ZooBC:
New Strategies in Blockchain

by Barton Johnston, Roberto Capodieci, and Stefano Griggio

Abstract

Since the emergence of Bitcoin, a plethora of Blockchain technologies have emerged, each approaching certain limitations of the original design in their own way. In the same spirit, as a first step in what we intend to be a sequence of gradual experimental improvements, we here propose a new Blockchain architecture and generally describe its structure. Among other points, we focus on reducing the blockchain download time to a constant, more evenly distributing block creation and network rewards among nodes which prove they are doing useful work for the network, and creating a flexible and testable technological base for others who which to extend the platform to include business logic specific to their domain. Finally, we explore the security and performance implications of each new strategy where they differ from other Blockchain technologies.

Index

Introduction

… intro goes here …

NOTE: This version is an initial DRAFT. This means that big chunks of content are missing, there may be typos, some information may change before the final first version is published, and has been published for a first, initial public review. If you are reading this in a PDF format, you can contribute with comments on the Google Document at this address:

https://docs.google.com/document/d/1RHbDHHH0JlAfU8bdgfawbnvlm-Ng_Tq-VA7P-n1p_80/edit

Furthermore, you can contribute to the discussion in our forum and Telegram group! Here are all the social media links for ZooBC:

- Website: https://zoobc.com
- **Forum**: https://zoobc.org
- Facebook page: https://fb.me/TheZooBC
- **Facebook group**: https://www.facebook.com/groups/ZooBC
- LinkedIn Page: https://linkedin.com/showcase/zoobc
- Twitter: https://twitter.com/TheZooBC
- Instagram: https://www.instagram.com/TheZooBC
- Telegram Channel: https://t.me/ZooBCnews
- **Telegram group**: https://t.me/ZooBlockchain

How this document is structured:

In **Section 1**, we first present a technical description of the system and all of its component mechanisms.

In **Section 2**, we present a security analysis offering arguments for and against our design decisions, and analyzing potential attack vectors. *(coming in a future version of this paper.)*

In **Section 3**, we provide appendices giving more complete information or calculations on certain technical details. *(coming in a future version of this paper.)*

Whenever you see this:
…
It means that more text will be written there soon.

Enjoy the read, please help with any insight you may have, and help us build a next generation Blockchain!

Section 1: Technical Description

This Section presents a reasonably complete explanation of the system in technical detail. The discussion of security considerations and attack vectors in Section 2 assumes the reader's familiarity with the following mechanisms, which enables us to reason about how they combine to form the security properties of the system as a whole.

...

Accounts

We use an account model, such that an account owns a balance and any other properties or assets. ...

Account Addresses

Most Blockchain software picks a specific digital signature algorithm and requires all users to create key pairs using that algorithm. We consider that some signature algorithms are increasingly respected in some countries by courts of law, as well as the convenience of sending funds to an account for which you are already certain the other party possesses the private key, although on another blockchain.

Therefore we design our protocol such that a user can select his own signature algorithm from a set of supported ***address types***. Each type specifies a unique address type code, as well as the format of the address, and the format of the corresponding digital signature. In this way it is possible, for example, to send funds to your friend's Bitcoin address on our blockchain, knowing that your friend will then be able to use his Bitcoin private keys to sign valid spends of those funds on our blockchain.

A full set of supported address types is detailed in Appendix 1.

Transactions

Every action executed on the Blockchain by a user is encoded in some ***transaction*** which specifies the action, any parameters, and a valid digital signature on the transaction data for the sender's account address.

A transaction may be as simple as transferring balance from one account to another, but in principle can have unlimited complexity, so long as it satisfies the properties that both its *validation* and *application* are purely deterministic operations which only read from, and write to, the portion of the node's database which is managed by the consensus algorithm.

Transaction Types

We do not support general "smart contracts" (on-chain code), despite the popularity of this approach, because we find it unsuitable for serious business logic. Code which cannot be updated in case of bugs or changing business rules (even if by a democratic process of distributing a node software update) is a liability. Further, implementing a VM supporting arbitrary logic increases both the blockchain code complexity, and the user complexity in terms of computing transaction execution costs and interface design.

Therefore we define a set of ***transaction types*** recognized by the network, where each type defines a particular behavior. While the initial set of transaction types we include are limited, our reference implementation is structured in such a way that an organization producing a customized version of this Blockchain can easily add or modify the transaction logic to their node software and corresponding wallet software.

A full set of transaction types we support, with their parameters and behaviors, are described in Appendix 2.

Transaction Propagation

When a transaction is first received from a peer or wallet software, the node will first validate that the transaction is legal. This includes validating that the transaction is terminated by a valid digital signature for it's sending account address, and that the parameters of the transaction are valid according to the transaction type's rules and the current state of the database (such as having enough balance to send funds.)

