Protocol Whitepaper
V2.0

wow@aave.com

December 2020

Abstract

This document describes the definitions and theory behind the Aave Protocol V2, including how it improves V1, new features and different aspects of the implementation.

Contents

1 Introduction 1

2 Architectural Specifications 2

3 Features Specifications 3
3.1 aTokens 3
3.2 Debt Tokenization 4
3.3 Variable Debt 4
3.4 Stable Debt 5
3.5 Flash Loans V2 5
3.6 Collateral trading 6
3.7 Repay with collateral 7
3.8 Credit Delegation 7

4 Gas optimisation and improvements 8
4.1 Implementation of the `pow` function 8
4.2 Removing `SafeMath` from `WadRayMath` and `PercentageMath` 8
4.3 Mapping users' loans/deposits with a `bitmask` 8
4.4 Reserve configuration with a `bitmask` 9

5 Formal Definitions 10

1 Introduction

The Aave Protocol marks a shift from a decentralised P2P lending strategy (direct loan relationship between lenders and borrowers, like in ETHLend) to a pool-based strategy. 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.

The following paper describes the key functionalities that differentiate the Aave Protocol V2 from V1. The main driver for the development of Aave V2 was to improve the sub-optimal solutions implemented in V1, precisely:

- Inability to upgrade the aTokens
- Gas inefficiency
- Architecture simplification favoring automated testing through fuzzers and formal verification tools
- Code simplification

Some of the solutions currently implemented in V1 were designed and developed when the Ethereum Network was notably different. The exponential growth of DeFi has grown the number of transactions twofold, with further pressure from increased gas costs due to the Istanbul release.

Moreover, Aave V2 offers the following new features:

1. **Debt Tokenisation.** The borrowers debt is now represented by tokens instead of internal accounting within the contract, enabling:
   - Code simplification with regards to the borrowing aspect of the protocol - some of the calculations are now implicit to the minting/burning logic of the debt tokens.
   - Simultaneous borrows with variable and multiple stable rates. In V1, borrowers' loans are either variable or stable with previous borrows automatically switched to the latest loan rate. In V2, users can hold multiple loans combining variable and multiple stable rate positions. Multiple stable rate positions are handled through the weighted average of the rates of the stable rate loans being taken.
   - Native credit delegation - through the concept of delegation, users can delegate to other addresses by opening a credit line. Some interesting tools can be developed on top, for example:
     - Borrow from a cold wallet - users can keep their collateral in cold wallets, simply delegating their credit to a hot wallet to borrow. The whole position can be managed from the hot wallet, with borrow, repay and add more collateral; while funds are kept under the safety of a cold wallet.
     - Credit delegation facilities - users can take undercollateralised loans as long as they receive delegation from other users that provide collateral.
     - Automated yield farming - credit delegation makes it possible to build tools to to automatically open credit lines to *farm* yields on multiple protocols.
2. **Flash Loans V2.** Flash Loans are a disruptive functionality of Aave V1, which allow the creation of a variety of tools to refinance, collateral swap, arbitrage and liquidate. Flash Loans have become a powerful mechanism of DeFi. In the initial V1 Flash Loan, a significant limitation is that it cannot be used within Aave. Aave V2 implements a solution to enable Flash Loans usage in combination with any of the other functionalities of the protocol, powering many new possibilities:
   - Collateral trading. Change the exposure from one/multiple collaterals to another, without the need of closing the debt positions.
   - Repay a loan with the collateral. Use collateral deposit in the protocol to repay a debt position.
   - Margin trading. Creation of margin positions that can be used for trading later on.
   - Debt swap. Change the debt exposure from one asset to another.
   - Margin deposits

2 Architectural Specifications

The diagram bellow describes the high level architecture of Aave V2:

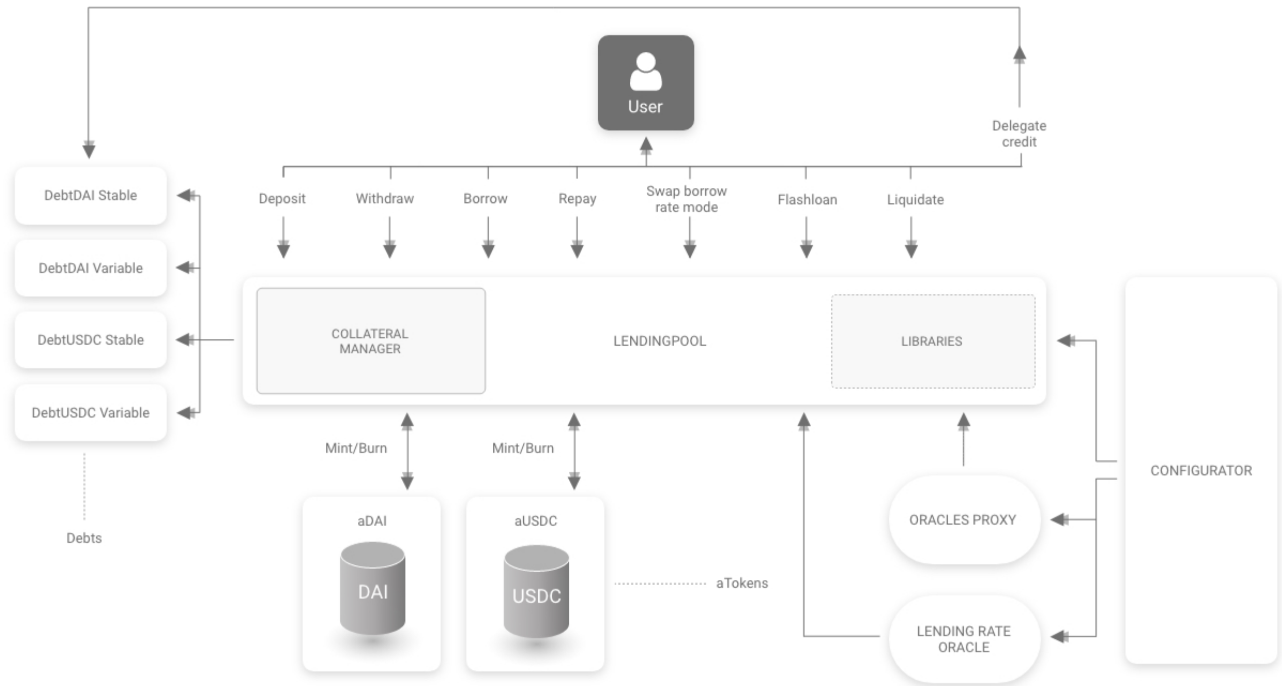

Figure 1: Aave V2 Protocol Architecture

Some major architectural differences distinguish V2 in Figure 1 from the compartimented structure of V1 in Figure 2:

1. Funds that were previously stored in the `LendingPoolCore` contract are now stored within each specific aToken. This gives the protocol better segregation between assets, which favors the implementation of yield farming aware aTokens.

2. `LendingPoolCore` and `LendingPoolDataProvider` have been removed and replaced with libraries. This reduced the gas footprint of all the actions by 15 to 20% while optimising the code complexity and verbosity.

