LiquidStake: a Scalable Reward Distribution System on the Ethereum Blockchain

Bogdan Batog
bogdanbatog@gmail.com

Lucian Boca
lucianboca@gmail.com

February 2, 2018
Draft 0.4

Abstract

A system capable of distributing periodic rewards towards a pool of participants in proportion to their stake is a key component for practical applications ranging from decentralized staking pools to dividend allocation in token-based digital cooperatives. However, the core financial processes involved in such a system face both bureaucratic challenges if the organization is established within a traditional regulatory framework and scalability challenges if the organizational logic is deployed on top of a blockchain platform. We present an efficient way of programming the operations required by such a system through an Ethereum smart contract that allows participants to register their stake and receive proportional rewards with high granularity in $O(1)$ time using a memory-optimal algorithm.

Keywords: Ethereum, smart contracts, staking pools, dividend distribution, DAOs, scalability

1 Introduction

A promising use case for a blockchain-based smart contracts platform such as Ethereum is the model of a digital cooperative in which people can simultaneously contribute to and derive economic reward out of their contribution.

Similar to investment funds or mining pools, the participants pool together resources and capital in exchange for a quantifiable stake in the organization. The organization will then periodically distribute its profits to all the participants, in direct proportion to their allocated stake.

1.1 Blockchain advantages

This participatory model can vastly benefit from a scalable, cost-effective way of deploying its core financial processes through a blockchain-based smart contract infrastructure, alleviating the bureaucratic overhead associated with intermediating the required economic exchanges in a traditional regulatory framework.

More specifically, the system needs to allow its participants to perform the following operations:

1. Deposit and withdraw their stake.
2. Receive rewards from the distribution of profits – equivalent to the concept of dividends.

We propose a scalable solution for implementing these operations through a staking contract on the Ethereum platform. The contract makes use of a time and memory-optimal algorithm and exposes a simple interface for accessing the reward distribution logic. Both the stake and the rewards can be quantified using either ethers or any transferrable Ethereum-based tokens.

Furthermore, we attempt to define a standard, adaptable interface for modeling generic staking contracts that organizations can use to distribute profits to their stakeholders in an ERC-compliant way [1] and provide an efficient implementation of the proposed standard. This approach will promote interoperability and consistency in the application layer, compatibility with existing standards (e.g. ERC20 [3] for quantifying the stake) and cultivate a network effect among all the participants of this economic model.

2 The problem

We will 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 naïve 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:

- It is not possible, due to the way the EVM works, to enumerate storage from within a contract. 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 [2].
- Iterating over all participants is prohibitively expensive 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 naïve approach unfeasible for handling a large number of participants or frequent reward distribution events.

3 Solution

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. Coupled with an algorithmic optimization that makes use of partial sum lookups, this allows for $O(1)$ implementations of all the core operations of the system.

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.

3.1 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 us consider the timeline induced by the chronological chain of all the `deposit`, `withdraw` and `distribute` events. For a given instant t on this timeline, let T_t be the sum of all active stake deposits. On a distribution 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.

3.2 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 the timeline until instant t:

$$S_t = \sum_t \frac{reward_t}{T_t} \tag{3}$$

We can make use of the array S_t to compute the total reward for a participant j through a partial sum lookup. Assuming that $stake_j$ is deposited at moment t_1 and then withdrawn at moment $t_2 > t_1$:

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

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.

3.3 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 $S_0[j]$:

$$total_{reward_j} = stake_j * (S - S_0[j]) \tag{5}$$

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.

3.4 Constant time algorithm

The algorithm will expose three methods:

- `deposit` is called when a new participant stake is added.
- `distribute` is called for every reward distribution event.
- `withdraw` will 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
```

4 Notes and future work

1. The stake and reward may be units of the same token (i.e. PoS pools) or they may be different kinds of tokens. In practice, most cooperatives will stake ERC20 tokens and distribute ether rewards. Secondary use case - stake tokens and distribute other tokens as rewards. When different tokens are used, the withdrawal action will execute two different transfers, instead of returning $deposited + reward$.

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

References

[1] Ethereum Request for Comments (ERCs). `https://github.com/ethereum/EIPs/issues/16`, 2015. [Online; accessed 1-Feb-2018].

[2] Darryl Morris. Ethereum Stack Exchange answer. `https://ethereum.stackexchange.com/a/15341/29330`, 2017. [Online; accessed 25-Jan-2018].

[3] Fabian Vogelsteller and Vitalik Buterin. ERC: Token standard 20. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`, 2015. [Online; accessed 1-Feb-2018].