Proof of Stake Made Simple with Casper

Olivier Moindrot
ICME, Stanford University
olivierm@stanford.edu

Charles Bournhonesque
ICME, Stanford University
cbournho@stanford.edu

Abstract

We study the recent paper Buterin and Griffith [2017] introducing Casper, a proof of stake consensus algorithm for blockchains. Proof of stake has several advantages compared to proof of work, and represents what blockchains will look like in the future. A set of validators cast public votes to decide on which blocks to finalize. They follow some rules which guarantee safety and liveness, and Byzantine Fault Tolerance of up to $1/3$ of validators.
Our goal is to explain as gently as possible the different aspects of Casper and Proof of Stake. To this end, we implement a simple version of Casper in `python`, available on github [1]. Using this simulation, we show how the consensus algorithm behaves when varying parameters like latency or when partitioning the network.

1 Introduction

Most public blockchains like Bitcoin (Nakamoto [2008]) and Ethereum (Wood [2014]) rely on proof of work to reach consensus. Participants, called miners, compete to solve cryptographic puzzles to add new blocks and receive a reward.
All this computation requires a lot of energy to run, and Bitcoin as of December 2017 consumes as much electricity as Denmark [2]. Because of high initial capital costs and economies of scale, miners get bigger and the system gets more centralized. Finally, the only way proof of work prevents attackers from breaking consensus is by spending a lot of computational effort on the main blockchain: to defend against an attacker the network needs to spend as much as the attacker.

Proof of stake uses a set of validators to reach consensus on the main chain. These validators deposit an amount of the blockchain's cryptocurrency and cast votes weighted by their stake. No unnecessary electricity is consumed, and the system is fully decentralized as there is no economy of scale. The biggest advantage of Casper-like proof of stake is that attackers can be identified and their deposit can be destroyed immediately. In proof of work, you can't destroy an attacker's hardware after an attack.

Our goal is to explain in details what is proof of stake and what is Casper. We first describe the protocol, the role of a validator, and how to finalize a block. We then prove that Casper guarantees accountable safety and plausible liveness. Our main contribution is to release a simple codebase to experiment with the consensus algorithm. We vary multiple parameters and present our results in section 4.

2 Related Work

The idea of proof of stake was first formerly introduced in King and Nadal [2012] for a cryptocurrency called Peercoin. The coin features a mix of proof of work and proof of stake to reach consensus.

[1] https://github.com/omoindrot/simple_casper
[2] https://digiconomist.net/bitcoin-energy-consumption

Peercoin represents a first category called *chain-based proof of stake* that imitate the proof of work mechanism by having randomly chosen validators propose new blocks.

Casper (Buterin and Griffith [2017]) and Algorand (Micali [2016]) come from a second category called *Byzantine fault tolerant* (BFT) based proof of stake. Taking inspiration from Practical Byzantine Fault Tolerance (Castro et al. [1999]), Casper requires validators to vote and cryptographically sign their vote message before broadcasting it to all other validators.

Proof of stake is intended as a long-term replacement for the proof of work system that is currently used in Ethereum. Casper represents an intermediary step to keep using proof of work but also add proof of stake as an additional layer of finality.

At first, Casper was following traditional consensus algorithms by using a prepare and commit message. Validators would first try to prepare a block, and then commit it to finalize it. A first version of the paper is available on github [3]. The most recent version of Casper (Buterin and Griffith [2017]) switched to using a single type of message, vote, that combines the roles of prepare and commit. This is the protocol we will focus on here.

3 The Casper Protocol

The goal of any blockchain consensus is to *finalize* blocks of transactions. Once a transaction is finalized, the participants can be sure that the transaction will never be reversed. Finalized blocks all belong to the *main chain*, and form a linear structure: there can't be any fork, which means two conflicting blocks (in different forks) can never be both finalized.

In proof of work, a block is finalized when it has enough chained descendants (6 for Bitcoin for instance).

In proof of stake like Casper, validators vote to decide which blocks get finalized and belong to the blockchain. Safety guarantees prove that if $2/3$ of validators follow the protocol, there can never be two finalized conflicting blocks.

