Protocol Whitepaper
V1.0

wow@aave.com

January 2020

Abstract

This document describes the definitions and theory behind the Aave Protocol explaining the different aspects of the implementation.

Contents

1 Introduction

The birth of the Aave Protocol marks Aave's shift from a decentralized P2P lending strategy (direct loan relationship between lenders and borrowers, like in ETHLend) to a pool-based strategy. Lenders provide liquidity by depositing cryptocurrencies in a pool contract. Simultaneously, in the same contract, the pooled funds can be borrowed by placing a collateral. Loans do not need to be individually matched, instead they rely on the pooled funds, as well as the amounts borrowed and their collateral. This enables instant loans with characteristics based on the state of the pool. A simplified scheme of the protocol is presented in figure 1 below.

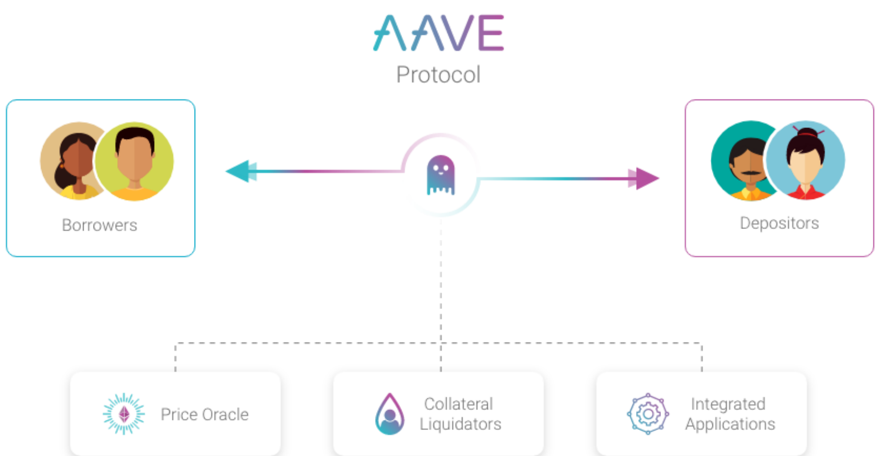

Figure 1: The Aave Protocol

The interest rate for both borrowers and lenders is decided algorithmically:

- For borrowers, it depends on the cost of money - the amount of funds available in the pool at a specific time. As funds are borrowed from the pool, the amount of funds available decreases which raises the interest rate.
- For lenders, this interest rate corresponds to the earn rate, with the algorithm safeguarding a liquidity reserve to guarantee withdrawals at any time.

1.1 Basic Concepts

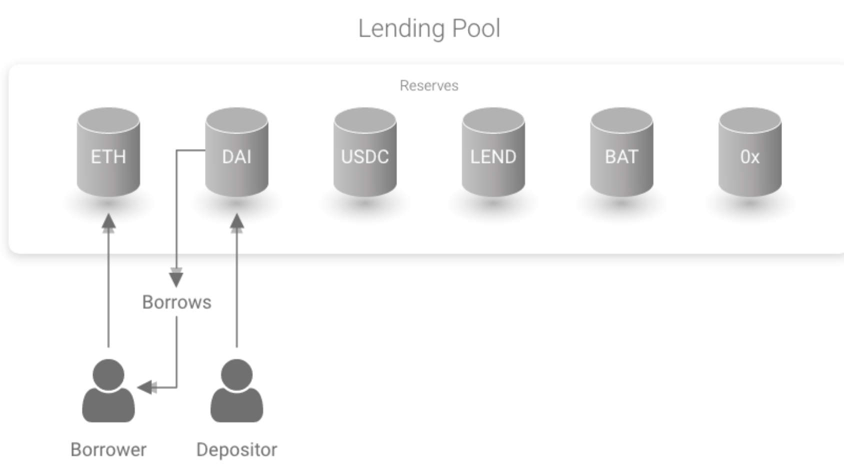

Figure 2: Lending Pool Basics

At the heart of a **lending pool** is the concept of **reserve**: every pool holds reserves in multiple currencies, with the total amount in Ethereum defined as **total liquidity**. A reserve accepts **deposits** from lenders. Users can

borrow these funds, granted that they lock a greater value as **collateral**, which backs the borrow position. Specific currencies in the pooled reserves can be configured as collateral or not for borrow positions, only low risk tokens should be considered. The amount one can borrow depends on the currencies deposited still available in the reserves. Every reserve has a specific **Loan-To-Value (LTV)**, calculated as the **weighted average** of the different LTVs of the currencies composing the collateral, where the weight for each LTV is the **equivalent amount of the collateral in ETH**; figure 3 shows an example of parameters.

Every borrow position can be opened with a **stable or variable rate**. Borrows have infinite duration, and there is no repayment schedule: partial or full repayments can be made anytime.

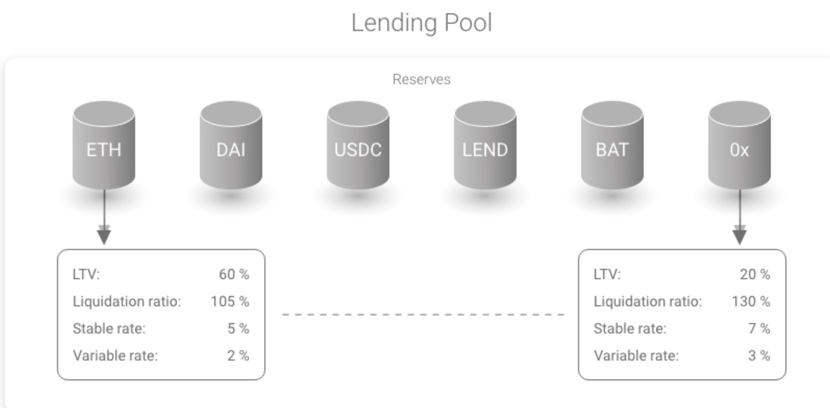

Figure 3: Lending Pool Parameters

In case of price fluctuations, a borrow position might be **liquidated**. A liquidation event happens when the price of the collateral drops below the threshold, L_Q, called **liquidation threshold**. Reaching this ratio channels a **liquidation bonus**, which incentivizes liquidators to buy the collateral at a discounted price. Every reserve has a specific liquidation threshold, following the same approach as for the LTV. Calculation of the average liquidation threshold L_Q^a is performed dynamically, using the weighted average of the liquidation thresholds of the collateral's underlying assets.

