I'm both providing a link to the working document, and a copy of it's current state. The document is likely to continue to evolve. The current state hasn't been wordsmithed or really edited much for easy digestability, but rather is a loosely structured dump of my thoughts on the subject.

Living document: https://notes.ethereum.org/g-npI1x7QRqoSzclvkLYzw?both

DHT Ethereum Data Network

An idea to try and use a single DHT for data storage.

TODO: This paper looks promising: https://arxiv.org/pdf/1608.04017.pdf

Data Structures

- Chain Data:
- headers
- blocks (bundles of transactions and uncles)
- receipts (bundled by block)
- witnesses (need to be referenced by header)
- headers
- blocks (bundles of transactions and uncles)
- receipts (bundled by block)
- witnesses (need to be referenced by header)
- State Data
- accounts
- referenced by path?
- bundled by tile?
- referenced by path?
- bundled by tile?
- contract storage
- dunno yet, tiling?
- dunno yet, tiling?
- contract code
- dunno yet
- dunno yet
- accounts
- referenced by path?
- bundled by tile?
- referenced by path?
- bundled by tile?
- contract storage
- dunno yet, tiling?
- dunno yet, tiling?

- contract code
- dunno yet
- dunno yet
- Canonical Chain Metadata
- canonical header chain (trie of tries proof)
- block-number-to-canonical-header-hash (trie of tries proof)
- transaction-hash-to-canonical-headaer-hash (trie of tries proof)
- canonical header chain (trie of tries proof)
- block-number-to-canonical-header-hash (trie of tries proof)
- transaction-hash-to-canonical-headaer-hash (trie of tries proof)

DHT Keyspace

Data from different chains needs to be namespaced differently?

Prefix:

eth//

TODO: is forkid suitable for this, probably yes for old forks and maybe no for new forks since the forkid changes as soon as a there is a pending new fork, need a better mechanism…

- Chain Data:
- Headers: /header/
- Blocks: /block/
- Receipts: /receipt/
- Witnesses: /witness/
- Headers: /header/
- Blocks: /block/
- Receipts: /receipt/
- Witnesses: /witness/
- State Data:
- Accounts: `/account/<??tile-identifier??>
- TODO: stateroot based granularity?
- TODO: stateroot based granularity?
- Contract Code: /contract-code/
- Is this an acceptable way to do contract code that handles deduplication?
- How do we mitigate against spam (we only want to store code that is part of at least one contract…)
- Is this an acceptable way to do contract code that handles deduplication?
- How do we mitigate against spam (we only want to store code that is part of at least one contract…)
- Account Storage: /storage/

  /
- TODO: stateroot base granularity?

- TODO: stateroot base granularity?
- Accounts: `/account/<??tile-identifier??>
- TODO: stateroot based granularity?
- TODO: stateroot based granularity?
- Contract Code: /contract-code/
- Is this an acceptable way to do contract code that handles deduplication?
- How do we mitigate against spam (we only want to store code that is part of at least one contract…)
- Is this an acceptable way to do contract code that handles deduplication?
- How do we mitigate against spam (we only want to store code that is part of at least one contract…)
- Account Storage: /storage/

  /
- TODO: stateroot base granularity?
- TODO: stateroot base granularity?

A possible alternate keyspace for state.

- /state//

  /account
- /state//

  /code
- /state//

  /storage/

These give state root level granularity but result in a complex amount of high key churn for each block, introducing something like a few thousand new keys for each block, many of which will not have their values changed…

Mapping keys to storage locations…

One desired property of the function which maps content to locations on the network would be for it to facilitate nodes storing chunks of the data.

- For headers/blocks/receipts/witnesses this would mean having two subsequent block numbers of data map themselves onto adjacent-ish locations in the network.
- For accounts we can accomplish this pretty easily by distributing the whole account range evenly across the network keyspace.
- This is potentially attackable by someone mining addresses that are close together and creating a heavy spot in the key space…
- This is potentially attackable by someone mining addresses that are close together and creating a heavy spot in the key space…
- For contract storage we'd want the keyspace for all contract storage to map evenly across the entire keyspace which isn't trivial since contract storage isn't evenly distributed…

Whatever the mechanism, we want uniform distribution across the entire address space…

Rendevous Hashing looks promising as a simple well tested way to map data into multiple locations across the network. Multiple locations might not be necessary but it seems like a nice mechanism to encourage redundancy.

Deciding who stores what…

One aim is for the network to have as simplistic a model as possible for participation. That means that writing a client for this

network should be as simple as possible.

Much of what follows may in many ways mirror the functionality of the SWARM network.

Content in the network is comprised of a key

and a value

. The key

would be one of the namespaces above. Each key maps to a set of storage location in the network s0, s1, s2, ...

.

Each node in the network's public key denotes their position in the network. The prescribed behavior of each node is as follows.

- Each node maintains a "Data Store" which is tied to their NodeID: D
- Each data store has a capacity C

measured in bytes.

- The current total size of all data in the data store (excluding key sizes) is S
- For each key, let r = radius(k, nodeid)

where radius(k, nodeid)

calculates the distance between the nodes location and the closest storage location for the given key.

