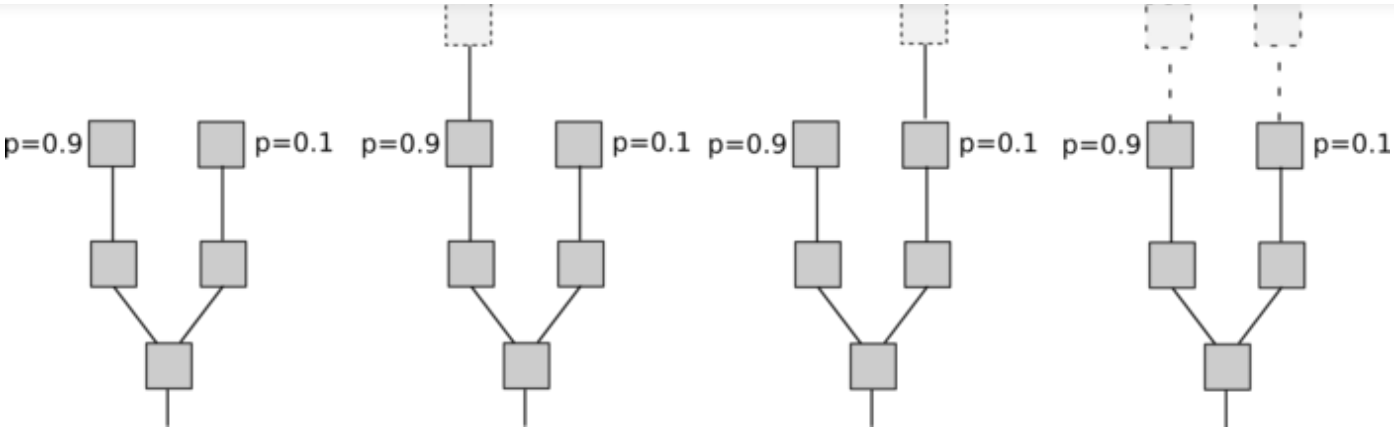
Vitalik Buterin Follow
Mar 4, 2017 · 9 min read · Listen

Safety Under Dynamic Validator Sets

Traditional consensus algorithms, whether they operate in a synchronous, partially asynchronous or fully asynchronous network model, and whether they are designed around simple faults, Byzantine faults or accountable faults, generally operate in a model where there is a fixed set of participants in the protocol, with the assumption that at least some fraction of that fixed set correctly follows the protocol.

