Everscalend functional specification

By OCamlPro

March 15, 2022

Contents

Chapter 1

Introduction

This document contains a functional specification of the "Everscalend" smart contracts. The source code is available at `https://github.com/SVOIcom/everscalend-contracts`, commit `8d24e268f9c44bd3e896fb6a28bbf8a42c7027a9`. This work is provided as a submission to the Formal Methods Sub-Governance Contest #39.

Chapter 2

High-level system description

2.1 System purpose

Everscalend is a DEFI (DEcentralized FInance) lending and borrowing system implemented on the Everscale Blockchain. It's main purpose is to provide Everscale users with a realiable way to lend and borrow cryptocurrency tokens. It makes it possible for users to generate profits on tokens they supply for lending and to acquire tokens by borrowing them instead of buying them.

2.2 Terms of the system domain

Term	**Definition**
Interest rate	The rate of profit that the lenders get when tokens are borrowed. The interest rates in Everscalend are algorithmically calculated and they increase when the borrowing demand increases and decrease when it does.
Reserve factor	Ranging from 0 to 1, it represents the portion of the interest rate that should be stored in the reserve whenever it is acquired.
Market	A pool of tokens of the same kind where all the tokens supplied by the users are stored. It also holds information like the exchange rate and reserve factor etc.

Term	**Definition**
vToken	A virtual token, it is used to represent the amount of tokens that a supplier owes the market and they get them whenever they supply tokens to the markets. There is a type of vToken for each type of token. When interest rate accumulates on the supplied tokens in the markets, the exchange rate from the vTokens to the real tokens increases, so when a supplier holding vTokens wishes to withdraw. They will be able to withdraw more than they supplied, the difference representing the interest that was accumulated on their supplied tokens.
Collateral	The amount of vTokens that a borrower has to have in order to be able to borrow a certain amount of some tokens.
Collateral factor	Ranging from 0 to 1, it represents the amount of tokens that can be borrowed for a collateral. e.g. a collateral factor of 0.9 allows the borrowing of a number of tokens worth 90% of the collateral.
Account health	A fraction of the sum of the USD value of a user's supplied tokens divided by the sum of the USD value of all the tokens they borrowed. It is used to determine whether a user is eligible for liquidation or not. When nominator is greater than the denominator in that fraction, we say that the user's account healthy, otherwise it is unhealthy.
Liquidation	The process in which some user's debt is liquidated by another user, by paying a portion of the owed tokens in exchange for the borrower's vTokens at a better exchange rate than the market.
Liquidation Multiplier	A value superior to 1. It is the amount by which the amount of vTokens that the liquidator should get by market price is multiplied to increase it.
Index	Refers to the interest rate index, which is a value that captures the history of interest rates of a market. It is updated after each transaction to compound the interest since the previous index.

2.3 Mathematical notations

The following mathematical notations are used in the document to simplify the equations. They are to be considered whitin the context in which they are used.

m_T:	Market for tokens of type T
vT:	the vToken type of token
$ER(T_1, T_2)$:	Exchange rate from the token T_1 to the token T_2, aka the x in: $1\ T_1 = (x \times 1)\ T_2$
$Ba(m)$:	Borrowed amount of tokens in a market m
$C(m)$:	Amount of cash in a market m
UR_m:	Utilization ratio of the market m
UM_m:	Utilization multiplier of the market m
UbR_m:	Utilization base rate of the market m
BIR_m:	Borrowing interest rate of the market m
RF_m:	Reserve factor of the market m
$rTB(m)$:	Real token balance of the market m
vTB_m:	vToken balance of the market m
Res_m:	Amount of tokens stored in the reserves of the market m
$CF(T)$:	Collateral factor for a token T
AH_u:	User's account health
BC_u:	User's borrowing capacity
$Ind_{m,n}$:	nth interest rate index of the market m
$Indl_m$:	Latest interest rate index of the market m
AI_m:	accumulated interest of the market m

2.4 System functioning

the Everscalend system uses markets, which are pools of tokens with algorithmically calculated interest rates, based on the supply and demand for the tokens they hold. Each market is unique to a cryptocurrency, and contains a transparent and publicly-inspectable ledger, with a record of all transactions and historical interest rates.

