Ethereum 2.0 Mauve Paper

Because mauve has the most RAM...

Over the past decade, projects such as Bitcoin, Namecoin and Ethereum have shown the power of cryptoeconomic consensus networks to bring about the next stage in evolution of decentralized systems, potentially expanding their scope from simply providing for the storage of data and messaging services to managing the "back-end" of arbitrary stateful applications. Proposed and implemented applications for such systems range from globally accessible cheap payment systems to financial contracts, prediction markets, registration of identity and real world property ownership, building more secure certificate authority systems and even keeping track of the movement of manufactured goods through the supply chain.

However, there remain serious efficiency concerns with the technological underpinnings of such systems. Because every full node in the network must maintain the entire state of the system and process every transaction, the network can never be more powerful than a single computer. The consensus mechanism most often used in existing systems, proof of work, consumes a very large amount of electricity in order to operate; the largest working blockchain using this mechanism, Bitcoin, has been shown to consume as much electricity as the entire country of Ireland.

This document proposes a solution to these problems based on the combination of proof of stake and sharding. Proof of stake itself is not a novel idea, having existed since 2011, but this new algorithm presents substantial benefits, both solving flaws in previous systems and even introducing new properties that are not present in proof of work. Proof of stake can be thought of as a kind of "virtual mining": whereas in proof of work, users can spend real-world dollars to buy real computers which expend electricity and stochastically produce blocks at a rate roughly proportional to the cost expended, in proof of stake, users spend real-world dollars to buy virtual coins inside the system, and then use an in-protocol mechanism to convert the virtual coins into virtual computers, which are simulated by the protocol to stochastically produce blocks at a rate roughly proportional to the cost expanded - replicating the exact same effect but without the electricity consumption.

Sharding is also not novel, having existed in distributed database design for over a decade, but research into its application for blockchains has so far been very limited. The basica approach is to solve scalability challenges via an architecture in which nodes from a global validator set (in our case created through proof of stake bonding) are randomly assigned to specific "shards", where each shard processes transactions in different parts of the state in parallel, thereby ensuring that work is distributed across nodes rather than being done by everyone.

We desire the following primary goals:

1. **Efficiency via proof of stake** - the consensus should be secured without mining, thereby greatly reducing electricity waste as well as the need for large and indefinite ongoing ETH issuance.
2. **Fast block time** - the block time should be maximally fast, though without compromising security.
3. **Economic finality** - once a block is made, after a certain amount of time a state of affairs should arise where the bulk of the validators have "fully committed" to that block, meaning that they lose their entire ETH deposits (think: 10 million ETH of value) in all histories that do not have that block. This is desirable because it means that even majority collusions cannot conduct medium or long-range 51% attacks without destroying all of their ether; the default validator strategies are designed to be conservative about their willingness to make high-value commitments so the risk for honest validators should be very low.
4. **Scalability** - it should be possible to run the blockchain with literally no full nodes, ie. in a situation where all nodes, including validators, keep up with only a small fraction of the data in the blockchain and use light-client techniques in order to access the rest. This way the blockchain can achieve transaction throughput much higher than that of a single machine while at the same time ensuring that the platform can be run off of nothing more than a sufficiently large number of consumer laptops, thereby preserving decentralization.
5. **Cross-shard communication** - it should be maximally feasible to interoperate between applications that are on different parts of the state that are stored by different nodes, and to build applications that exist across multiple such portions of the state if an individual application's usage reaches such a point that a single node's computing power and bandwidth can no longer sustain it.
6. **Computational censorship resistance** - the protocol should be resistant to attempts by even majority colluding validators across all shards to prevent undesired transactions from getting into the chain and being finalized. This exists to some extent in Ethereum 1.0 via "censorship resistance by halting problem", but we can make this mechanism much stronger by introducing a notion of guaranteed scheduling and guaranteed cross-shard messages.

We start off by describing an algorithm that achieves only (1) and (2), then in a second algorithm add (3), and in a third algorithm add (4) and (5) to a limited extent (the proviso being a limit, roughly proportional to the square of a node's computational capacity, in the case of (4) and a 24 hour delay for cross-shard messages, with the possibility to build faster messaging as a layer-on-top via dual-purposing deposits, in the case of (5)). Stronger degrees of satisfying (4) and (5), as well as any degree of (6), are left off the table for 2.0, to be revisited for Ethereum 2.1 and 3.0.

