Implementing Stateless Clients in Ethereum

Souvik Banerjee

ABSTRACT

Ethereum is a blockchain based cryptocurrency and platform for decentralized applications. A key problem that Ethereum currently faces is scalability. Ethereum can only process about 15 transactions per second, while other payment processors like Visa can process 24,000 transactions per second. One reason for these scalability problems is due to disk I/O operations. Stateless Clients is a proposal to improve the scalability of Ethereum by reducing the amount of I/O operations Ethereum nodes need to perform. The Stateless Clients proposal does this by using Merkle proofs of state values accessed during transaction execution. These Merkle proofs are called witnesses. Witnesses are created and sent along with transactions in blocks from miners to other Ethereum nodes. Ethereum nodes use witnesses to verify transactions without needing access to the Ethereum state, which greatly reduces the amount of I/O operations. However, a major problem with this proposal is that the size of the witnesses is extremely large, which makes using Stateless Clients impractical. In this paper, I show how Stateless Clients works in detail, discuss how I implemented Stateless Clients in the Parity Ethereum client, and present results that demonstrate that the Stateless Clients proposal is not practical.
Contents

1 Introduction 4

2 Background 5

2.1 Ethereum Virtual Machine 5

2.2 Ethereum Transactions and Blocks 5

2.3 Recursive-Length-Prefix (RLP) Encoding 7

2.4 Merkle-Patricia Trie 7

2.5 Consensus 10

3 Design 11

3.1 Motivation 11

3.2 Stateless Clients 13

3.3 Example 14

3.3.1 Block creation 14

3.3.2 Block verification 18

4 Implementation 20

4.1 State backend 20

4.1.1 Witness Creation 22

4.1.2 Witness Verification 22

4.2 Serialization of witnesses 23

4.3 Block sync protocol 24

5 Evaluation 26

5.1 Correctness 26

5.2 Results 26
1 Introduction

In the last few years, blockchain technology has become increasingly popular. Blockchain is a decentralized data sharing technology that works in a distributed environment consisting of multiple nodes. It can be defined as an immutable ledger that records transactions. The blockchain structure consists of cryptographically signed blocks that are linked to each other with cryptographic hashes. Each block contains transactions that show value transfers between different accounts. Each node in the network maintains its own copy of the blockchain, which makes the ledger a distributed ledger. The ledger is kept up to date using a distributed consensus mechanism, which allows nodes to validate transactions, package them into blocks, and update the block hash chain.

Blockchain and cryptocurrency are two different concepts, but their development has been intertwined. Blockchain was first suggested by Satoshi Nakamoto’s paper introducing Bitcoin [1] in late 2008. Bitcoin is a cryptocurrency and is a specific implementation of blockchain technology.

Ethereum [3] is a cryptocurrency that builds upon the ideas introduced in Bitcoin. Ethereum is a transaction-based state machine that starts with some initial genesis state, incrementally executes transactions, and arrives at a final state. These transactions are collected into blocks, and blocks are hashed to create a hash chain. The Ethereum protocol has a native currency called Ether and is abbreviated as ETH.

The goal of Ethereum is to create a platform for decentralized applications, and it does this by incorporating a built-in Turing complete programming language. This allows users to write smart contracts that contain arbitrary code and run on the decentralized Ethereum network. In contrast to Bitcoin, where transactions are simply value transfers between accounts, Ethereum transactions can create and run smart contracts. The ability to run arbitrary code makes Ethereum an extremely powerful abstraction layer on which decentralized applications can be built.

One problem with blockchain-based systems is that they are orders of magnitude slower than their centralized equivalent. Ethereum is only able to process about 15 transactions per second [12], while payment processors like Visa can process 24,000 transactions per second [22]. New distributed applications that run on Ethereum such as CryptoKitties [16] have slowed down transaction processing time even further. Improving the scalability of blockchain-based systems is crucial for widespread adoption.

A proposed solution to Ethereum’s scalability problems is called Stateless Clients [15]. Stateless Clients was proposed by Vitalik Buterin, the founder of Ethereum, in 2017. Stateless Clients aims to improve scalability by reducing I/O when verifying blocks. However, until now there has been no working prototype of Stateless Clients.

The structure of this paper is as follows: first, I will review some foundational concepts in Ethereum. Second, I will describe in more detail how Stateless Clients is designed and how I implemented Stateless Clients in Parity [6]. Third, I will show some of the results I obtained from my implementation.
2 Background

2.1 Ethereum Virtual Machine

Ethereum is a platform for deploying and running smart contracts [24]. Smart contracts are programs that enforce a contract and run on the Ethereum Virtual Machine, or EVM. Smart contracts are created with the Solidity programming language and compile to EVM opcodes [10]. The Ethereum blockchain is replicated on many nodes and all nodes verify the correct execution of the smart contract. This makes smart contracts useful for self-executing agreements between two parties.

These EVM opcodes allow the EVM to be Turing complete, limited only by the following factors:

1. Stack space: The stack has a capacity of 1024 entries, and each entry is 256 bits. The EVM is a stack-based virtual machine and the stack is used to push arguments to opcodes.

2. Execution time: Ethereum measures “computational effort” using gas. Each opcode in a contract requires a certain amount of gas to pay for its execution. Contracts run as part of transactions, and one field in the transaction is called the gas price. In order to pay for a contract’s execution, the amount of gas consumed during the execution is multiplied by the gas price, and the resulting product is deducted from the transaction sender’s account.

