polygon zkEVM

Knowledge Layer

Architecture

System Performance and the "Order then Prove" Paradigm

v.1.0

June 4, 2024

1 Introduction

In this document, we will delve into the distinction between the sequencing process and the proving process, emphasizing the necessity of separating these two steps within the overall transaction flow. By decoupling sequencing from proving, we can enhance the efficiency of the system. The central figures in this topic of the zkEVM architecture are the **Verifier** (Smart Contract), the **Prover**, the **Aggregator** and the **Sequencer**. Recall that, when a the L1 transaction submitting the proof π together with the corresponding publics is correctly executed, we say that a new L2 state is **consolidated** (See Figure 1) because at this moment the L2 state is proven to evolve correctly from $S_i^{L2_x}$ to $S_{i+1}^{L2_x}$ according to the batch processed.

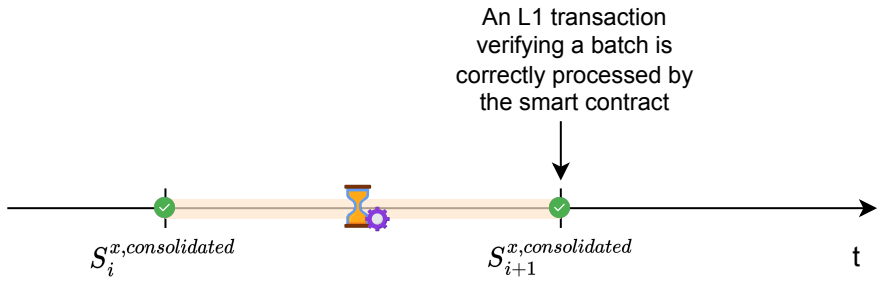

Figure 1: A consolidated batch is a batch that has been verified by the L1 smart contract.

2 zkEVM Key Performance Indicators

The main zkEVM key performance indicators (KPIs) are the **Delay** and the **Throughput**. The first one refers to the delay from the moment that a user sends an L2 transaction until the effects of the execution of the transaction can be "considered" part of the L2 state. For user experience (UX), this is the main KPI. The throughput measures the systems capacity of processing transactions. It can be computed in transactions per second, gas per second or batches per second.

Lets analyze and re-engineer our current system to improve these KPIs. There are 3 parameters that affect the KPIs, schematized in Figure 2:

1. `close_a_batch_time`: This is the time that it takes get enough transactions to create (close) a batch or a given timeout for this purpose.
2. `prove_a_batch_time`: This is the time that it takes to generate the proof for a single batch (obviously, the size of the batch can affect this time).
3. `block_time`: This is the minimum time that it takes to execute L1 transactions.

Let us explore how this parameters impact the delay and the throughput of the full system. To understand this better, we'll look at a simplified example of a processing pipeline, like an assembly line in a factory. Figure 3 shows this toy example.

From the previous Figure, we can identify two key performance indicators (KPIs) we care about: lead time (or delay) and production rate (or throughput). Lead time refers to how long it takes to produce the first car after starting the line. In this example, the lead time is 6 hours. Production rate tells us how many cars can be produced per unit of time. Here, the production rate is 1 car every 3 hours, which is equivalent to 1/3 cars per hour. Naturally, the question arises: how can we improve both these metrics? To improve both delay and throughput, two methods are employed: **horizontal scaling** and

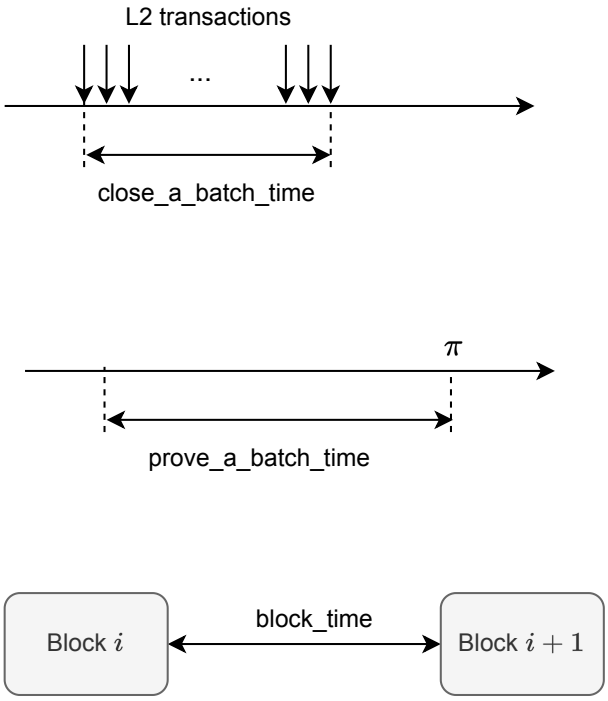

Figure 2: Parameters that affect the KPIs.