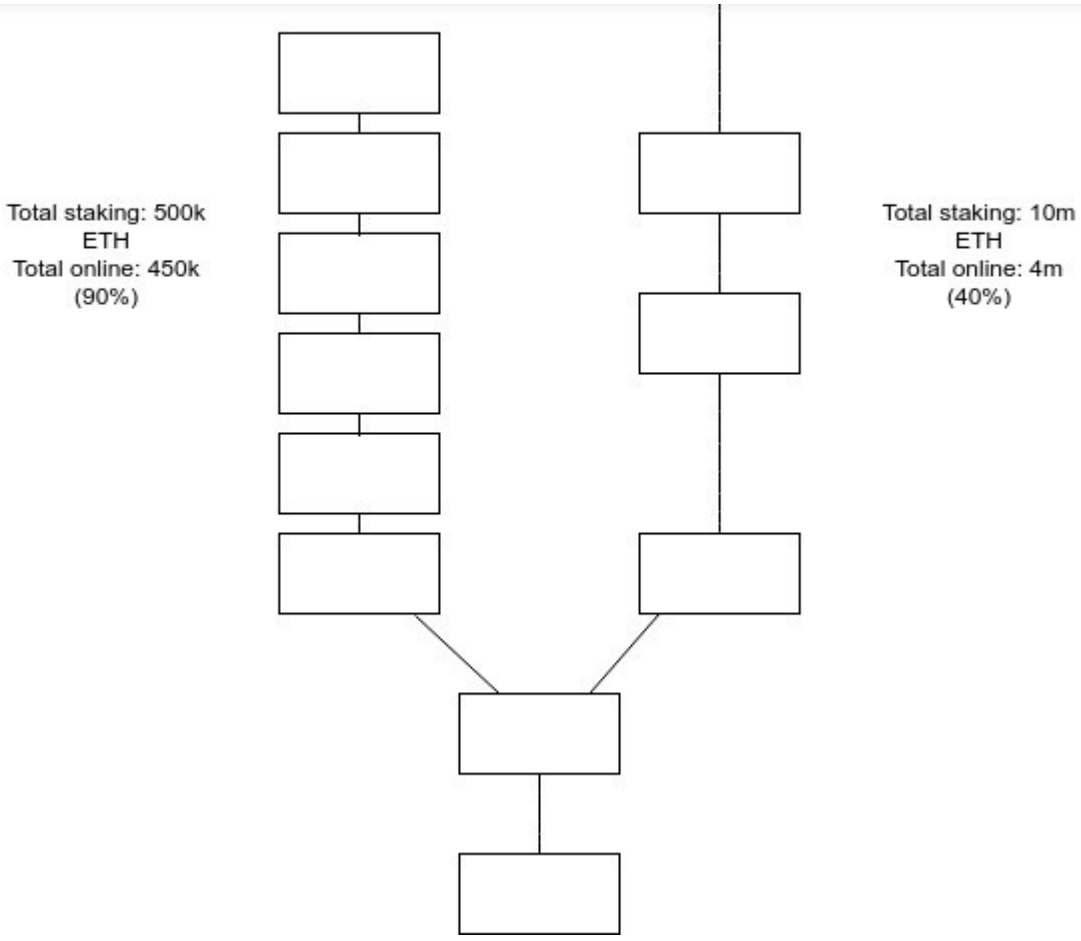
In proof of stake protocols, however, validators can come and go, and even the absolute size of the validator set can shrink and grow greatly over time. 80% of the validator set at one time may well be smaller than 20% of the validator set at another time, and what in a fixed-set model is clearly equivocation, in a dynamic-set model may not involve any equivocation at all. How do we deal with this?

The first case that is worth handling is **synchronous proof of stake**, ie. traditional chain-style proof of stake without finality (though possibly with a "dunkle inclusion" mechanic to penalize double-signing). This includes Peercoin-style algorithms, NXT-style algorithms, DPOS, the first algorithm described in the Mauve Paper, and basically everything else that is live right now. Here, the problem is simpler, as there is no possibility or goal of reaching or maintaining "economic finality"; instead, the goal is simply to create a positive-feedback mechanism that encourages validators to settle on one dominant block in any situation where there is a fork, and reinforce the stature of this block in the canonical chain over time.



Each validator's incentive in synchronous proof of stake (with dunkles) is to vote on the chain that they believe is most likely to end up being the canonical chain, and thereby themselves increase the chance that this will be the case; this creates a positive feedback loop, which causes the system to continuously converge on one ever-growing canonical chain.

With a fixed validator set, the fork-choice rule is simple: longest chain wins[1]. However, what if the validator set changes? Then, the attack that we need to avoid is as follows:



The left chain eventually grows “longer”, as block proposers show up 90% of the time on left and 40% of the time on the right, but it’s clearly not the chain that we want to be canonical — intuitively we want the chain that has the most staking ETH.

