

The zkEVM Architecture

Part V: Economics

Polygon zkEVM & Universitat Politècnica de Catalunya (UPC)

Marc Guzman-Albiol <marc.guzman.albiol@upc.edu>

Jose Luis Muñoz-Tapia <jose.luis.munoz@upc.edu>

Version: 0fd5d63ee0da211e457690d307873828c154069d

January 17, 2024

User Fees

Basic Ethereum Fee Schema i

The basic fee schema to which Ethereum users are used works as follows.

The gas is a unit that accounts the resources used when processing a transaction.

At the time of sending a transaction, the user can decide two parameters:

1. **gasLimit:**

- It is the maximum amount of gas units that a user enables to be consumed by the transaction.

2. **gasPrice:**

- It refers to the amount of Wei a user is willing to pay per unit of gas for the transaction execution.
- In more detail, there is a market between users and network nodes such that if a user wants to prioritize his transaction, then he has to increase the **gasPrice**.

- At the **start of the transaction processing**, the following amount of Wei is subtracted from the source account balance:

$$\text{gasLimit} \cdot \text{gasPrice}.$$

- Then,
 - If $\text{gasUsed} > \text{gasLimit}$, the transaction is reverted.
 - Otherwise, the amount of Wei associated with the unused gas is refunded.
- The refunded amount of Wei that is added back to the source account is calculated as:

$$\text{gasLimit} \cdot \text{gasPrice} - \text{gasUsed} \cdot \text{gasPrice}.$$

Generic User Fee Strategy of Layer 2 Solutions

- In general, Layers 2 follow the fee strategy of charging an L2 gasprice that is a percentage of the L1 `gasPrice`:

`L2GasPrice = L1GasPrice · L1GasPriceFactor.`

- For example:

`L1GasPrice = 20 Gwei`

`L1GasPriceFactor = 0.04 (4% of L1 gasPrice)`

`L2GasPrice = 20 Gwei · 0.04 = 0.8 Gwei`

- You can check the current fees at <https://l2fees.info>.

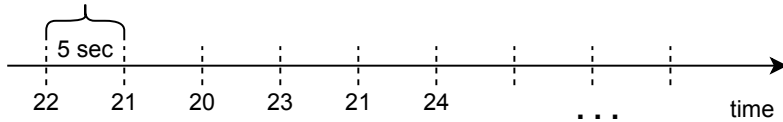
L2 User Fee Strategies are More Complex

However, this is not as easy as it may seem and there are additional aspects to consider:

- a) The **gasPrice** in L1 varies with time, so, how is this taken into account?
- b) Different **gasPrice** values in L1 can be used to prioritize transactions, how are these priorities managed by the L2 solution?
- c) The **gas/gasPrice** L1 schema may not be aligned with the actual resources spent by the L2 solution.

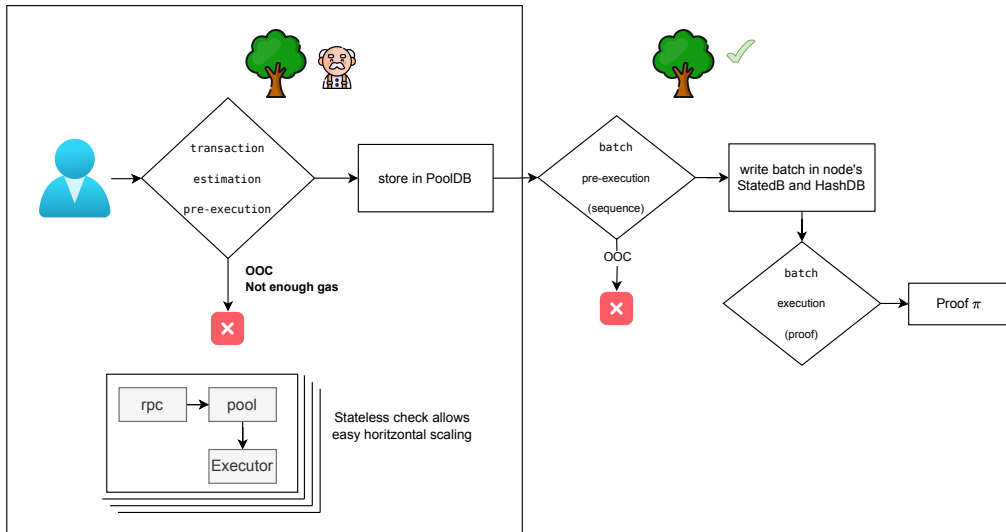
Obtaining L1 GasPrices

IntervalToRefreshGasPrices



In the example, we poll for the L1 gasPrice every 5 seconds and, as shown, gas prices vary with time.

