Lattice Fixed Reward Staking

Reward Distribution Algorithm

The Stardust Collective Team

May 2023

Replicates the behavior seen on SushiSwap's staking contract at https://github.com/sushiswap/StakingContract

Abstract

The Lattice Fixed Reward Staking contract is a time-based staking contract that allows a user to stake a certain amount of funds (staking token) in order to receive a reward token (reward token) given a certain period of time staked and the amount staked considering the global amount staked by all users. The rewards distribution algorithm uses several mathematical formulas and concepts in order to distribute rewards, the aim of this document is to explain in depth how the algorithm works, what formulas and concepts are used, and showcase some examples.

Algorithm

The base idea is to distribute rewards based on the staked ratio for a user ($usR = \frac{user\ staked\ amount}{global\ staked\ amount}$) and a given period of time staked, as the base example we can think of a user (user-a) that stakes STO 1,000, the base rewards for the program are RTO 30,000, the program timeline would be 30 days. Given the previous details, we can assume that a total of RTO 1,000 will be distributed per day for 100% of the global liquidity staked on that period (1 day).

Example Cases

- If user-a is the only one that stakes in the whole program timeline he will receive RTO 30,000, as he owned 100% participation in the pool for the whole program.
- Let's say user-a stakes STO 1,000 at the program start, then user-b stakes STO 3,000 on day 15. This means that for the first 15 days, user-a had a 100% allocation on the staking pool, so he will receive for those 15 first days a total of RTO 15,000 (RTO 1,000 per day), and for the next 15 days (16 day to 30) he shared participation with user-b of 25% - 75% respectively over the whole pool. Given the previous information for the 2nd period (days 16 - 30), user-a with 25% participation ($usR = \frac{1000}{4000}$) will receive a total of RTO 3.750 while user-b with 75% participation ($usR = \frac{3000}{4000}$) for that period will receive a total of RTO 11,250 (75% $\cdot$ RTO 15,000). Total rewards for both users would be, user-a $\Longrightarrow$ RTO 18,750, user-b $\Longrightarrow$ RTO 11,250
- Let's build on top of the previous example with an additional action, user-a stakes again more on day 20, he stakes STO 2,000 increasing his final balance to STO 3,000 (the same as user-b). The rewards now are calculated as follows for both users:

$$Rewards(user_a) = \frac{STO\ 1000}{STO\ 1000} \cdot \frac{15\ days}{30\ days} \cdot RTO\ 30000 + \frac{STO\ 1000}{STO\ 4000} \cdot \frac{5\ days}{30\ days} \cdot RTO\ 30000 + \frac{STO\ 3000}{STO\ 6000} \cdot \frac{10\ days}{30\ days} \cdot RTO\ 30000$$

$$Rewards(user_a) = 15000 + 1250 + 5000 = RTO\ 21250$$

$$Rewards(user_b) = \frac{STO\ 0}{STO\ 1000} \cdot \frac{15\ days}{30\ days} \cdot RTO\ 30000 + \frac{STO\ 3000}{STO\ 4000} \cdot \frac{5\ days}{30\ days} \cdot RTO\ 30000 + \frac{STO\ 3000}{STO\ 6000} \cdot \frac{10\ days}{30\ days} \cdot RTO\ 30000$$

$$Rewards(user_b) = 0 + 3750 + 5000 = RTO\ 8750$$

Given platform constraints (evm), we are unable to recalculate rewards accrued for each user every time the total staked liquidity changes (stake or withdrawal actions) this is because of gas fees, especially on the ethereum network. Taking into account the constraints mentioned we modified the distribution formula in order to get the same result, but being able to account for staked liquidity changes every time a user staked or withdrew without wielding exhaustive gas usage.

The algorithm now defines a variable called programRewardPerLiquidity (*PRpL*), which accumulates a reward vs global-liquidity rate for a given reward period. Reward periods are defined each time a user stakes (fn accrueRewardsPeriod()). A new rewards period is defined only when there is liquidity staked beforehand, thus generating / accruing rewards for previous participants / liquidity. We then calculate rewards for a given user in the following way.