Constants

We set:

- `BLOCK_TIME`: 4 seconds (aiming on the less ambitious side to reduce overhead)
- `SKIP_TIME`: 8 seconds (aiming on the less ambitious side to reduce overhead)
- `EPOCH_LENGTH`: 10800 (ie. 12 hours under good circumstances)
- `ASYNC_DELAY`: 10800 (ie. 12 hours under good circumstances)
- `CASPER_ADDRESS`: 255
- `WITHDRAWAL_DELAY`: 10000000, ie. 4 months
- `GENESIS_TIME`: some future timestamp marking the start of the blockchain, say 1500000000
- `REWARD_COEFFICIENT`: 3 / 1000000000
- `MIN_DEPOSIT_SIZE`: 32 ether
- `MAX_DEPOSIT_SIZE`: 131072 ether
- `V_LOSS_MAXGROWTH_FACTOR`: 32
- `FINALITY_REWARD_COEFFICIENT`: 0.6 / 1000000000
- `FINALITY_REWARD_DECAY_FACTOR`: 1000 (ie. 1.1 hours under good circumstances)
- `MIN_BET_COEFF`: 0.25
- `NUM_SHARDS`: 80
- `VALIDATORS_PER_SHARD`: 120

Minimal Proof of Stake

Note: an understanding of Ethereum 1.0 is assumed in this and later sections

We can create a minimal proof of stake algorithm, without finality, additional censorship resistance or sharding as follows. We specify that at address `CASPER_ADDRESS` there exists a "Casper contract", which keeps track of a validator set. The contract has no special privileges, except that calling the contract is one part of the process for validating a block header and that it is included in the genesis block rather than being added at runtime via a transaction. The validator set starts off as being some set determined in the genesis, and can be modified through the following functions:

- `deposit(bytes validation_code, bytes32 randao, address withdrawal_address)`: accepts a deposit containing a certain amount of ether. The sender specifies a piece of "validation code" (EVM bytecode, serving as a sort of public key that will be used to verify blocks and other consensus messages signed by them), a randao commitment (a 32-byte hash used to facilitate validator selection; see below) and the address that the eventual withdrawal will go to. Note that the withdrawal can go to an address which itself only releases funds under certain conditions, allowing dual-use of the security deposit if desired. If all parameters are accepted, this adds a validator to the validator set from the start of the epoch after the next (ie. if `deposit` is called during epoch `n`, the validator joins the validator set at the start of epoch `n+2`, where an epoch is a period of `EPOCH_LENGTH` blocks). The hash of the validation code (called the `vchash`) can be used as an ID for the validator; multiple validators with the same validation code are forbidden.
- `startWithdrawal(bytes32 vchash, bytes sig)`: begins the withdrawal process. Requires a signature that passes the validation code of the given validator. If the signature passes, then withdraws the validator from the validator set from the start of the epoch after the next. Note that this function does not withdraw any ether.

There is also a function:

- `withdraw(bytes32 vchash)`: Withdraws a validator's ether, plus rewards minus penalties, to the validator's withdrawal address, as long as the validator has withdrawn from the active set of validators using `startWithdrawal` at least `WITHDRAWAL_DELAY` seconds ago.

Formally speaking, the validation code is a piece of code that takes as input a block header hash, plus a signature, and returns 1 if the signature is valid and 0 otherwise. This mechanism ensures that we do not lock validators into any one specific signature algorithm, instead allowing validators to use validation code that verifies signatures from multiple private keys instead of a single one, use Lamport signatures if quantum-resistance is desired, etc. The code is executed in a black box environment using a new `CALL_BLACKBOX` opcode to ensure execution is independent of external state; this is important to prevent possible tricks where a validator creates a piece of validation code that returns 1 under circumstances favorable to the validator and 0 under circumstances unfavorable to the validator (eg. dunkle inclusion).

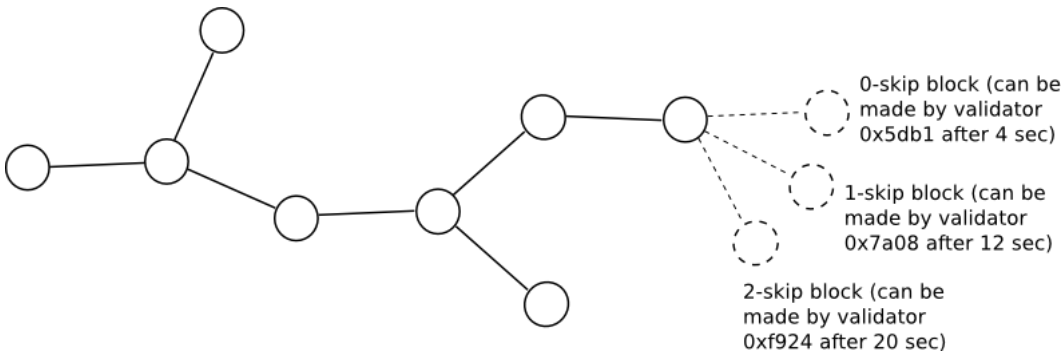
The `randao` value provided in the `deposit` function should be the result of computing a long chain of hashes, ie. `sha3(sha3(sha3(.....(sha3(x))...)))` for some secret `x`. The `randao` value provided by each validator is saved in the Casper contract's storage.

The Casper contract also contains a variable called `globalRandao`, initialized as 0. The contract contains a function `getValidator(uint256 skips)` which returns the validation code of the validator that is eligible to create the next block after the given number of "skips", ie. `getValidator(0)` returns the first validator (the validator that would normally create the block), `getValidator(1)` returns the second validator (the validator that would create the block if the first is unavailable), etc. Each one of these validators is pseudorandomly selected from the current active validator set, with the randomness weighted by the size of the validator's initial deposit, and seeded by the Casper contract's `globalRandao` value. Alongside the signature, a valid block must also contain the preimage of the currently saved `randao` for that validator; this preimage then replaces the saved `randao` value, and is also XORed into the contract's `globalRandao`. Hence, each block that a validator creates will require "unrolling" one layer of the validator's `randao`. This is an implementation of the in-chain randomness algorithm described with justification here.

