RED QUEEN'S SYNC PROTOCOL FOR ETHEREUM

ANDREW ASHIKHMIN & ALEXEY AKHUNOV

Abstract. TODO: abstract.

"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

In Red Queen's Synchronisation Protocol for Ethereum 1x seeders reply with data as of their most recent block. That results in an inconsistent trie on the leecher initially ("phase 1"), which we patch later ("phase 2"). The idea is similar to that of Leaf Sync (Swende [2019]).

TODO: mention the sync failure problem Akhunov [2019a] and the needs of light clients like Mustekala. Inspirations like BitTorrent, Parity's warp sync, Firehose Sync, Light Client Protocol.

N.B. Snapshot synchronisation rather than from genesis.

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 $\mathbf{k} \in \mathbb{B}_{32}$,

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

A key matches a prefix iff 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.)

3. Protocol Specification

We propose the following 3 request/reply operative pairs.

GetStorageSizes [+0x20, reqID: $\mathbb{N}$, blockAtLeast: $\mathbb{N}$, [account^0: $\mathbb{B}_{32}$, account^1: $\mathbb{B}_{32}$, ...]] Request storage trie sizes as of block #`blockAtLeast` or newer. Hashes of accounts addresses are used as keys.

StorageSizes [+0x21, reqID: $\mathbb{N}$, blockNumber: $\mathbb{N}$, [numLeaves^0: $\mathbb{N}|\varnothing$, numLeaves^1: $\mathbb{N}|\varnothing$, ...]] Reply to `GetStorageSizes`. Returns storage trie sizes as of block #`blockNumber` $\geq$ `blockAtLeast`. The elements returned must strictly match the accounts requested. The peer may return the empty list $\varnothing$ instead of the number of leaves for accounts it does not have enough information about.

GetNodeData2 [+0x22, reqID: $\mathbb{N}$, blockAtLeast: $\mathbb{N}$, [account^0: $\mathbb{B}_{32}|\varnothing$, prefix$^0_0$: $\mathbb{Y}$, prefix$^0_1$: $\mathbb{Y}$, ...], [account^1: $\mathbb{B}_{32}|\varnothing$, prefix$^1_0$: $\mathbb{Y}$, prefix$^1_1$: $\mathbb{Y}$, ...], ...] Request state or storage trie nodes as of block #`blockAtLeast` or newer. The empty list $\varnothing$ instead of the account hash signifies the state (rather than storage) trie. Note that this operative is similar to `GetNodeData` from Ethereum Wire Protocol

Date: March 2019.

PV63, but it uses prefixes rather than hashes as node keys[1]. TODO: prefix–node correspondence is trivial for branch nodes, not so much for leaf or extension nodes. TODO: prefix encoding consistent with the Yellow Paper.

NodeData2 [+0x23, reqID: $\mathbb{N}$, blockNumber: $\mathbb{N}$, [node$_0^0$: $\mathbb{B}$, node$_1^0$: $\mathbb{B}$, ...], [node$_0^1$: $\mathbb{B}$, node$_1^1$: $\mathbb{B}$, ...], ...] Reply to `GetNodeData2`. Returns trie nodes as of block #`blockNumber` $\geq$ `blockAtLeast`. The nodes returned must strictly match the prefixes requested. The empty list $\varnothing$ returned instead of a node means that the peer does not have enough information about the node requested.

GetSubtries [+0x24, reqID: $\mathbb{N}$, blockAtLeast: $\mathbb{N}$,
[account^0: $\mathbb{B}_{32}|\varnothing$,
[prefix$_0^0$: $\mathbb{Y}$, fromLevel$_0^0$: $\mathbb{N}$, subtrieHash$_0^0$: $\mathbb{B}_{32}|\varnothing$],
[prefix$_1^0$: $\mathbb{Y}$, fromLevel$_1^0$: $\mathbb{N}$, subtrieHash$_1^0$: $\mathbb{B}_{32}|\varnothing$],
...
],
[account^1: $\mathbb{B}_{32}|\varnothing$,
[prefix$_0^1$: $\mathbb{Y}$, fromLevel$_0^1$: $\mathbb{N}$, subtrieHash$_0^1$: $\mathbb{B}_{32}|\varnothing$],
[prefix$_1^1$: $\mathbb{Y}$, fromLevel$_1^1$: $\mathbb{N}$, subtrieHash$_1^1$: $\mathbb{B}_{32}|\varnothing$],
...
],
...
] Request state or storage subtrie leaves along with proof nodes as of block #`blockAtLeast` or newer. The empty list $\varnothing$ instead of the account hash signifies state rather than storage trie. `fromLevel` specifies the number of upper nodes to be excluded from the proof in case the chain has not moved ahead (reply block is not newer). `subtrieHash` is a means to avoid resending leaves that have not changed.