3. All actions now happen through `LendingPool`; replacing the previously required `redeem()` on every single aToken.

4. Debt tokens track users debt.

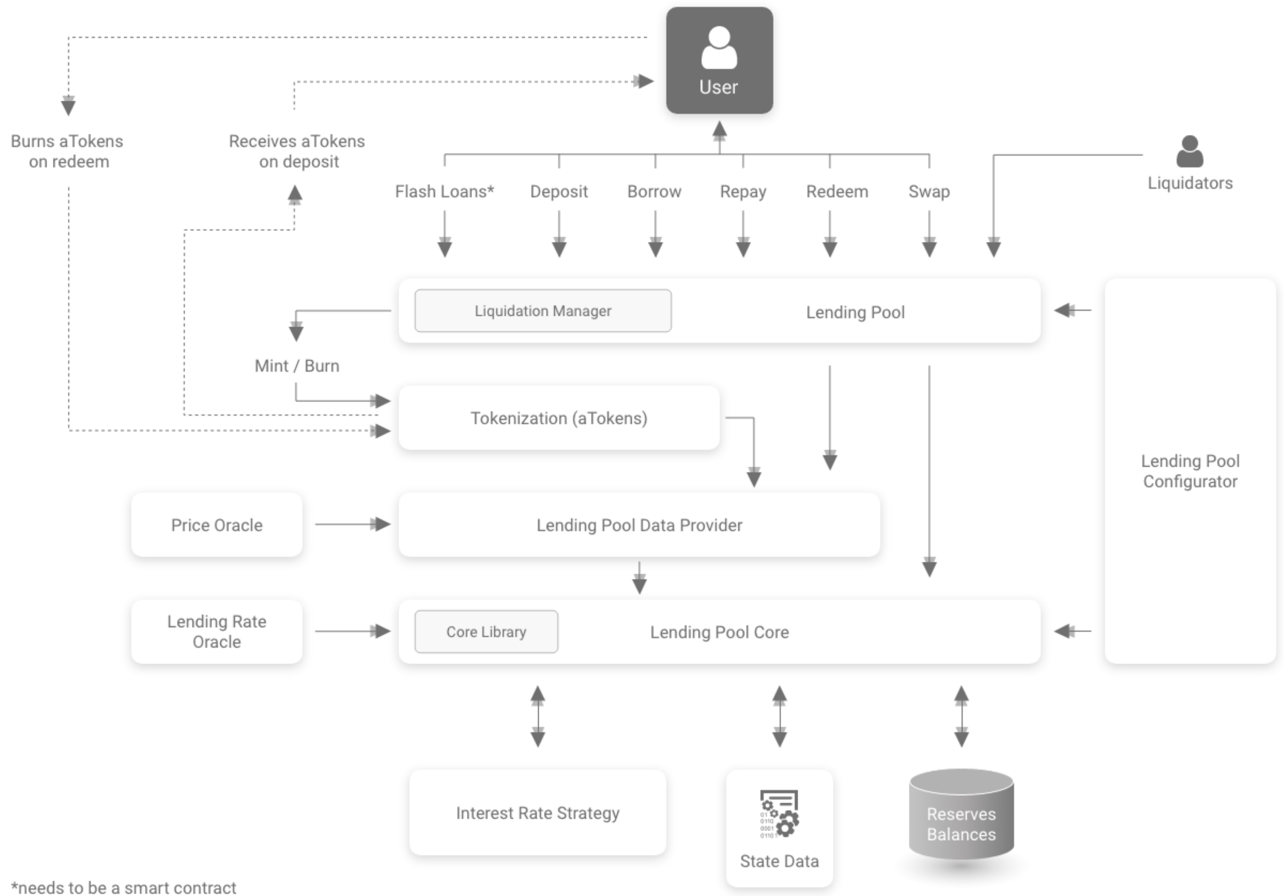

Figure 2: Aave V1 Protocol Architecture

3 Features Specifications

3.1 aTokens

V2 aTokens bring the following updates:

- Support of the `EIP-2612`
- No interest rate redirection in the base aToken implementation - this may be reintroduced in the future

The following definitions from V1 hold true in V2:

- LR_t^{asset}**, the current liquidity rate.**
  A function of the overall borrow rate and the utilisation rate $LR_t = \overline{R}_t U_t$
- LI_t**, 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.

$$LI_t = (LR_t \Delta T_{year} + 1) LI_{t-1}$$
$$LI_0 = 1 \times 10^{27} = 1\ ray$$

- NI_t**, reserve normalised income.**
  Ongoing interest cumulated by the reserve $NI_t = (LR_t \Delta T_{year} + 1) LI_{t-1}$

In V1, NI_t was stored as user x index $NI(x)$ after each action, and for a user x, the cumulated aToken balance was calculated as follows:

$$CB_t(x) = \frac{PB(x)}{NI(x)_t} I_t \text{ with } PB(x) \text{ the principal balance of user } x$$

In V2, the user index factually disappears as a storage variable, and is stored together with the principal balance as a ratio that is called **Scaled Balance**, ScB. The balance of the user is calculated leading to increase or decrease on every action resulting in the mint or burn of aTokens:

- **Deposits.** When a user deposits an amount m in the protocol, his scaled balance updates as follows:

$$ScB_t(x) = ScB_{t-1}(x) + \frac{m}{NI_t}$$

- **Withdrawals.** When a user withdraws an amount m in the protocol, his scaled balance updates as follows:

$$ScB_t(x) = ScB_{t-1}(x) - \frac{m}{NI_t}$$

At any point in time, the aToken balance of a user can be written as:

$$aB_t(x) = ScB_t(x)NI_t$$

3.2 Debt Tokenization

The total supply of debt token including debt accrued per second is defined as follows:

$$dS_t = \sum_{i \in users} ScB_t(i) \text{ with } ScB_t(i) \text{ the amount borrowed by each user i}$$

The total debt on an asset at time t is defined by:

$$D_t^{asset} = SD_t + VD_t \text{ with } SD \text{ the stable debt token supply and } VD \text{ the variable}$$

This definition replaces the total stable or variable borrows of V1, described in section 1.2 of the V1 Whitepaper.

