Small world sharding with statistical proof-of-stake quorums (DRAFT)

Jessica Taylor

November 24, 2021

This paper presents a scalable blockchain algorithm. It is an iterative improvement to the Inductive Consensus Tree Protocol (ICTP), but with better fault-tolerance (due to lack of a central point of failure), and it is simpler overall.

1 Theory of scalable blockchains

In conventional blockchains, such as Bitcoin and Ethereum, all full nodes will verify all transactions that happen. This causes a blow-up in bandwidth, computation, and storage required. If there are n nodes that each make t transactions, then the amount of bandwidth, computation, and storage required in total is $O(tn^2)$. Each transaction becomes increasingly expensive as n increases, leading to high transaction fees in conventional blockchains.

If we loosen the requirement that everyone verify all data, we run into a problem: how can everyone know that the blockchain's state is valid, without verifying it themselves? We now make a distinction in methods of solving this problem:

- A *positive* verification method attaches to blockchain data a "proof" that the data is valid, and that blockchain data it refers to (e.g. the previous block) has a valid proof.
- A *negative* verification method permits "proof"s that blockchain data is invalid; if such proofs are found, they are distributed to others so it is clear that the data is invalid.

The main problem with negative verification methods is that problems that inhibit the network (outages, nodes that uncooperatively refuse to share information with some other nodes, etc) could prevent proofs of invalidity from being discovered and widely shared. There is an inherent race condition with negative verification methods, where the proof of invalidity must be discovered and distributed before blockchain data is considered well-verified enough to consider transactions contained in it to have cleared.

For positive verification methods, there are a number of different ways of constructing proofs that are easier to check than to generate:

- A zk-SNARK may be included that verifies that the blockchain data is correct, and that blockchain data it refers to has a valid proof (also in the form of a zk-SNARK).
- Signatures from a supermajority of a trusted set of parties may be included, which assert that these parties have verified the blockchain data, and that referenced blockchain data has signatures by a supermajority.
- Signatures from a supermajority of a large random set of parties may be included, which assert that these parties have verified the blockchain data, and that referenced blockchain data has signatures by a supermajority.

There are a few issues with using zk-SNARKs for verification:

- Even if a zk-SNARK verifies that the data is valid, it doesn't verify that the blockchain data (e.g. the next block) is *unique*; there may, for example, be alternative versions of a blockchain block, one of which says a given transaction happened, and one of which doesn't, both of which are valid.
- The existence of a zk-SNARK verifying data as valid does not imply that that data is generally available; it may only be available to whoever created the SNARK.
- Verifying a zk-SNARKs within a zk-SNARK is computationally expensive.
- Software implementation is more difficult due to the complexity of zk-SNARKs.

The second method, a trusted set of parties, has the problem that it increases centralization, requiring everyone to trust a small set of parties. Those parties, themselves, must run powerful computers that verify a large number of transactions.

The third method is the main one considered in this paper.

1.1 Statistical quorums

Suppose there are n accounts that are staking money. Suppose we want a *pool* (set of accounts) of expected size k.

There are 2 ways of selecting such a pool:

- We could select each account independently to be part of the pool with probability k/n.
- We could create a pool of size k and select a random account for each space in the pool.

Either way, if at least a $h > 1/2$ portion of these stakers are honest, then if k is high enough, there will exist some $0 < q < hk$ such that, with high probability (getting higher as k is higher):

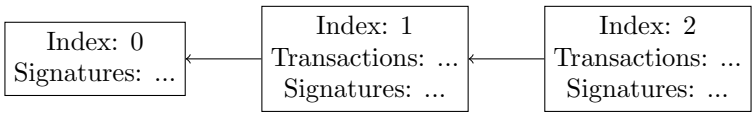

Figure 1: Non-sharded blockchain.

- There will exist at least q honest accounts in the pool.
- There will exist fewer than q dishonest accounts in the pool.

A subset of the pool of size at least q is called a *quorum*. If a valid quorum signs a block, that block is generally considered to be valid; hence, it is desirable for it to simultaneously be highly probable than there is an honest quorum (so the block may be signed), and highly improbable than there is a dishonest quorum (so no invalid blocks are signed, and there are no forks formed by two different valid blocks being signed by valid quorums).

This is due to the central limit theorem. Both the distributions over number of honest and dishonest accounts in the pool are binomial. In the case of selecting each account independently with probability k/n, both distributions are approximately Poisson.

