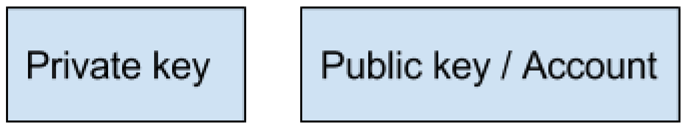
RaiBlocks distributed ledger network

by
Colin LeMahieu

Abstract:
Distributed ledgers have been around for several years as of the time of this writing yet adoption of the platforms has been low and initial adopter markets have failed to materialize. The transaction performance of these systems compared to centralized system is significantly worse relegating them to niche markets that can only capitalize on the benefit of decentralization while tolerating slow speed.
RaiBlocks is designed to be a scalable and efficient distributed ledger platform. The design makes several significant improvements over alternatives giving a simple system that can process transactions in seconds making it a useful in a digital world.
To get these performance improvements we apply a critical optimization borrowed from concurrent computing where we replace shared state with message passing. This eliminates shared state contention greatly improving global throughput and lowering transaction time.
RaiBlocks builds on an analogy from the electrical engineering discipline by equating network consensus to arbiters circuits. This gives RaiBlocks an established and well researched modeling basis for how the system comes to a distributed, egalitarian, and efficient conclusion.

Account:

An account is the public key portion of a digital signature keypair. As with all digital signature system, the public key can be given to everyone and the private key must be kept secret.

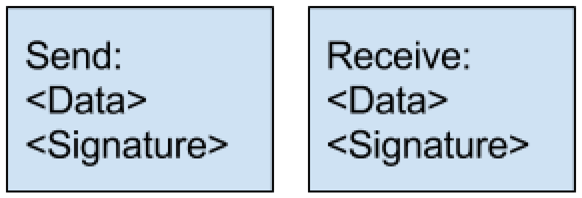


Balance:

Each account has an associated balance. The system is initiated with a genesis account containing the genesis balance. The genesis balance is a fixed quantity and can never be increased. The genesis balance is divided and sent to other accounts who further divide it and send it to other accounts. As such, the sum of all balances of all accounts will never exceed the initial genesis balance which gives the system an upper bound on quantity and no ability to increase it.

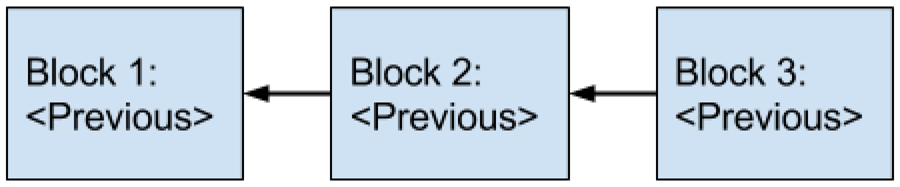
Block/Transaction:

The term block and transaction are often used interchangeably. The term transaction specifically refers to the verb of what is being done while block refers to the digital encoding of the transaction. Transactions are signed by the private key belonging to the account on which the transaction is performed.



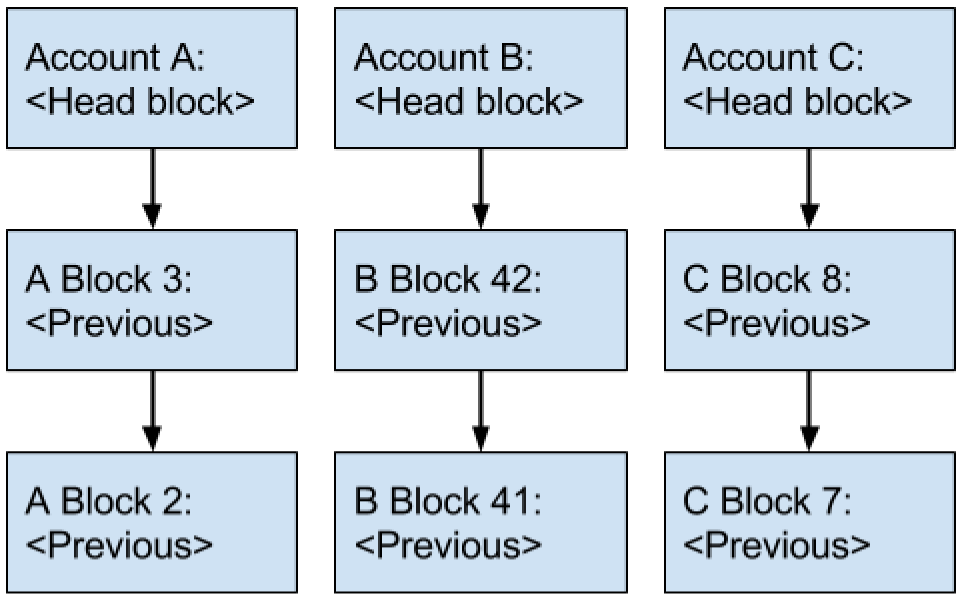
Chain:

Chain refers to a singly linked list of blocks, linked by the hash of the contents. By virtue of the fact that every block contains a hash of the previous block, essentially a random number, every block is unique and will have a unique hash value.



Ledger:

The ledger is the global set of accounts and each account has its own chain. Only one entity can add a transaction to an account's chain, the account owner. This means each chain is not a shared data structure eliminating all contention when adding transactions to the ledger.

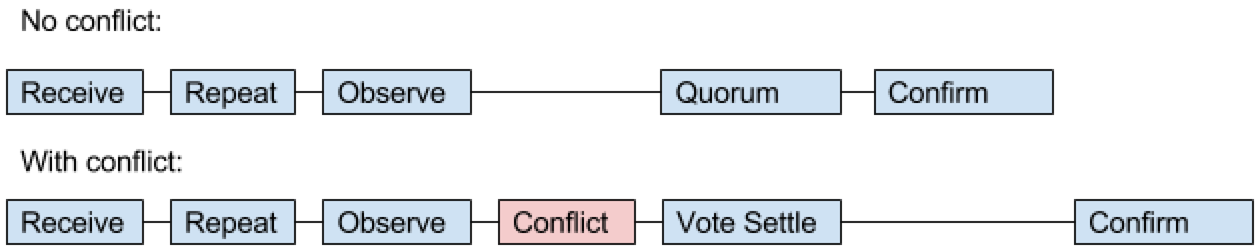


Node:

A node is a piece of software running on a computer connected to other nodes via the internet. The software manages the ledger and any accounts the node may control, if any. The node connects to other nodes over the internet using IPV6/IPV4 and UDP.

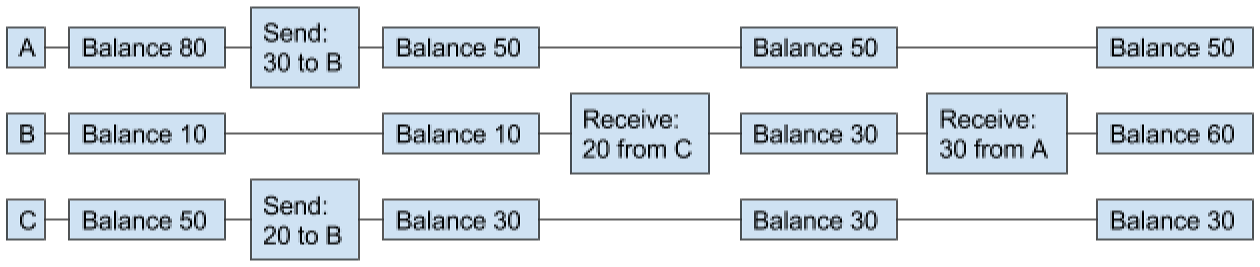
Confirmation procedure:

When a node receives a send block to an account it controls, it first runs the confirmation procedure followed by adding the block into its ledger. Next the node repeats the block back out to the network to announce the block it's observed. The node then waits and observes incoming publish and confirm messages to see if any conflicting blocks are published. Non-voting nodes will transmit unsigned publish blocks and voting nodes will sign the block with their voting key and publish a confirm message. A message is considered confirmed if there are no conflicting blocks and a 50% vote quorum has been reached. If there is a conflicting block the node will wait 4 voting periods, 1 minute total, and confirm the winning block.