The lenders and the borrowers of tokens interact directly with the system, earning and paying a variable interest rate, without having to negotiate terms such as the interest rate or the value of collateral with a peer or counterparty.

In this section we describe the functioning of the system.

2.4.1 Kinds of users

There are two kinds of clearly identifiable users:

- The admin who is the deployer of (at least one of) the main contracts of the system like `MarketAggregator`, `WalletController` or `Oracle`.
- The "normal" users who interact with the system.

And there are four roles that can be taken by the users and which determine their permissions when it comes to interacting with the system:

- Owner: this is the role of the admin only. It gives them the right to modify all the parameters of the system.
- Upgrader: this role can be given to a user by the owner and it gives them the right to upgrades some contracts' codes.
- Parameter changer: this role can be given to a user by the owner and it gives them the to modify some of the parameters of the system.
- Other: this is the role of all the users who don't have a privileged role and can only interact with the system to realise the market opeartions (supply, borrow ...) without being able to manually change the parameters of the system.

2.4.2 User capabilities

In this section we describe what (non-privileged) users can do by using the system.

2.4.2.1 Supply

A user that wishes to make a certain amount of their tokens available for borrowing has to supply them to the system. The supplied tokens are then aggregated to the tokens of the same kind that were supplied by other users.

The supplier receives vTokens for their supply that they can use as collateral to borrow other tokens. vTokens are a currency used to represent how many of the real tokens in the market a user can withdraw, they also determine how many he can borrow of other tokens.

2.4.2.2 Withdraw

A user who owns vTokens can pay with them to withdraw the tokens he supplied. The user can do it at any time provided that their account is healthy, aka $AH_u > 1$ with AH_u being the user u account health

2.4.2.3 Borrow

To borrow a certain amount of some token:

- There have to be enough tokens in the market for borrowing.
- The borrowed amount has to be within the user's borrowing capacity, noted BC_u with u being the user.

The user's BC_u represents the amount of vTokens that the user can still use as collateral to borrow. Once that amount is reached, the user can no longer perform any action that costs them vTokens like borrowing or withdrawing.

2.4.2.4 Repay

A borrower can repay the amount they borrowed by returning it to the market. By doing that the collateral they set for the borrowing is freed and can be used to borrow other tokens.

2.4.2.5 Liquidate

When a user's account is unhealthy, aka ($AH_u < 1$), the liquidation of his debt becomes possible. The liquidation process consists of selling the borrower's collateral vTokens at a discount in exchange for the repayment of the borrower's debt or a portion of it.

The purpose of this mecanism is to financially incentivize the liquidators to add liquidity to the system and pay other users' debts.

2.4.3 Key system algorithms

2.4.3.1 Interest acquisition

The values of interest rates increase when the demand is high and decrease when it is low. The calculation of the real interest requires some intermediary variables:

- The utilisation ratio UR_m for a market m:

  $$UR_m = \frac{Ba(m)}{rTB(m) + Ba(m)}$$

  The utilisation ratio unifies supply and borrowing demand into a single variable and shows which markets are demand.

- The borrowing interest rate:

  $$BIR_m = UR_m * UM_m + UbR_m$$

The accumulated interest can then be calculated as follows:

$$AI_m = Ba(m) \times BIR_m \times \Delta t$$

with Δt being the time difference in seconds, between the current time and the last moment at which the interest rate was calculated.

2.4.3.2 Interest rate indexes

Each time the interest rate is calculated in a certain market, the interest rate index associated to that market is updated. New indexes are calculated as follows:

$$Ind_{m,n} = (BIR_m \times \Delta t + 1) \times Ind_{m,n-1}$$

With $Ind_{m,0} = 1$.