3. Memory: Memory is used for storing variables and calling functions. No hard limit exists on the amount of memory that may be used, but memory must be paid for with gas. The cost grows quadratically according to the following equation.

\[ C_{\text{mem}}(a) \equiv G_{\text{memory}} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor \]  
(Equation 296 in [24])

\( G_{\text{memory}} \) is a constant defined to be \( G_{\text{memory}} = 3 \), and \( a \) is the amount of 256-bit words used.

4. Storage: Contracts may also store and read data in permanent storage. The SSTORE opcode stores a value to contract storage. This opcode costs 5,000 gas when writing to an existing value and 20,000 gas when writing to a new value. There is no hard limit on contract storage, but it is stored on each Ethereum node and is limited by hard drive capacity. Therefore, to discourage overuse of contract storage, the gas price is very high.

2.2 Ethereum Transactions and Blocks

Transactions are how the outside world communicates with Ethereum. A transaction can either transfer ETH between two accounts, create a smart contract, or call a smart contract. A transaction has a sender and a receiver which are identified by account addresses.
Each account in Ethereum is identified by a 160-bit address, and accounts can either belong to smart contracts or individuals. Smart contract addresses are determined when a contract is created using metadata from the account of the contract creator (the sender of the contract create transaction). Individual accounts use public and private keys, and the account address is the rightmost 160 bits of the SHA-3\(^1\) hash of the public key.

<table>
<thead>
<tr>
<th>Hash</th>
<th>0x29000544fb94145da4806b4506386a2b6775d4d4abca48fec1083646e14cd2f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
<td>4422322</td>
</tr>
<tr>
<td>Time:</td>
<td>10/24/2017 1:55:27 PM (a year ago)</td>
</tr>
<tr>
<td>From</td>
<td>0x44dE2E2B1f378D51E0D2A2A5E5EbE4d427AB955A9</td>
</tr>
<tr>
<td>To</td>
<td>0xddBeA7939F7693d761803320D39eA30D0AC6ea</td>
</tr>
<tr>
<td>Value</td>
<td>0.04132 ETH</td>
</tr>
<tr>
<td>Fee</td>
<td>0.00008 ETH</td>
</tr>
<tr>
<td>Gas Price</td>
<td>4 GWei</td>
</tr>
<tr>
<td>Gas Limit</td>
<td>121,000</td>
</tr>
<tr>
<td>Gas Used</td>
<td>21,000</td>
</tr>
<tr>
<td>Tools</td>
<td>Parity Trace</td>
</tr>
</tbody>
</table>

**Figure 1:** An example transaction, with sender and receiver addresses \(17\). Transactions are uniquely identified by their hash. This is the SHA-3 hash of the serialized transaction (Section 2.3).

**Blocks** in Ethereum are collections of transactions. Blocks contain transactions and a *header*, which contains metadata about the block. One field in the header is the *parent block hash*. Blocks “build” on top of previous blocks by including the SHA-3 hash of the previous block in the block header. This forms a “hash chain” of blocks. The hash chain makes it impossible to modify previous blocks. Modifying a previous block would change the hash of the modified block and all blocks afterward.

\(^1\)Ethereum uses the SHA3-256 hash function, also known as Keccak256 \(18\).
2.3 Recursive-Length-Prefix (RLP) Encoding

Ethereum has a native serialization encoding called Recursive Length Prefix (RLP) \cite{7}. RLP encodes arbitrarily nested lists of binary data. It is similar to other encoding schemes like JSON, but the range of types that are supported are limited to only lists and binary data. RLP is used to serialize blocks, transactions, and other data structures used in Ethereum.

2.4 Merkle-Patricia Trie

Ethereum stores account balances and other account metadata in a data structure called a Merkle-Patricia trie \cite{8}. A Merkle-Patricia trie builds upon both radix tries and Merkle trees \cite{20}. Each node in the trie is hashed by first serializing with RLP, then using the SHA-3 hash function. This builds a key-value map of node hashes to nodes (Figures 3 and 4).
<table>
<thead>
<tr>
<th>Node Hash</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xb45a...3e33</td>
<td>Root Node. Children: [0x3244...f31e, 0x5484...20f1]</td>
</tr>
<tr>
<td>0x3244...f31e</td>
<td>Non-leaf Node. Children: [0xf45e...3789, 0xa38c...f32b]</td>
</tr>
<tr>
<td>0x5484...20f1</td>
<td>Non-leaf Node. Children: [0x78b4...4bc7, 0x983b...ff3e]</td>
</tr>
<tr>
<td>0xf45e...3789</td>
<td>Leaf Node. Balance: 100 ETH</td>
</tr>
<tr>
<td>0xa38c...f32b</td>
<td>Leaf Node. Balance: 10 ETH</td>
</tr>
<tr>
<td>0x78b4...4bc7</td>
<td>Leaf Node. Balance: 12 ETH</td>
</tr>
<tr>
<td>0x983b...ff3e</td>
<td>Leaf Node. Balance: 50 ETH</td>
</tr>
</tbody>
</table>

**Figure 3:** Trie nodes, indexed by their hash.

Nodes connect with their children by using their hash, which is stored inside the node. This makes the hash of a node depend on the hashes of its children, which are dependent on the hashes of the grandchildren, and so on. The hash of the root node can then be used to uniquely identify the entire trie.

