Eth 2.0
Testing and Simulation

Hello!

I am Mamy Ratsimbazafy

Part of the Nimbus Eth1 and Eth2 implementer team at Status

“Adolescent Full Mustang”

@mratsim

@m_ratsim

status

Where are we?

https://github.com/ethereum/eth2.0-specs

Phase 0

- The beacon chain
- Status:
  - testnets incoming

Responsibilities

- Coordination layer
- Sharding
- Block processing
  - Proof-of-Stake
  - Managing validators, shards, committees and attestations.
- Additional features
  - Finality
  - RNG
  - Cross-shard communication
  - Eth1 transition

What's next?

https://github.com/ethereum/eth2.0-specs

Phase 1

- The shard data chains
- Status:
  - Spec written, no implementation started.
- Responsibilities
  - Consensus over the data (e.g. account balance)

Phase 2

- The VM / execution layer
- Status:
  - In discussion (eWASM?)
- Responsibilities
  - Executing transactions (from simple transfers to smart-contracts)

What's next?

https://github.com/ethereum/wiki/wiki/Sharding-roadmap

Phase 3

- Light clients

Phase 4

- Cross-shard transactions

What's next?

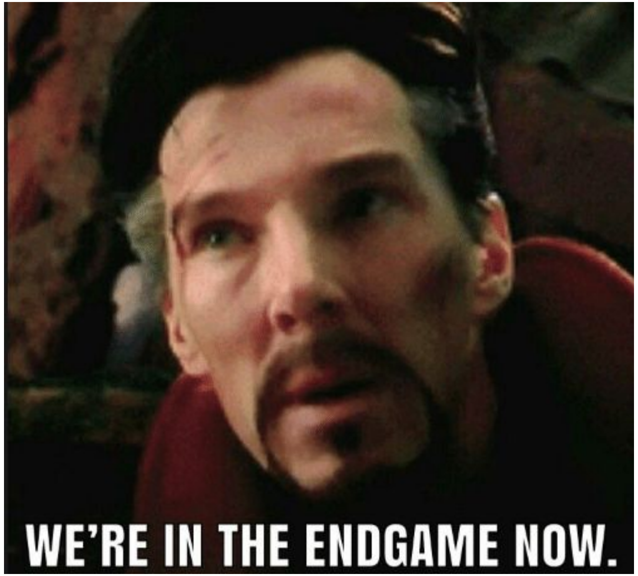
https://github.com/ethereum/wiki/wiki/Sharding-roadmap

Phase 5

- Ethereum 2.0 endgame

⟠ 16:00 - 16:45 : Ethereum 2.0 End game

- Vitalik Buterin
- Posts
  - **Fork-free sharding**: https://ethresear.ch/t/fork-free-sharding/1058/
  - **A model for tightly coupled sharding plus full Casper**: https://ethresear.ch/t/a-model-for-stage-4-tightly-coupled-sharding-plus-full-casper/1065
  - **In favor of forkfulness**: https://ethresear.ch/t/in-favor-of-forkfulness/1225



What's next?

https://github.com/ethereum/wiki/wiki/Sharding-roadmap

Phase 6

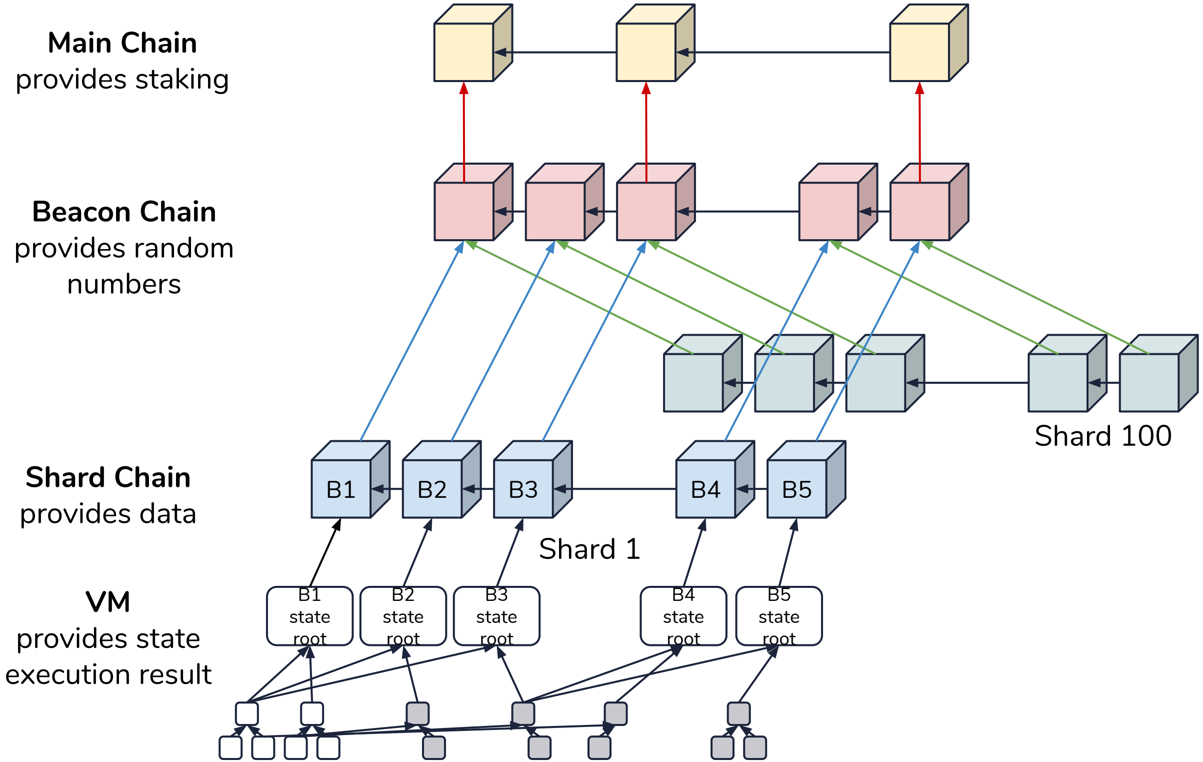
- Super quadratic sharding

Phase 6: Super-quadratic or exponential sharding

- Recursively, shards within shards within shards... Again, this may be difficult with the latest spec as it uses a beacon chain rather than a contract.
- Load balancing: Wikipedia, search results. Related: History, state, and asynchronous accumulators in the stateless model, State minimized implementation on current evm

