rLending White Paper v1.0

The rLending Dev Team
`support@rlending.app`

February 19, 2021

Abstract

We introduce an algorithmic distributed protocol that establishes autonomously regulated liquidity pools with interest rates derived from supply and demand. In this model, lenders earn revenue in exchange for their supplied assets and borrowers pay interests for the assets they used.

DEFI (Decentralized Finance) platforms are taking over the blockchain scene. They are predicated on the tenants of non-custodial, trust-less and secure systems with 100% up-time.

This means:

- No single entity stores funds on behalf of the users.
- Users are not required to wait long times for someone to process their transactions.
- Transactions are highly secure (thanks to open protocols and independent auditors).
- 24/7 accessibility from anywhere in the world.

In other words, there is no central bank or government authority imposing rules on how to operate. Instead, the platform was designed to be completely self-regulating. This is achieved by programming the protocol using distributed *Smart Contracts* deployed on the **RSK Network**[1]. Smart contracts handle the majority of logic that has historically been handled by human inputs, vastly optimizing transaction time and cost.

Introduction

Financial markets have had a difficult time keeping up with modern technologies and have stuck to a classic centralized model. In the last couple of years we have seen an increased awareness of the issues this presents. Demand has arisen for technologies that provide modern solutions [2] to issues such as facilitating investments and trade across blockchain platforms.

Classic Centralized Financial (*CEFI*) systems are trust-based systems (users have to trust that the exchange won't get hacked, abscond with the user's assets, incorrectly close out a position or even suffer downtime). Users rely on centralized authorities to manage their assets and often have to trust that the protocols implemented are the same as the ones advertised.

To properly develop a Decentralized *DEFI* ecosystem, **Open Finance**[3] platforms must be deployed on the main blockchains, since those are the ones that move the most volume, thus increasing decentralization.

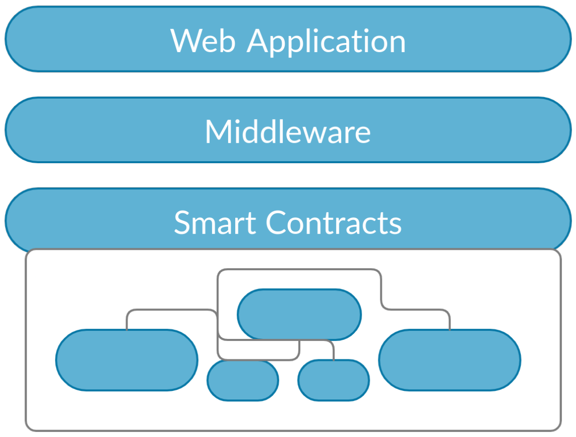
In this paper, we propose an algorithmic distributed protocol aiming to be deployed on the RSK network. Our main motivation is to allow users to lend crypto currencies as collateral and to borrow crypto-assets based on interest rates set by real-time supply and demand smart contracts.

---

[1] The RSK Network
[2] 2016 article on pegged side-chains
[3] 2020 article on Open Finance

1 Architecture

The proposed solution would require several implementation layers. The next graph shows the abstract scheme.



We consider the top level to be the application layer, consisting of the **rLending User Interface**[4].
In contrast, the low-level layer composing the **rLending Protocol**[5], is distributed (*deployed*) as a set of smart contracts on the RSK Network.
The middle layer is a custom made library that manages the interactions between the lower level (*back-end*) and the top level (*front-end*). This **middleware**[6], as the rest of the source code, is also publicly available and could be used to develop alternative applications to connect with the protocol, such as bots, alternative transaction explorers or automated liquidators.

1.1 The rLending Protocol

The **rLending protocol** establishes money markets, which are pools of assets with algorithmically derived interest rates, based on the supply and demand for the crypto-asset. Suppliers (and borrowers) interact directly with the protocol, earning (and paying) a floating interest rate, without having to negotiate terms such as maturity, interest rate, or collateral with a peer or counterparty. Each money market is unique to an asset already in the **RSK Network** (such as rBTC, an ERC-20 stablecoin such as rKovDAI, or any ERC-20 compatible token such as RIF), and contains a transparent and publicly-inspectable ledger, with a record of all transactions and historical interest rates.