**Figure 4:** An example Merkle-Patricia trie.
Figure 5: The same Merkle-Patricia trie from Figure 4 with a Merkle proof for the account with balance 12 ETH. Each node is labeled $N_1$ through $N_7$. The Merkle proof for $N_6$ is the hashes of the orange nodes and the value of the green node: \{h(N_2), h(N_7), N_6\}.

Merkle-Patricia tries also allow one to verify that values exist in the trie using a Merkle proof. A Merkle proof of a value $v$ is the set of sibling nodes along the path from the root node to the node with the value $v$. Merkle proofs allow one to verify that $v$ exists in a Merkle-Patricia trie without needing access to the whole trie. In Figure 5, the Merkle proof for the account with a balance of 12 ETH is shown (node $N_6$). The proof is the set \{h(N_2), h(N_7), N_6\}, where $h$ is the SHA-3 hash function. If we additionally know the root hash of the trie (which I will call $r$), then we can verify that $N_6$ exists in the trie with the following method:

First, find the hash of $N_3$. $N_3$ is a non-leaf node that only contains the hashes of its child nodes, which are $N_6$ and $N_7$.

$$h(N_3) = h([h(N_6), h(N_7)])$$

Then, find the hash of $N_1$. $N_1$, like $N_3$, only contains the hashes of its child nodes, which are $N_2$ and $N_3$.

$$h(N_1) = h([h(N_2), h(N_3)])$$

If $h(N_1)$ is equal to the expected root hash $r$, then we know that $N_6$ must exist in the trie.
Figure 6: The same Merkle-Patricia trie from Figure 5 with a Merkle proof for the account with balance 12 ETH. The Merkle proof for $N_6$ using the alternate definition is the set of nodes $\{N_1, N_3, N_6\}$.

Merkle proofs are secure because finding SHA-3 hash collisions is extremely difficult [18] and no collision has been found thus far. To break the security of Merkle proofs, an adversary needs to find collisions in the SHA-3 hash function to provide attacker-controlled values that hash to the expected hash $r$.

This definition of a Merkle proof is cumbersome to use in practice because sibling nodes need to be enumerated and hashed. An equivalent definition of a Merkle proof in the context of Merkle-Patricia tries is the set of nodes along the path to the value in the trie (Figure 6). This is an equivalent definition because non-leaf branching nodes contain the hashes of the sibling nodes. Therefore, including the entire non-leaf node in the Merkle proof automatically includes all sibling node hashes. This simpler definition of a Merkle proof will be used for the rest of this paper.

Merkle-Patricia tries are used to store the Ethereum world state, as well as contract storage, transactions within a block, and block receipts. The Ethereum world state is the state of all accounts in Ethereum and is uniquely identified by the SHA-3 hash of the root node. Every time an account changes (for example, if the account balance changes), the hash of the root node also changes. In Figure 4, the state root hash is $0xb45a...3e33$, as shown in Figure 3.

Ethereum is a transaction-based state machine, since applying a transaction to a state $S$ produces a new state $S'$. The hash of the root node (also known as the “state root hash”) is stored in the block header, uniquely identifying the state produced after executing the transactions in that block.

2.5 Consensus

Ethereum supports multiple consensus protocols, but the protocol that is most popular is Proof-Of-Work (PoW) [21]. Proof-Of-Work deters abuse by requiring that nodes perform computationally intensive tasks. This consensus protocol requires that nodes send proofs that the computationally intensive work has been done. These proofs are easy to verify but are difficult to create.
In PoW, blocks must be mined before they are added to the hash chain. Nodes which mine blocks are called miners. Any node in Ethereum may choose to be a miner. Miners mine blocks by choosing values for the nonce, a field in a block’s header. A block is successfully mined if the block’s hash satisfies a certain condition. By modifying the nonce field in the block header, serializing the block header with RLP, and finding the SHA-3 hash of the serialized header, we obtain the block hash. Once a nonce that satisfies the PoW condition has been found, the miner advertises the mined block on the network. Other nodes on the network verify that the block has been mined correctly by hashing the block header and checking if it satisfies the PoW condition.

Ethereum mining uses the Ethash algorithm [2]. This algorithm is designed to be difficult to run on application-specific integrated circuits (ASICs) by using a significant amount of memory. Ethash uses a resource that is several gigabytes large and is regenerated every epoch transition, which happens every 30000 blocks. This shared resource is called a DAG (directed acyclic graph) and is only needed to create the Proof-Of-Work. The DAG is not required for Proof-Of-Work verification.

Miners in Ethereum are rewarded by a predetermined block reward and by the transaction fees from the transactions they include in the block. Reward distribution is done by including the account address of the miner in the block’s header. This is called the block’s coinbase. Miners also do not coordinate mining blocks and race against each other to mine the next block in the block chain. Since there is no coordination, it is possible for more than one block to be generated simultaneously. If this happens, then the nodes in the Ethereum network use a protocol to choose one block to add to the main block chain, and the other blocks become uncle blocks. Miners are still rewarded for producing uncle blocks, but the reward is not as high as for producing a normal block.

