Special thanks to @vbuterin for the original idea, @djrtwo, @zilm and others for review and useful inputs.

TL; DR

an eth2 execution model alternative to executable shards with support of single execution thread enshrined in the beacon chain.

Recently published rollup-centric roadmap announces data shards as the main scaling factor of execution in eth2 allowing scalability upon a single execution shard and simplifying the overall design.

Eth1 Shard design supposes communication with data shards through the beacon chain. This approach would make sense if Phase 2 with multiple execution shards were going to be rolled out afterwards. With the main focus on rollup-centric roadmap, placing Eth1 on a dedicated shard (that is, independent from and frequently “crosslinked” the beacon chain) puts unnecessary complexity to the consensus layer and increases delays between publishing data on shards and accessing them in eth1.

We propose to get rid of this complexity by embedding eth1 data (transactions, state root, etc) into beacon blocks and obligating beacon proposers to produce executable eth1 data. This enshrines eth1 execution and validity as a first class citizen at the core of the consensus.

Proposal overview

- Eth1-engine is maintained by each validator in the system.
- When validator is meant to propose a beacon block it asks eth1-engine to create eth1 data. Eth1 data are then embedded into body of the beacon block that is being produced.
- If eth1 data is invalid, it also invalidates the beacon block carrying it.

Eth1 engine modifications

According to the previous, Eth1 Shard centric, design, eth1-engine and eth2-client are loosely coupled and communicate via RPC protocol (check eth1+eth2 client relationship for more details). Eth1-engine keeps maintaining transaction pool and state downloader which requires own network stack. It also should keep storage of eth1 blocks.

Current proposal removes the notion of eht1 block and there are two potentials ways of eth1-engine to handle this change:

- synthetically create eth1 block out of eth1 data carried by beacon block
- modify the engine in a way that eth1 block is not needed for transaction processing and eth1 data is used instead
- beacon block roots may be used to keep a notion of the chain which is currently required by state management
- beacon block roots may be used to keep a notion of the chain which is currently required by state management

The former option looks more short term than the latter one. It allows for faster turn of eth1 clients into eth1-engines and has been proved already by eth1 shard PoC.

We use term executable data

to denote data that includes eth1 state root, list of transactions (including receipts root and bloom filter), coinbase, timestamp, block hashes and all other bits of data required by eth1 state transition function. In the eth2 spec notation it may look as follows:

class ExecutableData(Container): coinbase: bytes20 # Eth1 address that collects txs fees state_root: bytes32 gas_limit: uint64 gas_used: uint64 transactions: [Transaction, MAX_TRANSACTIONS] receipts_root: bytes32 logs_bloom: ByteList[LOGS_BLOOM_SIZE]

A list of eth1-engine responsibilities looks similar to what we used to have for Eth1 Shard. Main observed items of it are:

- Transaction execution.

Eth2-client sends an executable data to the eth1-engine. Eth1-engine updates it inner state by processing the data and returns true

if consensus checks have been passed and false

otherwise. Advanced use cases, like instant deposit processing, may require full transaction receipts in the result as well.

- Transaction pool maintainance.

Eth1-engine uses ETH network protocol to propagate and keep track of transactions in the wire. Pending transactions are kept in the mempool and used to create new executable data.

- Executable data creation.

Eth2-client sends previous block hash and eth1 state root, coinbase, timestamp and all other information (apart of transaction list) that is required to create executable data. Eth1-engine returns an instance of ExcecutableData

.

- State management.

Eth1-engine maintains state storage to be able to run eth1 state execution function. * It involves state trie pruning mechanism triggered upon finality which requires state trie versioning based on the chain of beacon blocks.

Note:

Long periods of no finality can result in tons of garbage in the storage hence extra disk space consumption.

- When stateless execution and “block creation” is in place eth1 engine may optionally be run as pure state transition function with a bit of responsibility on top of that, i.e. state storage could be disabled reducing requirements to the disk space.
- It involves state trie pruning mechanism triggered upon finality which requires state trie versioning based on the chain of beacon blocks.

Note:

Long periods of no finality can result in tons of garbage in the storage hence extra disk space consumption.

- When stateless execution and “block creation” is in place eth1 engine may optionally be run as pure state transition function with a bit of responsibility on top of that, i.e. state storage could be disabled reducing requirements to the disk space.
- JSON-RPC support.

It is very important to preserve Ethereum JSON-RPC support for the sake of usability and adoption. This responsibility is going to be shared between eth2-client and eth1-engine as eth1-engine may

loose the ability to handle a subset of JSON-RPC endpoints on its own, e.g. those calls that are based on block numbers and hashes. This separation is to be figured out later.

Beacon block processing

ExecutableData

structure replaces Eth1Data

in the beacon block body. Also, synchronous processing of the beacon chain and eth1 allows for instant depositing. Therefore, deposits may be removed from beacon block body.

