Everscalend Functional Specification

Prepared by Pruvendo 03/20/22

Executive summary

The purpose of the present document is to provide the detailed functional specification of the *Everscalend* - decentralized lending system. The created specification is supposed to be turned into a formal specification and later used for the formal verification of the product.

This document considers the Everscalend code from the repository https://github.com/SVOIcom/everscalend-contracts/, published in commit record **8d24e268f9c44bd3e896fb6a28bbf8a42c7027a9**.

We studied the following smart-contracts: *MarketAggregator*, *Oracle*, *UserAccountManager, UserAccount*, *WalletController*, *WalletController, User*, interactions between modules entitled *SupplyModule*, *WithdrawModule*, *BorrowModule*, *RepayModule*, *LiquidateModule,* as well as and supplementary libraries and interfaces.

Please address all the questions and comments to the Telegram account @SergeyEgorovSPb.

Section I. High-level description

I.I. System purpose

Everscalend is a P2P secured lending platform that pays interest on deposits and focuses on creating fund pools. *Everscalend* is organized as a DeFi platform enriched with money market protocol powered by Everscale.

The system serves participants of an anonymous closed investment pool. According to available documentation, the benefit for clients is that they can earn interest income or pay a flexible interest rate for borrowed funds.

Thus the system is designed so that members of the pool can make savings in terms of TIP-3 tokens, and have an option to borrow from the pool. The pool operates on internal tokens, with USD pricing-to-market for every operation for client assets and liabilities. All operations are supported by blockchain records. A client has five operations to exercise: to make an input to the fund, to take it back, to borrow from the fund, to pay a credit back. Also there is an operation to cover the debt of a third party that has difficulties to pay back a loan.

I.II. Key principles

We used general principles of loan/credit systems for the analysis. They are: returnability, maturity, no-free lunch, coverage[1].

[1]https://www.finance-watch.org/take-care-of-the-crowd-legal-protection-of-retail-investors-in-crowdfunding-is-long-overdue/, P.Rose, S.Hudgins, Bank Management and Financial Services, 9-th ed.2013 , Microeconomics of Banking, X.Frexais, J-C.Rochet, 2-nd ed. 2018, Contemporary Financial Intermediations, S.Greenbaum, A. Thakor, 4-ed, 2019, Commercial bank financial management, F.Sinkey, 6 ed. 2002.

Returnability as a flow of funds from clients to the system is implemented with a client procedure Repay, which is supported by a collateral.

Maturity as a control for a time interval, no later when a loan must be paid.

No-free lunch as the impossibility to use loans for free seems to have vulnerabilities.

Coverage as collateral held by a creditor to be able to confiscate it in a case of default of a borrower.

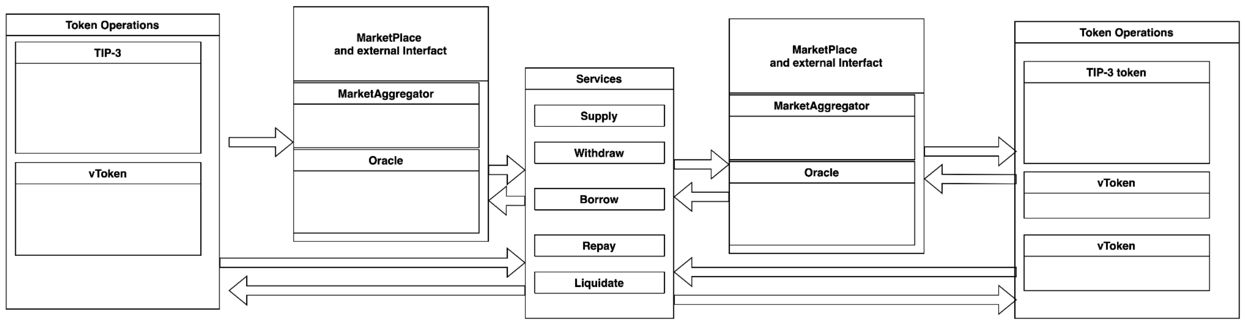
The system assumes that all credits are secured by a prerequisite deposits from borrowers[2]. Borrowing interest rate is dynamically determined only by current demand and supply of funds. Such a view on interest rate formation assumes:

- all credits are paid due, and there is no need to control the maturity of loans,
- there is a complete coverage of all credits.

The purpose of the system is to supply intertemporal operation, a loan, or a credit, for members of the pool. By means of loan/credit systems it means that some time later a loan should be followed by a repayment of debt, or by a default of a loan. Thus a record on borrowing must include an operation timestamp, a maturity of a loan or a date of covering a debt, as well as a status of a borrowing operation. These are the necessary requirements for functioning of a loan/credit system, and their presence will be checked later.

I.III. System architecture

[2] All operations are priced-to-market in terms of the USD



This section concentrates on the operational part of client processes.

Client oriented part of the system architecture is presented above. It provides five operations for a lending system. There are a few structural units, which make an architecture of the system:

- token operations unit,
- market operations unit,
- functional service unit.

The *Token Operations* unit supports operations with TIP-3 (external) and internal vTokens. This includes bookkeeping services for every token as a private ledgering of the user's operations, and an interface with the rest of the system.
The *Marketplace* unit supports book-keeping of all operations of the system for all clients of the marketplace, interacts with external market data sources, and produces pricing-to-market for every transaction.

The *Service* unit includes specific contracts for every client service: make an input to the system, withdraw from the system, borrow, payback, and liquidate. Each client process is supported by an execution of a series of contracts and is described further in more details.

Each client process exploits only one operation in the Services unit. If two units on the figure above have a few arrows between each one, then one from left to right always operates, unless there is another left-to-right arrow above. A reversed arrow operates for some contracts, which is explained further for every special case.

Order of units and execution of their contracts is service specific. There are three additional library services, shared by the client processes, not shown above:

- transfer tokens operation,
- unlock operation,
- check the financial health of a client.

The general structure of operations is approximately the following:

- Receive a request for the *MarketAggregator* from one of two sources: wallet controller for TIP-3 tokens or from *UserAccountManager* for internal tokens.
- Update information on markets and perform pricing to market for the current operation.
- Perform client service.
- Perform necessary registration after client service and make necessary requests to token services. If necessary, repeat a client service.
- Update information on financial health of a client and finalize an operation by delivering required tokens or returning unused tokens.

Most modules have a possibility to interrupt the current operation and safely cancel the transaction by returning tokens to a sender.

Operations of units

Every request to the system starts and finishes at the Token Operations unit, which consists of two independent blocks with similar functions. Each block processes a specific token.

Block TIP-3 serves and processes external tokens for client operations: a supply of TIP-3 tokens, a payment for credits in TIP-3 tokens, or a payment for a third party. The TIP-3 block includes three contracts: *TIP-3UserWallet*, *TIP-3Wallet* and *WalletController*. The block runs a private ledger for every client.

Block vTokens serves internal tokens services of Everscalend. It has two blocks, each containing one contract: *UserAccount* and *UserAccountManager*.

If a client process starts from the TIP-3 block, then it finally ends with the vToken block. If a client process starts from a vToken block, then it finally ends with a TIP-3 block, besides the Liquidate process. The block runs a private ledger for every client.

Public ledgering and pricing-to-market are served by the MarketPlace unit. It has two blocks, each with one contract. The Contract *MarketAggregator* updates and holds a public ledger of the system and sends/receives market data from *Oracle*. Contract *Oracle* is an interface to external data available only to the *MarketAggregator* contract.

Functional unit includes contracts, which serve core user services for clients. They are:

- *Supply* - accepts funds from clients.
- *Withdraw* - returns funds to a client.
- *Borrow* - supplies credits to participants.
- *Repay* - collects credit payments from participants.
- *Liquidate* - covers someone's debt by a third party.

The reversed pairs of operations are *Supply*-*Withdraw*, *Borrow*-*Repay*. Liquidate aims to cover imperfection of the second pair.

A user can have two roles - a *saver* and a *borrower*, and combine them. Every user must own a msig and TIP-3 token before any operation starts.

I.IV. Terms

In the document, we use several terms that are defined below.

Terms	Definition
TIP-3 token	TIP-3 token, external to the system.
vToken	TIP-3 token, internal to the system.

Multisig	Wallet that keeps native EVER coins
TIP-3 root	The root contract for the specified TIP-3.
TIP-3 wallet	An account in external token TIP-3.
Giver	A client, which supplies TIP-3 tokens to the pool, without an option to withdraw them.
TIP-3 Deployer	The owner of TIP-3 root.
Unit	A functional block of an architecture diagram. It unites operations with similar or closely tight goals.
MarketAggregator	A unit to evaluate operations in external currency, send/request service for external data, and hold/supply a ledger for all clients. At the moment the external currency is USD.
pricing-to-markets	Evaluation of tokens in terms of an external currency, currently USD.
Oracle	Interface with outside data sources.
Financial health	Financial soundness, eligibility to borrow and to cover debts of others.
Vulnerability	Operational, market and technical risks.
SupplyModule	Service to accept TIP-3 tokens from clients, deposit them and make relevant records in public and private ledgers.
WithdrawModule	Opposite to Supply. Service to return TIP-3 tokens to clients. It destroys internal tokens and provides access to external tokens to a client.
BorrowModule	Collateralized credit service. Rate of coverage of a credit is not clear from available documents.
Repay	Opposite to borrowing, a service to pay a credit back in terms of TIP-3 tokens.
LiquidationModule	Service to pay credit to cover insolvency for another borrower. Unlock operation for a client does not concern the client's credit history.
Public ledger	Anonymous record of all updated operations of the system on every (external, internal) pair of tokens. At the moment the pair is TIP-3(ever) and vToken.