3 Design

3.1 Motivation

Ethereum currently suffers from scalability problems. As stated in section 1, Ethereum can only support about 15 transactions per second which is too slow for widespread adoption. One reason for the lack of scalability is I/O. The amount of I/O nodes need to perform is significant and slows down transaction processing. Disk I/O has also been a source of denial-of-service (DoS) attacks in Ethereum [15]. Most of these I/O operations come from traversing and updating the Merkle-Patricia trie storing the Ethereum world state. Making matters worse, the way Merkle-Patricia tries are stored on-disk causes pointer-chasing random disk reads. Merkle-Patricia tries are typically stored in key-value stores such as LevelDB or RocksDB, where each value is stored using the SHA-3 hash of the value as the key. This randomizes the location of state trie nodes on-disk.
**Figure 7:** I/O Operations vs. block verification time. There is a positive correlation between the amount of I/O operations performed during block verification and block verification time.

Figure 7 shows that the number of I/O operations is correlated with how long it takes to verify a block. In Figure 8, the number of I/O operations is also increasing with block number. This means that the number of I/O operations will continue to increase in the future, further slowing down block verification time and affecting the scalability of Ethereum.
3.2 Stateless Clients

The Stateless Clients proposal, as the name suggests, changes Ethereum such that Ethereum nodes no longer need to store the Merkle-Patricia trie storing the Ethereum state. The term “client” is a synonym for “node”. The proposal accomplishes this by changing the Ethereum state transition function. The original Ethereum state transition function is the following:

$$STF(S, B) \rightarrow S'$$

Here, $S$ is the initial state, $B$ is an Ethereum block, and $S'$ is the newly updated state created from applying the transactions in $B$ to $S$. 

Figure 8: I/O operations performed during block verification
The Stateless Clients proposal changes the state transition function to $STF'$, which is defined here:

$$STF'(r_S, B, W) \rightarrow r_{S'}$$

Here, $r_S$ is the state root hash for $S$, $B$ is a block, and $r_{S'}$ is the state root hash for $S'$. $W$ is a “witness”, which are Merkle proofs for each value in the state trie the execution of $B$ needs to access. These witnesses are packaged along with blocks and are sent over the network to other Ethereum nodes. Since witnesses contain the state values the execution of $B$ requires, Ethereum nodes do not need to access the state trie when verifying blocks. Not having a state trie greatly reduces the number of I/O operations needed to verify blocks.

The only nodes that need to access the state trie are miners. In vanilla Ethereum, miners create blocks by including transactions and performing the Proof-Of-Work algorithm. The Stateless Clients proposal adds an additional step: generate the witness for the block. Miners need to execute the transactions in the block using their local copy of the Ethereum state trie and keep track of which state values were accessed to obtain Merkle proofs. The set of Merkle proofs is the witness for the block.

### 3.3 Example

In this section, I will show how the Stateless Clients proposal works from block creation to block verification using an example Merkle-Patricia trie and transaction.

#### 3.3.1 Block creation

The initial Merkle-Patricia state trie is shown in Figure 9. This trie contains 22 Ethereum accounts. Four accounts are shown, specifically accounts 0x1354...f78b, 0x298a...e2b1, 0x737b...fc91, 0xa98b...3645.
Figure 9: Initial Merkle-Patricia trie. This trie has 22 Ethereum accounts, shown in the last level of the trie. Four accounts are shown in an expanded view. The state root hash is 0xa34c...2f55. The remaining empty nodes are intermediate Merkle-Patricia trie nodes.

Now suppose account 0x1354...f78b wants to send 10 ETH to account 0x298a...e2b1. This Ethereum transaction is submitted to an Ethereum miner node running the Stateless Clients implementation. The miner then needs to include this transaction in a block; for simplicity, assume that this is the only transaction the miner will put in the block. The miner needs to perform the following steps, in order:

1. Execute the transaction and reward itself with the block reward locally, using the miner’s local copy of the Ethereum state trie.
2. Obtain Merkle proofs for all values in the state trie that were accessed in step 1.
3. Create a block header, and set the state root field to the expected state root of the state trie after executing the transaction and applying the block reward from step 1.
4. Obtain a nonce for the block that satisfies the Proof-Of-Work condition, and set the nonce field of the block header.
5. Send block to other peers.

The miner first needs to execute the transaction with its local copy of the state trie. While it executes the transaction, it keeps track of which accounts in the state trie were accessed. It then constructs Merkle proofs for all accounts that were accessed. After the transaction has been
executed, the miner also needs to calculate the block reward. Recall from Section 2.5 that miners receive a reward for mining blocks. Suppose the miner’s Ethereum address is 0x737b...fc91 and the block reward is 5 ETH. The miner also needs to obtain a witness for adding the block reward to its own address. Figure 10 shows the witnesses generated while creating the block; there are witnesses for accounts 0x1354...f78b, 0x298a...e2b1, and 0x737b...fc91. The state trie after executing the transaction and applying the block reward is shown in Figure 11.

Figure 10: The same Merkle-Patricia trie from Figure 9, but the nodes in green are the Merkle proofs for the accounts accessed.
After the witness is generated, the miner needs to create a block header. This block header contains many fields for block metadata, but most importantly for this example, it contains a field for the expected state root after the execution of the block. This new state root was calculated earlier when executing the transaction and applying the block reward and is shown in Figure 11. Once the block header is created, the miner can serialize and hash it in order to begin the process of obtaining a nonce for this block for Proof-Of-Work consensus. The nonce field in the header is set to the nonce that was found.

After the nonce is obtained, the block is broadcast to other Ethereum nodes. The witnesses are serialized and added to the end of the block, as shown in Figure 12.
3.3.2 Block verification

Verifiers are nodes that do not create blocks and only receive blocks from other peers to verify them. Verifiers are also known as full nodes. In Stateless Clients, verifiers do not store the Ethereum state trie, but they still store the blocks they receive. A verifier node running the Stateless Clients implementation needs to perform the following steps in order to verify blocks:
1. Receive block from peer.