If the transaction received is valid, the node will *propagate* the transaction by transmitting it to other connected peers, as well as store the transaction in its local *mempool*. In this way, valid transactions are echoed across the peer-to-peer network until they are retained in the mempool of the majority of nodes.

Upon receiving a valid transaction, the node will also return a special object we call a receipt to the sender. Receipts are used in the Proof of Participation algorithm described below.

Transaction Application

When a node receives a new valid block (described below), it will contain an ordered list of zero or more transactions, which the node will then *apply* in sequence.

Applying a transaction means executing the rules associated with the transaction's type to modify the database from an old state to a new state. In the simplest example, this can mean deducting balance from one account and adding it to another account.

...

Transaction Fees

Each transaction must include a small ***fee*** in tokens to pay for its execution on the network. The fee for each transaction is distributed among the registered nodes in the network in the same manner as coinbase rewards. This firstly serves as a form of spam protection, ensuring it is costly to overwhelm the network's limited transaction volume. Secondly it serves to incentivizes block creators to include as many transactions as possible, maximizing their collective reward for producing blocks.

The ***minimum fee*** for a transaction is computed differently for each transaction type, then multiplied by a fee scaling constant which can be periodically adjusted. A transaction may pay more than its required minimum fee in order to be accepted in a block faster when the network load is high, but it is illegal to include less fee than the computed minimum. For a transaction A, this minimum fee can be computed:

$$\texttt{min\_fee}(\texttt{Tx}_A) = \texttt{type\_fee}(\texttt{TxType}_A, \texttt{Tx}_A) * \texttt{fee\_scale}$$

Fee Scaling

One difficulty with transaction fees on any Blockchain network arises from extreme volatility in token value. Ideally the cost of the fees will remain approximately constant with respect to the stable currencies which users exchange for the token. For example, if the average minimum transaction fee is 1 token, this will fail to reduce network spam or incentivize block creators when the token value is US$0.0001, but it will make all transactions prohibitively expensive if the token value reaches US$100.

There are many strategies to approach this dilemma. On one extreme, the fee could be set by a trusted third party which signs such new information against a public key agreed upon by the entire blockchain; but such centralization is anathema to the ethos of decentralized systems, as it can be manipulated by a single actor who is beyond accountability. On the other extreme,

nodes could attempt to draw from public sources such as exchanges independently in order to validate transaction fees; but because such data is not tracked by the Blockchain's own consensus, this could lead to forks when such external information is not consistent when queried by different nodes at different times.

Therefore we conclude that safely reaching consensus on data from outside of the Blockchain, such as the token's value against other currencies, cannot be accomplished automatically. We implement a system by which operators of registered nodes on the network may take a regular vote on the appropriate multiplier, which we call the ***fee scale***, for minimum transaction fees.

It is potentially dangerous to put such a critical network parameter in the hands of node operators. A more complete discussion of the risk is presented in Section 2. We prefer this risk over the inherent risk of imposing a static minimum fee for the reasons described above.

We can consider two competing incentives of node operators which should constrain each other: in the small scale, a node operator wishes to maximize the transaction fees he collects in each block by pushing the network fee scale higher; but on the large scale, network fees that are high will reduce user's willingness to pay for any transactions on the blockchain, and this lack of usability may be reflected in the token price, which determines the real value a node operator stands to gain, incentivizing him to push the network fee scale lower.

This vote-based adjustment of the fee scale is accomplished with three mechanisms: Node operators committing to their votes, node operators revealing the votes they have previously committed, and the averaging calculation used to compute the new network fee scale.

Committing to Fee Votes

...

Revealing Fee Votes

...

Adjusting the Network Fee Scale

...

Blocks

A Blockchain system fundamentally solves the problem of uniquely ordering all transactions which have been broadcast to the network. …

Nodes are chosen in a pseudo-random lottery to append sets of transactions to the network's total transaction history. We call such a set of transactions, along with some meta-information, a ***block***.

The Block Chain

As indicated by the name, blocks are cryptographically ***chained*** together by including in the metadata of each block the hash of the previous block. Consequently, to modify the contents of any previous block would require all subsequent blocks to be re-created.

Cumulative Difficulty

...

The Block Seed

In this technology we use many strategies that depend on pseudo-random numbers which can also be agreed upon deterministically by the network. Because a strong hash function produces high entropy from arbitrary input data, the most obvious source of entropy to draw upon is the hash of a block. However, as the block creator has great freedom over how he structures his block, he may deliberately produce a block hash which, when used as a seed for random numbers, will disproportionately favor himself.

Therefore we include a special property in each block which we may call the ***block seed***. This property cannot be influenced by the block creator, and cannot be predicted in advance by anyone who does not have the private keys of the other block creators. We compute the block seed:

$$block\_seed_{H} = creator\_signature_{H}(hash(block\_seed_{H-1}))$$

Block Creator Selection

Because we know the complete set of potential block creators (described below in the section on Node Registration), we can pseudo-randomly select which node is responsible for creating

the next block. When a node misses his turn to create a block, his participation score is punished, and the network awaits a block from the next node in the random order.