An updated beacon block body:

class ExecutableBeaconBlockBody(Container): randao_reveal: BLSSignature executable_data: ExecutableData # Eth1 executable data graffiti: Bytes32 # Arbitrary data # Operations proposer_slashings: List[ProposerSlashing, MAX_PROPOSER_SLASHINGS] attester_slashings: List[AttesterSlashing, MAX_ATTESTER_SLASHINGS] attestations: List[Attestation, MAX_ATTESTATIONS] voluntary_exits: List[SignedVoluntaryExit, MAX_VOLUNTARY_EXITS]

We modify process_block

function in the following way:

def process_block(state: BeaconState, block: BeaconBlock) -> None: process_block_header(state, block) process_randao(state, block.body) # process_eth1_data(state, block.body) used to be here process_operations(state, block.body) process_executable_data(state, block.body)

It is reasonable to process executable data after

process_operations

has been completed as there are many places where operation processing may invalidate entire block. Although, this approach may be suboptimal and leaves a room for client optimizations.

Accessing beacon state in EVM

We change semantics of BLOCKHASH

opcode that used to return eth1 block hashes. Instead, it returns beacon block roots. This allows for checking proofs for those data that were included into either beacon state or block starting from 256

slots ago up to the previous slot inclusive.

Asynchronous state read has a major drawback. A client has to wait for a block before it can create a transaction with the proof linked to that block or the state root it produces. Simply speaking, asynchronous state accesses are delayed by a slot at minimum.

Direct state access

Suppose, eth1-engine has access to the merkle tree representing entire beacon state. Then EVM may be featured with opcode READBEACONSTATEDATA(gindex)

providing direct access to any piece of the beacon state. This opcode has a couple of nice properties. First, the complexity of such read depends on gindex

value and easily computable and hence allows for easy reasoning about the gas price. Second, the size of returned data is 32 bytes which perfectly fits in EVM's 32-byte word.

With this opcode one may create a higher level library of beacon state accessors providing convenient API for smart contracts. For example:

v = create_validator_accessor(index) # creates an accessor v.get_balance() # returns balance of the validator v.is_slashed() # returns the value of slashed flag

This model gets rid of state access delay. Therefore, with proper ordering of beacon chain operations and eth1 execution (the latter follows the former), crosslinks to slot N-1

shard data becomes accessible in slot N

, allowing rollups to prove data inclusion in the fastest possible way.

Also, this approach reduces data and computation complexity of beacon state reads by avoiding the need of proofs that are sent over the wire and further validated by contracts.

Note:

it might worth making the semantics of READBEACONSTATEDATA

opcode independent from particular commitment scheme (that is, merkle tree) at the very beginning allowing for easy upgradability.

The cost of direct access increases eth1-engine complexity. Capability of reading beacon state may be implemented in different ways:

- Pass state along with executable data.

The main problem of this approach is handling state copies of a big size. It could work if direct access would be restricted to a subset of state data requiring small portion of state to be passed to the execution.

- Duplex communication channel.

Having a duplex channel, eth1-engine would be able to ask beacon node for pieces of the state requested by EVM synchronously. Depending on the way the channel is setup, the delays may become a bottleneck for execution of those transactions that have beacon state reads.

- Embedded eth1-engine.

If eth1-engine would be embedded into beacon node (e.g. as a shared library) it could read the state from the same memory space via a host function provided by the node.

Analysis

Network bandwidth

Current proposal enlarges beacon block by the size of executable data. Though, it potentially removes Deposit

operation as the proposal allows for advanced depositing schemes.

Depending on the block utilization, expected increase is between10 and 20 percents according toaverage eth1 block size which slightly affects network interface requirements.

It worth noting that if CALLDATA

is utilized by rollups then eth1 block size may grow up to 200kb

in the worst case (with 12M

gas limit) giving executable beacon block size around 300kb

with 60%

increase.

Block processing time

Average processing times look as follows:

Operation

Avg Time*

, ms

Beacon block

12

Epoch

64

Ethereum Mainnet Block

200

*

Lighthouse on Toledo with 16K validators and Go-ethereum on Mainnet with 12M gas limit.

It is difficult to reason about beacon chain processing times, especially, in the case with relatively large validator set and crosslink processing (since shards are rolled out). Perhaps, at some point epoch processing will take near the same time as eth1 execution.

Potential approach of reducing processing times at epoch boundary is to process epoch in advance without waiting for the beginning of the next slot in case when the last block of the epoch arrives in time.

Asynchronous state access model allows for yet another optimization. In this case process_executable_data

may be run in parallel with the main process_block

and even process_epoch

payloads.

Solidifying the design

One may say that current proposal sets execution model in stone and reduces the ability of introducing more executable shards once we need them.

On the other hand, several executable shards introduces problems like cross shard communication, sharing account space and some others that are not less important and difficult to solve than the expected shift in the execution model.