Proof Of Problem (PoP)

Aditya Gupta (Zeeve)
National Institute Of Technology, Patna

This suggests an 'approach' to a consensus algorithm, a beginner level at that, in a gist it's add some bytes then verify it back, what goes on in this document is what's the idea and thought process behind this.

Also, note that this is written with quite new experience in blockchains, and the idea originated from Proof Of Work and my college assignment, plus it's exclusive for Hyperledger sawtooth blockchain, should be implementable on others, but my experience is quite for that now. Further this document assumes knowledge of Sawtooth.

1. Introduction

As given earlier, in a gist it's add some bytes then verify it back.

So, first of all, the addition of bytes HAS TO HAPPEN on the client side, ie. the transaction processor (not the client 'app'). So, let's consider the Proof of Work with the number of zeroes in hash verification. So, we add a 'nonce' value (just a number) To prove that work has been done in the block, we need ensures that the hash of the whole block data, has a certain number of zeroes. And since we shouldn't alter the transactions themselves, we have this nonce number, with no real significance other than a variable value that can be set again and again to get the hash.

Now, I thought of this **in a Problem-Solution way**, ie. there is a problem, which if solved in a particular block we will consider to add it (not talking of multiple valid blocks simulatiosly here), simple but expandable. It also aims to provide all nodes a fair share of blocks added per 'publishing' node.

So, the problem in Proof Of Work can be said is the need to prove that work was done, the solution ? The nonce, and then in the verification phase we verify if the problem was solved by checking number of zeroes in the hash.

2. The Problem

Based on this idea, there can be different problems. For eg. say the problem can be a certain mathematical one, I mean less resource consuming than proof of work. Say, the problem is that some given bytes at end, when xorred/any other operation with a number that only the validator and consensus engine know, is in a particular range.

3. A sample idea

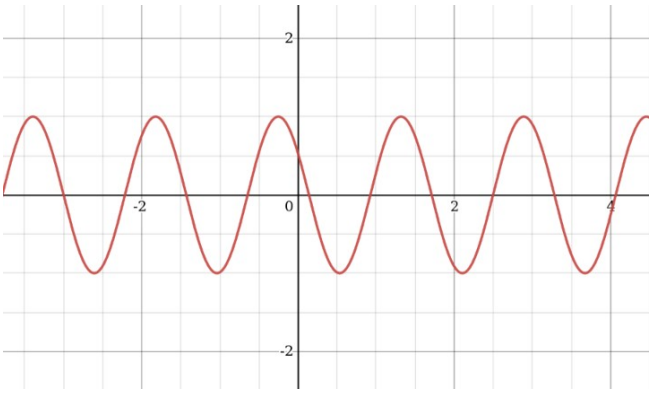
So the idea is that for each block, before it is even completed, it has to ask the validator a special 'key', then compile the block, send it to the validator, and (this is a highly variable part, i take a simple example

that it was compiled within say 1 minutes before being sent for addition).

What should be the key ? According to me, the easiest is to chose from something common to clients and the ledger itself, for eg. time is common, the chain itself.
Say, the key is just a value of a function, say

$$y = \sin(height * x + \sqrt{(time_{diff})})$$

x will be the 'nonce' provided by the transaction processor, and height is current chain height. The key is 'y'.



> Note: Recall that this is just a sample problem, and a simple key function

Then, when the consensus engine receives the block id. Apart from other checks, it computes the value of y again, since time has changed a little the value will change, but... if you focus on the function, time only 'shifts' the axes by small margins in a small time so the change in value won't be huge, if it was compiled say in this minute (to enforce such limit the implementer must introduce appropriate constants). When verified that the difference is less than the amplitude of this wave, plus they belong to same crest/trough.
After this the valid blocks are added to a list, and then we enforce the 'fair chosing', since we know the public keys of nodes that submitted these, we give blocks by different nodes more priority, so no one node can just take over.

Additionally, 'before adding we let the validator know', I can't think of something useful in this, but somehow this thing feels exploitable.

4. Code

// TODO

5. Conclusion

The idea was to think of it in a Problem-Solution way, it's not new, but I propose attempts based on this simplicity in design.