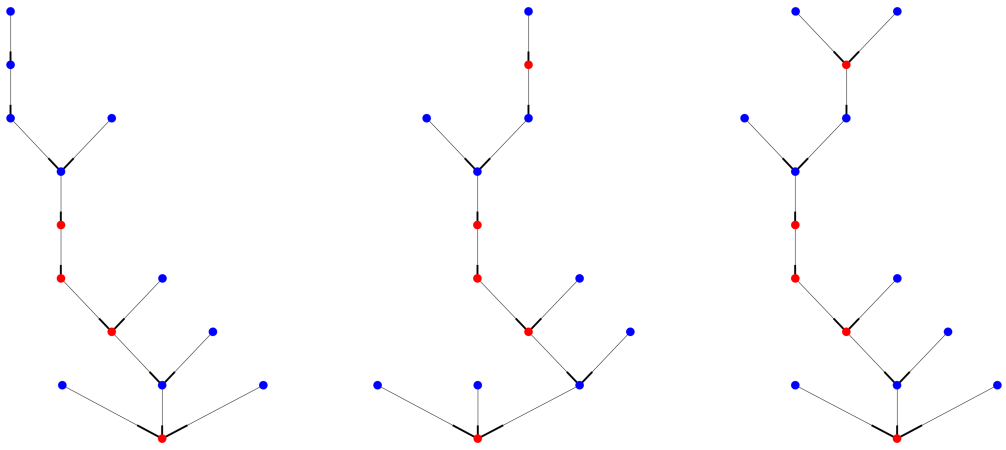

Figure 1: Visualization of 3 validators. The bottom node is the genesis block. Red nodes represent finalized checkpoints. Average latency is equal to block proposal time.

3.1 Validator and Votes

A validator can be anyone holding a minimum amount of the blockchain's cryptocurrency. To become a validator, a user can deposit an amount of the currency on the blockchain and agrees to lose its entire deposit if it breaks any rule. This deposit represents the *stake* of the validator. A validator's vote is proportional to its stake.

[3] https://github.com/ethereum/research/blob/master/papers/other_casper/casper_basic_structure.pdf

Table 1: Attributes of a vote message.

Notation	Description
s	hash of a justified checkpoint (the source)
t	hash of the target checkpoint we want to justify
$h(s)$	height of the source checkpoint
$h(t)$	height of the target checkpoint
S	Signature of the whole message with the sender's private key

To be more general, we consider that validators don't vote on every block in the blockchain, but instead vote on *checkpoints*, which are simply blocks of height a multiple of the *epoch*. For instance in Casper for Ethereum, the epoch is set to 100, so validators try to finalize checkpoints every 100 blocks.

In exchange for their deposit, validators therefore get the right to vote on checkpoints, and can earn a small reward when they collectively finalize checkpoints. This creates an incentive for the whole network of validators to cooperate towards finalizing the highest possible number of checkpoints, contrary to proof of work which is a zero-sum game for miners.

A vote is a message created and signed by a validator and broadcasted to all other validators. The vote message attributes are described in table 1.

The hash of a checkpoint is a unique identifier of the corresponding checkpoint. The height $h(s)$ and $h(t)$ are used to determine if the vote follows the rules of the protocols. These rules will be detailed in the next section 3.2.

When more than $2/3$ of validators casted a vote with c' as source and c as target, we call this a supermajority link $c' \rightarrow c$.
Justified checkpoint A checkpoint c is *justified* if it is the genesis block or if there is a supermajority link $c' \rightarrow c$ where c' is justified.
Finalized checkpoint A checkpoint c is *finalized* if it justified and there is supermajority link $c \rightarrow c'$ where c' is a direct child of c.

Having justified checkpoints is not enough because two conflicting checkpoints can be justified. However, as we will see in section 3.3, there is no way to finalize conflicting checkpoints if $2/3$ of validators follow the rules. Essentially, a checkpoint c becomes finalized when a second round of confirmation is added by justifying a direct child c' of c with a supermajority link $c \rightarrow c'$. This draws inspiration from traditional consensus algorithms with two rounds like *two-phase commit* (Gray [1978]).