1.1.1 Supplying

Unlike an exchange or peer-to-peer platform, where a user's assets are matched and lent to another user, the rLending protocol aggregates the supply of each user; when a user supplies an asset, it becomes a fungible resource, effectively forming a liquidity pool. This approach offers significantly more liquidity than direct lending; unless every asset in a market is borrowed **(see below: the protocol incentivizes liquidity)**, users can withdraw their assets at any time, without waiting for a specific loan to mature.
Assets supplied to a market are represented by an internal ERC-20 token balance (**"cToken"**), which entitles the owner to an increasing quantity of the underlying asset. As the money market accrues interest, which is a function of borrowing demand, cTokens become convertible into an increasing amount of the underlying asset. In this way, earning interest is as simple as holding **cTokens**[7].

1.1.2 Borrowing

Using cTokens as collateral, the **rLending Protocol** allows users to frictionlessly borrow assets from the liquidity pools (*markets*), for use anywhere in the RSK ecosystem. Unlike peer-to-peer protocols, borrow-

[4]rLending User Interface source code
[5]rLending protocol source code
[6]rLending middleware source code
[7]rLending cToken Documentation

ing from **rLending** simply requires a user to specify a desired asset; borrowing is instant and predictable. Similar to supplying an asset, each money market has a floating interest rate, set by market forces **(supply and demand)**, which determines the borrowing cost for each asset.

1.1.3 Collateralization

Assets held by users of the protocol, **represented by ownership of a cToken**, are used as collateral to borrow from the protocol. Each market has a **Collateral Factor**[8], ranging from 0 to 1, that represents the portion of the supplied underlying asset value that can be borrowed. Illiquid, small-cap assets tend to have low collateral factors; they do not make good collateral, while liquid, high-cap assets have high collateral factors. The sum of the value of an account's underlying token balances, multiplied by the collateral factors, equals a user's **borrowing capacity**.

Users are able to borrow up to, but not exceeding, their borrowing capacity, and an account can take no action (e.g. borrow, transfer cToken collateral, or redeem cToken collateral) that would raise the total value of borrowed assets above their borrowing capacity; this protects the protocol from default risk. Users should never borrow amounts close to their borrowing capacity, or else they run the risk of becoming prone to liquidation.

1.1.4 Risk and liquidation

If the value of an account's borrowing balance exceeds their borrowing capacity, a portion of the outstanding borrowing may be repaid in exchange for the user's cToken collateral, at the current market price minus a liquidation discount (**Liquidation Factor**[9]; this incentives an ecosystem of arbitrageurs to quickly step in to reduce the borrower's exposure, and eliminate the protocol's risk.

Any rBTC address that possesses the borrowed asset may invoke the liquidation function, exchanging their asset for the borrower's cToken collateral. As both users, both assets and prices are all contained within the **rLending protocol**, liquidation is frictionless and does not rely on any outside systems or managed order-books.

1.2 The Interest Rate Model

Rather than individual suppliers or borrowers having to negotiate over terms and rates, the **rLending protocol** utilizes an interest rate model that achieves an interest rate equilibrium, in each listed market, based on supply and demand. Following economic theory, interest rates (the "price" of money) should increase as a function of demand; when demand is low, interest rates should be low, and vice versa when demand is high. The utilization ratio U for each market α unifies supply and demand into a single variable:

$$U_a = Borrows_a / (Cash_a + Borrows_a)$$

The demand curve is expressed as a function of utilization. As an example, borrowing interest rates may resemble the following:

$$BorrowingInterestRate_a = 2.5\% + U_a * 20\%$$

The interest rate earned by suppliers is implicit and is equal to the borrowing interest rate, multiplied by the utilization rate.

The protocol does not guarantee liquidity; instead, it relies on the interest rate model to incentivize it. In periods of extreme demand for an asset, the liquidity of the protocol (the tokens available to withdraw or borrow) will decline; when this occur, interest rates rise, incentivizing supply and disincentivizing borrowing.

