2018/08/20 updated: 2018/08/19 00:12 AM (GMT) version

https://notes.ethereum.org/SCIg8AH5SA-O4C1G1LYZHQ

Repost of the content (warning: may accidentally become outdated):

Casper+Sharding chain v2.1

WORK IN PROGRESS!!!

This is the work-in-progress document describing the specification for the Casper+Sharding (shasper) chain, version 2.1.

In this protocol, there is a central PoS chain which stores and manages the current set of active PoS validators. The only mechanism available to become a validator initially is to send a transaction on the existing PoW main chain containing 32 ETH. When you do so, as soon as the PoS chain processes that block, you will be queued, and eventually inducted as an active validator until you either voluntarily deregister or you are forcibly deregistered as a penalty for misbehavior.

The primary source of load on the PoS chain is attestations

. An attestation has a double role:

1. It attests to some parent block in the beacon chain
2. It attests to a block hash in a shard (a sufficient number of such attestations create a “crosslink”, confirming that shard block into the main chain).

Every shard (e.g. there might be 1024 shards in total) is itself a PoS chain, and the shard chains are where the transactions and accounts will be stored. The crosslinks serve to “confirm” segments of the shard chains into the main chain, and are also the primary way through which the different shards will be able to talk to each other.

Note that one can also consider a simpler “minimal sharding algorithm” where crosslinks are simply hashes of proposed blocks of data that are not themselves chained to each other in any way.

Note: the python code at https://github.com/ethereum/beacon_chain and an ethresear.ch post do not reflect all of the latest changes. If there is a discrepancy, this document is likely to reflect the more recent changes.

Terminology

- Validator
- a participant in the Casper/sharding consensus system. You can become one by depositing 32 ETH into the Casper mechanism.
- Active validator set
- those validators who are currently participating, and which the Casper mechanism looks to produce and attest to blocks, crosslinks and other consensus objects.
- Committee
- a (pseudo-) randomly sampled subset of the active validator set. When a committee is referred to collectively, as in “this committee attests to X”, this is assumed to mean “some subset of that committee that contains enough validators that the protocol recognizes it as representing the committee”.
- Proposer
- the validator that creates a block
- Attester
- a validator that is part of a committee that needs to sign off on a block.
- Beacon chain
- the central PoS chain that is the base of the sharding system.
- Shard chain
- one of the chains on which transactions take place and account data is stored.

- Crosslink
- a set of signatures from a committee attesting to a block in a shard chain, which can be included into the beacon chain. Crosslinks are the main means by which the beacon chain “learns about” the updated state of shard chains.
- Slot
- a period of 8 seconds, during which one proposer has the ability to create a block and some attesters have the ability to make attestations
- Dynasty transition
- a change of the validator set
- Dynasty
- the number of dynasty transitions that have happened in a given chain since genesis
- Cycle
- a period during which all validators get exactly one chance to vote (unless a dynasty transition happens inside of one)
- Finalized

, justified

- see Casper FFG finalization here: https://arxiv.org/abs/1710.09437

Constants

- SHARD_COUNT
- a constant referring to the number of shards. Currently set to 1024.
- DEPOSIT_SIZE
- 32 ETH
- MAX_VALIDATOR_COUNT
- 222

= 4194304 # Note: this means that up to ~134 million ETH can stake at the same time

- SLOT_DURATION
- 8 seconds
- CYCLE_LENGTH
- 64 slots
- MIN_COMMITTEE_SIZE
- 128 (rationale: see recommended minimum 111 here https://vitalik.ca/files/Ithaca201807_Sharding.pdf)

PoW main chain changes

This PoS/sharding proposal can be implemented separately from the existing PoW main chain. Only two changes to the PoW main chain are required (and the second one is technically not strictly necessary).

- On the PoW main chain a contract is added; this contract allows you to deposit DEPOSIT_SIZE

ETH; the deposit

function also takes as arguments (i) pubkey

(bytes), (ii) withdrawal_shard_id

(int), (iii) withdrawal_addr

(address), (iv) randao_commitment

(bytes32), (v) bls_proof_of_possession

- PoW Main chain clients will implement a method, prioritize(block_hash, value)

. If the block is available and has been verified, this method sets its score to the given value, and recursively adjusts the scores of all descendants. This allows the PoS beacon chain's finality gadget to also implicitly finalize main chain blocks. Note that implementing this into the PoW client is

a change to the PoW fork choice rule so is a sort of fork.

