CREDIT NETWORK IMPLEMENTATION IN ETHEREUM

A Document explaining the design rationale, notations and the implementation of Ripple-like Credit Network in Ethereum.

2018

Last edited on November 15, 2018

Contents

1 Introduction

A credit network (as shown in Figure 1) is a directed graph $\mathbb{G}(\mathbb{E}, \mathbb{V})$ with users $u \in \mathbb{V}$ as nodes and trust between them as edge $e \in \mathbb{E}$. A directed edge from u to v represents the trust that v has on u. The edge has a upper limit which quantifies the trust that v has on u. The edge has a currency associated with the credit. This is analogous to a credit line. Credit lines have a credit limit and a user can utilize credit up to the credit limit.

The document is organized as follows: Section 2 describes the notations used in the implementation.

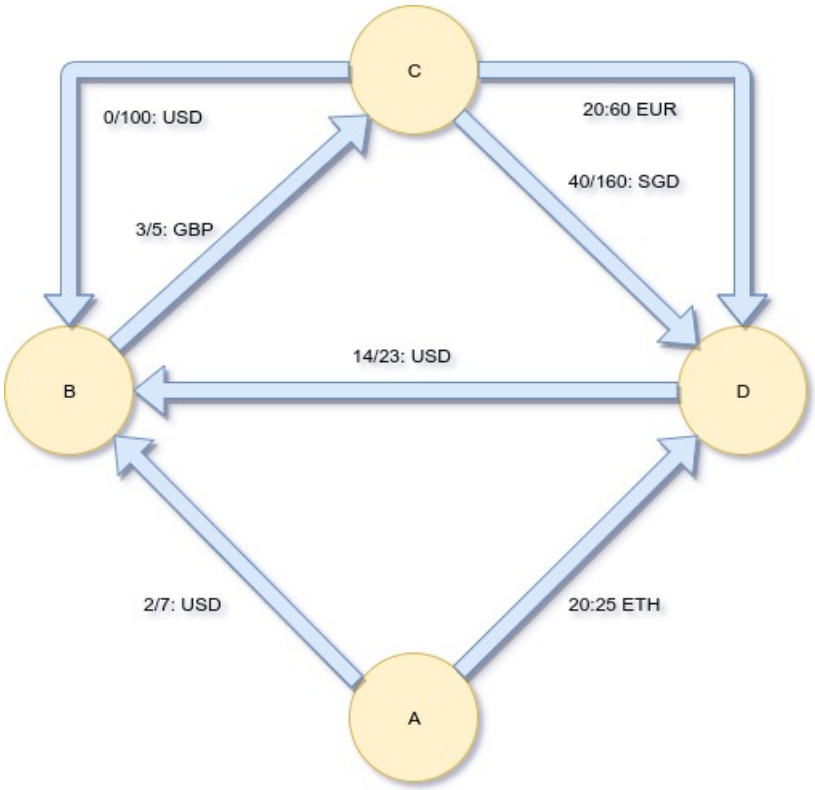


Figure 1: An example credit network graph: The edge from A to B shows that A owes B money.

2 Notations

2.1 Data Structures

A user u node has the following attributes:

1. address - a unique node identifier

A link is a directed edge $e \in \mathbb{E}$ of the graph $\mathbb{G}$ with the following attributes:

1. $from$: address
2. to: address
3. $currentVal$: Current Value / Balance
4. $UpperLimit$: Upper limit for the link
5. $ripplingEnabledFromTo$, $ripplingEnabledToFrom$: rippling flags for both users
6. $feesFrom$, $feesTo$: Fee in both directions
7. $currencyID$: currency of the link

   A link e: $u \rightarrow v$, where u,$v \in \mathbb{V}$ represents that u owes $e.currentVal$ IOUs of $e.currencyID$ to v. u, v are $e.from$, $e.to$ respectively.

2.2 Functions

2.2.1 Node Registration

addNode()

- A node u in the credit network graph G, is represented uniquely by the 20 byte Ethereum address.
- A user who wishes to register with the credit network is assumed to make a call himself to this function from a suitable client.