RPC Transaction Pre-execution

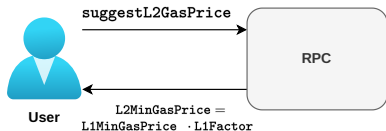


Sending a Transaction: User Experience

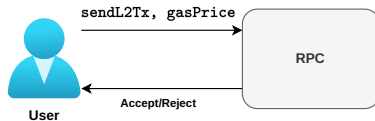
- We will operate in two steps: **gasPrice** suggestion and transaction sending.
- In the first step **A**, the user will ask via RPC call for a suggested **gasPrice** computed as

$$L2GasPrice = L1GasPrice \cdot L1GasPriceFactor$$

to sign its transaction with.



- In the second step **B**, the user sends the desired L2 transaction together with the **gasPrice**.

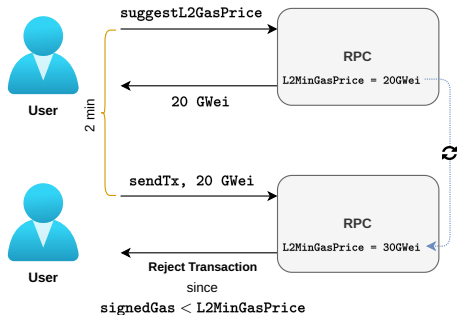


- If **gasPrice** provided by the user is less than the current **L2GasPrice**, the transaction is automatically **rejected**^a and not included into the pool (error `ErrGasPrice`).

^a Recall that the transaction can be rejected due to other checks explained before.

Sending a Transaction: Bad User Experience

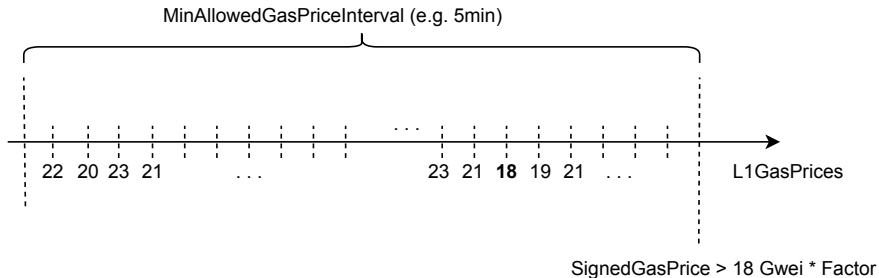
- However, this previous schema is not a good UX.
- Between steps **A** and **B** there is an unbounded interval of time.
- Henceforth, the **L2GasPrice** from step **A** can be different from the one present in step **B**, leading to the following unwanted situation:



- Observe that, since the **L2GasPrice** has been refreshed, the transaction sent by the prover will be rejected even though it was signed with the exact suggested **gasPrice**.

Sending a Transaction: `MinAllowedPriceInterval`

The solution is to allow transactions from users that have signed any L2 `gasPrice` that is above the minimum L2 `gasPrice` recorded during a period of time (called `MinAllowedPriceInterval`).



We can configure the previous parameters in the Polygon zkEVM node:

```
1 [Pool]
2 ...
3 DefaultMinGasPriceAllowed = 0
4 MinAllowedGasPriceInterval = "5m"
5 PollMinAllowedGasPriceInterval = "1s"
6 IntervalToRefreshGasPrices = "5s"
7 ...
```

<https://github.com/0xPolygonHermes/zkevm-node/blob/develop/docs/config-file/node-config-doc.md#75-pooldb>

- **DefaultMinGasPriceAllowed:** It is the default min gas price to suggest.
- **MinAllowedGasPriceInterval:** It is the interval to look back of the suggested min gas price for a transaction.
- **PollMinAllowedGasPriceInterval:** It is the interval to poll L1 to find the suggested L2 min gas price.
- **IntervalToRefreshGasPrices:** It is the interval to refresh L2 gas prices.

