In our research into how to efficiently store Ethereum's state data in an archival format similar to Erigon's reverse diff based approach we found a novel way to store large SSZ objects in a DHT such that:

- The data is evenly distributed across the keyspace of the DHT
- The data is cryptographically anchored to the SSZ merkle root for both gossip and retrieval.

This approach appears to be extensible to any SSZ object.

Portal Network context

While it is possible this approach could extend to other DHT networks, our research focuses solely on the Portal Network.

In order for our networks to be secure, Portal requires that all data be cryptographically anchored during both the gossip and retrieval. In this context, cryptographic anchoring is the process through which a node in the network is able to verify that the data they are looking at is both well formed and

that it represents data that belongs in the network. An example of this would be an ommer header, which would be well formed and would pass validation as a viable header, but it should not be stored in our networks since it is not canonical.

SSZ objects mapped to a DHT

For the SSZ object in question, we first compute the SSZ root hash which we'll refer to as the PrimaryRoot

.

Next, we serialize the object into it's encoded representation, a stream of bytes. We then compute a second SSZ root hash using the ssz schema of a simple List[uint8, max_length=...]

. This second root hash will be referred to as the FlattenedRoot

.

The bytes of the encoded SSZ object can then be mapped 1:1 onto the DHT keyspace based on the position of the bytes with respect to the total encoded length, with the 0th byte being mapped to the 0x00..00

location in the network, and the nth

byte being mapped to the node-id that corresponds to (2**256 -1) / n

.

In our setup, our individual nodes are able to trust that FlattenedRoot

and PrimaryRoot

both represent the same data. However, it seems this scheme would allow for nodes in the network to perform verification against both roots if desired.

Gossip

In order for nodes to accept a sequence of bytes for storage during the gossip process they must be able to verify:

- The location in the network that the bytes belong.
- That the bytes are cryptographically anchored to the respective ssz roots PrimaryRoot

and FlattenedRoot

During gossip a contiguous section the encoded SSZ object bytes would be selected, and a SSZ merkle proof against the FlattenedRoot

is constructed. We'll refer to this proof as the flat_proof

. In the event that the system also requires proofs against the PrimaryRoot

then this secondary proof would also need to be constructed against the subset of the data that the selected bytes represents which we'll call the primary_proof

. Care would need to be taken in the selection of bytes to ensure that its bounds are picked to match up with the encoded object boundaries.

The gossip payload would include these elements.

- The section of bytes to be stored.
- The flat_proof
- The primary_proof

if desired.

Since the flat_proof

is against a flat List[uint8, ...]

data structure, the accompanying merkle proof will have sufficient information to determine exactly where in the overall sequence of bytes the receive payload would belong. This allow the receiving node to cryptographically validate that the payload belongs in their storage based on on their node-id as well as verifying that the payload is correctly anchored to the FlattenedRoot

and optionally the PrimaryRoot

if both proving methods are desired.

Retrieval

For retrieval we assume the object has been fully seeded into the network and all portions of its encoded bytes are retrievable.

All SSZ objects have a “schema” and this schema provides a deterministic way to find where a specific part of that object is located in the encoded binary format without requiring the full data structure to be decoded. Only a small subset of the data must be decoded to retrieve individual parts of the object.

Since the raw bytes of the encoded SSZ object are available from the network, this facilitates what is effectively the ability to read bytes from anywhere in the stream by fetching them from the network. Each payload can be cryptographically verified with the accompanying flat_proof

and primary_proof

.

Each SSZ object will require some amount of custom logic in order to retrieve the data from the network, though a sufficiently advanced SSZ library should be able to implement this logic in a generic format as it would be the same logic necessary to do selective decoding of only part of an SSZ object.

Applications of this approach

We are exploring the idea of gaining more efficient storage by densely packing data from deep in Ethereum’s history into large SSZ objects which are then stored using this schema in the network. This format allows for both random access into the data structure as well as bulk retrieval of the full data structure by retrieving all of the byte sequences stored within the network.

This approach allows for efficient algorithms such as performing a binary search over a sorted SSZ list stored using this mechanism. This allows for us to implement a version of Erigon’s diff based storage approach to storing Ethereum’s state data, effectively trading more client side compute for less required storage in the overall network. For Ethereum’s archive data this results in roughly a 10x reduction in total needed storage for the merkle patricia tree and likely significantly higher savings for the verkle trie due to it’s very wide branching factor.

It is also likely that applying this approach to ethereum’s history data would allow for more efficient bulk retrieval of historical headers and block bodies.

There are also likely applications for storing the Consensus state in this manner as it would allow for random access retrieval of any part of the state.