Over the last month I spent some time exploring the idea of using a DHT to store and serve both the chain and state data for the Ethereum blockchain. The experiment was done using this codebase which implements a functional DHT network client capable of storing and serving key/value based data as well as an additional layer implementing a skip graph over the keys which facilitates range queries over the DHT.

Goals

Explore the feasibility of using a DHT to provide any of the chain or state data for Ethereum clients.

DHT Details

The [alexandria

](https://github.com/ethereum/alexandria) code which implements the fundamentals of the DHT is a slightly simplified version of the discovery v5 protocol which is focused on solving the discovery problem in both Eth1.x and Eth2 contexts.

At a high level the network is structured as follows.

- Every node on the network has a NODE_ID

which is a 256 bit integer derived from their public key.

- Every piece of content has a key

which is an unbounded byte string. Each key maps to a CONTENT_ID

which is a 256 bit integer. * *Note that the CONTENT_ID

and the node_id

share the same keyspace

- *Note that the CONTENT_ID

and the node_id

share the same keyspace

- The network uses the standard xor

distance metric to determine proximity.

Content is handled in the following way.

- Any node may store any content they wish.

- Nodes that are close

to a specific CONTENT_ID

are expected to maintain an index of which nodes have that content.

- Nodes are expected to advertise the content they have to the nodes which are close

to the content_id

.

- Content is referenced using the raw key

rather than the content_id

to ensure that content is not opaque and that additional semantic information can be derived from the key. * I.E. headers might be stored under the key header/

which would allow a client to validate the data is indeed a valid header with the given hash…

- I.E. headers might be stored under the key header/

which would allow a client to validate the data is indeed a valid header with the given hash…

The client then implemented the following conventions.

- Each client would allocate a fixed size storage, prioritizing content that is close to their position in the network.
- While a client’s database is not full it will store and serve all content it encounteres.
- Once a client’s database is full can use the distance metric as the eviction mechanism.
- While a client’s database is not full it will store and serve all content it encounteres.
- Once a client’s database is full can use the distance metric as the eviction mechanism.

The Skip Graph is a distributed data structure described in thiswikipedia article and this research paper. Explaining the overall concept is beyond the scope of this document but at a high level it accomplishes the following.

- insert new nodes into the keyspace with minimal coordination cost.
- query the overall keyspace for a specific key or the keys that are closes to it if the key doesn’t exist.
- traverse the keyspace in a concurrent manner.

Given the above mechanism that operate on the raw content keys, we then have to make some choices about how we construct our keys to facilitate efficient retrieval.

We will want most to be unambiguously retrievable by their unique identifier.

- header/<block-hash>
- transaction/<txn-hash>
- receipt/<txn-hash>

This type of retrieval is ideal for things like serving the eth_getTransactionByHash

JSON-RPC API.

We also probably want data to be retrievable by more contextual information about how it is located in the chain. For example we might store the mapping of block numbers to canonical block hash under the key chain//header/

. This is designed to facilitate things like:

- serving the eth_getBlockByNumber

JSON-RPC API by first looking up the block hash, and then looking up the actual header.

- efficient bulk retrieval of the header chain by executing a range query across the keyspace for the block numbers.

Another example:

storing the block bodies storing each of the block transaction’s hashes under: .../block/transactions/

. The transactions themselves would be stored under transaction/

. This allows an efficient range query to find all of the transaction hashes for a given block, then retrieval of the individual transactions in parallel.

My Findings

Between the DHT key/value retrieval mechanisms and the Skip Graph’s ability to perform more sophisticated queries across the keyspace combined with well thought out conventions for how we use the keyspace I believe we can create a network which exposes the ability to do the following types of things for the chain data.

- Efficient storage and retrieval of individual Headers, Transactions and Receipts.
- Efficient bulk retrieval of:
- Headers by block number range
- Block bodies (tansactions and headers)
- Receipts for a given block
- Headers by block number range
- Block bodies (tansactions and headers)
- Receipts for a given block

It should be possible to design the network such that all of the following JSON-RPC APIs could be served by querying data on demand from the network.

JSON-RPC Endpoint Analysis

Given that one of the big things that this network facilitates is serving of JSON-RPC APIs, here is a small analysis of the JSON-RPC endpoints with respect to this network.

- Requires tracking the header chain
- eth_blockNumber
- eth_blockNumber
- Requires state data
- eth_getBalance
- eth_getStorageAt
- eth_getTransactionCount
- eth_getCode
- eth_getBalance
- eth_getStorageAt
- eth_getTransactionCount
- eth_getCode
- Requires state data and an EVM engine
- eth_call
- eth_estimateGas
- eth_call
- eth_estimateGas
- Servable by querying data from network
- eth_getBlockTransactionCountByHash
- eth_getBlockTransactionCountByNumber
- eth_getUncleCountByBlockHash
- eth_getUncleCountByBlockNumber
- eth_getBlockByHash
- eth_getBlockByNumber
- eth_getTransactionByHash
- eth_getTransactionByBlockHashAndIndex
- eth_getTransactionByBlockNumberAndIndex
- eth_getTransactionReceipt
- eth_getUncleByBlockHashAndIndex
- eth_getUncleByBlockNumberAndIndex
- eth_getBlockTransactionCountByHash
- eth_getBlockTransactionCountByNumber
- eth_getUncleCountByBlockHash

