In the Beginning…

Vitalik's 2nd proposal solidified the idea of Execution Environment

s. Lets take a look at some of the initial changes it brought to the beacon chain (for now we'll completely

ignore the shards).

First, before the Execution Environment

proposal, the beacon chain looked something like this (fields omitted). Thanks to Casey for providing these initial diagrams:

[

1621×704 57.8 KB

](https://ethresear.ch/uploads/default/original/2X/0/0a8aec46443ac1f8a7eb65c639eaf4d2b59d82c6.png)

Similar to eth1, validators maintain basic account fields. Lets define this as the eth2 validator account tree. These accounts do not support code or storage roots.

Enter Execution Environments

Execution Environment

s came into the mix with Vitalk's phase 2 proposal. Without thinking of the shards, the beacon state could be updated as follows:

[

1722×689 80.7 KB

](https://ethresear.ch/uploads/default/original/2X/9/9c6d2614d72049e08d320e01eefde07f00a22a53.png)

We simply added the execution_environments

field to the Beacon State. To deploy a new Execution Environment

, we introduce a new transaction to the Beacon Chain, NewExecutionScript

(or a new Beacon Chain Operation to stay true to the current specifications). It could look like the following:

{ "sender": uint64, "slot": uint64, "code": bytes "pubkey": BLSPubkey, "signature": BLSSignature }

There was one major flaw to this method. Execution Environment

s held their balances on the beacon chain. This means Execution Environments

could not send eth between each other without putting load on the beacon chain (since naturally balances would have to be updated on the beacon chain). Lets continue the discussion.

Tracking ETH on Shards

Vitalik released a new proposal at Devcon, "Eth2 Shard Chain Simplification Proposal". It introduced an "operating system" by which Execution Environment

s could pass balances across shards to each other. Here was his associated diagram:

[

671×776 16.3 KB

](https://ethresear.ch/uploads/default/original/2X/f/f1a9cd71c079cd7ae20d2cd460a4fad311a11f56.png)

To add clarity to this diagram, the operating system defined how EEs pass eth between each other across shards. First of all, each EE would have a balance on each shard, ExecEnvID -> (state_hash, balance)

. To pass eth between Execution Environment s within a shard, a host function would just update the balance

field. To pass Eth across shards, a receipt root would be generated, shard -> [[EE_id, value, msg_hash]...]

. Details on how the receipt roots would arrive on different shards were left to the reader. Also, it seemed the overhead would limit

the amount of cross-shard transfers.

Next, we needed to more concretely define the details behind a system like this. Vitalik[released a problem statement].

At this point, lets take a step back and analyze where we have come to.

1. EEs hold a balance

and state_hash

on each shard

1. An operating system needs to be established to prevent leakage. AKA the protocol needs to make sure balance

or eth

can be passed across shards without eth

being lost or printed.

We began to realize we're basically defining an account model for Execution Environment

s. Lets first take a look at what the account model looks like in Eth1 and use a simple ERC20 contract as an example:

[

1012×1151 119 KB

](https://ethresear.ch/uploads/default/original/2X/4/4f1fbc271448555456f77d9820d5a9046d29d5e0.png)

An Execution Environment

's place is not so different within a shard - especially now that we store balance

and state_hash

on the shard. We'd likely see an almost identical account model within Eth2's shards:

[

2335×813 97.3 KB

](https://ethresear.ch/uploads/default/original/2X/4/43ccc0254bbc2abe4cfefc0e4f45db05c9868c92.png)

Whether the E2contract

leaf is on the beacon chain or not, the core of the E2ContractState

does not seem to be too different than what we have on eth1. There is a wasm code hash, balance, and state root. So lets view Execution Environment

s through a different lens. They seem to just represent top level contracts or accounts within the shards. This puts a new lens/image on the entire cross-shard discussion and problem statement. Lets review the points made earlier:

1. EEs hold a balance

and state_hash

on each shard

1. An operating system needs to be established to prevent leakage. AKA the protocol needs to make sure balance

or eth

can be passed across shards without eth

being lost or printed.

is just enshrining a transaction model or way to transfer eth between accounts, not too different than how eth1 has an enshrined transaction model. We'd also need a standardized serialization format (hint: SSZ!). We'd probably need a way for EEs to call into each other programatically and send eth between each other (hint: CALL Opcode). Vitalik takes a stab at how that may be done via a netting model and what is called a "meta-execution EE". Casey puts together a very clear explainer of what the netting model is. We put togethersome scribbles on how this may be put in action (don't let this confuse you too much - you can just keep following this writeup). The conclusion is that we're basically defining a core protocol layer - a transaction and account model for Execution Environment

s or what we can call as simply Top Level Contracts

. For the remainder of this writeup, I'll use these terms interchangeably.