2. Use witness in block to reconstruct the Merkle-Patricia trie.

3. Use the *parent block hash* from the block header to find the previous block in the blockchain.

4. Extract the *state root* field from the header of the previous block. This is the initial state root for the current block.

5. Execute transaction and apply block reward using the reconstructed Merkle-Patricia trie.

Block verification in Stateless Clients starts by first receiving a block with witnesses from a miner (Figure 12). The verifier then reconstructs the Merkle-Patricia state trie using the witnesses in the block. It deserializes the nodes and inserts them into a *HashMap*, where the key for each node is the SHA-3 hash of the node. Recall from Section 2.4 that nodes in a Merkle-Patricia trie link to other nodes using their SHA-3 hash. *HashMaps* make looking up nodes by their hash efficient. Hashing each node also implicitly verifies the witness, since a malformed witness would have incorrect hashes and the trie would not be able to be reconstructed completely.

To determine which node is the root of the trie, the verifier uses the *parent block hash* to look up the block header of the previous block. The previous block’s header contains a *state root* field, which is the state root for the current block. The reconstructed trie is shown in Figure 13.

![Figure 13: The Merkle-Patricia trie reconstructed from the witness in the block.](image-url)

The verifier then executes the transaction and applies the block reward, using the reconstructed state trie to get account balances. If the transaction execution or block execution tries to access a
trie node that is not in the reconstructed trie, then the witness in the block is malformed and the block is not valid. After executing the transaction and the block reward, there is a new state trie with a new state root as shown in Figure 14.

![Updated reconstructed Merkle-Patricia trie](image)

**Figure 14:** The updated reconstructed Merkle-Patricia trie. Note that the account balances and the state root hash have changed. The state root hash matches the expected state root hash in the block header in Figure 12.

### 4 Implementation

I implemented Stateless Clients in Parity, a popular Ethereum client written in the Rust programming language. The implementation was divided into several logical parts.

#### 4.1 State backend

Ethereum state in Parity is accessed through an instance of a `State` struct. Structs in Rust are similar to classes in C++ and Java. `State` handles the following tasks:

1. Create recoverable checkpoints.
2. Create and retrieve smart contracts.
3. Create and retrieve account data.

4. Transfer balances between accounts.

In addition to these tasks, State also maintains an internal cache of the accounts that were requested. Modifications to these accounts are cached and are written to the underlying storage only when the State object is destroyed.

The State struct provides a convenient wrapper around the Ethereum state trie. Other parts of Parity, such as the Ethereum Virtual Machine, use the functions exposed by State to execute transactions. However, the Ethereum state data is not stored inside the State object. State is a generic struct that uses a state backend (a Backend object) to read and write state data.

Backend is a Rust trait. Traits in Rust are similar to interfaces in other languages like Java. Structs that implement the Backend trait need to have the methods defined in Figure 15:

1. Get(key)
2. Insert(value)
3. Emplace(key, value)
4. Contains(key)
5. Remove(key)

**Figure 15:** Required methods for Backend

The “standard” backend that Parity uses is the StateDB backend. StateDB is a struct that contains a pointer to a RocksDB database object and implements each of the required methods by reading and writing from RocksDB. RocksDB is a popular key-value store that Parity uses to store the state trie. The key for each value in the state database is simply the SHA-3 hash of the value.
4.1.1 Witness Creation

In Parity, blocks are created by a Miner object. Since the Miner object executes transactions, it needs access to the Ethereum state, provided by a State object and a Backend object. I replaced the StateDB backend that the Miner uses with a backend I created called WitnessCreate. The WitnessCreate backend intercepts Get(key) and Contains(key) calls and records the values retrieved from the state database. The rest of the methods required for the Backend trait are implemented the same as in StateDB. After all transactions have been executed, the values recorded by WitnessCreate are extracted with a function called extractWitness. This witness is broadcast to the Ethereum network along with the block data once the miner successfully mines the block.

4.1.2 Witness Verification

Block verification in Parity has several phases:

1. verify_block_basic: Checks block integrity, header parameter integrity, and verifies seals for the block and uncles

2. verify_block_unordered: Checks transaction signatures and block nonce
3. **verify_block_family**: Check block information against parent and uncle blocks.

4. **verify_block_external**: Consensus engine-specific validation.

5. **verify_block_final**: Executes the transactions in the block, and checks block information against block execution results. This step makes sure the resulting state root, gas used, log bloom filter, and receipts root match what is recorded in the block header.

I modified the final step of the block verification process to use a `WitnessCheck` state backend instead of `StateDB`. When constructing the `WitnessCheck`, I use the witness that is broadcast with the block (Figure 16). The `WitnessCheck` backend implements the methods in Figure 15 by using this witness. `Insert(value)` and `Emplace(key, value)` are implemented by using an in-memory `HashMap` to record all write operations to the `WitnessCheck` backend. `Get(key)` is implemented by looking for values in the witness that have a SHA-3 hash equal to the key. If no values exist, then the in-memory `HashMap` is checked. This handles the case when a new value is written and later read from during transaction execution. `Contains(key)` is implemented by using the same logic as `Get(key)`, and `Remove(key)` only removes values from the in-memory `HashMap` leaving the witness data untouched. There are no read or write operations to the RocksDB state database when verifying blocks.