Subtries [+0x25, reqID: $\mathbb{N}$, blockNumber: $\mathbb{N}$,
[
[[node$_{00}^0$: $\mathbb{B}$, node$_{01}^0$: $\mathbb{B}$, ...], tooManyLeaves$_0^0$, [key$_{00}^0$: $\mathbb{B}_{32}$, val$_{00}^0$: $\mathbb{B}$, key$_{01}^0$: $\mathbb{B}_{32}$, val$_{01}^0$: $\mathbb{B}$, ...]$_{opt}$],
[[node$_{10}^0$: $\mathbb{B}$, node$_{11}^0$: $\mathbb{B}$, ...], tooManyLeaves$_1^0$, [key$_{10}^0$: $\mathbb{B}_{32}$, val$_{10}^0$: $\mathbb{B}$, key$_{11}^0$: $\mathbb{B}_{32}$, val$_{11}^0$: $\mathbb{B}$, ...]$_{opt}$],
...
],
[
[[node$_{00}^1$: $\mathbb{B}$, node$_{01}^1$: $\mathbb{B}$, ...], tooManyLeaves$_0^1$, [key$_{00}^1$: $\mathbb{B}_{32}$, val$_{00}^1$: $\mathbb{B}$, key$_{01}^1$: $\mathbb{B}_{32}$, val$_{01}^1$: $\mathbb{B}$, ...]$_{opt}$],
[[node$_{10}^1$: $\mathbb{B}$, node$_{11}^1$: $\mathbb{B}$, ...], tooManyLeaves$_1^1$, [key$_{10}^1$: $\mathbb{B}_{32}$, val$_{10}^1$: $\mathbb{B}$, key$_{11}^1$: $\mathbb{B}_{32}$, val$_{11}^1$: $\mathbb{B}$, ...]$_{opt}$],
...
],
...
] Reply to `GetSubtries`. Returns subtrie leaves with proofs as of block #`blockNumber` $\geq$ `blockAtLeast`. The subtries returned must strictly match the prefixes requested. If the peer does not have information regarding a particular subtrie, it should return the empty list $\varnothing$ (e.g. []) rather than [nodes, tooManyLeaves, leaves] for it. The nodes returned are the upper nodes of the trie down to the subtrie root, so that it is possible to verify that the leaves do belong to the Merkle Patricia trie in question. The first `fromLevel` upper nodes must be skipped if and only if `blockNumber` = `blockAtLeast`. (If `fromLevel` = 0, then the nodes must start with the root node.) `tooManyLeaves` is a boolean flag ($0 = false$, $1 = true$) indicating that the subtrie requested contains too many leaves. TODO: how many is too many? The leaves are represented as the list of their keys[2] and values. The peer may only return either all leaves of the subtrie or nothing. In case of `tooManyLeaves` the leaves should not returned[3]. If subtrie hash matches `subtrieHash` set in the request, the leaves also should not not sent.

[1]For a justification see Péter Szilágyi's comment at ETH v64 Wire Protocol Ring.

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

[3]In that case the peer must not return an empty list as that would imply that there are no leaves matching the given prefix.

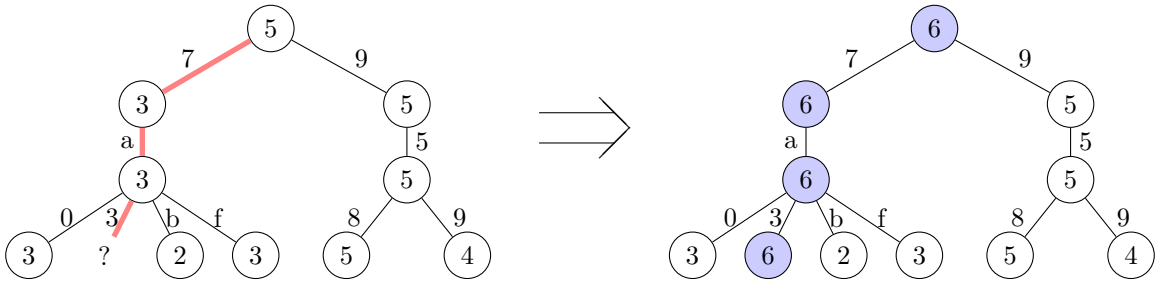

FIGURE 1. Illustration of phase 1 sync. Nodes are labeled with block numbers and edges are labeled with nibbles.