Native Integration of Eth1 Account Model

So lets take this conversation forward another step. If we're already enshrining a root level contract or account model, what is the point of Execution Environment

s at all? Why not just have contract wallets, ERC20 contracts, exchange contracts and everything else live at this root protocol layer - similar to eth1? I'm putting the eth1 account model below again just to remind you:

[

1012×1151 119 KB

](https://ethresear.ch/uploads/default/original/2X/4/4f1fbc271448555456f77d9820d5a9046d29d5e0.png)

So the big question is, how are Execution Environment

s actually different than an eth1 contract since here has to be an enshrined transaction, serialization format and call/send method anywyas? Simply put, it's not that different (lets ignore whether the wasm code is stored on the beacon chain or shard for now). However, in the direction of eth2, we wanted to introduce generality and a light core protocol layer with a unique degree of abstraction. Lets add a few more diagrams here:

Here we assume each EE gets a 32 byte state root in the shard. This state root may actually be stateful on the shard. We will cover the scenario where it isn't in a moment. For now, lets assume the shard stores this 32 byte state root (on the shard state or via a cache

works just fine). When a validator syncs to the shard, it downloads these state roots.

So why doesn't it just transform into the following?

Why don't users just manage their accounts or contract wallets at this top level? All the tools that are necessary exist here (wasm code, balance, transferrability). The answer is simple. We want to keep the core eth2 protocol light and unopinionated. There are several different ways of designing execution systems and instead of choosing or locking into one, Eth2 may allow for multiple to be designed and experimented with in parallel. The effect can be seen in the following diagram:

[

1418×321 6.8 KB

](https://ethresear.ch/uploads/default/original/2X/3/30d34dec8b7bfb076f1c210343ca6e8207077976.png)

Maybe EE1

follows a hexary-patricia tree. Maybe EE2

follows a sparse merkle tree. Regardless, you see that contracts (as we know them today in eth1) get pushed down one level in the tree of accounts. They get pushed down as child contracts.

What effects does this have?

Effects of a Minimal Protocol

Governance

An Execution Environment (or top level contract) will likely have to set its own governance rules. In eth1, if we needed to change the transaction format, account model, accumulator for the patricia tree, storage methods, or state rent, we would need to fork the core protocol.

Initially, this could also be done in eth2. Hard forks could change the wasm code for deployed EEs, adjusting its rules as desired. Long term however, in a world of multiple EEs, governance and upgrade patterns could no longer rely on system wide forks. Instead, EEs would have to choose whether to include ways to upgrade its own code - e.g. via coin voting, staking, or a multisig - or to remain immutable forever. Additionally, contracts could choose to offer migration functionalities to move to an upgraded new EE, e.g. via yanking patterns.

State Rent

Each EE or account model may present different pros/cons. Maybe EE1 provides limited state but no state rent. Maybe EE2 provides limitless state but it costs more and integrates a state rent model. Maybe EE1 restricts the total state to 1TB (these would all have direct effects on state provider pricing, etc.).

Ecosystem Funds

Each EE could have its own ecosystem fund. EE1 may say 0.5% of its gas fees get routed to an ecosystem fund. This ecosystem fund may then be routed towards developers who are working around/in this EE.

Shard Presence

EEs could reduce their shard footprint and only exist on a range of shards. They may have governance models in place to expand those over time. This may in effect decrease fees to use this EE in the short term until it expands and grows.

Accumulator

EEs would define their own accumulators with what fits in their model.

Transaction Model

EEs would define their own transaction structure and fields.

Serialization

Other serialization methods may show promise over SSZ that the core protocol provides. Leaving this in the hands of the EE is interesting.

Cross Shard Communication

With ETH as the native protocol currency, some enshrined method for cross shard ETH transfers will be necessary. Beyond that however, EEs could be free to choose their own communication standards such as bitfield or queue/nonce models for receipt processing.

The TL;DR is that this model seems to push innovation/adptation and encourages protocol developers to use Eth2 as a base protocol to try new models & approaches on execution frameworks. By pushing the complexity down a level in the tree, we do not

couple the protocol to these adaptations. Eth2 developers do not have to take opinions on ecosystem funds, state rent or areas of governance. If we want to try new transaction formats or serialization formats, we get to do this and we don’t need to fork the protocol! Eth2 seems to become a foundation for major innovation in the blockchain/cryptography space with a fluid way to advance & adapt over the years.

