An Efficient Approach to Calculating the Cumulative Variable Revenue of Every Stakeholder in the Context of a Distributed Financial Application

Alexis Schlomer

Abstract

This paper presents a new efficient way to compute the cumulative variable revenue of every stakeholder inside a dynamic participatory pool on a blockchain. Using state sharing and a lazy evaluation technique, we no longer need to block the pool during revenue distribution nor consider a constant sized pool.

1 Introduction

Blockchain technology has recently seen the rise of several distributed financial applications on networks such as Ethereum. The examples go from decentralized lending platforms, pool-based exchanges and yet more recently vaults executing complex investment strategies to maximize one's yield.
Most of these applications must rely on efficient code that keeps track of all the revenue aggregated by every participant at any time. In this paper, we will present a common problem one faces when dealing with distributed computing as well as an original and efficient way to overcome this.

2 The Problem Definition

We will first give an intuitive overview in plain English with an illustrative example before entering the realm of mathematics and giving it a formal definition.

2.1 In Plain English

Let us consider a very simple financial application that receives an arbitrary stream of revenue: for the sake of simplicity, we shall consider the product of sales of a wine producer. The revenues are automatically distributed to the stakeholders, according to their participation in the company. The participation at a given moment is determined by the ratio of personal money locked in the company over the total locked by all stakeholders combined.

2.1.1 Example

In this example, we consider that any actor can add or remove stake at any time and that the company can generate a variable revenue at any time as well.

Day 1: The company sold 100 bottles of wine, generating around \$1'000 in profit.

- Josh has \$10'000 locked in the company.
- Robert has \$90'000 locked in the company.

$\Rightarrow$ Josh gets $\frac{10'000}{10'000+90'000} \times 1'000 = \$100$.
Robert gets $\frac{90'000}{10'000+90'000} \times 1'000 = \$900$.

Day 2: Bob joins the company with \$50'000 before it generates a new \$2'000 in profit.

- Josh and Robert keep their previous stakes.
- Bob now has \$50'000 locked in the company

$\Rightarrow$ Josh has an accumulated profit of $100 + \frac{10'000}{150'000} \times 2'000 = \$233.33$.
Robert has an accumulated profit of $900 + \frac{90'000}{150'000} \times 2'000 = \$2'100$.
Bob has an accumulated profit of $\frac{50'000}{150'000} \times 2'000 = \$666.66$.

Using this simple example, we intuitively realize that unless updating the stakes and distributing the revenue at every interaction (or transaction) we cannot find the cumulative revenue of every stakeholder easily *in a distributed network*.

- Iterating over all users and updating their stake at every transaction reaches the gas limit quickly as the number of stakeholders increases. This is vulnerable to denial of service attacks.
- The revenue increment is discrete, variable and unpredictable. We cannot benefit from classic techniques that use a common aggregated index to indicate the relative increase of revenue (as is used in lending protocols).
- Any transaction can happen at any time. The staking pool is never blocked for revenue distribution.
- The total size of the pool is not constant. In our example it means that shares can both be burned or minted (which gives us more flexibility).

Let's formulate this problem mathematically.

2.2 Mathematical Description

As noticed in our previous example, we can assimilate all user interactions by serializable transactions. This is illustrated with the index i in the following notations.
Let us consider the following definitions:

- r_i The revenue generated at transaction i.
- s_i The up-to-date stake ratio held by a user at transaction i: $s_i \in [0; 1]$.

We want to calculate for any $i \in [0; \infty[$

$$\sum_{n=0}^{i} r_n s_n$$

The accumulated revenue of a given user at transaction i.

2.2.1 The Failures of a Naive Approach

The first obvious way to calculate this amount is by recording all values r_i and s_i for all n such that $0 \leq n \leq i$. This approach is terribly inefficient as it has to be replicated across all users and takes $\mathcal{O}(TotalUsers \times TotalTransactions)$ space on the blockchain. Furthermore, at every transaction we have a computation time of $\mathcal{O}(i)$.
Updating the whole database at every transaction to reflect the current cumulative revenues of all users (as would be done on a centralized system) sacrifices computation time for more acceptable spatial requirements. Yet this would still bring a computation time of $\mathcal{O}(TotalUsers)$.
On distributed systems we somehow need to bring this time to $\mathcal{O}(1)$.

3 The $\mathcal{O}(1)$ Computation Time Approach

3.1 A Quick Overview

An efficient solution we designed relies on a lazy evaluation technique combined with state sharing.

- *State Sharing:* Consists of storing a global shared state that gets updated at every transaction as well as multiple user states. A user state gets updated every time the related user issues a transaction, alongside with the global shared state.
- *Lazy Evaluation:* Consists of evaluating the cumulative revenue of a user only when explicitly asked by the user. This can be in the case of a normal blockchain lookup or in the case of a transaction that requires this value.

