Scalable Reward Distribution on the Ethereum Blockchain

Bogdan Batog   Lucian Boca   Nick Johnson

***Abstract*—A system capable of distributing periodic rewards towards a pool of participants in proportion to their stake is a key component for a wide range of practical applications, from decentralized staking pools to dividend allocation in tokenized organizations. We present an efficient way of initiating the transactions required by such a system and discuss its implementation as an Ethereum smart contract that complements a token contract by adding a staking mechanism. Our implementation decouples the reward distribution and withdrawal flows, makes use of a memory-optimal algorithm to coordinate the reward allocation and achieves $O(1)$ time complexity for all the core functions of the system. This approach scales to any number of registered stakeholders and allows for reward distribution events that have higher thoughput, higher granularity and generate an even load on the underlying network.**

I. Introduction

A promising use case for a blockchain-based smart contracts platform such as Ethereum is the creation of custom tokens that track fractional ownership or participation in an underlying asset or organization and periodically distribute rewards to the token holders, in direct proportion to the amount of tokens they choose to stake.

We propose a scalable solution for implementing the reward distribution logic through a staking contract on the Ethereum platform. Both the stake and the rewards can be quantified using any transferrable Ethereum-based tokens [1] or ethers.

II. Rewards in batched transactions

We start by outlining the core functions that a reward distribution system needs to efficiently perform:

1) `deposit(stake)` and `withdraw()` for participants to manage their stake allocation in the system.
2) `distribute(reward)` for the organization to pay back a reward that the system will distribute to the participants.

In a naive implementation, the staking contract would push fractional payments to all the participants each time a reward gets distributed. However, in such an implementation the `distribute` function would take $O(N)$ time to compute, where N is the number of participants.

Namely, there are two technical challenges that make this process inefficient:

- In case of Ethereum, due to the way the EVM works [2], it's not possible to enumerate all participant addresses, when stored as a map. Thus, keeping track of an iterable registry of participants and their corresponding stake usually requires additional data structures such as Linked or Double-Linked lists. [3]
- Iterating over all participants is prohibitively expensive [4] for a registry contract that manages tens of thousands of entries. In such a case, the relatively small fraction of the reward that needs to be distributed for each share doesn't justify the gas costs incurred by executing the entire series of operations on-chain.

This makes the naive approach unfeasible for handling a large number of participants or frequent reward distribution events.

III. Pull based reward distribution

Our proposal starts by modeling the reward distribution system as a buffer that collects and temporarily stores the rewards until they are explicitly withdrawn by each participant.

Instead of a *push-based* flow that splits and allocates the rewards immediately to all the participants, our solution stores the reward amounts and lets the participants make arbitrary withdrawals in a *pull-based* manner. An algorithmic optimization based of partial sum lookups allows for $O(1)$ implementations of all the core operations of the system. [5]

This strategy allows us to create staking contracts that can handle a few orders of magnitude more participants, support arbitrarily small rewards with granular (e.g. hourly) distribution schedules and operate with a substantially smaller on-chain computational overhead.

A. Reward model

We start by noting that absolute instants of the deposits, withdrawals and distribution events are not relevant, as the final outcome is only determined by the relative order of these events.

Without loss of generality we consider only one deposit action per address and only full withdrawals. Additional deposits to an existing address can be modeled as two separate addresses while partial withdrawals can be modeled by two simpler operations: a full withdrawal followed by a new deposit.

Let's consider the chronological order of all the `deposit`, `withdraw` and `distribute` events. At a given instant t on this timeline, let T_t be the sum of all active stake deposits. On a `distribute` event for $reward_t$, a participant j with an associated $stake_j$ will get a reward of:

$$reward_{j,t} = stake_j * \frac{reward_t}{T_t} \tag{1}$$

A simple implementation would iterate over all active stake deposits and increase their associated reward. But such a loop requires more gas per contract call as more deposits are created, making it a costly approach. A more efficient implementation is possible, in $O(1)$ time. [6]

B. Factor out reward computation

The total reward earned by a participant j for its stake deposit $stake_j$ is the sum of proportional rewards it extracted from all distribution events that occurred while the deposit was active:

$$total_{reward_j} = \sum_t reward_{j,t} = stake_j * \sum_t \frac{reward_t}{T_t} \tag{2}$$

where t iterates over all reward distribution events that occurred while $stake_j$ was active. Let's note this sum, from the beginning of timeline until instant t:

$$S_t = \sum_{k=0}^{t} \frac{reward_k}{T_k} \tag{3}$$

Assuming $stake_j$ is deposited at moment t_1 and then withdrawn at moment $t_2 > t_1$, we can use the array S_t to compute the total reward for participant j:

$$total_{reward_j} = stake_j * \sum_{t=t_1+1}^{t_2} \frac{reward_t}{T_t} \tag{4}$$

or,

$$total_{reward_j} = stake_j * (S_{t_2} - S_{t_1}) \tag{5}$$

This allows us to compute the reward for each `withdraw` event in constant time, at the expense of storing the entire S_t array in the contract memory.