Diving into the beacon chain

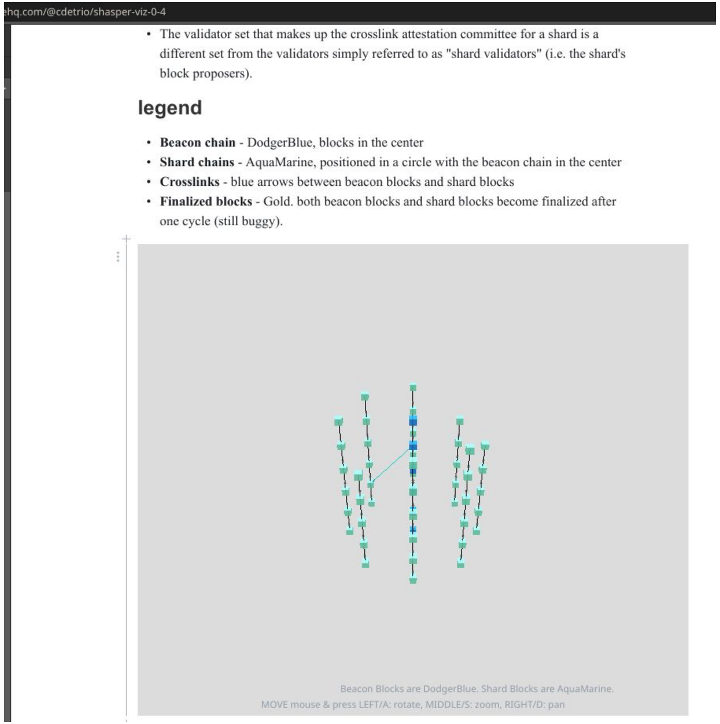
Credits: Hsiao Wei Wang, Ethereum Foundation

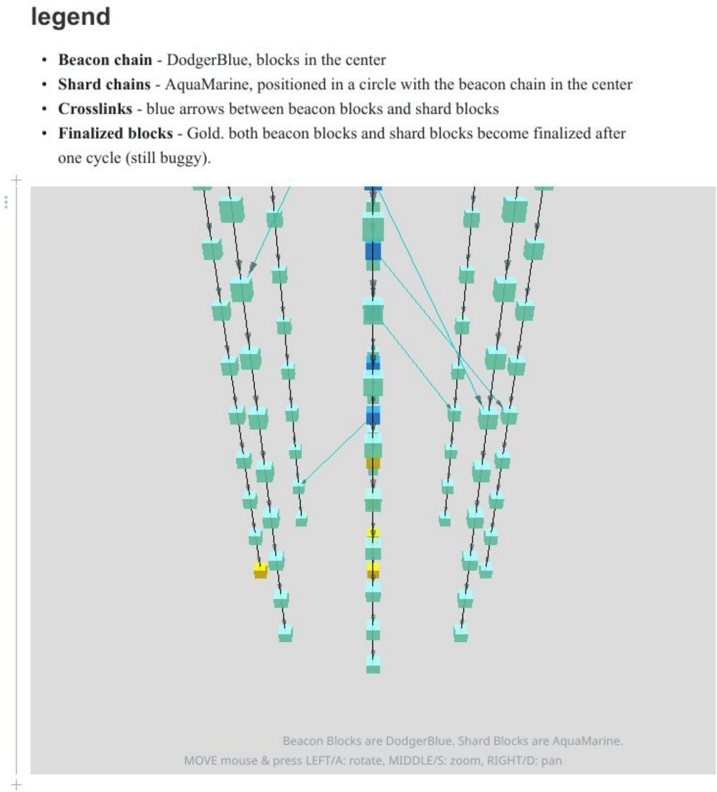


Diving into the beacon chain

https://observablehq.com/@cdetrio/shasper-viz-0-4

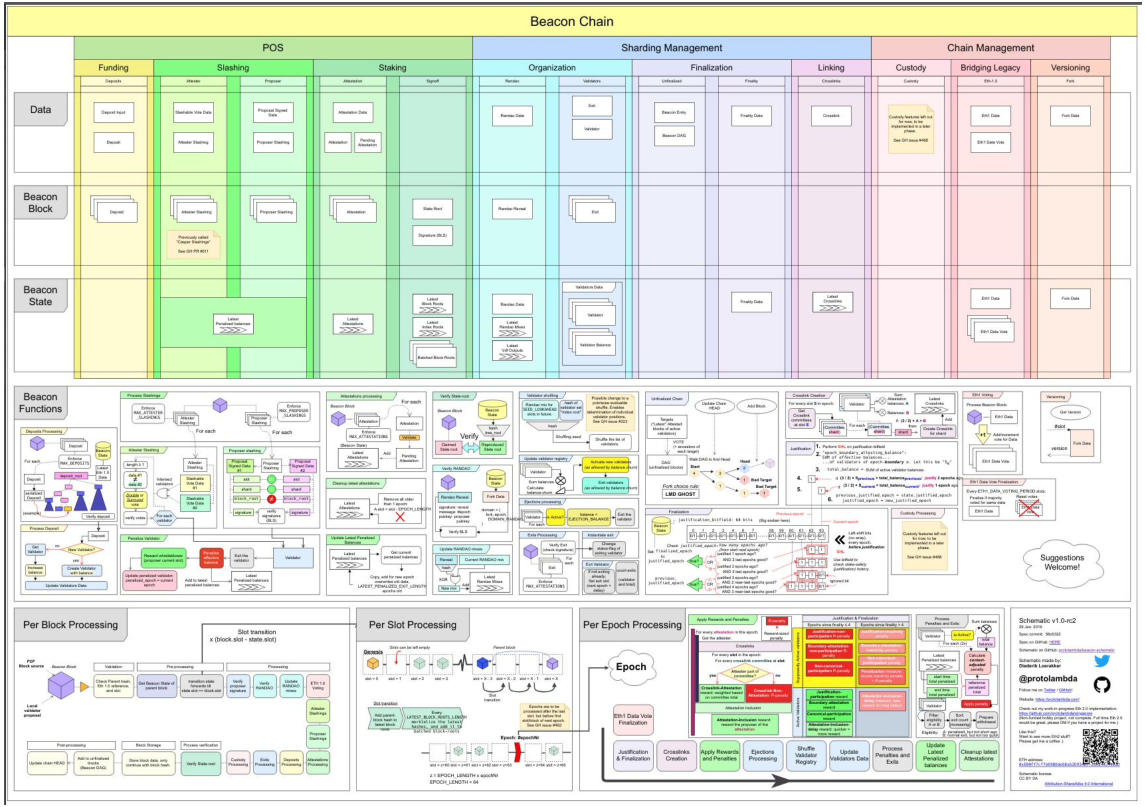
Credits: Casey Detrio, Ethereum Foundation





Diving into the beacon chain

Credits: Diederik Loerakker, @protolamba



Blast from the past

https://web.archive.org/web/20131219030753/http://vitalik.ca/ethereum.html

Ethereum: The Ultimate Smart Contract and Autonomous Corporation Platform on the Blockchain

In the last few months, there has been a great amount of interest into the area of using the Bitcoin blockchain, the mechanism that allows for the entire world to agree on the state of a public ownership database, for more than just money. Perhaps the first, and oldest, such alternative application is colored coins, which is a protocol that allows users to label specific bitcoins and treat them as assets representing some real world value - whether company shares, collectibles or even existing currencies like gold and USD. A more independent alternative, Ripple, also includes the ability to create custom currencies and assets, but adds a decentralized exchange. More recently, Mastercoin has started to go even further, allowing more complex financial contracts such as hedging, trust-free dice rolls, binary options and self-stabilizing currencies - essentially, almost any common financial instrument imaginable. Taken together, all of these projects can be thought of as initial efforts toward a sort of "cryptocurrency 2.0" - they are to Bitcoin what Web 2.0 was to the World Wide Web circa 1995.

