previous development update here: State Availability - GetNodeData DHT Approach (dev update) - #5 by pipermerriam

Another development update for the ongoing work to build out the State Network DHT for on-demand retrieval of the Ethereum State. We have a very early draft of a specification for this network available here: [WIP] Add first draft of state network spec by pipermerriam · Pull Request #54 · ethereum/stateless-ethereum-specs · GitHub

Design Goals

Our design goals are facilitating on-demand retrieval of the Ethereum “State”, meaning accounts, contract storage, and contract bytecode. The term “on-demand” in this context means that nodes are able to retrieve any arbitrary piece of data from the recent active state in a manner that allows it to be proven against a known recent state root.

We aim to support the “wallet” use case which is loosely defined as being able to read data from the network via eth_call

and build transactions through a combination of eth_estimateGas

and eth_getTransactionCount

(for nonce retrieval).

Performance requirements are that the majority of “normal” wallet operations can be performed in under the block time.

We are also aiming for resource constrained devices, meaning that clients from this network must be able to run the client on a single CPU, with <1GB of ram, <1GB of slow HDD. Bandwidth usage must be suitable for a residential internet connection.

High Level Network Design

The network is an overlay Kademlia DHT built on the Discovery V5 protocol that is already part of the beacon chain infrastructure and is supported by some of the core Ethereum mainnet clients as well.

Each node in the network has a node_id

as defined in the ENR specification which dictates their location

in the network. The network implements its own versions of PING/PONG/FINDNODES/FOUNDNODES messages for nodes to maintain their kademlia routing tables.

uTP for streaming data that exceeds the UDP packet size

We will be using a version of the uTP protocol.

https://www.bittorrent.org/beps/bep_0029.html

This protocol allows any two nodes in the network to establish a stream over the same UDP socket used for other DHT communication, allowing for reliable transmission of payloads that exceed the UPD paccket size.

Provability

All nodes in the network are assumed to have access to the header chain. The set of recent Header.state_root

values is used by nodes to validate proofs.

Data Storage

All data in the network has a key

and and id

. We refer to these as content_key

and content_id

.

The content_key

has semantic information and is the identifier

that nodes use to request data.

The content_id

is derived from the content_key

and dictates where in the network the content can be found. We use the same xor based distance function to determine the distance between a node_id

and a content_id

Each node in the network has a radius

which is a uint256 derived from the content they are storing. Each node in the network allocates some amount of storage. When this storage is not full, a node is considered to have a radius

value of 2**256-1 (MAX_RADIUS). When the nodes storage is full, the nodes radius

value is the distance between the node_id

and the content_id

that is furthest from the node_id

. Nodes are interested

in content that is within their radius (distance(node_id, content_id) <= radius

) and that is not already known/stored by the node. The Ping and Pong messages are used to communicate radius to other nodes. We refer to the area of the network that contains mostly interested

nodes for a given content_id

as the “region of interest”.

Nodes in the network store content that falls within their radius.

The network stores all of the intermediate trie data for both the main account trie and the contract storage tries, as well as all of the contract bytecode.

The network explicitly does not

deduplicate trie nodes and bytecode, storing multiple copies of any duplicated values to facilitate easy garbage collection.

Data Retrieval

Retrieval of content is roughly equivalent to the recursive lookup that nodes use to find a specific node or the node closest to a specific node_id. A specialized message FINDCONTENT is used to make this process slighly more efficient. To find a piece of content, a node will first start with data from their routing table, querying the node which is closest to the content_id

that is being retrieved, sending a FINDCONTENT message to that node. The response to this message will be one of:

- The raw bytes of the content itself
- The uTP ConnectionID for receipt of content that exceeds the UDP packet size
- A list of ENR records that are closer to the content than node serving the response.
- An empty response indicating that the node does not have the content and does not know about any closer nodes.

Retrieval happens one trie node at a time. This inneficiency is part of a broad set of trade-offs between total storage size, individual node responsibility, and efficiency of data retrieval. The data being retrieved from the network is incrementally proven at each stage, initially, fetching the root node for the trie, and then walking down the trie.

Data Ingress

As the chain progresses and new blocks are added, the network needs to learn about any new state data. This new state data will be provided by a small set of benevolent “data providers” who have the full state, and generate the proofs for new state data from each new block. These proofs will be against the post state root, and will only contain data that was modified or added during execution.

In almost all cases, proofs are expected to exceed the UDP packet size, and thus, will be transmitted using the uTP subprotocol.

Proofs are disceminated through two distinct mechanisms that both use the same set of base messages for communication.

1. Pushing proofs into the “region of interest” (typically originating from outside of the region)
2. Gossiping proofs within the “region of interest” (typically originating from inside the region)

Since the network stores both the leaves and the intermediate nodes of the trie, all of the nodes in these proofs will each need to be pushed to the part of the network that the content_id

for that node maps to. Thus, a proof for a single leaf, will also contain the full proof for all of the intermediate nodes needed to get to that leaf. We take advantage of this property to both reduce the amount of duplicate proof data that must be sent over the wire, as well as spreading the workload for pushing new proof data into the network across a larger number of nodes.

To push data into the network, a data provider will generate the full proof. They will then decompose that proof into a set of proofs, one for each leaf node. They will then sort these proofs by proximity to the content_id

of the leaf node. Starting with the leaf proofs that are closest

, they will perform a recursive network lookup to find nodes for whom the corresponding content_id

falls within their advertised radius, and use the Advertise message to let that node know that the proofs are available. If the node is interested in the advertised proof they will respond with a request for the proof, containing a connection_id

that the sending node should use to initiate a uTP stream for transmission of the proof. The data providers are only

responsible for disceminating proofs for the leaves.

The responsibility for pushing the proofs for the intermediate nodes, falls to the the recipient of the leaf proofs (and subsequently to the recipients of the intermediate proofs). Upon receiving a valid proof that the node is “interested” in, it will do two things. First, the node will gossip the proof to other nodes sourced from its routing table for which the content_id

falls within the node’s radius. Second, it will extract sub-proofs for the parent trie nodes, and push those to their region of interest (the same way that data providers did for the leaf proofs). This operation repeats recursively until it terminates at the state root.

Garbage Collection

This part of the network functionality is still under active research

The design of the network is currently well suited for being an “archive” node, however, the total network size necessary to store a full archive copy of the state would take widespread adoption, something that we don’t expect to achieve right away. For this reason, we need a mechanism that allows for garbage collection of trie data that is no longer part of the recent

state.

The scheme used for the content_key

is designed to ensure that content that is duplicated in multiple spots in the trie is also stored in multiple spots in the network. For example, two accounts with identical balance/nonce/code_hash/state_root

would be identical, however, we store them at distinct locations in the network by including the trie path in the content_key

. This allows us to generate exclusion proofs that show the node is not present in a recent state root, allowing nodes to discard old trie data.

Rollout and Development Plan

We are currently working on launching a very minimal prototype test network. This network will be focused on the data retrieval portion of the protocol. The goal of this experiment is to validate our assumptions about retrieval latency, which is a key performance number needed to validate the usability of the network for the indended use cases.

The uTP subprotocol is likely to be prioritized early, so that we can establish an independent spec for how uTP streams can be established between nodes in the discovery v5 network.

Work is underway to extend the Turbo-Geth API to support generation of the proofs that “data providers” will need to push data into the network.

After the test network experiment has been done, assuming we are indeed able to validate our projected latency numbers, we will move onto focusing on actual client development for the full protocol. Rough projections of the work that needs to be done suggest that once development is fully underway, we should be able to have roughly full proof of concepts available in a 6-month timeline, and production clients within a 12-month timeline.