Red Queen's Sync Protocol for Ethereum

Andrew Ashikhmin & Alexey Akhunov

Ethereum Core Dev Meeting
April 2019, Berlin

Motivation

- Current eth/63 snapshot sync (fast sync) is slow (a few hours for full nodes).
- Growing state size can potentially lead to sync failures due to history pruning.
- Parity's warp sync is vulnerable to a "rubbish data" attack.
- A number of similar proposals are emerging: Leaf Sync, Firehose.
- We also propose a sync algorithm with an in-depth performance and convergence analysis.
- The new protocol caters for light clients (e.g. Mustekala).
- Grand vision: swarms of light Ethereum nodes running on mobile, not unlike BitTorrent and Pied Piper :)

Protocol 1/6

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`, 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; an empty list should be used for omitted bytecodes.

Protocol 2/6

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. It will also only return nodes from the state trie, not arbitrary node data.

StateNodes (0x23)
[reqID: $\mathbb{N}$, [node$_0$: $\mathbb{B}$, node$_1$: $\mathbb{B}$, ...], [availableBlock$_0$: $\mathbb{B}_{32}$, ...]$_{opt}$]

Reply to `GetStateNodes`. An empty list 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 (as a list of `availableBlock` hashes).

Protocol 3/6

GetAccounts (0x24)

[reqID: $\mathbb{N}$, blockHash: $\mathbb{B}_{32}$, [prefix$_0$: $\mathbb{Y}$, prefix$_1$: $\mathbb{Y}$, ...]]

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$_0$: $\mathbb{B}_{32}$, ...]$_{opt}$
]

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 the requested block;
- 2 – too many leaves matching the prefix.

Protocol 4/6

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 an 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.

Protocol 5/6

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$^1_1$: $\mathbb{Y}$, ...]],
  ...
]

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$_0$: $\mathbb{B}_{32}$, ...]$_{opt}$
]

Protocol 6/6

GetStorageData (0x2a)

$[\text{reqID}: \mathbb{N}, \text{blockHash}: \mathbb{B}_{32},$
$\quad[\text{addressHash}^0: \mathbb{B}_{32}, [\text{prefix}_0^0: \mathbb{Y}, \text{prefix}_1^0: \mathbb{Y}, \ldots]],$
$\quad[\text{addressHash}^1: \mathbb{B}_{32}, [\text{prefix}_0^1: \mathbb{Y}, \text{prefix}_1^1: \mathbb{Y}, \ldots]],$
$\quad\ldots$
$]$

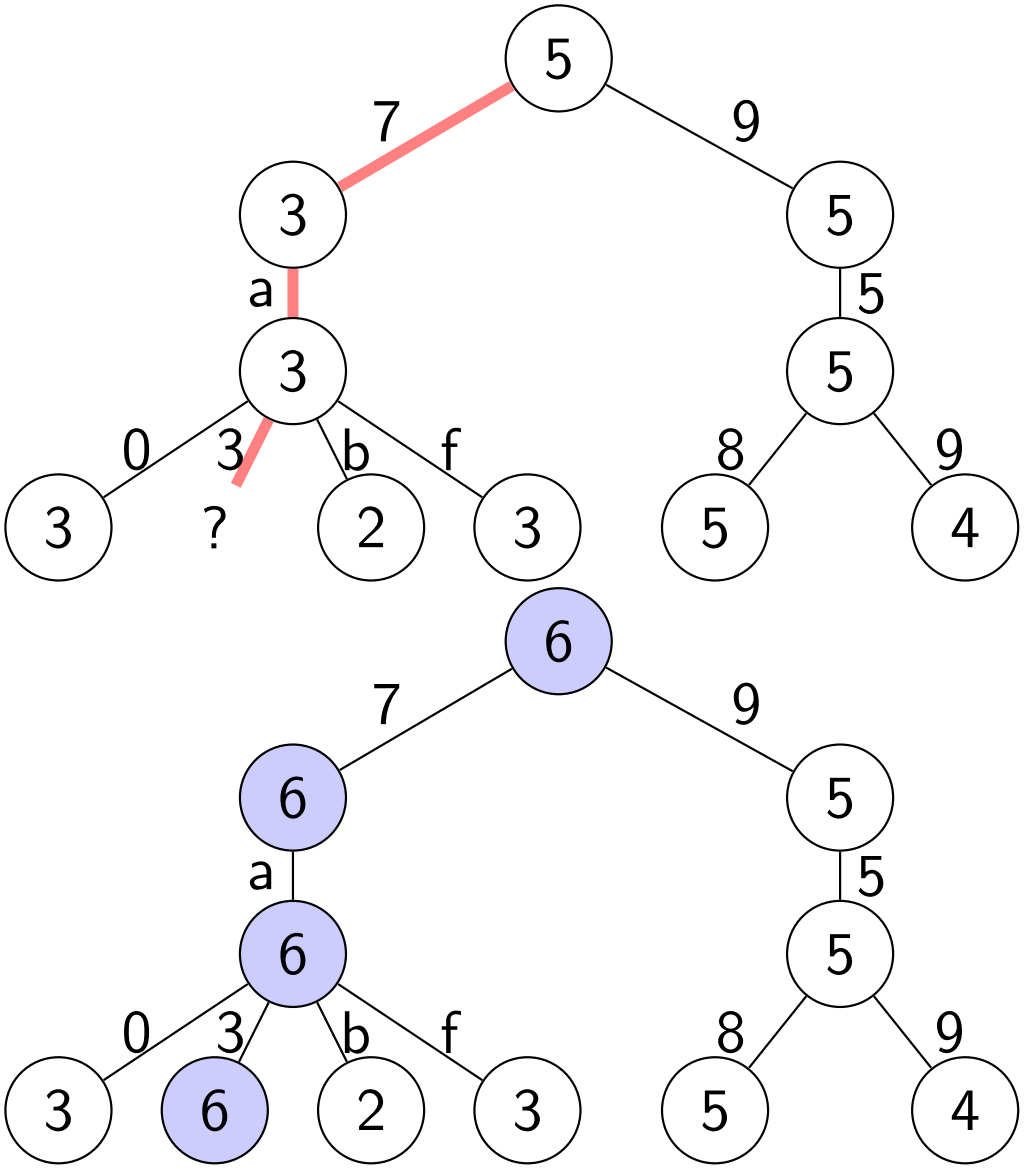
StorageData (0x2b)

