HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Stateless Ethereum Syncing

Comparing Beam Sync

Valerie Marie-Luise Vaske

Bachelor Thesis in "Applied Computer Sciences"

October 19, 2020

Author Valerie Marie-Luise Vaske
Matriculation Number: 1434213
valerie.vaske@gmail.com

First Examiner: Jussi Salzwedel M.Sc.
Abteilung Informatik, Fakultät IV
Hochschule Hannover
jussi.salzwedel@hs-hannover.de

Second Examiner: Jason Carver
Ethereum Foundation
Trinity Team
carver@ethereum.org

Declaration Of Independence

I hereby declare that I have written the submitted bachelor thesis independently and without outside help, that I have not used sources and aids other than those indicated by me, and that I have marked the passages taken from the works used, either literally or in terms of content, as such.

Hanover, October 19, 2020 Signature

Contents

Abstract

Ethereum is a Turing-complete blockchain that allows the deployment and execution of Smart Contracts through the Ethereum Virtual Machine (EVM).

Each Ethereum node can run a diverse plethora of different clients in different programming languages so that the whole network benefits from end-user friendliness and enhanced security.

Albeit currently being on the roadmap to a significant update to its infrastructure termed 'Eth 2.0', research and development on 'Eth 1.x' topics have not halted: Many of the researched and developed solutions will either contribute directly to a smoother transition towards Eth 2.0 or be beneficial for Eth 2.0.

In September 2019, Carver published an article that approaches the possibility for a node's stateless synchronization. The inspiration for this research and its implementation stems from an idea Buterin published in October 2017. The algorithm, termed 'Beam Sync', finds implementation in the Ethereum clients Nethermind and Trinity.

To fully understand the impact of Beam Sync and the idea of stateless Ethereum clients, this thesis will explain how the approach works and evaluate the positive and negative aspects of this algorithm against other clients' existing solutions, like Geth's Fast Sync and Parity's Warp Sync.

1 Introduction

1.1 Overview

In Eth 2.0, nodes are chosen in a different order to forge on new blocks onto the blockchain, resulting in the requirement to download the state in a quicker way than currently.

The predominant syncing algorithms Full Sync and Fast Sync both take a considerable amount of time until a new node is set up and ready to participate in network activity. However, these algorithms cause problems in an environment that rapidly and randomly switches between validating nodes, as they require syncing in a shorter period.
Optimally, the time between being chosen to verify the next block and verifying the next block should be as short as possible.

While the concept of stateless clients goes back as far as 2017, the discussion began again around the end of 2019. It was mainly fueled by the announcement of Jason Carver's implemented version of a partly stateless syncing algorithm termed 'Beam Sync'.

So far, Beam Sync finds implementation in both Trinity and Nethermind clients, even though Trinity is currently still in alpha version.

Moreover, while Beam Sync is a step in the right direction, several other steps need to be taken until the general goal of 'stateless Ethereum' can be achieved entirely.

A visualization of the progress made can be seen in the figures 1.1 and 1.2.

Figure 1.1 shows the roadmap's initial visualization, as published in the official Ethereum blog in January 2020.

The most significant milestones that need to be reached to progress on the roadmap towards stateless Ethereum are pink. The possible ways in which those milestones can be achieved are purple. Since these ways still needed to be researched, the outcome of those leads to the overall goal could not be determined.

Green parts are research topics that are not necessarily required for progressing the main roadmap but could be beneficial. [Hotchkiss, 2020e]

Figure 1.2 shows the updated version of the roadmap, published in the same blog in April 2020.

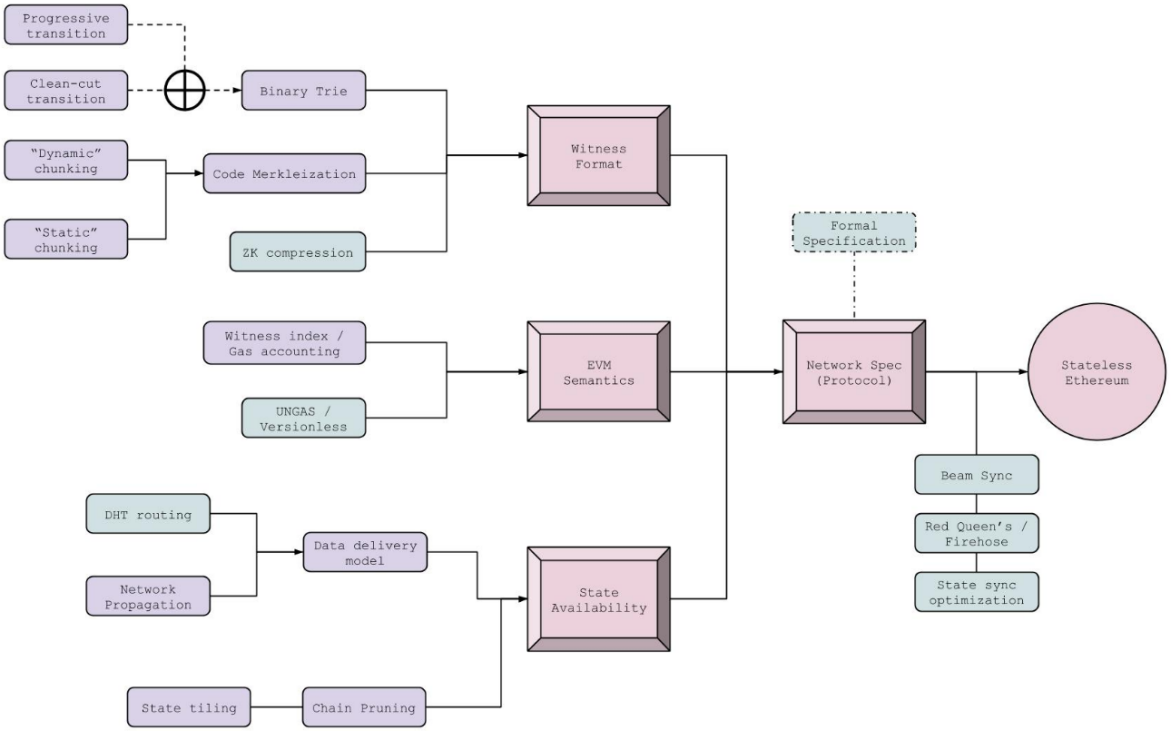

Figure 1.1: Stateless Tech Tree [Hotchkiss, 2020e]

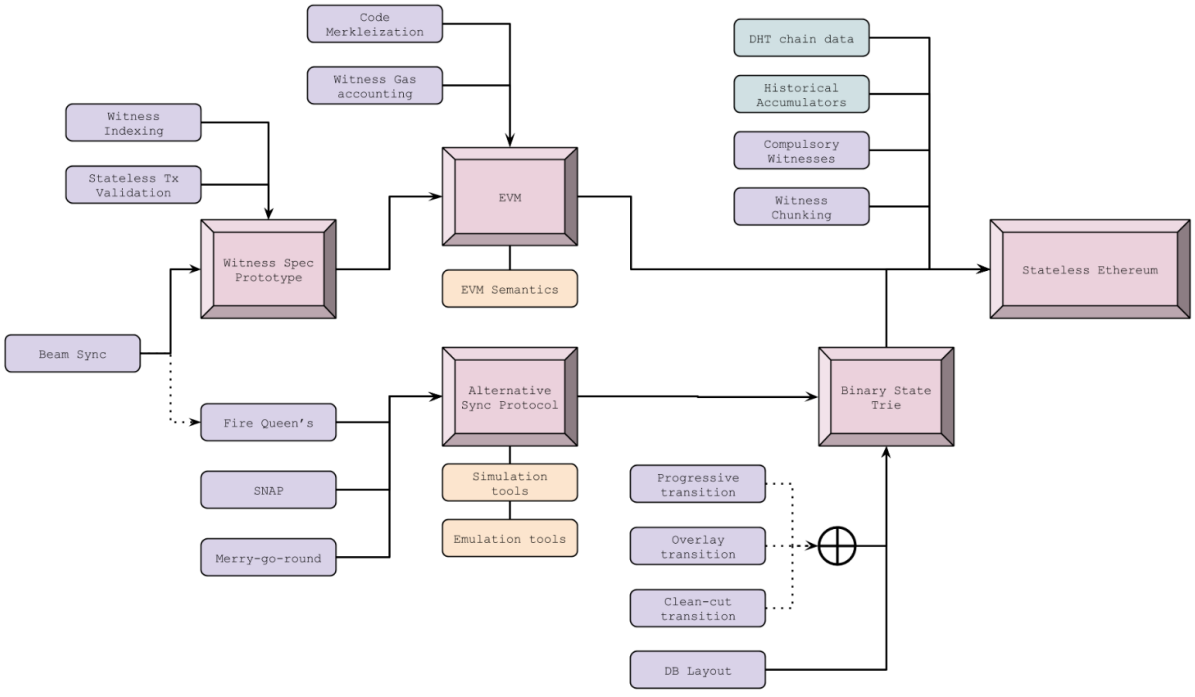

Figure 1.2: Updated Stateless Tech Tree [Hotchkiss, 2020g]

The newly added yellow topics are tools that are generally considered to be helpful overall but had not been created so far. [Hotchkiss, 2020g]

While the figures vary in their appearance, many items from figure 1.1 are still present in figure 1.2.

The previously purple topic 'Binary Trie' in figure 1.1 became a full milestone in figure 1.2. 'Code Merkleization' and 'Gas Accounting' are present in figure 1.1 but belong to the 'EVM' milestone in figure 1.2, while 'Witness indexing' and 'Stateless Tx Validation' have become attached to the reshaped milestone 'Witness Spec Prototype'.

Comparing both figures shows the tremendous effort and constant change the community puts into this transition, especially considering that only three months have passed between each publication.

1.2 Contributions

Although blockchain technologies have experienced much progress since their introduction, especially regarding general adoption, user-friendliness, and sophisticated tool-building, the number of people actively contributing to the infrastructure's code-based forefront is relatively small. This is especially true for Ethereum, which aims to have a maximally decentralized approach to developing its core features. By granting teams access to funding based on how vital the development pursued is regarding the ecosystem's overall benefit, teams and talented individuals contribute to the codebase on a global scale.