At the same time, there has been significant interest in "decentralized autonomous corporations" - autonomous entities that operate on the blockchain in a completely transparent and publicly managed way without any central control whatsoever. Rather than the relationships of the investors, owners and employees of the corporation being mediated by a legal contract or a set of organizational bylaws, the funds and corporate resources are managed directly on the blockchain. However, decentralized autonomous corporations are difficult to implement today, simply because the scripting systems of Bitcoin, and even proto-cryptocurrency 2.0 alternatives like Ripple and Mastercoin, are far too limited to allow the kind of arbitrarily complex computation that DACs require. Although these platforms have begun to offer increasingly complex contracts such as financial derivatives, order matching and trust-free bets, the way that the protocols are set up is inherently limited and closed-ended: each of these use cases is treated as a specific transaction type, not allowing any way for users to build contracts that the developers have not specifically chosen to include.

What this project intends to do is take cryptocurrency 2.0, and generalize it - create a fully-fledged, Turing-complete (but heavily fee-regulated) cryptographic ledger that allows participants to encode arbitrarily complex contracts, autonomous agents and relationships that will be mediated entirely by the blockchain. On-chain currencies, futures contracts, prediction markets, Namecoin-style domain name systems and even provably fair gambling sites will become trivial to implement, existing as simple, hundred-line-of-code contracts on the chain.

Basic Building Blocks

Network: Ethereum will run on its own network with a memory-hard proof of work (not yet released) and a 60-second block time using single-level GHOST (see http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf) to improve security and fairness with fast confirmation times. The restriction to single-level is done for simplicity and because a very fast block time is undesirable for other reasons - namely, blocks will potentially take a very long time to evaluate, so high levels of waste are computationally undesirable, and block validation time will be potentially very high variance.

Currency: The Ethereum network includes its own built-in currency, ether. The main reason for including a currency in the network is to serve as a mechanism for paying transaction fees for anti-spam purposes; of the two main alternatives, proof of work and feeless laissez-faire, the former is economically inefficient and unfairly punitive against weak computers and the latter would lead to the network being almost immediately overwhelmed by an infinitely looping "logic bomb" contract. Ether will have a theoretical hard cap of 2^128 units (compare 2^50.9 in BTC), although not more than 2^105 units will be released in the foreseeable future. For convenience and to avoid future argument, the denominations will be labelled:

1: wei
2^20: shamir
2^40: szabo
2^60: finney
2^80: ether
2^100: koblitz
2^120: turing

Issuance model: the issuance model will be Quark-like, with 64 ether per block released for 32768 blocks (~3 weeks), reducing by a factor of two every 32768 blocks until finally stabilizing at 1 ether per block. This means that the initial burst supply will be 2-4 koblitz, and from then on 1 koblitz will be released per two years (permanent linear inflation, eventually de facto zero inflation due to currency units being lost).

Transactions: transactions in Ethereum will be simple, with one sender, one recipient, a value, a fee and a message consisting of zero or more data items that are integers in `[ 0 ... 2^256 - 1]` (ie. 32 byte values). All transactions are valid; transactions where the recipient has insufficient funds simply do nothing. Transactions sent to the zero address (ie. whose hexadecimal representation is all zeroes) are a special type of transaction, creating a "contract".

In deserialized form, a transaction looks as follows:

```
[
    recipient (20 bytes),
    value (integer),
    fee (integer),
    [
        data item 0 (integer),
        data item 1 (integer),
        ...
    ],
    signature (65 bytes)
]
```

Contracts

Here is where we get to the actually interesting part of the Ethereum protocol. In Ethereum, there are actually two types of entities that can generate and receive transactions: actual people (or bots, as cryptographic protocols cannot distinguish between the two) and contracts. A contract is essentially a piece of code that lives on the Ethereum network, has an Ethereum address and balance, and can send and receive transactions. A contract is "activated" every time someone sends a transaction to it, at which point it runs its code, perhaps modifying its internal state or even sending some transactions, and then shuts down. The "code" for a contract is written in a special-purpose assembly language, executed in a virtual machine consisting of 256 registers, which are not persistent, and 2^256 memory entries, which constitute the contract's permanent state. The design principles behind contracts are as follows:

1. **Simplicity** - the Ethereum protocol should be as simple as possible, even at the cost of some efficiency. Any decent programmer should be able to re-implement it.
2. **Computational universality** - contracts can execute any function that anyone may want a contract to execute, and conditionally send out money to people based on the result of the calculations
3. **Size-universality** - contracts can exist for an arbitrarily long period of time and have arbitrarily many participants
4. **First class citizen property** - contracts can send and receive ether, make transactions (potentially to other contracts), read the state of other contracts and even create other contracts themselves
5. **Pigovian fee regulation** - the only mechanism for fighting spam or bloat is fees. You can run an infinite recursion bomb on top of Ethereum for as long as you are willing to keep feeding the contracts to pay for it
6. **Everything is a contract** - the contract is the basic data type of everything in the Ethereum network, except for ether itself. Want to make your own currency? Set it up as a contract. Want to make an order selling ether in exchange for units of another currency? Set up a contract to do that. Want to make a trust-free bet? Also a contract. Want to set up a full-scale Daemon or Skynet? Well, maybe you might want to have a few thousand interlocking contracts, and be sure to feed them generously, to do that, but nothing is stopping you. Ultimately, you may wish to even outsource some heavy computation to centralized parties by offering a bounty in the contract, using SCIP to verify the validity of the result; the sky(net) is the limit.

Examples of what contracts can do

Here are some examples of how a contract might work, written in high-level pseudocode:

1) **Simulate an entire currency as a single contract**. This is surprisingly easy to implement; the idea is that sending currency units requires sending a transaction to the contract with data item 0 as the recipient and data item 1 as the value. For a transaction to be valid, it must send 200000 finney to the contract in order to "feed" the contract (as each computational step after the first 16 for any contract costs a small fee)

```
if tx.value < 200000 finney: exit
if memory[1000]:
    from = tx.sender
    to = tx.data[0]
    value = tx.data[1]
    if to <= 1000: exit
```

Basic Building Blocks

Network: Ethereum will run on its own network with a memory-hard proof of work (not yet released) and a 60-second block time using single-level GHOST (see http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf) to improve security and fairness with fast confirmation times. The restriction to single-level is done for simplicity and because a very fast block time is undesirable for other reasons - namely, blocks will potentially take a very long time to evaluate, so high levels of waste are computationally undesirable, and block validation time will be potentially very high variance.
Currency: [...]
Issuance model: [...]
Transactions: transactions in Ethereum will be simple, with one sender, one recipient, a value, a fee and a message consisting of zero or more data items that are integers in [0 ... 2^256 - 1] (ie. 32 byte values). All transactions are valid; transactions where the recipient has insufficient funds simply do nothing. Transactions sent to the zero address (ie. whose hexadecimal representation is all zeroes) are a special type of transaction, creating a "contract".

Beacon chain spec - initial commit

https://ethresear.ch/t/convenience-link-to-casper-sharding-chain-v2-1-spec/2332

Full Casper chain v2

(Note: this is ~70% complete)

Main chain changes

- On the main chain there exists a contract; this contract allows you to deposit 32 ETH; the `deposit` function also takes as arguments (i) `validation_code` (bytes), (ii) `return_shard_id` (int), (iii) `return_addr` (address) and (iv) `randao_commitment` (bytes32)
- Main chain clients will implement a function, `prioritize(block_hash, value)`. If the block is available and has been verified, sets its score to the given value, and recursively adjusts the scores of all descendants.