2.4.3.3 Reserves

When interest is accumulated, a portion of it, the size of which is determined by the reserve factor, is put in a reserve, the rest of it is stored as cash. After accumulating the interest, the reserve is updated by adding $(AI_m \times RF_m)$ to it.

The purpose of the reserves is to protect a portion of the lenders interest so that they don't lose everything in case of liquidation, only their cash can be liquidated.

2.4.3.4 vToken exchange rate calculation

As stated previously the exchange rates from vTokens to real tokens depend on the supply and borrowing demand of those tokens. It is calculated as follows:

$$ER(vT, T) = \frac{rTB(m_T) + Ba(m) - Res_{m_T}}{vTB_{m_T}}$$

2.4.3.5 Account health and borrow capacity calculation

A user u's account health determines whether or not it is liquidable, if $AH_u < 1$ then it is otherwise it isn't. If the user's account is liquidable then the user can't withdraw their tokens or borrow more tokens before some other users liquidiates them or they supply more tokens to the market to improve their account's health. The account's health is calculated whenever the user tries to perform any market operation or gets one of his loans liquidated.
Given:

$$sval(T, vTsa) = \frac{vTsa \times ER(vT, T)}{ER(T, USD)} \times CF(T)$$

and

$$bval(m_T, ba, ind) = \begin{cases} 0 & ba = 0 \\ \dfrac{(ba \times ind)/Indl_{m_T}}{ER(T, USD)} & ba \neq 0 \land Indl_{m_T} \neq ind \\ \dfrac{ba}{ER(T, USD)} & ba \neq 0 \land Indl_{m_T} = ind \end{cases}$$

It is calculated as follows:

$$AH_u = \frac{\sum_{(T_i, vTsa) \in SI_u} sval(T_i, vTsa)}{\sum_{(m_{T_j}, ba, ind) \in BI_u} bval(m_{T_j}, ba, ind)}$$

Where:

- SI_u: The supply information of the user u, which consists of a collection of couples $(m_{T_i}, vTsa_i)$ with:
  - T_i: the kind of the supplied token;
  - $vTsa_i$: the amount of vTokens that the user got from the supply.
- BI_u: The borrow information of the user u, which consists of a collection of triplets (m_{T_j}, ba, ind) with:
  - T_j: the kind of the borrowed token;
  - ba: the amount that was borrowed;
  - ind: the value of the interest rate index when the borrowing occured.

A user's borrowing capacity is a value in USD that determines how many vTokens a user can use as collateral or redeem, the limit being that the vToken's worth has to be less that the user's borrowing capacity. The borrowing capacity is calculated as follows:

$$BC = (\sum_{(m_{T_i}, vTsa) \in SI_u} sval(T_i, vTsa)) - (\sum_{(m_{T_j}, ba, ind) \in BI_u} bval(m_{T_j}, ba, ind))$$

2.4.3.6 Liquidation

The amount of vTokens the liquidator gets is calculated thanks to the liquidation multiplier LU which is the value by witch the amount of tokens they would get at market price is multiplied to increase it. If a liquidator wants to repay a portion R_T of a borrower's debt in tokens of type T, the amount of the borrower's vTokens that the liquidator will get vTA is calculated as follows:

$$vTA = R_T \times ER(T, vT) \times LU$$

The value of the liqudation multiplier is set manually by the admin.

2.4.3.7 Module locking mecanism (TODO)

To avoid messing up with the pools of tokens and market parameters while a user is performing an operation, a locking mecanism is used, that prevents the modification of the data that is used during the user's operation. The locks are especially necessary when try to extract tokens from the markets by borrowing or withdarwing, or from other users through liquidation

Such locks are used in:

- `BorrowModule`: The lock prevents other users from borrowing at the same time.
- `LiquidationModule`: In this case the lock is on the liquidated user, the purpose is to prevent more than one user liquidating the same user's debt.
- `WithdrawModule`: To prevent other users from withdrawing while one of them has started that process.
- `UserAccount`: This conract uses two locks called `borrowLock` and `liquidationLock`. As their names suggest, `borrowLock` locks the user's account during the processing of a borrow request, stopping the user from doing another one before the current one is finished. The `liquidationLock` stops the user from withdarwing or borrowing while he is being liquidated.

