Digital Shares: Organization Profit Distribution With Unlimited Shareholders

Anatoly Ressin
BlockVis
a@blockvis.com

Simon Yakunin
BlockVis
s@blockvis.com

August 28, 2017

Abstract

The authors introduce Digital Shares, a ERC-20 compatible Ethereum contract which can distribute organization profit between unlimited number of shareholders. The main problem with such contracts is the limited amount of gas available for use in one transaction. When we have limited amount of gas and (potentially) unlimited number of shareholders then we have a problem. To distribute organization profit, contract needs to iterate all shareholders and calculate his/her profit amount according to number of shares. We propose a solution to overcome this issue.

1 Introduction

Digital Shares contract is a solution to overcome issues which are arising when organization needs to distribute profit between shareholders on Ethereum [1] blockchain.

ERC-20 [2] token stores share distribution in a mapping and allowes shareholders to transfer shares between each other. In Ethereum mapping data type is not iterable. So to overcome this, contract needs to store a list of shareholders addresses. When profit distribution occurs a contract must iterate over shareholder list and distribute profit according to number of shares which each particular shareholder have. When there are small number of shareholder this scheme works, but when number of shareholders grows, transaction, which distributes profit, runs out of gas. There are some possible solutions to overcome this. Contract can distribute profit in batches by 100 or 200 shareholders and store an index of last distributed shareholder, so when distribution method is called again the distribution resumes iteration from the last saved index. This solution is very limited. When there are, for example, 10000 shareholders we must run 10000 / 200 = 500 transactions. A contract must also stop all share transfers until all shareholders did not receive their profit. 500 transactions on

Ethereum blockchain will run for about 17s * 500tx = 2 hours. And this number will grow as long as the number of shareholders grows.

The proposed solution resolves this issues. A profit distribution transaction costs a constant O(1) amount of gas on unlimited number of shareholders.

2 Related Works

2.1 Polybius Token

Another approach of organisation profit distribution offers Polybius Token:

The dividends distribution process consists of two stages: gettng the accurate data about the PLBT distribution among users (dividends report) and spreading the company profit among the tokenholders. [3]

This approch is possible, but not recommended on Ethereum network. The recommended approach is to favor pull payments over push payments.[4]

2.2 Dividend-bearing tokens on Ethereum[5]

In this article the author proposes a solution with modifier updateAccount(address account) which must be executed on every share transfer. This approach is better, but from functional point of view we have a method misbehavior. When the method is called it should not do anything besides we asked it to do. When shareholder transfers his shares to another address at this moment two updateAccount calls are performed. The first one for source account and the second for target account. Besides this costs gas on every transfer call it may raise some exceptions in updateAccount modifier and shareholder would be unable to send his shares for some reason.

3 Digital Shares

The problem with profit distribution that the contract must be able to distribute profit according to shareholders share distribution. ERC-20 contract have mapping(address => uint256) balances storage to store share distribution. The contract cannot iterate on all shareholders when distributing profit, because transaction will run out of gas when there are many shareholders. The point is when profit distribution transaction performs, the contract must take a snapshot of shareholders share distribution and when shareholder withdraws profit a contract can calculate it easily. This can be achieved by having array of struct where each array item represents share distribution on a moment of profit distribution and amount of distributed profit:

```
struct Snapshot {
        uint256 amountInWei;
        mapping(address => uint256) balances;
```

```
}
Snapshot[] shapshots;
```

This way when shareholder withdraws his profit, the contract is able to get back in past and calculate shareholders profit for every profit distribution by using simple formula:

```
uint profit = 0;
for(uint i = 0; i < snapshots.length; i++) {
        Snapshot snapshot = snapshots[i];
        profit += snapshot.balances[msg.sender] * snapshot.amountInWei / totalSu
}
...
```

The point which makes this scheme impossible to work is that we cannot copy mappings in Ethereum. The only way we could copy balances is to have an array of known shareholders addresses and iterate over it to copy data into new snapshot. This is not possible either because when there will be several thousands of shareholders our copying procedure will run out of gas. But how can we take a snapshot of share distribution without iterating all shareholders balances? The solution is that each snapshot should store not the share balance itself, but the balance changes. So the Snapshot struct will look like the following:

```
struct Snapshot {
        uint256 amountInWei;
        mapping(address => int256) balances;
}
Snapshot[] shapshots;
```

Using this approach the first snapshot will store absolute share distribution and the following snapshots will store relative share distribution. This way when profit distribution occurs the contract must create only new empty snapshot and store how much profit the contract must distribute. When shareholder calls withdraw() method the contract calculates shareholder profit using almost the same formula:

```
int256 shares = 0;
uint profit = 0;
for(uint i = 0; i < snapshots.length - 1; i++) {
        Snapshot snapshot = snapshots[i];
        shares += snapshot.balances[msg.sender];
        profit += uint256(shares) * snapshot.amountInWei / totalSupply;
}
...
```

snapshots.length−1 is required because last snapshot contains current share balance changes and is not included in withdrawal at the moment when withdraw() is called. Althou this algorithm works, it can be improved further.