### 4.2 Serialization of witnesses

An RLP serialized block has the following layout:

1. **Header**
   - (a) Parent block hash
   - (b) Uncle block hash
   - (c) Block author
   - (d) State root hash
   - (e) Transaction root hash
   - (f) Block receipts root hash
   - (g) Log bloom filter
   - (h) Block difficulty
   - (i) Block number
   - (j) Gas limit
   - (k) Gas used
   - (l) Timestamp
   - (m) Extra data
2. Body

(a) Merkle-Patricia trie of serialized transactions
(b) Merkle-Patricia trie of headers of uncle blocks
(c) Serialized witness

I added an additional field in the Body section of the serialized block to store the serialized block witness. The block witness is a list of database values, which can be serialized easily with RLP. The new serialized block format is used when storing blocks on-disk, and when syncing blocks over the network. Adding a new field in this location maintains backwards compatibility with unmodified versions of Ethereum.

4.3 Block sync protocol

The block sync protocol in Ethereum also needed to be changed. Ethereum nodes use a peer-to-peer (P2P) transport protocol called RLPx \footnote{11} to communicate between nodes. The Ethereum Wire Protocol \footnote{5} runs on top of RLPx and is used to synchronize the following data among nodes:

1. Transactions: When a node receives a transaction, it also broadcasts the same transaction to its peers.

2. Blocks: When a node mines a new block or receives a new block from a peer, it broadcasts the same block to its peers.

3. State: Nodes may also synchronize their state tries by broadcasting Merkle tree nodes.
Protocol 1  Ethereum block synchronization protocol

Inputs. Peer $P_1$ contains blocks $B_K = \{b_1, b_2, b_3, \ldots, b_K\}$, the set of blocks up to block number $K$. Peer $P_2$ contains blocks $B_H = \{b_1, b_2, b_3, \ldots, b_H\}$, the set of blocks up to block number $H$ where $H < K$.

Goal. Peer $P_2$ contains the set of blocks $B_K$.

Steps:

1. $P_1$ and $P_2$ send a STATUS(CURRENTBLOCKNUMBER) message to each other.
2. $P_2$ notices that its latest block number $H$ is less than $P_1$’s latest block number $K$. $P_2$ sends GETBLOCKHEADERS($\{b_{H+1}, \ldots, b_K\}$) to request block headers for blocks $b_{H+1}, \ldots, b_K$ from $P_1$.
3. $P_1$ replies with a BLOCKHEADERS message, containing a list of block headers that $P_2$ requested.
4. $P_2$ sends GETBLOCKBODIES($\{b_{H+1}, \ldots, b_K : b_i \text{ is not an empty block}\}$) to $P_1$ to request block bodies, which are parts of the blocks excluding the header.
5. $P_1$ replies with a BLOCKBODIES message, containing a list of block bodies blocks that $P_2$ requested.
6. $P_2$ matches block bodies with the block headers it received earlier using root hashes of the transaction and uncle block tries. The matched blocks form $b_{H+1}, \ldots, b_K$ and are stored in $P_2$’s block chain.

Figure 17: Ethereum block sync protocol.

The block synchronization protocol flow in unmodified Ethereum is described in Figure 17. First, peers send a STATUS message to each other. The STATUS message notifies all participants of the latest block number present in every node. Nodes that need more blocks can request blocks from the node with a greater latest block number. The key part of the sync protocol is that blocks are split into headers and bodies that are sent separately. First, block headers are requested and sent (step 2). Then, in step 4 block bodies are requested for non-empty blocks. A block is empty if the block body contains no transactions or uncle block headers. Nodes are able to tell if blocks are empty by reading the block header. If a block is empty, then there is no need to request and send an empty body.

This protocol cannot be used for Stateless Clients without modifications. The reason is that no block in Stateless Clients is “empty”, each block at least contains a witness for the block. Even if a block contains no transactions or uncle blocks, it still updates the Ethereum world state because the miner of the block gets rewarded. Rewarding the miner is done by increasing the balance in the miner’s account, and the Ethereum node needs to have a Merkle proof of the current balance.
of the miner’s account.

The modifications that were made to this protocol are the following:

1. In step 4 peers request block bodies for all block headers, even if the block header represents a block with no transactions or uncle blocks.

2. In step 5 peers that send block bodies also send the witness and the hash of the state root for each block body. The witness is not used in the sync protocol but is used to reconstruct each block from its header and block body in step 6. The state root is used to match block bodies with block headers in step 6. The root hashes of the transaction trie and uncle block trie cannot be used for matching because empty blocks share the same hashes for these tries. The state root hash of a block is unique, even for empty blocks, so it may be used to match headers with block bodies.

5 Evaluation

5.1 Correctness

I tested my implementation of Stateless Clients for correctness by sending blocks from one instance of Parity to another. The first instance of Parity downloaded blocks from the real Ethereum blockchain and calculated the witness for the block. It then sent the modified block to the second instance of Parity, which was running with the stateless client modifications.

The stateless client Parity instance executed each block using the provided witness for any operations involving the Ethereum state. The stateless client is only able to verify blocks with the witness correctly if the state root hash after executing the transactions within the block matches the state root hash calculated by an unmodified Parity instance. I was able to verify that all blocks in the Ethereum blockchain, up to block 7.3 million, are correctly verified by the stateless client.