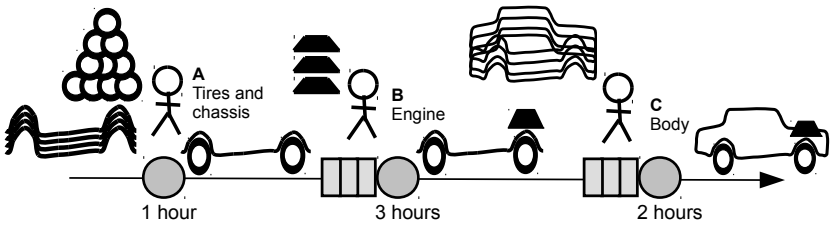

Figure 3: Toy example of a pipeline process: a factory that ensembles cars.

vertical scaling. Recall that the objective is to increase the throughput and reduce the delay.

Horizontal scaling involves adding more operators (CPUs) of smaller power in parallel, boosting throughput. However, the delay remains the same in this scenario. On the other hand, **vertical scaling** entails adding more powerful operators (CPU). In this case, both delay and throughput are reduced. As expected, vertical scaling is more expensive to implement compared to horizontal scaling.

Examining the example in Figure 4, initially, the delay is 6 hours, and the throughput is 1/3 hours. When applying horizontal scaling by changing the operator that lasts 3 hours to three operators that each last one hour, and the one that lasts 2 hours to two operators that each last one hour, the throughput is increased to 1 hour, but the delay remains at 6 hours. On the other hand, with vertical scaling, by replacing the two operators that take 3 and 2 hours with more powerful ones that take only 1 hour, the delay is reduced to three hours, and the throughput is also reduced to 1 hour.

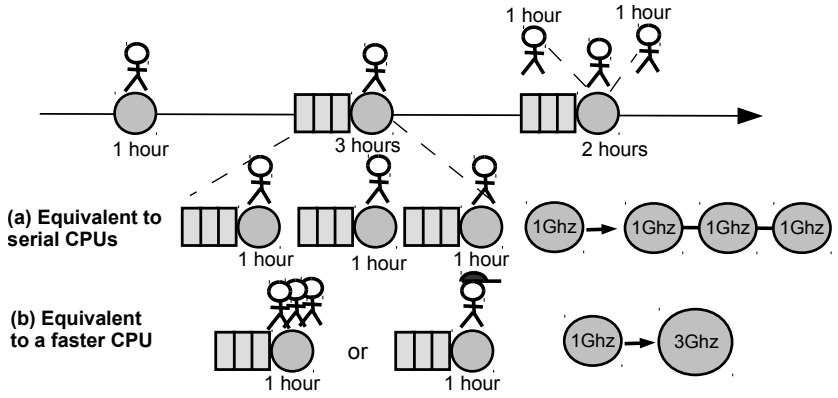

Figure 4: Example of horizontal and vertical scaling in a pipeline.

3 Improving Batch Processing KPIs

Our current design consists in first, closing a batch, generate the proof and then submit the L1 transaction to verify this batch. In this case, we have the following KPIs:

$$\texttt{delay} = \texttt{close\_a\_batch\_time} + \texttt{prove\_a\_batch\_time} + \texttt{block\_time} \text{ [seconds]}$$

$$\texttt{throughput} = \frac{1}{\texttt{prove\_a\_batch\_time}} \text{ [batches per second]}$$

Note that, for computing the throughput, we are assuming that closing, proving and verifying a batch can be done in parallel with respect to other batches, then it will depend on the longest part of the process, which is to prove the batch.

$$\texttt{close\_a\_batch\_time} \ll \texttt{prove\_a\_batch\_time}$$
$$\texttt{block\_time} \ll \texttt{prove\_a\_batch\_time}$$

In order to have some numbers:

$$\texttt{block\_time} = 15\,\text{s (average)}$$
$$\texttt{prove\_a\_batch\_time} = 120\text{s (min)}$$
$$\texttt{close\_a\_batch\_time} = 10\,\text{s (max)}$$

Improving the KPIs with Vertical Scaling Our goal is to increase the throughput and reduce the delay. The limiting factor in this case is the `prove_a_batch_time`. Vertically scaling the system, which means adding more resources to the existing machines, might seem like a straightforward solution to speed up proof generation. This can be done by running provers in more powerful machines, optimizing the proving system, or a combination of both. However, this approach has limitations:

- **Cost-Effectiveness**: Upgrading to very powerful machines often results in diminishing returns. The cost increase might not be proportional to the performance gain, especially for high-end hardware.
- **Optimization Challenges**: Optimizing the proof system itself can be complex and time-consuming.

