Tapio V1.5: Model

Nuts Finance

October 2023

Abstract

Tapio is new DeFi protocol that proposes a StableSwap AMM for Liquid Staking Derivatives "LSDs". This document explains the stableSwap model used by Tapio and the main functions proposed by this AMM for the user.

Contents

1 StableSwap model

The StableSwap AMM aims to provide low slippage when trading assets that are priced closely such as stablecoins (e.g. USDC, DAI, USDT, etc) or liquid staking derivatives (e.g. ETH, stETH, etc). In order to understand the StableSwap model it is interesting to study constant sum and constant product invariants. Indeed, the StableSwap bonding curve is a combination of these curves.

1.1 Constant Sum Invariant

A constant sum invariant algorithm is described as:

$$X + Y = C \tag{1}$$

where X and Y are pooled tokens and C is the invariant.

A trader wants to sell ΔX X tokens. Then:

$$X_0 + \Delta X + Y = C \tag{2}$$

$$Y = C - X_0 - \Delta X \tag{3}$$

$$Y_0 - \Delta Y = C - X_0 - \Delta X \tag{4}$$

$$\Delta Y = Y_0 - C + X_0 + \Delta X \tag{5}$$

$$\Delta Y = \Delta X \tag{6}$$

The price is given by:

$$P = \frac{\Delta Y}{\Delta X} = 1 \tag{7}$$

With a constant sum invariant, the price will always remain the same regardless of the balances in the pools. There is zero slippage (infinite leverage), so the pool can be drained completely.

1.2 Constant Product Invariant

A constant product invariant algorithm is described as:

$$X * Y = K \tag{8}$$

A trader wants to sell ΔX X tokens. Then:

$$(X_0 + \Delta X) * (Y_0 - \Delta Y) = K \tag{9}$$

$$\Delta Y = Y_0 - \frac{K}{X_0 + \Delta X} \tag{10}$$

$$\Delta Y = \frac{K\Delta X}{X_0(X0 + \Delta X)} \tag{11}$$

The price is given by:

$$P = \frac{\Delta Y}{\Delta X} = \frac{K}{X_0(X0 + \Delta X)} \tag{12}$$

When the balance of tokens X is increased due to trades, the cost for a Y token (Y supply decreased) became more expensive for K to remain the same. The price and the slippage depends on the $xy = k$ equation and depth of the pool. With the constant product invariant, the pool auto-adjusts the price ensuring that there is always liquidity.

1.3 StableSwap Invariant

On the one hand, The constant product invariant is self-regulating, but it makes it expensive to bring the pool out of balance. On the other hand, the constant sum invariant have no slippage, so the pool can run out of tokens or make the pool very unbalanced. The stableSwap invariant tries to achieve a compromise between Low slippage and Stable and Balanced pool.

The StableSwap compromise is that the pool can slide up or down the constant sum invariant curve only when pools are pretty balanced and the price is stable around 1. When the pool become unbalanced, the AMM needs to make it expensive to continue swapping like a constant product invariant does. The main idea consists in creating a linear invariant around 1, which becomes a product invariant as the tokens deviate away from 1.

The StableSwap invariant combines the sum invariant $\sum_i X_i = D$ with the product invariant $\prod_i X_i = (\frac{D}{n})^n$ where D is the total number of coins when they have an equal price:

$$\sum_i X_i + \prod_i X_i = D + (\frac{D}{n})^n \tag{13}$$

For $n = 2$, The graph looks almost identical to the graph of the product constant invariant. This is because the constant sum invariant is not amplified. To make the curve more effective, The constant sum invariant needs to be multiplied by a factor of χ.

$$\chi \sum_i X_i + \prod_i X_i = \chi D + (\frac{D}{n})^n \tag{14}$$

The point of χ is to give more importance to the low slippage part of the equation. Ideally the the curve should be linear when the pool is equally balanced, so χ should be a function of the number of X and Y tokens:

- χ will decrease when the pool is out of balance, leading to a steeper curve like the product invariant curve.
- χ will increases when the pool is balanced, leading to a linear curve like the sum invariant curve with no slippage.