At any point in time, a borrow position is characterized by its health factor H_f, a function for the total collateral and the total borrows which determines if a loan is **undercollateralized**:

$$H_f = \frac{TotalCollateralETH * L_Q^a}{TotalBorrowsETH + TotalFeesETH} \ \textit{when } H_f < 1, \textit{ a loan is considered undercollateralized and can be liquidated}$$

Further details on liquidation can be found in section 3.6.

1.2 Formal Definitions

Variable	Description	
T, **current timestamp**	Current number of seconds defined by `block.timestamp`.	
T_l, **last updated timestamp**	Timestamp of the last update of the reserve data. T_l is updated every time a borrow, deposit, redeem, repay, swap or liquidation event occurs.	
ΔT, **delta time**		$\Delta T = T - T_l$
T_{year}, **seconds**	Number of seconds in a year.	$T_{year} = 31536000$
ΔT_{year}, **yearly period**		$\Delta T_{year} = \frac{\Delta T}{T_{year}}$
L_t, **total liquidity**	Total amount of liquidity available in the reserve. The decimals of this value depend on the decimals of the currency.	
B_s, **total stable borrows**	Total amount of liquidity borrowed at a stable rate. The decimals of this value depend on the decimals of the currency.	
B_v, **total variable borrows**	Total amount of liquidity borrowed at a variable rate. The decimals of this value depend on the decimals of the currency.	
B_t, **total borrows**	Total amount of liquidity borrowed. The decimals of this value depend on the decimals of the currency.	$B_t = B_s + B_v$
U, **utilization rate**	Representing the utilization of the deposited funds.	$U = \begin{cases} 0, \text{ if } L_t = 0 \\ \frac{B_t}{L_t}, \text{ if } L_t > 0 \end{cases}$
$U_{optimal}$, **target utilization rate**	The utilization rate targeted by the model, beyond the variable interest rate rises sharply.	
R_{v_0}, **base variable borrow rate**	Constant for $B_t = 0$. Expressed in ray.	
R_{slope1}, **interest rate slope below** $U_{optimal}$	Constant representing the scaling of the interest rate versus the utilization, when $U < U_{optimal}$. Expressed in ray.	
R_{slope2}, **interest rate slope above** $U_{optimal}$	Constant representing the scaling of the interest rate versus the utilization, when $U \geq U_{optimal}$. Expressed in ray.	
R_v, **variable borrow rate**	$R_v = \begin{cases} R_{v_0} + \frac{U}{U_{optimal}} R_{slope1}, \text{ if } U \leq U_{optimal} \\ R_{v_0} + R_{slope1} + \frac{U - U_{optimal}}{1 - U_{optimal}} R_{slope2}, \text{ if } U > U_{optimal} \end{cases}$	
R_s, **stable rate**	Implemented in section 4.2. Expressed in ray.	
M_r, **average market lending rate**	Base stable borrow rate, defined for i platforms with P_i^r the lending rate and P_i^v the borrowing volume. Expressed in ray.	$M_r = \frac{\sum_{i=1}^{n} P_r^i P_v^i}{\sum_{i=1}^{n} P_v^i}$

Variable	Description	
R^t_{sa}, **average stable rate borrow rate**	When a stable borrow of amount B_{new} is issued at rate R_s:	$R^t_{sa} = \frac{B_s R^{t-1}_{sa} + B_{new} R_s}{B_s + B_{new}}$
	When a user repays an amount B_x at stable rate R_{sx}:	$R^t_{sa} = \begin{cases} 0, \text{ if } B_s - B_x = 0 \\ \frac{B_s R^{t-1}_{sa} - B_x R_{sx}}{B_s - B_x}, \text{ if } B_s - B_x > 0 \end{cases}$
	Check the methods `decreaseTotalBorrowsStableAndUpdateAverageRate()` and `increaseTotalBorrowsStableAndUpdateAverageRate()`. Expressed in ray.	
R_O, **overall borrow rate**	Overall borrow rate of the reserve, calculated as the weighted average between the total variable borrows B_v and the total stable borrows B_s.	$R_O = \begin{cases} 0, \text{ if } B_t = 0 \\ \frac{B_v R_v + B_s R_{sa}}{B_t}, \text{ if } B_t > 0 \end{cases}$
R_l, **current liquidity rate**	Function of the overall borrow rate R_O and the utilization rate U.	$R_l = R_O U$
C^t_i, **cumulated liquidity index**	Interest cumulated by the reserve during the time interval Δ_T, updated whenever a borrow, deposit, repay, redeem, swap, liquidation event occurs.	$C^t_i = (R_l \Delta T_{year} + 1) C^{t-1}_i$ $C^0_i = 1 \times 10^{27} = 1\ ray$
I^t_n, **reserve normalized income**	Ongoing interest cumulated by the reserve.	$I^t_n = (R_l \Delta T_{year} + 1) C^{t-1}_i$
B^t_{vc}, **cumulated variable borrow index**	Interest cumulated by the variable borrows B_v, at rate R_v, updated whenever a borrow, deposit, repay, redeem, swap, liquidation event occurs.	$B^t_{vc} = (1 + \frac{R_v}{T_{year}})^{\Delta T_x} B^{t-1}_{vc}$ $B^0_{vc} = 1 \times 10^{27} = 1\ ray$
B^t_{vcx}, **user cumulated variable borrow index**	Variable borrow index of the specific user, stored when a user opens a variable borrow position.	$B^t_{vcx} = B^t_{vc}$
B_x, **user principal borrow balance**	Balance stored when a user opens a borrow position. In case of multiple borrows, the compounded interest is cumulated each time and it becomes the new principal borrow balance.	
B_{xc}, **user compounded borrow balance**	Principal B_x plus the cumulated interests.	
	For a variable position:	$B_{xc} = \frac{B_{vc}}{B_{vcx}} (1 + \frac{R_v}{T_{year}})^{\Delta T_x} B_x$
	For a stable position:	$B_{xc} = (1 + \frac{R_s}{T_{year}})^{\Delta T_x} B_x$
H_f, **health factor**	when $H_f < 1$, a loan is considered undercollateralized and can be liquidated	$H_f = \frac{TotalCollateralETH * L^a_Q}{B_t + TotalFeesETH}$

2 Protocol Architecture

The current implementation of the protocol is as follows:

