# 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 $\pi$ for prefixes of state or storage trie keys $\mathbf{k} \in \mathbb{B}_{32}$,

$$\pi \in \mathbb{Y} \ \wedge \ ||\pi|| \leq 64 \tag{1}$$

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

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

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

*Date*: April 2019.

## 3. Protocol Specification

We propose the following new request/reply operative pairs[1]:

**GetBytecode** (0x20)
[reqID: $\mathbb{N}$, [codeHash$_0$: $\mathbb{B}_{32}$, codeHash$_1$: $\mathbb{B}_{32}$, ...]]

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: $\mathbb{N}$, [code$_0$: $\mathbb{B}$, code$_1$: $\mathbb{B}$, ...]]

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

**GetStateNodes** (0x22)
[reqID: $\mathbb{N}$, blockHash: $\mathbb{B}_{32}$, [prefix$_0$: $\mathbb{Y}$, prefix$_1$: $\mathbb{Y}$, ...]]

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 keys[2]. 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: $\mathbb{N}$, [node$_0$: $\mathbb{B}$, node$_1$: $\mathbb{B}$, ...], [availableBlock$_0$: $\mathbb{B}_{32}$, ...]$_{opt}$]

Reply to `GetStateNodes`. The empty list $\varnothing$ 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: $\mathbb{B}_{32}$, [prefix$_0$: $\mathbb{Y}$, prefix$_1$: $\mathbb{Y}$, ...]]

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

**Accounts** (0x25)
[reqID: $\mathbb{N}$,
  [
    [status$_0$: $\mathbb{N}$, [[key$_0^0$: $\mathbb{B}_{32}$, val$_0^0$: $\mathbb{B}$], [key$_0^1$: $\mathbb{B}_{32}$, val$_0^1$: $\mathbb{B}$], ...]$_{opt}$],
    [status$_1$: $\mathbb{N}$, [[key$_1^0$: $\mathbb{B}_{32}$, val$_1^0$: $\mathbb{B}$], [key$_1^1$: $\mathbb{B}_{32}$, val$_1^1$: $\mathbb{B}$], ...]$_{opt}$],
    ...
  ],
  [availableBlock$_0$: $\mathbb{B}_{32}$, ...]$_{opt}$
]

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

[2]In 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 pairs[3]. The peer may only return either all leaves of a subtrie 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 subtries.

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

**GetStorageSizes** (0x26)
[reqID: $\mathbb{N}$, blockHash: $\mathbb{B}_{32}$, [addressHash$_0$: $\mathbb{B}_{32}$, addressHash$_1$: $\mathbb{B}_{32}$, ...]]

Request storage trie sizes as of a specific block.

**StorageSizes** (0x27)
[reqID: $\mathbb{N}$, [numLeaves$_0$: $\mathbb{N}|\varnothing$, numLeaves$_1$: $\mathbb{N}|\varnothing$, ...], [availableBlock$_0$: $\mathbb{B}_{32}$, ...]$_{opt}$]

Reply to `GetStorageSizes`. The peer may return the empty list $\varnothing$ 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: $\mathbb{B}_{32}$,
  [addressHash$^0$: $\mathbb{B}_{32}$, [prefix$^0_0$: $\mathbb{Y}$, prefix$^0_1$: $\mathbb{Y}$, ...]],
  [addressHash$^1$: $\mathbb{B}_{32}$, [prefix$^1_0$: $\mathbb{Y}$, prefix$^1_1$: $\mathbb{Y}$, ...]],
  ...
]

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

**StorageNodes** (0x29)
[reqID: $\mathbb{N}$,
  [
    [node$^0_0$: $\mathbb{B}$, node$^0_1$: $\mathbb{B}$, ...],
    [node$^1_0$: $\mathbb{B}$, node$^1_1$: $\mathbb{B}$, ...],
    ...
  ],
  [availableBlock$_0$: $\mathbb{B}_{32}$, ...]$_{opt}$
]

Reply to `GetStorageNodes`. Similar to `StateNodes`.