In order to accomplish all of this, we need to introduce an incentive to push this complexity down a level in the tree. We do this via simply making it quite expensive

to deploy an Execution Environment

or Top Level Contract

. We suggest a model where a fee (say 500 eth) has to be locked up (aka a rent by deposit model), but there are many incentive mechanisms that can be evaluated, and these can be adjusted over time.

Enshrined Execution Model

So lets continue to move forward. What happens if instead of following this tiered model (or having Execution Environments), we just enshrine one model to the protocol to start off with? Well, we already talked about how the protocol is already

doing that for the top level or for Execution Environment

s. So lets explore the route where Eth2 essentially becomes a modernized eth1 - we likely add needed host functions for account abstraction, shard awareness, cross shard transactions, etc…

Our regular contracts (wallet contracts etc.) all get thrusted to the top tier of the tree. The roots for each of these may be stateless. They could also be stateful if we choose (or driven by a cache).

If we move in this direction, this means deploying at the top layer of the tree would be inexpensive. We would also be enshrining one accumulator, account model, execution model etc. into eth2. We basically have an upgraded eth1 that is shard aware. This approach may be interesting, because of reduced complexity. However, what drawbacks may this have?

1. Language-specific implementation for each client. Each client would likely build this logic in their own language
2. Any changes to the account model, accumulators, transaction model, etc. require forking the core protocol
3. State rent, state size limits needs to be coupled to the core protocol
4. We risk stunting innovation on the EE front (there are multiple efforts right now building an ORU, eth2 EE, ZK rollup EE, mixers, etc.)
5. We risk locking/coupling the protocol to one model and a more general/iterative model with parallel efforts is stunted

It may be argued, that this approach is not that

different and that we can still accomplish an EE-like model in this setup. Essentially, we can still push abstraction and contract frameworks in a model like this:

[

1015×323 5.95 KB

](https://ethresear.ch/uploads/default/original/2X/c/c3b3767568f2daf045dd455ca9caa7254ae8c203.png)

However, encapsulation and separation of concerns with the core layer seem to introduce issues. This new contract framework under experimentation may have general caveats. For example, if there is a different governance rule, state rent scheme, upgrade plan, etc. then it may be risky to encapsulate your logic in this child contract framework (when the top level provides functionality). For example, a contract may instead just decide to use storage at the top level.

[

1247×321 15.4 KB

](https://ethresear.ch/uploads/default/original/2X/9/93bc423b9dac4e283aa0b78d9393d86403f0ca20.png)

There are many other examples that can be used on breaking encapsulation. Forking the protocol may have an effect on these child contracts (or no effect and couples governance/updates). Why even go with the trouble of pushing abstraction down another level - when it appears to introduce more conplexity on top of what the protocol already has? How hard will it be to in the future decouple transaction, account logic etc. from the core, eth2 protocol and transition it to a lighter model? In reality, it appears this encapsulation and world of child contracts is discouraged in this model.

However, maybe we can accomplish the same result of simplicity and hyperfocus on just one account model to start and still get the benefits of a light, core protocol that pushes abstraction down a level? Perhaps we can have a phased, phase 2.

We go back to our original diagram where it is expensive to deploy a top level contract

or Execution Envrionment

in the higher tier of the tree.

[

1418×321 6.8 KB

](https://ethresear.ch/uploads/default/original/2X/3/30d34dec8b7bfb076f1c210343ca6e8207077976.png)

We hyperfocus on the development of EE1

in that diagram. Upcoming testnets and design choices still allow multiple top level contract s to form (or Execution Environment

s), but we take an opinion that the initial launch of phase 2 may not

allow for additional deployments until we are ready. It looks like this instead:

In the meantime, our testnets allow for multiple Execution Environment

s and we can experiment with how Execution Envrionment

s communicate with each other.

So what is the biggest difference here? It's that the logic for EE1

is defined entirely in wasm. Our roadmap still optimizes for a world of multiple Execution Environment

s and a light, general protocol. But the actual amount of work and complexity does not

increase from an enshrined model. We don't have to take a concrete opinion on cross-EE communication. We can iterate on that as we see more activity in our testnets.

How do we upgrade this Execution Environment

or EE1

?

In the early stages, we can choose to update one wasm script across all clients

and fork the protocol to update that. Once we find stability, we can expand deployments for multiple EEs and no longer require forks to upgrade the canonical EE.

Note: this scenario would not preclude us from launching several EEs from the start. We would even prefer to have more than just the one canonical EE ready for the phase 2 launch. However, the deployment of custom EEs may

be restricted in the beginning and only be opened up at a later time.

Cross Shard Transactions

So lets continue the discussion from where we left off earlier. Vitalik framed a model to pass eth between EEs and across shards to prevent eth leakage aka meta-ee discussion. We continue to build into our model (at the core protocol) a method like this (netting scheme). But nailing the details on the cross-EE discussion isn't as important right away. However, in a one EE model or multiple EE model, we still need to decide how the protocol supports cross shard transactions.

