Background

Recently, there's been a lot of buzz about the

validator

timing game. In this post, I want to delve into a different twist on this topic, exploring the interplay between consensus and execution clients with

Execution API. I'll walk you through various scenarios, showing how this game unfolds and why sometimes your validator might miss out on an attestation head vote, or in more extreme cases, even a block. It's important to note that what I'm describing here is specifically tied to the Prysm Consensus Layer Client. This might not hold for other clients, but that's the beauty of client diversity!

CL <> EL Interactions

First, let's dive into how the CL (

consensus layer) and EL (

execution layer) work together after

the merge. Your CL client has a few key tasks for the EL client, plus an extra request if it's proposing a block in the next slot. Here's what it boils down to:

1. CL client asks the EL client, "Can you validate if the execution part of this block (execution payload) is valid?"
2. Then CL client asks the EL client, "Can you make this block the new head? (ie ForkchoiceUpdate = FCU)"
3. If the CL client has a validator proposing the next slot, it adds, "And can you also begin preparing me an execution payload built on top of this new head? (ie FCU + attribute)"

CL Validations

Here's the key part: if the EL returns with an error at any point in this process, the block is a no-go. It doesn't matter what the consensus part of the block says - if the EL isn't happy, it's game over for that block.

Secondly, we will examine the operations conducted independently by a typical CL client, without interactions with the EL client. These operations include:

1. Validation of the block following gossip protocol rules before broadcast among peers.
2. Validation of the block's consensus component. (ie. attestations, deposits... etc)
3. Determination of the new head within the existing block tree, considering the new block and its attestations.
4. Update some caches.
5. Add block and state to local storage.

Validation of the block following gossip protocol rules before broadcast among peers.

Validation of the block's consensus component. (ie. attestations, deposits... etc)

Determination of the new head within the existing block tree, considering the new block and its attestations.

Update some caches.

Add block and state to local storage.

Regarding the computationally and latency-intensive part of these processes, they are:

1. Calculation of the hash tree root of the beacon state.
2. Persistent storage of the block.
3. Copy of the beacon state.
4. Shuffling of validator indices.
5. General signature verifications.

Calculation of the hash tree root of the beacon state.

Persistent storage of the block.

Copy of the beacon state.

Shuffling of validator indices.

General signature verifications.

Putting Everything Together

Tasks 1 through 3 are mandatory for each slot, while task 4 is required once at the end of each epoch. Task 5 is optimized through aggregation. Client implementations employ many optimizations. One example is performing these resource-intensive tasks during minimal contention (the latter half of the slot). This scheduling is strategic, and blocks typically arrive in the first half of the slot. Therefore, the state required for the subsequent slot is precomputed towards post-block processing. This approach ensures prompt processing of incoming blocks and encapsulates the essence of the "update some caches" operation.

Now that we've got the basics down let's look at how the Prysm CL client handles different situations based on when the block is processed and whether the block is head.

1. Block processed on time and is the head block
2. Block processed on time and is not the head block
3. Block was processed late and is the head block
4. Are you proposing the next slot?
5. Is this the last slot of the epoch?

Block processed on time and is the head block

Block processed on time and is not the head block

Block was processed late and is the head block

Are you proposing the next slot?

Is this the last slot of the epoch?

Block is processed on time and is the head block

Let's explore the happy scenario where the block is validated by consensus, execution, and data availability (in Deneb). The fork choice successfully updates the head and extends the local fork choice view with the new block and its attestations. After validations and update fork choice, the block is still on time (within 4s) and Prysm CL client is not lined up to propose the next slot, it will send an FCU to the execution layer client. After sending FCU, Prysm updates its caches for the upcoming slot.

What happens when a Prysm validator is set to propose a block next slot in this happy case scenario? After updating the cache for the next slot, the node sends a second FCU + payload attribute. Why two FCUs are sent? One with an attribute and the other without. The rationale is straightforward. The first FCU is tailored to enhance attester duty, providing attesters with the most current head information as quickly as possible. However, to determine a payload attribute for the proposer, it's necessary to compute the cache for the next slot. This cache computation is mandatory for proposers, but irrelevant to attesters. Therefore, the first FCU focuses on optimizing attester duty, while the subsequent FCU + attribute, fulfills the requirements of the proposer duty.

Block is processed on time and is not the head block

What if the block is processed on time, yet the head block differs from the incoming block? We're not updating the cache for the next slot in such instances. Updating the cache becomes redundant since any activity in the next slot won't utilize a non-canonical post state from the node's perspective. The Prysm CL client incorporates a separate background routine that performs cache updates at the 4s mark under the condition that the head slot diverges from the current slot. Note that in this scenario, block is not head, and block never arrives is the same! Additionally, if the current slot block has never arrived, the Prysm CL client will send an FCU + attribute for the proposer if it proposes the next slot.

Block is processed late

Lastly, consider the situation where the block is processed late. In such a case, there's no necessity for optimization or

dividing the process into two separate FCU calls. We could utilize a single FCU + attribute if there's a proposer for the next slot. Conversely, we'd use the FCU without the attribute if there's no proposer. This approach is straightforward to reason. However, it's crucial to note that we will only invoke the FCU + attribute if it's the new head. Given that a late block lacks a proposer boost, and certain client implementations may intentionally reorg such a late block, we will likely use the same FCU + attribute, as mentioned previously in the separate routine. The key takeaway here is that when the block is delayed, there is no necessity for two FCUs.

Finally

Having explored most of the scenarios, let me add a few more points. In the context of Deneb, fulfilling data availability requirements is a prerequisite before a client can update the head. This requirement will inevitably extend the process. It's a situation that continues to warrant close observation. Also, updating the cache in an epoch's last slot might take longer than other slots. This is primarily because there are additional processes to handle across the epoch boundary, such as shuffling. Some client implementations, such as Prysm and Lighthouse, will intentionally reorg late blocks. That sums it up! I hope this elucidates things clearly, offering a fresh perspective on how the consensus client functions and interacts with the execution client, all while fine-tuning for the validator client.

Thanks Potuz for the review