Gregory Magarshak November 20, 2017

Eliminating Double-Spending without Global Consensus
Community Coin Technology Whitepaper

ABSTRACT

Payment networks that process monetary transactions require a technology that will eliminate the double-spending problem. Today, there are many innovative solutions to this problem that rely on global consensus, such as Bitcoin, Ripple and Hashgraph. However, the process of achieving global consensus comes at a serious cost. First of all, it is slow and requires eventual knowledge of all consensus participants and transactions: any time a group of participants achieves a consensus, it may be overturned when they later learn of a larger group of participants that achieved a conflicting consensus, invalidating previous payments. And secondly, the process can completely break down when more than 1/3 of the participants are "dishonest", allowing them to prevent any further progress in the system indefinitely. Luckily, global consensus is not required for building a decentralized payment network. This paper describes a payment system without the double-spending problem, and without global consensus or blockchains. It is not subject to the vulnerability of 1/3 dishonest participants. The system is completely decentralized and can scale to communities of any size that want to run their own payment network.

1. Introduction

The Intercoin project is building technology for communities to spin up their own payment networks that can operate independently of a giant, global blockchain. The Intercoin Whitepaper describes the economic benefits to a community issuing their own currency and becoming self-sufficient.

Over the past decade, crypto-currencies have attracted a lot of attention and capital from around the world. They exhibit many desirable properties that plain cash or centralized banking systems don't have, among them being:

- **Permissionless Transactions:** If *A* wants to pay *B*, no one in the network can stop them.
- **Security:** If *A* has a certain amount of currency, no one can take it without *A*'s signature.
- **Electronic Payments:** Payments can be made without meeting in person.
- **Intelligence and Flexibility:** New applications can be developed that interact with the currency, including multi-signature transactions, smart contracts, etc.

Communities that wish to issue their own crypto-currencies need a technology that enables "crypto-currency for the rest of us". It would be great if such currencies exhibited the above four properties, but current technologies fall short of this goal. Proof-of-work blockchains can be dominated by a "51% attack" by anyone with a large enough "hash rate". Proof-of-stake systems lead to more "centralized control" over accepting transactions, undermining the "permissionless" quality of transactions that is desired in a crypto-currency. Some community currencies, such as Bristol Pounds, require a central bank.

This paper describes, from the ground up, a system to power payment networks which retain the desirable properties of crypto-currencies while enabling communities to manage their own.

1a. Overview

Here is a summary of the general ideas discussed in this paper. They proceed in a series of optimizations and clarifications, starting with a global consensus about all transactions, ending up with a flexible system that, ultimately, doesn't even *require* consensus in order to operate.

First of all, in practice there is no need to keep a global ledger of all transactions in the world. Instead, we can partition the whole system into "communities" that issue tokens which can only circulate within the community. This is the main idea behind the architecture of Intercoin, and in particular it has major technological advantages. When looking for whether a token is double-spent, we only need to look within the community, and not at all transactions in the world. Global payments are done by trading the token of community A for an "intercoin" token, and then trading that for a token of community B. (Many variations on this exchange can exist.) This architecture is also the proper one for micropayments for a certain type of interest (e.g. movies on Netflix or websites on the Brave browser). It's also good for doing ICOs, democratic control of monetary policy, calculating Local CPI, implementing Basic Income, and many other things.

Second, notaries do not need to track the balances of all accounts, but can track tokens and their history. If tokens are able to split into multiple tokens (e.g. $30 into $20 and $10) and multiple tokens can join into one (e.g. $5 and $5 into $10) that allows a subset of notaries to be "assigned" to watch for double-spends on a given token. Recipients of a token would require a certain minimum number of notaries before approving the transaction and releasing the goods. If the token is worth very little, for example $0.01, recipients may be content with just one notary, since the worst it can do is collude with many vendors to each release $0.01 of value. However, for a large transaction, many notaries may be required. Typically, when tokens join together into a larger token, their groups of notaries are joined too. Notaries may require payment for their services, which is implemented in a simple way by "splitting off" the payment from the token being spent. However, since notaries may be run by a community with known membership, notaries may even offer to process the first 100 transactions each day for free, leading to zero fees on everyday transactions for most participants.

Today, (nearly) all payment networks, including crypto-currencies, implement some sort of consensus about the balances in people's accounts. It turns out – and this is the big claim in this paper – consensus is overkill both for the problem of validating "legal" actions in a stream (e.g. moves in a chessgame) as well as for proving that a fork (e.g. double-spend of a token) occurred. Achieving consensus is necessary when multiple possible values or histories are being proposed, such that no one can independently check which one is correct. However, that's not the case with payments and smart contracts. The potential recipient is the entity interested in validating the history of the stream (token, chess game etc.) and can independently verify that all actions taken thus far have been correct (followed the rules in effect at that point, was signed by the correct participant, and so on). However, it may happen that two conflicting forks – which are both valid histories about a token – have been submitted to the network. In this case, honest notaries assigned to the token will gossip the transactions that caused the fork, with their non-forgeable cryptographic signatures, and detect the fork. As with validity, all it takes is one honest notary to gossip the "proof of fork" to the Recipient who can *check it for themselves*. Rather than consensus, we need something much weaker: the ability for an honest notary to "get the word out" about a "proof of fork", and not be suppressed by its neighbors.

