 data vote
    graffiti: Bytes32  # Arbitrary data
    # Operations
    proposer_slashings: List[ProposerSlashing, MAX_PROPOSER_SLASHINGS]
    attester_slashings: List[AttesterSlashing, MAX_ATTESTER_SLASHINGS]
    attestations: List[Attestation, MAX_ATTESTATIONS]
    deposits: List[Deposit, MAX_DEPOSITS]
    voluntary_exits: List[SignedVoulntaryExit, MAX_VOLUNTARY_EXITS]
    sync_aggregate: SyncAggregate
    # Execution
    execution_payload: ExecutionPayload  # [Modified in Deneb:EIP4844]
    bls_to_execution_changes: List[SignedBLSToExecutionChange,
    MAX_BLS_TO_EXECUTION_CHANGES]
    # [New in Deneb:EIP4844]
    blob_kzg_commitments: List[KZGCommitment, MAX_BLOB_COMMITMENTS_PER_BLOCK]
```

Listing D.4: BeaconBlockBody class as defined by the consensus specifications.

University
of Stavanger

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: Liam Cobb