The node's probability of finding a block is weighted *inversely* with his participation score. This has the consequence that nodes with lower scores are called on more frequently to create blocks, which via the Proof of Participation algorithm (described below) accelerates the rate at which they may gain or lose participation score. We think of this like a node taking on extra responsibilities to prove itself, relieving the nodes which have already proven themselves of the hard work.

Implied Data

...

Multisig

In managing high-value digital assets, it is risky to have one person holding the keys, or risk that a single key being lost or compromised could expose the asset. Therefore we implement native multi-signature features, allowing users to create accounts which require X-of-Y signatures.

Multisig Addresses

One of the address types we support is a ***multisig address***, which is the hash of a set of details regarding who is allowed to sign on behalf of the address. We refer to this set of raw details as a ***multisig info***.

This design leaves open the possibility that until the time of signing, the set of addresses which control a multisig address need not be revealed.

Multisig Info Object

Field	Description
MinSigs	The minimum number of signatures out of the provided address list which must be present to execute a transaction from this multisig address.
Nonce	A free number field allowing unique multisig addresses to be created between the same set of addresses.

Addresses	A list of account addresses which may legally sign for this multisig address. These addresses can in turn be multisig addresses, allowing for hierarchical multisig.

The Multisig Transaction Type

We implement all aspects of multisig behavior through a single transaction type. The behavior of this transaction type is somewhat complex, because it implements both on-chain and off-chain multisig behavior, with the possibility to preserve the anonymity of the signers and conceal the transaction being signed until the moment that all needed pieces of information have been exposed to the blockchain.

The transaction body of a Multisig Transaction has 3 optional components: a Multisig Info object (described above) revealing which set of addresses may sign for the transaction; the transaction being signed on (either the unsigned full data of the transaction, or its transaction hash); and a list of signatures on the transaction hash by other signers.

To manage the multisig process, the node keeps three consensus-managed tables: the Multisig Info table, the Pending Transaction table, and the Pending Signatures table. When a Multisig Info object is included in a Multisig Transaction, it will be saved in the Multisig Info table along with its hash (which is the corresponding Multisig Address.) When an unsigned transaction is included with a Multisig Transaction, the unsigned transaction and its hash are recorded in the Pending Transactions table. When some signatures on a transaction hash are included in a Multisig Transaction, a record for each is accumulated in the Pending Signatures table.

In this way, we can think of a Multisig Transaction as accumulating whichever pieces of required information the signers which to expose first. After adding any pieces of Multisig-related data to the relevant tables, the node will evaluate whether enough information is available to execute the specified Pending Transaction.

Specifically: if, for the specified Pending Transaction, the sender address is known to be a multisig address (because a Multisig Info object hashing to that address has already been revealed), there are enough signatures on its transaction hash in the Pending Signatures table (where each signature must be from an address specified in the Multisig Info), and the transaction is still valid at the time of application, then the Pending Transaction is executed as if it were a normal transaction on the blockchain occuring in place of the Multisig Transaction.

All three types of data have an expiry time, and so the Multisig Transaction will only execute in the case that these conditions are met in a timely fashion. Afterwards, unused pieces of data will be pruned from the consensus-managed tables.

Multisig Use Cases

The purpose of the complexity of this transaction type is to give users the power through one tool to accomplish various flavors of multisig transaction behavior. For clarity, some of the use cases which may be satisfied by this design are described below.

Off-Chain Multisig

In the simplest case, all optional parameters of a Multisig Transaction may be supplied in one transaction, including the Multisig Info describing who may sign for the account, the Pending Transaction to be executed by the account, and all needed signatures from the other account controllers.

In this case, the behavior of the Multisig Transaction approximates classical multisig behavior of Bitcoin and other blockchains, where the account controllers work together off-chain to prepare a single valid transaction that will be immediately evaluated and applied once broadcast to the network.

On-Chain Multisig

If the Multisig Info and Pending Transaction are already revealed, other account controllers may submit their signatures on the transaction as separate Multisig Transactions. In the extreme case, each needed signer may submit a Multisig Transaction appending only his own signature to the Pending Transaction hash, which will simply be accumulated in the Pending Signatures table until the number of signatures needed to execute the Pending Transaction is reached.

This may be useful in cases where some participants wish to prove their existing signatures on-chain before others feel confident to provide their own signatures, or in cases when another needed signer cannot be contacted but may be alerted by some application that a multisig transaction awaits his signature.

Anonymizing Multisig Addresses

Due to the structure of the Multisig Transaction, it is legal to submit a Pending Transaction from a multisig address, and all needed signatures on that transaction, before the Multisig Info (such as which addresses may sign for the address) are revealed. To further create plausible deniability for who the real multisig participants are, many other accounts may blindly submit signatures for the given transaction hash. In this case, the Pending Transaction will be executed as soon as a Multisig Info matching its sender address is submitted in a separate Multisig Transaction.