Improving the KPIs with Horizontal Scaling Another option is to scale the system **horizontally**. Horizontal scaling involves adding more processing units (workers) to distribute the workload across multiple machines. In the context of our batch processing system, this translates to spinning up multiple provers to work in parallel. Figure 5 illustrates a naive implementation of horizontal scaling. Here's how it works:

1. **Parallelized Proof Generation:** Multiple provers are spinned.
2. **Proofs Reception:** Each prover sends its individual proof to an aggregator.
3. **Proofs Verification:** The aggregator puts all these proofs into an L1 transaction and sends it to the smart contract so that batches can be verified.

With this approach we have to close the batches in series but their proofs can be generated in parallel. Notice that the verification cost is proportional to the number of proofs sent.

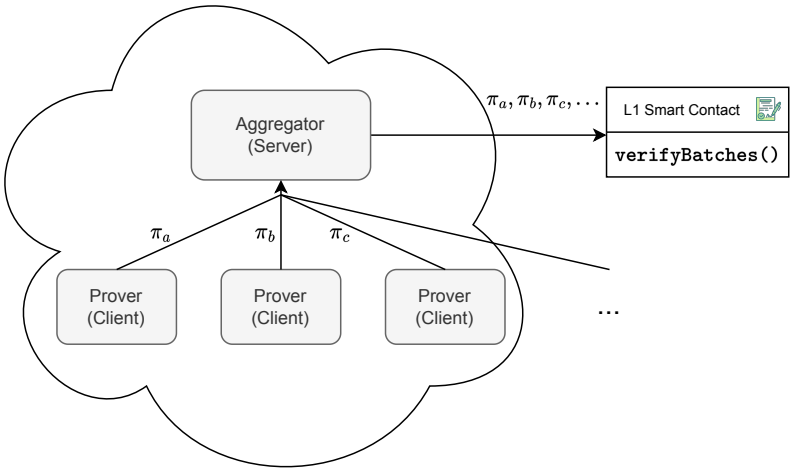

Figure 5: Naive approach of horizontal scaling.

The challenge with this straightforward approach is the associated costs both in terms of the space occupied by each proof and the verification expenses for each. Another option is to scale the system horizontally with **proof aggregation**, as shown in Figure 6. Here's how it works:

1. **Parallelized Proof Generation:** We initiate multiple provers.
2. **Proofs Reception:** Each prover sends its individual proof to an aggregator.
3. **Proof Aggregation:** The proofs are aggregated in a single proof.
4. **Proof Verification:** This aggregated proof is then encapsulated in an L1 transaction, which is transmitted to the smart contract for batch verification.

The cornerstone of this approach lies in our custom cryptographic backend, specifically its support for proof aggregation. This powerful technique allows us to combine multiple individual proofs into a single verifiable proof. The key advantage is that the verification cost on L1 remains constant regardless of the number of proofs aggregated. This is because

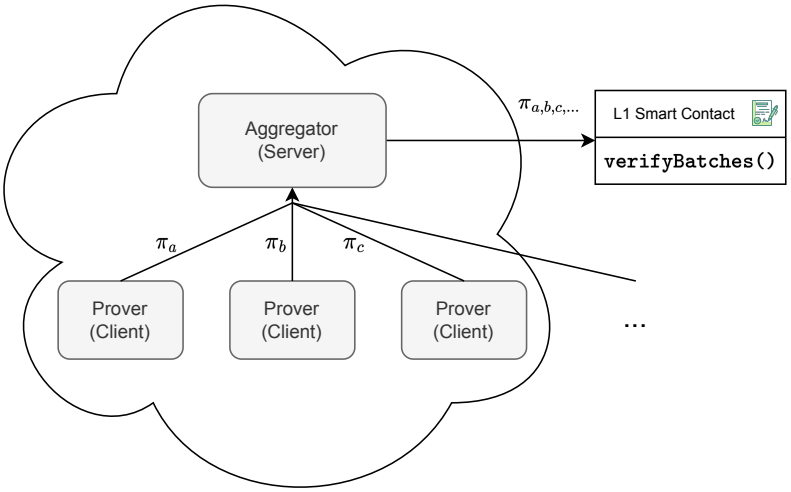

Figure 6: Horizontal scaling approach with proof aggregation.

the L1 smart contract only needs to verify the validity of the single, aggregated proof, not each individual proof.