Altogether, the data that a block must include in its extra data is as follows:

```
<vchash> <randao> <sig>
```

Where `vchash` is the 32-byte hash of the validation code, used to quickly identify the validator, `randao` is the (also 32-byte) randao reveal as described above, and `sig` is the signature, which can be of arbitrary length (though we limit the block header size to 2048 bytes).



The minimum time after which a block can be created is defined simply: `GENESIS_TIME + BLOCK_TIME * <block height> + SKIP_TIME * <total number of skipped validators since genesis>`. This in practice means that once a given block is published, the 0-skip validator for the next block can publish after `BLOCK_TIME` seconds, the 1-skip validator after `BLOCK_TIME + SKIP_TIME` seconds, and so forth. If a validator publishes a block too early, other validators will ignore that block until the prescribed time, and only then process it (this mechanism is further described, with justification, here; the asymmetry between the short `BLOCK_TIME` and longer `SKIP_TIME` ensures that average block times can be very short in the normal case while still retaining survivability under longer network latencies).

If a validator produces a block that gets included in the chain, they receive a block reward equal to the total amount of ether in the active validator set during that epoch, multiplied by the `REWARD_COEFFICIENT * BLOCK_TIME`. `REWARD_COEFFICIENT` thus essentially becomes an "expected per-second interest rate" for the validator if they always act correctly; multiply by ~32 million to get the approximate annual interest rate. If a validator produces a block that goes not get included in the chain, then this block's header can be included in the chain at any point in the future (up until the validator calls `withdraw`) as a "dunkle" via the Casper contract's `includeDunkle(header:str)`; this causes the validator to *lose* money equal to the block reward (as well as giving the party that includes the dunkle a small part of the penalty as incentive). Hence, a validator should only make a block if they are more than 50% certain that this block will make it into the chain; this discourages validating on all chains at once. The validators' cumulative deposits, including rewards and penalties, are stored in the state of the Casper contract.

The purpose of the "dunkle" mechanism is to solve the "nothing at stake" problem in proof of stake, where if there are no penalties but only rewards then validators are incentivized to try to make blocks on top of every possible chain. In proof of work, there is a cost to creating blocks, and so only creating blocks on top of the "main chain" is profitable. The dunkle mechanism attempts to replicate the economics of proof of work, creating an artificial penalty for creating off-chain blocks to substitute the "natural penalty" of electricity cost in proof of work.

Assuming a constant-sized validator set, we can define the fork-choice rule easily: count blocks, longest chain wins. Assuming the validator set can grow and shrink, however, this will not work well, as a minority-supported fork will eventually start producing blocks just as quickly as a majority-supported fork. Hence, we instead define the fork-choice rule by counting blocks, giving each block a weight equal to the block reward. Since block rewards are proportional to the total amount of ether actively validating, this ensures that chains with more actively validating ether "grow" in score faster.

We can see that this rule can conveniently be understood in another way: *fork choice as value-at-loss*. The principle is that we select the chain on which validators have put the most value *at stake*, ie. the chain such that validators have committed to lose the most money in all other chains except for that one. We can also equivalently view this as the chain in which validators lose the least money. In this simple model, it is easy to see how this corresponds to simply the longest chain with blocks weighted by block reward.

This algorithm is simple, but is arguably sufficient for a proof-of-stake implementation.

Adding finality

The next step is to add the notion of economic finality. We do this as follows. Inside of the block header, in addition to pointing to the hash of the previous block, a validator now also makes a claim on what the probability is that some previous block `FINALIZATION_TARGET` will be finalized. The claim is phrased as a bet, as in "I believe that block `0x5e81d...` will be finalized, and I am willing to lose `V_LOSS` in all histories where this is false provided that I gain `V_GAIN` in all histories where this is true". The validator chooses a parameter `odds`, and `V_LOSS` and `V_GAIN` are computed as follows (letting `total_validating_ether` be the total amount of ether in the active validator set, `MAX_REWARD` be the maximum allowed block reward and `bet_coeff` be a coefficient defined later):

- `BASE_REWARD = FINALITY_REWARD_COEFFICIENT * BLOCK_TIME * total_validating_ether`
- `V_LOSS = BASE_REWARD * odds * bet_coeff`
- `V_GAIN = BASE_REWARD * log(odds) * bet_coeff`

`V_GAIN` and `V_LOSS` values relative to the `BASE_REWARD`, assuming `bet_coeff = 1`

`FINALIZATION_TARGET` starts off null, though during block 1 it is set to block 0. `bet_coeff` is initially (ie. at genesis) set to 1, and another variable `cached_bet_coeff` is set to 0. However, every block, we set `bet_coeff -= bet_coeff / FINALITY_REWARD_DECAY_FACTOR`, and `cached_bet_coeff -= cached_bet_coeff / FINALITY_REWARD_DECAY_FACTOR`, though `bet_coeff` cannot decrease below `MIN_BET_COEFF` (this ensures that there is always at least some incentive to bet). When producing a block, a validator gains `BASE_REWARD * log(MAXODDS) * cached_bet_coeff`, where `MAXODDS` is the maximum possible odds, ie. `MAX_DEPOSIT_SIZE / BASE_REWARD`. What this mechanism accomplishes is that once a block is finalized, validators get rewards from it as though they continue to validate it betting with maximum odds; this ensures that there are no perverse incentives for validators to collude to delay finalizing a block in an attempt to get maximum revenues.