Beacon chain

There exists a beacon chain, where each block header contains the following fields:

- `parent_hash` (bytes32): self-explanatory
- `skip_count` (int): is this block a zero-skip block, one-skip block, etc etc
- `randao_reveal` (bytes32): the RANDAO reveal value (see below)
- `attestation_bitmask` (bytes): a bitmask specifying which of the validators in the committee have made signatures
- `attestation_aggregate_sig` (bytes): the aggregate signature corresponding to the attestation
- `ffg_signer_list` (bytes): a list of indices specifying which validators are making FFG votes
- `ffg_aggregate_sig` (bytes): the aggregate signature for FFG votes
- `main_chain_ref` (bytes32): a reference to a main chain block; must have height 0 mod 100
- `state_root` (bytes32): the beacon chain's state root
- `height` (int): block height
- `sig` (bytes): self-explanatory

The beacon chain has the following state variables:

- `validators: {pubkey: bytes, return_addr: address, return_shard: int, randao_commitment: bytes32, end_dynasty: int}[int]`
- `pending_validators: {pubkey: bytes, return_addr: address, return_shard: int, randao_commitment: bytes32, start_dynasty: int}[int]`
- `current_dynasty: int`
- `global_randao: bytes32`
- `last_justified_epoch: int`
- `current_checkpoint: bytes32`
- `validator_balances: bytes4[]`
- `validator_voted: bool[]`
- `validator_total_vote: int`
- `total_skip_count` (int): total number of skips
- `total_deposits` (int): total number of skips

We define the algorithm QUICK_SAMPLE as follows. The inputs are a 32 byte hash `seed`, a data length `n` and a count `c`; the goal is to output a list of `c` integers in `0...n-1` which represent randomly sampled elements of `n`.

- Let `k` be the smallest value such that `256**k >= n`.
- Initialize `o = []`, `source = seed`, `pos = 0`.
- While `len(o) < c`, repeat the following three steps:
  - If `pos + k > 32`, set `source = blake2s(source)` and `pos = 0`.
  - Treat `source[pos: pos+k]` as a big-endian integer and call it `m`. If `n * (floor(m / n) + 1) > 256**k`, continue without doing anything. Otherwise, append `m % n` to `o`.
  - Set `pos += k`
- Return `o`.

The following algorithm is used to verify block headers.

First basic checks:

- Check that the parent has already been verified. If not, put it in a queue and wait for the parent to be verified.
- Let `expected_time = GENESIS_TIME height * 2 + total_skip_count * 8`. Check that the local time exceeds `expected_time`; if not, put it in a queue and wait until local time reaches that value.
- Verify that `main_chain_ref` is either (i) the same as the parent, or (ii) a descendant of the `main_chain_ref` of the parent which has height mod 100, and which has already been processed
- Verify that `height` equals the parent's height plus 1

Now signature and committee checks:

- Let `committee_size = floor(len(validators) / 100)`. Let `indices = QUICK_SAMPLE(global_randao, n, committee_size + skip_count + 1)`, and let `expected_signer_index = indices[committee_size + skip_count]`.
- Check that `sha3(randao_reveal) == validators[expected_signer_index].randao_commitment`

- Verify `sig` against `validators[index].pubkey` and the hash of the header without the sig
- Let the *attestation committee* be `indices[:committee_size]`.
- Let `ones_count` equal the number of 1 bits in the `attestation_aggregate_sig`. Check that `ones_count * (1.5 + skip_count) >= committee_size`.
- Verify that `len(attestation_bitmask)` (in bytes) equals `ceil(committee_size / 8)`. Treat the `attestation_bitmask` as a bitfield, representing the subset of the committee that is included in the `attestation_aggregate_sig`. Check that bits with indices outside the committee (there are min 0, max 7 such bits at the end) are all set to 0. Verify that the `attestation_aggregate_sig` is a valid aggregate signature of the parent block hash for those validators.

Now Casper FFG cycle related operations, only if `height % 100 == 1`:

- Say that the *previous epoch is justified* if `validator_total_vote >= total_deposits * 2/3`.
- Say the `current_epoch` equals `floor(height / 100) + 1`
- Use REWARD_PENALTY_ALGO (not yet specified) to determine the `voter_reward` and `nonvoter_penalty` values for the last epoch, using `total_deposits` as input.
- Go through all active validators; if they voted in the last epoch, increase their balance by `voter_reward`, otherwise decrease it by `nonvoter_penalty`. If any validator's deposit size falls below 20 ETH, set their `end_dynasty` to equal `current_dynasty + 1`
- If the `last_justified_epoch` equals the `current_epoch` minus 2, and the previous epoch is justified, then set `dynasty += 1`, and go through all validators in the pending queue; if any of them specify start dynasty equal to or less than the `current_dynasty`, then move a maximum of `len(validators) / 50` to the active validator set, and add 32 ETH * the number added to `total_deposits`
- If the previous epoch is justified, set `last_justified_epoch = current_epoch - 1`
- Set the `validator_voted` array to all zeroes, with length `ceil(len(validators) / 8)`. Set `validator_total_vote` to 0.

Now the Casper FFG cycle related operations for every block:

- For all validators that voted,

Now some state transitions:

- Set `total_skip_count += skip_count`
- Set `global_randao = xor(global_randao, randao_reveal)`
- If the `main_chain_ref` is a new value, for every deposit made in the main chain segment between the new `main_chain_ref` and the old one, add a validator to the `pending_validators` set, using `current_epoch + 2` as the start epoch.