3.3 Variable Debt

The V1 Whitepaper introduces the following definition that hold true for V2:

- **VI_t^{asset}, the cumulated variable borrow index.**
  Interest cumulated by the variable borrows VB during a time ΔT, at variable rate VR, updated whenever a borrow, deposit, repay, redeem, swap, liquidation event occurs.

$$VI_t = (1 + \frac{VR_t}{T_{year}})^{\Delta T} VI_{t-1}$$

- **$VI(x)$, user cumulated variable borrow index**
  Variable borrow index of the specific user, stored when a user opens a variable borrow position.//

$$VI(x) = VI_{t(x)}$$

- **$PB(x)$, 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.

In V2 the debt tokens follow the same ever increasing logic as the aTokens of V1. The variable debt token follow the scaled balance approach. The concept of a normalised variable (cumulated) debt has been introduced:

$$VN_t = (1 + \frac{VR_t}{T_{year}})^{\Delta T} VI_{t-1}$$

With $ScVB_t(x)$ the scaled balance of a user x at time t, m the transaction amount, VN_t the normalised variable debt:

- **Borrows.** When a user x borrows an amount m from the protocol the scaled balance updates

$$ScVB_t(x) = ScVB_{t-1}(x) + \frac{m}{VN_t}$$

- **Repays/Liquidations.** When a user x repays or gets liquidated an amount m the scaled balance updates

$$ScVB_t(x) = ScVB_{t-1}(x) - \frac{m}{VN_t}$$

At any point in time, the total variable debt balance of a user can be written as:

$$VD(x) = ScVB(x)D_t$$

This variable debt token balance replaces the user variable borrow balance of V1, described in section 1.2 of the the V1 Whitepaper.

3.4 Stable Debt

For the stable rate debt, the following V1 definition holds true:

- $\overline{SR}_t^{asset}$**, overall stable rate**
  - When a stable borrow of amount SB_{new} is issued at rate SR_t: $\overline{SR}_t = \frac{SD_t\overline{SR}_{t-1}+SB_{new}SR_t}{SD_t+SB_{new}}$
  - When a user x repays stable borrow i for an amount $SB_i(x)$ at stable rate $SR_i(x)$:

$$\overline{SR}_t = \begin{cases} 0, \text{ if } SD_t - SB(x) = 0 \\ \frac{SD_t\overline{SR}_{t-1}-SB(x)SR(x)}{SD_t-SB_x}, \text{ if } SD_t - SB_x > 0 \end{cases}$$

$\overline{SR}_t$ is stored in the stable rate token for each specific currency. The **stable debt token** $SD(x)$ **balance for a user** x is defined as follows:

$$SD(x) = SB(x)(1 + \frac{\overline{SR}_t}{T_{year)}})^{\Delta T}$$

In V1, $SR(x)$, the stable rate of user x, is always equal to the stable rate of the last loan taken, with previous loans rebalancing upon new loans. From V2, the stable rate $SR(x)$ is calculated per asset reserve across the i stable loans:

$$\overline{SR}(x) = \sum_i \frac{SR_i(x)SD_i(x)}{SD_i(x)}$$

3.5 Flash Loans V2

The flowchart Figure 3 describes V1 Flash Loans. The check if funds have been transferred is performed in the steps:

1. Before transferring the funds to the executor U, the protocol takes a snapshot of the balance of the Flash-Loaned asset
2. At the end of the Flash Loan, a check enforces that the balance of the contract includes the Flash borrowed amount plus the Flash Loan premium. If the amount is incorrect, the transaction is reverted.

This limits the use of Flash Loans within Aave, as allowing other actions (especially deposits, repayments and liquidations) might expose the protocol to the risk of reentrancy. These actions are therefore mutexed by a reentrancy guard mechanism.

V2 offers a novel protection against reentrancy, allowing to use Flash Loans for all Aave transactions.