UserAccountManager	Internal token book-keeping service for internal tokens. It serves as an interface between MarketAggregator and client internal token data.
WalletController	TIP-3 token book-keeping service for external tokens. It serves as an interface between MarketAggregator and client TIP-3 token data.
Private ledger	Updated records of a client's operations with a token. A client has individual private ledgers for every token.
UserAccount	Private account for internal tokens. In the current version the internal token is vToken.
WalletController	TIP-3 token accounting management. It serves as an interface between MarketAggregator and client TIP-3 token data.
pool	The sum of all anonymized deposits of members of the system.
deposit	TIP-3 tokens, registered and holded by the system for every user.
lock / unlock	Ban on borrowing for a client. Lift a ban to borrow.

I.V. Client services (scenarios)

Each process provides only one service for a client and can be described independently. Every process makes relevant records in the private ledgers of a client and the private ledger of a marketplace.

The *Supply* process accepts funds nominated in external (currently only TIP-3 tokens), generates an equivalent to USD quantity of internal vTokens, and updates the pool.

The *Withdraw* process takes internal vTokens, destroys them and returns TIP-3 tokens, equivalent to the USD value of native tokens.

The *Borrow* process provides lending services by locking internal tokens of a client and supplying external (TIP-3) tokens.

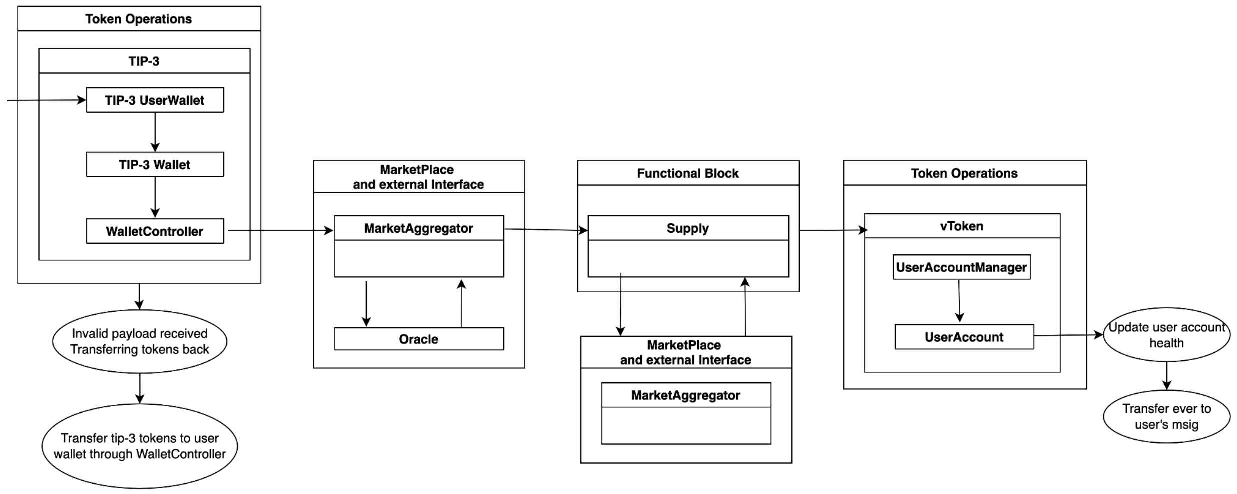
The *Repay* process provides a service for a borrower to pay credits back. The process accepts external (TIP-3) tokens, covers a debt in vTokens and unblocks the borrower.

The *Liquidate* process allows a third party to cover a debt for someone, by providing external (TIP-3) tokens, unblocking native tokens of this someone, and receiving residual internal tokens of this someone.

Each process is explained further in detail. Business concerns of each process are described in the relevant section.

Supply

To become a member of the *Everscalend* community, a client must make a preliminary input into a fund. This section describes operations of *Supply* client service. The current version assumes a submission in TIP-3 tokens.


Token Operations
TIP-3
TIP-3 UserWallet
TIP-3 Wallet
WalletController
Invalid payload received Transferring tokens back
Transfer tip-3 tokens to user wallet through WalletController
MarketPlace and external Interface
MarketAggregator
Oracle
Functional Block
Supply
MarketPlace and external Interface
MarketAggregator
Token Operations
vToken
UserAccountManager
UserAccount
Update user account health
Transfer ever to user's msig

	From	To	Operation	Meaning	Error
1	*user's Multisig*	*user TIP-3 wallet*	transfer request	Receive a request from a (prospective) client to accept a fund for a pool	
2	*user TIP-3 wallet*	*TIP-3 WalletControl ler wallet*	Token transfer	Register a request in a private ledger	
3	*TIP-3 WalletControl ler wallet*	*WalletControl ler*	performOper ationWalletC ontroller	Validate the operation	Invalid payload received, transfer tokens back.
4	*WalletControl ler*	*MarketsAggr egator*	performOper ationWalletC ontroller	Forward a supply request to the marketplace.	
5	*MarketsAggr egator*	*Oracle*	getAllTokenP rices	Get information about current prices to evaluate a new fund.	
6	*Oracle*	*MarketsAggr egator*	receiveAllUp datedPrices	Marketplace gets updated prices.	
7	*MarketsAggr egator*	*SupplyModul e*	performActio n	A new input was prices to market	
8	*SupplyModul e*	*MarketsAggr egator*	receiveCache Delta	Operation was registered at the public ledger and forwarded to	

				exectution.	
9	*MarketsAggregator*	*Supply*	resumeOperation	The pool was updated	
10	*Supply*	*UserAccount Manager*	writeSupplyInfo	Operation is forwarded to register in terms of internal tokens.	
11	*UserAccount Manager*	*UserAccount*	writeSupplyInfo	Permission to register changes!	
12	*UserAccount*	*System libraries to finalize*	Update user account health Transfer ever to user's Multisig	Finalize the operation by calculating financial health.	

A *Giver* is a member who supplies to a fund and does not have an option to withdraw a deposit at any time. It is similar to long-run investment in closed mutual funds. A *Client* is a member of the community who supplies the fund, but has an option to withdraw the deposit at any time. This is similar to short-run investment.

A constructed fund is an anonymous pool of the TIP-3 tokens. Any member is eligible to borrow from the fund. Every deployment of external tokens is managed by the process *Supply*. The process starts from a payload of a client and a valid *TIP3UserWallet.* If a payload is incorrect, the process is terminated and a sum is forwarded back to the sending wallet. The picture above presents a sequence of calls between functional units and their contracts for the process *Supply*.

If the payload is correct, a client request is validated by a *TIP3Wallet* contract and the *WalletController* contract forwards it further to the *MarketPlace* unit. The

MarketAggregator contract prices-to-market the deposit in terms of the USD, by making a request to the contract-interface *Oracle*.

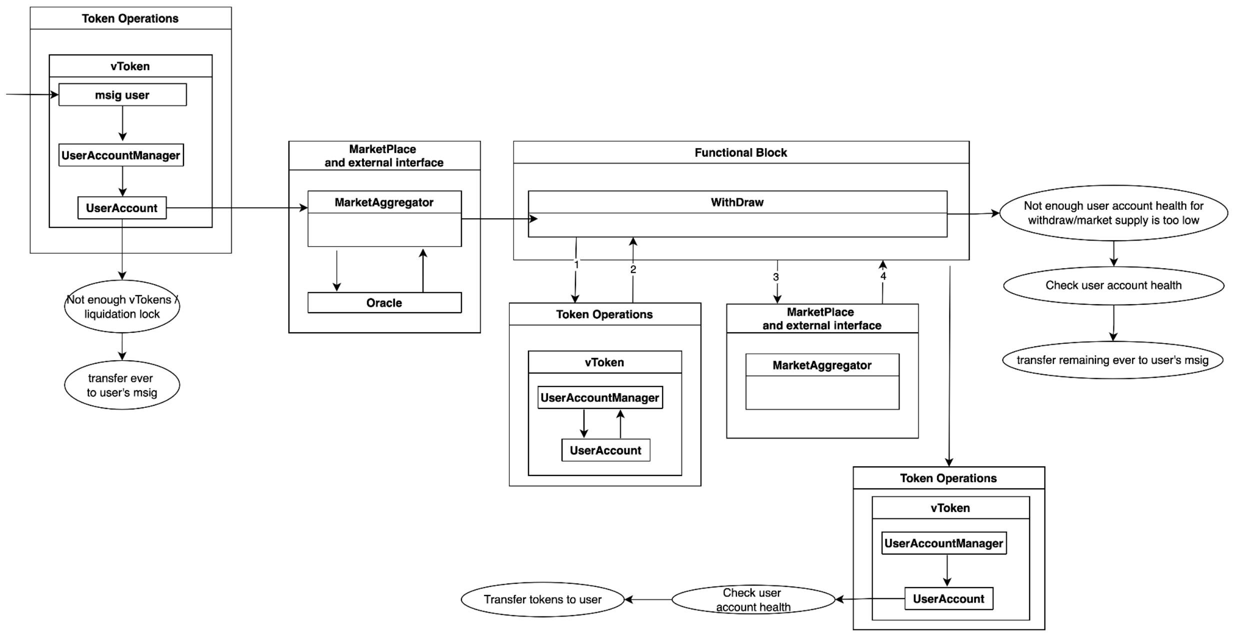
Then again the module *Supply is ac*tivated, it calculates a quantity of required tokens and makes a record with the *MarketAggregator* to the general investment fund that a new input has been added to the investment pool. The final record for a client about the end of the operation is sent by the *UserAccountManager*, to a personal ledger by the *UserAccount*.

The operation is finalized by providing the *Client* an equivalent quantity of internal (native) currency, which is recorded as *Multisig*. Success of the *Supply* operation enables the client to borrow. Service processes update the financial health of a client and unused TIP-3 tokens are returned.

Withdraw

Withdraw service allows a *Client* to take his TIP-3 tokens back from the pool. The operation is critical for protection of the pool from losses induced by external attacks.