When the Casper contract determines that the `FINALIZATION_TARGET` has been finalized (ie. the total value-at-loss that it knows about for that block passes some threshold), we set the new `FINALIZATION_TARGET` to equal the current block, we set `cached_bet_coeff += bet_coeff` and `bet_coeff` is reset to 1. Starting from the next block, the finalization process begins anew for the new `FINALIZATION_TARGET`. If there is a short-range chain split, there may be finalization processes going on for multiple blocks at the same time, which may even be at different heights; however, given the default validator strategy of betting on the block with the highest value-at-loss backing it, we expect the process to converge toward choosing one of them (convergence arguments here are basically the same as those for minimal proof of stake).

When finalization starts for a new block, we expect that at first the odds would be low, signifying validators' fear of short-range forks, but over time the odds that validators are willing to bet would increase. Particularly, validator bets would increase even more if they see that other validators have themselves put high-odds bets behind that block. The expectation is that value-at-loss on the block would increase exponentially, thereby hitting the "total deposit loss" maximum in logarithmic time.

In the block header's extra data, we now change the required data format to the following:

```
<vchash> <randao> <blockhash> <logodds> <sig>
```

Where `blockhash` is the prior block hash that the bet is made on, and `logodds` is a 1-byte value representing the `odds` in logarithmic form (ie. 0 correponds to 1, 8 corresponds to 2, 16 corresponds to 4, etc).

Note that we cannot allow validators total freedom in setting `odds`. The reason is that if there are two competing finalization targets, B1 and B2 (ie. there exist two chains, in one of which `FINALIZATION_TARGET` is set to B1 and in the other `FINALIZATION_TARGET` is set to B2), and a consensus starts forming around B1, then a malicious validator may suddenly place a high-odds bet on B2, with sufficient value-at-loss to sway the consensus, and thereby trigger a short-range fork. Hence, we limit `odds` by limiting `V_LOSS`, using the following rules:

- Let `V_LOSS_EMA` be an exponential moving average, set as follows. `V_LOSS_EMA` starts out equal to the block reward. During each block, `V_LOSS_EMA is set to V_LOSS_EMA * (V_LOSS_MAXGROWTH_FACTOR - 1 - SKIPS) / V_LOSS_MAXGROWTH_FACTOR + V_LOSS` where `SKIPS` is the number of skips and `V_LOSS` is the `V_LOSS` selected in that block.
- Set `V_LOSS_MAX` to `V_LOSS_EMA * 1.5`. Limit `V_LOSS` values to this value.

This rule is designed to incorporate a safety constraint: a validator can only risk `1.5x` after at least two thirds of (a sample of) other validators have risked `x`. This is similar in spirit to precommitment/commitment patterns in Byzantine-fault-tolerant consensus algorithms where a validator waits for two thirds of other validators to take a given step before taking the next step, and ensures some degree of safety as well as ensuring that even a majority collusion cannot engage in "griefing" attacks (ie. getting other validators to place large value-at-loss behind a block and then pushing the consensus the other way) without the collusion also itself incurring large expense (in fact, the collusion would lose money faster than the victims; this is a great property as it ensures that even under hostile-majority conditions bad actors can often be "weeded out" over time).

