Ledger Loops: Using cryptographically triggered IOUs to make the world go round.

Michiel B. de Jong

November 2016

Abstract

LedgerLoops is a novel financial technology. Rather than a cryptocurrency, it is better viewed as a decentralized protocol for ledger federation. At its core is the concept of *cryptographically triggered IOUs*.

In this whitepaper, the Whispering Merchants problem serves as a simplified abstraction of economic exchange in general, where assets flow in loops. Traditionally, money would flow in the opposite direction, thus solving the coordination problem by effectively reducing the loop to a chain of one-to-one transactions. But money has a number of downsides, mainly due to the need to trust people other than your direct economical neighbors. Value fluctuations in a currency act as a single point of failure, which can have devastating effects on economic trade. LedgerLoops aims to allow assets to move around, while every trader only needs to trusts their own direct peers.

In the Whispering Merchants problem, merchants sit in a circle, facing outward. Each merchant can only communicate with their two direct neighbors. Each merchant wants to sell an asset to their left neighbor, and buy an asset from their right neighbor. Each merchant only trusts their immediate neighbors, and they trust no one else (thus excluding fiat money as an option), nor are they interested in any other asset than the item offered by their right neighbor (thus excluding commodity money). The problem exists in achieving cooperation between the merchants so that all assets move one merchant to the left.

LedgerLoops is a novel solution to the Whispering Merchants problem, based on cryptographically triggering each local IOU, rather than moving some sort of currency around the circle of merchants.

1 Cryptographically Triggered IOUs

An IOU (short for "I owe you") is a message or document representing a debt. A cryptographically triggered IOU contains the same data as a standard IOU (debtor, beneficiary, asset owed, etc.), plus a cryptographic challenge.

The IOU is without value until the beneficiary presents a valid solution to the challenge. In the current version of the LedgerLoops protocol, the cryptographic challenge is defined by a public key and a cleartext; the challenge is simply to sign the cleartext. Only the person holding the corresponding private key will be able to do this, yet everybody can easily verify and agree if a solution to the challenge is valid or not.

The Whispering Merchants can use cryptographically triggered IOUs as follows to facilitate their trade. One merchant creates a cryptographic challenge. The identity of this initiator stays obfuscated. Remember all merchants are willing to give their asset to their left neighbor, but only if they can be sure that their own right neighbor will do the same.

The initiator gives a cryptographically triggered IOU to his left neighbor, for the item he wants to offer. Each next left neighbor does the same. The details of the challenge are reused each time, but the item offered is different in each locally created IOU.

Once the initiator receives a cryptographically triggered IOU from his own right neighbor, for the item he wants to receive, he solves the challenge and presents the solution to his right neighbor. Each next right neighbor now gives they asset they offered in exchange to the solution to the challenge.

Since the challenge used in all cryptographically triggered IOUs was the same one for all trades along the loop, the solution to this challenge acts as a cryptographic coin, which has a well-defined value for each participant, but which loses its value as soon as the loop of trades has been settled.

2 The LedgerLoops Protocol (version 0.5)

In the last few weeks, I have implemented a few design iterations of the LedgerLoops protocol. Future versions are likely to differ from this, but let me describe roughly how it currently works.

2.1 Peer-to-peer Ledgers

As Graeber noted in his book "Debt: the first 5,000 years", peer-to-peer debt is useful for time-skewed barter between two peers; I give you something now, and you give me something in exchange tomorrow; in the meantime, a debt exists. This is why LedgerLoops is based on peer-to-peer ledgers (that is, ledgers on which two people keep track of what they owe each other).