The operation is available only to members of the pool, i.e. without an earlier deposit in TIP-3 tokens this procedure is unavailable and must be


Token Operations
vToken
msig user
UserAccountManager
UserAccount
Not enough vTokens / liquidation lock
transfer ever to user's msig
MarketPlace and external interface
MarketAggregator
Oracle
Functional Block
WithDraw
1
2
3
4
Token Operations
vToken
UserAccountManager
UserAccount
MarketPlace and external interface
MarketAggregator
Not enough user account health for withdraw/market supply is too low
Check user account health
transfer remaining ever to user's msig
Token Operations
vToken
UserAccountManager
UserAccount
Check user account health
Transfer tokens to user

unavailable.

The process can be interrupted in the following cases:

- A *Client* has a lock to withdraw, if his financial status does not allow him to withdraw a required sum.
- A client wants to withdraw more than he owns.
- A market does not have enough tokens in the pool at the moment.
- The contract *Withdraw* has a lock, and a new operation can not start.

	From	To	Operation	Meaning	Error
1	*External input*	*UserAccount*	withdraw	Accept a request to withdraw. If not enough vTokens, transfer EVERs to users's Multisig	Not enough vTokens / liquidation lock
2	*UserAccount*	*UserAccount management*	requestWithdraw	Request to withdraw to a	

				book-keeping system	
3	*UserAccount management*	*MarketsAggregator*	performOperationUserAccountManager	Request to value withdrawing sum	
4	*MarketsAggregator*	*Oracle*	getAllTokenPrices	Request for USD price	
5	*Oracle*	*Withdraw module*	performAction	Withdrawal is valued and permitted	
6	*Withdraw module*	*UserAccount Manager*	requestWithdrawInfo	Withdrawal can be registered in private ledger	
7	*UserAccount Manager*	*UserAccount*	requestWithdrawInfo	Make a record in the private ledger	
8	*UserAccount*	*UserAccount Manager*	receiveWithdrawInfo	Record has been done	
9	*UserAccount Manager*	*Withdraw module*	withdrawTokensFromMarket	Can make withdrawal from the pool	
10	*Withdraw module*	*MarketsAggregator*	receiveCacheDelta	Request to decrease the pool	Not enough user account health for withdraw/market Check user account health supply is too low transfer remaining ever to user's msig
11	*MarketsAggr*	*Withdraw*	resumeOpera	Withdrawal	

	egator	*module*	tion	completed	
12	*Withdraw module*	*UserAccount Manager*	writeWithdrawInfo	Register operation	
13	*UserAccount Manager*	*UserAccount*		Register the withdrawal	
14	*UserAccount*	*Library services*	Check user account health Transfer tokens to user	Withdrawal done	

The operation is available only to members of the pool. This means that without an earlier deposit in TIP-3 tokens, this procedure is unavailable and must be unavailable.

A request contains a number of tokens to withdraw. It is checked by *UserAccount*, and if successful, it is forwarded to *UserAccountManagement*, which sends the request to *MarketAggregator. MarketAggregator*, through *Oracle*, makes a pricing-to-market request and makes a record into the public ledger. The *MarketAggregator* requests the *WithdrawModule* to calculate the number of withdrawable tokens. The result is forwarded to the *UserAccountManage and UserAccount* to cancel the required quantity of native tokens. Then the operation is forwarded back to the public ledger in *MarketsAggregator* and proceeds with the operation. Then it forwards to make a final record at the private ledger *UserAccount* via *UserAccountManager,* and then is forwarded to check the financial health of a client, before TIP-3 tokens are transferred to the client.

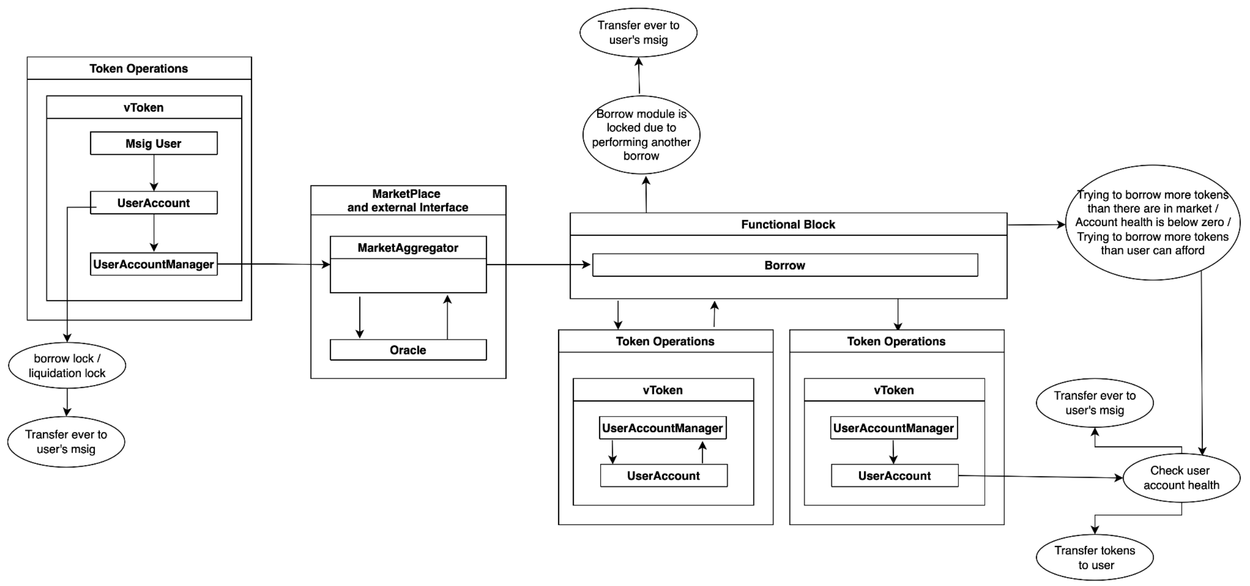
The operation is finalized by forwarding information about which tokens and how much to withdraw, withdrawing tokens to the user's TIP-3 wallet by activating *WalletController* and requesting TIP-3 Walet to supply TIP-3 tokens.

Borrow

This client service is a tool to provide liquidity in terms of the TIP-3 tokens. The service is available only to members of the pool, i.e. to those who have submitted TIP-3 tokens earlier.

	From	To	Operation	meaning	Error
1	*External input*	*UserAccount*	borrow	A client sends request to withdraw TIP-3 tokens	borrow lock / liquidation lock Transfer ever to user's msig
2	*UserAccount*	*UserAccount Manager*	requestIndex Update	Update Index	
3	*UserAccount Manager*	*MarketsAggr egator*	performOper ationUserAcc ountManager	Request to market for borrowing from the pool	
4	*MarketsAggr egator*	*Oracle*	getAllTokenP rices	Request to prices to external sources	
5	*Oracle*	*MarketsAggr egator*	receiveAllUp datedPrices	Provide updated prices	
6	*MarketsAggr egator*	*BorrowModul e*	performActio n	Permission to borrow	Borrow module is locked due to performing another borrow Transfer EVERs to user's Multisig
7	*BorrowModul*	*UserAccount*	updateUserIn	Update	

	e	*Manager*	dexes	private ledger of a client	
8	*UserAccount Manager*	*UserAccount*	borrowUpdateIndexes	Registration of operation	
9	*UserAccount*	*UserAccount Manager*	passBorrowInformation	Registration completed	
10	*UserAccount Manager*	*BorrowModule*	borrowTokensFromMarket	Final permission to borrow after registration	Trying to borrow more tokens than there are in market / Account health is below zero / Trying to borrow more tokens than user can afford Check health without token transfer Check user account health
11	*BorrowModule*	*UserAccount Manager*	writeBorrowInformation		
12	*UserAccount Manager*	*UserAccount*	writeBorrowInformation		
13	*UserAccount*	*Libraries of processes*		Check user account health Transfer ever to user's msig Transfer tokens to user	


Token Operations
vToken
Msig User
UserAccount
UserAccountManager
borrow lock / liquidation lock
Transfer ever to user's msig
MarketPlace and external Interface
MarketAggregator
Oracle
Transfer ever to user's msig
Borrow module is locked due to performing another borrow
Functional Block
Borrow
Trying to borrow more tokens than there are in market / Account health is below zero / Trying to borrow more tokens than user can afford
Token Operations
vToken
UserAccountManager
UserAccount
Token Operations
vToken
UserAccountManager
UserAccount
Transfer ever to user's msig
Check user account health
Transfer tokens to user

A client can borrow with an interest rate, dynamically calculated from current demand and supply of funds. The service can be interrupted in the following cases:

- A *Client* has a lock, if his financial status does not allow to be a payable borrower.
- A *Client* has a lock, if at the moment he is performing another borrowing operation.
- A *Client* does not have enough tokens to collateralize a required credit.
- A *Client* does not have adequate collateral to obtain another credit.

Ultimate sum of a credit is measured in TIP-3 tokens, so that clients can use it outside the system.