This behavior may be useful in cases where it is desirable to not reveal the controllers of an asset until they must act, especially if they are in conditions where they must individually submit their signatures on-chain. Alternately this may be used as a deterrent mechanism: a Pending Transaction may be irrevocably committed (signed and ready to execute), giving anyone who possesses the Multisig Info the power to single-handedly force the transaction to be executed by revealing it.

Concealing Pending Transactions

Even if the Multisig Info is already revealed, the unsigned Pending Transaction itself may be hidden from the blockchain until enough signers have already submitted their signatures on it's hash. (presumably, they would be informed of the contents of this transaction off-chain.) In this case, the Pending Transaction will be executed as soon as its full data is submitted in a separate Multisig Transaction.

This behavior may be useful when it is not convenient for the controllers of an account to pass around a partially-signed transaction, yet they do not want to reveal what action they plan to take unless it collects the necessary support of other account controllers. Like above, this may also be used as a deterrent mechanism, where a Pending Transaction will be executed the moment any person who possesses it chooses to submit it.

Hierarchical Multisig

Because a Pending Transaction may be any legal transaction on the blockchain, it may also itself be a Multisig Transaction. In this way, the creator of a multisig address may specify another multisig address as one of the controlling accounts. There is no limitation on how many multisig layers may be created this way.

In this case, a Pending Transaction may be created from the top-level multisig address. Before this transaction may be executed, a signature must be added by the controlling multisig address. The Multisig Transaction to add this signature becomes, itself, a Pending Transaction which must satisfy the conditions specified by its own Multisig Info before it will be executed.

This behavior may be useful when complex organizational structures are responsible for an asset. We can imagine approval from 2 out of 3 departments is needed, where each department must have the approval of 3 out of 5 managers, and perhaps a few of the managers represent oversight committees which themselves must give a majority vote to approve.

Node Registration

Although anyone can run a node and connect to the network, we restrict the creation of blocks and the distribution of coinbase rewards to a subset of ***registered nodes***. The process of how new nodes can become registered, and how nodes are ejected from the registry, is managed entirely by the network protocol. This design has two key motivations:

First, we intend to separate the account's private key from the process of block creation. In Blockchains where blocks are secured via digital signatures rather than a direct Proof of Work, the private key of the account must be present directly on the computer which creates a block. This creates a security hazard where the account's private key may be intercepted by an attacker if the computer hosting the node is breached.

Second, we intend to regulate the rate at which nodes join the network. The Proof of Participation algorithm described below has a weakness in the condition that an attacker can register many node's public keys at the same time, before the rest of the network has the opportunity to evaluate whether they are actually running nodes. Restricting the registration of new nodes preempts this attack. This attack vector is detailed in Section 2.

The Node Registry Life Cycle

Once a user has installed a node and allowed it to synchronize with the network, he may apply for a spot in the node registry by submitting a ***register node transaction***. This transaction includes some details about the node, as well as a proof that the owner controls the node's private key.

Upon submission of this valid registration request, the node enters into the ***registration queue***. The creation of new available seats in the registry, and the replacement of nodes in the registry by those in the queue, are strictly regulated by the protocol rules. When seats become available, nodes are admitted into the registry from the registration queue.

Once a node has a seat in the registry, it becomes subject to the Proof of Participation algorithm, which will gradually modify the node's participation score upwards or downwards based on its behavior on the network. The registry seat also entitles the owner to participation rewards (coinbase), with the amount of rewards received being proportional to the node's participation score.

During the node's tenure in the registry, the node's owner may update his node's details as he sees fit by submitting an ***update node transaction***. For example, if the node owner needs to change the node's key pair, or if the network address of the node has changed, or if the owner decides to increase his locked balance to increase his chances of remaining in the registry.

Additionally, if the node's owner loses his account's private key, so long as he still controls the node's private key, he may submit a ***claim node transaction*** from his new account to change the registered owner of the node. This has the deliberate consequence that if an attacker can access your node's private key, he can effectively claim your spot in the registry, and any funds you had locked to register the node.

A user may wish to remove their node from the registry deliberately to reclaim their locked funds, in which case they can submit a ***remove node transaction***. This will credit any locked funds for the node back to the owner's account.

Finally, a node may be automatically ejected from the registry if its participation score drops to zero. In this case the funds locked with the node registration are still returned entirely to the node owner's account. However, if he wishes to register the node again, he must start over with a register node transaction and wait to be selected from the registration queue.

Some aspects of this general process are discussed below in more detail.