- A user who wants to register with our service should have a Ethereum account by running a wallet or any other similar program. This is a pre-requisite as our service runs on top of it. So, the user should have an address, public key and signing key which are automatically issued to all the ethereum users.
- All the registered nodes are stored in a HashMap $addressMap$. The hash map is a mapping from the Ethereum Address to the Node Data Structure.

2.2.2 Link Operations

createLink(currencyID, currencyID, from, upperlimit, rippling flag, fees):

- A link between u and v in the credit network is established as follows:
  1. The nodes u and v are checked for registration in the credit network.
  2. Previous Links between u and v are checked. If it already exists, abort.
  3. Initialize the link data structure. Rippling Flags are disabled by default. Fees are set to zero by default.
  4. Add the link data structure to the mapping $hashLinkMap$.
- The mapping $hashLinkMap$ is a mapping from the 32 byte SHA3(keccak256) hash to the link data structure.
- The hash is generated by hashing the $currencyID$, $from$ and to fields of the link.
- A $\xrightarrow{5/50}$ B
  A owes 5 IOUs to B.
  Max amount A can owe to B is 50 IOUs.
  B calls $createLink(currencyID, A, 50)$.
- $from$ address of the link is set to the from argument passed to the function.
- Address of the caller of the function is to address. Caller is the one who will setup the link. Direction of the link is towards the caller.

- The initial value of the balance of the link will be 0.
- As one of the user is initializing the link, the caller is allowed to set the rippling flag and fee for the link.

updateLink(currencyID, other, ulim, rippling, fees):

- This function is used to update the various parameters of the link between the caller and *other*.
- If the caller is the $from$ address, the $from$ rippling fields ($ripplingEnabledFromTo$) and fees($feesFromTo$) are updated and similarly if the caller is the to address.

2.2.3 Payment

CreditNetworkPay(toPay, value, paths (upto 4 paths)) returns feesUsed:

- $toPay = v$ is the address of the final recipient of the amount.
- $value$ is the amount to be sent.
- $paths$ is the list of the nodes u_1,u_2,..., u_n in the order of flow of $value$.
- The actual payment path is as follows:
  $Caller \rightarrow u_1 \rightarrow u_2 \rightarrow .. \rightarrow u_n \rightarrow v$
- The pay function converts the currencies in between (if any intermediate has created an $offer$(Section 2.2.4).)
- The following cases are ensured:
  - Every node $u \in G$.
  - For any i $(1 \leq i \leq n)$ and nodes u_i and u_{i+1} in the path, if $u_i \rightarrow u_{i+1}$, then $ripplingFromTo$ is true, else $ripplingToFrom$ is true.
  - For every node u_i, balance of the link from u_x + value + $\sum_{n=x+1}^{\infty} fee(u_n)$ < upper limit of that link.
  - value + total fee($\sum_{n=1}^{\infty} fee(u_n)$) <= allow_max
- If u wants to pay x IOUs to v and u, v are connected by a path of the form $u \rightarrow u_1 \rightarrow u_2 \rightarrow ... \rightarrow u_n \rightarrow v$:

- Edges are considered undirected to find a path from the sender u to the receiver v of the transaction.
- The path is provided by a path finding algorithm off chain.
- The credit value on every edge in the path from u to v is updated depending on the direction of the edge as follows:
  1. Edges in the direction from u to v are increased by x, while reverse edges are decreased by x.
  2. If there is a change of currency in the path, the payment succeeds only if the intermediate node provides a currency exchange offer.
  3. In summary, when u pays x to v, an equivalent of x in v's choice of currency.
  4. To compute how much u has to send in order for v to receive y is a problem to be solved off the chain.
- While updating the values on the links, check if they satisfy the upper limit constraint.
- 4 paths are provided for atomic payments along multiple paths.

- Example

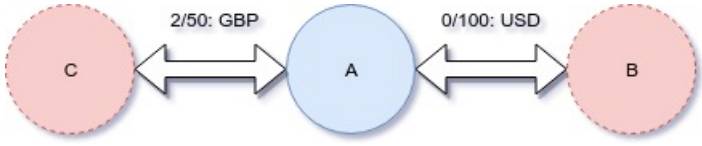
  $\text{A} \xrightarrow[\text{fee:}f_1]{x_1/t_1} \text{B} \xrightarrow[\text{fee:}f_2]{x_2/t_2} \text{C} \xrightarrow{x_3/t_3} \text{D}$

  Rippling flag of all the links is true.
  A wants to pay x to D.
  Values of the links after payment will be as follows.