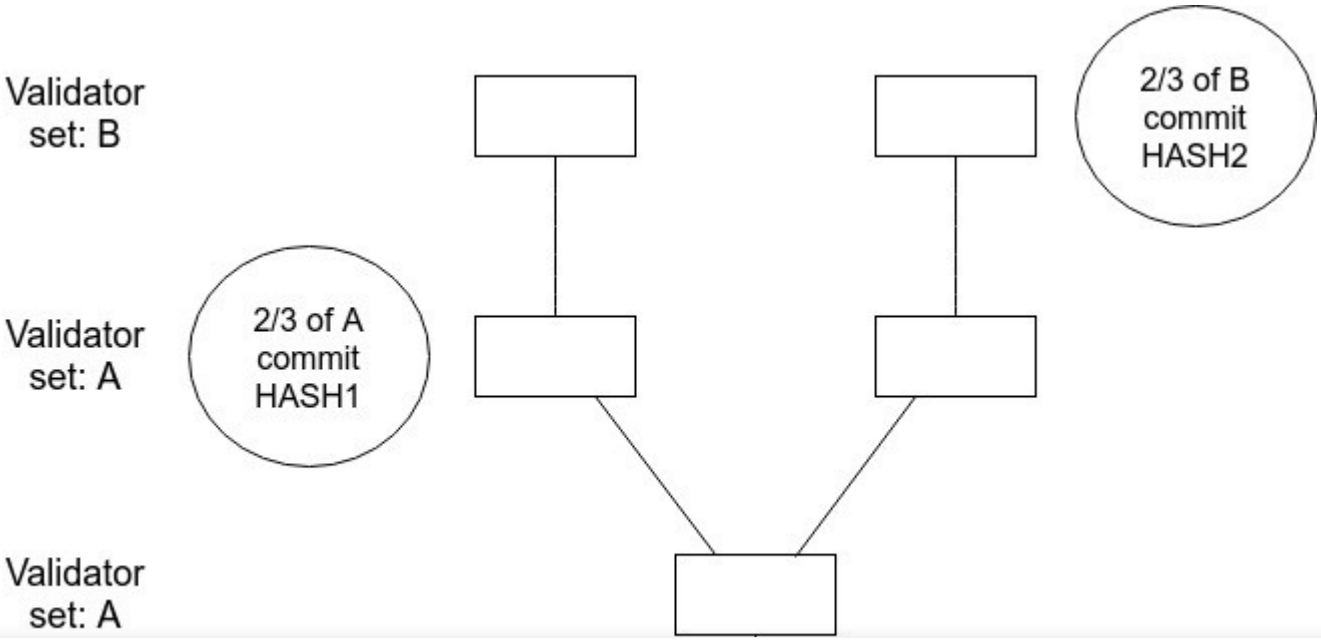
Suppose that a chain has 10 million ETH at stake, of which 450,000 (4.5%) is controlled by an attacker. Now, suppose that an attacker starts maintaining their own private chain starting from some block. They do not allow new validators to enter their chain, though they do include withdrawal transactions[2]. Over time, the only stake in the chain that remains belongs to the attacker, perhaps with a few exceptions. Even though this chain has much less ETH staking, the chain *looks* much “higher-quality”, with block makers showing up 90% of the time, whereas in the main chain block makers are ordinary users, and they may be offline much more often. Hence, a naive longest-chain model may well fall to this kind of attack.

To remedy this problem, we use the same trick as in proof of work: instead of just

Validator sets can now be swapped in and out arbitrarily, potentially even fully changing with every block. No matter what happens to the validator set, the economics still work the same way, creating incentives to converge toward the likely winning outcome, and chains backed by more ETH will win out over chains backed by less ETH.

The reason why the above case was fairly simple is that proof of stake schemes that penalize incorrectness evaluate every message independently of every other message, so the economics do not change even if the identities of the participants are swapped out every millisecond, as everyone has the same incentives no matter what their history is. Our more recent Casper designs, on the other hand, penalize *equivocation* — the act of a validator sending two messages that conflict with each other.

In that model, validator set changes are inherently difficult to deal with, as the safety proofs that work when you assume the validator set is the same no longer work if the validator set changes.