[8] rLending Collateral Factor Documentation
[9] rLending Liquidation Documentation

1.3 Middleware layer

We propose a custom middleware implementation based on **Ethers.js**[10]. This layer is used to facilitate the interaction with the protocol, by providing an interface to the core functions. Some of the most interesting functions are listed here:

- **getMarkets(address)** returns the markets a user has entered as collateral.
- **getCTokenMetadata(address)** fetches the metadata for a given cToken
- **getAccountLiquidity(address)** queries the protocol and fetches the accounts liquidity or shortfall
- **getTotalSupplysAndBorrows(address)** calculates the sum of all supplies and borrows for a given account
- **getAccountHealth(address)** calculates the account health factor
- **getAccountBalanceOfCtoken(address)** queries the cToken contract to fetch the account balance
- **balanceOf Underlying(address)** calculates the equivalent balance of underlying asset an account has
- **getSupplyRate()** gets the current supply rate per block for a given market
- **getBorrowRate()** gets the current borrow rate per block for a given market
- **enterMarket() / exitMarket()** perform said functions for a given account in a given market, respectively
- **getMaxWithdrawAllowed(address)** calculates the maximum amount of withdrawable assets an account may extract from the protocol
- **supply()**, **borrow()**, **repay()**, **withdraw()**, **liquidate()**, etc..

We strongly encourage developers to utilize its features, since the functionalities provided allow for an easier understanding of the protocol's capabilities, internal functions and structures.

1.4 Application layer

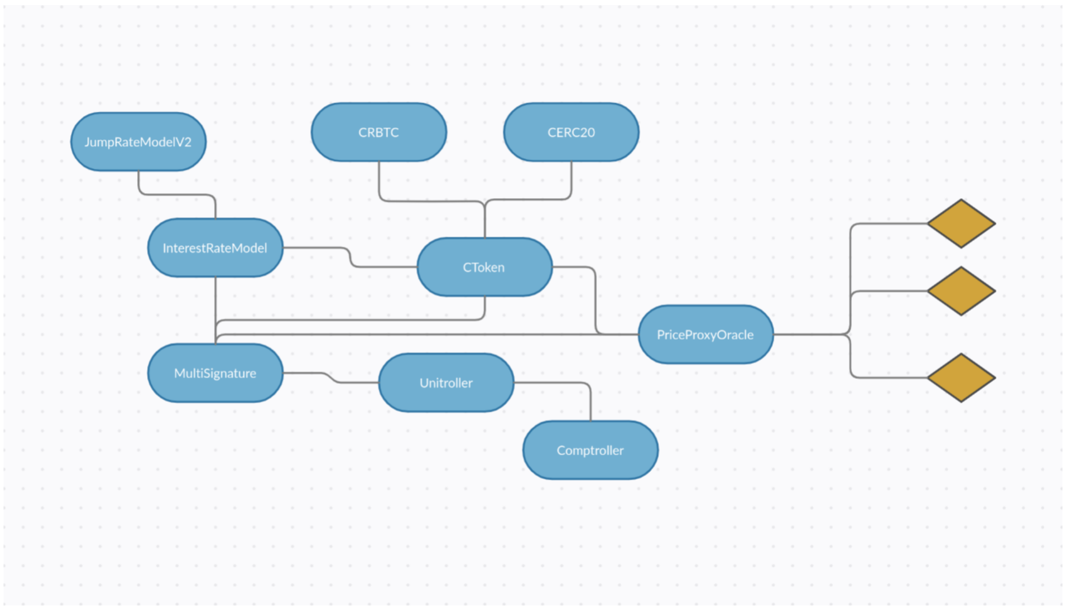
We propose a user interface implementation based on the **rBank PoC**[11]. We developed a web based application that is intended to facilitate the interaction with the protocol and provide the user with relevant data visualization screens. We expect users to be able to supply, borrow, repay debt, withdraw balance, check their current investments and analyze the current status of the market variables in an effective yet intuitive manner.