The netting scheme (linked above) is somewhat orthogonal to the discussion of transferring arbitrary data across shards (hint: receipts). The main question we need to decide: should the protocol offer a built in mechanism for cross-shard data transfers, aka an enshrined receipt system?

Having such a system provided by the protocol layer could simplify cross shard communication from the EE perspective, as the EE would not have to deal with most of the associated complexity (e.g. replay protection, ordering, …). This would however come at the cost of moving that complexity to the protocol, which goes against the general eth2 design goal of having a slim, minimal base layer. In particular, questions around guaranteed delivery would be challenging to answer:

- Does the protocol guarantee execution of a cross-shard transaction on the receiving shard?
- How would fee payment work under such a system?
- What if a shard would receive a high number of incoming transactions from multiple other shards?
- If delivery was not guaranteed by the protocol, who would be responsible for processing the incoming transactions?

We believe avoiding this complexity on the base layer fits better with the overall eth2 philosophy. Thus, it should be up to the Execution Environment

to implement cross-shard communication patterns such as through a bitfield/receipt mechanism, nonce/queue model, or via object capabilites with replay protection. This takes an overall passive

approach and supports models as described here.

However, we may find that certain properties we come across (such as in the object capabilities model), make sense to introduce supporting host functionality. For example, this functionality could enable the protocol to provide proper error or success handling in communication between EEs.

State Provider Network or Witness Protocol

One of the main research questions is regarding Static State Access (SSA) vs. Dynamic State Access (DSA).This writeup describes DSA & SSA. Here was an early overview and writeup on state provider networks in eth2 and how DSA/SSA affects the conversation. Since this writeup, we've somewhat shifted our hypothesis towards recommending an SSA model. It could offer major benefits by introducing a more elegant model around memory/storage which somewhat mimics Rust style ownership/borrowing syntax. Eth1 currently operates on a DSA model, but our early research suggests that moving to an SSA model could be possibe without loss of functionality (including for DeFi applications). We have been extending current solidity tooling with taint analysis in order to detect DSA in an automated format and foresee the necessary tooling for contract developers. If we can accomplish SSA properly, we will likely see significantly reduced complexity in the state provider network implementation and benefits in account abstraction

(AA) models.

Proposal for a Phase 2 Implementation Plan

Summarizing the narrative above, we suggest breaking up the implementation work into two steps, but with the goal of deploying it all at once. We organize these steps into the necessary work required at the core protocol vs. the Execution Environment

(EE) layer.

- Phase 2 base protocol with:
- Netting for cross-shard ETH transfers
- SSA (with the exception of the eth1 shard continuing to support DSA)
- Native support for transaction verification to support Account Abstraction (AA)
- Other host functions deemed necessary along the way
- Possible restricted EE deployment to just the canonical or several predefined EEs
- Netting for cross-shard ETH transfers
- SSA (with the exception of the eth1 shard continuing to support DSA)
- Native support for transaction verification to support Account Abstraction (AA)
- Other host functions deemed necessary along the way
- Possible restricted EE deployment to just the canonical or several predefined EEs
- Canonical / General Purpose EE (aka EF sanctioned EE):
- User level cross-shard communication capabilities (experimenting with different approaches - aka bitfield, nonce/queue, object capabilites
- Account Abstraction
- Passive cross-shard developer tooling
- Extending Solidity to support development of SSA smart contracts, development including access to all relevant eth2 host functions
- User level cross-shard communication capabilities (experimenting with different approaches - aka bitfield, nonce/queue, object capabilites
- Account Abstraction
- Passive cross-shard developer tooling
- Extending Solidity to support development of SSA smart contracts, development including access to all relevant eth2 host functions

With a canonical/general purpose EE, we will assume early updates will require forking the protocol before we remove restrictions on EE deployments.

Goals to reach for

:

- Strong, cross-EE communication standard
- Canonical EE, Optimistic Rollup EE, ZK Rollup EE supported on launch
- Unrestricted EE deployment
- Non-altruistic state network

Quilt has already been building tooling to support this roadmap. Our concrete plan is to have

an open EE testnet in 3-6 months with a testnet around the canonical/general purpose EE in 6 months to a year.

Thanks to my fellow Quilt team for contributing to this writeup:@adietrichs, @SamWilsn, @matt, and @gt

Also thanks to @rjdrost, @cdetrio, and @adlerjohn for review and guidance.

This writeup is the perspective from the Quilt team on the way forward with eth2 phase 2.