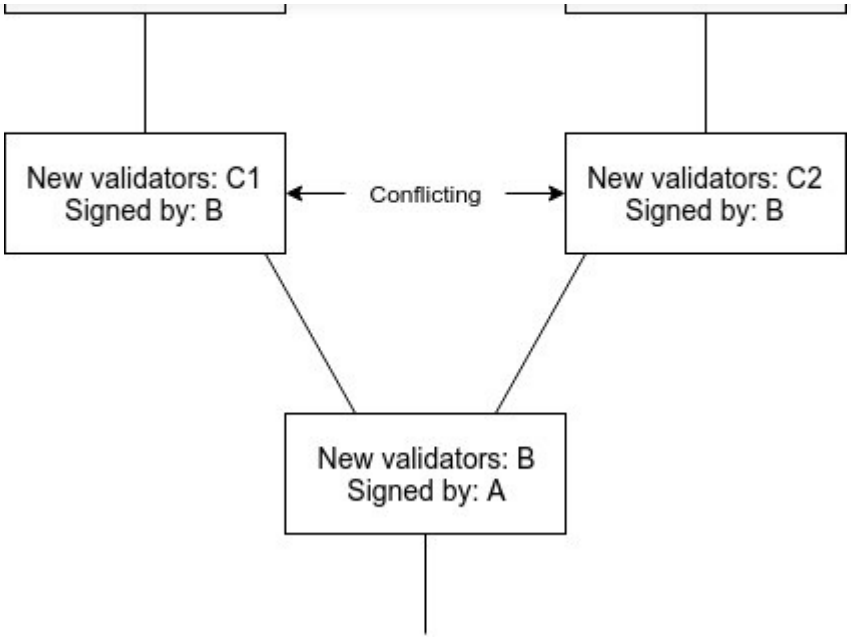
One possibility is to simply forbid the validator set from changing by more than, say, 1% per epoch, and thereby ensure that you maintain some degree of safety as long as a new block gets finalized at least once every 33 epochs. This actually works. However, it is highly suboptimal, and we can get much better results with a more thorough treatment of the problem.

Another immediate instinct that one might have is "in order to maintain continuity, we should wait for validator set A to finalize validator set B, and then let validator set B take the torch from there". To see how this works, consider the following toy algorithm:

- Every block must contain the hash of the validator set of the next block
- For a block to be valid, it must point to a valid parent block, as well as proof (eg. a sufficient number of COMMITs with the same sequence number) that the parent block was finalized. That is, finality must happen for every block.

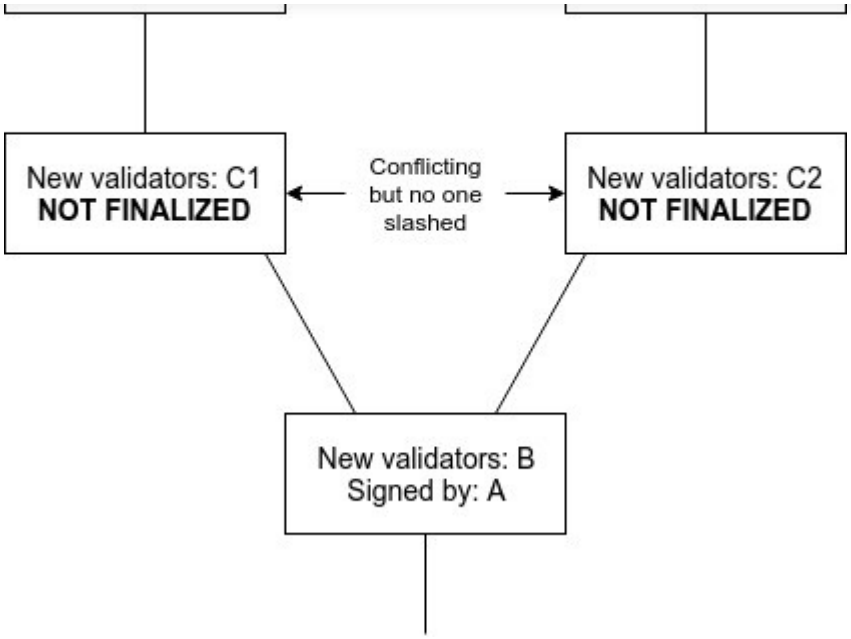
Note that here consensus happens between blocks, ie. there's a pattern of "create block 1, finalize block 1, create block 2, finalize block 2 ...", unlike our other algorithms where new blocks are being produced even while consensus on older blocks may still be finishing. In order for there to be a fork in this scheme, there must be some initial block height at which the blocks diverged, and so we can use our proof of accountable safety to show that at least 1/3 of the validators at that block height can be slashed.