2 Implementation

In this section we review the implemented solution and expose the main components and their facilities. We adopted many of the functionalities from the **Compound protocol**[12] since it is well established as being an open source platform with publicly available third-party successful audits. Technical details and source code for this section are provided in **the rLending documentation**[13] and the **the rLending github repositories**[14]. We encourage the reader to further research them before using the platform.

[10] Ethers.js documentation
[11] rBank proof of concept source code
[12] The Compound protocol
[13] The official rLending documentation
[14] The rLending public repositories

2.1 General Smart Contract relationship graph



2.2 CTokens contracts

At the core of the **rLending Protocol** are **cTokens**, identifying each underlying asset supported by the platform. This section describes the *CToken.sol* contract subsystem and its functionalities.

- **CTokens** (CErc20.sol and CRBTC.sol)

cTokens are self-contained borrowing and lending contracts. **CToken.sol** contains the core logic. Each *cToken* is assigned an interest rate and risk model (see **InterestRateModel** and **Comptroller** sections), and allows accounts to *mint* (supply capital), *redeem* (withdraw capital), *borrow* and *repay a borrow*. Each **CToken** is an **ERC-20** compliant token where balances represent ownership of a portion of the liquidity pool (see **market**).
In other words, since each asset supported by the **rLending Protocol** is integrated through a **CToken**, by minting them, users (1) earn interest through the **cToken's exchange rate**, which increases in value relative to the underlying asset, and (2) gain the ability to use **cTokens** as collateral for borrowing.
cTokens are the primary means of interacting with the Protocol and whenever a user mints, redeems, borrows, repays a borrow, liquidates a borrow, or transfers **cTokens**, they will do so indirectly using this contract.

There are currently two types of **cTokens**: **CErc20.sol** and **CRBTC.sol**. Though both types expose the *EIP-20* interface, **CErc20.sol** wraps any underlying *ERC-20* asset, while **CRBTC.sol** simply wraps **rBTC** (*the base currency within the RSK Network*) itself. As such, the core functions which involve transferring an asset into the protocol have slightly different interfaces depending on the type.

- **Comptroller contracts**(Unitroller.sol and Comptroller.sol)

This is the risk model system, which validates permissible user actions and disallows actions if they do not fit certain risk parameters. For instance, the Comptroller enforces that each borrowing user must maintain a sufficient collateral balance across all *cTokens*, or else incur in shortfall and becoming prone to liquidation.

Unitroller is a proxy contract that controls the storage and admin functions for the latest implementation of *Comptroller*. **rLending** uses **ComptrollerG4.sol** which in turn is the contract that updates the *Comptroller*'s risk model system, that keeps track of the listed markets, manages the user's mints and borrows, allowances and market limits.
If a user's borrowing balance exceeds their total collateral value (borrowing capacity) due to the value

of collateral falling, or borrowed assets increasing in value, the public *function liquidate()* can be called, which exchanges the invoking user's asset for the borrower's collateral, at a *slightly better*[15] than market price.
In case you need further information regarding the many roles this subsystem plays, we encourage you to read up on the official rLending documentation[16].

- **Interest Rate Model subsystem**

Markets are defined by an interest rate, applied to all borrowers uniformly, which adjust over time as the relationship between supply and demand changes. The history of each *interest rate*, for each asset market, is captured by an **Interest Rate Index**, which is calculated each time an interest rate changes, resulting from a user minting, redeeming, borrowing, repaying or liquidating the asset. Each time a transaction occurs, the**Interest Rate Index** for the asset is updated and compounded with the interest since the prior index, using the interest for the period, denominated by $r * t$, calculated using a per-block interest rate:

$$Index_{a,n} = Index_{a,(n-1)} * (1 + r * t)$$

The market's total borrowing outstanding is updated to include interest accrued since the last index:

$$totalBorrowBalance_{a,n} = totalBorrowBalance_{a,(n-1)} * (1 + r * t)$$

And a portion of the accrued interest is retained (set aside) as reserves, determined by a **reserveFactor**, ranging from 0 to 1:

$$reserves_a = reserves_{a,(n-1)} + totalBorrowBalance_{a,(n-1)} * (r * t * reserveFactor)$$