2.5 Architecture of the System

Figure 2.1 shows a simplistic representation of the architecture of the system. The main smart contracts are shown with their names only while the interfaces, libraries and the smart contracts that are not necessary for comprehension were omitted. The smart contracts are connected with arrows which are meant to show interactions between them, these interactions will be described below.

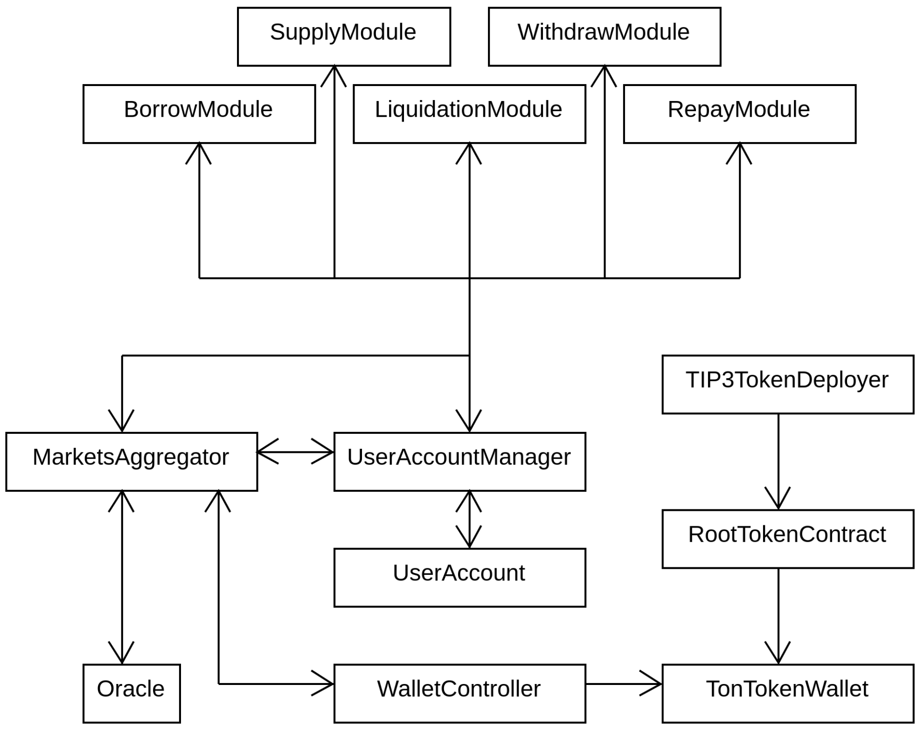

Figure 2.1: System architecture with smart contract interactions

2.5.1 Main smart contracts

2.5.1.1 MarketsAggregator

It mainly serves as a container for all the necessary information surrounding the markets, like the interest rates, the exchange rates and the borrow and supply amounts of each market.

Functionalities:

- Adding and removing markets.
- Updating the information on the markets. Either following market operations from users (like supplying, borrowing ... etc) or changes in the prices of the supported tokens.

Interactions:

- Oracle: to get token price updates.
- Operations modules: to perform market operations.

- UserAccountManager: to update account health after performing operations.
- WalletController: to transfer tokens to(from?) when necessary.

2.5.1.2 Operations modules

These smart contracts are used to help perform the market operations by doing the necessary math for these operations.

The smart contracts:

- SupplyModule
- WithdrawModule
- BorrowModule
- RepayModule
- LiquidationModule

2.5.1.3 Oracle

It is used by the MarketsAggregator smart contract to get price updates on the supported tokens in the markets.

2.5.1.4 TonTokenWallet

It serves as a TIP-3 token wallet and it handles token transfers to and from the wallet. Everscalend follows the TIP-3 token standard and these wallets are necessary to interact with the system. They can be delpoyed either by RootTokenContract or the a user who has the address of the RootTokenContract associated to the wallet's token.