For example, suppose that n is high enough for the Poisson approximation to be valid, and suppose that $h = 2/3$ portion are honest. Then if we form a quorum of expected size $k = 2000$, then the number of honest accounts is Poisson distributed with mean 1333.33, and the number of dishonest accounts is Poisson distributed with mean 666.67. The probability of there being fewer than $q = 1000$ honest accounts in the pool is then less than 10^{-21}, while the probability of there being at least 1000 dishonest accounts in the pool is less than 10^{-32}.

2 Algorithm overview

Here we present a basic overview of the algorithm.

2.1 Proof-of-stake blockchain with statistical quorums

There is a simple way to make a proof-of-stake blockchain with statistical quorums. Specifically, we form a random pool every r'th block, and require that the block is signed by a quorum of pool members. These pool members check the transactions in the block, and check that the block continues from a block containing a valid quorum's signatures. Assuming honest stakers never sign two different new blocks, there will be a uniquely identifiable new block.

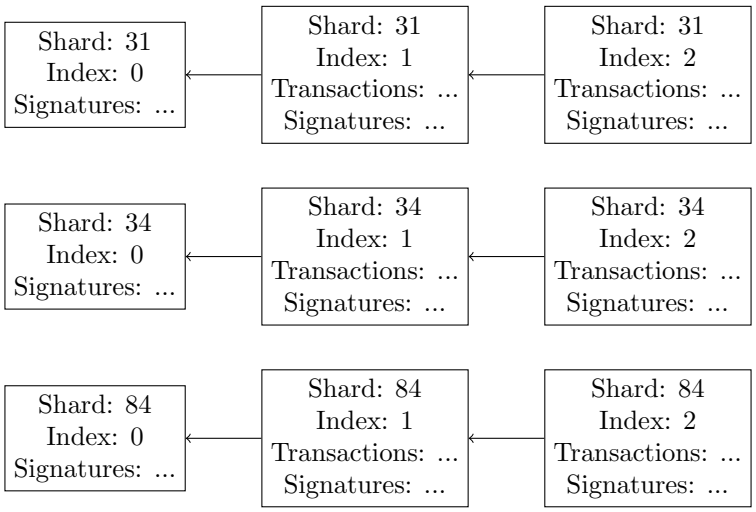

Figure 2: Non-networked sharded blockchain.

2.2 Sharding

We can run many of these blockchains in parallel; these parallel blockchains are called *shards*. Since they are proof-of-stake, there is not a potential problem with concentrating proof-of-work computation on a single blockchain to take it over. Different shards will handle transactions for different sets of accounts; generally, each account is assigned to a random shard.

Ideally, these different shards could share the set of stakers, since a larger population of stakers yields better statistical guarantees. Also, it would be desirable for the different shards to be able to send transactions to each other.

Suppose shard A is producing block number v_a, which is meant to receive money from someone in shard B; assume the resident of shard B sent the money in block b. Now, for block v_a of shard A to receive money from block v_b of shard B, there must be some checkable proof that block v_b of B has been validated before the time of block v_a of A. Luckily, there is such a proof: signatures by a quorum validating block v_b of B. If the quorum has attested that block v_b of B is, indeed, the v_b'th block of shard B, then validators of shard A can trust this, and can therefore check whether the send being receved from has happened.

This requires everyone to be able to potentially find out the members of the pool of ever shard. If pools are assigned at random at the beginning and never changed, this is easy; we will cover re-shuffling pools later.

This is the basic sharding algorithm. It is somewhat inefficient, since, for quorum threshold q, there must be a total of q^2 signature checks per receive transaction (q validators for shard A checking signatures of q validators for shard B). The rest of this paper, therefore, focuses on an algorithm with improved data efficiency.

