Validator Client Design

Status:Draft
Collaborators: Preston, Raul, Terence
Last Updated: 2018-11-26
Prysmatic Labs

Objective

This document aims to outline a formal design proposal for a validator client within the Ethereum 2.0 ecosystem. While this proposal specifically targets the Prysm client, the API and overall architecture should be a standard for other clients.

Background

The background of this design documents comes from the Client Architecture discussions done by the implementation teams in Prague pre-Devcon. The goal of the discussions were to figure out how to structure a beacon node implementation such that it could accommodate the Serenity roadmap including shards, validators performing their responsibilities, tracking PoW chain logs, and advancing a full PoS random beacon blockchain.

There have been a few discussions on the topic but none have really defined strict responsibilities for the different parts of the system.

Overview

The sections below will clearly define the roles of beacon nodes and validator clients

Beacon Node vs. Validator Responsibilities

Beacon Node

- Stores the canonical state of Serenity containing the validator set, cross-link information, and processed attestations
- Stores the beacon blockchain that is advanced by validators in the network proposing and attesting to beacon blocks
- Has a strong sense of time, validated with an NTP server, to track beacon slots
- Handles a set of peers in the network for initial and incoming sync and propagating blocks and attestations
- Has a gRPC server clients can connect to and provides a public API for Serenity

- Streams state and block-related information via event feeds

Validator Client

- Piece of software users interact with to stake ETH to secure the Serenity network
- Needs to store several important secrets: (1) RANDAO reveal, (2) Proof of Custody for shard data, (3) BLS private keys or provide interface to a hardware keystore
- Needs to track the following data: (1) shard data for phase 2 (state execution), (2) data blobs the validator has signed
- Does not need a strong sense of time (no internal clock), as new slots can be received via event streams from a beacon node
- Should have the ability to swap underlying beacon nodes efficiently based on users' whims to another beacon node that implements the Serenity protocol
- During its proposal slot, its sends a proposal request to the beacon-node which will use the request to run a state transition with the proposed block and then return the tree hash of the state.
- Then the validator client signs the proposal_signed data and sends the block to the beacon-node to be broadcasted.

Requirements for Validator Client

Validator client needs to listen to the following streams from the beacon node:

- `current_slot() -> slot`
    - Change signals for validator to perform duties
- `last_state_recalculation_slot() -> slot`
    - Change signals to validator to check committee and shard assignments

The Following is Taken from Danny Ryan's HackMD Document

The first thing a validator client does upon first starting is checking if a state recalculation has occurred:

- Validator updates local knowledge of duties if state recalc occurred
    - calls `committee_shard(validator_index)`
        - stores this `shard_id` for future attestation use
    - sync the shard corresponding to the received shard_id
    - calls `committee_slot(validator_index)`
        - stores this slot number for future attestation trigger
    - calls `beacon_proposer(slot)` on the committee slot
        - notes if is the beacon proposer as well as attester at that slot
- Validator continues to receive slot updates via `current_slot()`…
- Validator reaches committee slot previous noted
    - Proposer
        - calls `beacon_proposer(current_slot)`

        - sees that is the proposer
    - calls `beacon_block_candidate(current_slot)`
    - checks that `beacon_block_data` is not slashable wrt previously signed data
    - signs `beacon_block_data` to create `block_sig`
    - calls `submit_beacon_block(beacon_block_data, block_sig)`
- Attester
    - calls `attestation_candidate(current_slot, shard_id)`
        - uses previously stored `shard_id` from call to `committee_shard`
        - node knows about block just proposed so this will be the block in the attestation data
    - checks that `attestation_data` is not slashable wrt to previous signed data
    - signs `attestation_data` to create `attestation_sig`
    - calls `submit_attestation(attestation_data, validator_index, attestation_sig)`

The validator client submits the following messages to the beacon node:

- `submit_beacon_block(block_data, sig) -> status`
- `submit_attestation(attestation_data, validator_index, sig) -> status`
- `submit_shard_block(block_data, sig) -> status`

Requirements for Syncing Shard Chains

Responsibility Proposals

- Beacon chain syncs everything, validator has no local shard sync data data, validator sync via RPC
- Beacon chain syncs everything, validator stores local shard sync data, validator sync via RPC
- Beacon chain syncs only beacon chain, validator syncs the chains they need via peers

Detailed Design

Beacon Chain Fully-Managed Shard Syncing Approach, Validators Store No Shard Sync Data

The idea behind this approach is having validator clients as simple entities connected to beacon nodes via RPC.