Functionalities:

- Tranfer tokens to and from the wallet.
- Deploy a receiver's TonTokenWallet and transfer tokens to it.
- Burn tokens.

2.5.1.5 RootTokenContract

A contract that is deployed by a token owner and stores information about the root token (like its name, symbol and total supply).

Functionalities:

- Minting tokens.
- Deploying instances of TonTokenWallet.
- Updating the information on the root token.
- Change root token owner.

2.5.1.6 TIP3TokenDeployer

it deploys RootTokenContract contracts.

2.5.1.7 WalletController

It's the smart contract that controls all the TIP-3 wallets of the markets and managers the operations that require token transfers to and from those wallets.

Functionalities:

- Adding and removing market wallets.
- Getting information about market wallets.
- Decodes the payloads of messages that require TIP-3 token transfers and adds necessary information like message origin and token amount before passing to MarketsAggregator.
- Allows user to perform the supply, repay and liquidate operations.

Interactions:

- MarketsAggregator: to communicate to it the details of the market operations that require TIP-3 token transfer into the wallets of the markets.
- TonTokenWallet: to transfer tokens a user's wallet.

2.5.1.8 UserAccount

It's used to store information about how a user interacts with the markets, like how much they borrowed from which market and how much they supplied to which market. It also allows the user to perform the borrow and withdraw operations.

It interacts only with UserAccountManager, which can request the information and to which it returns it.

2.5.1.9 UserAccountManager

UserAccountManager serves as an intermediary between the user, the User-Account contract, the MarketsAggregator contract and the market operations modules.

Functionalities:

- Deploys UserAccount contract.
- Handles requests and data transfers from MarketsAggregator and market modules to the UserAccount contract.

Interactions:

- MarketsAggregator: to calculate user account health and perform market operations that were requested by the user.
- market modules: to perform matker operations.
- UserAccount: to get user information and to update it.

2.5.2 User interactions

The users mainly interact with TonTokenWallet and UserAccount to perform the market operations mentioned previously. The operations in which the users have to request tokens are done through the UserAccount (withdraw, borrow) smart contract, while the operations in which the users have to send tokens are done through TonTokenWallet (supply, repay, liquidate).

2.6 Usage scenarios

This section contains usage low level descriptions of usage scenarios which describe how the users interatct with the smart contracts and how the smart contracts interact between them.

Used notation:

UA:	UserAccount	UAM:	UserAccountManager
MA:	MarketsAggregator	WC:	WalletController
TTK:	TonTokenWallet	SM:	SupplyModule
WM:	WithdrawModule	BM:	BorrowModule
RM:	RepayModule	LM:	LiquidationModule

In the source code a lot of the interactions between the smart contracts are done with the intermediary of interfaces. In these usage scenarios we ignore the interfaces and present only the interactions between the contracts which the interfaces refer to.

2.6.1 Updating a user's account health

1. `UAM.calculateUserAccountHealth` is called from |UA| with a payload.
2. `UAM.calculateUserAccountHealth` calls `MA.calculateUserAccountHealth` which:
   - 2.1. Updates the markets' information and calculates the user's account health.
   - 2.2. If the user's account is unhealthy a `LiquidationPossible` event is emitted.
   - 2.3. A call is made to `UAM.updateUserAccountHealth` with the new account health.
3. `UAM.updateUserAccountHealth` calls `UA.updateUserAccountHealth` which calls, depending on the operation code provided in the payload, either `UAM.requestTokenPayout`, `UAM.returnAndSupply` or transfers the remaining gas to a povided address.

2.6.2 Supply

1. The user makes an internal transfer through `TTK.internalTransfer` with a payload containing the supply operation code.
2. The payload with information about the token wallet is passed to `WC.tokensReceivedCallback`.
3. The payload is decoded and after checking the operation code a call is made to `MA.performOperationWalletController` with the necessary information about the supply operation.
4. `MA.performOperationWalletController` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.

