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 crypto-currency, 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

3.1 Assessment Scope

3.1.1 At the participant level

A participant in the LedgerLoops protocol is a human person or legal organization (the ”user”) who uses a computer on which LedgerLoops software (the ”app”) runs. This computer is potentially also used for other purposes. The user supplies contact details about other users (the user’s ”neighbors”) to the app. This list of neighbors (containing contact details as well as personal notes such as nick names and profile photos) is sensitive information.

Once the user creates ledgers with their neighbors, the information in these ledgers (current balance as well as transaction history, and personal notes added to each transaction, which are likely to reflect evidence of real-world events) is the second item of sensitive information.

Each ledger is also personal between the user and the neighbor involved in that particular ledger; information from one ledger should not be leaked to one of the other ledgers, or otherwise become known to one of the user’s other neighbors.

When the app starts trying to route and resolve loops, this activity affects other software on the same computer, by using network, CPU and storage resources. The app should also not leak or damage sensitive information pertaining to other software on the same computer.

If incorrect information were to enter the user’s ledgers, the user’s real-world assets could be at risk. In this case, as well as when the app sends low-quality messages to neighbors, the user’s reputation with their neighbors (debtors and creditors), and possibly the user’s public reputation. An example of a low-quality message could be one that is malformed, or one that represents incorrect or unfounded information, like when the app were to continuously bother neighbors with messages that convey a statement of intent, convincing the neighbors to invest effort, but never or rarely leading to a trade deal, thus wasting other participants' time.

To summarize, the user would be negatively affected if:

- any of the following information is leaked, either to one of the user's other neighbors, or publicly:
  - identity of the user's neighbors
  - information contained in notes about the user's neighbors
  - ledgers (current balance as well as transaction history)
  - information contained in notes on the ledgers
  - other information, unrelated to the app but present on the computer
- the app would make excessive use of any of the following computer resources:
  - network bandwidth
  - CPU resources
  - storage resources
- the app's copy of the following information would be damaged (it may trigger real-world events, or be the only copy of this data):
  - ledger balance and transaction history
  - other information, unrelated to the app but present on the computer
- the app's incorrect or unreasonable behavior damages the user's reputation with:
  - debtor neighbors
  - creditor neighbors
  - third parties or the general public

3.1.2 At the network region level

Once a number of users rely on the network beyond their own neighbors to find and resolve ledger loops, these users would be negatively affected if this network clogs up or breaks down, affecting participants who are not neighbors of the user, but who would be candidates to participate in a ledger loop in which the user and their neighbors would also participate.

Even if the valid participants of the network are unaffected, an excess number of useless participants would make the valid participants hard to find.

Likewise, useless network traffic could slow down the processing of valid network traffic.

The network as a whole is decentralized, and several independent regions (connected components in the network graph) could exist. Low-quality areas in the network will mostly affect users at a short network distance from the low-quality region. Therefore, in terms of valuable assets, we should consider network regions, more than the global network as a whole. We can write these assets at the network region level down as:

- average quality of participants at distance d from a user (d¿=2)
- average quality of messages received from a user's neighbor, but triggered by messages at distance d¿=2.

3.2 System Modeling

As already defined a bit in the previous subsection, a user uses the app running on a computer. The app sends LedgerLoops messages, only to direct neighbors of the users, although these messages might cause ripple effects, triggering messages to other participants at distance d¿=2. The messages sent between neighboring users travel over a secure communication channel (the sending and receiving client software for this communication channel is not part of the app). Neighbors of a user can be divided into two categories: debtor (owes or offers something to the user), and creditors (to whom the user owes or offers something). Incentives for cooperation can be different for debtors than for creditors. For instance, if a creditor misbehaves, the user can block them (ignore messages from them) and tell the neighbor user in question through a side channel to change their app's behavior if they still want to access their credit. If the creditor does not comply, the user can confiscate their credit. However, if a debtor misbehaves, the user cannot so easily block their messages, as this would effectively allow the debtor to never pay back their debt - or at least not pay it back via LedgerLoops.

3.3 Threats

3.3.1 The user's own computer and LedgerLoops app

While the user's app is running on the computer, it needs to access all information and resources mentioned in the assessment scope above. Therefore, a malfunctioning app would put all these assets at risk. When the user's computer is running, even if the app is not, there is probably no way to protect against potential malware on the computer. Even if the app stores its data in encrypted format, the user would have to unlock this data when starting to use the app, and any passphrase or secret used to do this while starting up the app could be intercepted by such malware. The computer as a whole could, however, use disk encryption so that if an attacker gains physical access to the computer, but can not force the user to enter their passphrase, the data stored on the computer would be safe. In short, if the user's computer is compromised, the attacker could access the user's information, damage the user's reputation, damage the user's real-world assets, and the user's reputation. A malfunction (bug) in the app could also potentially put all of these assets at risk.