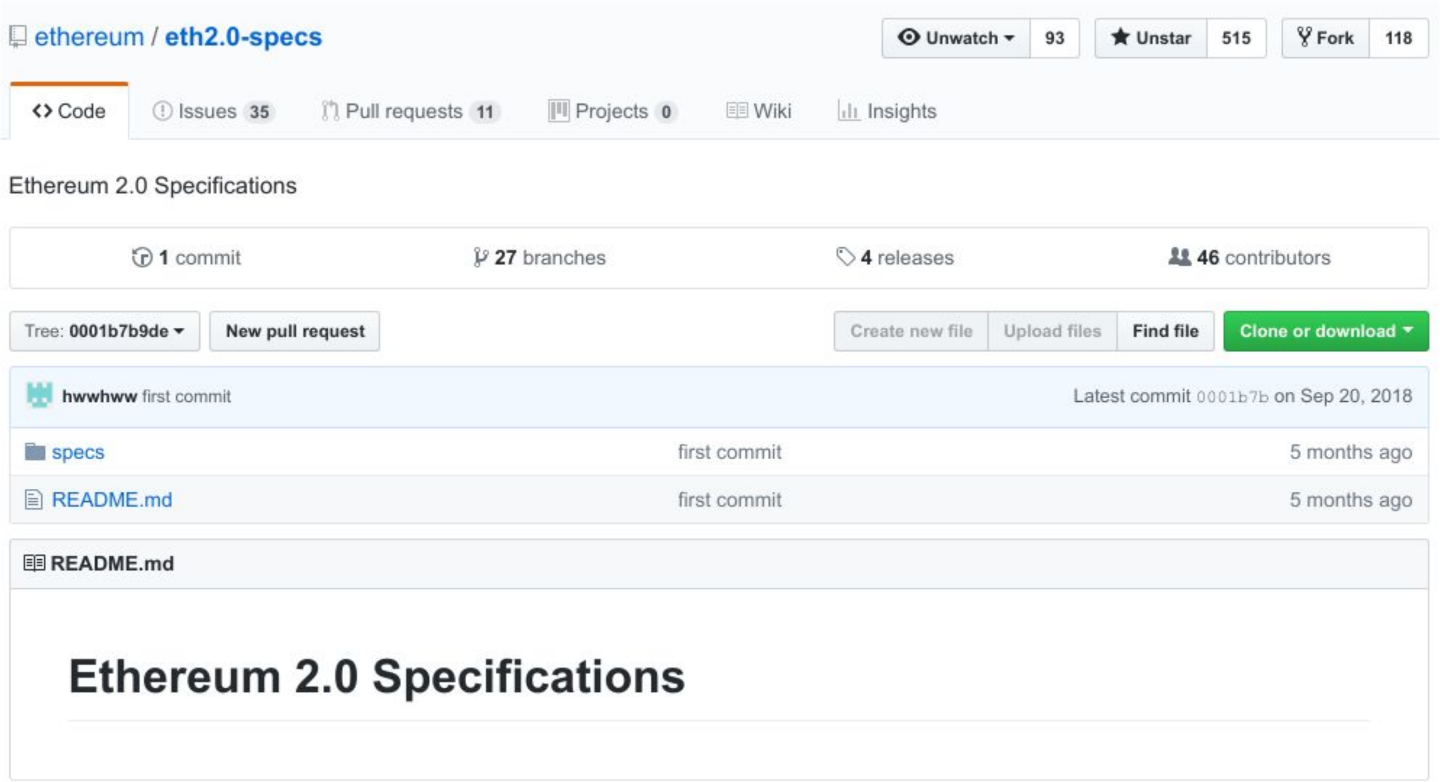
Verify that the post-state-root matches.

Life of a beacon chain implementer

June-Sept 2018 - The HackMD + ethresear.ch period

Life of a beacon chain implementer

Sept 2018-Feb 2019 - Living commit by commit



Life of a beacon chain implementer

Feb 2019–Present – Spec releases!

Pre-release

v0.1

4cdb667

Verified

The January pre-Release™

djrtwo released this 29 days ago · 234 commits to master since this release

The marks the first release in a series of weekly releases through February 2019. Phase 0 in v0.1 is relatively feature complete and approaching stable. Subsequent changes will occur on `dev` branch and only merged into `master` during a release accompanied by a changelog.

Know phase 0 semi-major items to add/change:

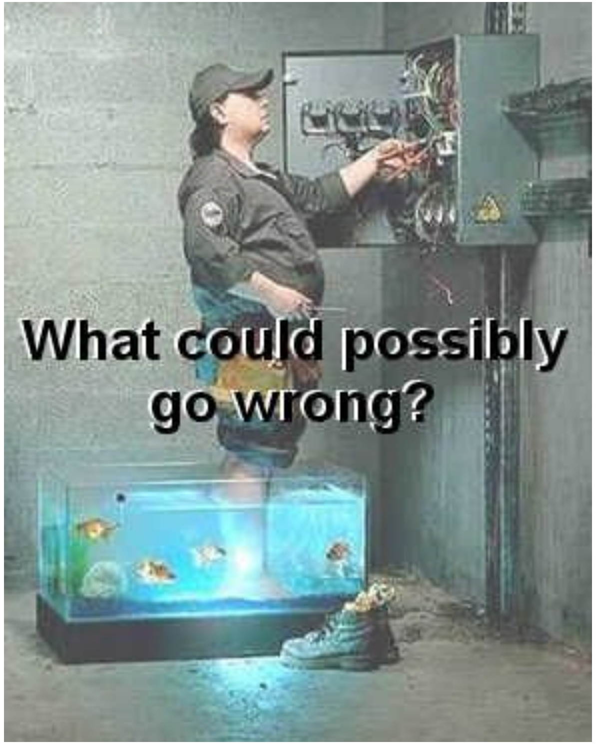
- Pointwise shuffling evaluation
- Enforced ordering of eth1 deposits
- Minor adjustments to BLS (e.g. hash functions, generators, serialization) to conform to standards.
- Adjustment of constants pending testnets and community input
- Bugs.

Assets 2

Source code (zip)

Source code (tar.gz)

What could go wrong?



What could go wrong?

Specs

- “Trivial” bugs (off-by-one)
- Edge cases (genesis, forced exits)
- Vulnerabilities (DOS attacks)
- Underspecification & interoperability (crypto & serialization)
- Slowness (shuffling)
- Incentives / Game Theory

Implementation

- Overflow/underflow
- Slowness (using naive spec algorithm)
- Spec miscomprehension
- Vulnerabilities
- Implementation “details”

What could go wrong?

Focus on implementation “details”

Cryptography

P2P Networking

Signature aggregation

Serialization

Consensus

Sync

State transition

What could go wrong?

Practical case - shuffling

Code 4
Commits 35
Issues 73
Wikis 0

States
Closed 65
Open 8

Advanced search Cheat sheet

73 issues in ethereum/eth2.0-specs

Use custom types in data structures
Opened by Nashatyrev 9 days ago • 4 comments

fix committee assignment bugs
2 bugs: - We were modifying current_ **shuffling** _epoch and then calling get_current_epoch_committee_count expecting that it was still getting the committe value of ...
Opened by djrtwo 2 days ago • 5 comments

[WIP] Break up crosslink_committees_at_slot
should we add an assert to check **shuffling** _epoch out of bound condition? assert get_previous_epoch(state) <= **shuffling** _epoch <= get_current_epoch(state)+1
Opened by djrtwo 7 hours ago • 8 comments

committee shufflings take at least 2 epochs to change
... validating and not all shards are represented in each **shuffling**). - This slows dc activations and exits by a factor of 2.
Opened by djrtwo on Jan 14 • 3 comments

Light client proposal
... , maintained similarly to the randao roots - To compute the seed used in get_ **shu** sha3(get_randao_mix(slot), get_active_index_root(slot)) where get_active_index_r
Opened by vbuterin on Jan 17 • 3 comments

get_crosslink_committees reads previous_seed during genesis_epoch
Problem Summary During GENESIS_EPOCH, get_crosslink_committees_at_slot(state.previous_ **shuffling** _epoch when epoch == current_epoch. Detail get_crosslink_committees_at_slot(...) first ...
Opened by paulhauner 13 days ago

Fix out-of-bounds in `get_ **shuffling** `
What Change get_ **shuffling** (...) so it gives shuffles using an index of active_valida value of active_validator_indices. Why The following fails with a IndexError: list inde

Remove Record suffix #434
no, get_ **shuffling** just returns the **shuffling** split across slots. get_shard_committees_at_slot returns an array of arrays of (validators, shard) tuples. Arguably, the naming and return value of these ...
Opened by sifnoc on Jan 13 • 24 comments