To make the leverage factor χ dimensionless, the sum invariant is multiplied by D^{n-1}. Then, the formula becomes:

$$\chi D^{n-1} \sum_i X_i + \prod_i X_i = \chi D^n + (\frac{D}{n})^n \tag{15}$$

To make χ dynamic (i.e. χ adapts to the relative ratio of tokens in the pool), we chose:

$$\chi = \frac{A \prod_i X_i}{(D/n)^n} \tag{16}$$

where A is a fixed constant and chosen by the StableSwap protocol team.

The value of χ in a balanced pool is equal to A, and it approaches closer to 0 as pools get more unbalanced.

Substituting χ, the StableSwap invariant becomes:

$$An^n \sum_i X_i + D = ADn^n + \frac{D^{n+1}}{n^n \prod_i X_i} \tag{17}$$

When the pool is unbalanced , we have to solve the equation for D, either analytically (for 2 coins) or iteratively for higher dimensions.

The process to run a StableSwap pool is as follows:

- Decide on the amplification coefficient A.
- Add tokens and Calculate D.
- When a trade is done, say X is deposited, apply fees.
- Solve the equation for Y, given the current parameters A, D and X.
- Recalculate D.

1.4 Calculation of D: *getD*()

To determine D that satisfies Eq.17, The Newton method is used.

Let function $f(D)$ be defined by:

$$f(D) = ADn^n + \frac{D^{n+1}}{n^n \prod_i X_i} - An^n \sum X_i - D \tag{18}$$

Then :

$$f^{'}(D) = An^n + \frac{D^n}{n^n \prod_i X_i} - 1 \tag{19}$$

Let D_k be the D value at iteration K. So, D_{k+1} is given by:

$$D_{k+1} = D_k - \frac{f(D_k)}{f'(D_k)} \tag{20}$$

$$D_{k+1} = \frac{An^n \sum_i X_i + n\frac{D_k^{n+1}}{n^n \prod_i X_i}}{(An^n - 1)D_k + (n+1)\frac{D_k^{n+1}}{n^n \prod_i X_i}} D_k \tag{21}$$

To Simplify, we consider the following notation:

$a = An^n$, $s = \sum_i X_i$, $p = \frac{D_k^{n+1}}{n^n \prod_i X_i}$
Then, D_{k+1} is given by:

$$D_{k+1} = \frac{(as + np)D_k}{(a-1)D_k + (n+1)p} \tag{22}$$

The newton algorithm is used to calculate D based on Eq.22 with $D_0 = \sum_i X_i$. The stopping criterion is a target convergence error (i.e. $|d_k - d_{k-1}| \leq \epsilon = 10^{-18}$) and/or a maximum number of iterations (e.g. $N_{max} = 255$).

1.5 Calculation of Y : $getY()$

The objective is to calculate the new balance Y of token j given the new balance of token i.

Starting from Eq.17 and multplying it by X_j, we obtain:

$$X_j^2 + X_j(b - D) = c \tag{23}$$

where $s = \sum_{n \neq j} X_n$, $p = \prod_{n \neq j} X_n$, $b = s + D/An^n$, $c = D^{n+1}/(n^n pAn^n)$.

D is calculated based on the actual balance of token i, while $\sum_{n \neq j} X_n$, $\prod_{n \neq j} X_n$ are calculated based on the new balance of token i.

Let $f(Y)$ be defined as:

$$f(Y) = Y^2 + Y(b - D) - c \tag{24}$$

Then,

$$f^{'}(Y) = 2Y^{'} + (b - D) \tag{25}$$

Let Y_k be the Y value at iteration k. So, Y_{k+1} is given by:

$$Y_{k+1} = \frac{Y_k^2 + c}{2Y_k + b - D} \tag{26}$$

The newton algorithm is used to calculate D based on Eq.26 with $Y_0 = D$. The stopping criterion is a target convergence error (i.e.

$|Y_k - y_{k-1}| \leq \epsilon = 10^{-18}$) and/or a maximum number of iterations (e.g. $N_{max} = 255$).

2 Tapio AMM

Tapio AMM proposes differents stableSwap pools to exchange LSDs. Each pool is independent, however the different pools shared the same LP token "tapETH". Tapio AMM is based on the contract "StableAsset" that represents a stableSwap pool. Three main actions are possible for the user:

- **mint**: The user provides a liquidity in LSDs into a stableSwap pool to mint tapETH tokens.
- **Swap**: The user exchanges a LSD token a to another LSD token b from a stableSwap pool.
- **redeem**: The user redeems his tapETH token to receive the pool tokens from a stableSwap pool.