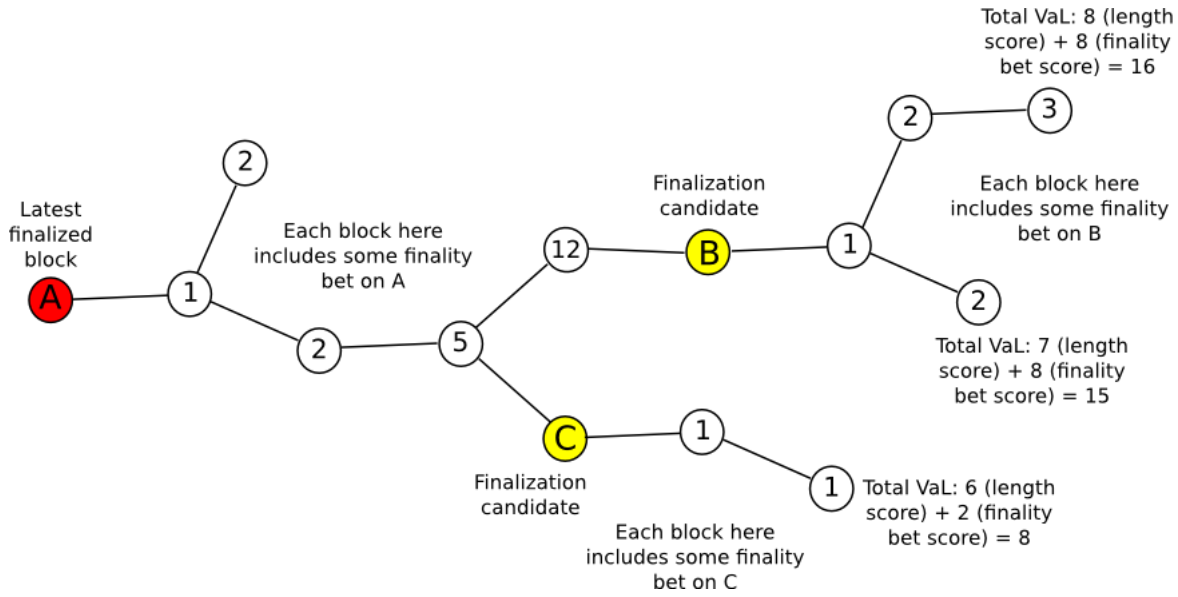
If a block is included as a dunkle, the bets are processed and both penalties and rewards may result. For example, if there are two blocks A1 and A2 at height 5000 that are competing finalization targets, and two blocks B1 and B2 (both with A1 as an ancestor) at height 5050, and a validator makes a block C on top of B1 making a bet for A1, then if B2 ends up being in the main chain, B1 and C will become dunkles, and so C will be penalized for "getting it wrong" on B1 vs B2 but still rewarded for "getting it right" on A1.

However, suppose that the `v_loss` in C is such that `v_loss < v_loss_max` if B1 is included, but `v_loss > v_loss_max` if B2 is included. Then, in order to preserve the desired value-at-loss properties, we institute an additional penalty: we penalize the validator by `v_loss - v_loss_max` even if their bet is correct. Thereby, we effectively decompose the bet of size `v_loss` into a combination of (i) a bet with value-at-loss `v_loss_max` and (ii) a pure destruction of `v_loss - v_loss_max`, thereby ensuring that this excessively sized bet still only shifts the fork choice rule by `v_loss_max`. This does mean that bets are in some sense not "pure", as a bet on some block may lead to a penalty even if that block ends up being finalized if too many of its children get forked off. The loss of purity in the betting model is deemed an acceptable tradeoff for the gain in purity in the value-at-loss fork-choice rule.

Scoring and strategy implementation

The value-at-loss scoring can be implemented using the following algorithm:

- Keep track of the most recent finalized block. If there are multiple, return a big red flashing error, as this indicates that a finality reversion event has taken place and the user of the client should probably use extra-chain sources to determine what's going on.
- Keep track of all finalization candidates that are children of that block. For each one, keep track of the value-at-loss behind that candidate.
- Keep track of the longest chain from each finalization candidate and its length, starting from the most recent finalized block.
- The "total weight" of a chain is the value-at-loss of its finalization candidate ancestor plus the length of the chain times the block reward. If there is no finalization candidate in the chain, then the length of the chain times the block reward is used by itself. The "head" is the latest block in the chain with the highest total weight.