Solution

The initial implementation was chosen for a specific ETH codepath; unfortunately ETH does not support pull payment strategy. In this new version 2, all the ETH-specific code has been removed, resulting in the following changes:

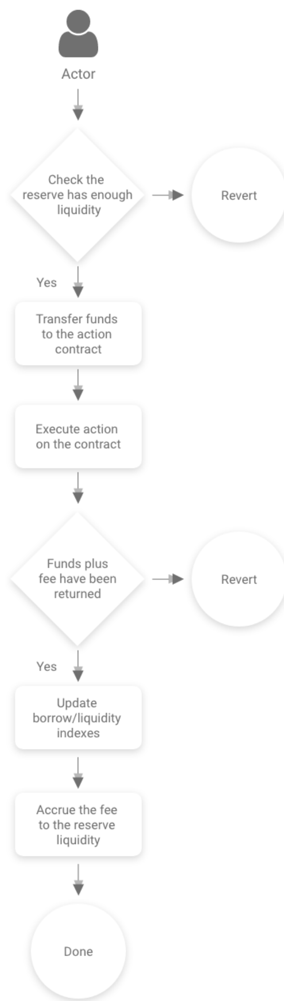

Figure 3: Aave V1 Flash Loan

1. The Flash Loan initiates with the transfer of funds to the Flash Loan executor u,
2. At the end, funds are pulled back from the executor u for an amount equal to the borrowed funds plus the fee. If pulling the funds does not succeed, the Flash Loan fails; for example, when there is lack of approval for the funds or lack of funds to cover the debt.

This technically removes the need to snapshot the balances before and after, therefore removing the need for reentrancy check.

Additionally, Flash Loans now support multiple modes: the Flash Loan caller may decide to keep the loan open, subject to appropriate liquidity to cover collateral requirements.

3.6 Collateral trading

The Aave protocol V2 offers a way of swapping assets deposited, both used as collateral or not. This is achieved by leveraging the v2 Flash Loans in the following way:

– The `flashLoan()` function is called by the user, passing as parameters the address of a receiver contract, implementing the `IFlashLoanReceiver` interface, which is an **untrusted contract** that should be verified by the user in advance; a list of underlying assets to swap from, a list of amounts of those assets and one extra parameter containing the asset to swap to and the max slippage chosen by the user, both encoded.

– The receiver contract will then use the funds received to swap them to the destination asset, depositing again on behalf of the user and withdrawing the user's deposits in the Flashed-currencies in order to repay the Flash Loan.

One example would be:

(i) An user has 100 LINK and 20 UNI deposited in the protocol, with a debt of 100 USDC. He wants to swap both his deposited LINK and UNI to AAVE, without needing to pay back any debt within one transaction.

(ii) The user calls the `flashLoan()` function passing as parameters the address of the receiver contract which contains the logic to do the operation, the addresses of LINK and UNI; 100 and 200 as amounts and, encoded, the address of the asset to swap to (AAVE) and 2% as maximum slippage on the trades.

(iii) The receiver contract will swap the indicated amounts of LINK and UNI to AAVE, by using a decentralised exchange.

(iii) The receiver contract will deposit the resultant AAVE on behalf of the user in the Aave Protocol.

(iv) The receiver contract will transfer the equivalent Flashed amount in aLINK and aUNI from the user, will redeem them to LINK and UNI and will approve the pool to pull those funds at the end of the Flash Loan to repay.

3.7 Repay with collateral

Also built by using Flash Loans v2, this feature allows for an user with one or multiple assets deposited as collateral in the protocol, to use them to repay partially or totally his debt/s positions. Same as with `Swap Liquidity` and, in general all the features based on Flash Loans, the repayment with collateral uses the `flashLoan()` function and a receiver contract, implementing the `IFlashLoanReceiver` interface. To this receiver is passed a list collateral assets to flash and use to swap and repay, a list of amounts of those assets, and, encoded, a list of the asset to repay debt, a list of debt amounts to repay, and a list with the borrow modes (stable or variable) for each debt asset. It's important to note that, in this case, the receiver contract will expect to receive an exact amount of **how much needs to be repaid**, differently to `Swap Liquidity`, where the amount expected is the exact collateral to swap. The flow on this feature would be:

(i) An user has 100 AAVE deposited in the protocol, and a debt of 200 USDC at variable rate. As he doesn't have at the moment USDC funds available to repay his loan, he wants to swap part of his AAVE collateral to USDC and use it for the repayment.

