eth1+eth2 client relationship

Since Vitalik proposed an [Alternative proposal for early eth1 <-> eth2 merge

](https://ethresear.ch/t/alternative-proposal-for-early-eth1-eth2-merge/6666) in Dec 2019, there has been an active conversation about what this merger might look like from a software perspective and an eagerness to begin prototyping. The vision is a hybrid in which core consensus work is managed by an eth2-client

and state/block-production is managed by an eth1-engine

– together forming an eth1+eth2 client.

This document aims to make more explicit the separation of duties between an eth2-client and an adjunct eth1-engine to give better foundation to the conversation, spec writing, and prototyping. This document does not

aim to define specific details of the protocol (e.g. precise methods eth1 client calls to eth2 engine), and any examples contained within are only that – examples to aid description and subsequent discussion.

This document assumes basic familiarity with eth2 and stateless Ethereum.

High level separation of concerns

The aim in an eth1+eth2 merger is to leverage the existing eth1 state, ecosystem, and software in the upgraded eth2 consensus context.

Broadly, what we think of today as an eth2-client

handles core PoS and sharded consensus. Essentially, the eth2 protocol and eth2 clients have been designed to be really good at producing and coming to consensus on a bunch of “stuff”. That stuff is a number of shard chains full of data and (eventually) state. The eth2 “consensus layer” is much more sophisticated and much more complex than the PoW consensus layer found in Ethereum today.

Eth1 clients today have a relatively simple and thin consensus layer – there is only one chain and PoW handles most of the complexity in sophisticated extra-protocol hardware. Most of the sophistication and optimizations of eth1 clients lie in the user layer – state storage/management, state sync, virtual machine execution, transaction processing, transaction pools, etc.

This separation of concerns makes for a great pairing when eth1 is integrated as a shard on eth2 – the eth2-client can handle the complexities of PoS and sharded consensus, while the adjunct eth1-client becomes an eth1-engine

and can handle the complexities of state, transactions, VM, and everything closer to the user layer.

Minimal changes, local communications

There are many potential paths to how eth1 and eth2 client software can be woven together (full merge, import eth1 as library, communication protocol between the two), but in this document we focus on what we consider to be the most minimally invasive and most modular approach – a local communication protocol between an eth2-client and a stripped down eth1-engine

.

Given the diversity in implementation of both eth1 and eth2 clients, this approach prevents client software lockin on either side, allows for client teams to remain independent and focus on their own stack, and keeps the software projects largely stable to allow for rapid prototyping.

What does it look like

Broadly, an eth1+eth2 client looks like the following:

The eth2-client and eth1-engine are run together, locally communicating over RPC driven by the eth2-client.

Each maintains it’s own p2p interface, connecting to peers and handling a networking protocol related to each particular domain.

eth2-client

[

1752×900 27.2 KB

](https://ethresear.ch/uploads/default/original/2X/9/9ac774797d644c1c7d47fd6707470b609f62f8ce.png)

- Beacon Chain and Beacon State
- Core consensus objects upon which the rest of the system is built
- Core consensus objects upon which the rest of the system is built
- Shard Chains
- eth1 shard-chain
- Many data only shard-chains
- eth1 shard-chain
- Many data only shard-chains
- Operation Mempool [not pictured]
- Attestations, deposits, exits, etc
- Attestations, deposits, exits, etc
- P2P Interface
- Consensus level messages
- Includes

eth1 shard-block gossip

- Consensus level messages
- Includes

eth1 shard-block gossip

- RPC to eth1-engine
- All calls are driven by the eth2-client
- All calls are driven by the eth2-client

eth1-engine

[

1752×897 56.3 KB

](https://ethresear.ch/uploads/default/original/2X/4/478a1d2100be71cafc069014557d6fe070d8e25f.png)

- EVM
- Execution and validation of eth1 shard-blocks
- Execution and validation of eth1 shard-blocks
- eth1 State
- The user-level eth1 state as in Ethereum today
- The user-level eth1 state as in Ethereum today
- TX Mempool
- User-level transaction mempool, ready for block production
- User-level transaction mempool, ready for block production
- P2P Interface
- Transaction gossip as on Ethereum today

- State sync
- No

eth1 shard-block gossip

- Transaction gossip as on Ethereum today
- State sync
- No

eth1 shard-block gossip

- RPC from eth2-client
- All calls are driven by the eth2-client
- All calls are driven by the eth2-client

Consensus

From the perspective of the core consensus, the eth2-client is in charge and drives the building of the beacon chain, the data shard-chains, and the eth1 shard-chain. Any knowledge that the eth1-engine gains about the eth1 shard-chain and the core consensus (beacon chain/state) is provided directly by the eth2-client via RPC.