- Each time a new piece of data with key k

and raw bytes value v

is encountered: * If S + len(v) <= D

: store the data

- If the radius of the new data is smaller than any existing data in the data store, then the data store evicts the items with the largest radius until there is room for the new item.
- If S + len(v) <= D

: store the data

- If the radius of the new data is smaller than any existing data in the data store, then the data store evicts the items with the largest radius until there is room for the new item.

Each client will also need to maintain some global state of the network. This is done by tracking the header chain. This global network state is vital for validating the data that is flowing into the storage network.

- Headers: standard header validation done by eth clients.
- Blocks: validate transactions and uncles against header
- Receipts: validate against header receipt root
- Witness: validate against hash from header
- Account State: validate against state root from header.

All of headers/blocks/receipts/witness

should be kept indefinitely.

Account state however may need to behave slightly differently. It would be ideal if this network could operate as an archive node, however that is complex. Since nodes don't have a full picture of the state trie, they will need to store proofs about the accounts they are tracking. That means naive storage will involve a lot

of duplication across the various proofs.

TODO: can we use a different hash function for account storage so that each network node maps to a single location in the account trie and the node instead stores accounts that are near

them. This allows for effective range queries across accounts that can be distributed across multiple nodes.

TODO: how do we do contract storage. Unification of the storage tries would make this a lot easier. The goal would be a simple mechanism that allowed us to map nodes into ranges of the storage in a manner than automatically balances across the nodes of the network while easily allowing nodes to store ranges of the storage items.

Storage Ranges

This is partially covered in the mapping content to storage locations section.

We want to optimize for certain use cases:

- Full nodes syncing
- requests chain data in sequential ranges
- requests state data in sequential chunks, enumerating the key space.
- requests chain data in sequential ranges
- requests state data in sequential chunks, enumerating the key space.
- Stateless nodes
- unknown, but likely has a totally random request pattern?
- unknown, but likely has a totally random request pattern?

In order to optimize for efficiency, the syncing node case suggests we want nodes to store ranges of data.

- Headers are constant size, so they can be chunked into constant size chunks.

Blocks/Receipts/Witnesses

are all variable size. No immediate obvious solution for efficient chunking which bounds the chunk sizes…

The various “state” values are also dynamic in size. Tiling is supposed to solve this for accounts. I don’t know how contract storage works but probably the same. Need to figure out how Tiling works.

Discarding old data:

Should/can the network operate as an archive node? or must it discard old state?

The account state data may need a different storage model than the chain data. We only really need

to keep around the data for recent tries. It would be ideal if we could just keep everything, but research needs to be done to evaluate feasibility. Under the assumption that keeping everything is too expensive for the network, then nodes would need logic to discard state data from old state roots.

Inclusion Proofs via trie of tries accumulators

Use a trie of tries to maintain light

proofs about chain history and efficient reverse lookups.

TODO: explain this in detail…

Data near the head of the chain can be maintained in memory. TODO: full description of what a node would keep in memory and evaluation of how hard it is to maintain those data structures, as well as how a node joining the network would build or acquire those structures.

How data is requested

A full syncing node

- Enumerates all headers, typically by range
- Enumerates all blocks, receipts, typically by hash, but often in sequential ranges.
- Enumerates the account state trie.

- at specific state root(s)
- at specific state root(s)
- Requests the code for all accounts by their code hash
- Enumerates the storage tries for all accounts
- at specific state root(s)
- at specific state root(s)

A stateless node

- Sparse requests for headers, blocks, receipts
- Random requests for proofs about accounts and their storage
- at a specific state root(s)
- at a specific state root(s)
- Random requests for the code for an account
- by address?
- by code-hash?
- by address?
- by code-hash?
- Random requests for index lookups (txn -> block, number -> block)

The volume of requests issues by a stateless node isn't necessarily bounded but there's likely an upper bound for which we could consider reasonable

. Maybe the protocol can take this into account?

Attacks that need to be understood and potentially mitigated

- Malicious routing?
- Sybil attacks which eclipse a part of the network to wall off data.
- Sybil attacks which eclipse a part of the network to wall off data.
- Ensure that routing doesn't result in cycles
- DOS/flooding/amplification

Things that are dealt with already?

Spam is probably not a big deal since all requests to store content should be verifiable against the header chain.

Thoughts on advertising what you have…

It is unclear whether the protocol needs the ability for nodes to be expressive about what data they have available. Reading some research documents on distributed content networks it appears that there are some formal mechanisms for this that can also be used to do things like provide anonymity see this paper

Thoughts on ensuring everyone's view of nodes is homogenous

Worth stating that an explicit goal of this network is for nodes to be treated homogenously. A "full" node could participate in this network alongside a fully stateless node and all other things being equal (hardware, bandwidth) the two nodes should be interchangeable from the perspective of any other peer, and

a peer should not be able to reliably discriminate against one or the other (ignoring the ability to use things like heuristics to discriminate based on latency or something similar).

“Infinite” Scaling With One Beneveolent Full Node

The network provides no guarantees about data persistance. In fact, it is expected for some pieces of data to go missing occasionally. An explicit goal is to have the network operate in such a way that a single “benevolent” node which has access to the full node data could monitor the data network for missing data and then push that data back into the network to patch the holes.