5.2 Results

I tested the Stateless Clients implementation in a variety of experiments. The purpose of these experiments was to determine if Stateless Clients is a viable solution to the scalability problems in Ethereum. All experiments were run on a server with an Intel Xeon CPU E3-1220 v5 at 3.0 GHz, 16 GB of RAM, and a 1.2 TB Intel 750 series NVMe SSD.
5.2.1 Witness Sizes

The first experiment I ran measured witness sizes. I imported the first 7.3 million blocks from the Ethereum blockchain into the modified Parity Ethereum client and generated witnesses for them.

![Average Witness Overhead](image)

**Figure 18:** Average witness size vs. block number

Figure 18 shows the average witness size per block. The witness size is the total amount in bytes of the RLP serialized witness. There is a lot of variability in witness sizes, so the graph shows a rolling average over the last 2000 blocks at each block number. The average witness size is 723 KB over all blocks. Witness size also generally increases with block number. This is because later blocks tend to contain more transactions than earlier blocks, so more state trie nodes are required for block execution. The size of the state trie also grows as more users create Ethereum accounts, further increasing the witness size. Finally, the increasing frequency of smart contracts
that can potentially touch many accounts in a single transaction also contributes to witness size.

The largest witness size was 49.2 MB at block 2306350. The large spikes in the graph from blocks 2.3 to 2.7 million are from a DDoS attack on the Ethereum network in late 2016 [19]. This attack created millions of Ethereum accounts and significantly increased the size of the state trie. As a result, the witness sizes for blocks in this range are also very large, since they contain millions of newly created accounts.

One important detail to note is that the witness size is much larger than the block size excluding the witness. Figure [19] shows the block sizes excluding the witness, which is the same as the block sizes in unmodified Ethereum. Figure [20] shows the overhead of including witnesses as a percentage increase over block size excluding witnesses. Witnesses increase block size by about 40x-140x, and the percentage increases with block number.

![Average Block Size](image)

**Figure 19:** Average block size (excluding witness) vs. block number
5.2.2 Data Directory Size

One closely related statistic to witness size is the directory size of the Parity data directory. Parity keeps all data inside a directory specified on the command line with the `--base-path` argument. This data includes serialized blocks and the state trie, stored in a RocksDB database. I ran a stateful Parity node in this experiment. A stateful Parity node uses the same stateless client code, but also maintains its own state trie just as in unmodified Ethereum. Stateful nodes are useful because they act as translators [15]; they act as a bridge between the existing Ethereum nodes and stateless nodes by generating witnesses for existing blocks and broadcasting them on the stateless node network. The disk space required for stateful nodes is significantly larger than the space required for an unmodified Parity node, as shown in Figure 21.

![Witness Overhead As Percentage](image)

**Figure 20:** Average witness size overhead as percentage increase over block size
Figure 21: On-disk size of stateful client and unmodified client vs. block number.

The disk space required for running a stateful client grows much more quickly than for running an unmodified Parity client. Due to space constraints, I was only able to collect disk usage data for the stateful client only for the first 2.3 million blocks. However, using the data from Figures 18 and 19, we are able to project what the on-disk size would be if the experiment had continued.
The projected data in Figure 22 was obtained by using cumulative sums of block size and witness size, since the stateful client needs to store the block with its witness locally. If this experiment had continued, then the actual disk usage would be slightly less than the projected usage. This is because RocksDB compresses blockchain data. However, even with the projected data, it is clear that the space required to store witnesses for each block makes the Stateless Clients proposal impractical. Currently, a new Ethereum block is mined every 15 seconds [14]. Using the projected rate of disk usage growth, 1.7 MB of data will have to be stored per new block, amounting to 9.8 GB of additional disk usage per day.
5.2.3 Database Operations

One advantage of Stateless Clients is that the number of database I/O operations is greatly reduced. The only database operations that need to be performed when verifying a block are the following:

1. One READ operation to check if the block has already been verified.
2. One READ operation to check the epoch transition (Section 2.5). This operation is not strictly necessary, and with further optimization this may be removed.
3. One READ operation to verify each uncle block.
4. One WRITE operation to store the new block data.

This is a significant reduction in database I/O operations over the unmodified Parity client, which is shown in Figure 8.
Stateless Clients reduces I/O by bundling witnesses into blocks, but this does not necessarily translate to faster block verification times. I tested this by importing the first 2.3 million blocks along with witnesses into an unmodified Parity instance and Parity with the stateless clients implementation. The 2.3 million blocks were serialized with RLP and stored in a file on-disk. Importing blocks in Parity involves reading each block from disk and verifying the block, using the same verification logic as when receiving blocks over the network from other Ethereum nodes. However, importing blocks from a file removes variability due to network conditions. The rate at which blocks were imported is shown in Figure 24.
Although Stateless Clients reduces I/O, the rate at which blocks can be verified is not greatly increased. Parity with the Stateless Clients implementation is only about 10% faster than unmodified Parity in this benchmark. To investigate the cause of such a small increase in throughput, I made another modification to the Stateless Clients implementation and ran the same benchmark again. This modification changes Stateless Clients so that it does not store the witness from the blocks it imports. Stateless Clients with this modification performs 2.5× faster than unmodified Parity in this benchmark. This suggests that storing large witnesses negates some of the performance advantages expected from Stateless Clients. Storing large witnesses involves a lot of I/O in terms of
the amount of bytes, although there are less I/O operations with Stateless Clients (Section 5.2.3). This modification also shows the effect that I/O has on overall performance. Reducing I/O by not storing witnesses and not storing the Ethereum state trie results in a significant performance increase.