Specifically, an adjunct eth1-engine must

have access to an eth2-client as it does not maintain its own consensus. In PoW in Ethereum today, an eth1-client checks the proof-of-work, forms a block tree, and runs the fork choice rule to find the tip of the chain. In eth2, these mechanics are much different and require deep familiarity of eth2's core consensus – thus it is entirely offloaded to the eth2-client. The eth2-client provides the up to date information about the head of the eth1 shard-chain so that the eth1-engine can maintain an accurate view of the eth1 state.

Because the eth1-engine is entirely reliant upon the eth2-client driving the consensus, we suggest that the communications between an eth2-client and an eth1-engine are all methods on the eth1-engine called by the eth2-client (e.g. addBlock

, getBlockProposal

, etc). This enforces a leader/follower relationship to reduce the complexity in reasoning about the system and to limit the business logic required in the eth1-engine.

From the perspective of the eth2-client and core consensus, the eth1 shard-chain is handled almost exactly the same as all of the other shard-chains (fork choice, crosslinks, block structure, signatures, etc). The primary difference is that the shard block contents can be executed against the eth1-engine and thus the eth1 shard-block data must be well-formed with respect to eth1 and has additional validation against this successful execution.

State

eth2 has a state related to the core consensus. This is called the "beacon-state". The beacon-state is thin (~10-40MB depending on the size of the validator set) and contains all of the information needed to understand the core consensus and how to process the shard chains. In fact, to process the consensus related portions of a shard-chain, a client must

have access to the beacon-state (e.g. recent crosslinks to run the shard-chain fork choice, current validator-set/shuffling to validate shard-chain signatures).

eth2's state does not reach all the way into user level state. The most it makes claims about is the availability of shard-chain data. Within this shard-chain data lies the actual user level state root, and in the case of the eth1 shard-chain, the current Ethereum user state root.

The following discuss the different levels of eth1 state in relation to an eth2-client:

eth2-client without

eth1-engine

The core eth2 protocol can

be run without an adjunct eth1-engine. An eth2-client alone can follow the beacon chain and follow the shard-chains (including the eth1 shard-chain). Without an eth1-engine, the client cannot execute the stateless eth1 shard-blocks and so

cannot fully validate them or get any useful user-level info out of them. Still, the head of the eth1 shard-chain can safely be found according to the assumptions made about the eth2 core consensus and validators.

eth2-client with stateless

eth1-engine

To run a validator, an eth2-client must be run with an adjunct eth1-engine. This can be done in a stateless manner (not storing entire eth1 state locally) because eth1 shard-blocks have witnesses available for execution. Beacon committees can check the availability of shard-block data and the validity of that data with respect to eth1 via a stateless call to the eth1-engine checking the validity of the format and execution of the block.

Other than validators, many user/application nodes might also run with a stateless or semi-stateful eth1-engine. Using the thin eth2-client to follow the head of the eth1 shard-chain and interact with it in a stateless or semi-stateless manner.

eth2-client with stateful

eth1-engine

To run a validator that can produce eth1 shard-blocks, the eth2 protocol must be run with the adjunct eth1-engine along with the full eth1 state (there are stateless block production methods being explored, but for simplicity, we leave that out of this initial discussion). The local state and tx mempool can then be used to form new, valid blocks on demand (discussed more below).

Other than validators, many user/application nodes might also run with a fully stateful eth1-engine – e.g. block explorers, archive nodes, state providers, etc.

Networking

For simplicity, eth2 and eth1 initially maintain their independent networking stacks and protocols. Some existing eth1 protocols are deprecated in favor of the eth2 protocol in response to a shift in responsibility (e.g. eth1 shard-block gossip). A migration of eth1 protocols to libp2p might be favored after the initial prototyping phase, or further down the line, to unify the networking stack, but it is not a requirement.

eth2-client and eth1-engine have access to the same discv5 DHT, but independently find peers of appropriate capabilities and independently maintain connections.