3.2 Slashing rules

To ensure that safety is achieved, two rules have to be followed by the validators. A validator cannot publish two distinct votes $(s_1, t_1, h(s_1), h(t_1))$ and $(s_2, t_2, h(s_2), h(t_2))$ such that:
Rule I. $h(t_1) = h(t_2)$: A validator must not publish two distinct votes for the same target height.
Rule II. $h(s_1) < h(s_2) < h(t_2) < h(t_1)$: A validator must not vote within the span of its other votes.

If a validator breaks either of these rules, then the validator's deposit gets destroyed or *slashed*. We may now see how these two rules ensure safety and liveness.

3.3 Accountable Safety and Plausible Liveness

Accountable safety Casper provides safety in the sense that two conflicting checkpoints cannot be finalized unless more than $1/3$ of the validators (weighted by stake) break a rule. *Accountable safety* is a better property that states that we can identify and punish this $1/3$ of validators. Casper allows to do that by simply looking at the votes cast by validators. Let's prove the accountable safety.

First, it is immediate to see that following the first rule there exists at most one justified checkpoint with height n. Then, if we have two conflicting finalized checkpoints a_n and b_n, we can suppose that $h(a_n) < h(a_{n+1}) < h(b_n)$. b_n is justified so there is a supermajority link from a checkpoint lower

than $h(a_n)$ to b_n. Therefore at least $1/3$ of validators violated the second rule since we also know that there is a supermajority link from a_n to a_{n+1}.

Plausible liveness The rules also provide plausible liveness: liveness provided that there exists children to extend the finalized chain. It is enough to simply take as source the justified checkpoint of the greatest height to ensure that the slashing rules are not violated.

Following the proof of liveness, the fork choice rule, i.e. the default behaviour of all validators, will be to follow the fork with the justified checkpoint of the greatest height. This fork can always be extended and contains the last finalized checkpoint, so it is the "main" fork.

3.4 Dynamic Number of Validators

As opposed to most Byzantine fault tolerant algorithms like Castro et al. [1999], the number of validators can easily change in Casper.

We define the *dynasty* of block b as the number of finalized checkpoints from the root to block b. We can define $DS(\nu)$ (the *start dynasty*) as the dynasty of the block when the validator ν joins the validator set. We can define similarly $DE(\nu)$ to be the dynasty of the block when the validator ν leaves the validator sets. To account for the varying set of validators it is necessary to modify the rules presented before. For a block of dynasty d, the *forward validator set* is the set of validators ν such that $DS(\nu) \leq d < DE(\nu)$. Then there is a supermajority link from source s to target t where t is of dynasty d if $2/3$ of the forward validator set of dynasty d and $2/3$ of the forward validator set of dynasty $d-1$ have published a vote for $s \to t$.

4 Results

4.1 Code Architecture

We have divided our code into several classes. The main class, which contains most of the logic of the code, is `Validator`. It defines the slashing conditions and the fork rule to follow. The whole simulation is run in a single thread, where we discretize time and simulate latency or disconnected nodes.

The file "`parameters.py`" contains parameters for the simulation:

- `NUM_VALIDATORS` is the number of validators that will vote for each checkpoint.
- `VALIDATOR_IDS` is the total available pool of validators. After a block gets finalized, we change the current set of validators by picking randomly `NUM_VALIDATORS` among the total number of validators.
- `INITIAL_VALIDATORS` is the initial set of validators.
- `BLOCK_PROPOSAL_TIME` is the number of "ticks" of time between each new proposed block.
- `EPOCH_SIZE` is the height difference between two checkpoints.
- `AVG_LATENCY` is the mean latency that we will give to the simulation

The class `Message` contains the vote messages sent between the nodes. We give each message a random integer between 1 and 10^30 as a hash.

The class `Network` simulates the transmission of messages in the network with function `broadcast`. All messages will integrate a latency component which is generated randomly for each message. In our case the latency is generated from a Poisson distribution of mean `AVG_LATENCY`.