C. Optimizing memory usage

The memory usage can be further optimized by noting that S_t is monotonic and we can simply track the current (latest) value of S and snapshot this value only when we expect it to be required for a later computation.

We will use a map $S_0[j]$ to save the value of S at the time the participant j makes a `deposit`. When j will later `withdraw` the stake, its total reward can be computed by using the latest value of S (at the time of the withdrawal) and the snapshot $S_0[j]$:

$$total_{reward_j} = stake_j * (S - S_0[j]) \tag{6}$$

This strategy makes it possible to achieve both time optimality and memory optimality, as for N participants it takes $O(N)$ memory to keep track of both the S_0 value map and the $stake$ registry.

As memory usage no longer depends on the number of `distribute` events, the algorithm is now suitable for very fine grained distribution: daily, hourly or even at every mined block.

D. Constant time algorithm

The algorithm will expose three methods:

- `deposit` to add a new participant stake.
- `distribute` to fan out reward to all participants.
- `withdraw` to return the participant's entire stake deposit plus the accumulated reward.

Algorithm 1: Constant Time Reward Distribution

```
function Initialization();
begin
    $T$ = 0;
    $S$ = 0;
    $stake$ = {};
    $S_0$ = {};
end

function Deposit ($address, amount$);
Input  : set $amount$ as the stake of $address$
begin
    $stake[address]$ = $amount$;
    $S_0[address]$ = $S$;
    $T$ = $T$ + $amount$;
end

function Distribute ($r$);
Input  : Reward $r$ to be distributed proportionally to
         active stakes
begin
    if $T$ != 0 then
        $S$ = $S$ + $r$ / $T$;
    else
        revert();
    end
end

function Withdraw ($address$);
Input  : $address$ to withdraw from
Output: $amount$ withdrawn
begin
    $deposited$ = $stake[address]$;
    $reward$ = $deposited$ * ($S$ - $S_0[address]$);
    $T$ = $T$ - $deposited$;
    $stake[address]$ = 0;
    return $deposited + reward$
end
```

IV. Notes and future work

The stake and reward may be units of the same token (e.g. PoS pools) or they may be different tokens. In practice, most organizations will stake ERC20 tokens and distribute either ether rewards (dividends) or other tokens (utility or loyalty points). When different tokens are used, the withdrawal action will execute two different transfers, instead of returning $deposited + reward$.

If the same token is used for quantifying both the stake and the reward, the algorithm will not compound the reward. An active user may effectively compound the reward by executing a $withdraw$ followed by a new $deposit$. We will examine the possibility to automatically compound for all participants.

The algorithm is loop free so it can also be implemented in Turing incomplete smart contract languages.

References

[1] F. Vogelsteller and V. Buterin, “ERC: Token standard 20,” https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, 2015, [Online; accessed 1-Feb-2018].

[2] “Solidity Reference Documentation,” http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html#storage-memory-and-the-stack, 2017, [Online; accessed 1-Feb-2017].

[3] D. Morris, “Ethereum Stack Exchange answer,” https://ethereum.stackexchange.com/a/15341/29330, 2017, [Online; accessed 25-Jan-2018].

[4] G. Wood, “Ethereum yellow paper,” https://ethereum.github.io/yellowpaper/paper.pdf, p. 24, 2017, Appendix G. Fee Schedule.

[5] B. Batog, “Constant Time Fee Redistribution Smart Contract,” https://github.com/bogdanbatog/token-savings/blob/master/paper/token_fee_redistribution.pdf, 2017, [Online; accessed 1-Feb-2018].

[6] N. Johnson, “Dividend-bearing tokens on Ethereum,” https://medium.com/@weka/dividend-bearing-tokens-on-ethereum-42d01c710657, 2017, [Online; accessed 27-Feb-2018].