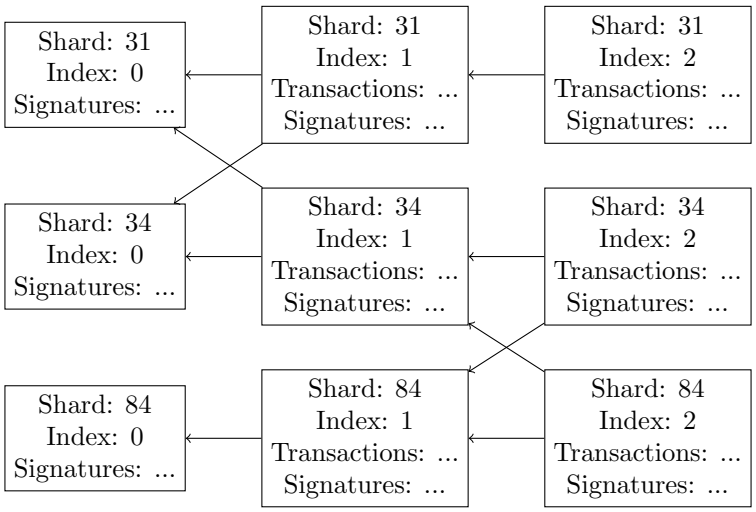

Figure 3: Small world sharding.

2.3 Small-world network

To alleviate the inefficiency, consider the following approach. Organize m different shards into a small-world network. For example, this could be done by choosing some base b, writing each shard's index (which is between 0 and $m-1$) in base b, and considering each shard's neighbors to be those whose index could be formed by modifying a single base-b digit. This way, the graph distance between any two shards is no more than $\lceil \log_b m \rceil$, and every shard has at most $b\lceil \log_b m \rceil$ neighbors.

Now, suppose that each block in each shard contains the hash codes of the most recent known blocks of every neighboring shard. The verifiers of a given block will check that the referenced other blocks are signed by an appropriate quorum.

We can now succinctly prove that some block a occurs after some block b, as follows. If there is a chain of blocks, starting at b and ending at a, each of which contains the hash code of the previous one, and each of which is signed by an appropriate quorum, then block a occurs after block b. (Note that each block necessarily contains the hash code of the previous block of the same shard, as usual in a blockchain).

We now require a receiving transaction to occur in a block that comes after the block containing the corresponding sending transaction. At this point, checking the transaction does not require downloading and verifying any *additional* signatures; the proof of the existence of a chain of blocks can omit the signatures themselves, providing only the hash code of the signature list in each block. Since the signatures of the neighboring blocks were already checked, and these signatures verify that the verifiers of these neighboring blocks already checked the signatures of blocks neighboring the neighboring blocks, and so

on recursively, no additional signature checking is necessary. If the new block contains many transactions, then the overall amount of work will be reduced.

We can consider a further efficiency improvement. Since each shard index can be written in $\lceil \log_b m \rceil$ base-b digits, we can say that a block whose index (taking the first block in a shard to have index 0, the next to have index 1, and so on) is c mod b will only include neighbors that differ at digit c. This does not significantly lengthen paths, since there will be a length $\lceil \log_b m \rceil$ path between any two shards, formed by modifying the digit at each position in sequence. (Inclusion of hash codes is to some degree unreliable e.g. due to network issues; such unreliability can increase the length of paths, but this is not a serious problem as long as the network starts working in the future)

2.4 Pseudorandomness

Pseudorandom numbers can be generated using threshold cryptography, as described in the ICTP paper. Pseudorandom numbers are generated on a per-shard basis; each shard runs its own random number generation procedure.

The primary use of random numbers is selecting stakers to participate in quorums. Quorums are occasionally reshuffled; they are re-sampled according to a snapshot of current account stakes.

Due to the absence of a total time order in a sharding system, creating a snapshot is somewhat tricky. Each block contains a timestamp estimating the time at which the block is created. For any time t and any shard, that shard either has a first block at time t or later, or has no blocks at time t or later. A snapshot for time t is taken to consist of the set of first blocks after time t for each shard. A block that is part of a t-snapshot will be indirectly referenced by all shards at some later time $t + q$, assuming no major network issues. (Note that the snapshot is a hypothetical object; no snapshot need be fully computed.)

At this point, the procedure for selecting a random staker is as follows. We select a random shard (and a recent block from it according to the canonical path) and keep it with probability proportional to that shard's total stake. For such proportional selection to be tractable, we assume an upper bound on stake per shard, which is yielded with high probability if accounts are randomly distributed between shards, and there is a maximum stake per account. If we didn't keep the shard, we re-sample another shard, repeating the procudre. Once we have selected a shard, we select a staker within that shard proportional to that staker's stake amount.

This procedure, repeated, selects a pool of an arbitrary size. The same account may be selected multiple times to participate in the same pool, in which case it has a number of votes equal to the number of times it was selected.