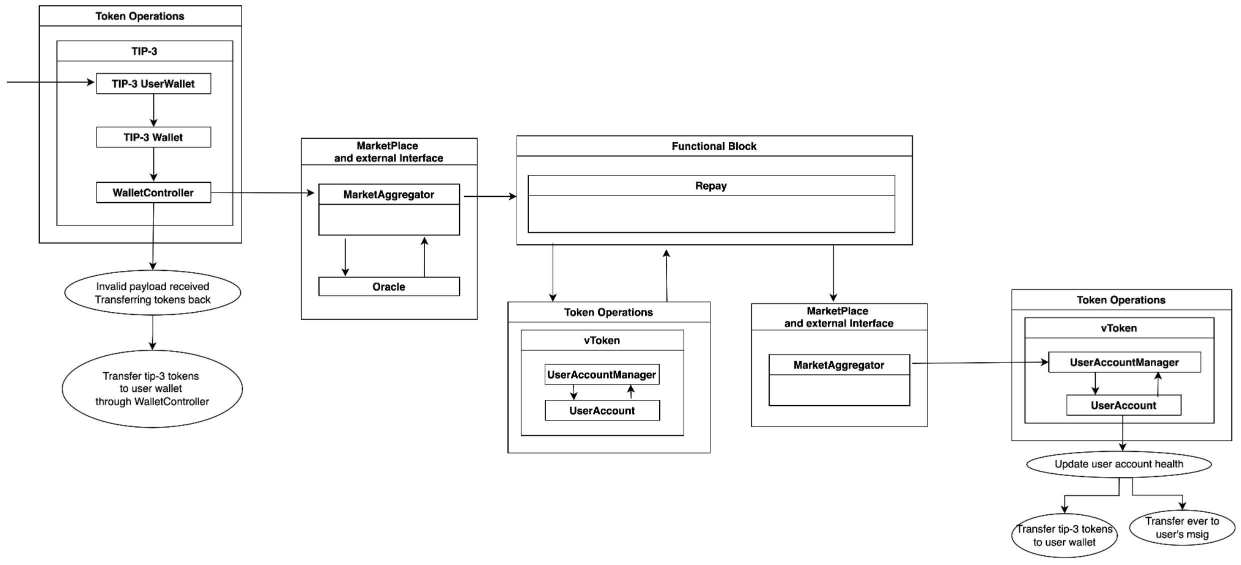
The process starts from the *Token* operations unit. A client forwards his/her Multisig and a number of the TIP-3 tokens to borrow to the *UserAccount*. This request is recorded by the *UserAccountManager.*

The accounting system of internal tokens forwards a request to the *MarketAggregator*, which records the operation in the public ledger by the *MarketAggregator*, and prices it to-market with an assistance of the *Oracle*. Further, a request is forwarded to the contract *Borrow*, which allows borrowing. Then the request is forwarded to the *UserAccountManager* and *UserAccount*, which updates the private ledger, and returns the request to the same contracts to block the client. After that the request is forwarded to free TIP-3 tokens for a client by library procedures.

Repay

This process enables a client to pay a credit back by submitting TIP-3 tokens. The process must be paired with a *Borrow*.

The process may be interrupted if a payload operation is incorrect, and the tokens will be returned to the initial wallet. The process can be interrupted if there is a lock for the operation.


Token Operations
TIP-3
TIP-3 UserWallet
TIP-3 Wallet
WalletController
Invalid payload received Transferring tokens back
Transfer tip-3 tokens to user wallet through WalletController
MarketPlace and external Interface
MarketAggregator
Oracle
Functional Block
Repay
Token Operations
vToken
UserAccountManager
UserAccount
MarketPlace and external Interface
MarketAggregator
Token Operations
vToken
UserAccountManager
UserAccount
Update user account health
Transfer tip-3 tokens to user wallet
Transfer ever to user's msig

	From	To	Operation	Meaning	Error
1	*user's Msig*	*user TIP-3 wallet*	transfer request	Accept tokens	
2	*user TIP-3 wallet*	*TIP-3 WalletController wallet*	Token transfer	Register tokens in internal ledger	
3	*TIP-3 WalletController wallet*	*WalletController*	tokensReceivedCallback	Registration of operation	
4	*WalletController*	*MarketsAggregator*	performOperationWalletController	Inform market about arrival of new tokens	Invalid payload received Transferring tokens back Transfer tip-3 tokens to user wallet through WalletController
5	*MarketsAggregator*	*Oracle*	getAllTokenPrices	Request price of USD	
6	*Oracle*	*MarketsAggregator*	receiveAllUpdatedPrices	Supply price USD	
7	*MarketsAggregator*	*RepayModule*	performAction	price-to-market of the new tokens	
8	*RepayModule*	*UserAccountManager*	requestRepayInfo	Forward information to internal	

				token system	
9	*UserAccount Manager*	*UserAccount*	sendRepayInfo	Register operation in private ledger	
10	*UserAccount*	*UserAccount Manager*	receiveRepayInfo	Update private ledger of a client	
11	*UserAccount Manager*	*RepayModule*	repayLoan	Inform about registration of an operation	
12	*RepayModule*	*MarketsAggregator*	receiveCacheDelta	Update the pool	
13	*MarketsAggregator*	*RepayModule*	resumeOperation	Update the pool	
14	*RepayModule*	*UserAccount Manager*	writeRepayInformation	Inform that operation is finished	
15	*UserAccount Manager*	*UserAccount*	writeRepayInformation	To register it in a private ledger	
16	*UserAccount*	*Libraries of services*		Update user account health Tokens left: Transfer tip-3	

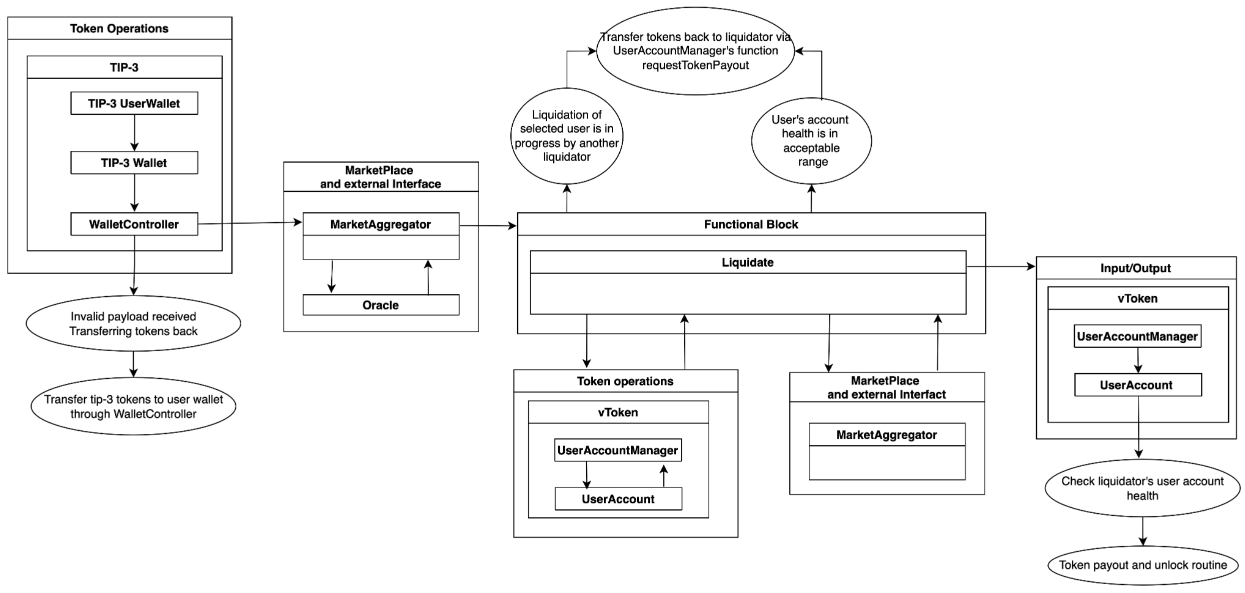
				tokens to user wallet Or Transfer ever to user's msig	

To pay back a credit, the *Client* must have TIP-3 tokens at the TIP-3 *UserWallet*. The tokens are priced-to-market by *MarketAggregator*, and then the *RepayModule* is called. No action is anticipated, if the USD value of tokens is not enough. Then the request is forwarded to *UserAccountManager* and *UserAccount* to register the request. Then the request is ready to be authorized by the *RepayModule*, which forwards the request to *MarketAggregator* to register in the public ledger. Then the request is allowed to be registered as paid in the private ledger of the client.

Liquidate

This process can be activated by a third-party agent (referenced as a "Liquidator" below, but also a member of the pool) to cover the insolvency of an agent, which can not (or does not) cover its financial liabilities, or, in another case, when a value of a collateral is not enough to cover credit. The Liquidator obtains the rest of internal tokens (*vTokens*) of the liquidated wallet with a multiplier.

After the liquidation, the person with covered debt is immediately unblocked and is entitled to perform all operations, contingent on his financial health.


Token Operations
TIP-3
TIP-3 UserWallet
TIP-3 Wallet
WalletController
Invalid payload received Transferring tokens back
Transfer tip-3 tokens to user wallet through WalletController
MarketPlace and external Interface
MarketAggregator
Oracle
Functional Block
Liquidate
Liquidation of selected user is in progress by another liquidator
Transfer tokens back to liquidator via UserAccountManager's function requestTokenPayout
User's account health is in acceptable range
Token operations
vToken
UserAccountManager
UserAccount
MarketPlace and external Interfact
MarketAggregator
Input/Output
vToken
UserAccountManager
UserAccount
Check liquidator's user account health
Token payout and unlock routine

	From	To	Operation	Meaning	Error
1	*msig*	*TIP-3 wallet*	transfer request		
2		*TIP-3WalletController wallet*	Token transfer		
3	*TIP-3WalletController wallet*	*WalletController*	tokensReceivedCallback		Invalid payload received Transferring tokens back Transfer tip-3 tokens to user wallet through WalletController
4	*WalletController*	*MarketsAggregator*	performOperationWalletC		

			ontroller		
5	*MarketsAggregator*	*Oracle*	getAllTokenPrices		
6	*Oracle*	*MarketsAggregator*	receiveAllUpdatedPrices		
7	*MarketsAggregator*	*LiquidateModule*	performAction	Transfer tokens back to liquidator via UserAccountManager's function requestTokenPayout	Liquidation of selected user is in progress by another liquidator
8	*LiquidateModule*	*UserAccountManager*	requestLiquidationInformation		
9	*UserAccountManager*	*UserAccount to liquidate*	requestLiquidationInformation		
10	*UserAccount to liquidate*	*UserAccountManager*	receiveLiquidationInformation		
11	*UserAccountManager*	*LiquidateModule*	liquidate		
12	*LiquidateModule*	*MarketsAggregator*	receiveCacheDelta	Transfer tokens back to liquidator via UserAccountManager's function requestTokenPayout	User's account health is in acceptable range
13	*MarketsAggregator*	*LiquidateModule*	resumeOperation		

14	*LiquidateModule*	*UserAccount Manager*	seizeTokens	Block tokens	
15	*UserAccount Manager*	*UserAccount to liquidate*	liquidateVTokens	Cancel native tokens	
16	*UserAccount to liquidate*	*UserAccount Manager*	grantVTokens		
17	*UserAccount Manager*	*liquidator's UserAccount*	grantVTokens	Transfer tokens back to liquidator via UserAccount Manager's function requestTokenPayout	User's account health is in acceptable range
18	*liquidator's UserAccount*	libraries		Check liquidator's user account health Token payout and unlock routine	

The process can be interrupted in the following cases.

