

RED QUEEN’S SYNC PROTOCOL FOR ETHEREUM

ANDREW ASHIKHMIN & ALEXEY AKHUNOV

ABSTRACT. As of early 2019, it takes new Ethereum full nodes a few hours to synchronise the blockchain state using the eth/63 protocol Buterin et al. [2019]. We propose a new protocol and an algorithm for Ethereum snapshot sync that is faster and more robust in respect of the growing state size. The new protocol is also tailored towards the needs of light clients and allows data storage formats other than the canonical Merkle Patricia trie. Performance results from a model implementation are encouraging.

”A slow sort of country!” said the Queen.
”Now, here, you see, it takes all the
running you can do, to keep in the same
place. If you want to get somewhere else,
you must run at least twice as fast as that!”

Lewis Carroll, Through the Looking-Glass
and What Alice Found There

1. INTRODUCTION

It currently takes new Ethereum full nodes a few hours to synchronise the blockchain state using the eth/63 protocol (Buterin et al. [2019]). Moreover, the growing state size might potentially result in complete sync failure, as described in Akhunov [2019a].

As part of the Ethereum 1x effort, we propose a new sync protocol and algorithm, which we call Red Queen’s. In our sync algorithm seeders reply with data as of their most recent block. That results in an inconsistent trie on the leecher initially (”phase 1”), which is patched later on (”phase 2”). The idea is similar to that of Leaf Sync (Swende [2019]). Other sources of inspiration include BitTorrent, Parity’s Warp Sync (Parity Technologies), and Firehose Sync (Carver and Cloutier [2019]).

Further, we strive to make the protocol work for light clients like Mustekala—see also Buterin et al. [2018].

N.B. In this document, we only discuss snapshot synchronisation rather than synchronisation from the Genesis block.

2. NOTATION

We mostly follow the conventions and notations of the Yellow Paper (Wood [2018]), for instance, \mathbb{Y} denotes the set of nibble sequences. We use the letter π for prefixes of state or storage trie keys $\mathbf{k} \in \mathbb{B}_{32}$,

$$(1) \quad \pi \in \mathbb{Y} \wedge \|\pi\| \leq 64$$

A key matches a prefix if and only if all their first nibbles are the same,

$$(2) \quad \text{MATCH}(\mathbf{k}, \pi) \equiv \forall_{i < \|\pi\|} : \mathbf{k}'[i] = \pi[i]$$

(\mathbf{k}' is a sequence of nibbles, while \mathbf{k} is a sequence of bytes.)

3. PROTOCOL SPECIFICATION

We propose the following new request/reply operative pairs¹:

GetBytecode (0x20)

[reqID: \mathbb{N} , [codeHash₀: \mathbb{B}_{32} , codeHash₁: \mathbb{B}_{32} , ...]]

Request EVM code of smart contracts. The operative is just like **GetNodeData** from the current version (eth/63) of Ethereum Wire Protocol (Buterin et al. [2019]), except:

- (1) includes a request ID;
- (2) will only return bytecode with the corresponding hash, not arbitrary node data.

Bytecode (0x21)

[reqID: \mathbb{N} , [code₀: \mathbb{B} , code₁: \mathbb{B} , ...]]

Reply to **GetBytecode**. Bytecode position in the response list must correspond to the position in the request list; the empty list \emptyset (i.e. []) should be used for omitted bytecodes.

GetStateNodes (0x22)

[reqID: \mathbb{N} , blockHash: \mathbb{B}_{32} , [prefix₀: \mathbb{Y} , prefix₁: \mathbb{Y} , ...]]

Request state trie nodes as of a specific block. Note that this operative is similar to **GetNodeData**, but it uses prefixes rather than hashes as node keys². It will also only return nodes from the state trie, not arbitrary node data. The prefix encoding is described in Appendix C of Wood [2018] (no additional flags are utilised).

StateNodes (0x23)

[reqID: \mathbb{N} , [node₀: \mathbb{B} , node₁: \mathbb{B} , ...], [availableBlock₀: \mathbb{B}_{32} , ...]_{opt}]

Reply to **GetStateNodes**. The empty list \emptyset returned instead of a node means that the peer does not have enough information about the node requested. In that case the peer should return blocks for which requested nodes are available.

