The Decentralization Trilemma Does Not Exist

By Vlad Usatii
Gemcoin Founder | Lead @ Gemcoin Blockchain Research

Satoshi didn't solve everything. When the whitepaper was published, we were in a desperate situation. The financial system that we'd grown accustomed to was no longer the viable option. We had bailouts and crises in major banks, left and right. This gave Satoshi a critical deadline: release the revolutionary blockchain technology before the banking system goes back to stable. Solving the double-spending problem while maintaining anonymity was about as far as he got. And so he published his whitepaper.

What did he neglect to solve, you may ask? Scaling. Or if you want to be specific: he had no mechanism by which a genesis block could split into multiple different databases (or shards) to save the space that each blockchain node had to allocate for the initial block download, commonly seen in Bitcoin today.

Note: by the time I got to this stage in the article, I looked up anything and everything involving splitting databases of data in blockchains, and found Buterin's implementation (or idea) of sharding.

Buterin's sharding idea got me thinking: is there a less complex way of dealing with large amounts of data in decentralized databases? More importantly (assuming I can explain what a simplified sharding model would look like): what are the *new* optimizations and tradeoffs with the implementation of a sharding algorithm?

In other words, is there a quadratic representation of efficiency (time to send or request a piece of blockchain data) on the x-axis and the number of shards on the y-axis? If there was, we'd know where to halt the number of sharded databases before the process starts taking more time than it did *without* its existence. This process is comparable to finding the extrema with calculus of something like $f(x) = x^n + ... + x^2 + x^1 + n$. You find where the derivative is equal to 0:

$$\frac{d}{dx}f(x) = nx^{x-1} + ... + 2x^1 + C \approx 0$$

So, with the idea of minimalism and optimization in mind, here is my solution to the scalability hypotenuse on the triangle by which we know the trilemma to encapsulate. And I guess I could retitle the blog post by something different, namely my idea's weird title: **nScale**.

What is nScale?

Let's say that some hypothetical blockchain reaches a level, such that the amount of information in one blockchain database is much too high for any sane person to download and sync. The only responsible way to handle this is to start partitioning data, or warning nodes that the old data will be deleted. Maybe a full node won't even need to download everything because of zk-SNARKS? An easier way (and probably the most intuitive way) of going about this is to shard the data.

Here is the starting spot of my formulation of such a process:

Whenever a user downloads the blockchain and runs `main.py` for the first time, they will store a thousandths-decimal timestamp value and multiply the timestamp by 10^3. The timestamp will look like this:

$\text{math.ceil}(1646294355131.9302) = 1646294355132$

If the node's number is even, they will store the first half of the blocks. If the node's number is odd, the same thing will happen, but with a different (and odd) chain of blocks. The even number node's blockchain will go by block number $0, 2, 4, 6$, and so on, while the odd node's blockchain will go by $1, 3, 5, 7, 9$, and so on. So that cheating can't occur, every user will still download the same block *headers*. This protects the header and mixHash from being faked, or having two chains request data from each other and have mis-linked block headers.

Now, if an odd node requests an account balance, an even node can be contacted, and a traversal of every block can be performed. Or, if there is someone out there working on zk-SNARKS that could see it playing a role in my system, then so be it.

On top of that, if block n is requested, an even node could simply find the block *header* and ask an odd node if they have such a header's block.

Naturally, a blockchain developer would want even more scaling. Why have two chains when you could have an infinite number of chains and traversals? That is where nScale comes in.

The idea behind nScale

What if we had n odds and evens? In other words, what if, based on the timestamp, or some rounded-down version of the timestamp, you could have your very own chain of unique blocks? As mentioned above, how many unique shards could be created before the blockchain starts to decrease in its efficiency?

The applications of such a feat would yield new discoveries in asynchronous state machine design (so, e.g., having 10 different chains running their own individual, "asynchronous", low-latency state machines), increase the capacity by which we could store data in blocks, and so on and so on.

To accomplish this monstrous task of nScaling, let me formulate the problem.

Formulation of the problem

We have n unique concatenated timestamps, all of which generate a bias towards nodes with the same unique concatenated timestamp, thus you have n unique shards of the chain.

Every node who downloads a chain will partition the main chain into their uniquely claimed piece of the pie. The following equation describes the system quite concisely:

$P = \frac{b}{a}$, where P is the amount of blocks downloaded to the groups of nodes with a matching concat-timestamp, b is the total blocks in the mainnet, and a is the number of unique timestamp possibilities that any node is subject to being a part of upon syncing to the network (on their first connection).

In other words, the more shards we have, the less blocks each user stores, making the system lightweight and clean.

The dirty side of nScale

Yes, there is always a caveat, but there *isn't* a trilemma. The problem is: at what point are there too many shards, and can sharding be automated when the blockchain reaches some checkpoint in block number or some other metric?

One solution to the arbitrariness of such an implementation could be the following:

$a = r \pmod{n}$, where a is the unique shard identifier for future sharding of data, r is the timestamp, and n is the number of shards there should be.

This would allow the inversely proportional equation P above to make more sense in the context of expansion.

Conclusion

We've explored some basic ideas in sharding, but I've touched on a scenario where this may be useful at first. I explained a generalization of such an implementation and the math that makes it possible. Questions? Concerns? Submit a pull request and tell me about it at https://www.github.com/VladUsatii/gemcoin.git.