Beacon chain

The beacon chain is the "main chain" of the PoS system. The beacon chain's main responsibilities are:

- Store and maintain the set of active, queued and exited validators
- Process crosslinks (see above)
- Process its own block-by-block consensus, as well as the finality gadget

Here are the fields that go into every beacon chain block:

fields = { # Hash of the parent block 'parent_hash': 'hash32', # Slot number (for the PoS mechanism) 'slot_number': 'int64', # Randao commitment reveal 'randao_reveal': 'hash32', # Attestations 'attestations': [AttestationRecord], # Reference to PoW chain block 'pow_chain_ref': 'hash32', # Hash of the active state 'active_state_root': 'hash32', # Hash of the crystallized state 'crystallized_state_root': 'hash32', }

The beacon chain state is split into two parts, active state

and crystallized state

.

Here's the ActiveState

:

fields = { # Attestations that have not yet been processed 'pending_attestations': [AttestationRecord], # Most recent 2 * CYCLE_LENGTH block hashes, older to newer 'recent_block_hashes': ['hash32'] }

Here's the CrystallizedState

:

fields = { # List of validators 'validators': [ValidatorRecord], # Last CrystallizedState recalculation 'last_state_recalc': 'int64', # What active validators are part of the attester set # at what height, and in what shard. Starts at slot # last_state_recalc - CYCLE_LENGTH 'indices_for_slots': [[ShardAndCommittee]], # The last justified slot 'last_justified_slot': 'int64', # Number of consecutive justified slots ending at this one 'justified_streak': 'int64', # The last finalized slot 'last_finalized_slot': 'int64', # The current dynasty 'current_dynasty': 'int64', # The next shard that crosslinking assignment will start from 'crosslinking_start_shard': 'int16', # Records about the most recent crosslink `for each shard 'crosslink_records': [CrosslinkRecord], # Total balance of deposits 'total_deposits': 'int256', # Used to select the committees for each shard 'dynasty_seed': 'hash32', # Last epoch the crosslink seed was reset 'dynasty_seed_last_reset': 'int64' }

A ShardAndCommittee

object is of the form:

fields = { # The shard ID 'shard_id': 'int16', # Validator indices 'committee': ['int24'] }

Each ValidatorRecord is an object containing information about a validator:

fields = { # The validator's public key 'pubkey': 'int256', # What shard the validator's balance will be sent to # after withdrawal 'withdrawal_shard': 'int16', # And what address 'withdrawal_address': 'address', # The validator's current RANDAO beacon commitment 'randao_commitment': 'hash32', # Current balance 'balance': 'int64', # Dynasty where the validator is inducted 'start_dynasty': 'int64', # Dynasty where the validator leaves 'end_dynasty': 'int64' }

And a CrosslinkRecord contains information about the last fully formed crosslink to be submitted into the chain:

fields = { # What dynasty the crosslink was submitted in 'dynasty': 'int64', # The block hash 'hash': 'hash32' }

Beacon chain processing

Processing the beacon chain is fundamentally similar to processing a PoW chain in many respects. Clients download and process blocks, and maintain a view of what is the current “canonical chain”, terminating at the current “head”. However, because of the beacon chain’s relationship with the existing PoW chain, and because it is a PoS chain, there are differences.

For a block on the beacon chain to be processed by a node, three conditions have to be met:

- The parent pointed to by the parent_hash

has already been processed and accepted

- The PoW chain block pointed to by the pow_chain_ref

has already been processed and accepted

- The node’s local clock time is greater than or equal to the minimum timestamp as computed by GENESIS_TIME + slot_number * SLOT_DURATION

If these three conditions are not met, the client should delay processing the block until the three conditions are all satisfied.

Block production is significantly different because of the proof of stake mechanism. A client simply checks what it thinks is the canonical chain when it should create a block, and looks up what its slot number is; when the slot arrives, it either proposes or attests to a block as required.

Beacon chain fork choice rule

The beacon chain uses the Casper FFG fork choice rule of “favor the chain containing the highest-epoch justified checkpoint”. To choose between chains that are all descended from the same justified checkpoint, the chain uses “recursive proximity to justification” (RPJ) to choose a checkpoint, then uses GHOST within an epoch.

