I'd like to spend a minute to explore the use cases for the "State Network". I'll start with a short description of what I mean by "State Network" and then will do my best to enumerate the various use cases, providing basic context for each of them.

The purpose is to establish a common ground from which we can determine the specifics of the network architecture and actually start building something.

What is the "State Network"

A special purpose decentralized storage network that is capable of serving the Ethereum chain history and

state. At this stage we are agnostic about whether this is a DHT, LibP2P, DevP2P, whatever…

Network Participants

The network design should assume a wide range of network participants, likely with the following rough shape.

- Leechers
- nodes that jump on and offline quickly (a few minutes or less), leeching a small amount of data from the network.
- such nodes might be part of a CLI that reads data on demand but doesn't operate as a long lived process.
- nodes that jump on and offline quickly (a few minutes or less), leeching a small amount of data from the network.
- such nodes might be part of a CLI that reads data on demand but doesn't operate as a long lived process.
- Ephemeral
- nodes that stay on the network for short periods of time (a few minutes up to maybe a few hours) and who may primarily leech but which are capable of also serving data and will do so if it is easy.
- such a node might be integrated into something like MyCrypto or other wallets and only active while the app is open.
- nodes that stay on the network for short periods of time (a few minutes up to maybe a few hours) and who may primarily leech but which are capable of also serving data and will do so if it is easy.
- such a node might be integrated into something like MyCrypto or other wallets and only active while the app is open.
- Long Lived, Low Resource
- nodes that stay on the network reliably

for many hours or days which act as data providers but have low resources (1 CPU, storage measured in the 100's of MB)

- such a node might be integrated into an ethereum client.
- nodes that stay on the network reliably

for many hours or days which act as data providers but have low resources (1 CPU, storage measured in the 100's of MB)

- such a node might be integrated into an ethereum client.
- Long Lived, High Resource
- same as the "low resource" counterpart, but these nodes have more CPU and storage, maybe serving the entire state or entire history.
- there would likely be very few of these
- such a node would likely be run by a benevolent enthusiast
- same as the "low resource" counterpart, but these nodes have more CPU and storage, maybe serving the entire state or entire history.
- there would likely be very few of these
- such a node would likely be run by a benevolent enthusiast

This lets us define some minimal resource targets since we want nodes with minimal CPU and storage to be able to operate on this network as first class citizens.

Use Cases: High Level

At a high level we want to support the following use cases. These are the core of of what the network needs to do.

- Syncing the full chain history (not the state)
- Facilitating on-demand retrieval of chain history to allow clients to forget

about some of the history.

- Facilitating on-demand reads of the state to allow clients without access to the state to do things like sending transactions (and the implicit gas estimation that goes with them).
- Serve the JSON-RPC API with a few exceptions without need of a full client.

Another requirement is that all retrieved data be provable. I won't get into details on this because the method for proving each piece of data is likely different, though they all likely link back to the header chain.

What follows is a more granular breakdown of exact capabilities necessary to fulfill these use cases.

Use Cases: Chain History

All of this data is easy to manage. Historical data doesn't change so it is a simple dataset that simply grows over time with a mostly constant rate that new data is added.

Header Chain

- Retrieve block headers by hash

This can be used for:

- A client syncing the header chain
- Serving JSON-RPC endpoints like eth_getBlockByHash

Block Bodies

- Retrieve the ordered set of transactions and uncles by block hash

This can be used for:

- A client syncing the block chain
- Serving JSON-RPC endpoints like eth_getBlockByHash

Receipts

- Retrieve the ordered set of receipts for a block by block hash

This can be used for:

- A client syncing the block chain
- Serving JSON-RPC endpoints like eth_getTransactionReceipt
- also requires the chain index for txn_hash -> block_hash

lookups.

- also requires the chain index for txn_hash -> block_hash

lookups.

Canonical Chain Index

- Given a block number lookup the canonical block hash
- This use case likely requires adding a header accumulator to the Eth1 protocol which would allow for inclusion proofs. Without this, verification of this data would require a local copy of the header chain which is multiple GB in size which violates the minimum resource requirements.

- This use case likely requires adding a header accumulator to the Eth1 protocol which would allow for inclusion proofs. Without this, verification of this data would require a local copy of the header chain which is multiple GB in size which violates the minimum resource requirements.
- Given a transaction hash, lookup the canonical block hash in which it was included.

This can be used for:

- Serving JSON-RPC endpoints like eth_getBlockByNumber

, eth_getTransactionByHash

, eth_getTransactionReceipt

Use Case: State

The state data is divided up into millions of small pieces. Some pieces change often. Some pieces rarely or maybe never change. The data is all anchored to a state root. Most data should be accompanied by an inclusion proof against a state root.

Account State

This data is evenly distributed across a trie that is somewhat straight forward to shard and distribute evenly without coordination.

- Ability to do on-demand lookups of individual accounts from the state trie and an accompanying proof for a recent state root
- unclear whether this can be a little flexible on the proof state root, or whether this needs to be anchored to a specific state root.
- unclear whether this can be a little flexible on the proof state root, or whether this needs to be anchored to a specific state root.
- Ability to retrieve the account bytecode associated with the account code hash.

This can be used for:

- Patching up a local state database.
- Serving JSON-RPC endpoints like eth_getBalance

, eth_getTransactionCount

, eth_getCode

, etc.

This is not

intended to be used for syncing the full state. We would rely on SNAP or MGR for such things. This could

be use to augment or assist in syncing the full state.

Contract State

This data is unevenly distributed and varies widely in shape and size depending on the account.

- Ability to do on-demand reads of arbitrary contract state with an accompanying proof for a recent state root.
- unclear whether this can be a little flexible on the proof state root, or whether this needs to be anchored to a specific state root.
- unclear whether this can be a little flexible on the proof state root, or whether this needs to be anchored to a specific state root.

This can be use for:

- Patching up local state database
- Serving JSON-RPC endpoints like eth_call

, eth_estimateGas

- Stateless block execution without a witness (beam sync)

Use Case: Eth2

I need help filling this section in as I know this type of network is desired for Eth2 but I’m unclear on exactly what needs Eth2 has.