Different fee are set by Tapio project and can be updated by the governance:

- **mint fee**: The user pays mint fee by reducing his mint amount of tapETH.
- **Swap fee**: The user pays swap fee by reducing his amount of token output.
- **redeem fee**: The user pays redeem fee either by increasing his redeem amount of tapETH or by reducing his amounts of received tokens.

These different fee are used to increase the totalSupply of tapETH to be distributed to tapETH holders.

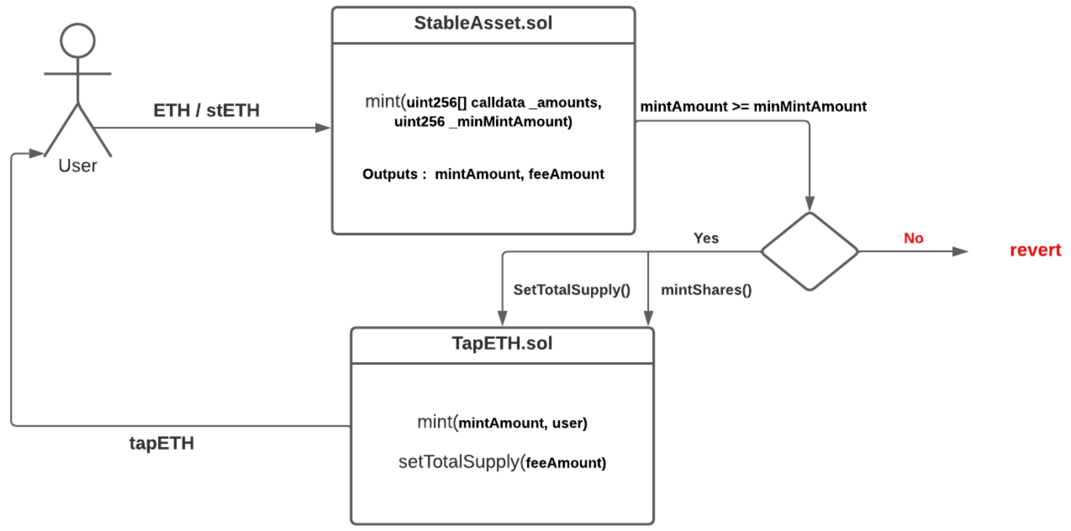
2.1 Mint function

The user can mint tapETH tokens by providing liquidity dX_i to the stableSwap pool. The logic of the function "mint" consists of the following steps:

1. Get A: $getA()$.
2. Calculate $D_{old} = getD(balances, A)$
3. Update token balances: $balances[i] = balances[i] + dX_i$
4. Calculate the $D_{new} = getD(balances, A)$
5. Calculate $\Delta D = D_{new} - D_{old}$
6. Calculate $mintAmount = \Delta D - feeAmount = \Delta D(1 - mintFee)$.
7. Revert if $mintAmount < minMintAmount$.
8. Mint $mintAmount$ of tapETH for the user.

9. Increase the total supply of tapETH by $feeAmount$.

The following UML diagram illustrates the different transactions generated by the function "mint":