With that in mind let us present our solution in detail.

3.2 The Solution in Detail

We will start by defining the following variables divided in two distinct categories as we mentioned above:

User State:

$ownStake$	The owned stake.
$lastTotalStake$	The total stake at last update.
$lastIncrementPerRevenue$	The increment per revenue at last update.
$ownAccumulatedTotal$	The owned accumulated revenue.
$lastIndex$	The global index at last update.

Global State:

$totalStake$	The total stake.
$incrementPerRevenue$	The increment per revenue.
$index$	The global index.

Now we consider the following updates at every transaction:

1. **Change of Personal Stake:**

   Let S_c be the variation of personal stake (supposing legal values), we perform the following updates:

   Phase 1: Update the user's accumulated total revenue:

$$freshOwn := \begin{cases} \frac{(index - lastIndex) \cdot ownStake}{lastIncrementPerRevenue \cdot lastTotalStake}, & \text{if } ownStake \neq 0 \\ 0, & \text{otherwise} \end{cases}$$

$$ownAccumulatedTotal := ownAccumulatedTotal + freshOwn$$

   Phase 2: Update the global state accordingly:

$$oldTotalStake := totalStake$$

$$newTotalStake := totalStake + S_c$$

$$incrementPerRevenue := \begin{cases} INIT \quad \text{strictly positive}, & \text{if } newTotalStake = 0 \\ \frac{incrementPerRevenue \cdot oldTotalStake}{newTotalStake}, & \text{if } oldTotalStake \neq 0 \\ incrementPerRevenue, & \text{otherwise} \end{cases}$$

$$totalStake := newTotalStake$$

Phase 3: Refresh user state:

$$ownStake := ownStake + S_c$$

$$lastIndex := index$$

$$lastIncrementPerRevenue := incrementPerRevenue$$

$$lastTotalStake := totalStake$$

2. **Revenue Increase:**
   Let R_i be the net positive increase of revenue at transaction i:

$$index := index + incrementPerRevenue \cdot R_i$$

3.3 Proof

Assertion:

$$\sum_{n=0}^{i} r_n s_n = ownAccumulatedTotal + freshOwn$$

With $freshOwn$ obtained the same way as seen above.

By an inductive argument, we will only study the freshly acquired revenue since the last user transaction: $freshOwn$. Hence our problem reduces to:

$$\sum_{i=l+1}^{n} r_i s_i = freshOwn$$

Where l represents the last user transaction and n represents the last global transaction induced by another user.

$$\sum_{i=l+1}^{v} r_i a_i \; + \; I_l = I_v$$

Where I_v represents the index at transaction v where $l \leq v \leq n$.
And a_i represents the increment per revenue at transaction i.

Hence by simple algebra,

$$\sum_{i=l+1}^{v} r_i a_i = I_v \; - \; I_l$$

Assuming the following equality,

$$a_i = a_{i-1} \cdot \frac{S_{i-1}}{S_i}$$

Where S_i represents the total capital staked after transaction number i.

By recursively applying this identity we get,

$$a_i = a_{i-1} \cdot \frac{S_{i-1}}{S_i} = a_{i-2} \cdot \frac{S_{i-1}}{S_i} \cdot \frac{S_{i-2}}{S_{i-1}} = a_{i-2} \cdot \frac{S_{i-2}}{S_i} = ... = a_l \cdot \frac{S_l}{S_i}$$

By combining both results we obtain,

$$\sum_{i=l+1}^{v} r_i a_i = a_l S_l \cdot \sum_{i=l+1}^{v} \frac{r_i}{S_i} = I_v - I_l$$

Hence, by some algebra and multiplying both sides by S_{own},

$$\sum_{i=l+1}^{v} \frac{r_i}{S_i} = \frac{I_v - I_l}{a_l S_l}$$

$$\sum_{i=l+1}^{v} r_i \cdot \frac{S_{own}}{S_i} = \frac{(I_v - I_l) \cdot S_{own}}{a_l S_l}$$

Which brings us directly to our result,

$$\sum_{i=l+1}^{n} r_i s_i = \frac{(I_n - I_l) \cdot S_{own}}{a_l S_l} = freshOwn \quad \square$$

All of these variables are known either in the shared state or in the user state.

4 Conclusion

This paper settles some basics of what is possible in distributed computing on blockchains. We can go further by considering negative revenues or even a continuous revenue stream to get a more realistic model. Additionally, such computations may become useful in other situations apart from decentralized finance. It has the advantage of proposing a great scaling capability, even on centralized systems. Yet, some problems would still need to be solved, such as handling the arithmetic on computers to avoid catastrophic overflows or heavy losses in precision.