Before finding a loop along which assets can move around, the first step of the algorithm consists of agents building up a ledger with each of their neighbors in the graph, by sending each other standard, unconditional IOUs, of the following format:

```
{
  protocol: 'local-ledgers-0.1',
```

```
  msgType: 'initiate-update',
  transactionId: <a unique string>,
  note: <a description of why this IOU is being sent, for instance
      "you paid for dinner yesterday">,
  debtor: <identifier of the sender; this can be just a first name,
      as long as it's different from the first name of the
      recipient>,
  addedDebts: {
    [currency]: amount, // for instance 'USD': 10,
  },
}
```

Even if the communication network is reliable and would detect network outages without losing any messages, a malfunction in the receiver's software could make it look like this message was received successfully when in fact it was not, so to confirm receipt, the recipient replies with:

```
{
  protocol: 'local-ledgers-0.1',
  msgType: 'confirm-update',
  transactionId: <a unique string>,
}
```

So in the current implementation, the exchange of assets is done asynchronously, creating debts on local ledgers, which then get fed into the cycle detection and settlement algorithm. Each participant will be happy if they can exchange one of their incoming debts against one of their outgoing debt, since the balance of each debt will go down, thereby minimizing their exposure to credit risk and enabling new trades to take place.

2.2 Routing

Finding cycles in a graph of which no one has a bird's eye view is an interesting computer science problem. Recently, 'Decentralized Cycle Detection' was included in a non-free IEEE publication. Unfortunately, due to the scientific journal's policy, no public version of the algorithm is available yet, but I can try to explain it in my own words:

* each leaf node (one that has either only in-neighbors or only out-neighbors), goes into 'deactivated' mode, and initiates a bread-first wave of messages through the network; from the point of view of each node, these messages mean as much as "I am sure I'm not part of a cycle". * each internal node will go into 'deactivated' mode and forward the wave of messages to their out-neighbors as soon as they have received a message from each of their in-neighbors, and vice versa. * nodes that have many in- and out-neighbors, but have at least one in-neighbor and at least one out-neighbor left from whom they have not yet received a message, can know that they

are either on a cycle, or on a path from one cycle to another. At least, they can now discard all neighbors from whom they receive such a message, and drastically reduce their search space if they want to look for cycles.

I adapted this algorithm slightly to make it work in graphs whose topology can change while the algorithm is running. At first, if a node has not sent or received any "confirm-update" messages yet, it has no neighbors in the debt graph, and is therefore created in deactivated mode.

* When a link is added, the nodes connected by the link both send out a breadth-first wave of messages meaning "I might be part of a cycle". * These messages travel to the leafs of the network, where they bounce back and become reply messages meaning "No cycles here". * Like in the original DCD algorithm, a node deactivates if it has received a "No" from either all out-neighbors or all in-neighbors, and then sends the "No cycles here" message on forward in its direction of travel. * Nodes also deactivate as necessary when links disappear from the network.

As a special remedy against race conditions, when an internal node has already been activated, and then gets another positive "I might be part of a cycle" from a neighbor, it does not forward that message (since its state does not change, so there's no news to report). This means that this second message dies at this internal node, and does not get a chance to bounce against the leaf nodes at the edge of the network. The naive solution for this would be to wait for the first message to come back (either bouncing against leaves, or cycling around), but if a cycle is created by adding a lot of new links at almost the same time, a race condition can occure where both messages get suppressed along the way. Therefore, the internal node that suppresses the second message gives back a positive reply in this special case.

The current version of the LedgerLoops protocol is still a very simplified proof-of-concept, and participants in a loop are not yet allowed to negotiate exchange rates for assets; they need to indicate the value of their asset in terms of a unit of value, which is now (confusingly, I admit) called currency. All trades in a Ledger Loop still need to have exactly the same value, expressed in the same unit of value.

A node can send its "I'm not on a cycle" message in error or while lying on purpose - when this happens, this messages more precisely means "I'm not on a cycle of nodes that cooperate correctly".

```
{
  protocol: 'ddcd-dfs-0.1',
  msgType: 'update-status',
  currency: <used as a unit of value here; for instance 'USD'>,
  value: <true if the sending node might be on a cycle, false if it
      definitely isn't>
  isReply: <true indicates this message is a reply to an earlier
      message>,
```

```
}
```

Once the search space has been reduced using this 'Dynamic DCD' algorithm, depth-first-search (DFS) is used to find actual cycles. To start a DFS tree, generate a treeToken and a pathToken, two strings that are long enough to be hard to guess. Send the following message to one of your out-neighbors:

```
{
  protocol: 'ddcd-dfs-0.1',
  msgType: 'probe',
  currency: <used as a unit of value here; for instance 'USD'>,
  treeToken: <a long string>,
  pathToken: <a long string>,
}
```

Each node forwards this message to one of their out-neighbors, until it reaches a leaf node. It is then backtracked to the last internal node which has more than one out-neighbor. This node generates a new pathToken but keeps the treeToken as it is, and sends the altered message to their next out-neighbor, until they also need to backtrack.