The modification to Stateless Clients to not store witnesses is not a solution that would be acceptable in a production-grade implementation. The reason is that stateless nodes receive witnesses over the network from other nodes. If stateless nodes discarded the witnesses they received, they would not be able to send those witnesses to other stateless nodes.

6 Related Work

There are several other potential solutions to scaling blockchains.

6.1 Ethereum 2.0 Sharding

Stateless Clients is part of a larger effort within the Ethereum community to develop Ethereum 2.0. Ethereum 2.0 is a set of specifications for the next version of Ethereum. One of the ideas proposed in Ethereum 2.0 is to shard the blockchain across multiple collections of nodes [9]. Currently, all Ethereum nodes process all transactions. Once a block is mined, it is broadcast to the whole network and every Ethereum node verifies the block independently. The problem with this approach is that the throughput of the entire system is limited by the throughput of a single node.

Ethereum 2.0 Sharding proposes to split the blockchain and state trie into partitions called “shards”. One sharding scheme is to create shards based on account addresses. For example, all account addresses starting with 0x0... would be in one shard, all account addresses starting with 0x1... would be in another shard, and so on until account addresses starting with 0xf.... This scheme creates 16 shards, one for each hexadecimal digit from 0x0 to 0xf. Every Ethereum node belongs to a shard and serves requests for that shard. Within a shard, Ethereum transactions are the same as in the current version of Ethereum. However, transactions that access accounts in different shards need to use a complex protocol involving transaction receipts [4]. Cross-shard communication is therefore relatively costly.

6.2 Hyperledger Fabric

Hyperledger Fabric [13] is a blockchain architecture that improves scalability in the permissioned setting. In contrast to permissionless blockchains like Ethereum, peers in permissioned blockchains are not completely untrusted. Hyperledger Fabric improves the performance of permissioned blockchains by using an execute-order-validate architecture.
1. Execute: Executing transactions and checking its correctness (*endorsing*).

2. Order: Use a consensus protocol to order transactions and batch them into blocks.


There are three components to this architecture: clients, peers, and orderers. In the execute step, clients send transactions to peers that execute them and reply back with the execution results. Then, clients send the transactions along with the execution results to the orderers, which use a consensus protocol to order the transactions and package them into blocks. Finally, peers receive these blocks and validate them.

This is similar to Stateless Clients in that clients do not have access to the blockchain state. Clients submit transactions to peers for execution. This works because in the permissioned setting, the peers are not completely untrusted.

### 7 Future Work

One area that could be explored in the future is different techniques for reducing witness size. In Section 5.2 I showed how the witness sizes make the Stateless Clients proposal impractical to use. Reducing witness sizes would not only make Stateless Clients feasible in terms of network and disk space constraints, but it would also increase block verification throughput (Section 5.2.4) due to smaller witnesses stored on-disk.

There are several techniques for reducing witness size that could be implemented [23]. One technique is to “cache” frequently used Merkle-Patricia trie nodes in verifier nodes. Nodes in the first few rows of the Merkle-Patricia trie are referenced often and can be stored locally at the verifier nodes. Then, witnesses no longer need to include full Merkle proofs of state values; they would only need to include Merkle-Patricia trie nodes that are not cached at the verifiers.

Another area that could be explored is implementing Ethereum 2.0 Sharding (Section 6.1) and determining whether it increases block verification throughput. This would require measuring the throughput of both intra-shard and inter-shard transaction verification to compare against unmodified Ethereum and Stateless Clients. Exploring the execute-order-verify architecture of Hyperledger Fabric and adapting it to Ethereum is also an area of interest.

### 8 Conclusion

In this project, I worked on implementing the Stateless Clients proposal in Parity, a popular Ethereum client. The goal of Stateless Clients is to improve the scalability of Ethereum by reducing I/O operations required to verify blocks. Stateless Clients does this by bundling a witness
along with each block, where the witness contains Merkle proofs of all the values accessed in the Ethereum state trie during the execution of the block. Miners generate witnesses when creating blocks by keeping track of the values accessed during block execution. Verifiers check witnesses for validity and reconstruct Merkle trees in order to verify blocks.

Stateless Clients does accomplish the goal of reducing I/O operations needed to verify blocks, but the extremely large witness overhead in each block makes using Stateless Clients impractical. The witness size can be $40 - 100 \times$ the size of the block without witnesses. The large witness size hurts the performance of Stateless Clients and requires a significant amount of disk space. Reducing witness size and exploring alternate architectures such as Ethereum 2.0 Sharding and the execute-order-verify architecture of Hyperledger Fabric are potential future topics for research.

9 Acknowledgements

I would like to first thank my advisor Prof. Vijay Chidambaram for his guidance and advice over the last two years. His mentorship was extremely valuable not only for this work but for the other projects I have worked on as part of his research group and for developing my skills as a CS researcher. Next, I would like to thank Soujanya Ponnapalli and Aashaka Shah for proposing the initial idea to me, organizing weekly meetings, and helping me understand blockchain concepts. I also want to thank Prof. Chris Rossbach for reviewing a draft of this thesis.

References


[6] Parity ethereum 2.2.11-stable. [https://github.com/paritytech/parity-ethereum/releases/tag/v2.2.11](https://github.com/paritytech/parity-ethereum/releases/tag/v2.2.11)


37