2. Merkle Trees and Directed Acyclic Graphs

A very common and useful element in distributed systems is the ***Merkle Tree***. Let *h* be a function that takes as input any document, and returns a highly-collision-resistant hash of its contents. Then a Merkle Tree based on *h* is a tree of hashes, where the label of each document is the hash of the labels of its direct children. The documents themselves may be replicated and hosted by many different servers in a network, but the Merkle Tree structure provides provides certain consistency guarantees.

For example, one can verify that a retrieved document has not been tampered with by computing its hash. Thus, network participants can refer to the document simply by its hash, and confidently retrieve it from each other on demand. Furthermore, one can refer to an entire Merkle Tree by its root hash, download the sub-trees and documents from various different network participants on demand, and be sure of its integrity. To verify a document *a* belongs in a Merkle Tree *b* one only needs to compute the hashes along the ***Merkle Branch*** of *a*, which is the set of all hashes in *b* along the path from *a* to the root of *b*, together with all their siblings. This operation is O(log *n*) where *n* is the number of all hashes in the tree

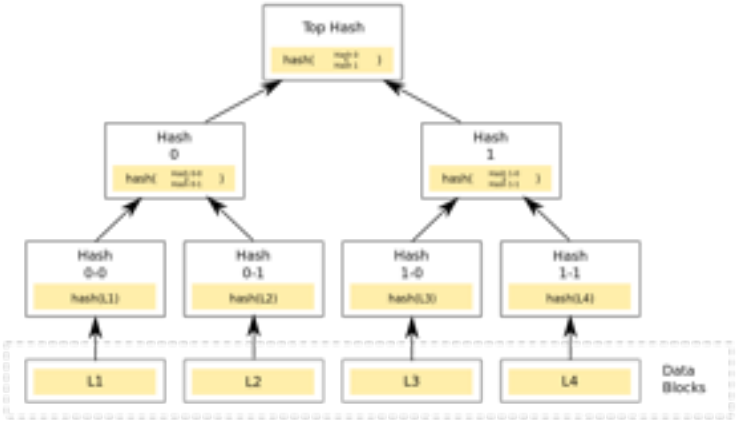


Diagram 1

A ***Merkle DAG*** generalizes the Merkle Tree to allow not just “forking” but also “merging”. A DAG is simply a graph with no cycles, whereas a tree is a DAG with the additional property that each child has at most one parent it points to.

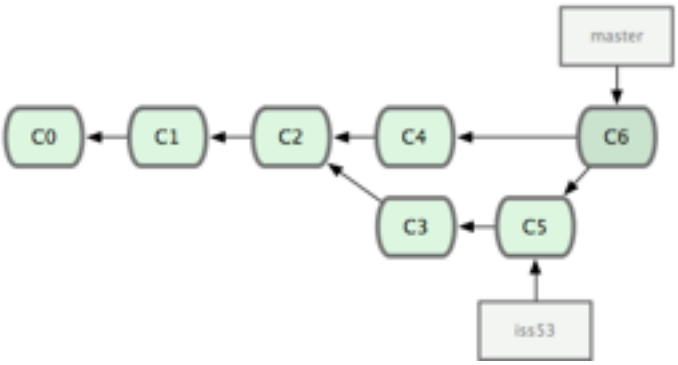


Diagram 2

Merkle DAGs are used in all kinds of applications, including version control (git, mercurial), file storage (BitTorrent, IPFS). They can be used to host decentralized conversations, where each comment references some previous comments, and much more.

Note that Merkle DAGs by themselves do not require any consensus to take place. Anyone can start their own git repository, for instance, and choose to share it with others. Any pair of people *a* and *b* can choose to connect and "pull" or "push" changes from/to one another, without the permission of anyone else, including a blockchain somewhere.

We will also define one more concept, called a ***Merkle Stream***, as being a Merkle DAG where each document has at most one child. Thus, it allows "merges" but not "forks" – the complementary concept of a Merkle Tree.

The arrows in the Merkle DAG usually point backwards in time. In the diagrams that follow, we are going to illustrate our DAGs with arrows pointing in the direction of increasing time. In any case, Merkle DAGs are an excellent fit for modeling data that evolves in time.

3. Permissionless Timestamping Network

This concept is at the heart of the payment systems we are developing. It is motivated by the problem of *proving* that some event *a* happened before event *b*.