(ii) The user calls the `flashLoan()` function passing as parameters: the address of the receiver contract which contains the logic to do the operation, the address of AAVE, 7 as collateral amount to swap (estimated AAVE needed to cover the 200 USDC debt), 200 as debt amount to repay and variable as borrow mode used.

(iii) The receiver contract will swap the 7 AAVE to USDC by using a decentralised exchange.

(iv) The receiver contract will use the resultant USDC from the swap to repay on behalf of the user his USDC debt in the protocol.

(iv) Once the debt is repaid, the receiver will transfer the amount of aAAVE needed to return the flashed AAVE from the user, will redeem it for AAVE, and will approve the pool to pull those funds at the end of the Flash Loan transaction.

(v) If at the end there are any leftovers of the 7 AAVE as result of the swap, these funds will be deposited back in the protocol on behalf of the user or sent directly to his wallet.

3.8 Credit Delegation

On top of debt tokenisation, V2 supports credit delegation: the `borrow()` function supports credit lines to different addresses, without the need for collateral, as long as the caller address as been granted the allowance.

This functionality is implemented through the function `approveDelegation()` on each debt token. Users will be able to draw up to their allowance for the specific debt mode (stable or variable). The `borrow()` function

has an `onBehalfOf` parameter for the caller to specify the address used to draw credit.

The implementation of credit delegation required some trade-offs:

- A delegator can delegate credit to multiple entities but a delegatee can only draw credit from a single delegator at once. One cannot aggregate the delegators debt in a single `borrow()`
- A delegator can simultaneously delegate stable and variable credit to an entity, but a delegatee cannot draw variable and stable credit from a single `borrow()`

4 Gas optimisation and improvements

4.1 Implementation of the `pow` function

In the initial release, the calculation of compounded interest relied on the exponential formula implemented with the Babylonian method, resulting in more time consuming and expensive execution.

The V2 release, optimises execution costs by approximating this exponential formula with a binomial expansion, which works well with a small base. The implementation used the following binomial expansion:

$$(1+x)^\alpha = 1 + \alpha x + \frac{1}{2}\alpha(\alpha-1)x^2 + \frac{1}{6}\alpha(\alpha-1)(\alpha-2)x^3 + \frac{1}{24}\alpha(\alpha-1)(\alpha-2)(\alpha-3)x^4 + ...$$

The function `calculateCompoundedInterest, (MathUtils.sol line 46)` implements the first three expansions which gives a good approximation of the compounded interest for up to a 5 year loan duration. This results in a slight underpayment offset by the gas optimisation benefits.

It's important to note that this behaves a little differently for variable and stable borrowing:

- For variable borrows, interests are accrued on any action of any borrower;
- For stable borrows, interests are accrued only when a specific borrower performs an action, increasing the impact of the approximation. Still, the difference seems reasonable given the savings in the cost of the transaction.

4.2 Removing `SafeMath` from `WadRayMath` and `PercentageMath`

The V1 `WadRayMath`library internally uses `SafeMath` to guarantee the integrity of operations. After an in depth analysis, `SafeMath` incurred intensive high costs in critical areas of the protocol, with a 30 gas fee for each call. This supported the refactoring of `WadRayMath` to remove `SafeMath`, which saves 10-15k gas on some operations.

4.3 Mapping users' loans/deposits with a `bitmask`

In the initial V1 release, the protocol loops through all the active assets to identify the user deposits and loans. This would result in high gas consumption and reduced scalability - as the cost of withdrawing/borrowing/repaying/liquidating assets would increase as more assets are listed on the protocol. Two improvement ideas where considered:

1. For each user, keep a list of assets being used as collateral/borrowed, updated whenever a user deposits/withdraws/borrows/repays. When calculating the total system collateral, rather than looping through all the supported assets, the function `calculateUserAccountData()` could target specific user's assets.
   This solution doesn't present any limitations, though it is more gas intensive - when considering costs of removing items from the list, checking each aToken balance, stable or variable debt, whether a user is depositing, borrowing or both...