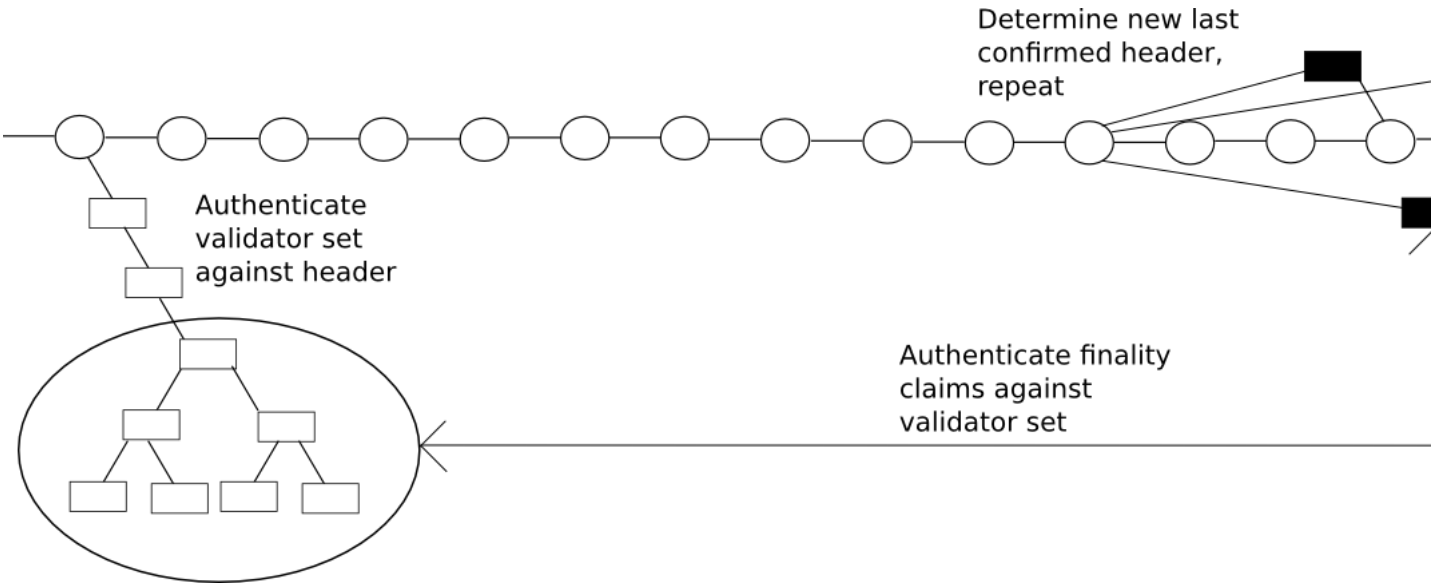


`v_loss` values here for illustrative purposes; in reality they will not be allowed to grow this quickly and a much higher `v_loss` on A will b required for B or C to become finalization candidates.

A simple validator strategy is to try to create blocks only on the head, and to make finality bets where the value-at-loss is 80% of the prescribed maximum.

Light-client syncing

This finality mechanism opens the door to a very fast light-client syncing algorithm. The algorithm consists of the following steps:

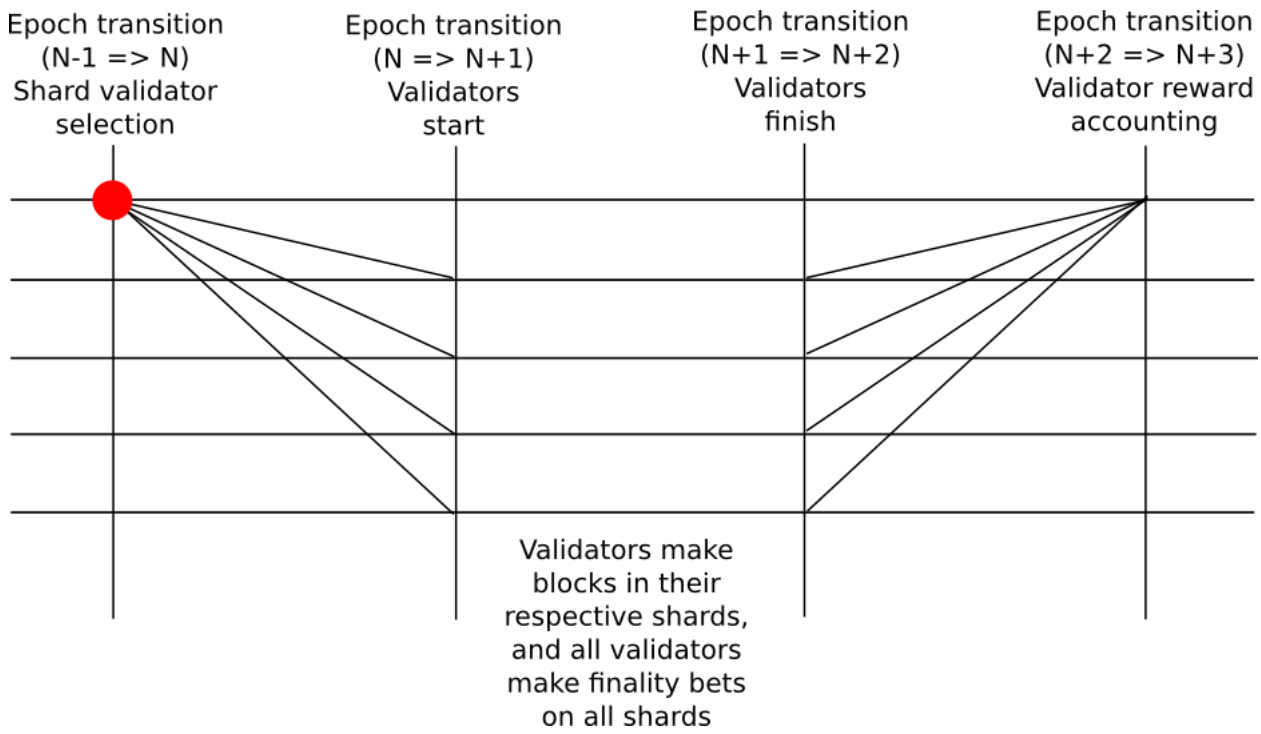


1. Let `x` be the latest state you already have confirmed (initially, the genesis state).
2. Ask the network for the most recent finalization target within either the same epoch as `x` or the epoch after that (remember: the finalization target is the block during which the protocol deems the previous finalization target to be finalized). Call the most recent finalization target `Fn` and the previous finalization target `Fp`
3. Ask the network for the `k` blocks right before `Fn`. These blocks will make bets that stake their entire pool of ether on `Fp`.
4. Authenticate the validators making those blocks by making Merkle branch queries against your already-finalized state, to verify their presence and position in the validator set, and against the pre-state of the first of the `k` blocks, to verify that they were selected correctly.
5. Set `x` to the post-state of `Fp`.
6. Repeat until you reach the most recent finalized block. From there, use the normal strategy above to find the head of the chain.