Proving that *a* happened after *b* is easy: the document *a* can simply reference the hash of *b*. This requires, of course, that whoever publishes *a* (call them *A*) has access to the content of *b* or at least its hash *h*. If *b* is originally published by *B* and disseminated widely, then *A* can come to possess *h* without needing *B* to expend resources sending it specifically to *A*; indeed, without needing *B*'s permission at all.

Now, proving that *a* happened before *b* requires *b* to reference the hash of *a*. It would be great if this would be done without *B* needing to expend resources specifically obtaining it from *A*, and even if *B* wouldn't have given permission to *A* had *A* specifically requested to timestamp *a*.

The Permissionless Timestamping Network (henceforth referred to as a **PTN**) is designed to achieve exactly this. It's a networking protocol that combines mechanisms and incentives that result in a network of computers constantly ready to timestamp any event in a permissionless manner. As long as at least one PTN participant agrees to timestamp an event, all PTN participants will eventually timestamp this event.

Any computer can choose to join a PTN simply by finding one or more existing participants that agree to peer with it. Participants in the network can thus add and remove **neighbors**. The **PTN protocol** simply consists of a loop run by a participant *A* that does the following every so often:

1. Get next neighbor *B* (cycling through neighbors) and its latest hash b_k
2. If their latest hash b_k didn't change, go to step 1 on the next tick
3. Generate a payload $p_{n+1} = a_n + b_k$ and its signature s_{n+1} using *A*'s private key
4. Generate a new hash $a_{n+1} = h\,(p_{n+1} + s_{n+1})$ and let neighbors read a_{n+1} and s_{n+1}
5. Extend the Merkle Skip List using a_{n+1} as described in "Merkle Skip Lists"

Thus each participant *A* generates a continuous Merkle Stream of hashes that prove that *A* has seen the hashes of its neighbors.

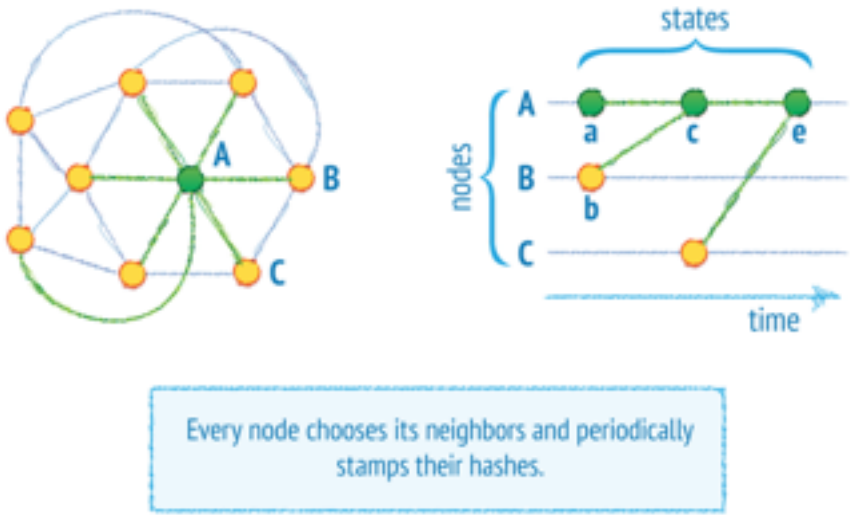


Diagram 3

Taken all together, this generates a network that can prove when a certain event *b* happened, relative to any given Merkle Stream. For example, in Diagram 3, we can see that relative to Merkle stream *A*, the event *b* happened before the event *d*. It is always unambiguous because each participant *A* timestamps its neighbors one at a time. Thus the Merkle Stream that is formed is a **binary Merkle Stream**, meaning that each child hash can have at most two parents.

Note that this timestamping network is permissionless. In Diagram 3, when *A* forms hash *c* from *a* and *b*, and then signs *c* with its own private key, that hash + signature serves as proof that *A* has seen the hash *b* happen before *c*. But hash + signature *b* may itself have served as proof that *B* saw hashes from some earlier events published by some other participants. By "timestamping" hashes from *B*, *A* serves to timestamp hashes from participants it may not like or even know about. It's all or nothing: to participate in the network, *A* must take actions which ensure permissionless timestamping

4. Relative Time Intervals with proof

Thus, a participant *A* in the PTN can get their event timestamped relative to the Merkle Stream of *every* participant *B* if they simply get it timestamped by *any* participant in the PTN, and then wait long enough for a chain of hashes to propagate to *B*, and back.

If *B* becomes slow or unavailable, then fallback Participants can take over for timestamping (this is described in Section 10: Availability).