The main problems in this rapidly evolving space are not only keeping up with the research and development that happens every day, but also translating it into a more comprehensible, accessible way while emphasizing its essence.

Therefore, this thesis will examine how the traditional Stateful Tech Tree works in detail and how exactly the Stateless Tech Tree is supposed to get realized.

Explaining the Stateful Tech Tree will be achieved by specifying the concepts of Full Sync in general, Fast Sync on the example on the Geth client, and Warp Sync on the example of Parity client.

Comparing those approaches to the concept of stateless clients in general as well as to Beam Sync in specific will show not only the current point of development in Ethereum's transition towards statelessness, it also lays out the possibilities Beam Sync and other stateless clients' concepts will bring with them.

Resulting from this ongoing effort, the questions that are attempted to be answered in this thesis are:

- What are the differences between stateful and stateless Ethereum in general?
- What are the differences between stateful and stateless syncing in specific?

A particular emphasis lies in comparing Beam Sync to traditional approaches and the question of whether this reasonably new algorithm is a truly stateless client that can be used in a production-ready environment.

1.3 Outline

In order to answer the research questions, the next chapters will address the following topics:

In chapter two, essential basics regarding the Ethereum blockchain will be covered briefly to establish a fundamental understanding.

This includes a short history and definition of the Ethereum blockchain and a general definition of its state and the way the state is stored. Finally, the current client distribution will be illustrated to enhance understanding of the later introduced syncing algorithms.

Chapter three emphasizes on the Stateful Tech Tree, explaining the theoretical concept in detail and delving into the practical implementations of the Stateful Tech Tree for Full Sync in general, as well as Fast Sync and Warp Sync in specific. The chapter will conclude by determining the time complexity class of stateful syncing algorithms in general.

In comparison to chapter three, chapter four will be covering Ethereum's approach to a Stateless Tech Tree. By first outlining the theoretical concept's difference in detail, touching on the subjects of witnesses, the binary trie, code Merklization and changes to the Ethereum Virtual Machine, the practical implementation of Beam Sync will then be divulged. After determining the time complexity class of stateless syncing algorithms, the chapter concludes with problems, opportunities, and vectors of attack this new approach faces.

Finally, the thesis will close with a conclusion in chapter five.

2 Basics

2.1 Ethereum-Blockchain

Vitalk Buterin introduced the basic idea for the Ethereum Blockchain in 2013 via the 'Whitepaper'. [Buterin, 2014] While the initial crowd sale took place in 2014, after the release of the project's technical specification in the 'Yellowpaper' [Wood, 2014], the project officially launched on 30th July of 2015.

The Ethereum Foundation, the swiss-based organization behind the project, promotes Ethereum as "a blockchain with a built-in fully fledged Turing-complete programming language, that can be used to create 'contracts' that can (...) encode arbitrary state transition functions (...)". [Buterin, 2014]

Unlike Bitcoin, Ethereum offers the possibility to execute code: Ethereum has native programming languages that allow writing programs called 'smart contracts'. These smart contracts can be compiled into bytecode and deployed onto the blockchain.

The possibility to execute these programs is made possible through the Ethereum Virtual Machine (EVM) and a global state. At the moment, Ethereum's mainnet runs on a Proof of Work consensus mechanism that is at a larger scale both energy-inefficient and unable to deliver the needed throughput of transactions required to grow the network on a global scale.

Over the past years, there have been consistent efforts to overcome these limitations by migrating towards a more energy-friendly consensus algorithm and a different underlying infrastructure termed 'Eth 2.0'.

Eth 2.0 will work on a Proof of Stake consensus mechanism. This change requires nodes to stake a certain amount of Ether to participate in the block building process. However, it also changes the way block validating nodes are chosen for participation.[1]

This, in turn, requires a faster way to sync up new nodes in the existing network.

The problem is that even with the fastest syncing algorithm, it takes several hours of syncing for a new node to participate in the validation process of the network.

Therefore, it is fundamental for the syncing process to speed up for Eth 2.0 to work radically.

[1] Ether is Ethereum's native currency.

It is pivotal first to understand what state exactly is and how it is stored in Ethereum to understand all aspects of this problem.

2.2 State

On the most basic level, the state of Ethereum is a list containing keys and values. The 'current state' is the state of Ethereum after a new block was added. Since new blocks are continually added, the current state changes constantly.

Understanding the constant changing of the state is of pivotal essence.

Griffin Hotchkiss explains state in several different blog posts in the following way:

"The complete 'state' of Ethereum describes the current status of all accounts and balances, as well as the collective memories of all smart contracts deployed and running in the EVM. Every finalized block in the chain has one state, which is agreed upon by all participants in the network. That state is changed and updated with each new block that is added to the chain."[2]

Considering that each block has its own state, the process of syncing usually begins at the Genesis block. Executing each block's historical transactions and state and verifying them against the available data will inevitably lead to a node whose data is in sync with all the other nodes' data in the network.

Measured by the amount of data required to store the state, the question arises on how to store it in order to traverse it easily.

It also opens up a debate about other approaches in syncing up a new node with different motivations, such as a quicker time frame until participation with the network is possible.

[2] [Hotchkiss, 2020a, Hotchkiss, 2019]

2.3 Modified Merkle Patricia Trie

The data structure to implement the theoretical concept of state on a technological level is a modified Merkle-Patricia-Trie. [Wood, 2014]

This data structure consists of different trees, which will be explained briefly.

2.3.1 Merkle Tree

A Merkle tree is, in its essence, a tree consisting of hashes. The leaf nodes contain a value, and each branch node contains hashes of those values. A branch node of a branch node contains a hash of a hash of the leaf's value.

An example can be seen in 2.1.

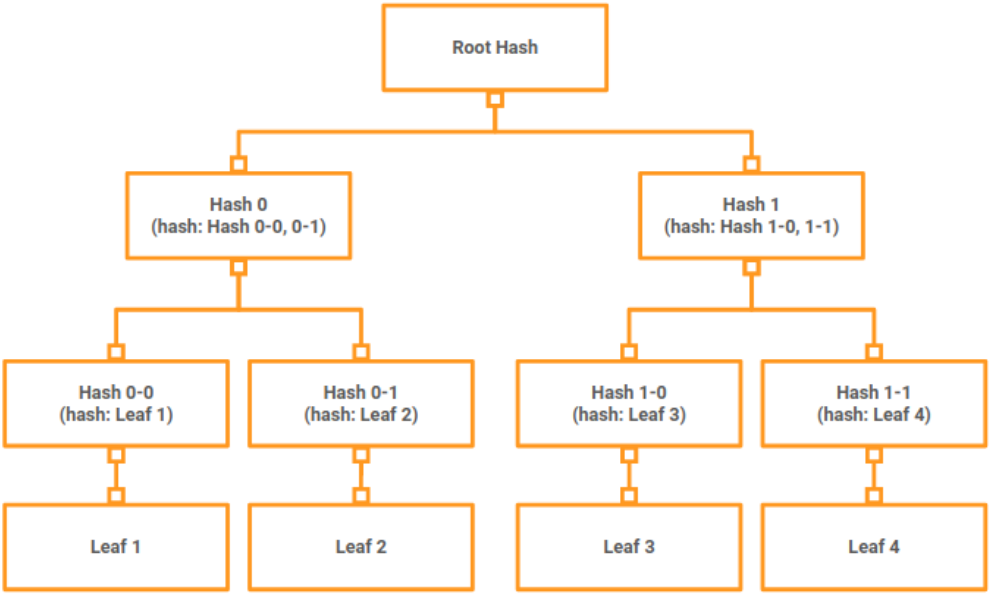

Figure 2.1: Example of a Merkle Tree

Applying this hashing-process recursively until all trees leaf nodes and branch nodes are traversed leaves one final root-hash to remain.

The most important aspect of a Merkle tree is that this single root hash can verify the whole data tree structure's integrity since the change of a single value in the data structure would lead to a completely different root hash.

Furthermore, this also allows for narrowing down quickly where precisely a fault in the overall data structure occurred.

2.3.2 Patricia Trie

"Patricia" is an acronym of the words "Practical Algorithm To Retrieve Information Coded In Alphanumeric". A Patricia Trie is a subversion of a Radix Trie, which can be seen in figure 2.2.

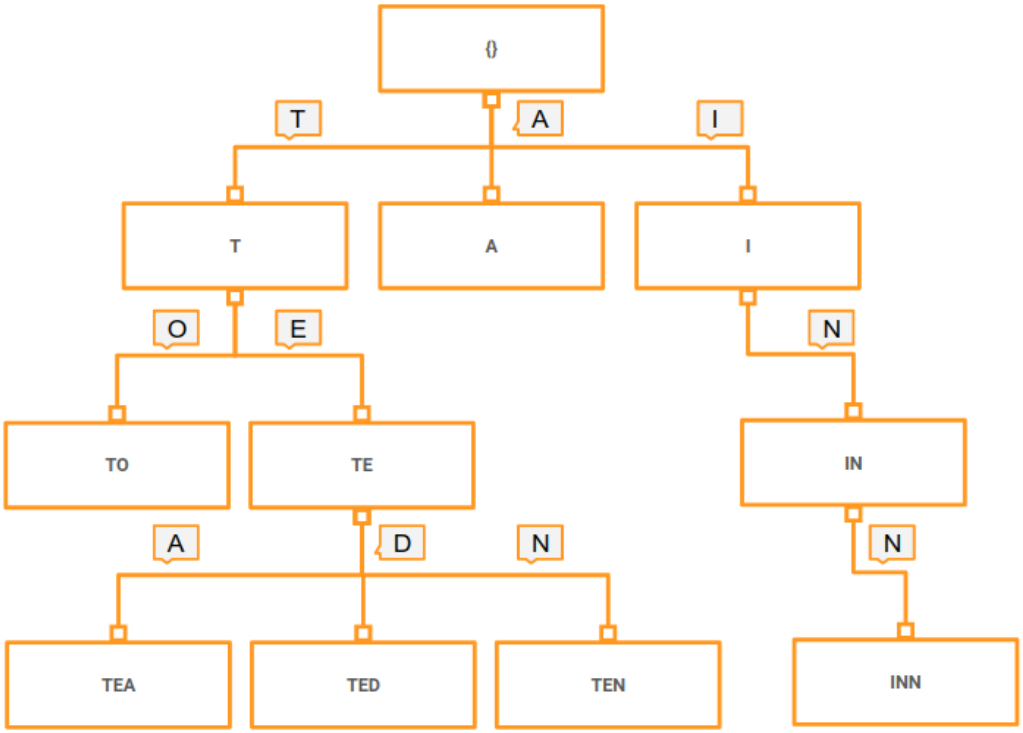

Figure 2.2: Example of a Patricia Trie

Depending on which branch of the tree is being traversed, the final result changes. Following the outermost left branch will lead first to "t", then adds an "o" and thereby forms the word "to". Instead of following the leftmost branch to the left, one could also follow the branch to the right that adds an "e", which offers three different result possibilities: tea, ted, ten.