At each tick of time, the network will check which messages arrived. Every `BLOCK_PROPOSAL_TIME` tick, one validator (chosen in round-robin fashion) will propose a new block to the network, and add it to the head of its blockchain.

The class `Block` contains attributes for each block (checkpoint or not). Notably, for each checkpoint we store its "tail", i.e. its last non-checkpoint block descendent; and for each block we store the "tail membership", i.e. its closest ancestor checkpoint. Our set of validators is varying so, if a block is of dynasty d, we keep the forward validator sets for dynasties d and $d-1$.

The class `Validator` contains the behavior of the validators, and is the main component of the simulation. After receiving a new block, the validator updates the list of blocks and their information (tails, tail memberships). In the function *check_head*, it possibly changes its "head", which is the block of the chain where new blocks will be added. Then if the received block was the highest checkpoint the validator has seen so far, the validator votes for this received checkpoint using the highest justified checkpoint as the source. This is in accordance with the fork choice rule described before.

When processing the votes from other validators, a validator checks that the slashing conditions are both respected and keeps a count of the votes. If more than $2/3$ of the current set of validators voted, then the new block gets added to a set of justified checkpoints.

4.2 Visualization

We added a visualization tool using *matplotlib* and *pygraphviz* to better understand how the blockchain evolves at each validator, and how checkpoints become finalized. At every epoch, for each validators we plot the checkpoints that form the chain. The blue nodes represent checkpoints and the red nodes represent finalized checkpoints. The goal was to be able to see concretely how the chain was growing and have a sense of the number of forks and stale blocks created. By having a plot for each validator, we can also observe the effects of latency: the graphs are sometimes different across validators, or different validators don't have yet finalized some new checkpoint (although all validators will eventually finalize the same checkpoints).

An example of such a plot can be found in figure 1.

4.3 Influence of Latency

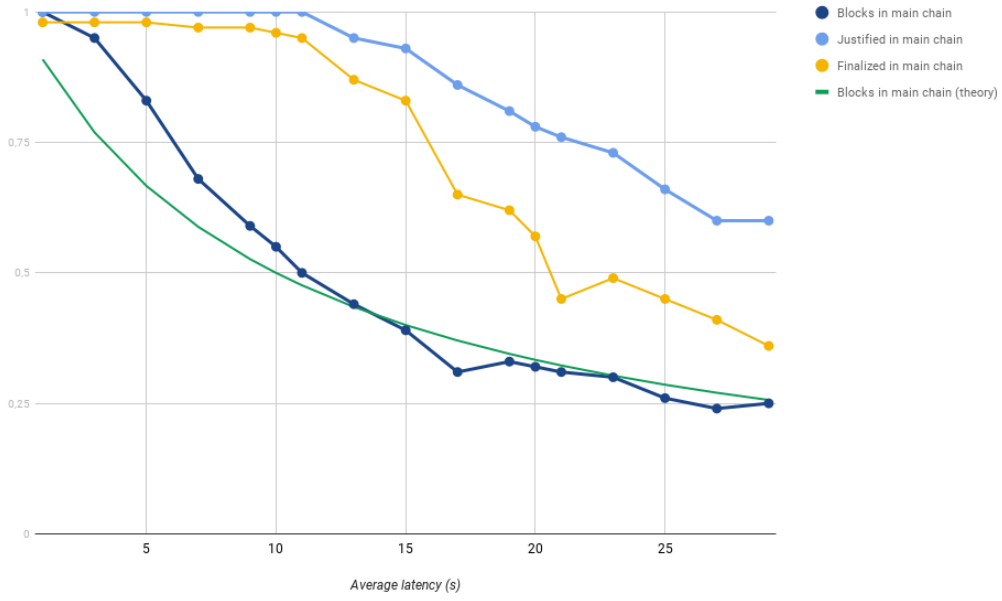

Figure 2: Influence of latency on consensus. Each new block added every 10 seconds

We plotted in light blue and orange the fraction of justified and finalized blocks in the main chain while changing the latency. We can observe that if we increase the latency, the fraction of justified and finalized blocks in the main chain decreases. Even with a high latency (average latency more than double the time to propose a block), the fraction of finalized checkpoints is around 50%. This is very robust since each finalized checkpoint finalizes the whole chain of its ancestors, so even with high latency we can reach consensus.