Keep latest 2^n RANDAO mixes in the state #295
Also ensures that we have the seed of the current **shuffling** in state. Right now, we discard the current **shuffling** seed immediately after performing the **shuffling** . Not very friendly for reconstructing the **shuffling** outside the specific state transition.
Opened by JustinDrake on Dec 12, 2018 • 4 comments

"proposer_slots" -> "proposer_nonce" #493
Opened by paulhauner on Jan 24 • 1 comment

RFC: drop shard_and_committee_for_slots from state #191
Opened by arnetheduck on Nov 29, 2018 • 4 comments

Understanding the `get_new_ **shuffling** ()` function #127
For the past several weeks, as part of my hackternship, I have been trying to understand the get_new_ **shuffling** () function. In its current iteration, the get_new_ **shuffling** () function is in charge of ...
Opened by Mikerah on Nov 14, 2018 • 3 comments

initial assignment of `state.persistent_committees` #239
Issue We are currently assigning state.persistent_committees as a **shuffling** of ValidatorRecords rather than just validator_indices persistent_committees=split(shuffle(initial_validator_registry ...
Opened by djrtwo on Dec 5, 2018 • 5 comments

Remove MIN_VALIDATOR_REGISTRY_CHANGE_INTERVAL #340
... in times of non-finality (attacks, short range forks, etc), I worry we end up making the **shuffling** extremely subjective and ultimately an attack vector. (for example: easier to construct reasonable looking blocks when the proposer for that slot is not entirely certain)
Opened by vbuterin on Dec 19, 2018 • 5 comments

assertion in `get_active_index_root` too strong #515
Issue In validator registry and **shuffling** seed data we set state.current_calculation_epoch = next_epoch and then do state.current_epoch_seed = generate_seed(state, state.current_calculation_epoch ...
Opened by djrtwo on Jan 29

Delay exits with penalty #350
Delaying exits with penalty by 1+epsilon epochs ensures that self-slashing single validators does not change the **shuffling** for the next epoch and so cannot (normally) be used as a way of manipulating the **shuffling** .
Opened by vbuterin on Dec 21, 2018 • 8 comments

Introduce swap-or-not shuffle #576
See #563 for discussion. Here is a more efficient implementation for **shuffling** an entire set; it can live here until we come up with an explicit "efficient implementation" doc: def shuffle ...
Opened by vbuterin 23 days ago • 20 comments

Mitigating attacks on light clients #403
... **shuffling** . Note that alternative **shuffling** algos do not fix this problem, because the step of filtering out inactive validators still requires a pass through the entire validator set. Second, it is ...
Opened by vbuterin on Jan 7 • 8 comments

helpers and notes for **shuffling** lookahead #520
beacon chain spec changes: - update get_crosslink_committees_at_slot to be able to get potential committees for slots from the next epoch. add registry_change param to get next epoch committees ...
Opened by djrtwo on Jan 30 • 7 comments

Possible alternative numer-theoretic **shuffling** algorithm #323
Motivation Construct a **shuffling** algorithm where you can compute the value in the shuffled list at any specific position relatively cheaply without computing all of the other values at the same time ...
Opened by vbuterin on Dec 14, 2018 • 13 comments

non-determinism in **shuffling** from `SEED_LOOKAHEAD` #405
Issue shufflings are calculated using a seed from SEED_LOOKAHEAD slots ago get_ **shuffling** (state.latest_randao_mixes[(state.slot - SEED_LOOKAHEAD) % LATEST_RANDAO_MIXES_LENGTH ...
Opened by djrtwo on Jan 7 • 3 comments

What could go wrong?

Practical case - shuffling

- Understanding the shuffling algorithms
- Determinism issues
- Attack vector concerns
- Performance concerns (light clients)
- Out-of-bounds bugs (in the specs not even in implementations)

And we have 9 teams, each implementing their own clients.

Why so many clients at launch?

Unlike Eth 1.0

Client implementations

Artemis (ConsenSys, Java)
- https://github.com/PegaSysEng/artemis

Harmony (Harmony, Java)
- https://github.com/harmony-dev/beacon-chain-java

Lodestar (ChainSafe System, Typescript / Javascript)
- https://github.com/ethereum/trinity

Lighthouse (Sigma Prime, Rust)
- https://github.com/sigp/lighthouse

Nimbus (Status, Nim)
- https://github.com/status-im/nim-beacon-chain

Prysm (Prysmatic Labs, Go)
- https://github.com/prysmaticlabs/prysm

Shasper (Parity Technologies, Rust)
- https://github.com/paritytech/shasper

Trinity (Ethereum Foundation, Python)
- https://github.com/ethereum/trinity

Yeeth (ZK Labs, Swift)
- https://github.com/yeeth/BeaconChain.swift

What do we do?

Common testing repositories

- https://github.com/ethereum/eth2.0-tests/
    - Handcrafted and generated test vectors
- https://github.com/ethereum/eth2.0-test-generators
    - Generate test vectors using Trinity reference implementation

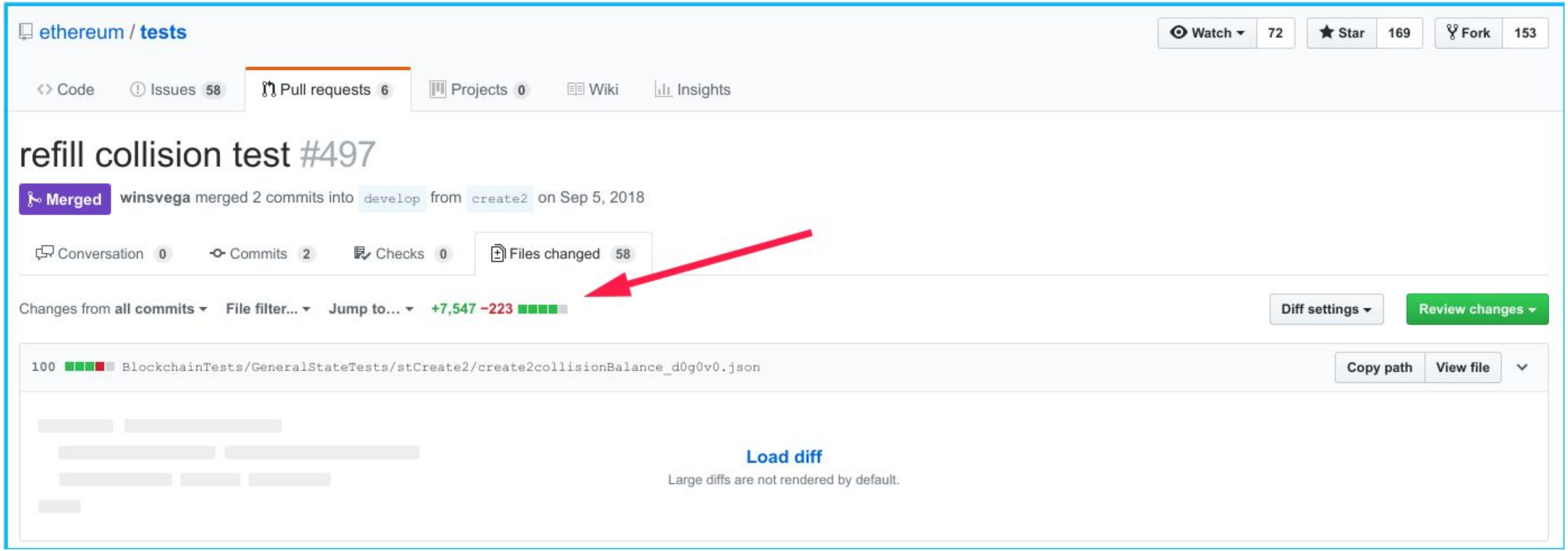
Status: test vectors for “relatively” stable and self-contained part of the spec

- Crypto: BLS signatures
- Shuffling
- Serialization

What did we learn from Eth 1.0?