When computing the L1 `gasPrice`, we can activate the `multigasprovider`:

```
1 [Ethereum]  
2 ...MultiGasProvider = false
```

When enabled, it allows using multiples sources for computing the L1 `gasPrice`.

Gas Price Suggester

```
1 [L2GasPriceSuggester]
2 Type = "follower"
3 UpdatePeriod = "3s"
4 Factor = 0.12
5 DefaultGasPriceWei = 0
6 MaxGasPriceWei = 0
7 CleanHistoryPeriod = "1h"
8 CleanHistoryTimeRetention = "5m"
```

L1/L2 Costs Issues

- Gas in Ethereum accounts the resources used by the transaction.
- In particular, it takes into account:
 - **Data availability** (the transaction bytes).
 - **Processing resources**, like CPU, Memory and Storage.
- Ethereum users are used to prioritize their transaction by increasing **gasPrice**.
- A big issue is that **there can be operations that consume low gas in L1 but that represent a major cost for L2**.
- The data availability costs are fixed once the transaction is known and they are directly proportional to L1 data availability costs.
- However, L2 execution is variable (because it depends on the state) and usually offers a smaller cost per gas.
- Henceforth, L2 transactions having high data availability costs and small execution costs are **highly problematic** in our pricing schema.

L1/L2 Costs Strategies

- Recall that the Ethereum fee is computed as $\text{gasUsed} \cdot \text{gasPrice}$, giving us two ways of solving the misalignment problem:

(A) Arbitrum Approach. Increase **gasUsed**.

- This approach is based on changing the gas schema to increase the Gas costs for data availability.
- This strategy is a relatively simple to implement and easy to understand but **it changes the Ethereum protocol**.
- An L1 Ethereum transaction may execute different when compared to the same transaction executed in L2.

(B) Effective Gas Price Approach. Increase **gasPrice**.

- If we do not want to modify the Gas, we have to increase **gasPrice** in order to cover the costs.
- Unlike the previous approach, this does not change the Ethereum specifications.
- However, it is complex to achieve a fair **gasPrice**.
- Moreover, we have to take into account that L2 users should be able to prioritize its transactions also increasing **gasPrice**, as they are used to.
- This is actually our approach.

Effective Gas Price Overview i

- The user signs a relatively high gas price at the time of sending the L2 transaction.
- Later on, by pre-executing the sent transaction, the **sequencer** establishes a fair **gasPrice** according to the amount of resources used.
- To do so, the **sequencer** provides a single byte **effectivePercentageByte** $\in \{0, 1, \dots, 255\}$ (1 Byte), which will be used to compute a ratio called **effectivePercentage**

$$\text{effectivePercentage} = \frac{1 + \text{effectivePercentageByte}}{256}.$$

- The **effectivePercentage** will be used in order to compute the factor of the signed transaction's **gasPrice** which should be charged to the user:

$$\text{txGasPrice} = \left\lfloor \text{signedGasPrice} \cdot \frac{1 + \text{effectivePercentageByte}}{256} \right\rfloor.$$

Effective Gas Price Overview ii

- For example, setting an `effectivePercentageByte` of $255 = 0xFF$ would mean that the user would pay the totality of the `gasPrice` signed when sending the transaction:

$$\text{txGasPrice} = \text{signedGasPrice}.$$

- In contrast, setting `effectivePercentageByte` to 127 would reduce the `gasPrice` signed by the user to the half:

$$\text{txGasPrice} = \frac{\text{signedGasPrice}}{2}.$$

- Observe that, in this schema, users **must trust the sequencer**.
- As having `effectivePercentage` implies having `effectivePercentageByte`, and vice versa, we will abuse of notation and use them interchangeably as `effectivePercentage`.