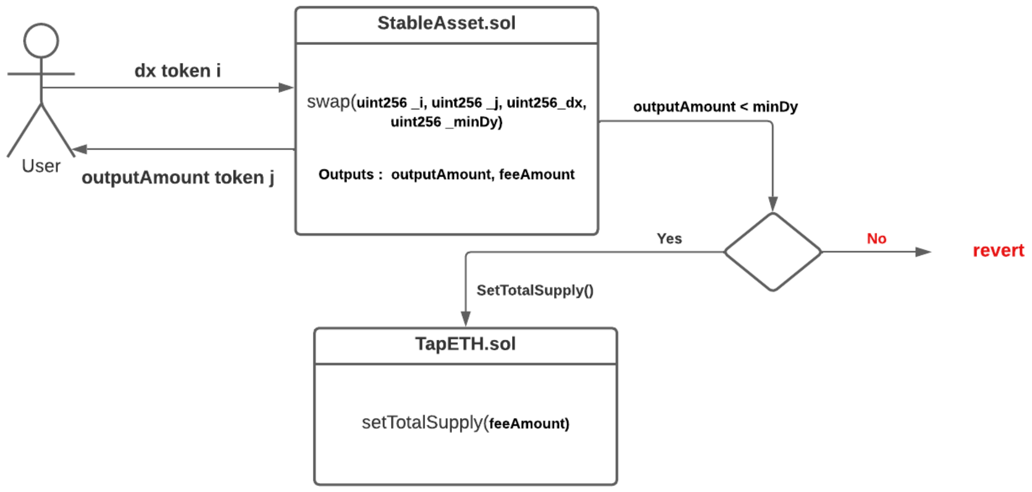

StableAsset.sol
mint(uint256[] calldata _amounts, uint256 _minMintAmount)
Outputs : mintAmount, feeAmount
ETH / stETH
User
mintAmount >= minMintAmount
Yes
No
revert
SetTotalSupply()
mintShares()
TapETH.sol
mint(mintAmount, user)
setTotalSupply(feeAmount)
tapETH

2.2 Swap function

The user can swap dx amount of token i to token j. The Logic of the function "swap" consists of:

1. Get A: $getA()$.
2. Calculate $D_{old} = getD(balances, A)$
3. Update the balance of token i: $balances[i] = balances[i] + dx$.
4. Calculate the new balance of token j: $Y_{new} = getY(balances, j, D_{old}, A)$.
5. Calculate $\Delta Y = Y_{old} - Y_{new}$.
6. Update the balance of token j: $balances[j] = Y_{new}$.
7. Calculate $outputAmount = \Delta Y - feeAmount = \Delta Y(1 - swapFee)$ (swap fee is taken from token output).
8. Revert if $outputAmount < minDy$.
9. Take dx of token i from the user.
10. Send $outputAmount$ of token j to the user.
11. Calculate $D_{new} = getD(realBalances, A)$.
12. Revert if $(D_{old} - D_{new}) \geq feeErrorMargin$
13. Increase the total supply of tapETH by $feeAmount$.

The following UML diagram illustrates the different transactions generated by the function "swap":



StableAsset.sol
dx token i
swap(uint256 _i, uint256 _j, uint256_dx, uint256 _minDy)
outputAmount < minDy
User
outputAmount token j
Outputs : outputAmount, feeAmount
Yes
No
revert
SetTotalSupply()
TapETH.sol
setTotalSupply(feeAmount)

2.3 Redeem function

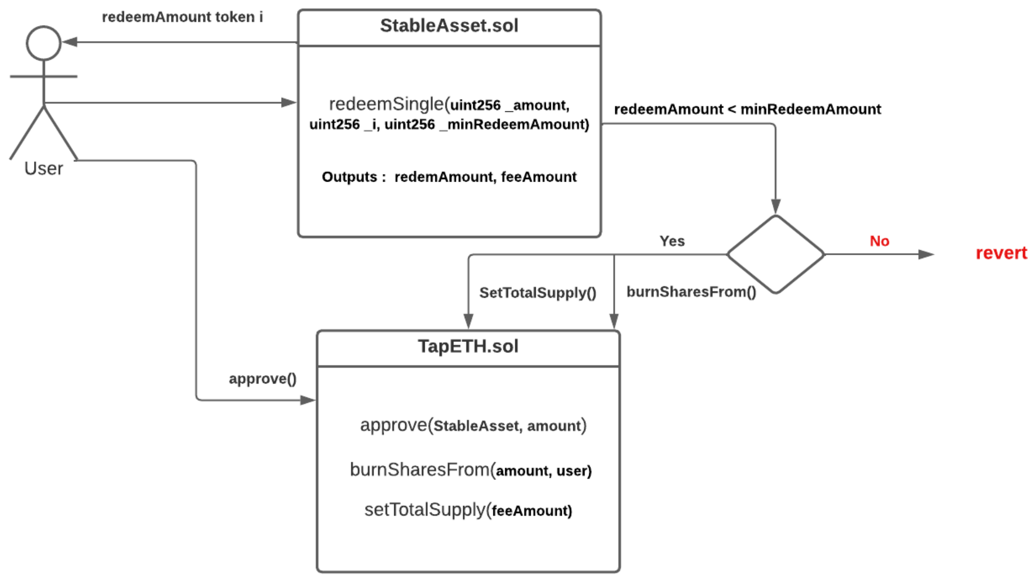
Tapio proposes three different redeem options.