Figure 4: Users Collaterals/Borrows Bitmask Structure

2. Create a `bitmask` with the structure figure 4. The `bitmask` has a 256 bit size, it is divided in pairs of bits, one for each asset. The first bit of the pair indicates if an asset is used as collateral by the user, the second whether an asset is borrowed by the user. This implementation imposes the constraints:
   - Only 128 assets can be supported, to add more, another `uint256` needs to be used.
   - For the calculation of the account data, the protocol would still need to query all listed assets.

   It presents the following advantages compared to a list based solution:
   - Additional assets cost only 5k gas, instead of 20k gas for writing on the list
   - Extremely cheap verification of a user's asset borrowing (with `0xAAAAA...!=0`) or depositing (`configuration=0`)
   - Immediate access to what assets are being deposited/borrowed/both - by fetching the aTokens/debt tokens balances
   - In `calculateUserAccountData()`, the configuration can be cached at the beginning and used to perform all the calculations, which greatly saves on SLOADs.

Both solutions were implemented and tested, the second solution was preferred since the 128 asset limitation provides space for growth, and more so in light of Aave's multimarkets.

4.4 Reserve configuration with a `bitmask`

A `bitmask` has also been introduced to store the reserve configuration, defined in Figure 5. A similar packing could have been achieved by using `uint32` and `booleans`, the `bitmask` benefits from more gas efficiency, and more so when updating multiple configurations at once.

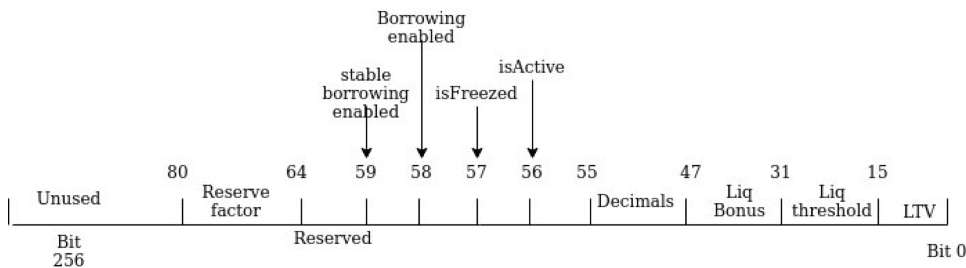

Figure 5: Reserve Bitmask Structure

5 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^{asset}, **total liquidity of an asset**	Total amount of liquidity available in the assets reserve. The decimals of this value depend on the decimals of the currency.	
SD_t^{asset}, **total stable debt token**	Total amount of liquidity borrowed at a stable rate represented in debt tokens.	
VD_t^{asset}, **total variable debt tokens**	Total amount of liquidity borrowed at a variable rate represented in debt tokens.	
D_t^{asset}, **total debt**	Total amount of liquidity borrowed.	$D_t^{asset} = VD_t^{asset} + SD_t^{asset}$
U_t^{asset}, **utilisation rate**	The utilisation of the deposited funds.	$U_t^{asset} = \begin{cases} 0, \text{ if } L_t^{asset} = 0 \\ \frac{D_t^{asset}}{L_t^{asset}}, \text{ if } L_t^{asset} > 0 \end{cases}$
$U_{optimal}^{asset}$, **target utilisation rate**	The utilisation rate targeted by asset reserve, beyond the interest rates rise sharply.	
$\#R_{base}^{asset}$, **base borrow rate**	Constant for $B_t^{asset} = 0$. VR for variable rate SR for stable.Expressed in ray.	
$\#R_{slope1}^{asset}$, **interest rate slope below** $U_{optimal}$	Constant representing the scaling of the interest rate versus the utilisation, when $U < U_{optimal}$. VR for variable rate SR for stable. Expressed in ray.	
$\#R_{slope2}^{asset}$, **interest rate slope above** $U_{optimal}$	Constant representing the scaling of the interest rate versus the utilisation, when $U \geq U_{optimal}$. VR for variable rate SR for stable. Expressed in ray.	
$\#R_t^{asset}$, **borrow rate**	$\#R_t^{asset} = \begin{cases} \#R_{base}^{asset} + \frac{U_t^{asset}}{U_{optimal}^{asset}} \#R_{slope1}^{asset}, \text{ if } U_t^{asset} < U_{optimal}^{asset} \\ \#R_{base}^{asset} + \#R_{slope1}^{asset} + \frac{U_t^{asset} - U_{optimal}}{1 - U_{optimal}} \#R_{slope2}^{asset}, \text{ if } U_t^{asset} \geq U_{optimal} \end{cases}$	