TODO: describe subtrie hash more rigorously. Proof nodes must be sent in any case; they give us a means to detect faulty or malicious peers. Note that state trie replies do not inline storage tries unlike Leaf Sync.

TODO: is block number OK given chain reorgs?

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 to send `GetSubtries` 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 `blockAtLeast` to the block of the root of the current (partially populated) trie, `fromLevel` to the number of populated nodes down the path/prefix that are of the same block as the root, and `subtrieHash` is not used (set to $\varnothing$). Having received a reply, the leecher verifies its proof. If the proof is valid, the leecher writes received 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 represent leecher's state before sending a request. Root's block is 5, so it sets `blockAtLeast` = 5. The leecher sets `fromLevel` to 1 since there is no need to re-send the root as part of the proof. It cannot set `fromLevel` higher as the other nodes along the path are older than the root and thus have to be refreshed. Suppose that the seeder replies with data as of a newer block #6. Since the block has changed, the seeder ignores `fromLevel` and sends a full proof. The leecher 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].

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 `GetNodeData2` operative. The leecher refreshes the trie level by level, starting from the root (level 0) and descending down to the penultimate level, level $d_2 - 2$. 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. Thus for each level only the nodes that have been 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 will converge. We analyse convergence conditions in the next section.

After all trie levels from the root down to the penultimate level are up to date, the leecher determines which nodes at the ultimate level $d_2 - 1$ have to be refreshed. For such nodes it sends a `GetSubtrie` request with a prefix of size d_2 corresponding to that note. The last nibble is set arbitrarily. `subtrieHash` is set in order to avoid leaf resending in case the leaves the leecher obtained during phase 1 are still valid. Given a reply the leecher finds out which of the nibbles are still valid and which have to be refreshed with `GetSubtrie`. Repeating the process for each stale node at the last level $d_2 - 1$ 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 to finding out storage sizes.

5. Performance Analysis

For 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].

Some notation (TODO: internal notation consistency + cross-check against the Yellow Paper):

o – `Leaves` reply overhead in bytes.

b – size of a branch node in bytes.

l – average leaf size in bytes.

$||R_b||$ – number of nodes in reply.

$||R_l||$ – number of leaves in reply.

Thus the size of a reply, assuming the average leaf size, is

$$S(R) = o + ||R_b||b + ||R_l||l \tag{3}$$

TODO: RLP changes the formula slightly.

t – total number of leaves in the trie.

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. Their number is, roughly speaking, equal to the size of a Merkle Patricia trie of depth d_1, times the number of block changes during phase 1. So, to reduce the overhead, we want d_1 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 KB). TODO: the formula.

Now let us find the maximum size d of the request prefix π that makes sense to use when we are catching up (phase 2 of the sync). Let assume that only one leaf that matches π has changed. (If we know that there are no changes, there is no need for a sync request.) That is a reasonable assumption if we are not too many blocks behind, there are not that many changes per block, and d is sufficiently large. (For instance, if we are 100 blocks behind, and there are 500 leaf changes per block, then $d \geq 4$ will suffice.) Consider two options: request the prefix π or send requests with prefixes $\pi \cdot 0, ..., \pi \cdot 15$ of size $d + 1$ (not necesseraly all 16 of them). In the first case we receive a reply of the size, on average,

$$S = o + ||R_b||b + \frac{t}{16^d}l \tag{4}$$

For the second case we need to send at most two requests, as the first reply will give us the information to identify which nibble has changed. (With the probability $\frac{1}{16}$ the second request is not necessary.) The combined size of those 1 or 2 replies, on average, is

$$S' = \left(1 + \frac{15}{16}\right) o + (||R_b|| + 1)b + \frac{t}{16^{d+1}} l \tag{5}$$

It does not make sense to prefer requests with longer prefixes if $S \leq S'$. Solving this inequality, we obtain

$$16^d(16b + 15o) \geq 15tl \tag{6}$$

Hence we can set the sync algorithm parameter d_2 as

$$d_2 = \left\lceil \log_{16} \frac{15tl}{16b + 15o} \right\rceil \tag{7}$$

TODO: convergence analysis.

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`.

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`.