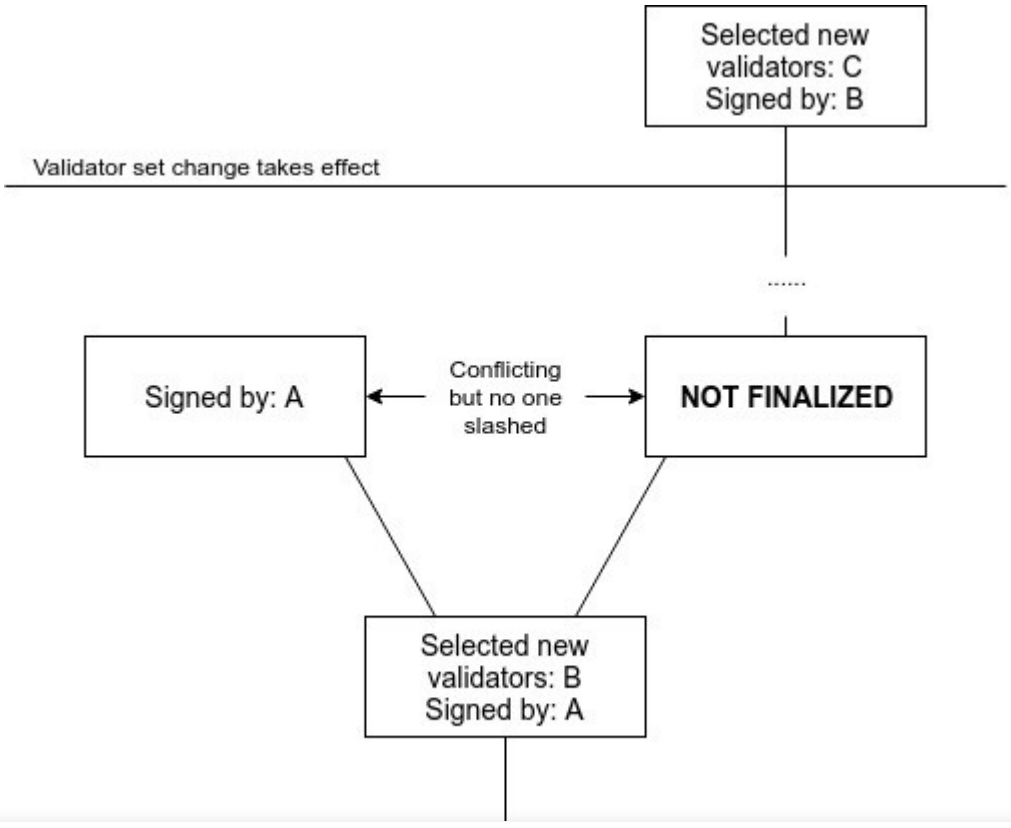



Graphical proof that if every block is finalized, and every block contains a commitment to the next validator set, then if two conflicting blocks are finalized that implies a conflict within a single validator set, which can then be used to slash offending validators.

However, if we try to apply this to the "interwoven consensus" model where proposing new blocks and finalizing existing blocks happen in parallel, as is the case with our latest versions of Casper, then there is a challenge. Because there is not a 100% probability that consensus will be achieved in any single epoch, we cannot simply allow a block in epoch N to determine the validator set for epoch N + 1; if epoch N ends up not finalizing, and there are two competing candidates for epoch N, then you run the risk of later finalizing two conflicting chains with disjoint validator sets.