The pathTokens which are included in addition to treeTokens are not really used yet; each node can remember what the last out-neighbor was they sent this treeToken to. However, a node may decide that its out-neighbor is unreachable, and after some timeout continue trying with the next sibling. Once this happens, it would be possible to find multiple cycles on one treeToken (but they would then have different pathTokens).

If the probe reaches a node which it has already visited, this node intends to become the initiator of a LedgerLoop.

2.3 Cryptographically triggered IOUs

The initiator generates a keypair and a random string to be signed. It then sends the following message to the out-neighbor to which it sent the treeToken that looped around:

```
{
  protocol: 'ledgerloops-0.5',
  msgType: 'conditional-promise',
  transaction: {
    id: <a newly generated unique string>,
    currency: <the currency from the probe>,
    amount: <a small amount, for instance 0.01 USD, makes success
        more likely>,
  },
  challenge: {
    name: 'ECDSA',
```

```
    namedCurve: 'P-256',
    pubkey: <base64 representation of public key in spki format>,
    cleartext: <base64 representation of a random array of bytes>,
  },
  routing: {
    protocol: 'ddcd-dfs-0.1',
    treeToken: <the treeToken for which a loop was detected>,
    pathToken: <the pathToken that looped back (so not the one that
        was sent forward initially>,
  },
}
```

Once this message reaches the initiator, the solution to its challenge is sent round in the opposite direction:

```
{
  protocol: 'ledgerloops-0.5',
  msgType: 'satisfy-condition',
  transactionId: <the transaction id from the conditional-promise
      message>,
  solution: <the base64 representation of a valid signature for the
      challenge from the conditional-promise message>,
  routing: {
    protocol: 'ddcd-dfs-0.1',
    treeToken: <the treeToken for which a loop was detected>,
    pathToken: <the pathToken that looped back (so not the one that
        was sent forward initially>,
  },
}
```

The routing information could be left out in this message type, but is in version 0.5 for practical reasons.

After the receiver of a satisfy-condition message has verified its signature, it replies with the "confirm-update" message format earlier (using the transactionId from the "satisfy-condition" message, and once all nodes along the loop have done this, the trade is complete.

If a node decides it has been waiting for too long, it can request a revokation of the cryptographically triggered IOU they sent earlier, by sending this messages behind it:

```
{
  protocol: 'ledgerloops-0.5',
  msgType: 'please-reject',
  transactionId: <the transaction id from the conditional-promise
      message>,
}
```

A node should probably wait a few milliseconds before forwarding a

cryptographically triggered IOU, in case the IOU has been buffered in a slow place for a while, and now has a "please-reject" message following directly behind it.

To reject the cryptographically-triggered IOU you received earlier, use this message:

```
{
  protocol: 'ledgerloops-0.5',
  msgType: 'reject',
  transactionId: <the transaction id from the conditional-promise
      message>,
}
```

The rejection will only be valid if it is sent by the recipient of the conditional-promise message in question.

3 Security Discussion

4 Conclusion

The ledger loop length can be quite short (three or four participants), but none of the participants know this precisely. Participants can roughly tell that a loop where the rounds take a long time is more likely to have more participants than one on which message loop around quickly, but this information is obfuscated by differences in speed of the communication network, and by participants who choose to wait a little bit before forwarding a messages.

No central authority or Unique Node List was used, each participant only ever trusted their own direct peers, and none of the participants exposed the contents of their private ledgers to the rest of the network.

Also, nobody except the initiator knows who the public key belonged to that triggered all LedgerLoops contracts, and each peer only exposed their identity to their direct neighbors in the ledger loop. Unless participants volunteer to make this public, nobody (except for each participant's direct neighbors) knows exactly who particpated in which ledger loop. The initiator can also not know all the trades that were triggered by their challenge/solution pair.

This is a work in progress. Unresolved issues are documented here: https://github.com/michielbdejong/ledgerloops/issues