- It must be interrupted if a payload operation is incorrect. Then tokens are to be returned to the initial wallet.
- If a user is liquidated at the moment by somebody else. A sum is returned to the Liquidator.
- Financial state of a user is adequate and liquidation is not required any more.
- Only one liquidation per period is allowed for a Liquidator.

A Client must have a payload to start an operation. Then tokens are received by the *WalletController* contract, and forwarded to The *MarketAggregator.* It prices-to-market the tokens it receives, records the operation into the public ledger and initializes the *LiquidationModule*. It makes settlements with a preliminary request to the *UserAccount* contract. Then internal tokens, i.e. the virtual vTokens, are confiscated from a user, who is unable to cover debt. The tokens are transferred to the liquidator. Formally, this is the end of the process, and few supplementary library operations are left.

Then the someone whose debt is uncovered obtains unlock after the debt is covered. One needs to calculate the residual transferred to the liquidator. Formally, this is the end of the process, and few supplementary operations are left. The someone whose debt is uncovered is unlocked, as the debt is covered. One needs to calculate a residual sum, transfer it to the liquidator and to make appropriate records in ledgers.

At this point a fork of operations is required. First, one needs to unblock the user whose debt was covered. Second, the residual sum must be returned to the liquidator.

The process is the most complicated. It controls two user wallets, one of a client, and another for the third party.

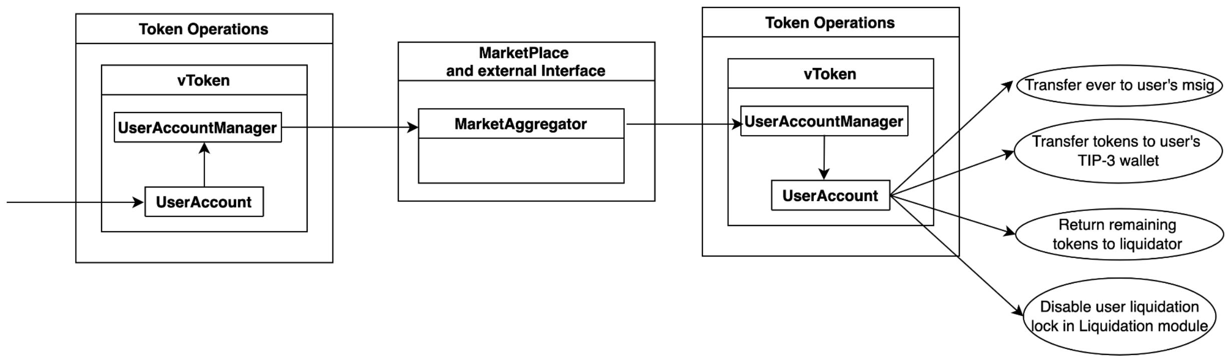
Supplementary processes

Financial health service

This service checks and controls the financial status of clients. It obtains data on the client's vTokens from private leverage in *UserAccount* and *UserAccountManager* and forwards to *MarketAggregator*. *MarketAggregator* receives data, takes several market indicators, forwards back the calculated financial health to the *UserAccountManage* and *UserAccount*. Then information on instant financial health is recorded and is ready for further usage.

Then few library outcomes are possible.

- Transfer ever to the user's Multisig.
- Transfer tokens to the user's TIP-3 wallet.
- Return remaining tokens to the liquidator.
- Disable user liquidation lock in the *Liquidation* module.

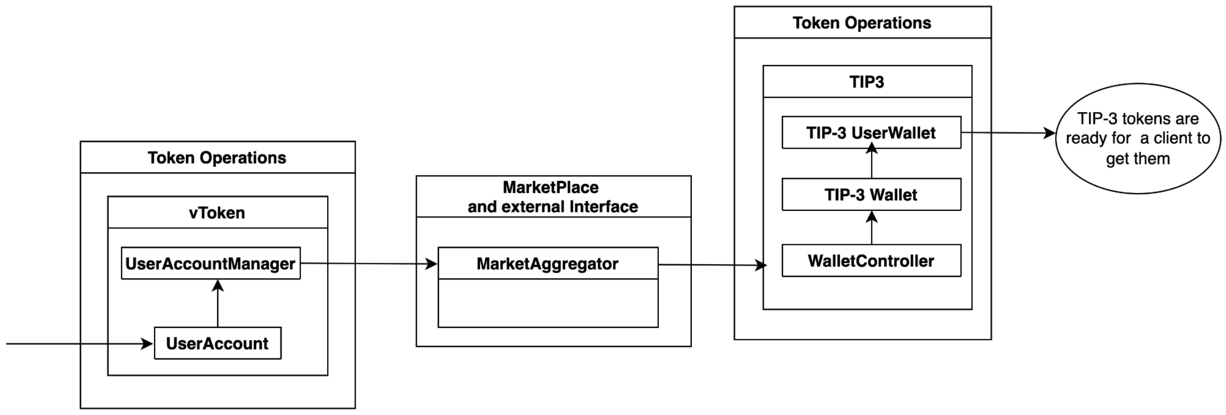

Token Operations
vToken
UserAccountManager
UserAccount
MarketPlace and external Interface
MarketAggregator
Token Operations
vToken
UserAccountManager
UserAccount
Transfer ever to user's msig
Transfer tokens to user's TIP-3 wallet
Return remaining tokens to liquidator
Disable user liquidation lock in Liquidation module

	From	To	Operation	Meaning
1	*Some client service*	*UserAccount*	Start evaluation of financial health	Start evaluation of financial health
2	*UserAccount*	*UserAccount Manager*	calculateUserAccountHealth	Data preparations
3	*UserAccount Manager*	*MarketsAggregator*	calculateUserAccountHealth	Calculation of client's financial health
4	*MarketsAggregator*	*UserAccount Manager*	updateUserAccountHealth	Update info in private ledger on financial health.
5	*UserAccount Manager*	*UserAccount*	updateUserAccountHealth	Finalize: Transfer ever to user's msig Need balance Token withdraw required.

				Liquidation finalizing. . Return remaining tokens to liquidator Disable user liquidation lock in Liquidation module.

Token Withdrawal service

This service controls and registers a transfer of TIP-3 tokens, if a withdrawal is completely validated. This service is conceptually independent from borrowing.



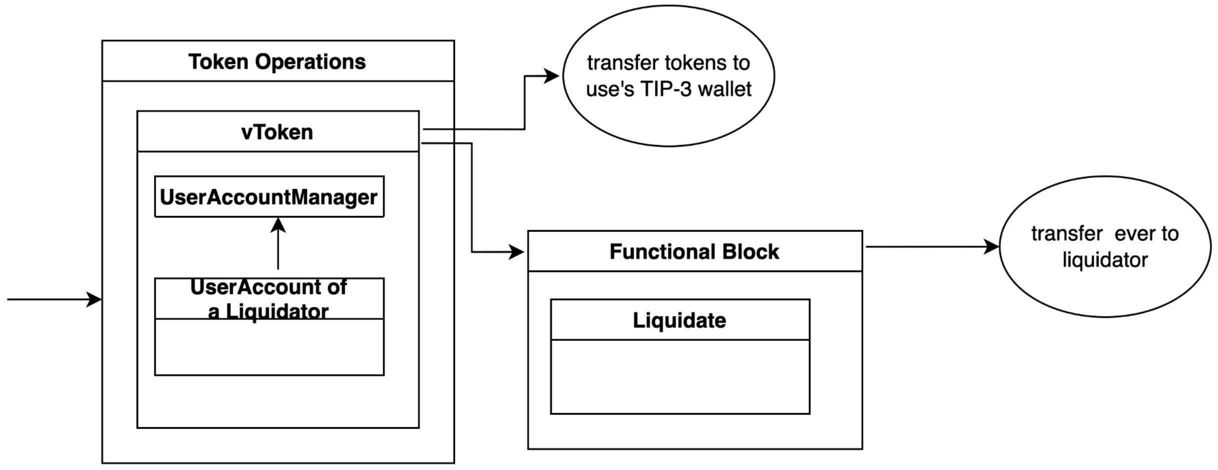
	From	To	Operation	Meaning
1	*Client service*	*UserAccount*		Start to free TIP-3 tokens to a client
2	*UserAccount*	*UserAccount Manager*	requestTokenPayout	Request a permission for operation
3	*UserAccount Manager*	*MarketsAggregator*	requestTokenPayout	Provide a permission with necessary requisites

4	*MarketsAggregator*	*WalletController*	transferTokensToWallet	Forward freed tokens to a bookkeeper
5	*WalletController*	*TIP-3 WalletController wallet*	transfer	Transfer tokens to a wallet of a client
6	*TIP-3 WalletController wallet*	*user TIP-3 wallet*	accept transfer	Register that tokens have arrived
7	*user TIP-3 wallet*	*user's Msig*	return ever to user's msig	Inform a client that TIP-3 tokens are available

If a withdrawal request is completely validated, then a client can obtain tokens. This request is forwarded to *MarketAggregator*, which makes a record, changes the pool, then operates the *UserAccountManager*, which allows the *WalletController* to free TIP-3 tokens for the client.