  $\text{A} \xrightarrow{x_1 + x + f_1 + f_2/t_1} \text{B} \xrightarrow{x_2 + x + f_2/t_2} \text{C} \xrightarrow{x_3+\text{x}/t_3} \text{D}$

2.2.4 Offers

An offer is basically an exchange rate between edges of different currencies set up by the node. An offer is therefore associated with a node and it's two adjacent edges.



Consider a payment from C to B via A for an amount of 20 GBP in the setup shown in Fig 2.2.4. The exchange rate as set by A is 2 USD for every GBP. Therefore, 20 GBP reaches the link $C - A$ and it converts to 40 USD and proceeds to link $A - B$. The arrows are bi-directional to indicate that the direction of the links do not matter to set up an exchange offer. If a non integer rate is applied then the conversion rate is applied and floored. For example, if the exchange rate is 100 USD for 6534 INR then 1 USD only gives 65 INR.

In order to set up the order, the user must call the addOffer function in the contract. addOffer(inputLinkIdx, outputLinkIdx, inputAmount, outputAmount):

- *inputLinkIdx* is the index of the link of the source currency.
- *outputLinkIdx* is the index of the link of the destination currency.
- *inputAmount* is the amount of source input currency.
- *outputAmount* is the equivalent amount of source input currency in terms of the output currency.
- The caller of the function $u \in \mathbf{G(V)}$. Otherwise the function will fail.
- The function will also fail if the links do not exist.
- If the links do not differnt currency IDs the function will not create an exchange offer and will fail.
- Self links are not allowed. i.e $inputLinkIdx \neq outputLinkIdx$.
- Basic sanity cases are not allowed. Such as $inputAmount \neq 0$, $outputAmount \neq 0$, and $inputLinkIdx \leftrightarrow u \leftrightarrow outptuLinkIdx$ (u must be present as source/destination in both the links).

2.3 Fee for a Transaction

If u wants to pay x to v, and so u sends only x on the path, some of it is consumed as fees by intermediary nodes which results in v receiving lesser amount than anticipated. So, u may want to know how much extra he needs to send in for v to receive x. This can be achieved by dry-running CreditNetworkPay function using only one path. This will test if it is possible

to use the credit network to transfer value to receiver from the caller of the function through the given path. It also sums up the fees along the path and returns it. The successful execution of a transaction indicates that the fees don't exceed the given limit.

2.4 Finding Path

Whenever the state of the credit network graph changes, the contract emits a log. Logs provide a way for interested third parties to subscribe to the changes to the state of the graph and can be used to provide path finding and/or other services to the user. An interested user/service providers can use the logs to build a graph off chain and provide a path finding application.

3 Benchmarks

In order to quantify the performance of transitive credit network scheme, various measurements were made for all the external functions.

3.1 Gas Costs

The amount of gas used for every function call is recorded using the truffle framework and *ganache-cli* as the test network. The ETH/USD gas price is obtained using the *coinmarket* API.

In Ethereum, the miners choose the transactions to be mined based on the amount of gas provided to the miner. Therefore a user has to provide incentives to the miner in order to mine their transaction. *ethgasstation.info* provides the minimum gas price to be provided to the miner so that the transaction is mined. Similarly, it also provides the average and the highest gas price to be paid in order to decrease the time taken to mine the transaction. An important point to be noted here is that the more the gas price, the faster the transaction is mined.

The gas cost of the payment function is dependent on the path length and the number of currency conversions. They are evaluated on a number of inputs with varying hop lengths ($H_i \implies$ there are i intermediate nodes between the sender and the receiver) and different conversion times ($C_i \implies$ there are i currency conversions in the payment).