2.3.3 Modification

The modifications of Ethereum's Merkle-Patricia-Trie is because a regular Merkle-Patricia-Trie is a binary tree. The Ethereum state tree is hexadecimal, allowing for Branch nodes in the Ethereum state tree to store up to 17 values.

2.4 Current Client Distribution

Based on data from the website ethernodes.org the overall distribution of different clients participating actively in Ethereum's mainnet can be seen in figure 2.3.

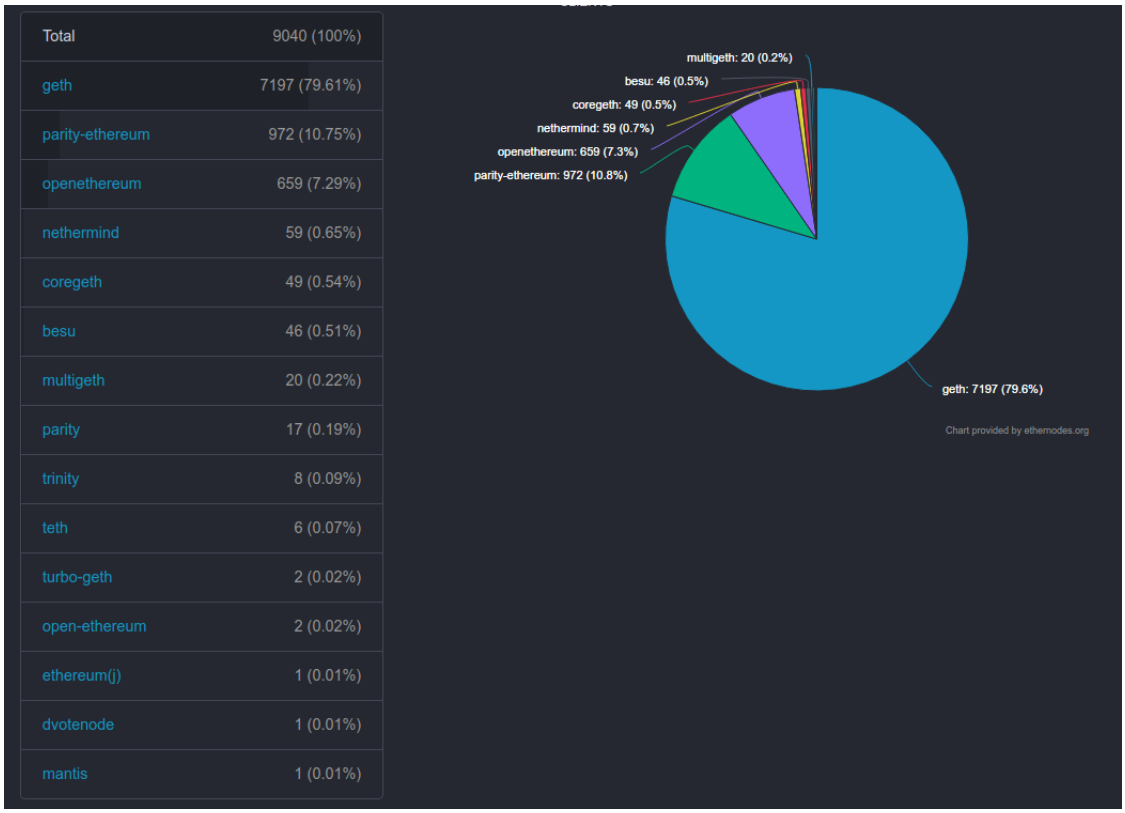

Figure 2.3: All Client Distribution according to Ethernodes.org

According to this figure, 79.6% of all nodes in the Ethereum network use Geth client, only 10.8% of nodes use Parity client, 7.3% use OpenEthereum, and only 0.7% respectively only 0.09% use Nethermind or Trinity clients.[3]

If this data gets purged by not completely synced nodes, there are only minor changes in the previous figure (as can be seen in figure 2.4).

Geth is still dominating with 82.2%, followed by Parity with 10.56% and Open-Ethereum with 5.93%. Nethermind and Trinity clients slip to 0.41% retrospectively 0.04%.[4]

[3] Changed data from October 4th: Geth - 81.15%, Parity - 9.92%, OpenEthereum - 6.79%, Nethermind - 0.73%, Trinity - 0.02%. Nethermind and Tritinty are currently the only clients using Beam Sync. It is also noteworthy that the Parity client is now maintained under the name OpenEthereum.

[4] Changed data from October 4th: Geth - 83.32%, Parity - 9.86%, OpenEthereum - 5.62%, Nethermind - 0.45%, Trinity - 0%.

Considering that OpenEthereum alone, as the third place from the top, has overall more market share than all following clients following summed up, it seems reasonable to focus solely on the three predominantly used clients in this thesis.

Additionally, it is crucial to notice that Trinity is still in the public alpha version intending to release a Beta version soon. Even though overall stability, functionality, and documentation are continually progressing, the current trinity usage has not yet reached its full potential.

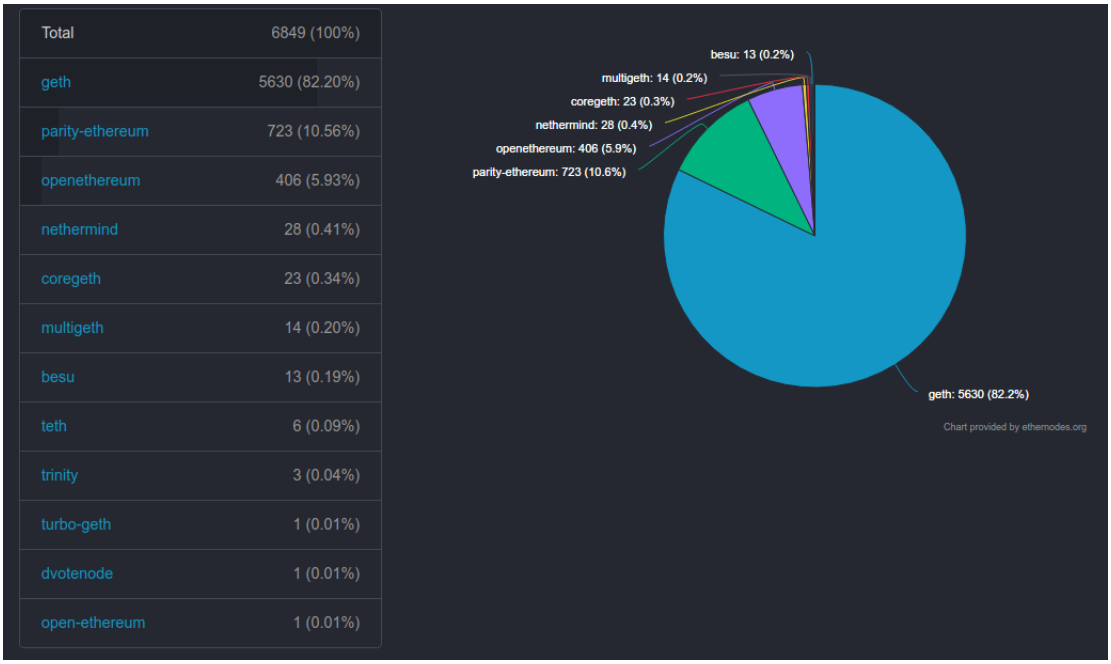

Figure 2.4: Fully Synced Client Distribution according to Ethernodes.org

3 Stateful Tech Tree

As covered in the previous chapter, Ethereum uses the concept of a modified Merkle-Patricia-Trie to store its state.

How this is achieved in detail is outlined in Ethereum's Yellowpaper. [Wood, 2014]

The following chapters will explain how exactly the Stateful Tech Tree works in theory and practice.

3.1 Theoretical concept

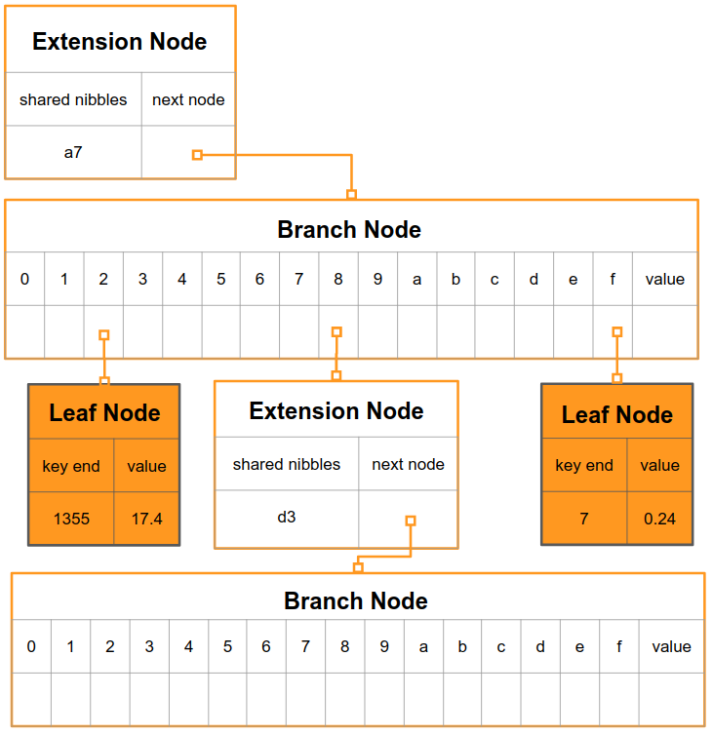

Figure 3.1: Simplified Types of Ethereum Nodes

At its core, Ethereum uses three different types of nodes in the trie. A simplified example of this is illustrated in figure 3.1.