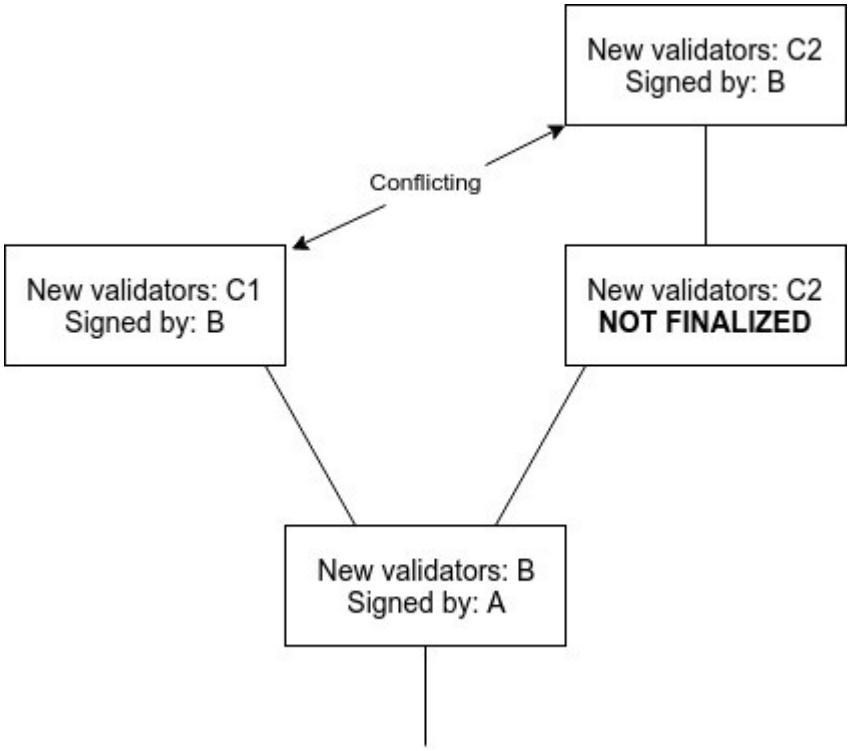


Well how about we solve this problem by having epoch N * 50 determine the validator set for epoch (N + 1) * 50, and hoping that you always finalize at least one epoch every fifty. However, this also leads to a problem:

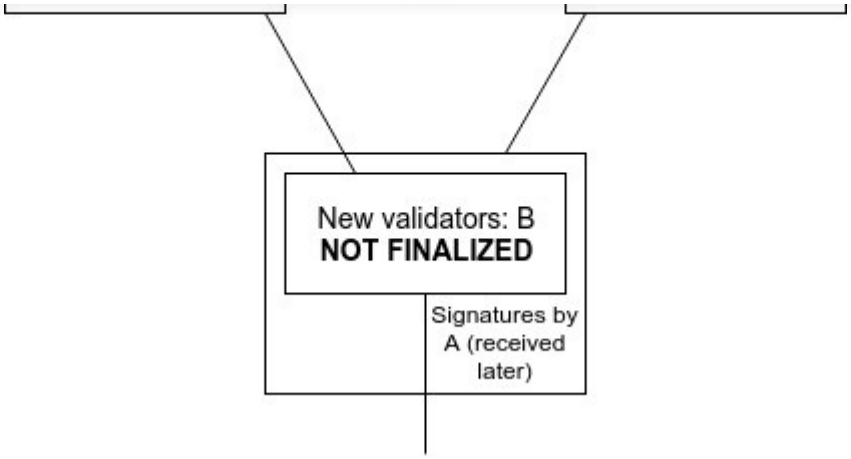


given block was finalized: if a block was finalized, then the evidence can be submitted into the chain in time for the next block or it might not, but if a block was not finalized, then the evidence can definitely not be submitted. We can think of this as a “yes or maybe” oracle on finality.

Let’s take advantage of this. If the oracle says maybe, we don’t change the validator set; if it says yes, we do.