Two phase transmission:

Moving a balance from one account to another requires a transaction in both the sending and receiving chain. This allows both accounts to modify their chain at will and in any order they choose.

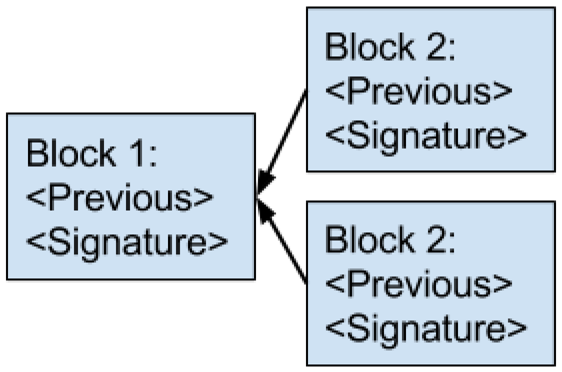


<u>Network flooding:</u>
Each node in the network must be aware of all transactions as they occur. When a node receives a block it hasn't seen before it broadcasts this block to all other nodes it's aware of. This is called network flooding and gives the greatest probability that all nodes will receive a copy of the transaction.

<u>Network echo period:</u>
In an ideal case, each node will receive a duplicate copy of the same transaction from every node it's aware of; this is a result of network flooding. The time between receiving the first copy to the time it receives the last duplicate is what we call the network echo period. This property is important because it gives us a view of transactions accepted by other nodes on the network. This period is probabilistic based on the network latency between nodes but a node can establish a reasonable bound for itself on the duration of its own network echo.

<u>Quorum/Partitioning:</u>
An important property of coming to an agreement is determining quorum or in the case of networked system, determining both if the network is partitioned and preventing a sybil attack. RaiBlocks is able to make use of a fixed quantity it controls, the genesis amount, to determine normal participation e.g. quorum, and to limit participation directly to users who have an interest in maintaining the system. An account's participation weight is its balance as a percentage of the total supply.

<u>RaiBlocks bad actor:</u>
RaiBlocks provides the ability for accounts to do balance transfers from their chain without ambiguity and without contention since only one owner/signing key is responsible for each chain and the specification requires the owner to pick a linear order for entries to its chain. This is something systems that use shared state e.g. a monolithic block chain, are not able to accomplish. There is no guarantee an account owner will follow the specification either through poor programming or malicious intent. We call this ambiguous situation a “fork” and it’s conceptually a break in the singly linked list nature of their chain.



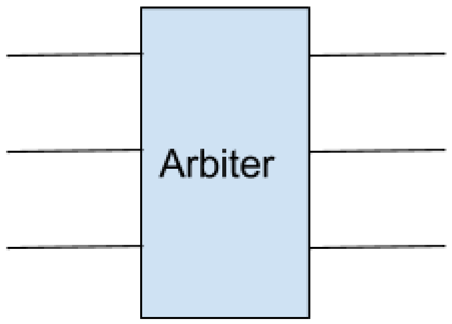
As we can see here, there are two signed blocks claiming block 1 as their predecessor. Depending on the order these blocks were received by different nodes in the network, they could have a conflicting views of the account state. We can determine this ambiguity wasn’t caused by some random person trying to break someone else’s chain by the nature of the fact that the blocks are signed and signatures can only be generated by the account owner. We also know this situation wasn’t accidental or at the very least it was the result of incorrect programming due to the specification requiring nodes to not emit these sequences.

<u>Fork risks:</u>
The network is designed so all nodes resolve forks to a single branch which means

losing branches will be rolled back and deleted which also means removing all dependent blocks in any chain in the system. This means accounts that are receiving balances must allow adequate time for settling to complete or risk having their chain rolled back. In reality the settling protocol is on the order of a couple seconds at which point a critical mass of nodes have decided on a single branch and the decision won't be undone.

Arbiter circuit:

An asynchronous arbiter is an electronic circuit that takes multiple inputs and signals at most one output. If more than one output is signalled we call this a glitch on the output. There are mountains of literature on why arbiters act the way they do, an exercise left to the reader, and the synopsis is: There is no deterministic arbiter, we can only avoid glitches probabilistically and if the arbiter is well designed the probability converges exponentially over time.