5. `MA.performOperation` calls `SM.performAction` which:
    - 5.1. Calculates the amount of vTokens to provide the user.
    - 5.2. Builds a data structure with the changes to the market (market delta) to which the supply was made.
6. The market delta is sent to `MA.receiveCacheDelta` which updates market informations and calls `SM.resumeOperation` with the new market information.
7. The information about the supply operation is then sent to `UAM.writeSupplyInfo` which transfers it to `UA.writeSupplyInfo`.
8. `UA.writeSupplyInfo` calls `UAM.calculateUserAccountHealth` with a payload having the `NO_OP` operation code.
9. After updating account health, the remaining gas is transferred back to the supplier.

2.6.3 Withdraw

1. The user calls `UA.withdraw` with the address of their TIP-3 wallet, the ID of the market to which their supplied their tokens and the amount their wish to withdraw.
2. `UA.withdraw` calls `UAM.requestWithdraw` with the withdrawal information.
3. `UAM.requestWithdraw` calls `MA.performOperationUserAccountManager` with the withdraw operation code.
4. `MA.performOperationUserAccountManager` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.
5. `MA.performOperation` calls `WM.performAction`.
6. `WM.performAction` locks the module `WM` and requests from `UAM` the user's borrow and supply information which is provided by `UA` and goes through `UAM.receiveWithdrawInfo` which sends it to `WM.withdrawTokensFromMarket`.
7. `WM.withdrawTokensFromMarket` checks that:
    - The user's account is healthy.
    - The user supplied at least as many tokens as the amount that they wish to withdraw.

- The worth in USD of the user's free collateral is equal or greater than the worth in USD of the amount of tokens they wish to withdraw.

Then calls `MA.receiveCacheDelta` with the withdrawal information.

8. `MA.receiveCacheDelta` updates market information and calls `WM.resumeOperation` with the new market information.
9. `WM.resumeOperation` unlocks the module `WM` and sends the withdrawing information to `UAM.writeWithdrawInfo`
   which calls `UA.writeWithdrawInfo` which:

   9.1. updates the user's supply information by decreasing it with the withdrawn amount.

   9.2. builds a payload with the amount of tokens that need to be sent, the user's TIP-3 wallet address and the opeartion code `REQUEST_TOKEN_PAYOUT`.
10. A call is made to `UAM.calculateUserAccountHealth` with the user's new supply and borrow information.
11. After the user's account health is updated, `UAM.returnAndSupply` is called.
12. `UAM.returnAndSupply` calls `MA.requestTokenPayout` and `WM.unlock` (which does nothing because the module was unlocked during the call to `WM.resumeOperation`).
13. `MA.requestTokenPayout` calls `WC.transferTokensToWallet` which calls `WC.transfer`.
14. `WC.transfer` calls `TTK.internalTransfer` which updates the user's balance with the amount of tokens that he requested to withdraw.

2.6.4 Borrow

1. The user calls `UA.borrow` with the address of their TIP-3 wallet, the ID of the market from which they wish to borrow and the amount of tokens they wish to borrow.
2. `UA.borrow` locks the user account and calls `UAM.requestIndexUpdate` with the borrowing information.
3. `UAM.requestIndexUpdate` calls `MA.performOperationUserAccountManager` with the `BORROW_TOKENS` operation code.
4. `MA.performOperationUserAccountManager` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.