This enables provable assertions about access permissions, which are illustrated in Diagram 4:

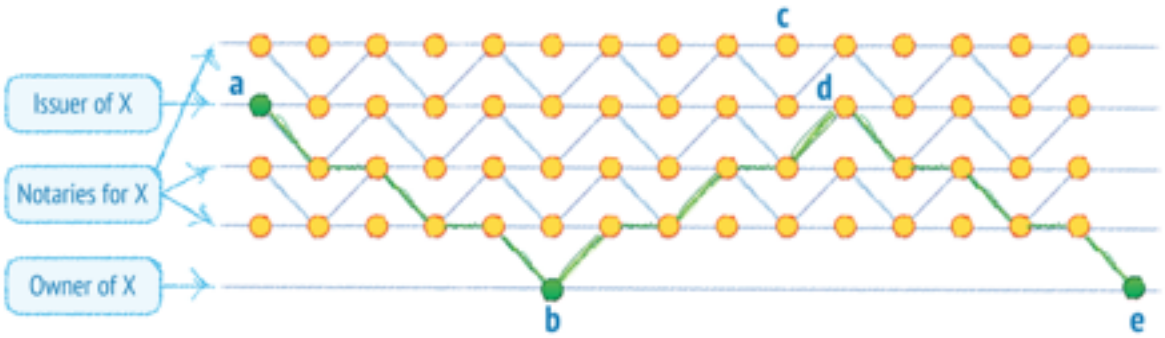


Diagram 4

Here, a certain resource *X* is issued by a participant *A* (the "Issuer") and assigned to another participant *B* (the "Owner"). The intermediate participants between *A* and *B* serve as *notaries*, propagating a chain of hashes from *a* to *b*, each one signing their payload with their private key.

Suppose at point *b*, the Owner takes some action which requires owning *X*. At the moment, they can prove to anyone that *X* was issued to them before *b*, but not that *X* hasn't been subsequently revoked before *b*. That will soon change when we extend the PTN protocol.

By generating a hash of *b* and broadcasting it to its neighbors in the PTN, the hash chain eventually reaches back to the Issuer (hash *d*) and then back to the owner (hash *e*). Now, the owner can at the very least prove that *b* happened between *a* and *d* – that is to say, the owner has a Relative Time Interval with Proof (**RTIP**) of when they undertook action *b* on resource *X*. But to do this, they will need the chain of hashes *b* – *d* – *e*, discussed in 6b: Hash Chains.

As an aside, note that the time intervals are not absolute, but relative to a single Merkle Stream. Moreover, the Merkle Stream is a binary Merkle Stream. Otherwise, there could easily arise ambiguity as to which event occurred first. For example, in Diagram 6, the owner may take two conflicting actions, that would be timestamped together in *d*, creating ambiguity.

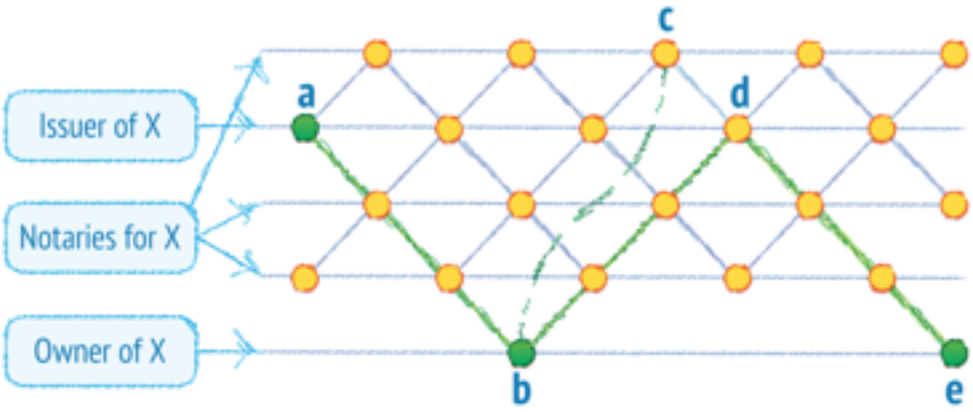


Diagram 6

4b. Merkle Skip Lists

Each participant in a busy network may engage in timestamping quite often – perhaps once every few milliseconds, for instance. This will result in a large Merkle Stream of hashes. To be usable as timestamps, there needs to be a way to compare and prove which hash happened before another in the Merkle Stream.

We could, of course, prove that one hash in the Merkle Stream happened after another by producing all the hashes between them. But, that would potentially be a very large amount of hashes for the participant to store, and for the network to transmit as proof that one event happened before another.

One potential solution would involve participants incrementing some counter t_n with each hash. But, this requires a complex system of participants policing one another for cheating (such as failing to produce some $t_{n+1} > t_n$) and then gossiping this proof to invalidate any results relying on the comparison of timestamps. (For example see 8.1: Misbehaving Notaries).