Despite the fact that arbiters are non-deterministic, all of our computing hardware includes them and the world functions on top of them. We achieve this by making the convergence happen as fast as possible and by determining an MTBF with which we're comfortable. The mantra on arbiters is: if you want to be more sure of avoiding a glitch, make the physics of the arbiter work faster or wait longer and we adopt this methodology in RaiBlocks.

We view forks as signalling multiple inputs, the arbiter as the network as a whole, and the outputs are the block the network chooses to survive arbitration. A glitch would be node that thinks it has followed the arbitration protocol, accepts a balance into its chain and later has its chain rolled back.

Fork identification:
Identifying a fork is faster than resolving it, indeed we're able to identify forks within one network echo period by virtue of the fact that all nodes flood the network with any transaction they accept. A receiving account trying to avoid risk of having blocks reverted due to a bad sender should wait several network echo periods to see if anyone announces a conflicting block. If they see a conflict they should wait even longer until resolution has completed before reevaluating if they want to accept the balance. A receiver can continue interacting with other accounts while they wait for the problem account to resolve due to each chain acting asynchronously.

Fork resolution:
The arbitration itself is performed in a distributed fashion by the whole network as a balance-weighted vote on which branch of a fork to accept. Each node which controls an account with a balance evaluates the tally of votes it's seen and changes its vote to match the entry with the highest vote total. Periodically each node broadcasts its vote until the end of the resolution period. Using balance weighted voting allows nodes to identify quorum and makes sure only interested parties are voting.

Global throughput limit:
There is no inherent limit to how many transactions can be processed by the system. Other systems commit a slate of transactions based on timing parameters and limit the number of transactions that can be included in each slate. We process each transaction individually and each chain operates asynchronously.

Trustless tallest block intractability:

One frequently cited advantage of using Proof of Work systems over Proof of Stake is the trustless ability to determine what the current state of the network is; the current state is always the tallest chain. With Proof of Stake, nodes entering the system could be provided a view of the system that is out of sync with the rest of the world and since this view would also have different voters, trust would be needed to resolve the situation.

We argue the trustless nature of determining the tallest block chain is intractable, trusted sources are commonly used already when downloading node software, and a node with an out-of-sync view of the network would be quickly discovered when you're trying to interact with the rest of the world and everyone rejects your transactions.

Determining the tallest block chain requires evaluating all possible chains for their length. Since downloading the longest chain can range from hours to weeks at the current size, downloading all possible chains to evaluate which is the largest isn't reasonable. A node could be fed a significant amount of short, junk forks opening up them up to a denial of service situation. Contemporary node software have a built in floor under which it's assumed there is no longer a branch which means the user is trusting the software programmer to know the longest chain in existence. In addition, users trust software writers to not exfiltrate encryption keys purposely or accidentally.

In a real usage situation an out of sync node would be quickly discovered when interacting with a retailer or vendor that would reject your attempted payments. If someone synchronizes their block store using an unknown source and attempts to engage in transactions with a well known retailer or a well known bank, it's most likely the node was incorrectly bootstrapped.

Block reward incongruent with system's health:

Block rewards or transaction fees are frequently used as an incentive to process

protocol transactions and run nodes. We argue these rewards only reward a small part of everything that’s required to keep the protocol healthy and if the system provides bonafide utility, keeping it operating has greater external incentives compared to block rewards.
Keeping a distributed ledger system running requires processing transactions, serving up historical data, writing node software, running companies, satisfying legal requirements, and marketing. If there are incentives for performing these other services and those incentives are transacted out of chain, surely the small expense of processing transactions can be rolled into these other expenses.

<u>Additive block reversal difficulty:</u>
An often cited advantage of using a Proof of Work where nodes pick the chain with the most amount of work is that it becomes increasingly difficult to reverse transactions the farther they go back. While this is correct by the definition of the mathematical asymptote, what a users needs is a percentage probability that their transaction will not be rolled back and they want the time they have to wait to be as short as possible. If >99.99% assurance can be reached in 15 seconds with one system and 60 minutes with another, it makes little difference to the user going forward the asymptotic convergence as time goes to infinity.