Unlock service

Currently this service finalizes only the liquidation process. The service aims to prevent operations of an agent whose financial health does not satisfy some criteria. Such an operation may be required in other cases too. Current realization does not concern a borrowing history of a client. It automatically unlocks borrowing after a debt is liquidated.


Token Operations
vToken
UserAccountManager
UserAccount of a Liquidator
transfer tokens to use's TIP-3 wallet
Functional Block
Liquidate
transfer ever to liquidator

A request to unlock the user is forwarded from the *UserAccount* to the *UserAccountManager*. Then the *UserAccountManager* forwards a request to the liquidation Module. The module performs two operations. It transfers EVER tokens to a liquidator as a compensation for the liquidation

	From	To	Operation	Meaning
1	*liquidator's UserAccount*	*UserAccount Manager*	returnAndSupply	Receive a request to transfer
2	*UserAccount Manager*	*LiquidateModu le*	unlock	transfer *TIP-3 Wallet* to an unsuccessful borrower
3	*LiquidateModule*	*LiquidateModu le*	Transfer ever to liquidator's msig	Reimburse ever tokens to msig of liquidator
	LiquidateModule	*liquidator's Msig*	Transfer ever to liquidator's msig	Finalize the reimbursement

Short contract description

MarketsAggregator

The contract securely holds the public market ledger of all operations and communicates with the external interface to obtain price of tokens in terms of USD. Pricing to market depends on the instant exchange rate.

Oracle

It’s an interface between contracts and external pricing data suppliers.

UserAccountManager

It performs two operations. First, it deploys and updates user’s accounts using an existing account of a user. Second, it serves as a two-way request transmitter between *MarketsAggregator* and modules to the contract *UserAccount*. It contains many one-liners type get/forward parameters. This function centralizes control of the data traffic.

UserAccount

This function holds a ledger of a client, what and where it holds, and operations. It can be communicated only with the *UserAccountManager* contract that works as a dispatcher.

WalletController

It receives and sends TIP-3 tokens, initializes information from payloads, enriches it if necessary and forwards it to *MarketAggregator*. Only this contract type has TIP-3 wallets.

Section II. Functional properties

II.1. Generic properties

Numbers

All the numbers are either integers or common fractions (that are represented as $\frac{numerator}{denominator}$, where *numerator* and *denominator* are integers). The following properties for all the numbers are in place:

`NUM.1`	All the integers are not negative
`NUM.2`	All the numerators are not negative
`NUM.3`	All the denominators are positive
`NUM.4`	All the numerators and denominators are relative primes
`NUM.5`	All the common fractions converted to integer are converted to the closest integer equal or below the fraction

The arithmetic operations related to common fractions are identical to the common arithmetics.

Roles

All the users can be splitted into four categories:

- *Owner* - the person who created the contract system
- *Upgrader* - the person who can upgrade the contract system
- *Tuner* - the person who can change the parameters.
- All other users

The following properties are in place.

`ROL.1`	*Owner* can be changed by *Owner* only
`ROL.2`	*Upgaders* and *Tuners* can be changed by *Owner* only
`ROL.3`	All the changes can be made exclusively by specially designated messages, not having any side effects
`ROL.4`	*Owner* is both an *Updater* and a *Tuner*
`ROL.5`	All the contracts can be upgraded. The upgrade can be initiated by *Upgrader* only, keeping all the state parameters in place

Financial operations

Interest Rate

The interest rate calculations are performed against *Markets* described below. The following statements must be correct:

- The calculation can be performed only when (otherwise, it does nothing):
  - `realTokenBalance` is positive
  - `totalBorrowed` is positive
- `borrowRate` is $baseRate + \frac{totalBorrowed \cdot utilizationMultiplier}{totalBorrowed + realTokenBalance}$
- `dt` is the time passed between NOW and `lastUpdateTime`, in seconds.
- `simpleInterestFactor` is $dt \cdot borrowRate$
- `index` is $index \cdot (1 + simpleInterestFactor)$
- `totalBorrowed` is $totalBorrowed \cdot (1 + simpleInterestFactor)$
- `totalReserves` is
  $totalReserves + reserveFactor \cdot totalBorrowed \cdot simpleInterestFactor$

Exchange rate

Exchange rate is calculated for Markets described below according to the following formula: $\frac{currentPoolBalance + totalBorrowed - totalReserve}{vTokenSupply}$

User health

If user health is more than *one* the user is considered as healthy, and not healthy otherwise. When the user's health is calculated the following actions are performed:

- For all the Markets the Interest rate and Exchange rate stories are applied before the calculation.
- The User health is calculated as $\frac{supplyFactor}{borrowFactor}$, where:
  - $supplyFactor = \sum_{markets} \frac{suppliedTokens \cdot exchangeRate \cdot collateralFactor}{tokenPrice}$
  - $borrowFactor = \sum_{markets} \frac{borrowedTokens \cdot marketIndex}{borrowIndex}$
- If the *User* is not healthy, a *LiquidationPossible* event must be emitted.
- For the *User*:
  - If the *User* health is less than one `borrowLock` and `liquidationLock` are `TRUE`, and `FALSE` otherwise.
  - `accountHealth` is the calculated account health.
  - All the markets are updated with marketUpdate story where indices are taken from *marketIndex*.

II.II. Oracle

The *Oracle* contract provides current exchange rates between the set of tokens and USD. The rates can be updated both externally (by the call of the external rate supplier) or internally (when the *Oracle* requires the rates from the external rate supplier). The detailed properties are described below.

Token addition or removal

`ORT.1`	Tokens can be added or removed by *Tuner* only.
`ORT.2`	Upon token addition the internal rate update request must be called.

Rate update

`ORU.1`	External rate update can be initiated by rate information provider only.
`ORU.2`	Internal rate update can be initiated by anybody.

II.III. UserAccount

State

`owner`	Multisig wallet that owns the account.
`borrowLock`	`TRUE` if the user has an loan being handled or financially unhealthy.
`liquidationLock`	`TRUE` if the user is financially unhealthy.
`userAccountManager`	Manager of user accounts.
`accountHealth`	Financial health of the user.
`suppliedTokens`	Tokens supplied for each market
`borrowedTokens`	Tokens borrowed for each market
`borrowIndex`	Index of the market where borrowed

Stories

marketUpdate

For the provided market:

- `borrowedTokens` become $borrowedTokens \cdot \frac{provided\ index}{borrowIndex}$
- `borrowIndex` becomes a provided new index

Initialization

Upon creation of the new user, the following properties are in place.

`UAI.1`	● `borrowLock` and `liquidationLock` are `FALSE` ● `userAccountManager` is the creator of the contract ● `owner` is the supplied parameter ● `accountHealth` is arbitrary ● `suppliedTokens` is empty
`UAI.2`	`owner` and `userAccountManager` are never changed

Supply

`UAS.1`	Can be initiated by `userAccountManager` only.
`UAS.2`	`suppliedTokens` for the given market are increased by the provided parameter.
`UAS.3`	`borrowedTokens` and `borrowIndex` are modified according to the marketUpdate story.
`UAS.4`	`accountHealth`, `borrowLock` and `accountHealth` are updated according to User health story.

Withdraw

`UAW.1`	Can be initiated by `owner` only
`UAW.2`	Does nothing if `liquidationLock` or `suppliedTokens` for the specified market are less than the required amount
`UAW.3`	The operation itself is performed by *WithdrawModule*
`UAW.4`	Upon completion: ● `suppliedTokens` for the given market is decreased by the provided number of tokens to withdrawn ● `accountHealth` is updated according to the corresponding story ● Tokens to withdraw must be returned to the provided TIP-3 wallet

Borrow

`UAB.1`	Can be initiated by `owner` only
`UAB.2`	If `borrowLock` or `liquidationLock` or `accountHealth` less or equals than `one` then does nothing
`UAB.3`	The operation itself is performed by BorrowModule
`UAB.4`	`borrowLock` becomes `TRUE` before coming to *BorrowModule*
`UAB.5`	After the operation: ● `borrowLock` becomes `FALSE` ● marketUpdate story for the specified market is executed according to its specification ● Borrowed tokens are transferred to the specified TIP-3 wallet ● `borrowedTokens` are increased by the specified number of borrowed tokens ● `accountHealth` is updated according to the corresponding story

Repay

`UAR.1`	Can be initiated by `userAccountManager` only
`UAR.2`	The operation itself is performed by RepayModule
`UAR.3`	Upon completion: ● `borrowedTokens` and `borrowIndex` are filled with the number arrived from RepayModule ● Repaid tokens are transferred to the specified TIP-3 wallet ● `accountHealth` is updated according to the corresponding story

Liquidation

`UAL.1`	Can be initiated by `userAccountManager` only
`UAL.2`	Before the operation `suppliedTokens` for the specified market of “donor” *User* is decreased by the specified number
`UAL.3`	The operation itself is performed by LiquidationModule
`UAL.4`	In case of liquidation abort all the tokens are returned back to the “donor” User
`UAL.5`	Upon completion: ● `suppliedTokens` for the specified market of “recipient” *User* is increased by the specified number ● `accountHealth` is updated according to the corresponding story ● Repay is forcibly performed

II.IV. UserAccountManager

It’s an intermediate contract that does not perform any calculations and does not keep any state, so all the properties are related to the access.

`UAM.1`	New *User* can be created by anybody

`UAM.2`	Supply update for the *User* can be requested by SupplyModule only
`UAM.3`	• Withdraw request must come from the *User* • Withdraw update must come from the WithdrawModule
`UAM.4`	• Borrow request must come from the *User* • Borrow update must come from the WithdrawModule
`UAM.5`	Repay update for the *User* can be requested by RepayModule only
`UAM.6`	• Liquidate request must come from the LiquidationModule • Grant request must come from the “donor” *User* and then resent to the “recipient” *User* • Repay request must come from the “recipient” *User*

II.V. MarketAggregator

State

State of the *MarketAggregator* is defined by the following variables.