Pros

- This is simpler as it restricts the validator client to simply worry about managing its secrets and signing data. It's associated beacon node will have to do all the heavy lifting.

Cons

- This makes a validator client tightly coupled to a beacon node that holds its data
- Places a lot of responsibility on beacon node in terms of resource requirements, and an upper cap on how many shards a single beacon node will be able to handle at once
- This approach does not really warrant the need for a validator client by itself if it is so tightly coupled it does not make sense for the validator to be a separate piece of software
- Prevents swapping of underlying beacon node
- Makes a multi-tenant setup harder as the beacon node will have to sync more shard information

Beacon Chain Managed Shard Syncing Approach, Validators Request Sync Data via RPC

This approach entails a validator client acting as a sidecar and requesting its underlying beacon node to sync shard data and hand it off to the client via RPC. The beacon node then deletes the data once the validator has received it.

Pros

- Compared to the first approach the resource requirements for running a beacon node will be lower, as shard state data is not persisted and is instead deleted after passing it on to the validator client.

Cons

- If for example, a beacon node receives a request to sync shard 5 from validator client A, retrieves the sync data via p2p and then hands it off to client A and deletes it locally after a certain time. Then, validator client B connects after a while and also needs shard 5, so beacon node will have to sync again and do the same inefficient process
    - This leads to a lot of complex interactions in how to manage the lifecycle of shard sync and tightly couples responsibilities between the two processes
    - The implementation of this approach is very non-trivial

Validator Managed Shard Syncing Approach

This approach defines validator clients and beacon nodes as two separate responsibilities, each connected to a respective list of peers to handle sync for beacon blocks and shard blocks. In this approach, a beacon node will handle a Casper Proof of Stake set of validators shuffled every epoch to perform proposal/attestation responsibilities in shard committees. It serves as the "heartbeat" of Ethereum 2.0, giving a single source of truth for shuffling and cross-links in the beacon chain. Validator clients can easily connect to this node via RPC to receive real-time,

streaming updates on the beacon chain. They can then sync the shards they need from other validator clients via RPC.

https://github.com/ethresearch/p2p/issues/5
A potential concern with this approach is the idea of having two types of “nodes” in ETH 2.0. We argue this is more about semantics than anything else, as we can still have the validator client and beacon node bundled together in our release and allow users to create third-party, end-user validator managers that are simple pieces of software with a good UI that allows them to visualize what their beacon/validators are doing underneath the hood.

Pros

- Clear separation of concerns - beacon nodes handle the beacon chain, validators handle syncing the shard chains they have to participate in
- Easier multi-tenant setup: that is, a validator client is not bound to any particular underlying beacon node and can be easily swapped out if needed
    - This allows for setups with a lighthouse beacon node and a prysm validator client or viceversa depending on personal preference
- Easier onboarding for phase 1 and 2 as validators will have the synced shard data they need for state execution
- The idea of a validator client managing shard sync fits nicely with the idea of a proof of custody - giving a validator more responsibility in terms of what it needs to sync and store

Cons

- Validator clients cannot be“super lightweight” machines in terms of resource utilization as in approach 1. This is also not much of a con as users will need to run their beacon node anyways, so the previous approaches also suffer from this. This can be mitigated, however, by having a third piece of software that is more user-facing and allows for easy management of a validator clients from a low-resource machine.

A potential argument against this is how a validator client touching the peer-to-peer network is an attack vector. However, the same can be said for the previous approach as the validator would be tightly coupled with a beacon node which would also be touching the peer-to-peer network. By proxy, all approaches have that security fault and there should be other ways of mitigating it.

Beacon Node: handles validators, cross-links, shuffling, syncs beacon blocks via p2p, provides RPC endpoint
^
|
Validator Node: connects to beacon node to obtain shuffling and committee info, syncs shard chains via p2p, proposes/attests to shard data, keeps track of signed data and secrets
^
|

End-User Serenity Client: end-user front-end to manage underlying beacon/validator node interactions in a multi-tenant setup

Further Reading

[Danny Ryan's Minimum validator interface document]