About the **effectivePercentage** computation

- We could account the pricing resources by means of the number of **consumed counters** present in our proving system.
- Nevertheless, comprehending this can be challenging for users, and it is crucial to prioritize a positive user experience in this specific aspect.
- Moreover, stating the efficiency through counters is not intuitive for users at the time of prioritizing their transactions.
- Henceforth, our actual goal is to compute **effectivePercentage** only by using **Gas** and prioritizing users transactions by means of using **gasPrice**.

Introduction of the `breakEvenGasPrice`

- Our goal as service providers is to **not accept transactions in which we loose money**.
- In order to achieve this, we will calculate the `breakEvenGasPrice`, considering a secure threshold to avoid losses in the event of unexpected issues.
- As explained before, we will split the computation in two to take into account differently costs associated with data availability and costs associated with used Gas.

breakEvenGasPrice: Costs Associated with Data Availability i

- Costs associated with Data Availability will be computed as

$$\text{dataCost} \cdot \text{L1GasPrice},$$

where **dataCost** is the cost in Gas for data in L1.

- In the Ethereum ecosystem, the cost of data varies depending on whether it involves zero bytes or non-zero bytes

$$\text{NonZeroByteGasCost} = 16, \quad \text{ZeroByteGasCost} = 4$$

- In particular, **non-zero bytes** cost 16 Gas meanwhile **zero bytes** 4 Gas.

breakEvenGasPrice: Costs Associated with Data Availability ii

- Also recall that, when computing non-zero bytes cost we should take into account some constant data¹, always appearing in a transaction:
 - The **signature**, consisting on 65 bytes.
 - The **effectivePercentageBytesLength**, consisting on 1 bytes related to the RLP-encoded fields length.
- This results in a total of 66 constantly present bytes.
- Taking all in consideration, **dataCost** can be computed as:

$$(\text{txConstBytes} + \text{txNonZeroBytes}) \cdot \text{NonZeroByteGasCost} + \text{txZeroBytes} \cdot \text{ZerByteGasCost},$$

where **txZeroBytes** (resp. **txNonZeroBytes**) represents the count of zero bytes (resp. non-zero bytes) in the raw transaction sent by the user.

¹This data can obtain zero bytes, but to optimize a little bit the processing we count them all of them as non-zero bytes.

breakEvenGasPrice: Computational Costs

- For the computational cost, we will simply use the following formula:

$$\text{gasUsed} \cdot \text{L2GasPrice},$$

where recall that we can obtain `L2GasPrice` by multiplying `L1GasPrice` by chosen factor less than 1:

$$\text{L2GasPrice} = \text{L1GasPrice} \cdot \text{L1GasPriceFactor}.$$

- In particular, we will choose a factor of 0.25

$$\text{L1GasPriceFactor} = 0.25.$$

- Observe that, unlike data costs, in order to compute computational costs we will need to **execute** the transaction.

effectiveGasPrice Formula

- Now, combining both **data** and **computational** costs, we will refer to it as **totalTxPrice**:

$$\text{totalTxPrice} = \text{dataCost} \cdot \text{L1GasPrice} + \text{gasUsed} \cdot \text{L1GasPrice} \cdot \text{L1GasPriceFactor}.$$

- We can compute **breakEvenGasPrice** as the following ratio:

$$\text{breakEvenGasPrice} = \frac{\text{totalTxPrice}}{\text{gasUsed}}.$$

- This calculation helps to establish the gas price at which the total transaction cost is covered.
- Additionally, we incorporate a factor **netProfit** ≥ 1 that allows us to achieve a slight profit margin:

$$\text{breakEvenGasPrice} = \frac{\text{totalTxPrice}}{\text{gasUsed}} \cdot \text{netProfit}.$$

- Observe that we still need to introduce here **gasPrice** prioritization.

- Prioritization of transactions in Ethereum is determined by the signed **gasPrice**; higher values result in higher priority.
- To implement this, consider that users are only aware of two **gasPrice** values: the one signed with the transaction, called **gasPriceSigned**, and the one obtained from the RPC, that we will call **gasPriceSuggested**.