`userAccountManager`	Manager of user accounts
`walletController`	Wallet controller
`oracle`	Oracle address
`markets`	The list of created markets
`tokenPrices`	The list of externally found prices for each token as a common fraction
`modules`	The list of attached *Module* contracts

Each market is defined by the following variables:

`token`	Address of the TIP-3 root token
`realTokenBalance`	The cumulative amount of token entered into this

	market
`vTokenBalance`	The cumulative amount of internal tokens issued for this market
`totalBorrowed`	The overall amount of the borrowed tokens
`totalReserved`	The overall amount of reserved tokens
`index`	Internal calculation factor
`baseRate`	Base interest rate
`utilizationMutlipl ier`	Aux parameter used to calculate interest rate
`reserveFactor`	Currently unused
`exchangeRate`	Exchange rate of TIP-3 tokens for USD
`collateralFactor`	Coverage ratio for a credit.
`liquidationMultipl ier`	Multiplicative bonus for a liquidator
`lastUpdateTime`	Timestamp when the market information was updated

Generic properties

For all the operations but initialization and code update the following properties must be in place.

`MAG.1`	Code may be changed only by *Updater.*
`MAG.2`	`contractCodeVersion` is the same.
`MAG.3`	`userAccountManager` is the same, but a special no-side-effect-method called by a *Tuner.*

`MAG.4`	`walletController` is the same, but a special no-side-effect-method called by a *Tuner.*
`MAG.5`	`oracle` is the same, but a special no-side-effect-method called by a *Tuner.*

Initialization

Upon the initialization of the *MarketAggregator* the following properties are in place:

`MAI.1`	Owner is defined as an initialization parameter
`MAI.2`	All the other state variables are empty

Market creation

While the new market is created the following properties are in place:

`MAC.1`	`markets` are added by a new market
`MAC.2`	New market has the following properties: ● `token` - introduced as a call parameter ● `realTokenBalance` - 0 ● `vTokenBalance` - 0 ● `totalBorrowed` - 0 ● `totalReserve` - 0 ● `index` - 1/1[3] ● `baseRate` - introduced as a call parameter ● `utilizationMultiplier` - introduced as a call parameter ● `reserveFactor` - introduced as a call parameter

[3] Here and later common fractions are represented in the form `numerator/denominator`

	• `exchangeRate` - introduced as a call parameter • `collateralFactor` - introduced as a call parameter • `liquidationMultiplier` - introduced as a call parameter • `lastUpdateTime` - must be NOW[4]
`MAC.3`	New market can be created by *Tuner* only.
`MAC.4`	`marketId` must be introduced as as call parameter.
`MAC.5`	In case the market with the specified id exists the operation must fail and all the extra money must be returned back.
`MAC.6`	In case of success MarketCreated event must be emitted with: • `marketId` • created market as a structure

Market update

Any market can be updated with the following properties in place.

`MAU.1`	All the extra money must be returned.
`MAU.2`	If the market with the supplied `marketId` does not exist the operation must fail.
`MAU.3`	The operation can be performed by *Tuner* only.
`MAU.4`	The following variables of the market are updated: • `baseRate` - changed as a call parameter • `utilizationModifier` - changed as a call parameter • `reserveFactor` - changed as a call parameter • `collateralFactor` - changed as a call parameter • `liquidationMultiplier` - changed as a call parameter • `exchangeRate` - if `vTokenBalance` is zero then change as a call parameter, otherwise no action • `lastUpdateTime` - must be NOW • Other variables follow interestRate story • `exchangeRate` is calculated according to Exchange Rate story or

[4] As a system TVM function

	stays the same if `vTokenBalance` is the same

Market removal

Any market can be removed with the following properties in place.

`MAR.1`	All the extra money must be returned.
`MAR.2`	If the market with the supplied `marketId` does not exist the operation must fail.
`MAR.3`	The operation can be performed by *Tuner* only.
`MAR.4`	*MarketDeleted* event must be emitted.

Modules addition and removal

Modules (such as *Repay*, for example, can be dynamically added or removed) with the following properties in place:

`MAM.1`	All the extra money must be returned.
`MAM.2`	The operation can be performed by *Tuner* only.

Operation management

While the operations are performed by the attached *Modules*, the *MarketAggregator* handles the overall management and dispatching.

MAO.1	Operation performing can be initiated by *WalletController* or *UserAccountManager* only.
MAO.2	All the prices must be updated from *Oracle* before the operation.
MAO.3	Real action must be transferred to the corresponding module if it exists.

Miscellaneous

MAD.1	Forcible rate update can be initiated by the `owner` only.
MAD.2	`userAccountManager`, `walletController` and `oracle` can be changed by *Tuner* only.

II.VI. SupplyModule

State

`userAccountManager`	Manager of user accounts.
`marketAggregator`	MarketAggregator

Parameter update

SUU.1	`userAccountManager` and `marketAggregator` can be altered by Tuner only.

Supply action

`SUA.1`	Operation can be initiated by `walletController` of the `marketAggregator` only.
`SUA.2`	• For the given market (in `marketAggregator`): ◦ tokens to provide is calculated as $\frac{tokenAmount}{exchangeRate}$ ◦ `realTokenBalance` is increased by the given token amount ◦ `vTokenBalance` is increased by tokens to provide • `TokenSupplied` event must be emitted • The User corresponding to the given wallet is instructed to update his data as described here with tokens to provide (via UserAccountManager)

II.VII. WithdrawModule

State

userAccountManager	Manager of user accounts
marketAggregator	MarketAggregator

Parameter update

WUU.1	userAccountManager and marketAggregator can be altered by Tuner only.

Withdraw action

WIA.1	Operation can be initiated by User (via UserAccountManager) only
WIA.2	• User health is recalculated according to the specified story and *User* is instructed to update it • If the *User* is healthy and wants to withdraw less or equal tokens than he has then for the specified market (in marketAggregator) : ◦ tokens to send is calculated as $tokensToWithdraw \cdot exchangeRate$

	○ tokens to send in USD is calculated as $\frac{tokensToSend}{tokenPrice}$ ○ As described in the user health story the health can be represented as $\frac{supplyFactor}{borrowFactor}$. If supplyFactor subtracted by borrowFactor is less than tokens to send in USD do nothing ○ If tokens to send is more than realTokenBalance subtracted by totalReserve do nothing ○ realTokenBalance decreased by tokens to send ○ vTokenBalance is decreased by *tokensToWithdraw* (provided parameter) ○ *TokenWithdraw* event must be emitted ○ The User corresponding to the given wallet is instructed to update his data as described here with *tokensToWithdraw* and tokens to send (via UserAccountManager)

II.VIII. BorrowModule

State

userAccountManager	Manager of user accounts
marketAggregator	MarketAggregator

Parameter update

BOU.1	userAccountManager and marketAggregator can be altered by Tuner only

Borrow action

`BOA.1`	Operation can be initiated by User (via UserAccountManager) only
`BOA.2`	● In case *tokensToBorrow* (as a provided parameter) is greater than `realTokenBalance` subtracted by `totalReserve` (here and later, for the given market, in `marketAggregator`) do nothing

	• In all other cases, user health is recalculated according to the specified story and *User* is instructed to update it • If the *User* is unhealthy it's instructed to release `borrowLock` • If the *User* is healthy: ○ As described in the user health story the health can be represented as $\frac{supplyFactor}{borrowFactor}$. If $\frac{supplyFactor - borrowFactor}{tokenPrice} < tokensToBorrow$ (where *tokensToBorrow* is a provided parameter) do nothing ○ `totalBorrowed` is increased by *tokensToBorrow* ○ `realTokenBalance` is decreased by *tokensToBorrow* ○ *TokenBorrow* event must be emitted ○ The *User* must be instructed (via *UserAccountManager*) to complete the process as described here with the provided native and TIP-3 wallets as well as *tokensToBorrow*

II.IX. RepayModule

State

userAccountManager	Manager of user accounts
marketAggregator	MarketAggregator

Parameter update

REU.1	userAccountManager and marketAggregator can be altered by Tuner only

Repay action

`REA.1`	Operation can be initiated by `walletController` of the `marketAggregator` only

REA.2	• Tokens to repay (here and later, for the specified market in marketAggregator, as well as for the specified User) are calculated as: $\frac{tokensBorrowed \cdot index}{borrowIndex}$ • Tokens to return is the maximum of *0* and tokens to repay subtracted by *tokensForRepay* (the latter is the specified parameter) • *tokensBorrowed* is the maximum of *0* and *tokensForRepay* subtracted by tokens to repay • *borrowIndex* is `index` • *tokenDelta* is minimum of tokens to repay and *tokensForRepay* • `totalBorrowed` is decreased by *tokenDelta* • `realTokenBalance` is increased by *tokenDelta* • *RepayBorrow* event must be emitted • The *User* must be instructed (via *UserAccountManager*) to complete the process as described here with the provided native and TIP-3 wallets as well as tokens to return, *tokensBorrowed* and *borrowIndex*