Note that steps 1 through 5 allow you to light-verify an entire day's worth of blocks in what could likely be optimized down to two network requests and a couple seconds worth of computation.

Sharding

Now, we consider expanding from one shard to multiple shards. We construct a model as follows. Instead of having one single blockchain, we now have multiple interrelated chains, which we call "shards". There are `NUM_SHARDS` shards, numbered shard 0 to shard `NUM_SHARDS - 1`, where shard 0 simply operates as a regular proof-of-stake blockchain with finality as above, but shards `1 ... NUM_SHARDS - 1` work differently. At the start of every epoch, a random `VALIDATORS_PER_SHARD` validators are selected for each shard, and are assigned as validators for that shard for the next epoch (ie. the validators for epoch `n+1` are assigned at the start of epoch `n`). `getValidator(skips)`, when called to determine the validator within one of these shards, simply picks randomly (with an even distribution, as the deposit-size weighting was already done at selection time) from one of the selected validators. The "finality" bets for shards `1 ... NUM_SHARDS - 1` are not made inside the shards; rather, they are made in shard 0. When a bet is made, it is stored, and the bet is *evaluated* only after the end of the subsequent epoch (ie. finality claims for blocks during epoch `n+1` are evaluated in shard 0 at the start of epoch `n+3`).



Diagonal lines represent required cross-shard communication

If a validator has been selected for a shard, that validator will need to call the `registerForShard(bytes32 vchash, uint256 shard, uint256 index, bytes32 randao)` function of the Casper contract, where `vchash` is the validator's validation code hash, `shard` is the shard ID, `index` is a value where `0 <= index < VALIDATORS_PER_SHARD` where `getShardValidator(uint256 shard, uint256 index)` returns the given validation code hash, and `randao` is a randao commitment. This function generates a receipt that can then be confirmed on the target shard using `confirmReceipt(uint256 receiptId)` in order to induct the validator.

`getShardValidator` is itself a function with similar logic to `getValidator`, though it relies on a separate source of randomness. This source of randomness is derived as follows:

1. During each epoch, for `0 <= k < 24`, keep track of the number of times that the kth last bit of the `globalRandao` is equal to 1 minus the number of times that the kth bit is equal to 0.
2. At the end of each epoch, let `combinedRandao` be the value such that for each `0 <= k < 24`, the kth bit is equal to 1 if during that block there were more 1s in the `globalRandao`'s during that epoch and 0 otherwise. Bits above the 24th are all zero. Use `sha3(combinedRandao)` as the source of randomness.

This uses Iddo Bentov's low-influence functions to increase the cost of exploitation for this source of randomness, as this particular random seed has substantial economic consequence and will thus be a greater-than-normal target for exploitation.

Cross-shard finality bets are NOT in the block header, so as not to overly bog down light clients; instead, the validator is expected to create a transaction calling a `registerFinalityBets(bytes32[] hashes, bytes logodds)` function during any block that they create that expects `NUM_SHARDS` hashes and a byte array `NUM_SHARDS` bytes long with each byte representing odds for the corresponding block hash.

The typical workflow for a validator would be to maintain a "full node" for shard 0, and also keep track of which future shards they will be assigned to. If a validator is assigned to a shard, they would download the state using Merkle tree proofs and make sure that they have the state downloaded when they need to start validating. For that epoch, they would act as a validator for that shard and make blocks; at the same time, they would make finality bets on all shards, where they would infer what to bet on by looking at (i) what the longest chain is on each shard, (ii) other validators' finality bets, and (iii) various secondary heuristics and mechanisms that try to catch successful 51% attacks within a shard (eg. fraud proofs). Note that the probability of being assigned to any given shard is proportional to the validator's deposited ether; hence, a validator that is twice as rich would have to do twice as much computation. This property is considered desirable, as it increases fairness and reduces pooling incentives, and introduces an element where processing transactions and storing the blockchain itself becomes a form of "hybrid proof of work".

The intention of the sampling mechanism is to ensure that the system is secure against even attackers that have up to ~33-40% of the total deposited ether (less than 50% because an attacker with 33-50% of deposits may "get lucky" on some given shard), while only relying on a few validators to actually verify transactions; because the sampling is random, attackers cannot choose to concentrate their stake power on a single shard, a fatal flaw of many proof-of-work sharding schemes. Even if a shard does get attacked, there is a second line of defense: if the other validators see evidence of an attack, then they can refuse to make finality claims that follow the attacker's forks, instead confirming the chain that appears to be created by honest nodes. If an attacker on a shard tries to create a chain out of invalid blocks, validators of other shards can detect this, and then temporarily become fully validating nodes on that shard, and make sure that they are only finalizing valid blocks.