Introducing Priority ii

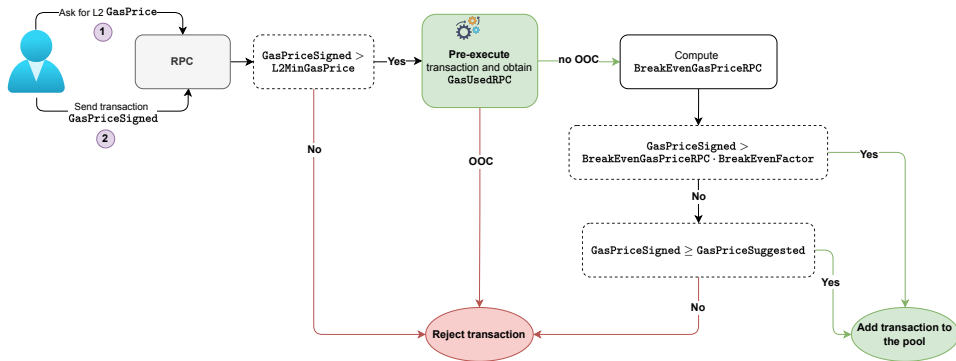
- In the case that `gasPriceSigned > gasPriceSuggested`, we establish a priority ratio as follows:

$$\text{priorityRatio} = \frac{\text{gasPriceSigned}}{\text{gasPriceSuggested}}.$$

- If `gasPriceSigned ≤ gasPriceSuggested`, the user has chosen not to prioritize its transaction (and maybe we can reject the transaction due to low gas price).
- In this case, we establish a priority ratio to be 1.
- The `effectiveGasPrice` will be computed as:

$$\text{effectiveGasPrice} = \text{breakEvenGasPrice} \cdot \text{priorityRatio}.$$

gasPrice Flows: RPC i



1. The user asks to the RPC for a suggested L2 **GasPrice**.
2. The users sends the transaction together with a selected **GasPriceSigned**.
3. The RPC pre-executes the transaction (**important**, using a wrong state root) and uses the gas used **GasUsedRPC** in order to compute the **BreakEvenGasPriceRPC**.
4. We have two cases here:
 - If the transaction pre-execution runs out of counters (**OCC** error), we immediately reject the transaction.
 - If not, the RPC computes the **BreakEvenGasPriceRPC** and we continue the flow.

5. Now, we have two options:

- If $\text{GasPriceSigned} > \text{BreakEvenGasPriceRPC} \cdot \text{BreakEvenFactor}$, we immediately accept the transaction, storing it in the pool.
- **BreakEvenFactor** (which is equal to 1.3) is introduced to provide a wider safeness threshold.
- Otherwise $\text{GasPriceSigned} \leq \text{BreakEvenGasPriceRPC} \cdot \text{BreakEvenFactor}$, we are in dangerous zone because we may be facing losings.

6. In the bad path, we allow two options:

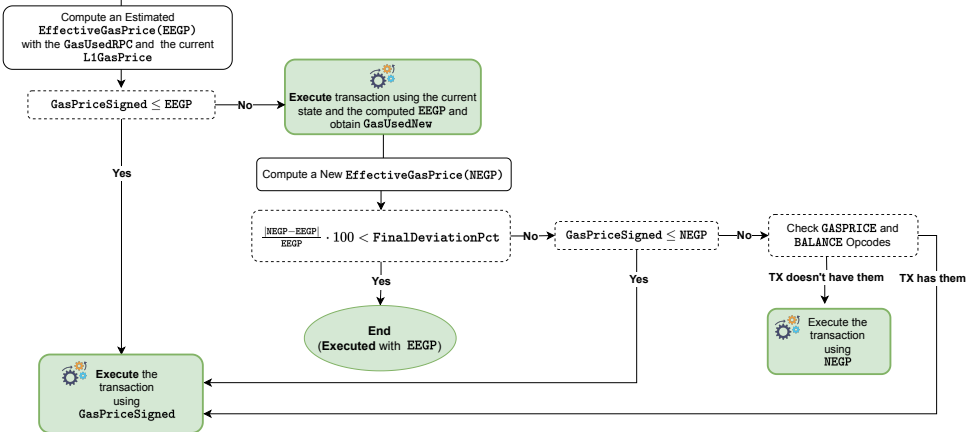
- If $\text{GasPriceSigned} \geq \text{GasPriceSuggested}$, we take the risk and we introduce the transaction into the pool.
- Otherwise $\text{GasPriceSigned} < \text{GasPriceSuggested}$, we immediately reject the transaction because its highly probable that we face losings.