For a description see: Beacon chain Casper mini-spec

For an implementation with a network simulator see:
https://github.com/ethereum/research/blob/master/clock_disparity/ghost_node.py

Here’s an example of its working (green is finalized blocks, yellow is justified, grey is attestations):

Beacon chain state transition function

We now define the state transition function. At the high level, the state transition is made up of two parts:

1. The crystallized state realculation, which happens only if block.slot_number >= last_state_recalc + CYCLE_LENGTH

, and affects the CrystallizedState

and ActiveState

1. The per-block processing, which happens every block (if during an epoch transition block, it happens after the epoch transition), and affects the ActiveState

only

The epoch transition generally focuses on changes to the validator set, including adjusting balances and adding and removing validators, as well as processing crosslinks and setting up FFG checkpoints, and the per-block processing generally focuses on verifying aggregate signatures and saving temporary records relating to the in-block activity in the ActiveState

.

Helper functions

We start off by defining some helper algorithms. First, the function that selects the active validators:

def get_active_validator_indices(validators, dynasty): o = [] for i in range(len(validators)): if validators[i].start_dynasty <= dynasty < \ validators[i].end_dynasty: o.append(i) return o

Now, a function that shuffles this list:

def shuffle(lst, seed): assert len(lst) <= 16777216 o = [x for x in lst] source = seed i = 0 while i < len(lst): source = blake(source) for pos in range(0, 30, 3): m = int.from_bytes(source[pos:pos+3], 'big') remaining = len(lst) - i if remaining == 0: break rand_max = 16777216 - 16777216 % remaining if m < rand_max: replacement_pos = (m % remaining) + i o[i],

o[replacement_pos] = o[replacement_pos], o[i] i += 1 return o

Here's a function that splits a list into N pieces:

def split(lst, N): return [lst[len(lst)*i//N: len(lst)*(i+1)//N] for i in range(N)]

Now, our combined helper method:

def get_new_shuffling(seed, validators, dynasty, crosslinking_start_shard): avs = get_active_validator_indices(validators, dynasty) if len(avs) >= CYCLE_LENGTH * MIN_COMMITTEE_SIZE: committees_per_slot = int(len(avs) // CYCLE_LENGTH // (MIN_COMMITTEE_SIZE * 2)) + 1 slots_per_committee = 1 else: committees_per_slot = 1 slots_per_committee = 1 while len(avs) * slots_per_committee < CYCLE_LENGTH * MIN_COMMITTEE_SIZE and slots_per_committee < CYCLE_LENGTH: slots_per_committee *= 2 o = [] for i, height_indices in enumerate(split(shuffle(avs, seed), CYCLE_LENGTH)): shard_indices = split(height_indices, committees_per_slot) o.append([ShardAndCommittee(shard_id = crosslinking_start_shard + i * committees_per_slot // slots_per_committee + j, committee = indices) for j, indices in enumerate(shard_indices)]) return o

Here's a diagram of what's going on:

We also define:

def get_indices_for_slot(crystallized_state, active_state, slot): ifh_start = crystallized_state.last_state_recalc - CYCLE_LENGTH assert ifh_start <= slot < ifh_start + CYCLE_LENGTH * 2 return crystallized_state.indices_for_slots[slot - ifh_start]

def get_block_hash(crystallized_state, active_state, curblock, slot): sback = curblock.slot_number - CYCLE_LENGTH * 2 assert sback <= slot < sback + CYCLE_LENGTH * 2 return active_state.recent_block_hashes[slot - sback]

get_block_hash(, , h)

should always return the block in the chain at height h

, and get_indices_for_slot(, , h)

should not change unless the dynasty changes.

On startup

- Let x = get_new_shuffling(bytes([0] * 32), validators, 1, 0)

and set crystallized_state.indices_for_slots

to x + x

- Set crystallized_state.dynasty = 1
- Set crystallized_state.crosslink_records

to [CrosslinkRecord(dynasty=0, hash= bytes([0] * 32)) for i in range(SHARD_COUNT)]

- Set total_deposits

to sum([x.balance for x in validators])

- Set recent_block_hashes

to [bytes([0] * 32) for _ in range(CYCLE_LENGTH * 2)]

All other values in active and crystallized state can be set to zero or empty arrays depending on context.