The Node Registry

Each node on the network maintains a table we call the ***node registry***. The registry serves first and foremost as a mapping between some user's account address and the node he operates, empowering the node to sign blocks on the user's behalf without exposing his private key to insecure conditions, and allowing the user's account to be rewarded for the node's behavior. Each node's entry in the registry declares the following properties:

Field	Description
Public Key	The public key corresponding to the node's configured private key. When blocks or proof of participation messages are produced by a node, the signatures can be validated against this key.
Account Address	The account address of the node owner's account. This account may legally change or remove the node registration, and any participation rewards earned by the node are credited to this account.
Network Address	The network address (such as IP Address or domain name) at which this node can be reached. A running node should always be responsive at this address, otherwise it will be penalized by the proof of participation algorithm.
Locked Balance	An amount of funds which the node's owner has put up as collateral to compete for a spot in the registry, and to incentivize him to

	maintain the security of his node's private key.
Participation Score	A score tracked for this node over time by the proof of participation algorithm. A higher score increases the number of rewards received, while falling to zero will cause the node to be ejected from the registry.

The Node's Public Key

Each node keeps its own private key, which is used to sign blocks on its owner's behalf. The corresponding public key can be published to the network through the process described below, allowing other nodes to validate that blocks and peer-to-peer messages actually originate from this node.

In the event that a node's private key has been compromised, the owner can rotate the node's key pair and update his entry in the node registry at any time, by signing a transaction with his account's private key.

Locked Balance

To maintain a node in the registry, the owner's account must lock some amount of funds. When a node is added to the registration queue, this amount is deducted from the owner's account and kept in trust by the network. Later, if the node is ejected from the registry, or if the owner removes it from the registry or queue deliberately, the locked funds are returned entirely to the owner's account.

This serves a few purposes. New spots in the registry are only opened gradually, therefore the amount of locked balance is used to prioritize which node in the queue will take the new spot.

There is a type of attack on the proof of participation system where all nodes may voluntarily share their private keys with each other. Therefore, in the case another user can prove possession of your node's private key, we allow that they may claim your locked funds. This incentivizes node operators to treat the security of their node's private key seriously. This attack vector is detailed in Section 2.

The Registration Queue

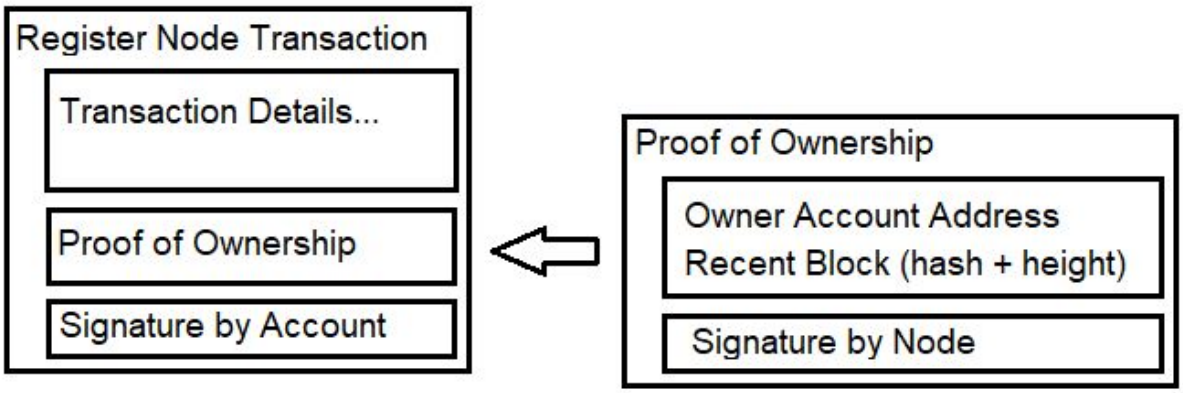
The ***node registration queue*** is a list of nodes which have been defined by a node registration transaction, but which have not yet been admitted into the node registry.

The queue is prioritized by the amount of funds locked by each registering account, such that the account which committed the most funds in its registration transaction will be added to the registry first when new admissions are allowed.

Registering a Node

Before registering a node, the owner must collect from the node a special message we call a ***proof of ownership***. This message simply contains the owner's account address, a recent block hash, the height of the block hash, and a signature by the node's private key on the rest of the information.

The proof of ownership message is then bundled inside a ***register node transaction***, together with the node's public key, the amount of funds to lock from the sending account, and the account's signature.



When this transaction is executed by the network, the owner account's funds will be locked, and the information will be added to the registration queue.

…

Updating the Node Registry

…

Ejection from the Node Registry

...

Proof of Participation

We propose a novel consensus algorithm which involves proving registered nodes are reliably online and propagating data to each other, computing a ***participation score*** for each, and using this score as a weighting coefficient to the node's pseudo-random chance to be elected to create a block or receive participation (coinbase) rewards. If a registered node's participation score falls to zero it is automatically ejected from the node registry.