gasPrice Flows: RPC. Some Considerations

- It is important to remark that, **once a transaction is included into the pool, we should actually include it into a block.**
- Hence, if something goes bad in later steps and the processing consumes far more gas than expected, we will **lose money having no possibility to overcome that situation.**
- On the contrary, if the process goes well and the processing consumes less gas than expected, we can reward the user by modifying the previously introduced **effectivePercentage**.
- Also observe that all the transactions stored in the **Pool** should be ordered from larger to lower priorities (or, equivalently, **effectiveGasPrice**).

gasPrice Flows: Sequencer i

Sequence a transaction from the pool



1. The sequencer takes a transaction from the **Pool** and recomputes the estimated **EffectiveGasPrice** (which we will call **EEGP**) using the **GasUsedRPC** stored by the RPC and the current **L1GasPrice**.
2. At this point, we have two options:
 - If $\text{GasPriceSigned} \leq \text{EEGP}$, there is a risk of loss.
 - In such cases, the user is charged the full **GasPriceSigned** and we end up the flow.
 - Conversely, if $\text{GasPriceSigned} > \text{EEGP}$, there is room for further adjustment of the user's gas price.
3. In this case, we recompute a new **EffectiveGasPrice** (which we will call **NEGP**) with the execution-related data compute from the **correct state root**.

4. We have two paths:

- If the difference between EEGP and NEGP is higher than some a parameter `FinalDeviationPct` (which is 10 in the actual configuration):

$$\frac{|\text{NEGP} - \text{EEGP}|}{\text{EEGP}} \cdot 100 < \text{FinalDeviationPct},$$

we end up the flow just to avoid re-executions and save execution resources.

- On the contrary, if the difference equals or exceeds the deviation parameter, there is a big difference between executions and we may better adjust gas price.

5. In the later case, two options arise:

- If `GasPriceSigned` \leq `NEGP` there is again a risk of loss.
- In such cases, the user is charged the full `GasPriceSigned` and we end up the flow.
- Otherwise, if `GasPriceSigned` $>$ `NEGP`, means that we have margin to adjust the gas price.
- However, we want to **save executions**, leading us to end up the process using a trick explained below.

6. We check if the transaction processing includes the two opcodes that use the gas price:
 - The **GASPRICE** opcode.
 - The **BALANCE** opcode from the source address.
7. If it is the case, to save one execution, we simply execute the transaction using the full **GasPriceSigned** to ensure we minimize potential losses and we end up the flow, as before.
8. If not, and with the intention of optimizing an execution while making a slight adjustment to the gasPrice, we proceed by executing the transaction using the **NEGP**.

Pool Effective Gas Price

```
1  ...  
2  [Pool.EffectiveGasPrice]  
3  Enabled = false  
4  L1GasPriceFactor = 0.04  
5  ByteGasCost = 16  
6  ZeroByteGasCost = 4  
7  NetProfit = 1.2  
8  BreakEvenFactor = 1.3  
9  FinalDeviationPct = 10  
10 L2GasPriceSuggesterFactor = 0.3
```

<https://github.com/0xPolygonHermes/zkevm-node/blob/develop/docs/config-file/node-config-doc.md>

- **L1GasPriceFactor**: is the percentage of the L1 gas price that will be used as the L2 min gas price
- **ByteGasCost**: cost per byte that is not 0.
- **ZeroByteGasCost**: cost per byte that is 0.
- **NetProfit**: is the profit margin to apply to the calculated breakEvenGasPrice.
- **BreakEvenFactor**: is the factor to apply to the calculated breakEvenGasPrice when comparing it with the gasPriceSigned of a tx.
- **FinalDeviationPct**: is the max allowed deviation percentage BreakEvenGasPrice on re-calculation
- **L2GasPriceSuggesterFactor**: is the factor to apply to L1 gas price to get the suggested L2 gas price used in the calculations when the effective gas price is disabled (testing/metrics purposes)