[1]USD/ETH rate as on November 1, 2018 is $199.6/ETH

[2]Minimum Gas Price accepted on the network is 1GWei/gas

[3]Average Gas Price of the transactions accepted in the network is 2GWei/gas

[4]Gas Price for the fastest acceptance of the transaction is 4GWei/gas

[5]H_{number} stands for the number of hops

[5]C_{number} stands for the number of currency conversions

Function	**Gas Used**	**Min Fee** [12]	**Avg Fee**[3]	**Max Fee**[4]
AddNode	43474	0.0087	0.0174	0.0347
CreateLink	117990	0.0236	0.0471	0.0942
UpdateLink	37263	0.0074	0.0149	0.0298
Pay[5](H_0C_0)	45586	0.0091	0.0182	0.0364
Pay(H_1C_0)	59438	0.0119	0.0237	0.0475
Pay(H_2C_0)	73226	0.0146	0.0292	0.0585
Pay(H_3C_0)	87014	0.0174	0.0347	0.0695
Pay(H_4C_0)	100803	0.0201	0.0402	0.0805
Pay(H_5C_0)	114593	0.0229	0.0457	0.0915
Pay(H_1C_1)	64150	0.0128	0.0256	0.0512
Pay(H_2C_1)	77939	0.0156	0.0311	0.0622
Pay(H_3C_1)	91728	0.0183	0.0366	0.0732
Pay(H_4C_1)	105517	0.0211	0.0421	0.0842
Pay(H_5C_1)	119307	0.0238	0.0476	0.0953
AddOffer	74015	0.0148	0.0295	0.0591
CancelOffer	51571	0.0103	0.0206	0.0412

Table 1: Gas Usage for the contract functions

A Implementation

A.1 Data Structures and State Variables

1. ```
   struct Node {
       address addr;
   }
   ```

2. ```
   struct Link {
       Node from;
       Node to;
       uint upperLimit;
       bool ripplingEnabledFromTo;
       bool ripplingEnabledToFrom;
       uint currentVal;
       uint feesFrom;
       uint feesTo;
   }
   ```

3. ```
   struct Offer {
       uint8 inputCurrencyID;
       uint32 inputAmount;
       uint8 outputCurrencyID;
       uint32 outputAmount;
       address provider;
   }
   ```

4. addressMap : A mapping from address to the Node data structure

5. hashLinkMap : A mapping from link index to the Link data structure

6. hashOfferMap: A mapping from offer index to the Offer data structure

A.2 Functions

An *external* function is a function that is available to users and can be called from any client/contract. An *internal* function is a function that is available only to the contract. A *view* function is a function that queries the state of the contract. These functions do not consume any gas. A function will fail if called with the incorrect parameters. A failed function call will not be mined
   ```

and therefore it is up to the user to use timeouts to detect failures and verify if the function call works using a dry run before sending out a transaction.

A.2.1 addNode

- parameters: None
- return values: None
- External function

A.2.2 createLink

Creates an edge e from the parameter u to the caller ($= v$).

- parameters: currencyID, address of u, upperLimit, rippling flag for caller, fees for the caller
- return values: None
- External function

A.2.3 updateLink

- parameters: index of the link (a 160 bit RIPE-MD hash), upper limit for the caller, rippling for the caller, fees for the caller
- return values: None
- External function

A.2.4 creditNetworkPay

- parameters: address of the person to pay, amount to pay, an array of addresses or [], an array of addresses or [], an array of addresses or [], an array of addresses or []
- return values: None
- External function

A.2.5 addOffer

- parameters: index of the input link, index of the output link, input amount, output amount
- return values: None
- External function

A.2.6 cancelOffer

- parameters: index of the offer
- return values: None
- External function

A.2.7 checkNode

- parameters: address of the node
- return values: bool (success/failure)
- View function

A.2.8 checkLink

- parameters: index of the link
- return values: bool (success/failure)
- View function

A.2.9 viewLink

- parameters: address of the link origin, address of the link destination, currency ID
- return values: Index of the link
- View function

A.2.10 linkInfo

- parameters: index of the link
- return values: address of the link origin, address of the link destination, current credit in the link, credit limit of the link, concatenated rippling flags, rippling fees for link origin, rippling fees for the link destination, currency ID
- View function