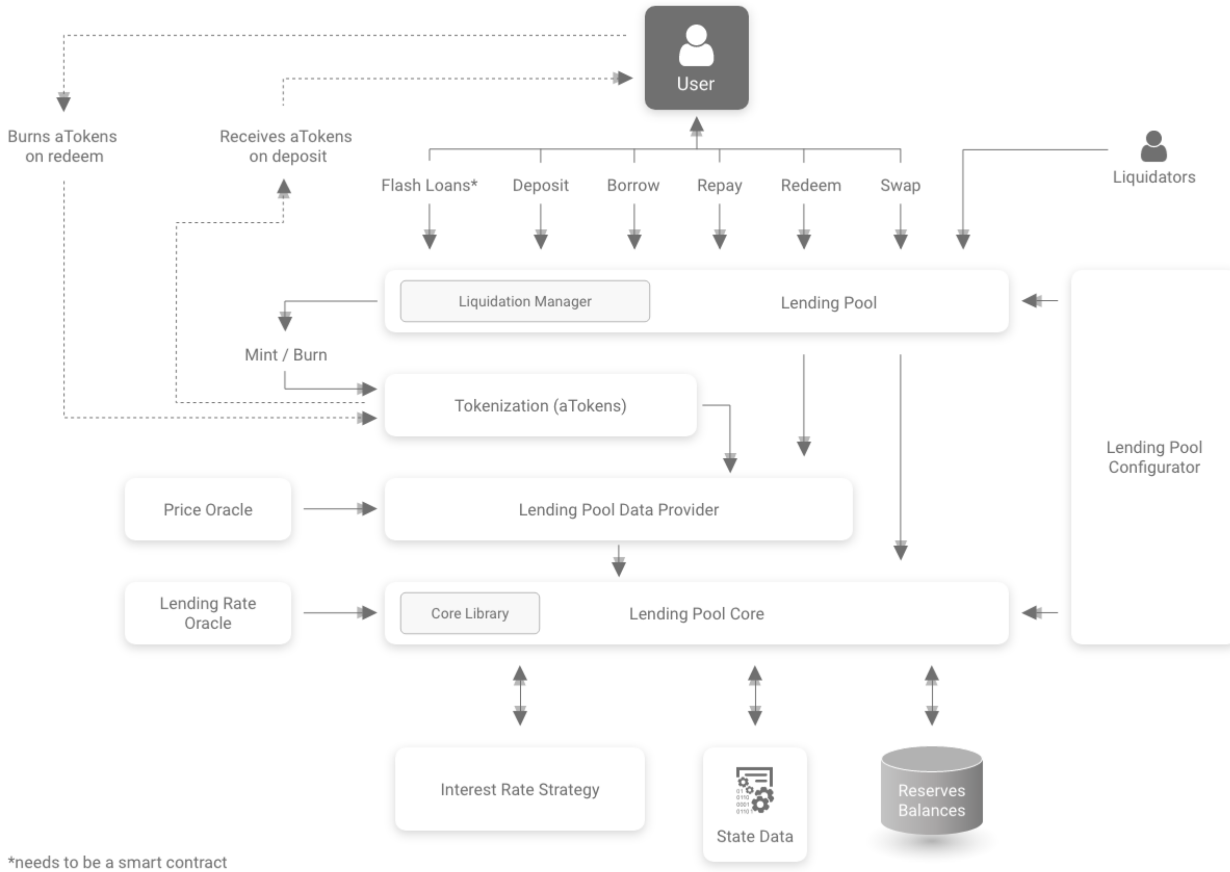

Figure 4: Protocol Architecture

2.1 Lending Pool Core

The `LendingPoolCore` contract is the center of the protocol, it:

- holds the state of every reserve and all the assets deposited,
- handles the basic logic (cumulation of the indexes, calculation of the interest rates...).

2.2 Lending Pool Data Provider

The `LendingPoolDataProvider` contract performs calculations on a higher layer of abstraction than the `LendingPoolCore` and provides data for the `LendingPool`; specifically:

- Calculates the ETH equivalent a user's balances (Borrow Balance, Collateral Balance, Liquidity Balance) to assess how much a user is allowed to borrow and the health factor.
- Aggregates data from the `LendingPoolCore` to provide high level information to the `LendingPool`.
- Calculate of the Average Loan to Value and Average Liquidation Ratio.

2.3 Lending Pool

The `LendingPool` contract uses the `LendingPoolCore` and `LendingPoolDataProvider` to interact with the reserves through the actions:

- Deposit
- Redeem
- Borrow
- Repay
- Rate swap
- Liquidation
- Flash loan

One of the advanced features implemented in the `LendingPool` contract is the **tokenization** of the lending position. When a user deposits in a specific reserve, he receives a corresponding amount of **aTokens**, tokens that map the liquidity deposited and accrue the interests of the deposited underlying assets. Atokens are minted upon deposit, their value increases until they are burned on redeem or liquidated. Whenever a user opens a borrow position, the tokens used as collateral are locked and cannot be transferred. Further details on the tokenization are in section 3.8.

2.4 Lending Pool Configurator

The `LendingPoolConfigurator` provides main configuration functions for `LendingPool` and `LendingPoolCore`:

- Reserve initialization
- Reserve configuration
- Enable/disable borrowing on a reserve
- Enable/disable the usage of a specific reserve as collateral.

The `LendingPoolConfigurator` contract will be integrated in Aave Protocol governance.

2.5 Interest Rate Strategy

The `InterestRateStrategy` contract holds the information needed to update the interest rates of a specific reserve and implements the update of the interest rates. Every reserve has a specific `InterestRateStrategy` contract. Specifically, within the base strategy contract `DefaultReserveInterestRateStrategy` the following are defined:

- Base variable borrow rate R_{v_0}
- Interest rate slope below optimal utilisation R_{slope1}
- Interest rate slope beyond optimal utilisation R_{slope2}

The current variable borrow rate is:

$$R_v = \begin{cases} R_{v_0} + \frac{U}{U_{optimal}} R_{slope1}, \text{ if } U \leq U_{optimal} \\ R_{v_0} + R_{slope1} + \frac{U - U_{optimal}}{1 - U_{optimal}} R_{slope2}, \text{ if } U > U_{optimal} \end{cases}$$

This interest rate model allows for calibration of key interest rates:

- At $U = 0$, $R_v = R_{v_0}$
- At $U = U_{optimal}$, $R_v = R_{v_0} + R_{slope1}$
- Above $U_{optimal}$, the interest rate rises sharply to take into account the cost of capital.

The stable borrow rate follows the same model described in section 4.2.

2.6 Governance

The rights of the protocol are controlled by the LEND token. Initially, the Aave Protocol will be launched with a decentralized on-chain governance based on the DAOStack framework which will evolve to a fully autonomous protocol. On-chain implies all votes are binding: actions that follow a vote are hard-coded and must be executed.

To understand the scope of the governance it's important to make the distinction:

- The **Aave Protocol** is bound to evolve and will allow the creation of multiple lending pools with segregated liquidity, parameters, permissions, and type of assets.
- The **Aave Lending Pool** is the first pool of the Aave protocol until the Pool Factory Update is released and anyone can create their own pool.