The nodes have different responsibilities in the trie:

- **Extension nodes** contain two elements, a key and a pointer to the next node.
- **Branch nodes** contain up to 17 elements: 16 for every hexadecimal character and an additional field for value.
- **Extension nodes** contain two elements, a key and a pointer to the next node.

3.2 Practical Implementation

In the practical implementation, every client follows the theoretical concept outlined in the Yellowpaper to ensure uniformity of behavior across different clients. Both the Ethereum network and ecosystem benefit from having different clients in different programming languages participating in and upholding the infrastructure. Having a diverse ecosystem also ensures that more developers can participate more easily. It is also beneficial for the network's safety and security regarding resilience against different vectors of attack.

All of these factors apply to the underlying spirit of decentralization.

The majority of clients implement the state's theoretical concept by creating a distinct [key, value] pair for each node, where the value is the node itself, and the key is the hash of that node. An example of this can be seen in Figure 3.2

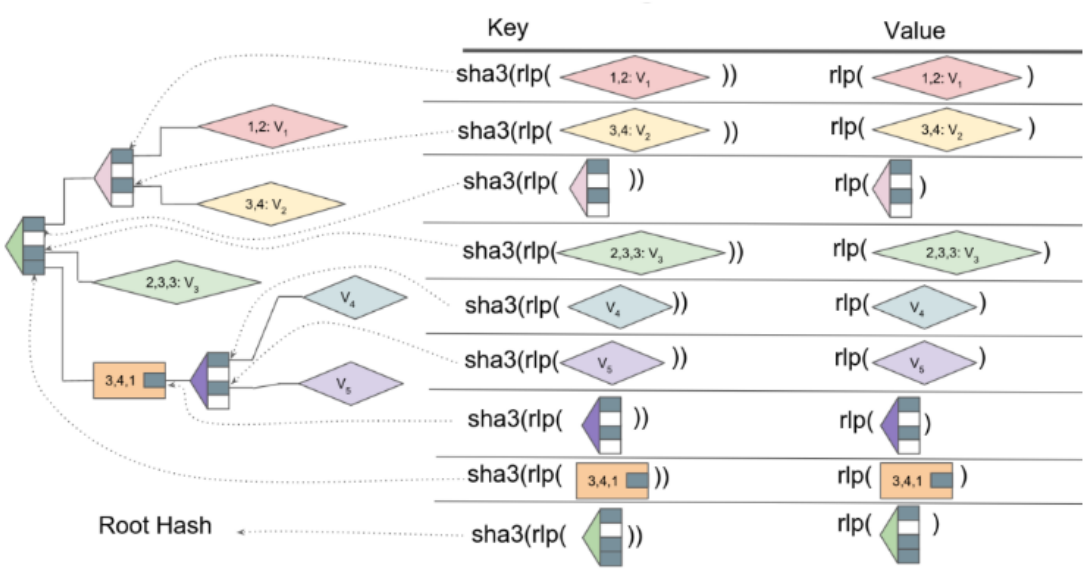

Figure 3.2: Key Value Structure in Tree [Hotchkiss, 2019]

According to the Patricia-Trie properties, the trie can be traversed downwards by following the pointers in the extension nodes and upwards by hashing the trie's branch.

A problem becomes apparent when changes to the trie are made: Changing a single entry leads to rehashing all parent node hashes of the specific branch recursively up to the root node. This causes intensive read/write operations on the actual disk that stores the trie.

"Those extra node types, leaf and extension, theoretically save on memory needed to store the trie, but they make the algorithms that modify the state inside the regular database more complex." [Hotchkiss, 2019]

So while the concept of the data-structure is aligned with the inherent feature of a blockchain, namely its immutability, it also puts a significant strain on the underlying hardware considering that every interaction with the trie results in various changes in vast parts of the trie.

Aside from the downside of having to change significant portions of the trie when single entries change with each block, the trie's growth also makes it harder for new nodes to sync.

The problem is not as bad for nodes that already store a part of the chain locally and have to re-sync from their last syncing point to the current state.

However, for a new node that wants to participate in the network for the first time, the process of rebuilding state is substantial: Starting at the Genesis block means re-executing all previous state changes along the length of the blockchain and computing the corresponding trie rehashing operations that lead to those state changes. Adding those read/write changes to the underlying hardware while following the blockchain's complete length up to the current state is a time-consuming process. That's why different clients started to implement different syncing algorithms to help them overcome this issue.

3.2.1 Full Sync

As already mentioned, the idea of Full Sync is to start the synchronization process with the Genesis block. It downloads the entire chain, including not only the blocks but also the receipts and logs. All of this data is then verified computationally.

The downsides of this approach, namely extensive computations and a longer time to sync, are scaled on the other side by increased security: Instead of trusting the downloaded data, each node syncing with Full Sync verifies for themselves that the network data is correct.

Clients

For a long time, Full Sync was the default when syncing the blockchain. This is why many clients still use it as either default or enable its usage as an option.

Geth and OpenEthereum both to have an option to sync a node with Full Sync.

However, both clients also have other syncing options, as the blockchain's growing size will take a longer time to sync.

3.2.2 Fast Sync

Fast Sync, as the name suggests, has a time advantage over Full Sync. This is achieved by trusting recent 'checkpoint' blocks. [Hotchkiss, 2019]

Instead of requesting transactions from the start of the blockchain, namely the Genesis block, clients can request state entries from these checkpoints. [Hotchkiss, 2019]

The downside of this time advantage is that trust is required in the data that is provided by the checkpoint.

For example, this could be exploited if the majority of miners colluded and would produce an arbitrary state change that violates the EVM rules. A fast-syncing node would not notice this violation and, therefore, participate in the faulty state after syncing.

Client: Geth

As can be seen in the Figures 2.3 and 2.4, Geth-Client is by far the most popular Ethereum client in use.

It is written in the language Go, which also explains the name: 'Geth client' is an abbreviation of 'go-ethereum client'.

Geth uses Fast Sync as the default syncing mode but also enables the use of Full Sync and Light Sync.[1][Team, 2020a]

A fast-syncing node that reached the end of the sync cycle will turn into a Full Node.

3.2.3 Warp Sync

Warp Sync uses an approach similar to Fast Sync: "Every 5,000 blocks, nodes will take a consensus-critical snapshot of that block's state." [OpenEthereum, 2020]

Considering that Ethereum adds new blocks every 15 seconds, a new snapshot is made every 21 hours.

The snapshot consists of three different parts: a unique manifest, state chunks, and block chunks.

While Block chunks contain blocks and their transaction receipts, State chunks contain the state of a given block. [OpenEthereum, 2020]

The manifest corresponding to the snapshot holds metadata of the snapshot. Through its uniqueness, the manifest can be used to identify snapshots.

[1] Light Sync will sync a Light Node by storing only the Header Chain and requesting everything else through the network peers. As used primarily for mobile phones or embedded devices, this sync mode will not be included in detail in this thesis.

Client: OpenEthereum (a.k.a. Parity)

Warp Sync is used natively in the OpenEthereum client. OpenEthereum is written in the Rust programming language. According to the statistics from figure 2.3, OpenEthereum is the third most widely used Ethereum client, with parity-ethereum being the second most commonly used client. [2]

3.3 Time Complexity Class

In overall consideration, all stateful syncing algorithms take a steady $O(n)$ time complexity to sync their nodes.

This is based on the inherent concept of downloading all blockchain-data: If the blockchain data size is n, the time it will take for a stateful node to sync up is $O(n)$.

The longer the chain gets, the longer the syncing takes, meaning that the time complexity is directly dependant on the number of blocks created.

Also, after the chain has reached the block where it originally started syncing, another syncing process needs to occur since new blocks were forged onto the blockchain while the node was still syncing up.

There are some additional factors, such a whether the client verifies the downloaded data by itself or whether the source can be trusted entirely.

However, those factors are either $+$ or $*$ operations, therefore do not change the complexity class overall.

[2] Parity Technologies started in 2015 and built the Parity client. Since the client's maintenance grew as more features were added, and the ecosystem of Ethereum changed constantly, Parity Technologies transitioned the client into a decentralized project called 'OpenEthereum' by the end of 2019. [Technologies, 2019] The project gets managed by developers and organizations that depend on the client and will continue its maintenance.

4 Stateless Tech Tree

After a post from Vitalik Buterin from 2017 regarding a concept-proposal for stateless clients, the idea for a Stateless Ethereum Tech Tree gained traction again since approximately Devcon V in October 2019 and was continued remotely and during the Stateless Summit in Paris at the beginning of 2020.[1]

Especially during the Stateless Summit, the main questions that were attempted to be answered can be summed up to

- What is the core problem that's tried to be solved?
- What is the first reasonable milestone that is being worked towards?
- Is it worth to investigate a zero-knowledge scheme for historical witnesses? [Hotchkiss, 2020d]

Asking these questions on a summit that happened only seven months ago and was attended by roughly 30 people in total (locally) is an excellent reminder of how few people are currently actively working on this issue in Ethereum in general, as well as how fluid and malleable the approach being taken is.

Even now, every part of perceived global research and development of Ethereum is due to a handful of single individuals (and their teams) building and influencing each other and on top of each other's ideas.

Hotchkiss summarised the subjective main topics into the following categories:

- Syncing primitives,
- Transition to Binary Trie,
- EVM,
- Data delivery in the stateless paradigm and
- the draft witness specification.[Hotchkiss, 2020d]

[1] The Ethereum Foundation supports the organization of hackathons and conferences to promote community-activity and innovation in the Ethereum ecosystem. Part of this is the yearly Developer Conference (DevCon), a gathering of developers in the Ethereum family that started in 2014 in Berlin and has been held every year except 2019 in different countries worldwide.

Since the Stateless Tech Tree is not completely implemented, new ideas or data insights of existing approaches can quickly turn the general development into another direction. [Hotchkiss, 2020h].

4.1 Theoretical concept

In his 2017 post "The Stateless Client Concept" on ethresear.ch, Buterin proposes to change the existing state transition function $STF(S, B) -> S'$ where S and S' are defined as states, B as a block and STF as a state transition function.