Our intentions with this strategy include promoting availability of network nodes, more evenly distributing stewardship of the blockchain history among many parties, and more evenly rewarding the participation of all nodes composing the network. Additionally, we create an incentive scheme for the network of all registered nodes to organize itself into an optimal topology which minimizes the number of hops any piece of data must take to propagate through the entire network.

Overview

"Participation" could be defined in many ways, but for the purposes of this algorithm we mean that a node is transmitting pieces of blockchain-related data (such as blocks and transactions) to many other nodes in a timely fashion. While tracking the entire set of node-to-node transmissions is infeasible, we use a random sampling strategy to pick some small subset of transmissions from the network to evaluate who is participating.

Such proofs of transmission would be meaningless if they could be produced on-demand; therefore, nodes must regularly publish "commitments" of their transmission activity to the network. When a node is called on to produce some small sample of its past transmission activity, it should also prove that the records it produces were included in commitments already published on the Blockchain, which makes such records impossible to produce just-in-time.

Our network protocol specifies that each transmission should return a ***receipt***: a special object, digitally signed by the receiver, and uniquely identifying the sender. The receipt is a claim that a particular piece of data was transmitted between those nodes at that time. A receipt object also includes a "commitment": the Merkle Root of a set of other receipts which had previously been collected by the receiver.

When a node creates a block, it may include some receipts of its past transmission activity. These receipts can be validated objectively by all nodes on several criteria. The number and quality of the included receipts is used to calculate whether the node's participation score should rise or fall.

Because we wish to measure only recent activity on the network, we impose a ***height filter*** on receipts which may be included in a block, such that receipts expire some number of blocks after they were created. The expiration time is a function of the number of registered nodes, because a greater number of block creators means a greater average time between blocks created by any given node, which consequently expands the average time we expect between a receipt's publication and the time it was previously committed.

Because we wish to prove the node has received and propagated the full set of data items on the blockchain, we impose a ***data filter*** on the receipts which may be included in a block. Based on the number of data items recently included in the Blockchain, we pseudo-randomly restrict the set of data hashes which may be legally included, in a way that is not predictable before the block creator's turn. Consequently a node must keep receipts from all data items transmitted in order to reliably produce the randomly-selected subset.

Because we wish to prove the node has been in communication with many other nodes, and also to guarantee every node has a fair opportunity to accrue participation score, we impose a ***peer filter*** on the receipts which may be included in a block. Periodically the network pseudo-randomly computes a new network topology that governs which peers a given node is allowed to publish receipts from. Unlike the data filter, this filter is assumed to be known by all nodes at the beginning of each new topology period, to ensure that nodes have a fair chance to reorganize their peer connections to collect the receipts they will be allowed to publish in the future.

A receipt is only worth a lot of participation score if the publishing node can prove that the receipt was a member of a "commitment" (Receipt Merkle Root) already included in a previously-published receipt, which proves the new receipt that matches these filters existed before the filter criteria were known. When such a proof accompanies a receipt in a block, we say this receipt is ***linked***.

If a node does not have any receipts he can prove this way, he can still include un-linked receipts (not included in a merkle root published in a previous receipt) for much less score. While these receipts prove little about the publishing node, they do include commitments from their creators, which can later be used to prove pre-existence when those creators publish their own receipts.

We assign each block a value by counting a small number of points for each un-linked receipt and a larger number of points for each linked receipt. There is a fixed maximum number of receipts which can be included in a block, therefore we can clearly state that the maximum block value is given for a block filled with linked receipts, and the minimum block value (0) is given for a block with no receipts.

If the total value for the block is more than half of the maximum block value, the creating node's participation score will increase for this block; else it will decrease. In the case that a node

misses his turn to create a block entirely, he will lose the same amount of participation score as if he had produced a block with zero receipts.

The Receipt Object

Field	Size	Description
Sender Public Key	32 bytes	Public key of the sending node.
Receiver Public Key	32 bytes	Public key of the receiving node.
Data Type	4 bytes	A code indicating the type of the Datum that was sent. (Block, Transaction, File Chunk, etc.)
Data Hash	32 bytes	The hash of the Datum that was sent.
Ref Block Height	4 bytes	The height of a recent reference block
Ref Block Hash	32 bytes	The hash of the recent reference block, at the height specified
Receipt Merkle Root	32 bytes	A merkle root of receipt objects previously received by the sending node.
Receiver Signature	32 bytes	The digital signature of the receiving node (receipt producer) on all of the above data.
	200 bytes (total)	

Producing Receipts

When a node transmits some piece of data to another node (such as a transaction or block), the receiving node (after validating the data) should produce a receipt object and return it to the sender. This receipt should be created and returned regardless of whether the receiving node had already received and rebroadcast the same data from another node, so long as the data itself is valid.