Within the Aave Protocol, the governance will take place at two level :

1. The **Protocol's Governance** voting is weighted by LEND for decisions related to protocol parameters and upgrades of the smart contract. It can be compared to MakerDAO's governance where stakeholders vote on current and future parameters of the protocol.
2. The **Pool's Governance** where your vote is weighted based on your share of pool liquidity expressed in aTokens. The votes cover pool specific parameters such as assets used as collateral or to be borrowed.

   Each Pool will have its own governance, under the umbrella of the Protocol's Governance.

More details on the Governance will be published in a Governance Proposal to the community.

3 The `LendingPool` Contract

The actions implemented within `LendingPool` allow users to interact with the reserve. All the actions follow this specific sequence:

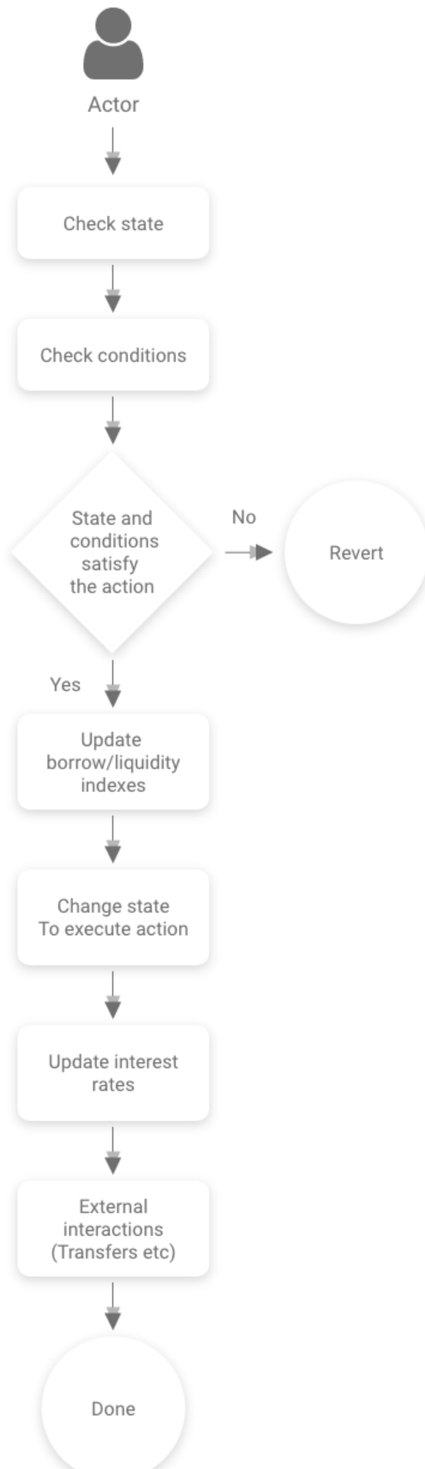

Figure 5: The `LendingPool` Contract

3.1 Deposit

The deposit action is the simplest one and does not have any particular state check. The sequence of action is:

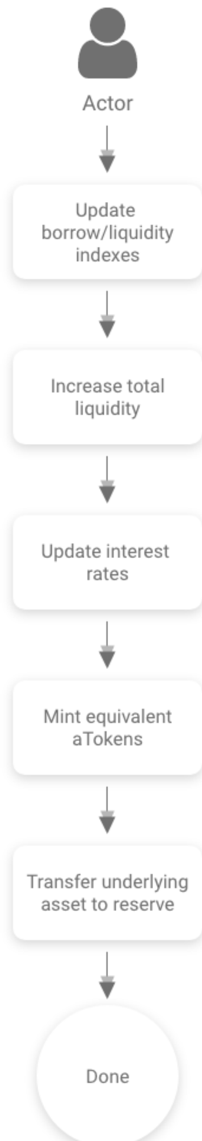

Figure 6: Deposit funds

3.2 Redeem

The redeem action allows users to exchange an amount of aTokens for the underlying asset. The actual amount to redeem is calculated using the aToken/underlying exchange rate E_i in section 3.8. The action is defined as follows:

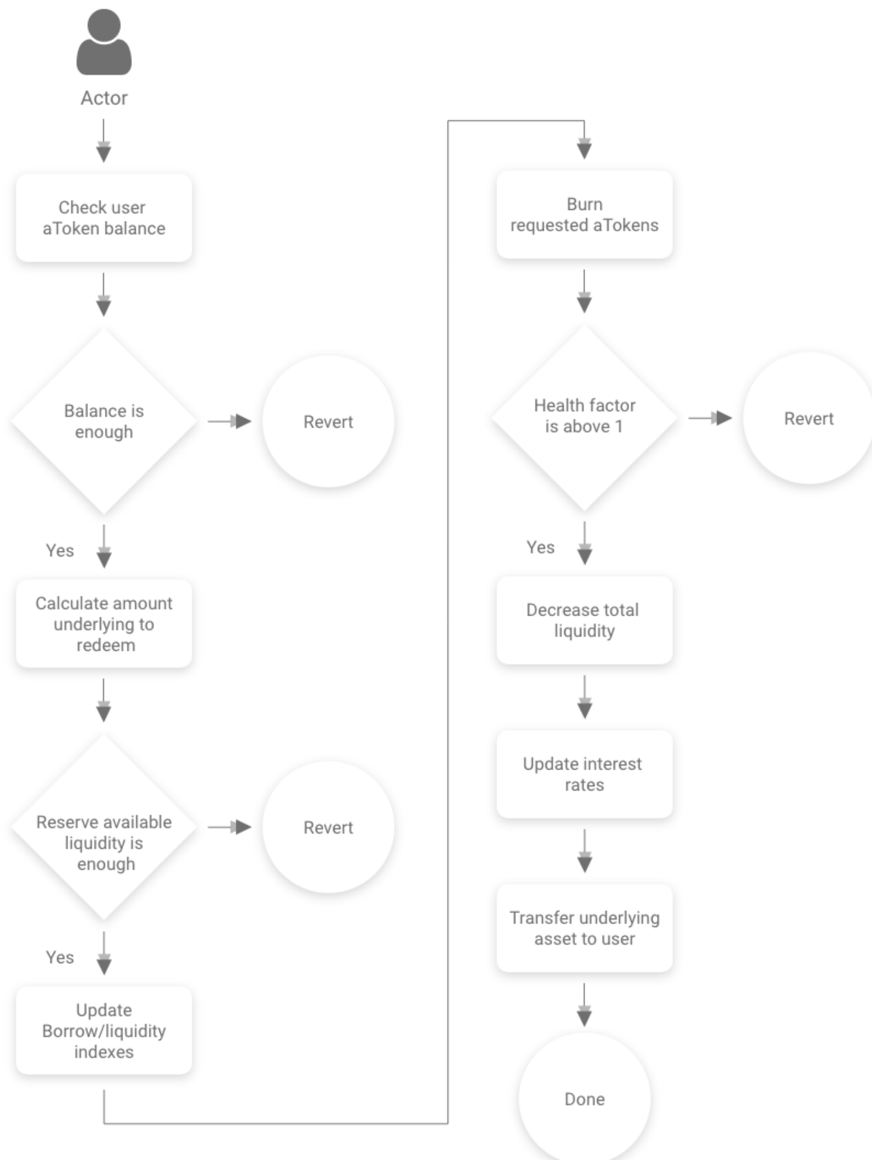

Figure 7: Redeem funds

3.3 Borrow

The borrow action transfers to the user a specific amount of underlying asset, in exchange of a collateral that remains locked. The flow of action can be described as follows:

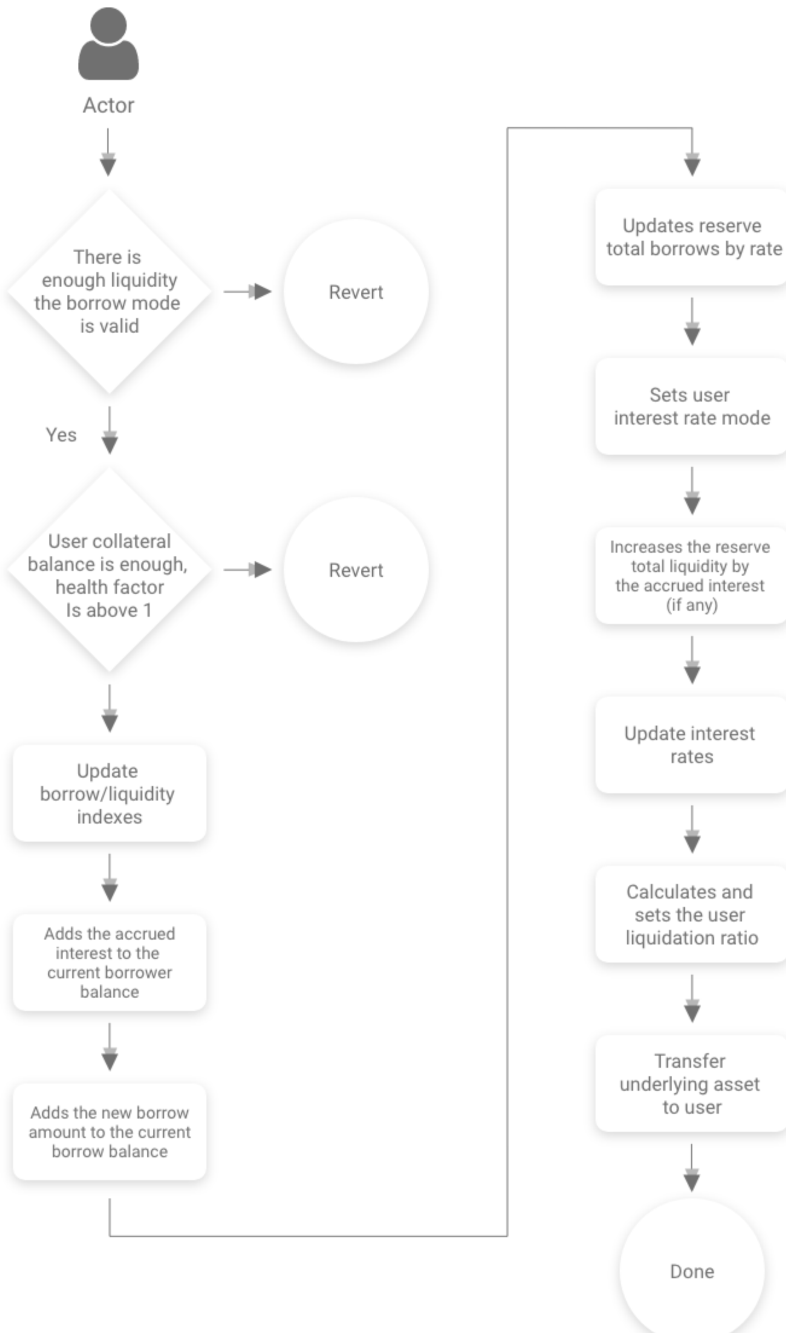

Figure 8: Borrow funds

3.4 Repay

The repay action allows the user to repay completely or partially the borrowed amount plus the origination fee and the accrued interest.

Figure 9: Repay a loan

3.5 Swap Rate

The swap rate action allows a user with a borrow in progress to swap between variable and stable borrow rate.