His proposed changes would define S as the root of S, B as (B, W) with W being a 'witness', so that STF's definition needs to be changed to STF' in the form of taking the state root and a 'block-plus-witness' as input to then output the new state by using the witness as a database to query accounts, storage keys or other state data. [Buterin, 2017]

Therefore, the change would mostly consist of blocks being updated with an extra 'witness' layer that clients can process.

Additionally, he stated that this modified form of blocks and clients could easily operate alongside traditional full nodes if a "translator node" communicates between the two sorts of nodes: either by adding witness information to a regular block so that a block produced by a full node client can be processed by stateless clients or by stripping away the witness-part of a modified block if the block was produced by a stateless client and shall be processed by a full node client and then subsequently rebroadcast the block in the corresponding network. [Buterin, 2017]

Buterin's original post was commented on over the years until 2019 when implemented solutions of this concept began to become more concrete.

The emergence of an implemented Proof of Concept by the end of 2019 was preceded by a medium-article of Alexey Akhunov in march 2019.

He condenses the main idea of Buterins concept as "stateless clients (...) [removing] the requirement of access to the entire state when executing transaction in the block" and provides the following figures to illustrate the difference of the stateless approach versus the current stateful approach. [Akhunov, 2019]

In figure 4.1 the idea of the stateful process can be seen. Each block execution (pink star) takes the historical state (yellow rectangle) and the transaction data inside the block bodies as input to make the EVM transition from one state to a modified state. After the transition was successful, the result is a new (current) state and receipts/logs (green rectangles).

The most current state in this figure is displayed at the brightest yellow rectangle.

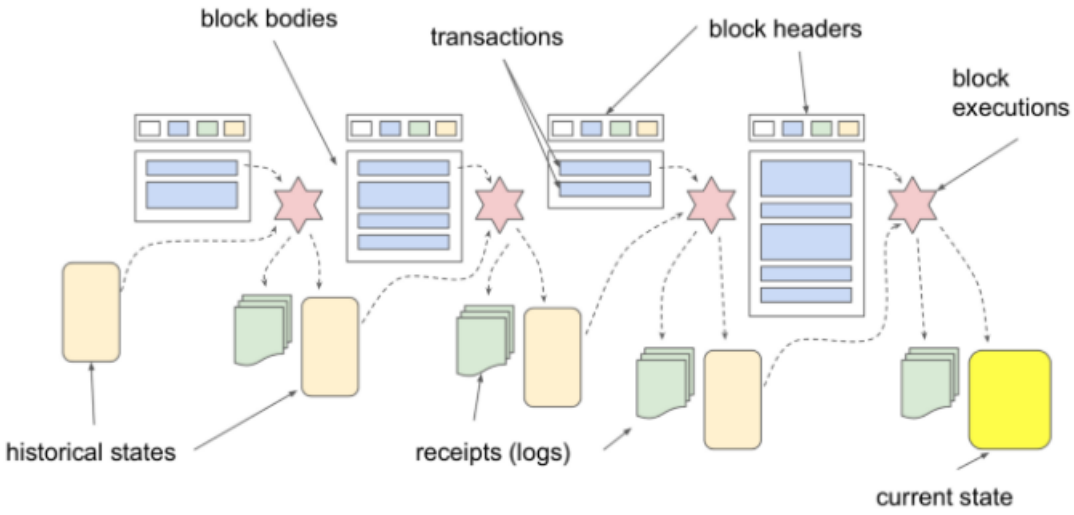

Figure 4.1: Stateful Process [Akhunov, 2019]

The essential difference in the idea of statelessness can be seen in figure 4.2:

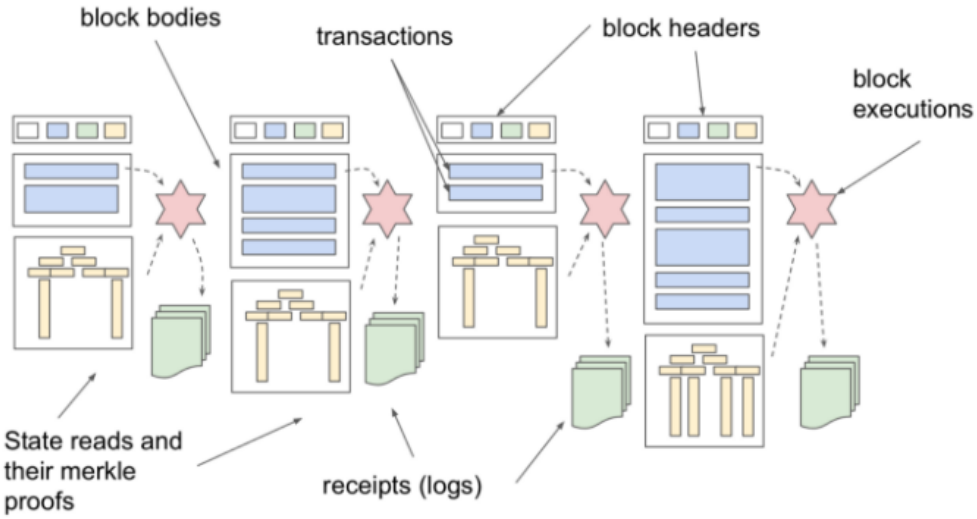

Figure 4.2: Stateless Process [Akhunov, 2019]

Instead of taking the historical state as input for a new block execution, there is also the possibility to take a set of state reads and their Merkle proofs (or a witness) as input. This would also allow for a state transition to complete but eliminate the immediate need for historical state data. Precisely as the stateful process does, this would also produce new receipts after each block execution.

However, this raises the question of how exactly the block proofs need to be structured and appended onto the block data (besides the block header and the block body) to use the existing network infrastructure.

Akhunov writes about a "separate data structure, containing all the extracts from the states that are necessary to execute all the transactions in the block." [Akhunov, 2019]

The combination of both processes can be seen in figure 4.3, where each block execution also outputs a new state.

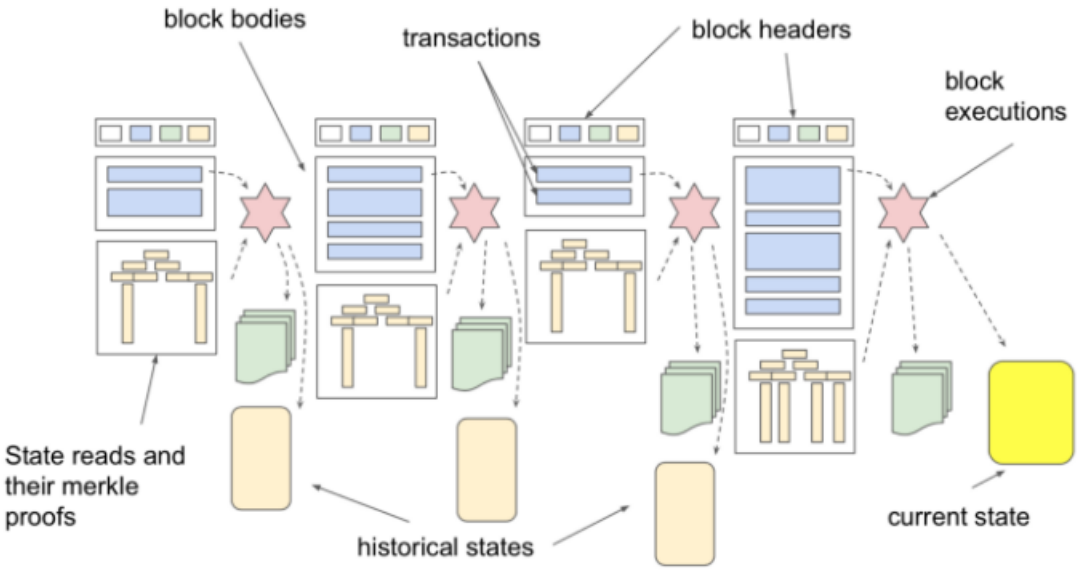

Figure 4.3: Combination of Stateful and Stateless Process [Akhunov, 2019]

In this figure, the connection between a new (historical) state and the next state read is missing: There needs to be a relation between a newly produced state and the state reads required for the next block execution - at least when the transactions in the next block touch any parts of the changed state created in the previous step. It is possible that the transactions in the next block do not touch any of the 'newly changed' state but would then touch another part of the state that was changed in a previous block execution. Akhunov may have omitted this relation for simplicity's sake.

An interesting take on Akhunov's article is his answer to whether the concept of statelessness can be introduced with or without a hard fork. Taking into consideration the

progress that was made since the publication of his article, the question should now be rephrased to: Is it possible to take all the necessary steps before introducing Eth 2.0 so that instead of hard-forking to introduce stateless Ethereum, this can be rolled out parallelly during the introduction of Eth 2.0?

4.1.1 Witnesses & Specification

As previously mentioned, a witness is needed for a state transition to be successful. Akhunov loosely defines witnesses as a "separate data structure, containing all the extracts from the states that are necessary to execute all the transactions in the block". [Akhunov, 2019]

In a blog post on ethresear.ch 1.5 years later, he concretizes his definition of witnesses:

While a witness is a separate piece of a Merkle proof that gets attached to blocks or transactions in the broadest sense, he declares it as "the auxiliary input that the oracle provides to a non-deterministic algorithm". [Akhunov, 2020]

In this case, the oracle produces input for the non-deterministic algorithm representing the modified 'stateless' Ethereum transition from one state to another.

In the official ethereum-repository on GitHub, the Readme for the stateless-ethereum-specs states this in a more specific manner: "Witnesses are a set of Merkle branches proving the values of all data that the execution of the block accesses." [contributors, 2020b]

They follow the idea of introducing block validators to read the encoded witnesses and check their validity. This can be achieved using the state data provided by a state provider for block validation or using the above Merkle branches, then comparing the computed Merkle root with the presented Merkle root. [contributors, 2020b]

What witnesses do is, therefore, generally well understood and sufficiently outlined. How they do it, however, was for a long time and still is an ongoing topic in general research.