The receiving node will populate the “Sender Public Key” with the public key of the node which sent him the data, and the “Receiver Public Key” with his own node public key. “Data Type” is filled with a type code stating the data is a block, transaction or potentially other kind of transmission, and “Data Hash” is filled with the hash of the data item which was transmitted. “Ref Block Height” is filled with the node’s current block height, and “Ref Block Hash” with the hash of the block at that block height.

To fill the “Receipt Merkle Root”, the receiving node will look up a recent Merkle Root in its “batch table” (described below.) While the content of this Merkle Root field cannot be validated by other nodes, it is in the receiver’s interest to include a proper commitment which he can later use to prove the prior existence of other receipts he has collected.

Finally the node signs all the above data with it’s private key, guaranteeing the receipt could not be produced without his involvement. The receipt object is then sent back to the node which transmitted the data. Repeated failure to return a valid receipt object may result in the sending node blacklisting the receiving node.

Collecting Receipts

As a node broadcasts pieces of data to other nodes, it will collect and save the receipts from each receiving node that are returned. These receipts are organized into “batches”, where a Merkle Root is calculated for each batch which can be included in future receipts produced by the node.

The number of receipts which we allow to be proven by a single Merkle Root is limited, therefore it is not in the node’s interest to save any receipts which it already knows will not be usable later. The node can safely discard any receipts which it already knows will not be legal to include in a future block, in particular receipts which do not match his peer filter.

The maximum batch size is governed by a constant we constant we define in the protocol, ***rmr max depth***: The maximum allowed depth of a Receipt Merkle Tree. Functionally, this translates into the maximum number of intermediate hashes you are allowed to publish along with a receipt to prove its membership in a previously published Merkle Root.

Receipts only need to be collected in-memory until the node is ready to finalize the batch and write it to the database. When it is time to finalize the batch, the node will first compute the Merkle Root of all receipts in the batch and save a new record in the “batch table” connecting the root to the block height at which it was created.

The node then adds all of the receipts included in the batch to the “receipt table”. Each record in the receipt table will specify the Batch ID (Merkle Root) that the receipt belongs to, and also its Sequence Number within the batch.

In this way, as the node collects receipts, it keeps a personal record of all the information it will later need to notice that one of its commitments has been published by another node, and to construct the necessary proof that some of its receipts were previously committed.

Batch Table Structure

Field	Type	Description
Batch Merkle Root	32-byte blob	The merkle root of all receipts included in this batch.
Created Height	4-byte int	The block height when this batch was created.

Receipt Table Structure

Field	Type	Description
Batch ID	8-byte int	The first 8 bytes of the Batch Merkle Root
Seq Number	4-byte int	This receipt's position within its batch
Receipt Hash	32-byte blob	The pre-calculated hash of the receipt object
[Receipt Data]		The full data of the receipt object, structured the same way as the receipt object specified above.

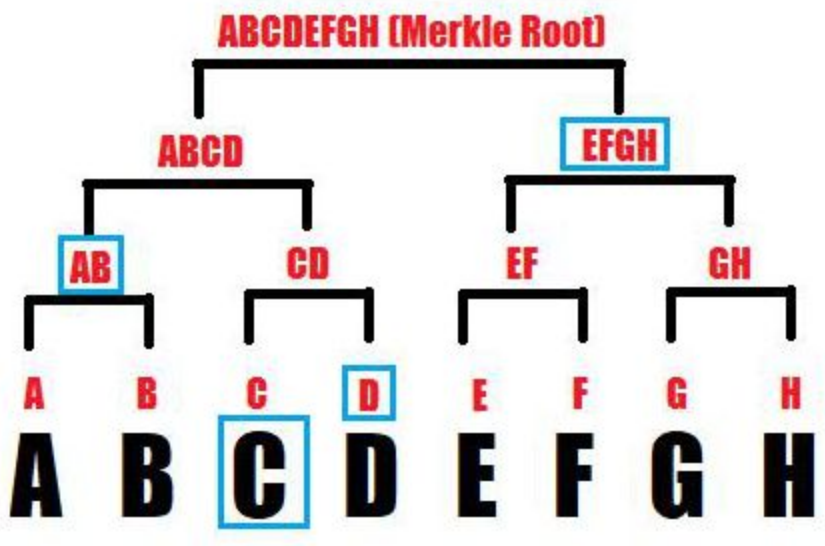
Pruning Old Receipts

As all receipts have an expiration time (based on the block height they include), there is no benefit to keeping them forever. Additionally, each batch stores the current block height when it was created, making it easy to detect when all receipts within a batch have expired. Therefore, we can efficiently and safely delete old receipts.

The node can run a pruning process which occasionally checks if any batches are expired, and clean them from the database. Therefore, the entire receipt storage process should fit in a constant memory footprint, cleaning the old ones as new ones are accumulated.

Proving Linked Receipts

When a receipt is included in a block, it can either be “un-linked” (meaning it’s prior existence cannot be proven by any Merkle Root previously published) or “linked” (when prior existence can be proven.)