GetAccounts (0x24)

[reqID: \mathbb{N} , blockHash: \mathbb{B}_{32} , [prefix₀: \mathbb{Y} , prefix₁: \mathbb{Y} , ...]]

Request state trie leaves (i.e. accounts) as of a specific block from subtrees corresponding to specified prefixes.

Accounts (0x25)

[reqID: \mathbb{N} ,
 [
 [status₀: \mathbb{N} , [[key₀⁰: \mathbb{B}_{32} , val₀⁰: \mathbb{B}], [key₀¹: \mathbb{B}_{32} , val₀¹: \mathbb{B}], ...]_{opt}],
 [status₁: \mathbb{N} , [[key₁⁰: \mathbb{B}_{32} , val₁⁰: \mathbb{B}], [key₁¹: \mathbb{B}_{32} , val₁¹: \mathbb{B}], ...]_{opt}],
 ...
],
 [availableBlock₀: \mathbb{B}_{32} , ...]_{opt}
]

¹For some extra information see Péter Szilágyi's comment at ETH v64 Wire Protocol Ring.

²In a radix trie there is a trivial one-to-one correspondence between nodes and prefixes. Since Ethereum employs a modified radix trie with extension nodes, we define node's prefix as the shortest one such that all leaves descending from the node match the prefix and, conversely, all leaves that match the prefix descend from the node in question.

Reply to **GetAccounts**. Returns accounts that match requested prefixes as key-value pairs³. The peer may only return either all leaves of a subtree or nothing. **status** must take one of the following values:

- 0 – success;
- 1 – no data as of requested block, the peer should return a list of its available blocks;
- 2 – too many leaves matching the prefix.

We propose to support up to 4096 leaves per prefix and return **status** = 2 for bigger subtrees.

Note that state trie replies do not inline storage tries, unlike Leaf Sync.

GetStorageSizes (0x26)

[reqID: \mathbb{N} , blockHash: \mathbb{B}_{32} , [addressHash₀: \mathbb{B}_{32} , addressHash₁: \mathbb{B}_{32} , ...]]

Request storage trie sizes as of a specific block.

StorageSizes (0x27)

[reqID: \mathbb{N} , [numLeaves₀: $\mathbb{N}|\emptyset$, numLeaves₁: $\mathbb{N}|\emptyset$, ...], [availableBlock₀: \mathbb{B}_{32} , ...]_{opt}]

Reply to **GetStorageSizes**. The peer may return the empty list \emptyset instead of the number of leaves for accounts it does not have enough information about. In that case the peer should return blocks for which requested data is available.

GetStorageNodes (0x28)

[reqID: \mathbb{N} , blockHash: \mathbb{B}_{32} ,
 [addressHash⁰: \mathbb{B}_{32} , [prefix₀⁰: \mathbb{Y} , prefix₁⁰: \mathbb{Y} , ...]],
 [addressHash¹: \mathbb{B}_{32} , [prefix₀¹: \mathbb{Y} , prefix₁¹: \mathbb{Y} , ...]],
 ...
]

Request storage trie nodes as of a specific block. Similar to **GetStateNodes**.

StorageNodes (0x29)

[reqID: \mathbb{N} ,
 [
 [node₀⁰: \mathbb{B} , node₁⁰: \mathbb{B} , ...],
 [node₀¹: \mathbb{B} , node₁¹: \mathbb{B} , ...],
 ...
],
 [availableBlock₀: \mathbb{B}_{32} , ...]_{opt}
]

Reply to **GetStorageNodes**. Similar to **StateNodes**.

GetStorageData (0x2a)

[reqID: \mathbb{N} , blockHash: \mathbb{B}_{32} ,
 [addressHash⁰: \mathbb{B}_{32} , [prefix₀⁰: \mathbb{Y} , prefix₁⁰: \mathbb{Y} , ...]],
 [addressHash¹: \mathbb{B}_{32} , [prefix₀¹: \mathbb{Y} , prefix₁¹: \mathbb{Y} , ...]],
 ...
]

Request storage subtree leaves as of a specific block. Similar to **GetAccounts**.

³It is feasible to return suffixes rather than full keys given that prefixes are known, but we deem the performance gain to be insignificant.