As Hotchkiss phrased it: "Making [witnesses] viable for all clients is, after outlining concrete specification, a matter mostly of semantics and format." [Hotchkiss, 2020b]

The current specification for witnesses can be found in the ethereum/stateless-ethereum-specs repository on GitHub. [contributors, 2020a]

Especially the following aspects under 1.1.2 in the Witness Specification Readme,

- **Chunkable.** It should be possible to split witnesses in chunks independently verifiable to speed-up witness propagation in the network. The witness format does not limit a chunk size. That makes it easy to experiment with and find the best size for efficient relaying properties.
- **Compact.** The witness should have a binary format that is compact to save bandwidth when propagating the network.

are still in general discussion, and will be tended to in a later chapter.

The open questions regarding the structure of witnesses are therefore mainly concerning their

- Size,
- Indexing,
- Gas Accounting,
- Chunking and
- Stateless Transaction Validation.

The following chapters will outline each question's primary problem as well as summarize approaches for possible solutions.

Size

Hotchkiss states that "[w]itnesses are about 30% code and 70% hashes". [Hotchkiss, 2020c]

Additionally to ordinary blocks containing a block header and a block body, modified blocks containing witness data will be much bigger as per the data of the hashes that need to be included.

While ordinary blocks are in the range of being several 100 kb's in size, witnesses are probably in the range of being around 1 MB big. [Hotchkiss, 2019]

This additional data puts a big strain on the infrastructure, as regular blocks "make the propagation of blocks quick relative to network latency and the 15 seconds block time". [Hotchkiss, 2019]

However, since witnesses are much bigger, "[s]yncing a witness is much much slower relative to network latency and block time" and could prove problematic. [Hotchkiss, 2019]

The main question regarding witness blocks is, besides compressing their size, how to best handle them:

One approach aims to compress and reduce their size by changing the Ethereum trie hexadecimal structure into a binary trie structure.

Another approach deals with the ways witnesses can be stored on the network to make them easily accessible but still not overly present.

Indexing

Indexing witnesses is still a problem: Supposing nodes have no means of verifying that they got the correct witness for the transactions they want to execute before locally validating them. In that case, they need to request another witness and possibly another and another. This puts not only stress on the network in general, but there are also local computations happening that could have been prevented while time gets lost.

Introducing a native way in a stateless client ecosystem to verify that the requested witness is the correct one for the transactions that a node wants to execute would be beneficial.

The most common proposal to solve this is to change the protocol producing block headers to include a witnessHash.[2]

Adding this requirement into the protocol is believed to prevent potential attack vectors and make witness generation an explicit requirement for block-producing miners. [Hotchkiss, 2020f]

Gas Accounting

Another problem that the introduction of witnesses faces is their cost. The more transactions accumulate inside a block body, the more state gets touched through the transactions inside a block body. However, this concludes in more hashes needing to be included in the witness ergo, a bigger witness size in total. Nevertheless, the bigger the witness of a block gets, the more gas it will require upon creation.[3]

While it seems evident that each transaction's sender must pay a part of the combined gas cost of the witness [Hotchkiss, 2020g], the solution is not as simple as this would likely lead to an overpayment of the witness.[4]

Letting miners keep the overpaid gas could prove a smart move in that it could incentivize miners to produce witnesses. The downside however is that the current security model does not allow "trusting sub-calls (in a transaction) with a portion of the total committed gas."[5]

[2] [Hotchkiss, 2019], [Hotchkiss, 2020f]

[3] Gas is the computational power needed to make a change in the EVM. It usually gets denominated in gwei, which is a fraction of Ethereum's native currency Ether.

[4] The reason for this will be further explained in the next section "Chunking".

[5] [Hotchkiss, 2020c], [Hotchkiss, 2020g]

Chunking

Witness Chunking is closely related to Code Merklization under section 4.1.3 and deals with how the calling of smart contract methods should be handled regarding the usage of witnesses.

The basic idea is that a witness should not include the entire contract code but just those methods of the contract that were called.

Hotchkiss states that an additional 50% reduction in witness size could be possible due to chunking. He also summarizes that chunking could be done in two different approaches, static and dynamic, that might be compatible with each other and coexist.[6] [Hotchkiss, 2020c]

This also explains why the gas accounting of witnesses is not as simple as it seems: If the transaction sender pays for their code in the transaction to be processed into a witness, but part of their code gets 'chunked off', the sender paid more than necessary, resulting in the witness being overpaid with the miner potentially able to claim the difference.

Chunking is especially beneficial for partial-state nodes that observe certain parts of the state and rely on different witness chunks for other transactions. [Hotchkiss, 2020g]

Stateless Transaction Validation

A general problem of a different nature arises considering the verification of transactions in a stateless environment.

In the Stateful Tech Tree, nodes complete the entire syncing cycle to build a state before the validation of a transaction is possible. This stems from the two steps needed for validating a transaction: "First, the transaction nonce is checked to be consistent with all transactions from that account, and discarded if it is not valid. Second the account balance is checked to ensure that the account has enough gas money." [Hotchkiss, 2020b]

These steps, however, prove to be challenging to solve when navigating a stateless environment: As syncing nodes do not have access to the complete state, they cannot verify these transactions.

"In summary, clients need to quickly check if incoming transactions (waiting to be mined into a future block) are at least eligible to be included in a future block. (...) The current check, however, requires accessing data which is a part of the state, i.e. the sender's nonce and account balance. If a client is stateless, it won't be able to perform this check."[7]

[6] The difference between static and dynamic is the point at which a cleave happens: After a fixed size or after a JUMPDEST instruction in the codebase.

[7] [Hotchkiss, 2019], [Hotchkiss, 2020g]

Having no method for stateless clients to check transactions before validating them would open up the possibility of attacking the network by spamming invalid transactions.

Several suggestions have been made to solve this, the most favorable being the transactions requiring "to include merkle proof[s] of the sender's account". [lithp, 2020]

By this, the network would gain the ability to check whether transactions could theoretically be correct, even if the proofs are deprecated. In theory, this should discourage "transaction floods" because of the accompanying cost. [lithp, 2020]

4.1.2 Binary Trie

Apart from the introduction, specification, careful consideration, and implementation of witnesses as a data structure, a particular emphasis lies on the change of Ethereum's underlying state trie.

As determined in chapter 2.3, Ethereum's state is stored in a modified Merkle-Patricia-Trie, with the modification being the use of a hexadecimal trie.

While this data structure serves the need of the stateful ecosystem well, it needs to be improved for a stateless environment.

Figure 4.4 shows a hexadecimal trie as it is typically used. This example contains one branch node, one account node, and 2*15 = 30 Hashes, which equates to 32 hash elements.

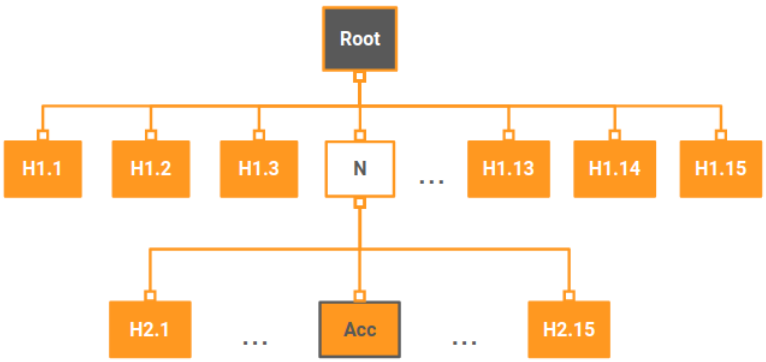

Figure 4.4: Example of a Hexary Trie

In figure 4.5, the same trie is stored in a binary form. This trie contains eight hashes, seven branch nodes, and one account node, which equates to 16 elements in a block witness.

While Hotchkiss states, "switching Ethereum's state to a Binary Trie structure is key to getting witness sizes small enough to be gossiped around the network without running into bandwidth/latency issues" [Hotchkiss, 2020g], Igor Mandrigin provides accompanying data: His conclusion is that binary witnesses are between 47 - 49 % smaller on average. [Mandrigin, 2020]

However, since these numbers prove that binary witnesses are smaller and therefore faster to transmit over a network, the question of how to change from the existing hexadecimal structure to the binary format remains. The best-case scenario disrupts the functionality and availability of the mainnet as little as possible.

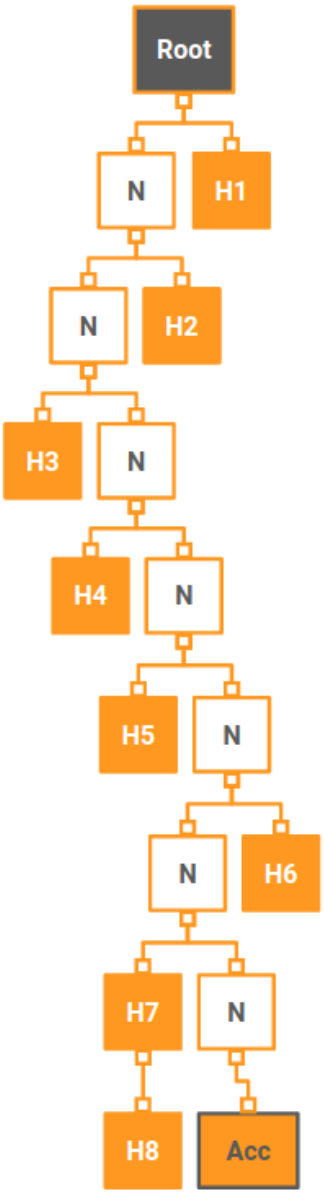

Figure 4.5: Example of a Binary Trie

Hotchkiss summarized in his January blog post that there are two possibilities of "progressive transition" and "compute and clean-cut" to update the hexadecimal trie [Hotchkiss, 2020c]. However, in March 2020, during the Stateless Summit, Guillaume Ballet introduced a different option.

His main idea is to use both tries during a transition period in which the mainnet can continue working unaffected while the binary trie is built continually. [Ballet, 2020]

In his summary of the Stateless Summit, Hotchkiss picks up the sentiment towards the new approach: "While formerly it was thought that the sound transition strategy to a binary trie would require a momentary halt to the chain and a re-computing of a new binary state, the new thinking is that the transition can be accomplished without network interruption with sufficient client coordination." [Hotchkiss, 2020d]

Overlay Conversion Method

Approaching the conversion from hex to binary trie is thought to work in three phases.

The first phase declares a block at a certain height to have two state roots: One for the read-only hex trie and one for the binary overlay trie.