A.2.11 getOfferBy

- parameters: address of the exchange node, index of the input link, index of the output link
- return values: index of the offer
- View function

A.2.12 PathPay

- parameters: address of the payment initiator node, address of the destination node, amount to be paid, an array of intermediate nodes
- return values: Fees used for the payment
- Internal function

A.3 Contract Events

- NewNodeRegistration: Logs the address of the user who registered
- NewLinkSetup: Logs the following for the edge e: $u \to v$:
  1. address of node u
  2. address of node v
  3. currency ID
  4. rippling flag for v
  5. rippling fees for v

6. credit limit for e

- UpdateLink: Logs the following for an edge e:$u \rightarrow v$:
  1. index of the link
  2. caller $\in u, v$
  3. rippling flag for the caller
  4. rippling fees for the caller
  5. credit limit for the caller
- Pay: Logs the following for a payment operation:
  1. Address of the payment initiator node u
  2. Address of the payment receiving node v
  3. An array of intermediate nodes $u_1, u_2, .., u_n$ for the payment flow from $u, u_1, u_2, .., u_n, v$
  4. The amount of money paid in the initial currency
- NewOrder: Logs the following for a newly created exchange offer:
  1. Address of the node creating the exchange offer
  2. Index of the source link
  3. Input amount for the source link
  4. Index of the destination link
  5. Output amount for the destination link
- CancelOrder: Logs the index of the exchange offer that was cancelled.

B Gas Usage Notes

1. addNode Costs:
   (a) To call the empty parameter-less function that does not perform any operation it costs 21964 gas. This can be considered as the function setup cost.
   (b) The code to check if a node already exists takes 907 gas.

(c) Storing one node costs 20967 gas.

(d) Throwing an event log consumes 709 gas.

2. createLink Costs:

   (a) Empty Function call costs 23452 gas.

   (b) Checking the existence of both the nodes cost 827 gas.

   (c) Computing the hash index for the link costs 1762 gas.

   (d) Checking if the link already exists costs 379 gas.

   (e) Storing the Link costs 88638 gas.

   (f) Logging the event for createLink costs 1651 gas.

3. updateLink Costs:

   (a) Empty Function call costs 23547 gas.

   (b) Retrieving the link from storage costs 0 gas.

   (c) Testing for link origin costs 691 gas.

   (d) Setting the rippling flag 5641 costs gas.

   (e) Setting the fees costs 5323 gas.

   (f) Setting the link upper limit costs 5273 gas.

   (g) Logging the UpdateLink event costs 2245 gas.

   (h) It costs 42434 gas for the link origin and 42720 gas for the link terminal.

4. CreditNetworkPay Costs:

   (a) Empty Function call costs 26060 gas.

   (b) Checking if there exists at least one link costs 32 gas.

   (c) Setting the pre-loop values costs 505 gas.

   (d) One iteration of the loop costs 9122 gas.

   (e) Post loop validation (Check if the money reached the correct payer) costs 41 gas.

   (f) Logging the Pay event costs 2713 gas.

5. addOffer Costs:
    (a) Empty Function call costs 24788 gas.
    (b) Checking if the node exists costs 412 gas.
    (c) Checking if the amount arguments are sane costs 70 gas.
    (d) Checking if the input and the output links are not the same costs 50 gas.
    (e) Checking if the caller is intermediate between the two links costs 1992 gas.
    (f) Creating the offer index by hashing costs 1761 gas.
    (g) Storing the offer costs 42625 gas.
    (h) Logging the NewOrder event costs 2239 gas.
6. cancelOffer Costs:
    (a) Empty Function call costs 23073 gas.
    (b) Checking if a node exists costs 412 gas.
    (c) Checking the offer is being cancelled by the owner costs 428 gas.
    (d) deleting the offer from storage costs 11546 gas.
    (e) Logging the cancelOrder event costs 1082 gas.
7. Other Function Costs:
    (a) The other functions such as $getOfferBy$, $viewLink$ and $checkLink$ do not cost any gas as they are pure functions or view functions. These functions do not modify the state of the contract and hence are 'free'.