5. `MA.performOperation` calls `BM.performAction`.
6. `BM.performAction` locks the module `BM` and requests from `UAM` to update the market's indexes which is done by calling `UA.borrowUpdateIndexes`.
7. `UA.borrowUpdateIndexes` gets from the markets the updates indexes and passes them along with the user's borrow and supply into to `UAM.passBorrowInformation` which calls `BM.borrowTokensFromMarket`.
8. `BM.borrowTokensFromMarket` checks that:
   - That there are enough tokens in the market for the borrowing.
   - That the user's account is healthy.
   - That the user has enough collateral to make the borrowing.
9. A `TokenBorrow` event is emitted with the borrowing information.
10. `BM.borrowTokensFromMarket` calls `MA.receiveCacheDelta` with the borrowing information and the information about the changes to the market after the borrowing.
11. `MA.receiveCacheDelta` updates the market information and calls `BM.resumeOperation` with the new market information.
12. `BM.resumeOperation` unlocks the module `BM` and sends the borrowing information to `UAM.writeBorrowInformation` which calls `UA.writeBorrowInformation` which:
    - 12.1. Updates market information and the user's borrowing information.
    - 12.2. Unlocks `UA`.
    - 12.3. Builds a payload with the `REQUEST_TOKEN_PAYOUT` operation code.
13. `UAM.calculateUserAccountHealth` is called with the payload as well as the user's supply and borrow information.
14. After the user's account health is updated, `MA.requestTokenPayout` is called.
15. `MA.requestTokenPayout` calls `WC.transferTokensToWallet` which calls `WC.transfer`.
16. `WC.transfer` calls `TTK.internalTransfer` which updates the user's balance with the amount of tokens that they borrowed.

2.6.5 Repay

1. The user makes an internal transfer through `TTK.internalTransfer` with a payload containing the repay operation code.
2. The payload with information about the token wallet is passed to `WC.tokensReceivedCallback`.
3. The payload is decoded and after checking the operation code a call is made to `MA.performOperationWalletController` with the necessary information about the supply operation.
4. `MA.performOperationWalletController` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.
5. `MA.performOperation` calls `RM.performAction` which calls `UAM.requestRepayInfo`.
6. `UAM.requestRepayInfo` calls `UA.sendRepayInfo` which updates market information and transfers it with information about the user's TIP-3 wallet to `UAM.receiveRepayInfo` which calls `RM.repayLoan`.
7. `RM.repayLoan` calculates how much of the loan will be repayed and the changes to the markets after the repayment. A `RepayBorrow` event is emitted with the repayment information. That information is then sent to `MA.receiveCacheDelta` which updates market informations and calls `RM.resumeOperation` with the new market information.
8. `RM.resumeOperation` calls `UAM.writeRepayInformation` which transfers it to `UA.writeRepayInformation`.
9. `UA.writeRepayInformation` calls `UAM.calculateUserAccountHealth` with one of the two operation codes:
   - `REQUEST_TOKEN_PAYOUT` if there are leftover tokens after the repayment.
   - `NO_OP` otherwise.
10. After updating account health, if there are leftover tokens after the repayment, they are transferred to the user's TIP-3 wallet.

2.6.6 Liquidate

1. The user makes an internal transfer through `TTK.internalTransfer` with a payload containing the liquidation operation code.
2. The payload with information about the token wallet is passed to `WC.tokensReceivedCallback`.

3. The payload is decoded and after checking the operation code a call is made to `MA.performOperationWalletController` with the necessary information about the liquidation.

4. `MA.performOperationWalletController` calls the Oracle to request the newest token prices, the response is receive with `MA.receiveAllUpdatedPrices` which updates all the prices in the markets and calls `MA.performOperation`.

5. `MA.performOperation` calls `LM.performAction` which calls `UAM.requestLiquidationInformation`.

6. `UAM.requestLiquidationInformation` calls `UA.requestLiquidationInformation` which updates market's indexes and calls `UAM.receiveLiquidationInformation` with the new indexes and the user's supply and borrow information.

7. `UAM.receiveLiquidationInformation` calls `LM.liquidate` which:

   (a) Checks the account health of the user that is targeted for liquidation to check that it is still required.

   (b) Selects the minimum between the provided amount of tokens for the liquidation and the borrowed amount by the targeted user as the liquidation amount.

   (c) Calculates the USD value of the liquidation amount.

   (d) Calculates the how many of the targeted user's collateral to seize.

   (e) Emits a `TokensLiquidated` event with the liquidation information, update the markets and the liquidated user's borrow information.

   (f) Calls `MA.receiveCacheDelta` which updates market informations and calls `RM.resumeOperation` with the new market information.