2.3.1 Redeem single side

The user can redeem *amount* of tapETH in order to receive token i. The Logic of the function consists of:

1. Get A: $getA()$.
2. Calculate $D_{old} = getD(balances, A)$
3. Calculate $redeemAmount = amount - feeAmount = amount(1 - redeemFee)$.
4. Update $D_{old} = D_{old} - redeemAmount$.
5. Calculate the new balance of token i: $X_{new} = getY(balances, i, D_{old}, A)$.
6. Calculate $\Delta X = X_{old} - X_{new}$.
7. Revert if $\Delta X < minRedeemAmount$.
8. Burn *amount* of tapETH from the user.
9. Send ΔX of token i to the user.
10. Calculate $D_{new} = getD(realBalances, A)$.
11. Revert if $(D_{old} - D_{new}) \geq feeErrorMargin$
12. Increase the total supply of tapETH by $feeAmount$

The following UML diagram illustrates the different transactions generated by the function "redeemSingle":

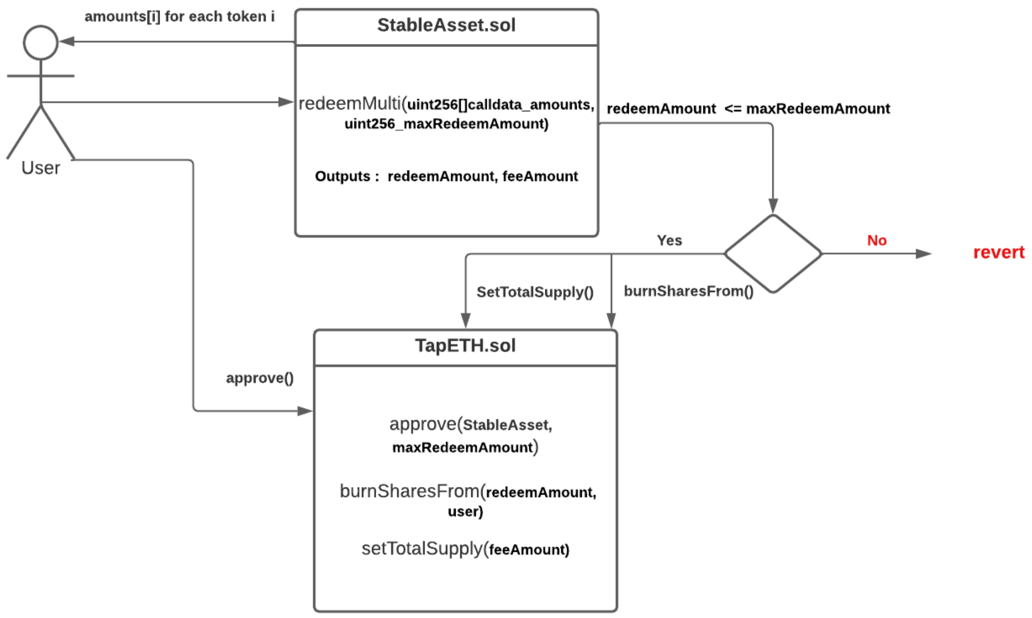

redeemAmount token i
StableAsset.sol
redeemSingle(uint256 _amount, uint256 _i, uint256 _minRedeemAmount)
Outputs : redemAmount, feeAmount
redeemAmount < minRedeemAmount
User
Yes
No
revert
SetTotalSupply()
burnSharesFrom()
TapETH.sol
approve()
approve(StableAsset, amount)
burnSharesFrom(amount, user)
setTotalSupply(feeAmount)

2.3.2 Redeem multi sides

The user can specify the amount of each token that he wants to receive by redeeming his tapEth tokens. The Logic of the function consists of:

1. Get A: $getA()$.
2. Calculate $D_{old} = getD(balances, A)$
3. Update balance of each token i: $balances[i] = balances[i] - amounts[i]$.
4. Calculate $D_{new} = getD(balances, A)$
5. Calculate $\Delta D = D_{old} - D_{new}$
6. Calculate $redeemAmount = \Delta D + feeAmount = \Delta D(1 + redeemFee)$
7. Revert if $redeemAmount > maxRedeemAmount$
8. For each token i, take $amounts[i]$ from the user.
9. update $D_{old} = D_{old} - redeemAmount$.
10. Calculate $D_{new} = getD(realBalances, A)$.
11. Revert if $(D_{old} - D_{new}) \geq feeErrorMargin$.
12. Increase the total supply of tapETH by $feeAmount$.