When a transaction reads or updates the account, the binary trie will be searched first. If the account cannot be found in the overlay trie, the system will search for the account in the hex trie.

This allows for converting the hex trie in the background.

After phase one is complete, phase two can begin in which the new binary base root will be made read-only. At this point, there will still be two tries existing: Both are binary, but the base trie is read-only while the overlay trie is read-write.

As soon as an agreement is found on how the converted trie looks like, phase three can start merging the tries.

Merging the trees happens by deleting keys from the overlay and reinserting them into the base tree each time a new block is produced.[8]

Over time, this leads to the existence of a read-only binary tree containing all state data.

4.1.3 Code Merklization

As already mentioned in the subchapter regarding chunking, code Merklization is the idea of discarding code parts to gain a smaller witness size.[9] The difference is that code Merklization focuses on the smart contract side of compression instead of on the witness size.

Since witnesses contain parts of the state trie, transactions calling on contract methods need to equip witnesses with large amounts of data to verify the codeHash. [Hotchkiss, 2020g] Retrospectively, this would grow the witness size.

Therefore, the idea of code Merklization is "splitting up contract bytecode so that only the portion of the code called is required to generate and verify a witness for the transaction". [Hotchkiss, 2020g]

This, too, can be done in two approaches: splitting the data after "fixed code chunk sizes (...) or if code is validated with all static jumps enforced at validation". [Hotchkiss, 2020f]

The downside of code Merklization is that it has not been researched thoroughly, meaning no implementations are available.

[8] [Ballet, 2020], [gballet, 2020]
[9] See chapter 4.1.1.

4.1.4 Ethereum Virtual Machine Changes

Another problem stateless Ethereum is facing concerns the Ethereum Virtual Machine.

For example, the generation and propagation of witnesses through the network "needs to be accounted for in EVM operations". [Hotchkiss, 2020g]

The problem is that changes to the EVM are regarded as rather complicated, resulting in no unified idea of approaching this issue.

Hotchkiss summarizes that "some gas operations are expected to get more expensive no matter what, but nothing has really been determined with regard to the EVM, and we won't be able to know what the best course is until we get more data". [Hotchkiss, 2020d]

4.2 Practical Implementation

As should have become apparent in the previous chapters, a truly stateless Ethereum is still under construction. While some advances have been made regarding certain parts of the implementation, other aspects still need to be researched more until a course of action can be recommended and implemented.

Until then, the general direction of practical implementation is open to alternative suggestions, as is the general research.

4.2.1 Beam Sync

Jason Carver states in his article "Intro to Beam Sync" that while "Beam Sync is a direct evolution of Fast Sync" it essentially is a merge between a "guided Fast Sync to simulate a Stateless Client at first" which later "fade[s] into a Full Sync". [10] [Carver, 2019]

Using block witnesses to determine the state's parts that need to be requested from peers to execute a block makes this possible.

"Over time, as more and more of the state is touched by the beam-syncing node, it can build up a complete state and switch over to full Sync. " [Merriam, 2020]

Retrospectively, Beam Sync dealt with a lot of general problems:

- **Witness specification:** The first pull request for the "Witness Specification Draft" on the ethereum/stateless-ethereum-specs repository was made in March 2020 by Igor Mandrigin. Before that, the general scope of duties for witnesses was broadly understood but never formally specified.
- **Chunking and Code Merklization:** As mentioned previously, there is still no solid consensus on how to achieve either of these, so no usable implementation has been published that can be used to shrink the size of witnesses at the moment.
- **Binary Trie:** The idea of switching from a hex trie to a binary one was generally thought of, but specific approaches on how the transition may be achieved were not formalized until the beginning of this year.

A problem that still exists is that peers on the network do not store historical state indefinitely. Carver cites the 120 blocks held by Geth as a benchmark. [Carver, 2019]

[10] In this context it is important to note that the article was published by the end of September in 2019. A lot of the progress towards Stateless Ethereum that was covered in the previous chapters hadn't been made at that time. The implementation of Beam Sync alongside with the articles accompanying it essentially fuelled on the general discussion of Stateless Ethereum again, two weeks before Devcon V and months before the Stateless Summit happened.

This problem, however, turned into a significant advantage for Beam Sync's algorithm:

Beam Sync initiates the initial sync process at the tip of the chain. Since there will inevitably be a point when the network peers have moved on and will not deliver the requested historical states anymore, Beam Sync pivots.

Pivoting means essentially surrendering the 'launch block' and choosing a new, more current launch block. Ultimately this will ensure that the most current state gets stored in the local database first by starting from a fixed point on and then keeping up with the head of the blockchain. A background process queries and holds the state that was never touched or was too old and fills up the local database.

In this manner, a beam-syncing node can, over time, build the entire historical state before switching to Full Sync.

The beam-syncing node "essentially operat[es] in a stateless paradigm, (while) moving gradually toward statefulness as it back-fills its local database". [Merriam, 2020]

4.2.2 Clients

Beam Sync is implemented in the clients Nethermind and Trinity.

Trinity

The Trinity client is written in Python. Jason Carver is part of the Trinity Team. The disadvantages of a Python-written client against faster clients, for example, clients written in Rust or Go, were essential in the research of alternative syncing algorithms and the development of Beam Sync.

This project, alongside py-EVM, is the successor of the py-ethereum project.

Since Trinity operates in a public alpha stage, it is not meant for production use yet. [Team, 2020d]

Nethermind

Beam Sync is also implemented in the Nethermind client, a .NET client for Ethereum. [Team, 2020b]

On its official documentation, Nethermind lists a comparison of sync modes that include Fast Sync and Beam Sync and can be seen in figure 4.6.

While Beam Sync is significantly slower than Fast Sync regarding pure syncing time, it is also substantially faster regarding the time to RPC. This time frame indicates how

Sync Mode	Disk Space needed	Full current state	Full current and all historical states	Can sync a full archive node from this	Time to sync	Time to RPC
archive	~5TB	YES	YES	YES	~3 weeks	~3 weeks
fast sync	~300GB	YES	NO	YES	~15 hours	~15 hours
fast sync without receipts	~120GB	YES	NO	YES	~10 hours	~10 hours
fast sync without bodies and receipts	~60GB	YES	NO	NO	~6 hours	~6 hours
beam sync	~320GB	YES	NO	YES	~30 hours	~20 minutes
beam sync without receipts	~140GB	YES	NO	YES	~20 hours	~20 minutes
beam sync without bodies and receipts	~80GB	YES	NO	NO	~10 hours	~20 minutes
beam sync without historical headers	~60GB	YES	NO	NO	~9 hours	~5 minutes

Figure 4.6: Nethermind Comparison Statistics [Team, 2020c]

long the client will need to run in the corresponding sync mode before RPC queries can be made, and therefore interaction with the blockchain data is made possible.

The difference of 20 GB in the needed total disk space between Fast Sync and Beam Sync can be explained with additional witness data.

4.3 Time Complexity

The time complexity class of Stateless Ethereum can be considered to be pure $O(1)$, as only the request of the single most current witness would be needed until a node can perform remote process calls and start interacting with the blockchain.

The time complexity of Beam Sync can generally be considered $O(1)$.

After requesting the state stored in the block witness, and receiving that state, interaction with the blockchain data can happen nearly 'instantly'.

"Fast Sync must download the full state before executing its first block. Beam Sync only needs to download a single block's witness." [Carver, 2019]

The amount of time a beam syncing node takes to sync the complete blockchain is still $O(n)$ in the worst case as the time grows alongside the growth of blockchain data.

4.4 Problems, Opportunities, and Vectors of Attack

Beam Sync is not a purely stateless syncing algorithm. If it were, the corresponding client "would never keep a copy of state; it would only grab the latest transactions together with the witness, and have everything it needs to execute the next block". [Hotchkiss, 2019]

However, simply not storing the requested state is not the solution to turn beam syncing clients into fully stateless clients.

Piper Merriam of the Trinity team summarizes this as follows: "A client that is syncing towards being a full node can be viewed as triggering some accumulation of 'debt' on the network, as they are requesting more data than they are providing to the network. The network can handle some finite volume of this behavior. Under the assumption that all nodes eventually become full nodes, most nodes eventually pay back this network debt by providing data to other nodes." [Merriam, 2017]

In the strictest sense, a stateless node can thereby be considered slightly malicious - by requesting data, it puts additional stress on the network without contributing back. This behavior can be compensated as long as such nodes do not occur in large numbers.

Beam Sync is also not an efficient stateless client, as Merriam acknowledges: "'Beam' Sync can be viewed as a highly inefficient stateless client. The mechanism by which it fetches state data on-demand via the GetNodeData primitive requires traversal of the state trie one node at a time. Naively this means issuing requests for single trie nodes to a peer as the client steps through EVM execution, encountering missing state data." [Merriam, 2017]

Additionally, however, there is a lot of data to be gained: "We can learn a lot about desirable network primitives by trying to break [B]eam Sync, and then iterate on those lessons to bring [B]eam Sync closer to the 'real' stateless [s]ync paradigm." [Hotchkiss, 2020f]

As per the end of June 2020, there is even a whole different approach to reaching stateless Ethereum termed 'reGenesis'. This roots in the idea that certain historical state parts may become completely obsolete.

Hotchkiss released a modified stateless Ethereum roadmap that can be seen in figure 4.7.

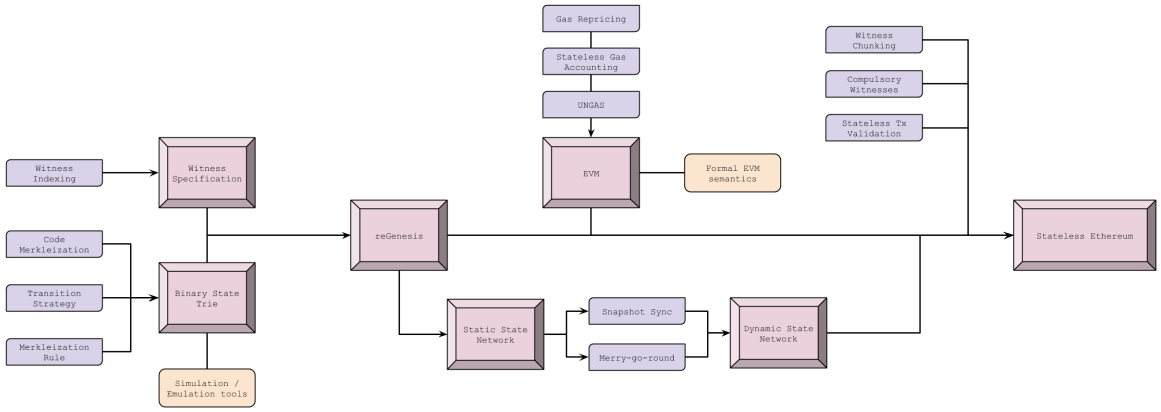

Figure 4.7: ReGenesis Tech Tree [Hotchkiss, 2020h]

This changes the direction of the previous roadmap drastically, with fewer steps to be taken until stateless Ethereum can be reached.