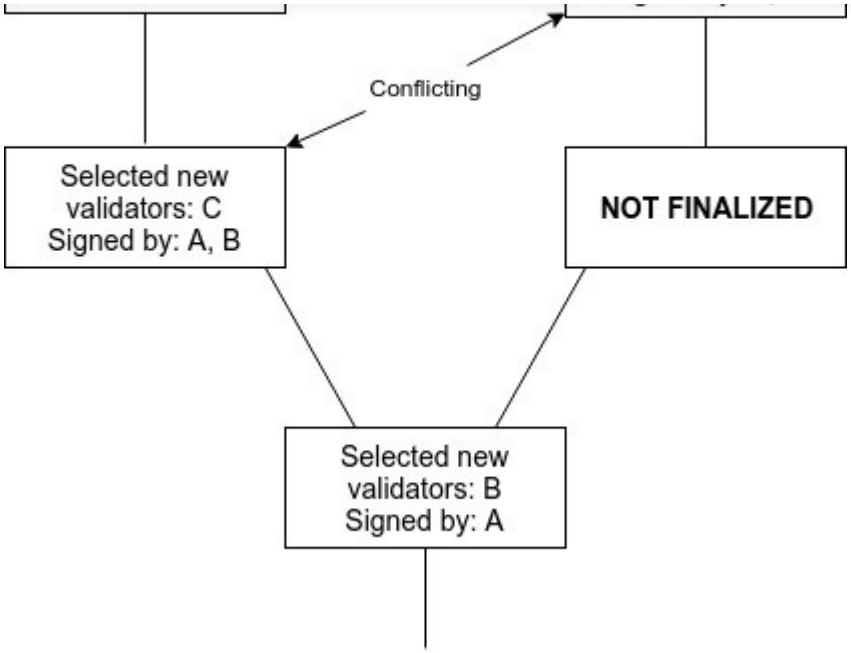


Unfortunately, things aren’t quite so simple.

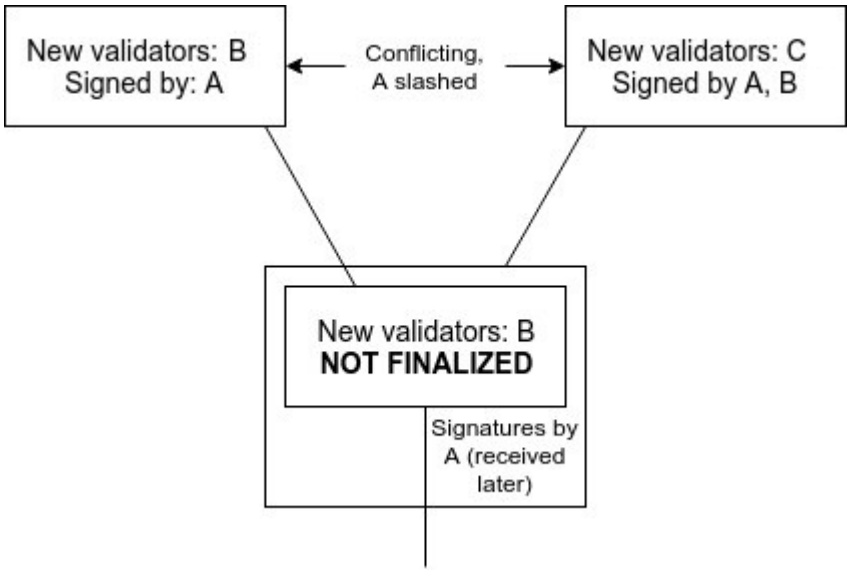


The problem here is that we don't immediately have consensus on whether or not there was consensus, and so one can make a child of a parent with two different validator sets.

And so here is where we get to our solution. We will keep the same scheme as above, except we will add the proviso that every block must be finalized by both the previous validator set and the new validator set. That is, in the slashing condition description, in any situation where we previously would require a certain type of message from 2/3 of the active validator set, **we now require that message from the union of some 2/3 of the old validator set and some 2/3 of the new validator set.**



Now, let's look at the situation when there is a disagreement over whether or not a given block was finalized.