II.X. LiquidationModule

State

userAccountManager	Manager of user accounts
marketAggregator	MarketAggregator

Parameter update

LIU.1	`userAccountManager` and `marketAggregator` can be altered by Tuner only

Liquidator action

`LIA.1`	Operation can be initiated by `walletController` of the `marketAggregator` only
`LIA.2`	● (here and later, the variables are provided for the specified market in `marketAggregator`, as well as, for the specified User) tokens to liquidate is minimum of *tokensBorrowed* and *tokensProvided* (both are the specified parameters) ● User health is recalculated according to the specified story and *User* is instructed to update it ● In case the *User* is healthy instruct the *User* to abort the liquidation and stop ● Tokens to liquidate is $min(tokensBorrowed,\ tokensProvided)$ (the latter is the supplied parameter) ● Tokens to return is *tokensProvided* subtracted by tokens to liquidate ● *fTokensToLiquidateUSD* is $\frac{tokensToLiquidate \cdot liquidationMultiplier}{tokenPrice}$ ● *fvTokensCollateralUSD* (here *marketToLiquidate* is used, that is the specified parameter) is $\frac{supplyTokens \cdot exchangeRate}{tokenPrice}$ ● For the specified market: ○ `totalBorrowed` must be decreased by tokens to liquidate ○ `realTokenBalance` must be increased by tokens to liquidate ● If *fvTokensCollateralUSD* is less than *fTokensToLiquidateUSD* then for the specified market to liquidate: ○ Tokens from reserve is $(fvTokensCollateralUSD - fTokensToLiquidateUSD) \cdot tokenPrice$ ○ If tokens from reserve is greater than `tolalReserve` the *User* is instructed to abort the liquidation ○ For the market to liquidate: ■ `totalReserve` is decreased by tokens from reserve ● Tokens to seize are (market to liquidate parameters are used) calculated as: $\frac{min(fvTokensCollateralUSD,\ fTokensToLiquidateUSD) \cdot tokenPrice}{exchangeRate}$ ● `borrowedTokens` (for the *User*) is `tokensBorrowed` minus tokens to liquidate ● `borrowIndex` (for the *User*) is `index` ● The *User* must be instructed (via *UserAccountManager*) to update its state completing the liquidation

II.XI. WalletController

`WAC.1`	Configuration, such as `marketAggregator`, as well as a list of markets can

	be altered by *Tuner* only
`WAC.2`	Payment to the user TIP-3 wallets can be initiated by MarketAggregator only
`WAC.3`	Upon token receival one of the following operations can be automatically initiated, depending on payload: Supply, Repay or Liquidation. The operation itself is performed by the corresponding module.

Section III. Possible issues

III.1. Business-level issues[5]

A loan is an intertemporal exchange, which always has a time interval between a start and a final termination of a deal. At the moment of a loan expiration a borrower must have a motivation to pay back the loan, a fee. This is the standard framework of any traditional lending/credit system. There are few basic concerns for the current implementation.

- The data structure *MarketInfo* does not supply enough information to control the maturity of a loan. Existing data structure fits the needs of instant operations, like *Supply* and *Withdraw*, but not those of *Borrow*, *Repay* and *Liquidate*, which process intertemporal operations. To support maturity of loans, one needs to add another time stamp for a termination time of a loan, and a state variable for a loan.
- There is no state variable of a loan, be it not-paid back, paid or defaulted with expired maturity. *MarketInfo* data structure should have a variable to control the status of a loan, and register cummulative debts overdue.
- It is impossible to control overdue debts, for individual clients and for the system. This can be done with a debt status variable.
- Pricing-to-market collateral is instant pricing, which is vulnerable to the volatility of cryptomarkets.

[5] The present analysis was done by PhD in Economics(both Russian and EU(Ca Foscari, Venezia)), expert in microfoundations of macro and monetary economics as well as in game theory. The findings can be disputable by the community and require additional non-technical sessions with interessants.

- Participants can not know a deadline to repay.
- Credit risk is beyond any control in the system.
- *borrowRate* does not depend on debts overdue. Their accumulation disturbes all variables, where *borrowRate* is used.
- There is information asymmetry between a borrower and a creditor. Higher interest rates will attract more clients with riskier projects, who are usually considered as less reliable loan/credit payers.
- A creditor is not protected from asymmetric information actions of borrowers.
- A collateral has any value in items of a real (tangible) world beyond the system. There is any outside enforcement to support a credit discipline. There is much experience that an outside enforcement is important for long-run sustainability of a credit system.
- If a maturity is not controlled, it is not clear when the service Liquidate should be activated.

Loan market regulation concerns

The system has control variables, analogous to regulation of a central bank.

Assignment of values for *collateralFactor* - reservation ratio for loans, *liquidationMultiplier* and other exogenous variables is not transparent. *Every parameter* from the list of exogenous parameters has an impact on motivations of members of the system. Due to community property of the system and tradition of the crypto society clients must have clear information and trust not only to a technical side how the system operates, but also to a Regulator of the parameters. Changing and updating these parameters is similar to bank regulation from a Central bank. Thus an Owner takes high responsibility for assigning these issues.
If a decision to update the parameters is made by a collective decision, then an outcome can be similar to the Prisoners' Dilemma: a result is approved by everybody, but it damages the system and everybody. One of the possible mechanisms comes from a contradiction in interests between savers and borrowers, which both participate in the decision making committee.

Changing concerns

Contemporary financial systems are sensitive not only on values of parameters, but also on frequency of changing parameters and discussion of possible future changes,[6] tick size of a change,[7] words in announcement about changes.[8] At the moment one can say nothing about how sustainably the system will operate, but it is possible to claim that the current variant is vulnerable to fluctuations, coming from operations at the market itself.

Collateral concerns

The EverscaleLend has a parameter collateralFactor, which is supposed to regulate coverage of issued loans. The documents say nothing about how its value is chosen, how and when to update it. It follows from the documents that this variable can be updated only by an Owner or an agent, who has obtained a right for the operation from the Owner.

The presented version of the system follows the contemporary financial architecture, established by the Act of the Bank of England, 1844. It prohibited banks from issuing bills, but allowed credits with incomplete coverage, i.e. coverage with not complete insurance of credits.

One may definitely say that a reduction of *collateralFactor* below one may induce a credit boom, with later delinquency problems, as we can observe nowadays worldwide. A delinquency is followed by an activation of the *Liquidation* process. However, one can not guarantee that if there is an increase in the number of clients with inadequate financial health, there will be an increase in volunteer liquidators to save defaulted members of the pool, even if there is a bonus for a liquidation. Either each liquidator will have to spend more and to earn from a liquidation, or there will be a mass delinquency. However, there are some problems, outlined in Subsection Liquidation concerns. Mass liquidation may result in reduction of activities, which will deteriorate the reputation of the system.

[6] https://www.emerald.com/insight/content/doi/10.1108/CFRI-07-2018-0068/full/html
[7] https://link.springer.com/article/10.1007/s11156-010-0171-6
[8] https://www.sciencedirect.com/science/article/abs/pii/S0378426617302017

Collective decision making of the members of the community depends on credit and borrowing activity of the members, which have opposite interests.

Financial health concerns

Instant pricing to market. Financial health depends on instant market data. Thus the financial health of **any** collateral may instantly become inadequate, not only for a currently processed client. This means that the system needs continuous monitoring of values of all collaterals, transparent margin call system and an implementable protocol of actions for such cases.

Volatility of financial health. Cryptomarkets are very volatile and are subjects for manipulations. The fluctuations can induce transitory states of inedequate financial health, which is a way to misapplication of *LiquidateModule*.

Comparability of values. Construction of financial value follows movements of involved markets. This means that one can not compare values of financial health at different moments of time, as the values are constructed relative to an instance state of a market, it can not serve as a benchmark.

No debt overdue. Construction of the values, entering the financial health variable, do not depend on debts overdue.

Liquidations concerns

The *Liquidation* client service does not have a clear activation protocol. It may be activated in two cases.

- A debt is overdue.
- Collateral for a debt is not enough to cover the debt.

Both cases have the same problems. Either there must exist an automatic procedure to generate a margin call to a client, to sell the collateral with a risk of crashing the market. Or a procedure to inform and involve relevant clients (liquidators) to cover the debt. If the procedure of assigning a liquidator is not transparent, this is a way of misapplication.

These interruptions are not covered in the list of interruptions for *LiquidateModule*.

- A liquidator has enough funds, but a collateral value in USD is not enough to become a liquidator, and the operation must be blocked for such clients.
- Liquidation process can damage the financial health of a liquidator to pay back its own loan.

Liquidator is assumed to have a bonus for the operation, defined by the *liquidationMultiplier* parameter. This facility opens a way minimum for two misapplications.

Unlock. It is not clear why the unlock operation always follows after a liquidation and does not depend on the repayment discipline of an unsuccessful borrower. Credit history does not exist.

Token pump. If *liquidationMultiplier* is greater than one, then one may seek an arbitrage opportunity to use the operation as a money pump. There are no restrictions to have a beyond system collusion between a borrower(s) and a liquidator, one takes a credit, another earns on liquidation of delinquency for this credit. Realization depends on the current state of the Everscalend marketplace, costs of operations and the current economic environment around. Probably this option can be closed by properly adjusted costs. From another side, costs are bounded by external factors. Similar operations can be performed by combinations of pumping and dumping a market.

Raider seizure. If a multiplier for liquidation is more than one, i.e. there is a bonus for the operation, then one may think about liquidation as a source to increase a stock of native tokens. Probably this option can be closed by properly adjusted costs.

Multimarket issues

At the moment the system is expected to become multimarket, with borrowing in many tokens. This will offer clients new arbitrage opportunities between the markets within the system. For example, borrowing in one token to pay loans at another market. Thus the addition of a new market becomes not a technical exercise, but a problem of sustainability of the whole system. Multimarket operations will not allow markets to stay independent any more.

III.II. Technical-level issues

While the code audit was not a part of the present activity, the team found a few issues to be presented.

Place	Severity	Description
The whole project	MAJOR	It's highly discouraged to use mapping for keeping data that can lead to expensive transactions and even to incorrect functionality. Instead it's encouraged to use the values as child contracts, thus preventing long structures in memory.
BorrowModule	MINOR	The requirement to ask for less tokens than the total amount subtracted by reserve looks too strong. Probably it's better to replace it less or equal.
RepayModule	NOTE	Unnecessary initialization of tokensToRepay, it's unconditionally overridden later.

III.III. Possible threats and attacks

The following potential threats and attacks are to be considered at the later stages of the formal verification:

- Unauthorized access
- Breaking the synchronization of the internal variables

- Moving the internal parameters into an incorrect state (such as negative values)
- Meeting the business risks
- Incorrect recalculation of the internal variables
- Unlifted locks