- eth_getUncleCountByBlockNumber
- eth_getBlockByHash
- eth_getBlockByNumber
- eth_getTransactionByHash
- eth_getTransactionByBlockHashAndIndex
- eth_getTransactionByBlockNumberAndIndex
- eth_getTransactionReceipt
- eth_getUncleByBlockHashAndIndex
- eth_getUncleByBlockNumberAndIndex
- Log filters are complicated and require an index of the full chain data. (cannot serve)
- eth_newFilter
- eth_newBlockFilter
- eth_newPendingTransactionFilter
- eth_uninstallFilter
- eth_getFilterChanges
- eth_getFilterLogs
- eth_getLogs
- eth_newFilter
- eth_newBlockFilter
- eth_newPendingTransactionFilter
- eth_uninstallFilter
- eth_getFilterChanges
- eth_getFilterLogs
- eth_getLogs
- DevP2P/Mining/Key-management (Not Relevant)
- eth_protocolVersion
- eth_syncing
- eth_coinbase
- eth_mining
- eth_hashrate
- eth_gasPrice
- eth_accounts
- eth_sign
- eth_pendingTransactions
- eth_sendTransaction
- eth_sendRawTransaction
- eth_getWork

- eth_submitWork
- eth_submitHashrate
- eth_protocolVersion
- eth_syncing
- eth_coinbase
- eth_mining
- eth_hashrate
- eth_gasPrice
- eth_accounts
- eth_sign
- eth_pendingTransactions
- eth_sendTransaction
- eth_sendRawTransaction
- eth_getWork
- eth_submitWork
- eth_submitHashrate

Complexities, Attack and Mitgations

I thought about the following issues and problems

Eclipse Attacks

A malicious actor which wished to attack the network could choose a popular piece of content, then mine public keys that map to node ids which are very close to the content. Given enough nodes with node ids that are in close proximity to the content, the malicious actor would then be in control of the majority or all nodes that are expected to maintain the index for where that content can be found.

Mitigating against this type of attack is non trivial and should be a continued area of invistigation and research. The severity of this attack is a function of how we intend to use this network. I believe that this network can still be of high value even if very little is done to mitigate against this attack vector.

Simple mitigation mechanisms might be to use a slow hash function like scrypt to derive the node_id from the public key, increasing the cost of mining node ids. There are network level mechanisms can be put into place to detect malicious nodes, allowing nodes to find and blacklist malicious actors.

Flooding

A malicious actor could flood the network with junk content. This could be done at the level of the entire network, or at the level of a specific area of the keyspace in an attempt to overwhelm a specific area of the network.

If we assume clients have access to the header chain (more on this below) then all content stored in this network should be verifiable. That means that a client can ignore keys that are not in a known format which mitigates against generic flooding of junk data. That restricts content to a predefined keyspace, meaning that all content can be verified against he header chain before storage.

To the best of my knowledge, this provides strong mitigation against these two types of attacks.

DOS

Any individual node on the network would be subject to DOS attacks. However, given the nature of DHT networks, the network as a whole should be quite resiliant to these types of attacks…

Back to Discovery V5

Extending Discovery V5 to support the mechanisms above is both straight forward. The only requirements necessary for the network to operate would be for the nodes on the network to maintain the in-memory index of where content can be found on the network. The network does not

depend on all nodes storing content. The minimum requirements for a client on this network should be very

small.

What about the Account and Contract State

The Ethereum state data (accounts and contract storage) is a more complex data set to manage. Given the quick rate of change, storing and retrieving this data in a DHT is more complex but feasible.

My current opinion is that we should not

aim to store this data in the DHT during initial roll out and implementation, but rather should plan to extend the network at a later data to support storage and retrieval of this data. The network itself should not require any new functionality, but the client functionality is likely to be quite complex and it is likely to be highly dependent on the availability of block witnesses which we currently cannot guarantee.

How to do it when we do it

When we do choose to do this, I believe that it should/would be done loosely as follows.

- A client would choose what part of the account and contract storage they wish to maintain and store a proof for the full set of that data.
- A client would ingest witnesses to keep their proof up to data.
- A client can choose to maintain as many historical proofs as they wish such as keeping the last 128 blocks worth of proofs available.

Using the keyspace a client wanting part of the state would look that content up the same way they would look up any other content, either by exact key/value or via a range query. The conventions we choose for how to use the keyspace will matter because they will determine how efficiently we can retrieve the data.

The other main issue is the rate at which this data is changing. With a new block coming in every 15 seconds, and each block touching a few thousand keys in the state, that means that the network would incur baseline overhead of a few thousand messages every 15 seconds as clients advertise the new

state data they have. Coming up with a scheme that minimizes how often clients have to advertise this data while still making the data retrievable with state-root level granularity will be important to ensuring the network can operate efficiently.

Next Steps

I believe that this line of research and investigation should continue. The next steps I see are:

1. Talk with the minds behind Discovery V5. The most ideal scenario is for this to be part of that network so that we get the benefit of all Ethereum clients being part of this network by default.
2. Define the scope of work. What data do we wish to serve over this network?
3. Build out MVP and document the protocol from the existing discovery v5 clients and create a “testnet” of sorts to allow Ethereum clients to start participating and leveraging this network.

I would advocate eliminating the Skip Graph from the initial MVP and focusing on key/value based storage and retrieval, after which we can layer on the Skip Graph.