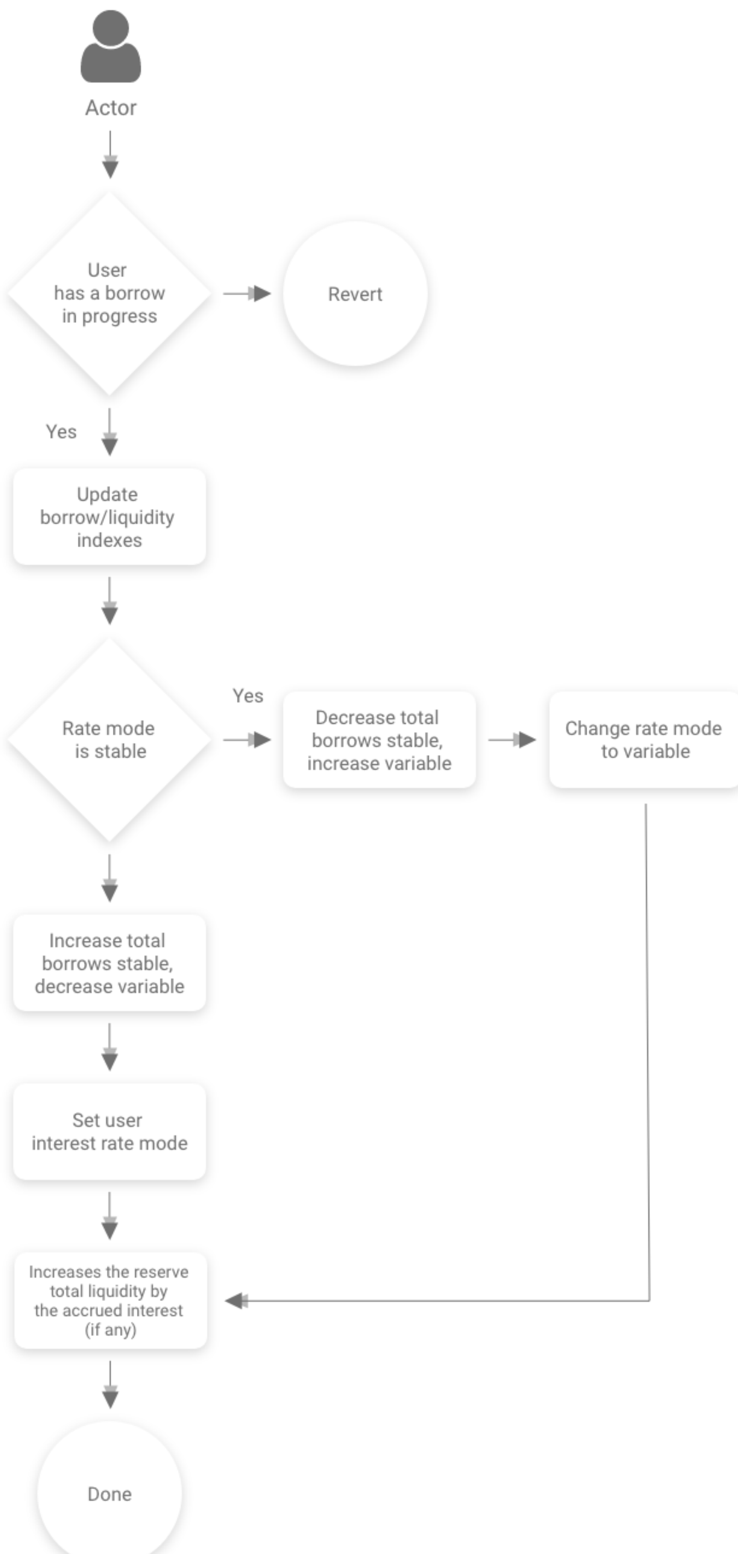

Figure 10: Swap Rate

3.6 Liquidation Call

The `liquidationcall` contract allows any external actor to purchase part of a collateral at a discounted price. In case of a liquidation event, a maximum of 50% of the loan can be liquidated, which will bring the health factor back above 1.

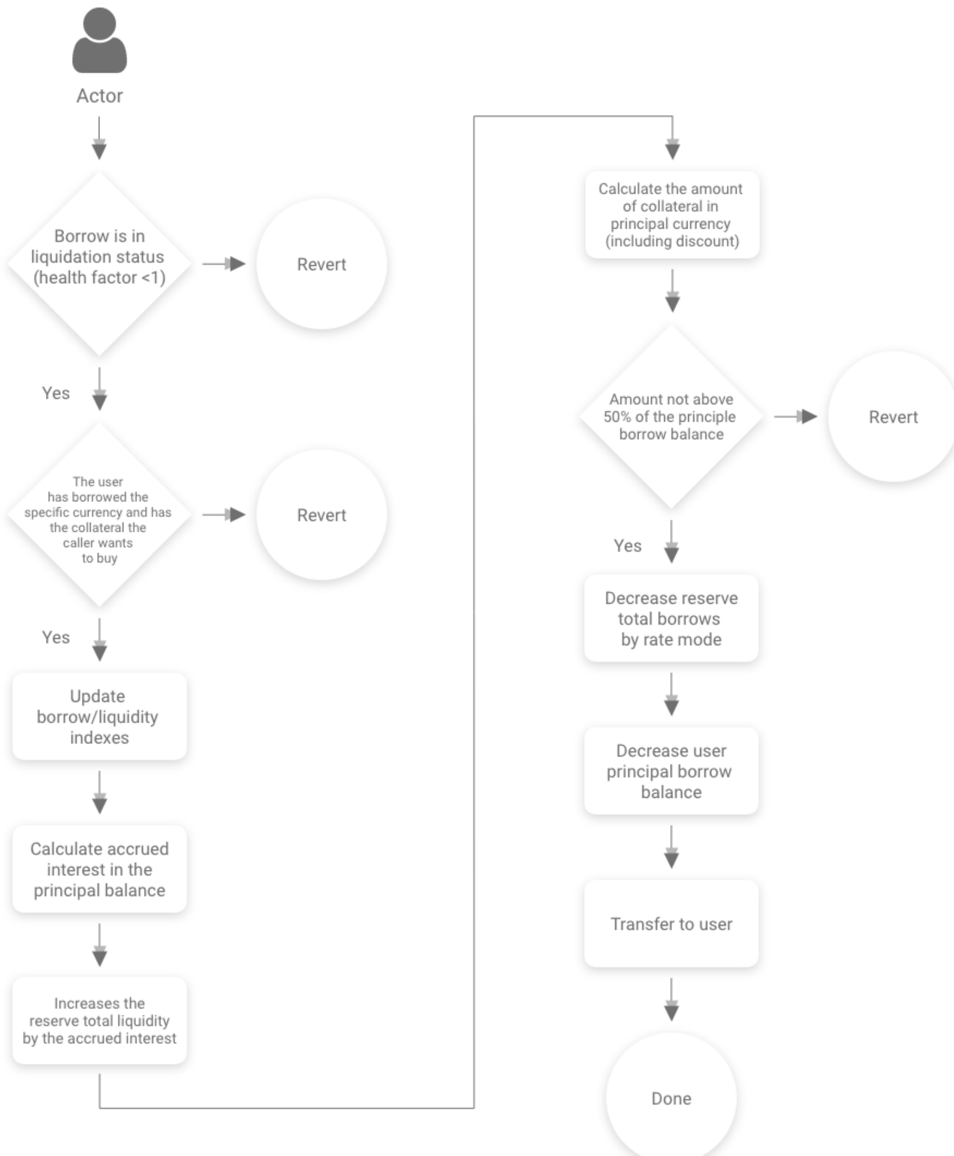

Figure 11: Liquidation

3.7 Flash Loans

The flash loan action will allow users to borrow from the reserves within a single transaction, as long as the user returns more liquidity that has been taken.