Note that the pool members of a given shard need to know the pool members of neighboring shards, so that they can check whether neighboring blocks are endorsed. Since each shard has a small number of neighbors, this is not especially expensive if pools are only occasionally re-shuffled.

When transaction volume is too high, it is necessary to increase the number of shards. When this is done, it is necessary to re-assign account data to the new

shards. For this to work, there has to be global consensus about when shards should be re-shuffled; a simple way to do this is to say that transaction volume is measured from some canonical shard, although this has the problem that this canonical shard becomes a single point of failure. There are likely better ways of deciding when to increase the number of shards, which may be included in future versions of this paper.

3 Algorithm specification

Now we proceed to a more thorough definition of the algorithm.

3.1 Cryptographic primitives

We assume the following cryptographic primitives:

- A hash function, such as SHA256.
- A digital signature algorithm, such as RSA or ECDSA.

3.2 Accounts

An account is a hash code of a public key. It is a fixed number of bits (e.g. 256 for SHA256).

3.3 Shards

There are s shards. An account can be mapped to a shard pseudorandomly by taking its value (as a binary number) mod s.

3.4 Actions

An action is taken by an account. Actions consist of a sequence of parts, which are executed in sequence.

A send part contains the following fields:

- The amount of money to send.
- The receiving account.
- A unique identifier.

A receive part contains the following fields:

- The account being received from.
- The index of the block containing the send.
- The hash code of the send part being received from.

While there are other action parts for handling transfers between balance and stake, they are not elaborated here.

An action contains the following fields:

- The hash code of the last block in the shard.
- A list of parts.
- A fee paid to execute this action.
- A signature by the account executing the action.

3.5 Merkle patricia trees

Each block contains a Merkle patricia tree, which tracks the state of each account the block's shard is responsible for, at the time of the block. A node in a tree contains the following fields:

- Optionally, an entry.
- A list of hash codes of child nodes and their corresponding suffixes.

The Merkle Patricia tree represents a mapping from quaternary strings (*keys*) to entries. Keys are assigned as follows:

- The key for account a's balance is the quaternary representation of a, followed by 0.
- The key for account a's stake is the quaternary representation of a, followed by 1.
- The key for a send part sent by account a with hash code s is the quaternary representation of a, followed by 2, followed by the quaternary representation of s.
- The key for a Boolean indicating whether account a has received a send part with hash code s is the quaternary representation of a, followed by 3, followed by the quaternary representation of s.

The exact way keys are assigned is not important; it is more important that they can easily assigned to uniquely reference any account data. Each shard contains only the data for accounts corresponding to that shard.

Most nodes in the tree are preserved from one block to the next block in the same shard; only the nodes that are on the path to modified keys must be changed. This is structurally identical with operations on purely functional tree data structures.

3.6 Blocks

A block contains the following fields:

- The index of the block's shard.
- The block's index (within the shard).
- The hash code of the previous block.
- A timestamp approximating when the block is created.
- The hash codes of latest blocks of appropriate neighboring shards.
- The hash code of the root Merkle Patricia node for the current account data.
- The new actions being executed in this block.
- The total amount of stake of accounts in the Merkle Patricia tree.
- The hash code of a list of signatures by a quorum.

4 Additional features

4.1 Smart contracts

Smart contracts can be defined that use the actor model. That is, different smart contracts may send messages to each other (which may contain money), which get received at some nondeterministic future time.

To implement smart contracts, we differentiate accounts into *user* accounts, which have a public key, and *contract* accounts, which are controlled by smart contract code.

Contracts are specified in WebAssembly. Each contract module must include code specifying three functions:

- **init**, which is called on contract initialization.
- **command**, which executes an operation (perhaps consuming one or more messages sent to the contract) producing a new state.
- **query**, which reads contract state to produce a result but does not modify the state.

Contract module code is run through a preprocessor that performs the following modifications:

- inserts a step-counter which computes gas costs during execution based on computational resource usage.
- canonizes nondeterministic outputs such as floating point numbers.

Contracts have access to an API with the following functions and more:

- Sending and receiving messages, which may contain money.
- Reading and writing data fields that are part of the contract's state.

4.2 Privacy

Privacy is possible by hiding account data, instead having the verified data in each shard store a Merkle root of the private account data. zk-SNARKs can be used to show that the private account data is transformed lawfully, which requires receive actions to receive from an actually-existing send. This feature is not elaborated on further here, since it is essentially the same as in ICTP.