The dark blue curve is the fraction of blocks in the main chain (some blocks are in other forks than the main chain: they are called *stale* blocks). The green curve is a theoretical estimate of the fraction of blocks that should be in the main chain. If B is our `BLOCK_PROPOSAL_TIME` and l is the latency, then the proportion of blocks in the main chain should be around $\frac{l}{l+B}$ because for one validator that receives new block in time, there are $\frac{l}{B}$ that receive the block late, creating new forks. Our results are very close to this theoretical estimate.

4.4 Influence of Disconnected Validators

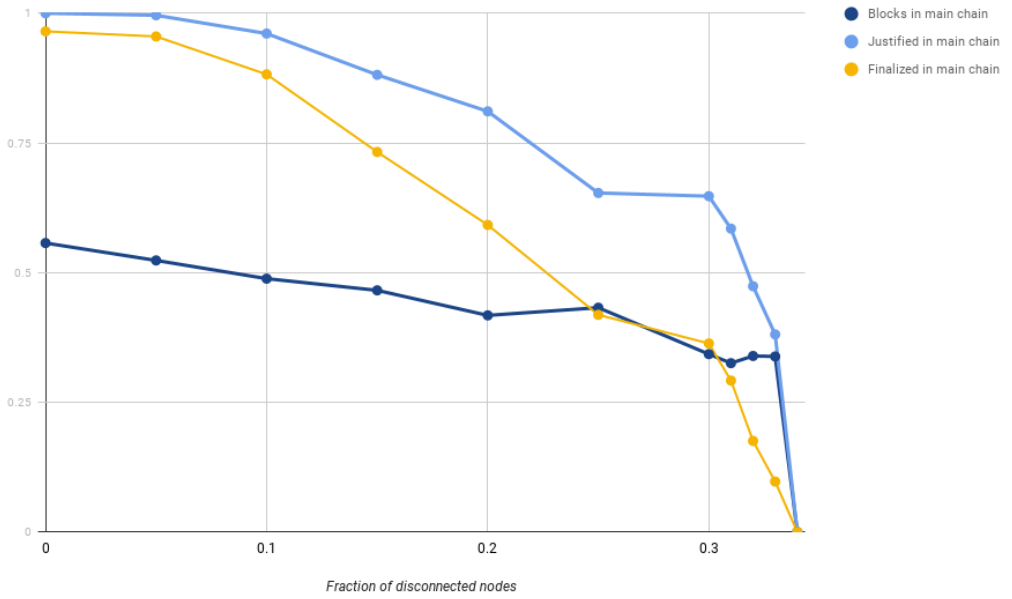

Figure 3: Influence of disconnected validators on consensus. Average latency and time between blocks is 10 seconds

In figure 3 we can observe the same metrics as in figure 2 while varying the fraction of disconnected validators (simulated by adding an infinite latency for messages from and to them). We observe that for a third of disconnected validators there cannot be any more justified nodes because consensus requires $2/3$ of validators.

However, the algorithm is very robust to disconnected nodes or network partitions: with even 30% of disconnected validators, we still have around a third of nodes that are finalized which mean we can reach consensus at a lower speed ($1/3$ of the best speed possible).

5 Conclusion

We have detailed and hopefully made it easier to understand how Casper manages to reach consensus using proof of stake. Our implementation of Casper allows anyone with basic knowledge of `python` to understand the details of the protocol and to visualize what happens inside the network. With plots of the blockchain as seen by each validator, we can better understand how latency affects the propagation of blocks and the number of justified or finalized checkpoints.

We have experimented with various parameters for Casper and found that consensus can be reached even in adversed conditions. The algorithm is surprisingly robust to disconnected nodes and will reliably reach consensus even with high latency.

References

V. Buterin and V. Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

S. King and S. Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

S. Micali. Algorand: the efficient and democratic ledger. *arXiv preprint arXiv:1607.01341*, 2016.

S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.