ENR

An eth1+eth2 client utilizes a single ENR because the node sits behind one logical network identity with multiple capabilities.

eth1 capability (state, transactions, etc) is signified with the existing eth

(or maybe new eth1

) key in the ENR.

eth2 capability (core consensus) is signified with the eth2

key in the ENR.

The existence of each signifies the node’s ability and willingness to speak the class of underlying network protocols.

Wire protocols

eth2 protocols

- eth2 req/resp
- Status
- Beacon block sync
- Shard block sync
- Status
- Beacon block sync
- Shard block sync

- Core consensus gossip
- Beacon blocks
- Attestations
- Shard blocks (including eth1 shard)
- Other validator operations
- Beacon blocks
- Attestations
- Shard blocks (including eth1 shard)
- Other validator operations

eth1 protocols

- Subset of eth1 wire protocol
- Transaction gossip
- Sync methods (getnodedata or new methods)
- Get receipts
- Transaction gossip
- Sync methods (getnodedata or new methods)
- Get receipts
- NOT
- messages related to block hashes, headers, or bodies
- messages related to block hashes, headers, or bodies

Why eth2-client handle eth1 block gossip?

eth2 is designed to generically handle the production, gossip, and validation of shard-blocks. We aim to make the eth1 shard as standard and comformant to the rest of the shards as possible. With respect to the core consensus, the main difference eth1 blocks have compared to the rest of the shards is the ability to execute/validate the contents of the block against the eth1 engine.

When a validator is working to crosslink an eth1 shard-block into the beacon-chain, the eth2-client would make an additional call the eth1-engine to execute and validate the block.

When a stateful eth1+eth2 node receives a new eth1 shard-block, the eth2-client would make an additional call to the eth1-engine to validate the block and update the local state storage.

Transaction gossip and mempool

The eth1-engine maintains user transaction gossip and the eth1 transaction mempool in almost the exact same way as in Ethereum today. The same network protocols and local mechanics can be used to gossip and maintain the pool, ready for block production.

The primary difference is how knowledge of spent transactions is ascertained and how the pool is utilized for block production, but these are arguably in a layer right outside of the pool.

eth1 shard-blocks are provided to the eth1-engine from the adjunct eth2-client. Transactions included in such blocks should be cleaned from the mempool in a similar way to Ethereum mainnet PoW blocks today.

eth1 shard-blocks are produced on demand from the adjunct eth2-client via the contents of the mempool. This RPC method and the underlying functionality is similar to getWork

but would return the full block contents rather than just a hash.

Block production

Within the eth2 protocol, all blocks (beacon, shard, eth1-shard) must be produced and signed by a PoS validator from the core consensus. To this end, the eth2-client is ultimately responsible for all

block production.

For beacon blocks and non-eth1 shard-blocks, the eth2-client has everything it needs to produce valid blocks.

For the eth1 shard-blocks, the eth2-client does not have immediate/ready access to eth1 state, transactions, and other underlying eth1 structures to produce a valid block. Instead, when a validator is assigned to produce an eth1 block, the eth2 client requests a viable eth1 block data (TXs, state root, etc) from the eth1-engine. The eth2-client then bundles this eth1 block data into a full shard-block (adds slot, proposer_index, proposer_signature, etc) and broadcasts the block to the network.

The eth1-engine is able to produce valid/viable eth1 block data because it manages the eth1 transaction mempool in the same way that it does on Ethereum mainnet today, and it maintains up-to-date info on the head eth1 state via updates from the eth2-client.

Next steps

If this general design is agreed upon, the next steps include

- Ensure assumptions about eth2-client driving eth1-engine are in harmony with and do not place unexpected burden on existing eth1 software.
- Define more explicitly the communication protocol for driving the eth1-engine – e.g. new_head(block)

, validate_block_transition(block)

, get_proposal(parent_root)

, etc

- Define networking components – e.g. which subset of eth1 protocol is needed, how specifically will ENRs work for discovery
- Extend Phase 1 eth2 spec to add in eth1-capable

flag to validators for eth1 block production. (A Phase 1 eth2 client can be used to start and just assume all validators are eth1-capable

)

- Prototype!

Any and all input and feedback is appreciated both here and in the[Eth R&D eth1+eth2 merge channel](#).