StorageData (0x2b)

```

[reqID:  $\mathbb{N}$ ,
 [
  [
    [status00:  $\mathbb{N}$ , [[key000:  $\mathbb{B}_{32}$ , val000:  $\mathbb{B}$ ], [key010:  $\mathbb{B}_{32}$ , val010:  $\mathbb{B}$ ], ...]opt],
    [status10:  $\mathbb{N}$ , [[key100:  $\mathbb{B}_{32}$ , val100:  $\mathbb{B}$ ], [key110:  $\mathbb{B}_{32}$ , val110:  $\mathbb{B}$ ], ...]opt],
    ...
  ],
  [
    [status01:  $\mathbb{N}$ , [[key001:  $\mathbb{B}_{32}$ , val001:  $\mathbb{B}$ ], [key011:  $\mathbb{B}_{32}$ , val011:  $\mathbb{B}$ ], ...]opt],
    [status11:  $\mathbb{N}$ , [[key101:  $\mathbb{B}_{32}$ , val101:  $\mathbb{B}$ ], [key111:  $\mathbb{B}_{32}$ , val111:  $\mathbb{B}$ ], ...]opt],
    ...
  ],
  ...
],
 [availableBlock0:  $\mathbb{B}_{32}$ , ...]opt
]

```

Reply to **GetStorageData**. Returns storage data that match requested prefixes as key-value pairs. The peer may only return either all leaves of a subtree or nothing. **status** values are the same as in **Accounts**.

4. SUGGESTED SYNC ALGORITHM

Here we suggest a possible algorithm for full state and storage snapshot synchronisation using the protocol specified above; light clients are out of scope. We describe a modus operandi where the seeder replies with its most recent data, and the leecher has to handle trie data coming from different blocks. We suggest to perform synchronisation in two stages: during phase 1 the leecher obtains leaf data (with the necessary proof nodes) as of any reasonable block height, while during phase 2 it patches up the trie in order to catch up to the most recent block⁴. The idea was proposed in Swende [2019].

Let us focus on the state trie for the moment; we shall come back to storage sync later. For phase 1 we suggest sending **GetAccounts** requests with a single prefix per request, ditto for phase 2. All requested prefixes are of size d_1 during phase 1 and of d_2 during phase 2, $d_2 \geq d_1$. We elaborate on the values of d_1 and d_2 later. The leecher gradually builds the first upper d_2 levels of the Merkle Patricia trie⁵. (The full trie can be constructed if so desired, but only the upper d_2 levels are necessary for our algorithm.) Populated nodes are marked with the block number as of they are valid. The algorithm preserves the following invariant: parent's block is always no older than child's block.

During phase 1 the leecher requests each possible prefix of size d_1 exactly once (barring network failures and faulty peers). When sending a request, the leecher sets its **blockHash** to the latest block. For simplicity's sake, we assume that all blocks are strictly ordered and ignore chain re-orgs. Having received a reply, the leecher might choose to verify received leaves by asking for the intermediate nodes (**GetStateNodes**) to the subtree root. Then the leecher writes the leaves to the database and updates the nodes along the prefix/path. By the end of phase 1, the leecher will have all accounts populated, albeit inconsistently.

Figure 1 shows an example of a phase 1 step with $d_1 = 3$ and $d_2 = 4$. Say the leecher is interested in prefix $\langle 7a3 \rangle$. The trie on the left represents leecher's state before sending a request. Root's block is 5, so the leecher sets **blockHash** accordingly. Suppose that the seeder replies that it only has data as of a newer block #6. The leecher sends **GetAccounts** again with updated **blockHash**. Then it saves received leaves to its database and updates the nodes ($\langle \rangle$, $\langle 7 \rangle$, $\langle 7a \rangle$, $\langle 7a3 \rangle$). The result is displayed on the right of Figure 1.

⁴The Red Queen's race is a nice metaphor for phase 2.

⁵ d_2 is small enough so that we can reasonably assume that (almost) all nodes in question are branch nodes; see Akhunov [2019b].

The overhead of the sync algorithm during phase 1, compared with Parity’s warp sync, is in the proof nodes sent alongside the leaf data. The overhead grows with d_1 , so we want the trie depth to be as low as possible. On the other hand, small d_1 implies a large number of leaves per reply, which can be brittle or inefficient. Thus we set d_1 to the smallest value possible such that the replies are, on average, no larger than a certain size (say 32 KiB). We denote that maximum size as m . During phase 1 a **Accounts** reply contains at most d_1 nodes and its average number of leaves is $\frac{t}{16^{d_1}}$, which gives us

$$(4) \quad d_1 n + \frac{t}{16^{d_1}} l \leq m$$

For the state trie the limit of 32 KiB yields $d_1 = 5$.