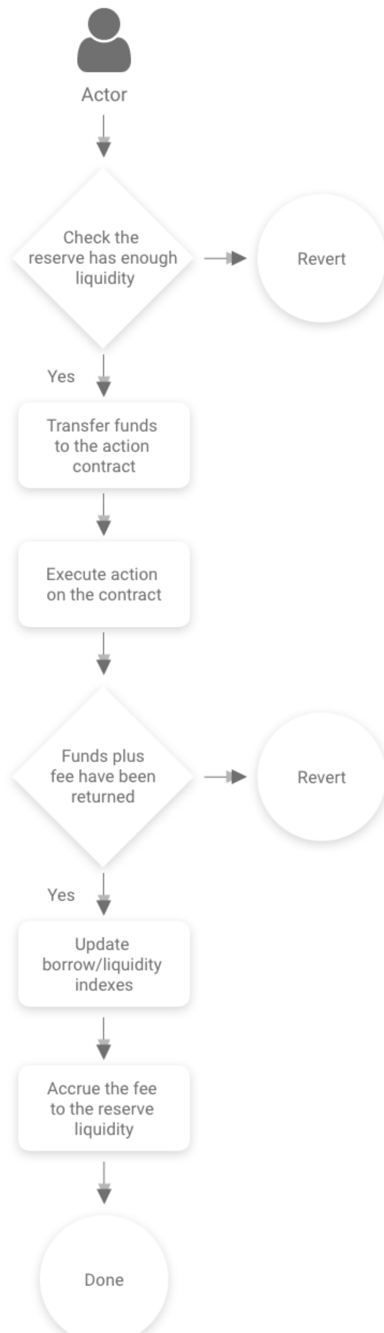

Figure 12: Flash Loan

Flash loans temporarily transfer the funds to a smart contract that respects the `IFlashLoanEnabledContract.sol` interface. The address of the contract is a parameter of the action. After the funds are transferred, the method `executeOperation()` is executed on the external contract. The contract can do whatever action is needed with the borrowed funds. After the method `executeOperation()` is completed, a check is performed to verify that the funds plus fee have been returned to the `LendingPool` contract. The fee is then accrued to the reserve, and the state of the reserve is updated. If less funds than what was borrowed have been returned to the reserve, the transaction is reverted.

3.8 Tokenization

The Aave protocol implements a tokenization strategy for liquidity providers. Upon deposit, the depositor receives a corresponding amount of derivative tokens, called Aave Tokens (aTokens for short) that map 1:1 the underlying assets. The balance of aTokens of every depositor grows over time, driven by the perpetual accrual of interest of deposits. aTokens are fully ERC20 compliant.

aTokens also natively implement the concept of interest rate redirection. Indeed, the value accrued over time by the borrowers' interest rate payments is distinct from the principal value. Once there is a balance of aTokens, the accrued value can be redirected to any address, effectively splitting the balance and the generated interest. We call the continuous flow of accumulated interest over time the **interest stream**.

To implement this tokenization strategy, Aave introduced the following concepts in the aToken contract:

1. **User x balance index I_x^t:** Is the value of the reserve normalized income I_x^t at the moment of execution of the last action by the user.
2. **Principal balance B_p:** Is the balance stored in the balances mapping of the ERC20 aToken contract. The principal balance gets updated on every action that the user executes on the aToken contract (deposit, redeem, transfer, liquidation, interest rate redirection)
3. **Redirection address A_r:** When a user decides to redirect his interest stream to another address, a new redirection address A_r is provided. If no redirection of the interest stream is performed, A_r is 0
4. **Redirected Balance B_r^x:** Whenever a user redirects his interest stream, the balance of user redirecting is added to the redirected balance B_r of the address specified by B_r. Defined as follows:

   $$B_r^x = \sum_X B_p$$

   Where X is the set of users redirecting the interest stream to the user x

   The redirected balance decreases whenever a user $x_0 \in X$ redeems or transfers his aTokens to another user that is not redirecting to x.
5. **Current balance** B_c: Is the balance returned by the `balanceOf()` function of the aToken contract. Defined as follows:

   $$\mathrm{B}_c^x = \begin{cases} 0, \text{ if } B_p^x = 0 \text{ and } B_r^x = 0 \\ B_p^x + B_r^x(\frac{I_n}{I_x} - 1), \text{ if } A_r <> 0 \\ B_p^x \frac{I_n}{I_x} + B_r^x(\frac{I_n}{I_x} - 1), \text{ if } A_r = 0 \end{cases}$$

3.8.1 Limitations of the tokenization model

The described tokenization model has many advantages compared to the widely used, exchange rate based approach, but also some drawbacks, specifically:

1. **It's impossible to transfer the whole balance at once:** Given the perpetual accrual of the interest rate, there is no way to specify the exact amount to transfer, since the interest will keep accruing even while the transfer transaction is being confirmed. This means that having exactly 0 balance after a transfer is impossible, rather, a very small balance (dust balance) will be left to the from account executing the transfer. Note that this could have been avoided by adding specific logic to handle this particular edge case, but this would have meant adding a non standard behavior to the ERC20 transfer function, and for this reason we avoided it. Even though this is not a relevant issue, it's important to note that is possible to completely clear the remaining balance by either 1. execute another transfer, which will most likely transfer the remaining dust balance as it would be too small to accrue interest in a reasonably short amount of time, or 2. redeem the dust balance and transfer the underlying asset.
2. **Interest stream can only be redirected if there is a principal balance:** This means that only accounts that have a principal balance B_p can redirect their interest. If users redeem or transfer everything, their interest redirection is reset. As a side effect of this, interest generated only by the redirected balance B_r cannot be redirected.

4 Stable Rate Theory

The following chapter explains how the stable rates are applied to the system and the limitations.

Implementation of a fixed rate model on top of a pool is complicated. Indeed, fixed rates are hard to handle algorithmically, as the cost of borrowing money varies with market conditions and the liquidity available. There might therefore be situations (sudden market changes, bank runs ...) in which handling stable rate borrow positions would need using specific heuristics based on time or economical constraints. Following this reasoning, we identified two possible ways of handling fixed rates:

1. **Imposing time constraints**: fixed rates might work perfectly fine in a time constrained fashion. If a loan has a stable duration, it should survive extreme market conditions, as the borrower must repay at the end of the loan period. Unfortunately, time constrained fixed rate loans aren't suitable for our specific use case of open ended loan. It would require a certain degree of UX friction where users would need to create and handle multiple loans with different times constraints.

2. **Imposing rates constraints**: An interest rate calculated at the beginning of a loan might be impacted by market conditions, keeping it from staying fixed. If the rate diverges too much from the market, it can be readjusted. This would not be a pure fixed rate, open term loan - as the rate might vary throughout the loan duration – yet users will experience actual fixed rates during specific time periods, or when there is enough liquidity available. This particular implementation has been chosen to be integrated into Aave's Protocol under the name **stable rate**.