The following UML diagram illustrates the different transactions generated by the function "redeemMulti":

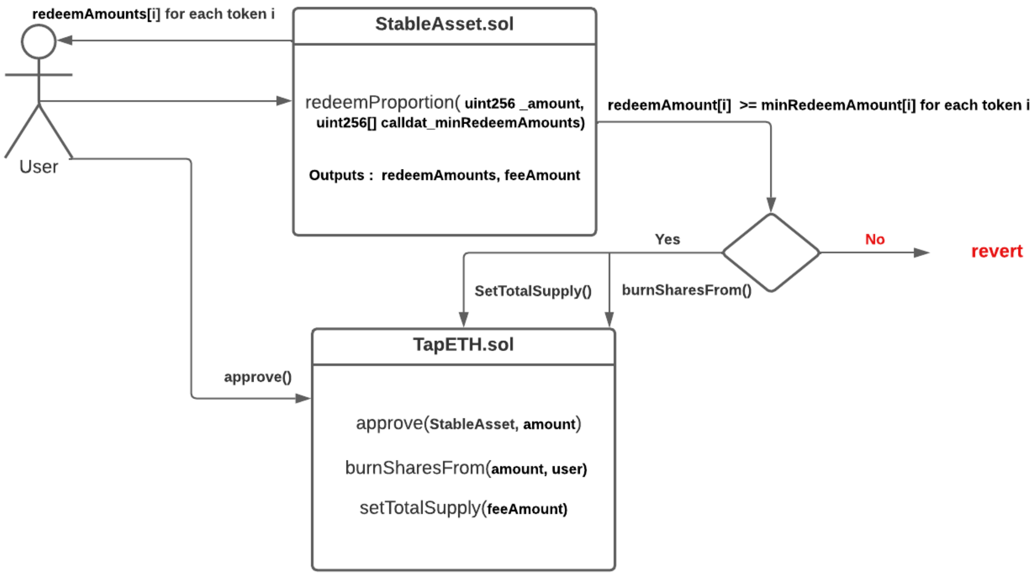

amounts[i] for each token i
StableAsset.sol
User
redeemMulti(uint256[]calldata_amounts, uint256_maxRedeemAmount)
Outputs : redeemAmount, feeAmount
redeemAmount <= maxRedeemAmount
Yes
No
revert
SetTotalSupply()
burnSharesFrom()
approve()
TapETH.sol
approve(StableAsset, maxRedeemAmount)
burnSharesFrom(redeemAmount, user)
setTotalSupply(feeAmount)

2.3.3 Redeem in balanced proportion

The user can redeem *amount* of tapETH in order to receive tokens in balanced proportion. The Logic of the function consists of:

1. Get A: $getA()$.
2. Calculate $D_{old} = getD(balances, A)$.
3. calculate $redeemAmount = amount - feeAmount = amount(1 - redeemFee)$.
4. For each token i:
   - Calculate $tokenAmounts[i] = redeemAmounts * balances[i]/D_{old}$.
   - Revert if $tokenAmounts[i] < minRedeemAmounts[i]$.
   - Send $tokenAmounts[i]$ of token i to the user.
5. Update $D_{old} = D_{old} - amount$.
6. Burn $amount$ of tapETH from the user.
7. Calculate $D_{new} = getD(realBalances, A)$.
8. Revert if $(D_{old} - D_{new}) \geq feeErrorMargin$.
9. Increase the total supply of tapETH by $feeAmount$.

The following UML diagram illustrates the different transactions generated by the function ”redeemProportion”:


redeemAmounts[i] for each token i
User
StableAsset.sol
redeemProportion(uint256 _amount, uint256[] calldat_minRedeemAmounts)
Outputs : redeemAmounts, feeAmount
redeemAmount[i] >= minRedeemAmount[i] for each token i
Yes
No
revert
SetTotalSupply()
burnSharesFrom()
approve()
TapETH.sol
approve(StableAsset, amount)
burnSharesFrom(amount, user)
setTotalSupply(feeAmount)