However, we can store timestamps in a way that allows provable comparisons, a way which is a compact and doesn't require storing the entire Merkle Stream. Thus we introduce the concept of a ***Merkle Skip List:***

Diagram 5

On the bottom we have the Merkle Stream being generated by a Participant as described in Section 3 and illustrated in Diagrams 3 and 4. However, whenever the Participant adds a new hash, it now takes additional actions. Every 2^n hashes, it generates a hash for a "fast lane n" Merkle Stream as shown in Diagram 5. (We can also have any factor besides 2, such as every 10^n hashes).

Now, the timestamps to be stored and transmitted consist of the chain of hashes from the root of the Merkle Skip List to the hash in question. So for example, the timestamps of hashes *a* and *b* are illustrated in Diagram 5 as Merkle branches along the green paths in the Merkle Skip List. The size of such timestamps is O(log *n*) where *n* was the total number of hashes in the Merkle Stream the Participant has produced. And to maintain the Merkle Skip List going forward, the Participant simply needs to store the latest hash on the Merkle Stream and each of its "fast lanes", which is also O(log *n*).

Merkle structures have a great property that the existence of a hash implies with near-certainty the existence of the data and integrity of the hash order. This allows comparing timestamps with proof, ensuring that if a Participant is available and reports something, it is always correct.

5. Ownership

Although the PTN can furnish all participants with proof about the time interval they undertook an action with a resource X, relative to its "Issuer", this is not enough to prove ownership of *X* during that time interval. To handle questions of ownership or more complex questions of authorization to do a certain action with resource *X*, we will need to extend our protocol beyond simple timestamping.

Suppose the Issuer simply signed resource X over to Mallory in a transaction. Now, Mallory can prove that she was once given ownership of the resource, and by posting it to the PTN, she can even obtain the the RTIP of her action fairly quickly.

Since we want our crypto-currencies to support Permissionless Transactions, Mallory's signature alone should be enough to reassign ownership of *X*, without permission from some third party ledger, whether a centralized bank or a decentralized consensus.

Now Mallory signs a transaction where she attempts to assign ownership of *X* to Alice. Alice can verify the above proofs, but she still wants to make sure that Mallory's ownership of *X* hasn't been revoked after she's been given owneship. In crypto-currency terms, Alice wants to make sure that Mallory didn't "double-spend" the tokens she is now offering to Alice in a signed transaction.

Thus, the double-spend problem can be reduced to a search and incentive problem. Namely, Alice needs to query participants on the network as to whether any one of them can send over a previous transaction, signed by Mallory, assigning ownership to someone else (e.g. Bob).

Search problems do not require global consensus, and indeed, benefit from **sharding**: for each resource *X*, only some of the participants in the network agree to act as Notaries (as depicted in Diagram 4). When the Issuer signs a transaction assigning ownership of *X* to its first owner (Mallory, or some previous owner), this transaction is gossiped through a chain of notaries, which all sign off on having seen it.

Additionally, these notaries begin to keep a compact key-value ledger storing the latest transaction and receiving owner's id (e.g. public key) for each resource *X* which they have agreed to be notaries for. A **valid transaction** has to be signed by a minimum number of notaries, promising that they stored it.

A valid token *X* should always contain in it the list of notaries that have agreed to be responsible for storing the latest transaction related to *X*. This is an extension to the PTN protocol called PLN, described in section 7. Thus, when Alice needs to find a transaction signed by Mallory giving away *X*, she knows exactly who to ask. If any one of these notaries responds with a transaction timestamped later than the one in which Mallory got ownership of *X*, Alice refuses to accept Mallory's attempted double-spend payment (see section 8.2). Moreover, this proof, when presented to the network, can penalize Mallory and/or reward the first notary which furnished Alice with the proof (see section 9. Incentives).

Unlike global consensus protocols, this system does not require 2/3 of the notaries to be honest, or to agree on a value. As long as even one notary furnishes the proof of double-spend, Alice rejects Mallory's payment. The question of incentives is handled in section 9.

6. Routing

As participants form and break connections with neighbors, we update a distributed routing table that is used for forwarding information required for proofs. The routing table stores ids of participants in the network, and for each one, the minimum number of steps to that participant.

Here is how the table is first generated: when a participant joins the PTN, it records its own id with the value 0, and the ids of its neighbors with the value 1. Then, it asks its neighbors to send their routing tables, adds 1 to each of the values (for the extra hop) and simply takes the minimum value of any duplicate ids it sees. For each participant id it sees, it then records the id of the neighbor that is "closest" to that participant. So, for instance, if one neighbor *B* takes 4Ish hops to get to participant *C*, while another takes 8 hops, then under *C* the routing table records 4 + 1 = 5 as the number of hops and *B* as the neighbor to forward stuff to.

Having formed this table, the participant then generates a unique id, sends it to its neighbors and they update their own routing tables as above, with their neighbors updating their routing tables, and so on. Because of the unique id, the wave of updates doesn't double-back on itself.