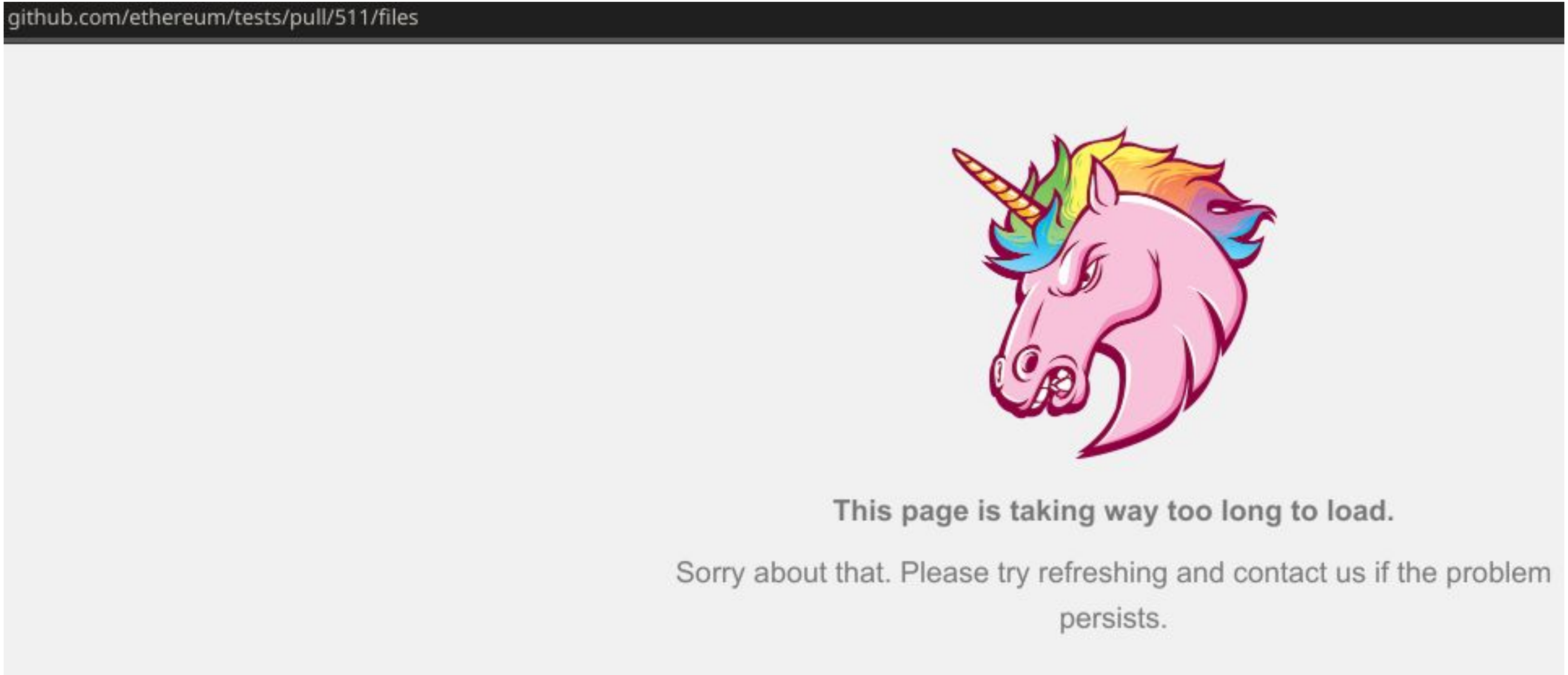
- A test repo that can be submoduled is good
- Having no comment (json) in test files is bad
- Having an insane amount of lines of code to review is worse

What did we learn from Eth 1.0?



What did we learn from Eth 1.0?

What did we learn from Eth 1.0?



Coming soon

More unit tests

- Merkle tree hashing
- Fork choice (proof-of-stake)
- Beacon state “god object”

Client-specific testnets

TODO - cross-client testnet

Pending - wire protocol:
https://github.com/ethereum/eth2.0-specs/issues/593

Libp2p interop framework:
https://github.com/libp2p/interop#interoperability-tests-for-libp2p

Sharding P2P POC:
https://github.com/ethresearch/sharding-p2p-poc/tree/master/docs

Ethereum sharding P2P requirement

What does a node in the sharding p2p network need?

- A node should be able to subscribe to multiple shards simultaneously
- A node should be able to jump(i.e., unsubscribe A and then subscribe B) between shards with low latency

Design

In the current stage, we are building a gossip layer on top of a PubSub system. The basic concept is that every shard is one-to-one mapped to a topic in the PubSub and every node will subscribe to the topics they are interested in. **NOTE**: To avoid adding too many details in this documentation, please refer to PubSub documents for basic understanding about what a topic is and how publishing/subscribing works.

- If a node wants to publish shard-specific messages, it **publishes** them to the topic corresponding to that shard.
  - E.g. We can agree on using the topic "Shard_9_collation" as the topic for the collation messages in shard 9. In this manner, collations in shard 9 are published to that topic, and nodes subscribing that topic will get the published collations.
- A node interested in a shard **subscribes** to the topic corresponding to that shard, in order to receive messages regarding the shard.

Simulations

Kinds of simulations for Eth 2.0

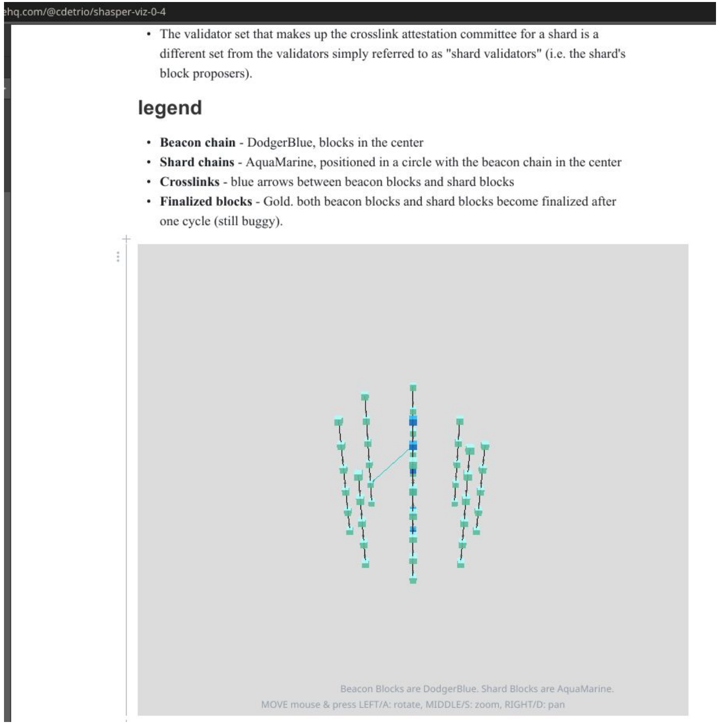
- High-level overview
- Full simulation with a real client (coming soon™)
- Sharding simulations
- Consensus simulations