8. `LM.resumeOperation` recovers the sent information and passes it to `UAM.seizeTokens` which calls `UA.liquidateVTokens`.

9. `UA.liquidateVTokens` updates the user's borrow and supply information and calls `UAM.grantVTokens`.

10. `UAM.grantVTokens` calls `UA.checkUserAccountHealth` on the target user's account and wallet addresses to check their account health. Also calls `UA.grantVTokens`.

11. `UA.grantVTokens` checks the liquidator's account health and builds a payload with the operation code `RETURN_AND_UNLOCK` to pass to the function that checks the user's account.

12. Once the new user's account health is recovered `UAM.returnAndSupply` is called.

- if there are leftover tokens after the liquidation then a call is made to `MA.requestTokenPayout` with the tokens to return and the liquidator's TIP-3 wallet.
- A call is made to `LM.unlock` to unlock it and return the remaining gas to the liquidator's wallet.

13. `MA.requestTokenPayout` calls `WC.transferTokensToWallet`.
14. A call is then made to `TTK.transfer` which calls `TTK.internalTransfer` to update the user's balance by increasing it with the returned amount of tokens.

Chapter 3

Risks

3.1 Insolvency risk

Insolvency is when a borrower's loan becomes worth more than their collateral. In this case neither they nor the liquidators are incentivized to repay the loan, which removes liquidity from the market since these users will hold on to loans that will not get repaid. Insolvency can happen if the underlying tokens of the collateral vTokens lose their value quickly or the borrowed tokens' price increases rapidly.

3.2 Illiquidity risk

Illiquidity is when there aren't enough tokens in the market for a supplier to do a withdrawal or for a borrower to borrow some amount. It is problematic because users are supposed to have control over their tokens and be able to withdraw them whenever they want to, given that their account's health allows it. Illiquidity can happen as well if the price of the borrowed token increases rapidly, that will disincentivise the borrowers and the liquidators from repaying the loan as they would rather keep holding their tokens or selling them. Same as the suppliers which will lead to a bank run (an event in which suppliers will try to withdraw their supplies as quickly as possible to avoid losing tokens) and that will lead to the slowest suppliers, especially if they want to borrow big amounts, to lose some or all of their tokens since the loans aren't getting repaid.

3.3 Unsound math risk

Math operations use approximations and rounding. It could lead in some particular cases to errors that could affect the functioning of the system or introduce vulnerabilities that can be taken advantage of.

3.4 Centralization risk

The risk that comes with having an administrator or super user role which gives the detainer of that role the capability to unilaterally and arbitrarily modify the functioning of the system is in itselft risky.

Focusing too much power on a single entry point means that losing access to it, or losing it's possession to a malicious user could lead to bad consequences. Especially since many values like the collateral factor and liquidation multiplier are editable with an admin role, the contract owner can also decide who can and who can't change market parameters. Therefore such infromation needs to be clarified an a system put in place making sure that major changes to the system can't be done individually and manually like that.

3.5 Locking risk

When a locking mecanism is used, there is a risk of malicious users taking advantage of to affect the capabilities of other users to realise some operations. There is also the risk of programming errors which might lead to a lock that does not get unlocked or that is unlocked at the wrong time and that can affect negatively the functioning of the sytem.

3.6 Double spending risk

Double spending is when a user tries to use the same amount of tokens to pay for two different things and profit off of that. It can happen for example if a supplier wishes to get twice the amount of vTokens with the same supply or when a liquidator tries to liquidate with the same amount twice.

3.7 Visibility risk

This risk involves all the risks of having functions in contracts that are accessible to users which are not intended to use them. Especially if they are non-pure functions which can write information on the system and significantly affect it's functioning.

Chapter 4

System Properties

4.1 System properties

This section lists the system properties that should always be verified by contract. The mathematical notations defined in 2.3 are used.

4.1.1 MarketAggregator

4.1.2 UserAccount

4.1.3 UserAccountManager

Chapter 5

Code audit