Similarly, when a connection between participants is removed, the table can be regenerated, or quickly patched to reflect the new routing information.

Now, for participant *A* to find send a message to participant *C*, they just have to send an envelope their neighbor *B*, with a final address of *C*, and participants will forward it along the network according to the routing table.

6b. Hash Chains

Using the Routing mechanism above, participants can append their hash to growing hash chains in the messages being routed. A typical route is shown in Diagram 4, where the Owner of a resource takes an action at *b*, then submits it to the PTN for timestamping relative to the Issuer at *d*, and by the time it gets back to the owner at *e*, the Owner would be able to prove the timestamp relative to the Issuer by producing the Hash Chain which includes *b, d, e*.

Suppose a message is routed from *A* to *D*, and has been submitted to participant *B*. Now on the way from *A to D*, if *C,* a neighbor of *B*, is closer than *B* to *D* according to their internal routing tables, *C* is supposed to collect the hash chain along the way to *D* and back, prepend and append its own hashes, and then pass it to *B*, which will do the same downstream. This hash chain is the *b – d – e* hash chain illustrated in Figure 4.

Building the Hash Chain relies on the intermediate participants in the RTN to behave correctly when routing messages. This is an additional requirement that participants must police each other about, and if one cheats, the others must gossip proof of the cheating and disconnect from that participant. The main way a participant can cheat is by not fulfilling its obligation to pass the Hash Chain along to the next participant.

Hash Chains provide proof that a certain action happened "before" or "after" a certain timestamp relative to a Merkle Stream. They constitute proof of an RTIP.

7. Ledgers

With the above machinery, we are now ready to extend the PTN into a Permissionless Ledger Network. Each notary is going to keep a ledger of resources *X* that it is responsible for, and for each *X*, it will store the latest transaction it saw, and id of the participant who received *X* in that transaction. This doesn't depend on global consensus – each notary is capable furnishing cryptographic proof of a double-spend if one occurred.

When *X* is first issued, some notaries are selected to store *X* in their ledgers. Forwarding of the transaction from the Issuer to the Owner proceeds by routing messages through these notaries.

Messages travel alongside the edges of the Merkle DAG depicted in Diagram 4. That is to say, they are sent along with the payload of the regular PTN, when each participant signs the payload of its appropriate neighbor along the chain of hashes.

Suppose Mallory generates a transaction *T* to transfer ownership of resource *X* to Alice. *X* contains a history of transactions, including when it was first issued, the list of participants that agreed to serve as notaries for *X*, and the "latest" transaction where *X* was assigned to Mallory.

Alice then queries several notaries listed in the resource *X*, and waits. All the notaries are supposed to check what they see as the latest transaction for *X*. If this latest transaction does not include the Mallory as the latest Recipient, then the notary generates a claim of a double-spend. It includes the transaction *T*, which was signed by Mallory, and the older transaction signed by the Mallory where they originally gave away *X*. This puts Mallory "on the hook", caught "red-handed" with two transactions constituting a double-spend, both signed by her.

A bona-fide proof of the double-spend would require the notary to store *all* transactions for *X*, including the transaction signed by Mallory where she originally gave away *X*, and all transactions since. One can compare the timestamps of the two transactions relative to the Issuer's Merkle Stream, because the Issuer has been generating these timestamps as described in Section 4b: Merkle Skip Lists. The Issuer can't collude with Mallory by fiddling with their timestamps because the PTN aspect is designed to be cryptographically tamper-proof.

Thus, in the PLN, each notary for resource *X* has to:

- Participate in the PTN protocol, so as to get accurate RTIPs
- Store all new valid transactions it comes across for resource *X*
- Report any double-spends to a user who claims to have received *X* in a transaction

Although it doesn't take much computing power to carry out these tasks, the notary may want to be incentivized for this. After all, if all notaries of *X* refuse to report double-spends, they go unreported (See 8.2 Colluding Notaries). Similarly, if all miners of bitcoin stopped or were DDOSed (See 10. Availability) then the network would go down.

Thus, the notaries must gossip the ledger among themselves, accepting every valid transaction they find. There is no need for consensus about the ordering, because every transaction already has an RTIP and valid transactions form a Hash Chain that can be quickly verified. Once again, the notaries must police one another to make sure they all follow the above protocol. If one notary reports that a double-spend exists, and some others report it does not, this constitutes a proof that the others are Misbehaving (section 8.1) .

7b. Tokens

Tokens can split, making them arbitrarily divisible. Thus, Alice can split a token *X* which she owns into several parts, e.g. *Y* and *Z* comprising 30% and 70%, respectively. Then, she can sign over ownership of *Y* to Bob, thus paying him 30% of the value of *X*.

Tokens can also merge, in order to send simplify sending many small amounts in one transaction. When tokens *X* and *Y* merge into *Z*, this gets added to the history on all the ledgers.

