Background:

- A layer-1-minimizing phase 2 state execution proposal
- An Optimistic Generic Gas Market Executor for Phase 2
- Phase One and Done: eth2 as a data availability engine

Here is another proposal for phase 2 that's halfway in between my earlier one (#5397) and Casey's "phase one and done" proposal.

- For each execution environment, we store an array states: List[{slot: slot, value: bytes32}, SHARD_COUNT]

in the beacon chain (execution environments could restrict themselves to fewer shards to cut storage reqs and hence deployment costs), which starts at zero for each shard.

- Every shard block specifies which execution environment E

is intended to execute it (this is the choice of the proposer)

- Each shard chain also stores a data structure shard.states: Map[int -> {slot: slot, value: bytes32}]

which stores states for different execution environment. The goal of this data structure is to only store states that are newer than those stored in the beacon chain.

- The "current state" cur_state(shard, exec_env) -> Bytes32

can be computed via shard.states[E].value if E in shard.states and shard.states[E].slot > beacon_chain.envs[E].states[s].slot else beacon_chain.envs[E].states[s].value

. That is, take the newest record available, whether it's from the shard or the beacon chain.

- When a block appears in a shard s

specifying execution environment E

, then let pre_state = current_state(s, e)

. Compute post_state = exec(beacon_chain.envs[E].code, pre_state, block_data)

and save shard.states[E] = {slot: current_slot, value: post_state}

.

- The full contents of shard.states

should be included in any crosslink, and if a crosslink is accepted, these values should be included into the beacon chain.

- If a shard execution sees that the beacon chain's record for an execution environment is the same as that of the shard, it should remove the record from the shard (this lets the shard states decrease in size over time)

The goal here is that the consensus-layer state per shard is reduced to only 32 bytes per execution environment, and this is all stored in the beacon chain; the shards don't have a significant concept of state on their own.

In practice, this would mean that there would be a two-layer structure for users,as described here. Roughly:

- Users

send transactions.

- An execution environment specific class of actor called a relayer

packages up these transactions and adds Merkle proofs. They rebroadcast the packages and pass along a fee to the proposer.

- Block proposers

accept the package that pays them the most.

The execution environment's code could use the 32 byte state field to store a state root, and verify Merkle proofs provided by the relayer to implement state transitions. The concept of "full state" would be an execution-environment-specific higher-level concept.

Advantages

- We can merge the crosslink and persistent committees into one, halving the amount of in-consensus verification that we need to do (the verification doesn't quite

halve, because the verification we're removing would be the cheaper persistent stateful verification rather than the more expensive persistent stateless variety, but it's still a gain…). The merged committees could have a rotation period of eg. 1-3 hours.

- We remove the need to specify the Merkle tree structure in-protocol, allowing execution-environment-layer experimentation in: different kinds of Merkle trees (SMTs, red-black trees, etc…), using SNARKs or STARKs to compress Merkle witnesses, accumulators other than Merkle trees (RSA accumulators, etc)
- The decision of what rent scheme to use gets pushed up to execution environments to decide

Disadvantages

- More stuff to formally verify (alternatively, we could agree that for some limited time, execution environments are subject to low-controversy DAO-style rescue forks)