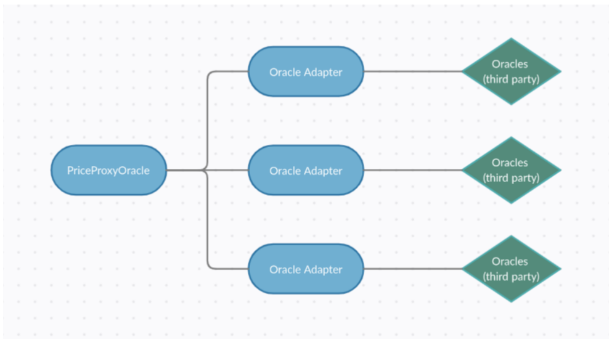
This way, a borrower's balance, including accrued interest, is simply the ratio of the current index divided by the index when the user's balance was last checkpointed. The balance for each borrower address in the cToken is stored as an account checkpoint. An account's checkpoint is an internal Solidity[17] structure describing the balance at the time interest was last applied to that account.
InterestRateModel is an interface that wraps the different versions of interest models. Each one of them improves on the previous form for calculating borrow and utilization rates. The latest implementation being **JumpRateModelV2.sol**.

2.3 Price Oracle contracts

This subsystem controls the integration of **Price Oracle Feeds** for any given market listed in the **rLending Protocol**. A Price Oracle maintains the current exchange rate of each supported asset while the protocol delegates the ability to set the value of assets to a committee[18] which pools prices from the top exchanges. These exchange rates are used to determine borrowing capacity and collateral requirements. The rates are also used for all functions which require calculating the value equivalent of an account.

- **PriceOracleProxy.sol**

This contract manages the mapping of the *cTokenAddress* to *adapterAddress*. Through this proxy the protocol abstracts itself from any oracle's interface. Each **PriceOracleAdapter.sol** implements a specific oracle that is to be included in the mapping, making the oracle's interface code-agnostic from the proxy side.



[15]see liquidation factor
[16]The official rLending Documentation
[17]Solidity documentation
[18]see Multi-sig contract

- **OracleAdapters Layer**

Given that we foresee the posibility of diverse Oracle systems being deployed in the network, we developed a middle-layer contract adapter system to interact with each Oracle as intermediary. This way, the **rLending Protocol** is capable of receiving price information from every kind of Oracle, regardless of it's implementation.

2.4 Gnosis Multisignature Integration

At launch **rLending** is relying on *Gnosis Multisig*[19] to create a committee of specialists that will manage the protocol's most critical functionalities. After a reasonable time on main-net, we propose moving on to a Governance based schema.
This means that the initial **rLending** deploy will have *admin/guardian* addresses pointing to *Gnosis Multisig* contract, that has distributed the capability of updating core values within the protocol:

- The ability to list a new cToken market
- The ability to update the interest rate model per market
- The ability to update the oracle or any adapter address
- The ability to withdraw the reserve of a cToken
- The ability to update the Risk model contracts
- The ability to choose a new admin

3 Final notes and considerations

- We acknowledge that **rLending** will launch as a partially decentralized application. As stated in the previous section: the protocol relies on a committee regulating the Gnosis Multi-Signature contracts. After a grace period, the protocol should fully transition to a governance system.
- Threat models and Security checks, as well as audits for the inherited Compound code, have been reviewed both by the rLending team and third parties with satisfactory results.
- We strongly encourage inexperienced users to carefuly study the documentation and gain a deep understanding of the risks involved before using the platform.
- **rLending** is distributed under *BSD-3 Licence*[20]

Contact: support@rlending.app
Git: github.com/riflending

Try the rLending Open Beta

[19] https://github.com/gnosis/MultiSigWallet/blob/master/contracts/MultiSigWallet.sol
[20] https://github.com/riflending/rlending-protocol/blob/master/LICENSE