As an aside: this history can also be useful for calculating things like how much on average was spent on food, or track money laundering. While the Permissionless Transactions and Security features do not allow seizing or freezing funds in anyone's account, such a feature can be added on top of the PLN protocol as an extension, if that's what the community desires to do. For example, a cryptographic proof that "authority X has blacklisted this money" could be accepted by notaries and furnished as a warning to anyone who accepts the token from then on. In this way, communities could combat money laundering by marking some tokens and inhibiting their circulation (whether split or combined with others), while other money in people's accounts would continue to circulate. Perhaps this "money laundering" tag would persist even as Community Coins are exchanged for Intercoin, which is then exchanged into other Community Coins, and so on.

8. Attacks on the system

8.1 Misbehaving Issuers or Notaries in the PTN

In the PTN protocol, each participant simply executes a loop timestamping its neighbors one by one in turn. Since each payload is also signed, any breaking of protocol by a participant can result in losing access to the network.

The PTN is designed to ensure that no participant can refuse to timestamp a transaction (which is different from Bitcoin, etc.) because they only have a choice of timestamping their neighbors or not – and if not, their neighbors won't timestamp them either, so they'll lose access to the network.

The growing hash chains are built up by each notary as the messages are passed along between the Owner and the Issuer through the notaries. Due to the PTN protocol, if *one* notary accepts the a transaction, they all do. A notary can't prevent the timestamping of a transaction on *X* when it timestamps its neighbor's hash: it's all or nothing.

8.2 Colluding Notaries in the PLN

Because the Owner of a resource *X* is ultimately required to sign any transfer of *X*, no one can steal the resource from the Owner, or during the transaction to the Recipient.

Likewise, it takes only *one* notary to furnish proof of a double-spend involving transactions where the Owner signed away *X* to someone else, for the Recipient to reject the transaction.

However, if *all* notaries for *X* conveniently "forgot" or never recorded a transaction involving Mallory signing *X* over to Alice, then Mallory could then sign over *X* to Bob, and the double-spend would never be discovered, because Bob didn't think to ask Alice (who would have had proof of the double-spend). In order for this situation to arise, Mallory must collude with *every* notary.

Thus, every recipient including Alice should require *X* to have a *minimum* number of notaries that they think won't collude with Mallory. However, then a group of notaries could theoretically collude with one another to blacklist certain tokens or transactions by refusing to update their Ledgers for that token or owner. If this happens, then no one would accept the Owner's payments anymore, because the minimum number of notaries would no longer be updating their ledgers about *X*.

This is why each token *X* should be watched by a large enough number of notaries that even if many of them collude (or are blown up like in the movie Fight Club), the number of remaining notaries is still large enough that recipients accept transactions.

At the end of the day, however, finding a double-spend is reduced to a search and incentive problem. The incentives of the notaries have to be strongly aligned with being active members of the network, recording transactions and reporting double-spending when it is discovered.

8.3 Possible Blockchain per Token

In some sense, the scheme described is similar to one blockchain per token. Transaction *T* involving token *X*, signed by its Owner, is submitted by the Recipient to the PTN, and results in a Hash Chain about *T* which the Recipient stores as proof of the transaction. Valid transactions involving *X* are stored by each Notary together with a Hash Chain of transactions. No Consensus is involved, however, and it takes only one honest "defector" to make progress in the system. Given enough notaries, if no one defects and reports a double-spend, reasonable Recipients would accept the transaction.

An approximation to this could be obtained by taking the Ripple Consensus Algorithm, where validators agree to take all the valid transactions they see, and punish each other if they don't. But, unlike having a global blockchain for all tokens, each Blockchain would only be for a token *X*. In effect, this represents decentralization and sharding. In this scheme, the notaries do in fact sign off on "having validated" the new transaction on *X*, and combine the signatures in a Blockchain, which is a Merkle Tree. The Owner of the token can then prove having received it by storing a branch of the Merkle Tree, that fed into that Block, and the notaries would only need to store the "trunk" of the Merkle Tree, which grows as the token circulates.

In either of these systems, there is no ever-growing Blockchain of all transactions that everyone needs to keep. The logical global Blockchain is sharded into one Blockchain per token. Moreover, the list of notaries is encoded in *X*, and it's in the Owner's interest to keep *X*. Any subsequent transactions involving *X* are signed by the owner and the notaries do not even need to keep those transactions, but only the latest transaction – or to keep them even more honest, they have to run a Blockchain and keep only the trunk of its Merkle Tree. The Owners and Recipients store the rest of the hashes from its Merkle Branches and throw them away after they are no longer needed.

When tokens are redeemed by the Community Payment Network (e.g. for Intercoin), they can be destroyed, thus freeing up the notaries from even having to store the ever-growing trunk of the Merkle Tree – although it doesn't take that much memory to store, it's nice to know that even this can eventually be thrown away, making the entire system bounded in memory.