Variable	Description	
VR_t^{asset}, **variable rate**	The rate of variable borrows based on the formula above. In ray.	
SR_t^{asset}, **stable rate**	The rate of stable borrows based on the formula above. In ray.	
$\overline{SR}_t(x)$, **overall stable rate of a user** x	Overall stable rate of an user across i loans	$\overline{SR}(x) = \sum_i \frac{SR_i(x)SD_i(x)}{SD_i(x)}$
$\overline{SR}_t^{asset}$, **overall stable rate**	When a stable borrow of amount SB_{new} is issued at rate SR_t:	$\overline{SR}_t = \frac{SD_t\overline{SR}_{t-1}+SB_{new}SR_t}{SD_t+SB_{new}}$
	When a user x repays an amount $SB(x)$ at stable rate $SR(x)$:	$\overline{SR}_t = \begin{cases} 0, \text{ if } SD - SB(x) = 0 \\ \frac{SD\overline{SR}_{t-1}-SB(x)SR(x)}{SD_t-SB(x)}, \text{ if } SD - SB(x) > 0 \end{cases}$
	Check the methods `decreaseTotalBorrowsStableAndUpdateAverageRate()` and `increaseTotalBorrowsStableAndUpdateAverageRate()`. Expressed in ray.	
$SD_t^{asset}(x)$, **stable debt balance per asset of user x**		$SD_t(x) = SB_t(x)(1 + \frac{\overline{SR}_t}{T_{year}})$
$\overline{R}_t^{asset}$, **overall borrow rate**	Overall borrow rate of the asset reserve, calculated as the weighted average of variable VD_t and stable borrows SD_t borrows.	$\overline{R}_t = \begin{cases} 0, \text{ if } D_t = 0 \\ \frac{VD_tVR_t+SD_t\overline{SR}_t}{D_t}, \text{ if } D_t > 0 \end{cases}$
LR_t, **current liquidity rate**	Function of the overall borrow rate $\overline{R}$ and the utilisation rate U.	$LR_t = \overline{R}_tU_t$
LI_t, **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.	$LI_t = (LR_t\Delta T_{year} + 1)LI_{t-1}$ $LI_0 = 1 \times 10^{27} = 1\ ray$
NI_t, **reserve normalised income**	Ongoing interest cumulated by the reserve.	$NI_t = (LR_t\Delta T_{year} + 1)LI_{t-1}$
VI_t, **cumulated variable borrow index**	Interest cumulated by the variable borrows VB, at rate VR, updated whenever a borrow, deposit, repay, redeem, swap, liquidation event occurs.	$VI_t = (1 + \frac{VR_t}{T_{year}})^{\Delta T_x}VI_{t-1}$ $VI_0 = 1 \times 10^{27} = 1\ ray$
$VI(x)$, **user cumulated variable borrow index**	Variable borrow index of the specific user, stored when a user opens a variable borrow position.	$VI(x) = VI_{t(x)}$
$VN_t(x)$, **user normalised variable debt**		$VN_t(x) = (1 + \frac{VR_t}{T_{year}})^{\Delta T(x)}VI_{t-1}(x)$

Variable	Description		
$PB(x)$, **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.		
$ScB(x)$, **scaled balance of user x**		For a deposit:	$ScB_t(x) = ScB_{t-1}(x) + \frac{m}{NI_t}$
		For a withdrawal:	$ScB(x)_t = ScB_{t-1}(x) - \frac{m}{NI_t}$
$aB(x)$, **user aToken Balance**			$aB(x) = ScB(x)_{t-1}NI_t$
$ScVB(x)$, **variable scaled balance of user** x		For a deposit:	$ScVB_t(x) = ScVB_{t-1}(x) + \frac{m}{VN_t(x)}$
		For a withdrawal:	$ScVB_t(x) = ScVB_{t-1}(x) - \frac{m}{VN_t(x)}$
$VD_t(x)$, **variable debt balance of user x**			$VD_t(x) = ScVB_t(x)D_t$
HF, **health factor**	when $HF < 1$, a loan is considered undercollateralised and can be liquidated		$HF = \frac{CollateralinETH^{asset} * LT^{asset}}{CB^x + TotalFeesETH}$