**GetStorageData** (0x2a)
[reqID: $\mathbb{N}$, blockHash: $\mathbb{B}_{32}$,
  [addressHash$^0$: $\mathbb{B}_{32}$, [prefix$^0_0$: $\mathbb{Y}$, prefix$^0_1$: $\mathbb{Y}$, ...]],
  [addressHash$^1$: $\mathbb{B}_{32}$, [prefix$^1_0$: $\mathbb{Y}$, prefix$^1_1$: $\mathbb{Y}$, ...]],
  ...
]

Request storage subtrie leaves as of a specific block. Similar to `GetAccounts`.

[3]It 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: $\mathbb{N}$,
  [
    [
      [status$^0_0$: $\mathbb{N}$, [[key$^0_{00}$: $\mathbb{B}_{32}$, val$^0_{00}$: $\mathbb{B}$], [key$^0_{01}$: $\mathbb{B}_{32}$, val$^0_{01}$: $\mathbb{B}$], ...]$_{opt}$],
      [status$^0_1$: $\mathbb{N}$, [[key$^0_{10}$: $\mathbb{B}_{32}$, val$^0_{10}$: $\mathbb{B}$], [key$^0_{11}$: $\mathbb{B}_{32}$, val$^0_{11}$: $\mathbb{B}$], ...]$_{opt}$],
      ...
    ],
    [
      [status$^1_0$: $\mathbb{N}$, [[key$^1_{00}$: $\mathbb{B}_{32}$, val$^1_{00}$: $\mathbb{B}$], [key$^1_{01}$: $\mathbb{B}_{32}$, val$^1_{01}$: $\mathbb{B}$], ...]$_{opt}$],
      [status$^1_1$: $\mathbb{N}$, [[key$^1_{10}$: $\mathbb{B}_{32}$, val$^1_{10}$: $\mathbb{B}$], [key$^1_{11}$: $\mathbb{B}_{32}$, val$^1_{11}$: $\mathbb{B}$], ...]$_{opt}$],
      ...
    ],
    ...
  ],
  [availableBlock$_0$: $\mathbb{B}_{32}$, ...]$_{opt}$
]

Reply to `GetStorageData`. Returns storage data that match requested prefixes as key-value pairs. The peer may only return either all leaves of a subtrie 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 block[4]. 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 trie[5]. (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 subtrie 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 <7a3>. 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 (<>, <7>, <7a>, <7a3>). The result is displayed on the right of Figure 1.

---

[4]The 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].

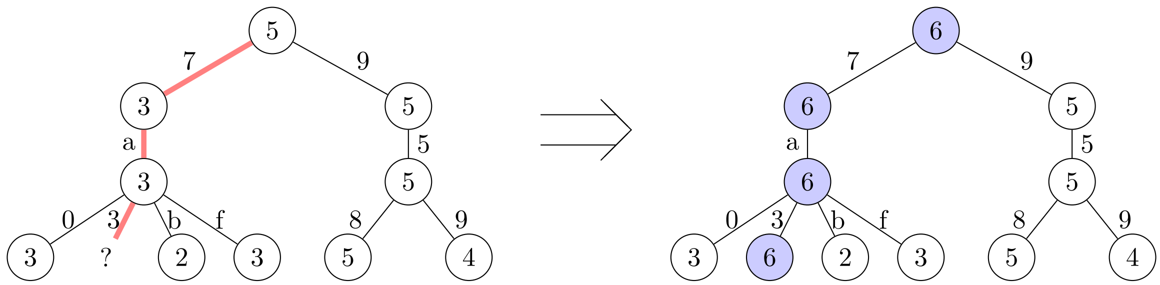


Figure 1. Illustration of phase 1 sync. Nodes are labelled with block numbers, and edges are labelled with nibbles.

