Familiarity with Portal Network and how Ethereum stores its state (especially Merkle Patricia Trie) is required and recommended.

To avoid confusion between different types of nodes

, I will use following notation:

- NodeId
- network node - The node or its id that representing a client in the Portal Network
- node_hash
- tree node - The node or its hash that is part of the Ethereum's state, including both account tree and all smart contract trees

Problem overview

Most intuitive approach of distributing Ethereum state over Portal Network is to define distance

function between NodeId

and node_hash

. Network node will store tree nodes that are close, e.g. when distance is smaller then radius

(that is configured based on clients preferences).

Simplest distance function that can be used for this purpose is XOR

. But if we want to obtain any information from the state, we have to traverse the trie manually and make network lookup for each tree node along the path. Considering the average leaf depth of 7-9, this can take significant amount of time. Accessing the smart contract storage would almost double the number of lookups.

Solution proposal

My idea is to utilize the path to the tree node while calculating the distance. For example:

def distance(node_id, path, node_hash): key = path + node_hash[len(path):] return xor(node_id, key)

Code is given as a reference, not as implementation. There are additional things to consider, like handling nibbles, including prefix from the extension and leaf nodes, etc.

Because we use path to replace the most significant bytes in the node_hash

, we are basically adjusting the distance based on similarity between path

and NodeId

. If NodeId

would be interpreted as a path in a tree, then that NodeId

would be responsible for storing the tree nodes close to that path, creating vertical slice

of a state tree.

For a given leaf node that falls within vertical slice of a NodeId

, that NodeId

will have all tree nodes on the path starting at some depth (that depends on its radius), and have some probability of storing tree nodes in the higher levels. This will significantly improve the network lookup while doing tree traversal.

Visualization

We will assumed that NodeId

is 0x01020304...

and its radius represents ~0.01% of entire space (0x00068D...

).

Let’s explain what we see:

- blue border represents the path to our NodeId
- we expect tree nodes close

to it to be saved

- we expect tree nodes close

to it to be saved

- green nodes are always saved, regardless of what their node_hash

is

- blue nodes are never saved
- yellow nodes have two meanings:
- they represent nodes that are only sometimes saved (depending on their node_hash

)

- they usually contain all 3 types of children (blue, green, yellow)
- there is exactly one path of yellow nodes, from root to the leaf (can be calculated with xor(node_id, radius)

)

- they represent nodes that are only sometimes saved (depending on their node_hash

)

- they usually contain all 3 types of children (blue, green, yellow)
- there is exactly one path of yellow nodes, from root to the leaf (can be calculated with xor(node_id, radius)

)

Issues and things to consider

Smart contract storage

Content in the smart contract storage tree is not equally distributed. Most contracts have storage slot with small values (0, 1, 2, …) used. If the same path scheme would be used for every smart contract storage tree, then some network nodes will have significantly more data than others.

This can relatively easily be rebalanced, by combining the address of the smart contract with the path and node_hash of the corresponding tree.

Popular smart contract

Another issue is that certain smart contracts (e.g. popular ERC20 tokens (WETH, stablecoins), oracles…) are modified in almost every block. That creates the hot-spot

and breaks linear relationship between radius

and amount of data that network node wants to store.

This can be mitigated, but only to some point, by using path

in our distance

function only up to the first few bytes (e.g. 8) and reducing the radius to the point that randomness of node_hash

start to play the role.

This, however, break the colocality of the tree-nodes on the same path, which is what we wanted to achieve in the first place. It’s still better than total randomness, but very likely not good enough for practical use.