$[\text{reqID}: \mathbb{N},$
$\quad[$
$\quad\quad[$
$\quad\quad\quad[\text{status}_0^0: \mathbb{N}, [[\text{key}_{00}^0: \mathbb{B}_{32}, \text{val}_{00}^0: \mathbb{B}], \ldots]_{opt}],$
$\quad\quad\quad[\text{status}_1^0: \mathbb{N}, [[\text{key}_{10}^0: \mathbb{B}_{32}, \text{val}_{10}^0: \mathbb{B}], \ldots]_{opt}],$
$\quad\quad\quad\ldots$
$\quad\quad],$
$\quad\quad\ldots$
$\quad], [\text{availableBlock}_0: \mathbb{B}_{32}, \ldots]_{opt}$
$]$

Suggested Sync Algorithm 1/2

- Always request data as of the most recent block.
- Phase 1: get all leaves, probably as of different blocks.
- Simultaneously request necessary intermediate nodes to verify the leaves.
- Keep the top of the trie in memory.
- Phase 2: patch up the trie.
- Descend from the root level by level to find out stale nodes and subtries.

Suggested Sync Algorithm 2/2



Performance Analysis 1/3

- d_1 – trie depth durung phase 1.
- l – the average leaf size in bytes, counting both key and value. For the state tri $l \approx 115$. For storage tries $l \approx 42$.
- 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.
- m – maximum reply size (approximately), say 32 KiB.

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. On the other hand, small d_1 implies a large number of leaves per reply, which can be brittle or inefficient. During phase 1 an `Accounts` reply contains $\frac{t}{16^{d_1}}$ leaves on average, which gives us

$$\frac{tl}{16^{d_1}} \leq m \tag{1}$$

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

Performance Analysis 2/3

- d_2 – trie depth durung phase 2.
- n – the average node size in bytes. Essentially equal to the size of a branch node as most nodes transferred are branch nodes. $n \approx 530$.
- δ – the average number of leaf changes per block for a trie. For the state trie it is in the ballpark of 300.

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.
The total reply size per 1 block necessary not to lag behind is no more than "Red Queen's Size"

$$\texttt{RQS} \stackrel{\text{def}}{=} C(d_2, \delta)\, n + \delta\, \frac{tl}{16^{d_2}} \tag{2}$$

Performance Analysis 3/3

- b – the network bandwidth available to the leecher.
- τ – the block time, currently 15 seconds.

The value of d_2 that minimises $\texttt{RQS}$

$$d_2^* = \frac{1}{\ln 16} \ln \left(\frac{t/ \ln 16}{n} \right) \tag{3}$$

For the state trie the optimal $d_2^* = 6$ and the entailing $\texttt{RQS}$ is about 0.7 MiB.
The convergence condition for the state trie alone is

$$b > \frac{\texttt{RQS}}{\tau} \tag{4}$$

For the Ethereum main net as of February 2019 this critical minimum bandwidth is about 0.4 Mbit/s.

Emulation Performance Results

STATE TRIE SYNC ONLY

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

Modelling code and a more formal paper are hosted at https://github.com/yperbasis/silkworm.