The following threat descriptions all assume the user's computer is clean and the app functions correctly.

3.3.2 Social Engineering

A user could be convinced by an attacker to add a neighbor with whom they don't have a real-world trust relationship. Adding a malicious debtorcould expose information about the user's creditors. For instance, if an attacker wants to know if the user frequents a certain restaurant, they could buy a meal voucher from that restaurant, and at the same time offer something attractive to the user. If the user accepts the offer from the attacker, then the next time the user eats at that restaurant, the user's app could cooperate with the restaurant's app and the attacker's app to find a loop. If such a loop is found, then the attacker knows that either the user just visited the restaurant, or the user owes something to a participant, (who owes something to a participant, who ...) owes something to the restaurant.

Adding a malicious creditor could likewise expose information about the user's debtors.

3.3.3 Network Area Attacks

Nobody except a user's own neighbors is allowed to send the user messages directly, so standard Spam and Denial-of-Service attacks will not work to degrade a user's LedgerLoops app (it might still work to affect the user's computer or the user's internet connection, of course).

However, apart from the attacker becoming a neighbor of the victim (which should be quite hard if the user is careful), the attacker could try to control many nodes in the network area surrounding the user (so nodes that are two or three hops away from the victim). This would allow the attacker to degrade the quality of the user's network area, for instance by injecting a lot of useless messages (low-quality traffic), and by making nodes look attractive for cooperation but never reaching agreement.

3.3.4 Traffic Analysis

Even if all of the user's neighbors (and their computers and their apps) are clean, an attacker in the same network area as the user could obtain information about the user by analysing their own ledger loops. If the attacker has several two-hop connections with the user, they could potentially analyse the messages they receive, and send their own messages to test hypotheses, to gather metadata about the user. They will not be able to know which of the ledger loops the user took part in, and it could be that what they are modeling as the targeted user is in fact two or three different users, but by playing it smart, they could potentially create an ever more detailed model of the targeted user.

3.3.5 Figure-8 Attacks

An attacker could aim to control two or more participants in the same ledger loops. However, if a ledger loop goes through participants 0, 1, 2, . . . , 9, 0 and the attacker controls nodes 3 and 7, we can write this as "0, 1, 2, (3/7), 4, 5, 6, (3/7), 8, 9, 0". This is equivalent to two separate ledger loops: "0, 1, 2, (3/7), 8, 9, 0" and "(3/7), 4, 5, 6, (3/7)", where the $3 \to 7$ traffic from the first loop cancels out the $7 \to 3$ traffic from the second one. A threat where the attacker controls two nodes in a ledger loop can therefore be analyzed as two separate attacks.

3.3.6 Timing Analysis

One of the most worrisome threats may be timing analysis: in principle, a participant in a ledger loop knows that the ledger loop contains at least three participants, but does not know an upper bound of the number of participants. However, knowledge about network link speeds can help reveal such an upper bound. For instance, if the attacker knows that each network hops takes one second, and that a message was passed around the loop in four seconds, then they know there are no more than four participants in the loop. It does not work in the other direction: if a message takes 1000 seconds to go around the loop, this might be because one of the participants stalled for 996 seconds before forwarding the message. However, if all participants are known to always stall one second, then a roundtrip time of 8 seconds can also give evidence of a length-four loop. Randomizing the stall time helps a little bit to obfuscate loop length, but the attacker can average out this randomization by analyzing a large number of ledger loops together. Also, if all participants happen to randomly pick a low stall time, then a message could still travel around that loop in under 5 seconds. The most dangerous form of this threat is that where the attacker discovers with reasonable certainty that they are participating in a length-three loop, since this tells the attacker that their own debtor is also a neighbor of their own creditor.

3.3.7 Lying

LedgerLoops is a tool with which a user keeps track of debts and credit between themselves and their direct neighbors. The app has no leverage on the real world, other than through the user's own actions. This is why the neighbor always relies on the real-world trustworthiness of their own neighbors. For instance, an attacker who owes the user a significant asset could simply tell the user that their computer was compromised, and they no longer have a record of this debt. Even if the communication channel used cryptographic signatures, the attacker could deny having signed that message, stating that in fact some malware on their computer must have caused this message to be sent.

3.4 Mitigation

Make sure your computer is clean and use a peer-reviewed LedgerLoops app. Regularly make backups of ledger data, and regularly check if all ledgers still look as expected. Block neighbors who send low-quality messages, and contact them through other channels before removing the block.

...

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