8.4 Collusion between partners in a transaction. In order to accept ownership of *X* and finalize the transaction (like endorsing a check), the recipient of *X* must satisfy themselves that the transaction was valid. This is a far easier task than verifying every transaction in a global ledger, and can be done quickly. However, any recipient who skips this validation runs the risk of being duped: their transactions with *X* will not be accepted by the subsequent recipients. Presumably, X was double-spent or simply a bogus token in the first place. In any case, an invalid transaction will result in a Merkle Stream that isn't accepted by future recipients who *do* validation, and thus recipient who doesn't validate the transactions will only be hurting themselves.

9. Incentives

If any participant in the PTN receives proof that its neighbor has improperly signed a payload and generated a hash, they are supposed to stop listening to that participant's hashes immediately.

What if a participant and its neighbors collude to keep the participant in the network? Other nodes must kick them out by association also, by checking hashes at least a few routing levels back. Like Ripple, correctness is required for membership in the network. By shielding their misbehaving neighbor, the neighbors run the risk of getting kicked out as well, so they have a strong incentive to not shield their neighbor.

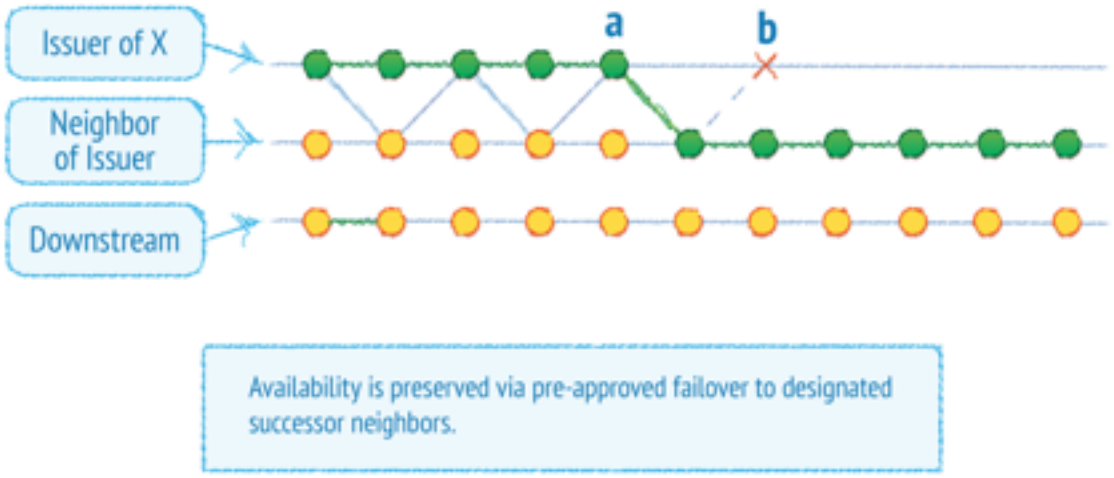
Of course, if the majority of the participants in the network decide to fork the protocol and do something else, then no one can stop them. We could attempt to make it more valuable to stay in the protocol than to fork it, but at the end of the day, crypto-currencies including Bitcoin can and have been forked. (There may be some improvement here, since we are not dealing with global consensus. The owners many of the resources *X* may refuse to cooperate with the new fork, and thus the forked network may lose / disrupt lots of owners. The difficulty of transition alone may be a deterrent for forking. More research needs to be done in another paper on this.)

Just like with Ripple, to prevent SPAM each participant might be required to hold a minimum amount of tokens on reserve, and spend them (the network may destroy them) for each transaction they submit to the PLN. However, by design, the amount of resources used by the PTN is very minimal, and the PLN is almost as small except that each ledger has to keep the latest transaction in memory, and forward messages with growing hash chains.

10. Availability

Although this kind of setup eliminates the need to trust any of the participants, it is still vulnerable to availability issues. Suppose the Issuer of a resource becomes slow with its timestamping, delaying operations with X. Or, perhaps it simply stops responding to its neighbors, or is progressively kicked out of the network as other participants learn of its treachery.

It is assumed that it's expensive for the Issuer to lose access to the network, so loss of availability won't happen often. However, each token *X* can have multiple fallback Issuers. If the primary Issuer goes down, the RTIP can be calculated relative to one of its neighbors. In general, the RTIP would be the minimum of the RTIPs relative to the fallback participants, in the order they were listed.



However, since discovering double-spends for each token relies on a certain number of Notaries to keep track of it, if enough of them are taken out of commission, or the signal to them is disrupted, then Recipients may stop accepting the tokens, because they are worried that the rest of the Notaries are colluding with the Owner.

The PTN concept can be adapted to many more applications than money transfer, including the Internet of Streams.