At the beginning of phase 2, the leecher updates the trie in order to figure out which subtries have to be refreshed. For that, it uses the `GetStateNodes` operative. The leecher refreshes the trie level by level, starting from the root (level 0) and descending to the level $d_2 - 1$. Nodes at the same level may be requested in batch. Having refreshed nodes for a level, it knows which child nodes one level below have to be refreshed since the leecher can compare their hashes it currently holds with received fresh data. Therefore, only the nodes that have actually changed need to be requested. The leecher might have to restart the node refresh process from the root if new blocks are mined in between; however, provided a certain network bandwidth, the process converges. We analyse convergence conditions in the next section.

When all trie levels from the root down to the level $d_2 - 1$ are up to date, the leecher knows which nibbles at that last level have changed since phase 1. It refreshes the leaves corresponding to such nibbles using `GetAccounts` requests. That concludes the algorithm for state trie synchronisation.

The algorithm for storage sync is similar for large storage tries. Its parameters $d_1$ and $d_2$ are optimised based on the storage size as described in the next section. Small storage tries can be obtained in bulk requesting the empty prefix (meaning the entire trie) for a number of them in one go. `GetStorageSizes` provides a means of finding out storage sizes.

## 5. Performance Analysis

In this analysis, we assume that all tries are well balanced. We also assume that all top nodes up to a certain trie level $i$ are branch nodes, not leaf nor extension nodes. This is a reasonable assumption if $i$ is not too big—see Akhunov [2019b]. Also, we simplify the byte size function of the replies[6], ignoring overheads caused by auxiliary data such as `reqID`, RLP encoding, and the network layer. Let us introduce some notation:

$n$ – the average node size in bytes. $n$ is essentially equal to the size of a branch node as most nodes transferred are branch nodes. $n = 530$ is a reasonable estimate.

$l$ – the average leaf size in bytes, counting both key and value. For the state trie it is the average account size plus the size of its hash key, resulting in about 115 bytes. For storage tries $l \approx 50$.

$t$ – total number of leaves in a trie. For the state trie it is the number of accounts, which is about $53 \cdot 10^6$ as of February 2019—see Akhunov [2019b].

$b$ – the network bandwidth available to the leecher.

$\tau$ – the block time, currently 15 seconds.

$\delta$ – the average number of leaf changes per block for a trie. For the state trie it is in the ballpark of 300.

$||R_n||$ – the number of nodes in a reply $R$.

$||R_l||$ – the number of leaves in a reply $R$.

We use the following simplified formula for the byte size of a reply $R$

$$S(R) = ||R_n||n + ||R_l||l \tag{3}$$

[6]Total request size is much smaller than total reply size, so we ignore requests as well.

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

$$d_1 n + \frac{t}{16^{d_1}} l \leq m \tag{4}$$

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 block[7]. At each level at most $\delta$ nodes can change, subject to $\delta$ being smaller than the number of nodes at the level. Thus

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

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

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

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 $\tau$. 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 subtries that need to be refreshed is no more than $\delta$; each subtrie 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

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

(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`

$$d_2^* = \frac{1}{\ln 16} \ln \left( \frac{tl \ln 16}{n} \right) \tag{8}$$

(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

$$b > \frac{\mathtt{RQS}}{\tau} \tag{9}$$

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.

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

## 6. Conclusion

Proposed Red Queen's protocol and algorithm can alleviate slow synchronisation times caused by the large size of the Ethereum blockchain state. The protocol is also flexible enough to cater for light clients, not only full nodes. Our emulation results are promising and corroborate protocol's good scalability and low overhead. A natural next step forward is to fully implement Red Queen's in a proper Ethereum client and fledge it into a next version of the standard Ethereum Wire Protocol.

## 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_W4E8lROazqYymPAF7Ew.

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_zWVelA.

Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, December 2018. URL https://github.com/ethereum/yellowpaper.