Let's delve deeper into how this new horizontal scaling strategy boosts the system's throughput. A crucial metric in this process is the `aggregation_time`, which represents the time it takes to combine proofs from N batches, which is currently around 12 seconds. The throughput can be computed as follows, in batches per second:

$$\frac{N}{\max\left(\texttt{prove\_a\_batch\_time}, N \cdot \texttt{close\_a\_batch\_time}, \texttt{block\_time}, \texttt{aggregation\_time}\right)}$$

Observe that we can verify N batches taking the maximum of the time periods of the denominator, since operations in the denominator can run in parallel for sets of N batches. In the case that the maximum time in the denominator is `prove_a_batch_time`, notice that we have increased the systems throughput by factor of N.

On the other hand, the delay in this scenario can be computed as follows:

$$\texttt{delay} = N \cdot \texttt{close\_a\_batch\_time} + \texttt{prove\_a\_batch\_time} + \texttt{aggregation\_time} + \texttt{block\_time}$$

Henceforth, with the *straight aggregated batches approach* as you can observe, we are substantially increasing the delay regarding the *single batch approach*. As we have discussed earlier, the delay is a critical factor affecting the user experience. To retain the throughput gains while improving the delay, we can adopt a two-step approach for batch processing: first, order (also known as **sequence**) and then **prove**. This segmentation allows for optimization in each step, potentially enhancing the overall delay without compromising the achieved throughput improvements.

Enhancing Delay: Order then Prove The idea behind decoupling the batch sequencing (ordering) process from the batch proving process is:

- Provide a low delay response to users about their L2 transactions.
- While being able to aggregate transactions to provide a high system throughput.

Sequencing an L2 batch involves deciding the L2 transactions that will be part of the next batch (i.e., create or close the batch) and send them to L1. As the sequence of batches is written in L1, we have the data availability and immutability provided by L1. Sequenced batches may not be immediately proven, but they are guaranteed to be proven eventually. This creates a state within the L2 system that reflects the eventual outcome of executing those transactions, even though the proof hasn't been completed yet. We call these states **virtual states** because they represent a future state that will be consolidated once the proof is processed. More precisely, the Virtual state (See Figure 7) is the state reached after executing a batch that has been sequenced in L1 but that is not proved yet. This state is trustless since it relies on the L1 security.

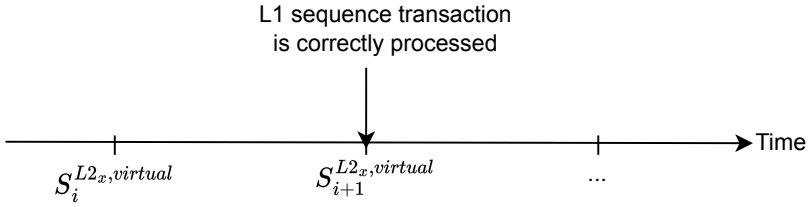

Figure 7: Definition of the Virtual state.

It's crucial for our users to understand that once a transaction is in the virtual state, its processing is guaranteed. The notable improvement lies in the ability to close batches more rapidly than the block time, providing a more efficient and expedited processing mechanism.

Now, let's adopt a revised definition for the delay, considering it as the duration from the moment a user submits an L2 transaction until that transaction reaches a virtual state. From the user's perspective, once the transaction is in the virtual state, it can be regarded as processed.

$$\texttt{delay}^{(\texttt{to\_virtual})} = \texttt{close\_a\_batch\_time} + \texttt{block\_time}$$

Remark that our smart contract allows us to sequence multiple batches per L1 sequence transaction, in this case, the delay can be computed as shown below. Observe that we have experienced an improvement in the delay.

$$\texttt{delay}^{(\texttt{to\_virtual})} = N \cdot \texttt{close\_a\_batch\_time} + \texttt{block\_time}.$$

Below, we present several advantages of decoupling sequencing batches from proving batches:

- **Queue Management**: This approach enables effective management of the queue for sequenced batches that are pending consolidation.
- **Flexibility in Delay and Prover Resources**: It becomes possible to adjust the amount of delay by adjusting the number of provers in operation.
- **User Perception**: Decoupling allows for adjustments in delay and resource allocation without impacting the perceived delay experienced by users.