4 Improvements

4.1 Duplication

When shareholder calls withdraw() method twice the profit amount will be calculated from the beginning and already paid profit will be calculated again. So the contract needs to store latest paid snapshot and next time when withdraw() is called by shareholder, iteration is started from latest paid snapshot:

```
mapping(address => uint) payed;
...
function withdraw() returns (bool) {
        int256 shares = 0;
        uint profit = 0;
        for(uint i = 0; i < snapshots.length - 1; i++) {
                Snapshot snapshot = snapshots[i];
                shares += snapshot.balances[msg.sender];
                if (i >= payed[msg.sender]) {
                        profit += uint256(shares) * snapshot.amountInWei / totalS
                }
        }
        payed[msg.sender] = snapshots.length - 1;
        ...
}
```

4.2 Gas Usage

Let's look at the cycle. We can see that division by totalSupply is done on every iteration and this costs gas. We can improve this algorithm by moving division out of cycle:

```
function withdraw() returns (bool) {
        int256 shares = 0;
        uint profit = 0;
        uint numerator = 0;
        for(uint i = 0; i < snapshots.length - 1; i++) {
                Snapshot snapshot = snapshots[i];
                shares += snapshot.balances[msg.sender];
                if (i >= payed[msg.sender]) {
                        numerator += uint256(shares) * snapshot.amountInWei;
                }
        }
        profit = numerator / totalSupply;
        payed[msg.sender] = snapshots.length - 1;
        ...
}
```

4.3 Rounding Errors

Sometimes the profit between shareholders cannot equally divided. For example, organization have only 30 shares and 3 shareholders each having 10 shares. When we want to distribute 10 wei, we cannot do it, because each shareholder must receive 3.3(3) wei of profit. To resolve this problem contract stores undivided wei count, so it can return undistributed shareholders profit when next time shareholder will come for his profit.

```
mapping(address => uint) unpayedWei;
...
function withdraw() returns (bool) {
        int256 shares = 0;
        uint profit = 0;
        uint numerator = unpayedWei[msg.sender];
        for(uint i = 0; i < snapshots.length - 1; i++) {
                Snapshot snapshot = snapshots[i];
                shares += snapshot.balances[msg.sender];
                if (i >= payed[msg.sender]) {
                        numerator += uint256(shares) * snapshot.amountInWei;
                }
        }
        profit = numerator / totalSupply;
        unpayedWei[msg.sender] = numerator % totalSupply;
        payed[msg.sender] = snapshots.length - 1;
        ...
}
```

4.4 Redundant Cycles

It can be seen that each cycle within withdraw() method starts from 0. When number of snapshots gets high this can consume a lot of gas.

After the cycle ends, we have current share balance of a particular shareholder stored in shares variable. We can store this balance, so when the same shareholder calls withdraw() method again, the contract will take this balance as an absolute value and count the differences from the last point:

```
mapping(address => uint) unpayedWei;
...
function withdraw() returns (bool) {
        int256 shares = 0;
        uint profit = 0;
        uint numerator = unpayedWei[msg.sender];
        uint lastSnapshotIndex = snapshots.length - 1;
        for(uint i = payed[msg.sender]; i < lastSnapshotIndex; i++) {
                Snapshot snapshot = snapshots[i];
                shares += snapshot.balances[msg.sender];
```

```
                numerator += uint256(shares) * snapshot.amountInWei;
            }
            snapshots[lastSnapshotIndex].shares[msg.sender] += shares;
            profit = numerator / totalSupply;
            unpayedWei[msg.sender] = numerator % totalSupply;
            payed[msg.sender] = lastSnapshotIndex;
            ...
}
```

5 Metrics

The table contains statistic data about gas consumption of two most important methods. Both methods are invariant from the shareholders count.

Method/Number of distribute() calls	1	10	100	1000
distribute()	74320	74320	74320	74320
withdraw()	69511	80482	190192	1287292

As we can see distribute () method uses constant amount of gas.

withdraw() method depends only on number of sequential distribute () method calls. But once the withdraw() is called the gas consumption of the next withdraw)) method call starts from the beginning. This way, the shareholder may not withdraw his profit for a long period of time and withdraw all of his profit with one large transaction. The gas consumtion of withdraw() method is so low, that if organization profit distribtution is done every day, then shareholder may not withdraw his profit for 1000 days and still be able to withdraw his profit with one transaction.

6 Conclusion

The proposed solution is able to handle any number of shareholders within organization. It consumes constant amount of gas on organization profit distribution and gas consumption of profit withdrawal is only dependant on profit distribution count.

References

[1] *Ethereum White Paper*
`https://github.com/ethereum/wiki/wiki/White-Paper`

[2] *ERC-20 Token Standard*
`https://theethereum.wiki/w/index.php/ERC20_Token_Standard`

[3] *POLYBIUS TOKEN WHITEPAPER*
`https://polybius.io/media/token_whitepaper.pdf`

[4] *ConsenSys: Ethereum Contract Security Techniques and Tips*
https://github.com/ConsenSys/smart-contract-best-practices#favor-pull-over-push-payments

[5] Nick Johnson, *Dividend-bearing tokens on Ethereum*
https://medium.com/@weka/dividend-bearing-tokens-on-ethereum-42d01c710657