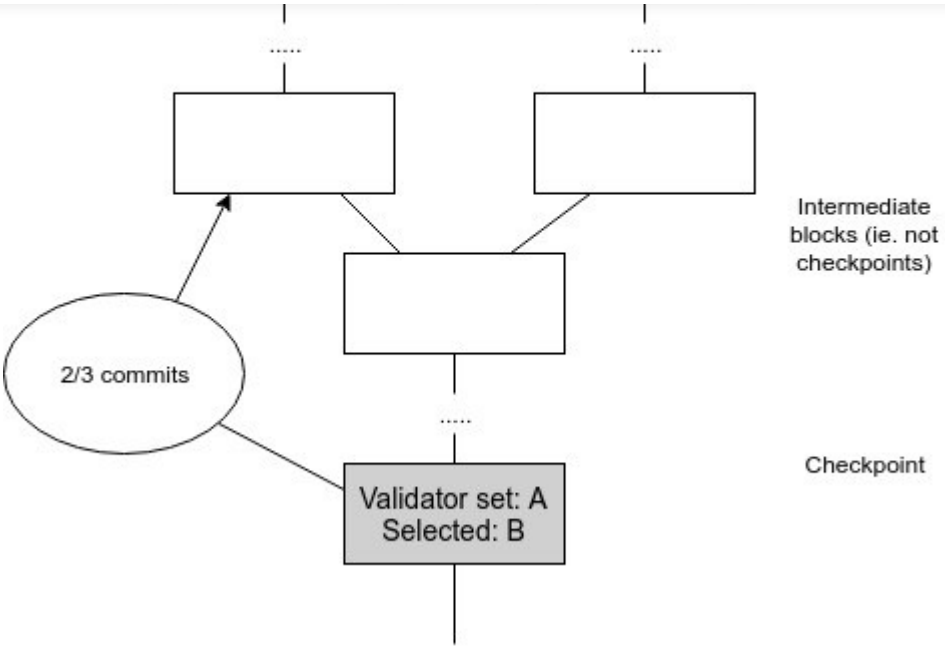


The proof that this works is fairly simple. Given any parent block, a child of that block can only have two possible validator sets: the validator set of the parent block by itself, and the validator set of the parent block joined to the validator set committed to in that block. If there are two children, and both get finalized, then 1/3 of the validator set in

Note that there are two ways to implement this scheme. One is to use the union rule explicitly, counting signatures from both the parent and the child validator set. The other is to limit the number of validator set changes (additions plus removals) per block to some `VAL_ROTATE_LIMIT` (say, `VAL_ROTATE_LIMIT = 5%`), and set the thresholds to `(2 + VAL_ROTATE_LIMIT) / 3` instead of a simple two thirds, so that hitting this higher threshold on the old set is enough to get you to a sufficiently high threshold on the new set as well (altogether, you get a fault tolerance level of `(1 - VAL_ROTATE_LIMIT) / 3`)[3]. There is no clear argument for the superiority of one approach over the other; it depends on your personal preferences.

How do we combine this with all of our other consensus machinery, including the consensus-with-a-proposal-beacon scheme described in this earlier post, and the fork choice rule? The scheme above works well in any model where there is a chain of checkpoints, regardless of whether the checkpoints follow each other directly or there are blocks in between. All that matters is that, given a checkpoint, if that checkpoint gets finalized then one can make a child checkpoint that reflects the finalization or one that doesn't.



Left: 2/3 commits on the first checkpoint got into the blockchain in time for the next checkpoint, so the next checkpoint on the left includes both A and B as validators. Right: the commits did not get in, so the next checkpoint includes only A as validators. If both checkpoints at the top finalize, 1/3 of A gets slashed.

For the fork choice rule, instead of counting commits with respect to a single validator set, we can take the minimum of the number of commits with respect to the older validator set and the number of commits with respect to the newer validator set. And that's all we need.

1. *There is a simple proof that this is equivalent to Vlad's metric of value-at-loss: the longest chain is by definition the chain containing the largest number of blocks that are dunkles in all chains other than that one.*
2. *Sure, you can get around this particular attack by requiring withdrawals to include recent block hashes. But there are plenty of other ways by which an attacker might get their hands on a chain with a small amount of stake.*
3. *The real optimum is* `(2 + 2 * VAL_ROTATE_LIMIT) / (3 + 2 * VAL_ROTATE_LIMIT)` *. To derive this, see that if your threshold is* `t` *, and the new validator set removes* `c`

`VAL_ROTATE_LIMIT` *that is at least* `t - t*d / (1+d)` *, or* `t/(1+d)` *. Safety tolerance is thus* `t + t/(1+d) - 1` *, and liveness tolerance is* `1-t` *; the maximum is when the two are equal, so* `t + t/(1+d) - 1 - 1 + t = 0` *, or* `(t*(1+d) + t + t*(1+d)) = 2*(1+d)` *, so* `t = (2 + 2*d) / (3 + 2*d)` *, with a tolerance of* `1-t` *. At* `VAL_ROTATE_LIMIT = 0.05` *, this gives us* `t ~= 0.6774` *. The comparable formula for higher values of* `VAL_ROTATE_LIMIT` *, where removing validators is the easier way to get a fault, is* `t = (2 - VAL_ROTATE_LIMIT) / (3 - 2 * VAL_ROTATE_LIMIT)` *.*

Thanks to Karl Floersch