ReGenesis is exciting because it "sidesteps some of the largest challenges with Statelessness, i.e., how witness gas accounting works during EVM execution". [Hotchkiss, 2020h]

5 Conclusion

As has been shown in the previous chapters, Ethereum's default Stateful Tech Tree stores state in a hexadecimal Merkle-Patricia-Trie and has different approaches towards syncing.

While Full Sync is the most common default sync mode and offers the highest security but longest synchronization time, Fast Sync and Warp Sync compromise security by accepting data from alternative sources in favor of syncing speed. Their respective clients are widely used and make up most of the overall client distribution.

Because of the stateful clients' structure, the time complexity class stays roughly constant for either syncing algorithm: stateful nodes need a completed syncing cycle before interaction with the blockchain in all cases, even if they do not start syncing at the Genesis block.

In comparison, Ethereum's new approach to a Stateless Tech Tree, albeit still in development, has made vast advances in the last year alone. The suggested theoretical concept from 2017 has become more refined due to theoretical and practical research.

A draft for witness specification has been submitted. Although the points of overall witness size, witness indexing, witness gas accounting, witness chunking, and stateless transaction validation still need more research before implementation can happen, a foundation has been built that gives further research an aligned direction.

Also, a unified approach has been reached regarding the hex trie conversion process towards a binary one using the overlay conversion method.

Other milestones, such as changes to the EVM, still need more data before suggestions on their modification can be made.

The results of comparing the complexity classes of both approaches also looks promising: By splicing the syncing process in two different strains, one allowing for near-instantaneous interaction with the blockchain while the other builds the local database in regular time, the overall of substantially quicker interaction with the blockchain has been made possible.

In summary, it was established that the approach of stateless clients is more than the application of different syncing algorithms: A stateless paradigm requires deeply rooted changes in the underlying protocol of Ethereum. By appending a new data structure onto regular blocks, a witness layer is created that allows for rapid interaction with the blockchain for new nodes.

Alongside the changes in the protocol, there needs to be careful consideration for the specific implementation, as well as for the impact on the network infrastructure and the ecosystem in general.

While specific topics still need more research until a working solution can be evaluated and possibly considered for implementation, other aspects have allowed for a functioning Proof of Concept (Beam Sync) being implemented in two clients.

Concerning the research questions differentiating between the Stateful and Stateless Ethereum Tech Trees in general, it has been found out that though the stateless approach is still under development, the differences to the stateful approach are vast and promising.

About the stateful and stateless syncing algorithms in specific, no generalized answer has been found since Beam Sync operates only partially in a stateless environment.

Regarding the approach of Beam Sync it was revealed that it is not a truly stateless client, since it builds a stateful database in the background process, however operates in a stateless paradigm during the initial syncing process. Through the implementation in the Nethermind client, Trinity found usage in a production-ready environment.

It is safe to assume that without the publication of an implemented Proof of Concept for Beam Sync at the end of 2019, a great deal of research and development would not have happened in 2020.

Considering the overall research and development in the direction of stateless clients, it is reasonable to assume that a genuinely stateless client will be introduced into the Ethereum ecosystem soon, offering the possibility to shorten the time frame in which clients can sync the blockchain significantly while also introducing a new data structure on the blocks.

List of Figures

Bibliography

[Akhunov, 2019] Akhunov, A. (2019). Data from the Ethereum stateless prototype. https://medium.com/@akhounov/data-from-the-ethereum-stateless-prototype-8c69479c8abc. Accessed: 16.10.2020.

[Akhunov, 2020] Akhunov, A. (2020). Terminology: What do we call "witness" in "Stateless Ethereum" and why it is appropriate. https://ethresear.ch/t/terminology-what-do-we-call-witness-in-stateless-ethereum-and-why\-it-is-appropriate/7602. Accessed: 11.10.2020.

[Ballet, 2020] Ballet, G. (2020). Ethereum state tree format change using an overlay. https://medium.com/@gballet/ethereum-state-tree-format-change-using-an-overlay-e0862d1bf201. Accessed: 14.10.2020.

[Buterin, 2014] Buterin, V. (2014). A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/. Accessed: 10.10.2020.

[Buterin, 2017] Buterin, V. (2017). The Stateless Client Concept. https://ethresear.ch/t/the-stateless-client-concept/172. Accessed: 11.10.2020.

[Carver, 2019] Carver, J. (2019). Intro to Beam Sync. https://medium.com/@jason.carver/intro-to-beam-sync-a0fd168be14a. Accessed: 16.10.2020.

[contributors, 2020a] contributors, . (2020a). Block Witness Formal Specification. https://github.com/ethereum/stateless-ethereum-specs/blob/master/witness.md. Accessed: 12.10.2020, Latest Commit: 5141f46.

[contributors, 2020b] contributors, . (2020b). stateless-ethereum-specs. https://github.com/ethereum/stateless-ethereum-specs/blob/master/README.md. Accessed: 12.10.2020, Latest Commit: 3f96836.

[gballet, 2020] gballet (2020). Overlay method for hex to bin tree conversion. https://ethresear.ch/t/overlay-method-for-hex-bin-tree-conversion/7104. Accessed: 14.10.2020.

[Hotchkiss, 2019] Hotchkiss, G. I. (2019). The 1.x Files: The State of Stateless Ethereum. https://blog.ethereum.org/2019/12/30/eth1x-files-state-of-stateless-ethereum/. Accessed: 18.10.2020.

[Hotchkiss, 2020a] Hotchkiss, G. I. (2020a). The 1.x Files: A Primer for the Witness Specification. https://blog.ethereum.org/2020/05/04/eth1x-witness-primer/. Accessed: 18.10.2020.

[Hotchkiss, 2020b] Hotchkiss, G. I. (2020b). The 1.x Files: February call digest. https://blog.ethereum.org/2020/02/28/eth1x-files-digest-no-3/. Accessed: 10.10.2020.

[Hotchkiss, 2020c] Hotchkiss, G. I. (2020c). The 1.x Files: January call digest. https://blog.ethereum.org/2020/01/17/eth1x-files-digest-no-2/. Accessed: 10.10.2020.

[Hotchkiss, 2020d] Hotchkiss, G. I. (2020d). The 1.x Files: Stateless Summit Summary. https://blog.ethereum.org/2020/03/12/eth1x-files-stateless-summit-summary/. Accessed: 10.10.2020.

[Hotchkiss, 2020e] Hotchkiss, G. I. (2020e). The 1.x Files: The Stateless Ethereum Tech Tree. https://blog.ethereum.org/2020/01/28/eth1x-files-the-stateless-ethereum-tech-tree/. Accessed: 18.10.2020.

[Hotchkiss, 2020f] Hotchkiss, G. I. (2020f). The 1.x Files: The State(lessness) of the Union. https://blog.ethereum.org/2020/02/18/eth1x-files-the-statelessness-of-the-union/. Accessed: 12.10.2020.

[Hotchkiss, 2020g] Hotchkiss, G. I. (2020g). The 1.x Files: The Updated Stateless Tech Tree. https://blog.ethereum.org/2020/04/02/eth1x-stateless-tech-tree/. Accessed: 18.10.2020.

[Hotchkiss, 2020h] Hotchkiss, G. I. (2020h). The Stateless Tech Tree: reGenesis Edition. https://blog.ethereum.org/2020/08/25/the-1x-files-tech-tree-regenesis/. Accessed: 15.10.2020.

[lithp, 2020] lithp (2020). The current transaction validation rules require access to the account trie. https://ethresear.ch/t/the-current-transaction-validation-rules-require-access-to-the\-account-trie/7046. Accessed: 18.10.2020.

[Mandrigin, 2020] Mandrigin, I. (2020). Stateless Ethereum: Binary Tries Experiment. https://medium.com/@mandrigin/stateless-ethereum-binary-tries-experiment-b2c035497768. Accessed: 14.10.2020.

[Merriam, 2017] Merriam, P. (2017). The Data Availability Problem under Stateless Ethereum. https://ethresear.ch/t/the-data-availability-problem-under-stateless-ethereum/6973. Accessed: 11.10.2020.

[Merriam, 2020] Merriam, P. (2020). Stateless Clients: A New Direction for Ethereum 1.x. https://medium.com/@pipermerriam/stateless-clients-a-new-direction-for-ethereum-1-x-e70d30dc27aa. Accessed: 18.10.2020.

[OpenEthereum, 2020] OpenEthereum (2020). Warp Sync Snapshot Format - Wiki. https://openethereum.github.io/wiki/Warp-Sync-Snapshot-Format. Accessed: 18.10.2020.

[Team, 2020a] Team, G. (2020a). Official Geth Documentation. https://geth.ethereum.org/docs/faq. Accessed: 18.10.2020.

[Team, 2020b] Team, N. (2020b). Official Nethermind Documentation. https://docs.nethermind.io/nethermind/. Accessed: 15.10.2020.

[Team, 2020c] Team, N. (2020c). Official Nethermind Documentation. https://docs.nethermind.io/nethermind/ethereum-client/sync-modes. Accessed: 15.10.2020.

[Team, 2020d] Team, T. (2020d). Official Trinity Documentation. https://trinity-client.readthedocs.io/en/latest/. Accessed: 15.10.2020.

[Technologies, 2019] Technologies, P. (2019). Transitioning Parity Ethereum to OpenEthereum DAO. https://www.parity.io/parity-ethereum-openethereum-dao/. Accessed: 18.10.2020.

[Wood, 2014] Wood, D. G. (2014). Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf. Accessed: 02.09.2020.