Per-block processing

First, set recent_block_hashes

to the output of the following:

def get_new_recent_block_hashes(old_block_hashes, parent_slot, current_slot, parent_hash): d = current_slot - parent_slot return old_block_hashes[d:] + [parent_hash] * min(d, len(old_block_hashes))

The output of get_block_hash

should not change, except that it will no longer throw for current_slot - 1

, and will now throw for current_slot - CYCLE_LENGTH * 2 - 1

A block can have 0 or more AttestationRecord

objects, where each AttestationRecord

object has the following fields:

fields = { # Slot number 'slot': 'int64', # Shard ID 'shard_id': 'int16', # List of block hashes that this signature is signing over that # are NOT part of the current chain, in order of oldest to newest 'oblique_parent_hashes': ['hash32'], # Block hash in the shard that we are attesting to 'shard_block_hash': 'hash32', # Who is participating 'attester_bitfield': 'bytes', # The actual signature 'aggregate_sig': ['int256'] }

For each one of these votes [TODO]:

- Verify that slot < block.slot_number

and slot >= max(block.slot_number - CYCLE_LENGTH, 0)

- Compute parent_hashes

= [get_block_hash(crystallized_state, active_state, block, slot - CYCLE_LENGTH + i) for i in range(CYCLE_LENGTH - len(oblique_parent_hashes))] + oblique_parent_hashes

- Let attestation_indices

be get_indices_for_slot(crystallized_state, active_state, slot)[x]

, choosing x

so that attestation_indices.shard_id

equals the shard_id

value provided to find the set of validators that is creating this attestation record.

- Verify that len(attester_bitfield) == ceil_div8(len(attestation_indices))

, where ceil_div8 = (x + 7) // 8

. Verify that bits len(attestation_indices)....

and higher, if present (i.e. len(attestation_indices)

is not a multiple of 8), are all zero

- Derive a group public key by adding the public keys of all of the attesters in attestation_indices

for whom the corresponding bit in attester_bitfield

(the ith bit is (attester_bitfield[i // 8] >> (7 - (i %8))) % 2

) equals 1

- Verify that aggregate_sig

verifies using the group pubkey generated and hash((slot % CYCLE_LENGTH).to_bytes(8, 'big') + parent_hashes + shard_id + shard_block_hash)

as the message.

Extend the list of AttestationRecord

objects in the active_state

, ordering the new additions in the same order as they came in the block.

Verify that the slot % len(get_indices_for_slot(crystallized_state, active_state, slot-1)[0])

'th attester in get_indices_for_slot(crystallized_state, active_state, slot-1)[0]

is part of at least one of the AttestationRecord

objects; this attester can be considered to be the proposer of the block.

State recalculations

Repeat while slot - last_state_recalc >= CYCLE_LENGTH

:

For all slots s

in last_state_recalc - CYCLE_LENGTH ... last_state_recalc - 1

:

- Determine the total set of validators that voted for that block at least once
- Determine the total balance of these validators. If this value times three equals or exceeds the total balance of all active validators times two, set last_justified_slot = max(last_justified_slot, s)

and justified_streak += 1

. Otherwise, set justified_streak = 0

- If justified_streak >= CYCLE_LENGTH + 1

, set last_finalized_slot = max(last_finalized_slot, s - CYCLE_LENGTH - 1)

- Remove all attestation records older than slot last_state_recalc

Also:

- Set last_state_recalc += CYCLE_LENGTH
- Set indices_for_slots[:CYCLE_LENGTH] = indices_for_slots[CYCLE_LENGTH:]

For all (shard_id

, shard_block_hash

) tuples, compute the total deposit size of validators that voted for that block hash for that shard. If this value times three equals or exceeds the total balance of all validators in the committee times two, and the current dynasty exceeds crosslink_records[shard_id].dynasty

, set crosslink_records[shard_id] = CrosslinkRecord(dynasty=current_dynasty, hash=shard_block_hash)

.

TODO:

- Rewards for FFG participation
- Rewards for committee participation

Dynasty transition

TODO. Stub for now.

Note: this is ~70% complete. Main sections missing are:

- Validator login/logout logic
- Logic for the formats of shard chains, who proposes shard blocks, etc. (in an initial release, if desired we could make crosslinks just be Merkle roots of blobs of data; in any case, one can philosophically view the whole point of the shard chains as being a coordination device for choosing what blobs of data to propose as crosslinks)
- Logic for inducting queued validators from the main chain
- Penalties for signing or attesting to non-canonical-chain blocks (update: may not be necessary, see Attestation committee based full PoS chains)
- Slashing conditions
- Logic for withdrawing deposits to shards
- Per-validator proofs of custody

- Full rewards and penalties
- Versioning and upgrades

Slashing conditions may include:

Casper FFG height equivocation Casper FFG surround Beacon chain proposal equivocation Shard chain proposal equivocation Proof of custody secret leak Proof of custody wrong custody bit Proof of custody no secret reveal RANDAO leak