Let $C(d, \delta)$ be the maximum number of trie nodes from the upper d levels of a trie that can change (on average) per block⁷. At each level at most δ nodes can change, subject to δ being smaller than the number of nodes at the level. Thus

$$(5) \quad C(d, \delta) = \sum_{i=0}^{d-1} \min(16^i, \delta)$$

If $16^2 \leq \delta \leq 16^3$ and $d \geq 3$, then

$$(6) \quad C(d, \delta) = C'(d, \delta) \stackrel{\text{def}}{=} \delta(d - 3) + 273$$

We now analyse the minimum bandwidth required for the algorithm to converge during phase 2. At the very least, “to keep in the same place”, we need to sync all changes per 1 block no slower than the block time τ . As previously described, the algorithm updates d_2 upper levels of the trie. So the upper bound on the number of nodes to be refreshed is $C(d_2, \delta)$. The number of subtrees that need to be refreshed is no more than δ ; each subtree has $\frac{t}{16^{d_2}}$ leaves on average. Summing up, the total reply size per 1 block necessary not to lag behind is less than

$$(7) \quad \text{RQS} \stackrel{\text{def}}{=} C(d_2, \delta) n + \delta \frac{t}{16^{d_2}} l$$

(RQS stands for Red Queen’s Size). Though it is an upper bound, for our purposes RQS is close enough to the actual value. Differentiating, we find the value of d_2 that minimises RQS

$$(8) \quad d_2^* = \frac{1}{\ln 16} \ln \left(\frac{t l \ln 16}{n} \right)$$

(Obviously, one has to round d_2^* up or down.) For the state trie the optimal $d_2^* = 6$ and the entailing RQS is about 0.7 MiB. Reiterating, the convergence condition for the state trie alone is

$$(9) \quad b > \frac{\text{RQS}}{\tau}$$

For the Ethereum main net as of February 2019 this critical minimum bandwidth is about 0.4 Mbit/s. Table 1 shows performance results of an emulation of the sync protocol for various state trie sizes. The modelling code used is hosted at <https://github.com/yperbasis/silkworm>. Network latency was ignored in the emulation.

Bandwidth	10M	50M	100M
1 Mbit/s	03:39	18:44	39:04
10 Mbit/s	00:20	01:39	03:17
100 Mbit/s	00:02	00:10	00:20

TABLE 1. Emulated times of state trie sync for 10M, 50M, and 100M dust accounts.

Convergence analysis for large storage tries (e.g. CryptoKitties) is similar to the state trie analysis above.

⁷To be more precise mathematically, $C(d, \delta)$ is an upper bound on the expected value.

6. CONCLUSION

TODO: conclusion.

REFERENCES

- Alexey Akhunov. Looking back at the Ethereum 1x workshop 26–28.01.2019 (part 1), January 2019a. URL <https://medium.com/@akhounov/looking-back-at-the-ethereum-1x-workshop-26-28-01-2019-part-1-70c1ebd93266>.
- Alexey Akhunov. Looking back at the Ethereum 1x workshop 26–28.01.2019 (part 2), February 2019b. URL <https://medium.com/@akhounov/looking-back-at-the-ethereum-1x-workshop-26-28-01-2019-part-2-d3d8fdcede10>.
- Vitalik Buterin et al. Light Client Protocol, 2018. URL <https://github.com/ethereum/wiki/wiki/Light-client-protocol>.
- Vitalik Buterin, Gavin Wood, et al. Ethereum Wire Protocol (ETH), 2019. URL <https://github.com/ethereum/devp2p/blob/master/caps/eth.md>.
- Jason Carver and Brian Cloutier. Firehose Sync v0.2, April 2019. URL https://notes.ethereum.org/0v_W4E8lR0azqYymPAF7Ew.
- Parity Technologies. Warp Sync Snapshot Format. URL <https://wiki.parity.io/Warp-Sync-Snapshot-Format>.
- Martin Holst Swende. Leaf sync, March 2019. URL https://notes.ethereum.org/kphcc_CKT4a5sUs_zWVe1A.
- Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, December 2018. URL <https://github.com/ethereum/yellowpaper>.