Cross-shard communication

Cross-shard communication in this scheme works as follows. We create a `ETHLOG` opcode (with two arguments `to`, `value`), which creates a log whose stored topic is the empty string (note that's the empty string, not 32 zero bytes, a traditional log can only have 32-byte strings as topics), and whose data is a 64-byte string

containing the destination and the value. We create an opcode `GETLOG`, which takes as a single argument an ID defined by `block.number * 2**64 + txindex * 2**32 + logindex` (where `txindex` is the index of the transaction that contained the log in the block, and `logindex` is the index of the log in the transaction's receipt), tries to obtain that log, stores a record in the state saying that the log has been consumed, and places the log data into a target array. If the log has the empty string as a topic, this also transfers the ether to the recipient. In order for a log to be obtained successfully, the transaction that calls this opcode must reference the log ID. If `v = 0`, we allow the `r` value in the signature to be repurposed for this (note: this means that only EIP 86 transactions can be used here; we hope that by this time EIP 86 transactions will be the dominant form of transaction).

The consensus abstraction is now no longer a single chain; rather, it is a collection of chains, `c[0] ... c[NUM_SHARDS - 1]`. The state transition function is now no longer `stf(state, block) -> state'`; rather, it's `stf(state_k, block, r_c[0] ... r_c[NUM_SHARDS - 1]) -> state_k'` where `r_c[i]` is the set of receipts from chain i from more than `ASYNC_DELAY` blocks in the past.

Note that there are several ways to "satisfy" this abstraction. One is the "everyone is a full node" approach: all nodes store the state of all shards, update the chains of all shards, and thereby have enough information to compute all state-transition functions. However, this is uninteresting as it is not scalable.

The more interesting strategy is the "medium node" approach: most nodes select a few shards that they keep up with fully (likely including shard 0), and act as light clients for all other shards. When computing the state transition function, they need the old receipts, however they do not store them; instead, we add a network protocol rule requiring transactions to come with Merkle proofs of any receipts that their transactions statically reference (here it becomes clear why static referencing is required: otherwise if an arbitrary `GETLOG` operation can be made at runtime, fetching data for the logs would become a slow process bottlenecked multiple times by network latency, and there would be too much pressure for clients to store all historical logs locally). The strategy that is ultimately deployed in a live network would likely be the the full-node strategy with mandatory receipt Merkle proofs initially, loosening to encourage more and more medium nodes over time.

Note that importing a Merkle proof, as a packet of data, from one shard directly into the other shard is not required; instead, the proof-passing logic is all done at validator and client level and serves to implement an interface at the protocol level where access to information that can be Merkle-proven is assumed. The long `ASYNC_DELAY` reduces the likelihood that a reorganization in one shard will require an intensive reorganization of the entire state.

If shorter delays are desired, one mechanism that can be implemented on top of the protocol is an in-shard betting market, ie. A can make a bet with B in shard `j` saying "B agrees to send 0.001 ETH to A if block X in shard `i` is finalized, and in exchange A agrees to send 1000 ETH to B if that block is not finalized". Casper deposits can be dual-used for this purpose - even though the bet itself happens inside of shard `j`, information about A losing would be transmitted via receipt to shard 0 where it can then transfer the 1000 ether to B once A withdraws. B would thus gain confidence that A is convinced enough that the block on the other shard will be finalized to make the bet, and also gains a kind of insurance against A's judgement failing (though if the dual-use scheme is used the insurance is imperfect, as if A is malicious they may lose their entire bet and so have nothing to give B).

There is a scalability limitation on this scheme, proportional to the square of a node's computing power, for two reasons. First, an amount of computing power proportional to the number of shards must be done to compute rewards on shard 0. Second, all clients must be light clients of all shards. Hence, if nodes have N computing power, then there should be O(N) shards with each shard having O(N) processing capacity, for a net total capacity of $O(N^2)$. Going above this maximum will require a somewhat more complex sharding protocol, composing claim validations in some kind of tree structure; this is out of scope here.

Future work

- Reduce the `ASYNC_DELAY` in such a way that cross-shard receipts are available as soon as the other shard is finalized.
- Reduce the `ASYNC_DELAY` to something as low as several times network latency. This allows for much smoother cross-shard communication but at the cost of a reorg in one shard potentially triggering reorgs in other shards; a mechanism needs to be devised to effectively handle and mitigate this.
- Create a notion of "guaranteed cross-shard calls". This comes in two components. First, there is a facility that allows users to buy "future gas" in some shard, giving them insurance against gas supply shocks (possibly caused by an attacker transaction-spamming the network) and second, a mechanism where if a receipt from shard `i` to shard `j` is made, and there is gas available, then that receipt must be included as quickly as possible or else validators that fail to include the receipt get penalized (with penalties increasing quickly to eventually equal the validator's entire deposit). This ensures that if a cross-shard receipt is made, that receipt **will** be processed within a short (perhaps <10 blocks) timeframe.
- Create a version of this scheme that does not carry the $O(N^2)$ limitation.
- Look into atomic transactions theory and come up with feasible notions of synchronous cross-shard calls.

Acknowledgements

Special thanks to:

- Gavin Wood, for the "Chain Fibers" proposal
- Vlad Zamfir, for ongoing research into proof of stake and sharding, and particularly the realization that state computation can be separated from block production
- Martin Becze, for ongoing research and consultation
- River Keefer, for reviewing
- Zoltu, for convincing me that single-letter variable names are evil and math really should adopt the long-standing basic principle of software development that variable names should be sufficiently descriptive so as to be maximally self-documenting