Let's say user-a stakes RTO 1,000 at program start, programRewardPerLiquidity stays at 0 RTO/STO for the given unexisting period as user-a is the first user staking (no liquidity was staked before him), his account data, records the amount staked and the last

programRewardPerLiquidity value reported in the program when he staked, then a 2nd user (or any other user interaction that updates the global staked liquidity) stakes RTO 1,000 at day 5, a reward period is calculated before executing the staking process, the ratio added to the programRewardPerLiquidity variable is defined as follows.

$\Delta PRpL_1 = \frac{R_1 \cdot A}{L_1}$ where R_1 represents the rewards to be accrued to the immediate previous moment, L_1 represents the immediate previous liquidity staked (before performing changes in the call), and A represents a magnitude constant that enables us to increase our final number magnitude if the preliminary rate is near 0 or needs more precision.

$R_1 = TR \cdot \frac{\Delta RP}{TPL}$ where TR represents the total rewards available for the project at this moment in time, for our case RTO 30,000; where TPL represents the total program length in seconds, for our case 30 days represented in seconds; and where ΔRP is the delta between the last time a reward period was accrued and the current time (block.timestamp), for our case 5 days as the 1st participant (user-a) participated just at the beginning of the program.

Given the above formulas and replacing the values, we get.

$R_1 = RTO\ 30000 \cdot \frac{5}{30} = RTO\ 5000$ and $\Delta PRpL_1 = \frac{RTO\ 5000 \cdot A}{STO\ 1000} = 5\text{A RTO/STO}$

And internally RTO 5,000 tokens are deducted from the TR amount, leaving a balance of RTO 25,000 available for the next rewards period.

If user-a, let's suppose he withdrew on day 10, he would recalculate the programRewardPerLiquidity delta for the given previous period, and the values obtained would be.

$R_2 = RTO\ 30000 \cdot \frac{(day\ 10 - day\ 5) = 5}{30} = RTO\ 5000$ and $\Delta PRpL_2 = \frac{RTO\ 5000 \cdot A}{STO\ 2000} = 2.5\text{A RTO/STO}$

leaving a final programRewardPerLiquidity ($PRpL$) of $7.5A\ RTO/STO = 5A\ RTO/STO + 2.5A\ RTO/STO$.

Now for the reward calculation for user-a, we multiply the user programRewardPerLiquidity delta (current global programRewardPerLiquidity - user-a last recorded programRewardPerLiquidity $= \Delta uPRpL_a$) by the staked liquidity of the user at the time of withdrawal, the math series operation, gives as a result, the total rewards for the user for both staking periods where the user had 100% and 50% participation respectively, the formulas are shown bellow

$$Rewards(user_a) = \frac{\Delta uPRpL_a \cdot uStk_a}{A}$$

$$\Delta uPRpL_a = PRpL - uPRpL_a$$

For our case and filling in the variables

$$\Delta uPRpL_a = 7.5\ A\ RTO/STO - 0 = 7.5\ A\ RTO/STO$$

$$Rewards(user_a) = \frac{7.5\ A\ RTO/STO \cdot STO\ 1000}{A} = \frac{7.5\ RTO \cdot STO\ 1000}{STO} = RTO\ 7500$$

Finally and before finishing the operation, if the user-a decides to withdraw their rewards (RTO 7,500), their last recorded programRewardPerLiquidity would be set to the last calculated one to offset their rewards (mathematically claim them), in the following opportunity (re-stake) the delta would be recalculated using this last value, thus ensuring the user would not receive more rewards than expected.

Final Words

The described algorithm enables the users to stake and obtain rewards based on the immediate proportion staked compared to other participants at any given period of time without consuming extensive gas fees, the algorithm above applies not only to the first user but to any user who stakes in the program.

As there is a really high chance of the 1st user not being able to stake exactly at the program start, the concept of lost rewards was introduced, this concept enables rewards to be marked as lost for a specific period of time in which no liquidity is staked but the program timeline is valid, the difference, in this case, is that the accrueRewardsPeriod() function removes those unaccrueable rewards from the program and marks them as lost.

On the other hand, if the user had the intention to withdraw partially or re-stake again on a staked amount and was not interested in claiming their rewards, the user can keep their rewards in the pool until he decides to withdraw them, to achieve this, the user last recorded programRewardPerLiquidity is offset but the new amount that would be staked considering his current rewards.

Attachments

- Example Calcs - https://bit.ly/3MTKQuW