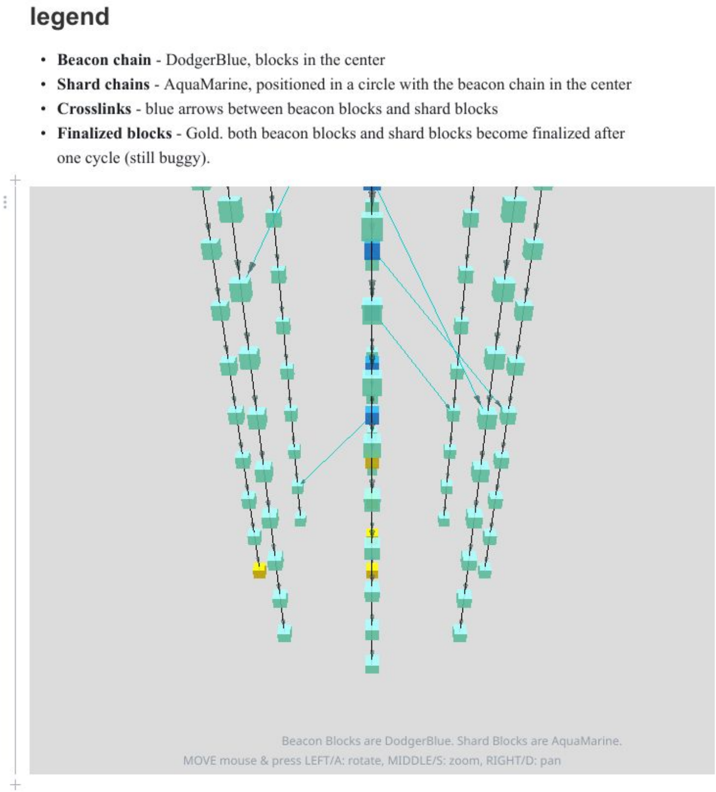
Some are in color ;)

Simulation - High-Level Overview

https://observablehq.com/@cdetrio/shasper-viz-0-4

Credits: Casey Detrio, Ethereum Foundation

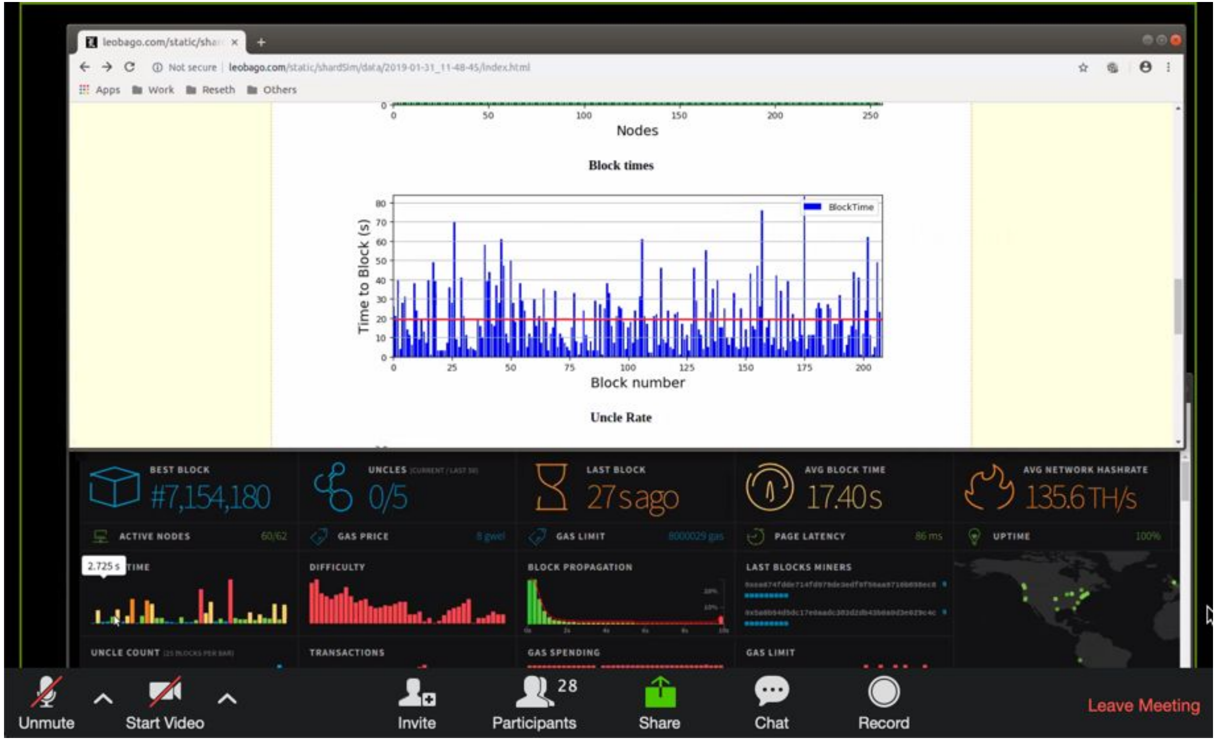




Simulation - Consensus

https://github.com/leobago/shardSim

Credits: Leo Bautista Gomez - Barcelona Supercomputing Center



Available research simulators

- Jannik Luhn - https://github.com/jannikluhn/sharding-netsim (sharding)
- Leo Bautista Gomez - https://github.com/leobago/shardSim (consensus)
- EF + libp2p + Whiteblock - https://github.com/ethresearch/sharding-p2p-poc (sharding)
- Consensys - https://github.com/ConsenSys/wittgenstein (consensus)
- Protolambda - https://github.com/protolambda/lmd-ghost (consensus)
- Vitalik
  - https://github.com/ethereum/research/tree/master/clock_disparity
  - https://github.com/ethereum/research/tree/master/ghost
- Whiteblock - https://github.com/zscole/nonce (whole blockchain, needs client)

And all client teams coming soon™

Looking for a place to start?

Justin Ɖrake
@drakefjustin
Follow

The phase 0 spec (even not fully polished) is slick!

10 ETH bounty to the first person to write in Go (MIT license) the full state transition function (BeaconState, BeaconBlock) -> (BeaconState, Error) in 1,024 lines or less.

9:27 AM - 27 Feb 2019

10 Retweets **48** Likes

10 10 48

Preston Van Loon @preston_vanloon · Feb 27
Replying to @drakefjustin
The line constraint is a bad idea!

Ways to reduce line count:
- write really long lines
- no tests
- no godoc comments
- no comments at all
- ignore errors

1 13

Justin Ɖrake @drakefjustin · Feb 27
Right, the code obviously needs to pass go fmt! Any standard Go library can be used, plus a library for BLS12-381 BLS signatures.

2 1

Justin Ɖrake @drakefjustin · Feb 27
Comments (including godoc comments) don't count towards line count.

1 1

Justin Ɖrake @drakefjustin · Feb 27
Tests also obviously don't count towards line count.

1 1

Thank you!