4.1 Lending Rate Oracle

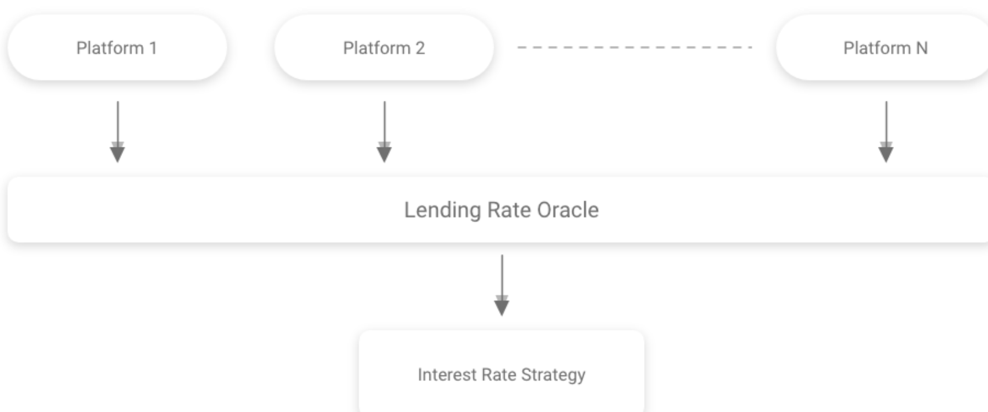

Figure 13: Lending Rate Oracle

The first component to be integrated into the Protocol protocol is a Lending Rate Oracle, which will provide information to the contracts on the actual market rates that other lending platforms, both centralized and decentralized, are providing. The average market lending rate M_r is defined for i platforms with P_r^i the lending rate and P_v^i the borrowing volume:

$$M_r = \frac{\sum_{i=1}^{n} P_r^i P_v^i}{\sum_{i=1}^{n} P_v^i}$$

The market rate will be updated daily, initially by Aave.

4.2 Current Stable Borrow Rate R_s

The current stable borrow rate is calculated as follows:

$$R_s^t = \begin{cases} M_r + \frac{U}{U_{optimal}} R_{slope1}, \text{ if } U \leq U_{optimal} \\ M_r + R_{slope1} + \frac{U - U_{optimal}}{1 - U_{optimal}} R_{slope2}, \text{ if } U > U_{optimal} \end{cases}$$

With:
- M_r the average market lending rate.
- R_{slope1} the interest rate slope below $U_{optimal}$, increases the rate as U increases.
- R_{slope2} the interest rate slope beyond $U_{optimal}$, increases as the difference between U and $U_{optimal}$ increases.
- U is the utilization rate.

Note: R_s does NOT impact existing stable rates positions – this is applied only to new opened positions.

4.3 Limitations on Stable Rate Positions

To avoid abuses on stable rate loans, the following limitations have been applied to the stable rate borrowing model:

1. Users cannot deposit as collateral more liquidity than what they are trying to borrow. Eg. a user deposits 10 million DAI collateral, tries to borrow 1 million DAI. This is to prevent the following attack vector:

   $$\text{Given: } B_s = 18\%APR,\ M_r = 9\%APR,\ R_l = 12\%APR$$

   Users might try to artificially lower B_s to the value of M_r by depositing a huge amount of liquidity which would cause B_s to drop, then borrow from the same liquidity at a lower rate, withdraw the liquidity previously deposited to cause B_s and the liquidity rate R_l to raise again; then finally deposit the amount borrowed to earn interest on the previously borrowed funds. Although this attack can still be carried out using multiple accounts, this particular constraint makes the attack more complicated as it requires more money (and a different collateral currency). This works well in combination with the interest rate rebalancing in the next section.

2. Borrowers will only be able to borrow up to T_r of the available liquidity at the current borrow rate. So, for every specific value of B_s, there is only up to T_r of liquidity available for a single borrower. This is to avoid that a specific borrower would borrow too much available liquidity at a too competitive rate.

4.4 Stable Rate Rebalancing

The last and perhaps most important constraint of the stable rate model is the rate rebalancing. This is to work around changes in market conditions or increased cost of money within the pool.
The stable rate rebalancing will happen in two specific situations:

1. **Rebalancing up**. The stable rate of a user x is rebalanced to the most recent value of B_s when a user could earn interest by borrowing:

   $$B_s^x < R_l \text{ with } B_s^x \text{ the stable borrow rate of user } x$$

2. **Rebalancing down**. The stable rate of a user x is rebalanced to the most recent value of B_s, if:

   $$B_s^x > B_s(1 + \Delta_{Bs})$$

   with Δ_{Bs} a rate delta established by governance which defines the window above B_s to rebalance interest rates. If a user pays too much interest beyond that range, the rate is balanced down.

4.5 The Rebalancing Process

The LendingPool contract exposes a function rebalanceStableBorrowRate(address reserve, address_user) which allows to rebalance the stable rate interest of a specific user. Anybody can call this function: however, there isn't any direct incentive for the caller to rebalance the rate of a specific user. For this reason, Aave will provide an agent that will periodically monitor all the stable rates positions and rebalance the ones that will be deemed necessary. The rebalance strategy will be decided offchain by the agent, this means that users that satisfy the rebalance conditions may not be rebalanced immediately. Since those conditions depend on the liquidity available and the state of market, there might be some transitory situations in which an immediate rebalance is not needed.

This does not add any element of centralization to the protocol. Even if the agent stops working, anybody can call the rebalance function of the LendingPool contract. Although there isn't any direct incentive in doing it ("why should I do it?") there is an indirect incentive for the ecosystem. In fact, even if the agent should cease to exist, depositors might still want to trigger a rebalance up of the lowest borrow rate positions, to increase the liquidity rate and/or force borrowers to close up their positions, increasing the available liquidity. In case of a rescale down, instead, borrowers have a direct incentive in performing a rebalance of their positions to lower the interest rate.

The following flowchart explains the sequence of actions of the function rebalanceStableBorrowRate(). The compounded balance that is accumulated until the instant at which the rebalance happens, is not affected by the rebalance.

Figure 14: Rebalancing

5 Conclusion

The **Aave Protocol** relies on a lending pool model to offer high liquidity. Loans are backed by collateral and represented by aTokens, derivative tokens which accrue the interests. The parameters such as interest rate and Loan-To-Value are token specific.

Aave improves Decentralized Finance's current offering, bringing two key innovations to the lending ecosystem:

- **Stable Rates** to help borrowers' financial planning;
- **Flash Loans** to borrow without collateral during a single transaction.

Following the launch of the mainnet, Aave will uphold its commitment to decentralization through additional features. The Pool Factory will allow anyone to launch their own lending pool based on our smart-contracts. Governance will be on-chain with rights represented by:

- The LEND token at Protocol level for updates of the smart contract;
- aTokens at Pool level for pool specific parameters.