The Height Filter

The Data Filter

The Peer Filter

The Spine Chain

One of our objectives is to reduce the Blockchain download time to a constant (or nearly so.) The Blockchain download time is a notorious problem, as traditionally each node must download the entire history of previous blocks before it can begin evaluating the validity of transactions, and the size of this historical record necessarily continues to grow for the lifetime of the chain.

Overview

In order to address this in our design, each node periodically (at a block height agreed by the network) takes a ***snapshot*** of the current state of his database, and computes the hash of this snapshot. Any new node joining the network then only needs to request the latest database snapshot, and download it in chunks from its peers, to come up to a recent database state, from which it can finish downloading the recent blocks to catch up to the rest of the network.

In order to secure this information, our architecture calls for a second light-weight parallel Blockchain, which we call the ***spine chain,*** to track the hashes of these snapshots as they appear. Blocks on the spine chain are only created once per day on average, and contain no transactions, resulting in a chain that is extremely light to download.

Therefore the first step for any new node is to download the blocks from this spine chain until it has identified the chain with the highest cumulative difficulty, and extract the most recent database snapshot hash from this chain, to decide which snapshot hash it should request from peers on the network.

Spine Chain Consensus

...

Snapshots and Backups

File Distribution

Archival Nodes

Coinbase Distribution

Conventionally, only the node which produces a block on the network is rewarded with newly minted tokens. If the network becomes very large, the chance that any given node will produce a block (especially one with a lower participation score) in some period of time, and therefore receive a reward, becomes small.

One of our major objectives with the Node Registration and Proof of Participation algorithms is to more fairly reward the full set of network participants, and in a more timely manner. In keeping with this goal we implement a pseudo-random lottery to reward many accounts per block, with their chance to win being weighted by their participation score.

Here we detail how we compute the amount of new tokens minted per block, and how they are distributed to network participants.

Coinbase Schedule

Styled after Bitcoin, many Blockchains offer a fixed reward per block for some period of blocks, after which the reward amount is cut in half for the next period, and so on. This geometric reduction ensures that the earlier participants are rewarded more than the later ones, and also that the number of tokens produced will approach, but not exceed, a target total supply.

We feel it is cleaner to define a smooth curve over all blocks rather than explicit halving events, such that the number of new tokens minted produced by a block is a simple function of its block height. We give this formula as:

[include formula]

Based on this rate of distribution, we expect to reach a target supply of 1 billion tokens over XX years.

Recipient Selection

Each block, a list of coinbase recipients is computed deterministically from the current state of the Node Registry and the new block's seed. We define an *ordering function* to compute a pseudorandom number for each node, weighted by its pop score, then select up to a maximum of X nodes with the lowest computed order numbers.

Where `hash` is the first 8 bytes of a SHA3-512 hash, PK_N is the public key of node N, PS_N is the pop score of node N, and BS_H is the first 8 bytes of the block seed at height H, we give the ordering function as:

$$order_N = xor(hash(BS_H), PK_N) * (1 / PS_N)$$

Once we have computed this list of winners, the total block reward available according to the provided Coinbase Schedule function is divided evenly between the winners, with the newly minted tokens being credited to the winning node's associated account.

It is worth noting that pop score has a fixed maximum, which any good node should attain after some time. Therefore in a large network of good nodes, rewards should be frequent, fairly distributed, and not directly tied to recent block production.

Conclusion

Section 2: Security Considerations

Fee Scaling Attack

[attack vector where all registered nodes vote for a fee too high to disrupt the network]

Digital Signatures: Weakest Link Attack

[attack vector where someone tries to create a signature using the weakest of the available signing algorithms for a transaction address]

[mitigated by explicitly listing the recipient account's signature type in the transaction, such that this algorithm is required to spend]

Proof of Participation: Ghost Attack

[attack vector where an attacker controls enough of the registry that he doesn't need real nodes, or even proxies, to generate receipts on demand]

[mitigated by limiting the rate of node registry (exposing new nodes to the existing nodes before other attacking nodes can be registered), and requiring nodes to compete for amount of locked funds meaning they need scarce resources to register]

Proof of Participation: Key Share Attack

[attack vector where many nodes agree to reveal their private keys to each other to ensure they can generate pop couplets on demand]

[mitigated by the rule which allows ownership of a node's public key to claim the node's locked funds for yourself, creating incentive to report nodes which have made their private keys public]

Spine Chain: Nothing At Stake Attack

[attack vector where someone rebuilds a spine chain with a higher cumulative difficulty than the "real" chain to trick new nodes]

[mitigated by difficulty calculation involving who signed on each new signing key]

Section 3: Appendices

Appendix 1: Digital Signature Types

List of supported digital signature algorithms

Appendix 2: Transaction Types

Introduction …

Send Funds Transaction

...

Register Node Transaction

...

Update Node Transaction

...

Claim Node Transaction

...

Remove Node Transaction

...