# The zkEVM Architecture

## Part VI: 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: d2193c5e264d65b18689e43f32f8562c250eb991

February 22, 2024

# Outline

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`.

# Basic Ethereum Fee Schema ii

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

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

- Then,
  - If $\texttt{gasUsed} > \texttt{gasLimit}$, 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:

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

# Generic User Fee Strategy of Layer 2 Solutions

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

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

- For example:

  $\texttt{L1GasPrice} = 20$ Gwei
  $\texttt{L1GasPriceFactor} = 0.04$ (4% of L1 `gasPrice`)
  $\texttt{L2GasPrice} = 20 \text{ Gwei} \cdot 0.04 = 0.8$ Gwei

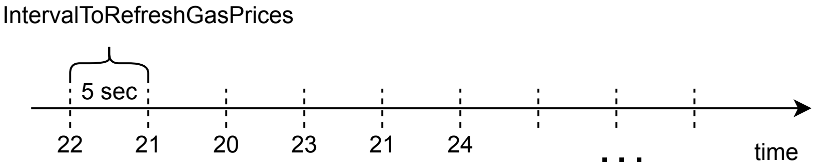
- You can check the current fees at `https://l2fees.info`.

# L2 User Fee Strategies are More Complex

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

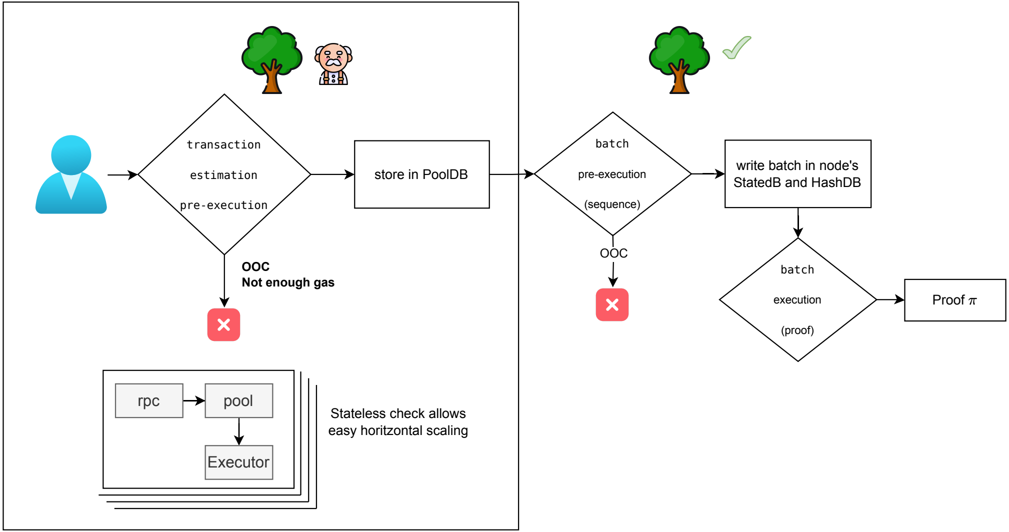
a) The gas price in L1 varies with time, so, how is this taken into account?
b) Different gas price values in L1 can be used to prioritize transactions, how are these priorities managed by the L2 solution?
c) The L1 gas schema may not be aligned with the actual resources spent by the L2 solution.

# Obtaining L1 GasPrices




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

# RPC Transaction Pre-execution



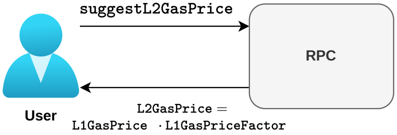
# Gas Price Suggester: Naive Approach

- We will operate in two steps: `gasPrice` suggestion and transaction sending.

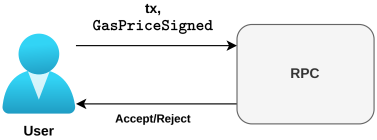
- First, the user will ask via RPC call for a suggested gas price computed as

$$\mathtt{L2GasPrice} = \mathtt{L1GasPrice} \cdot \mathtt{L1GasPriceFactor}$$

to sign its transaction with.



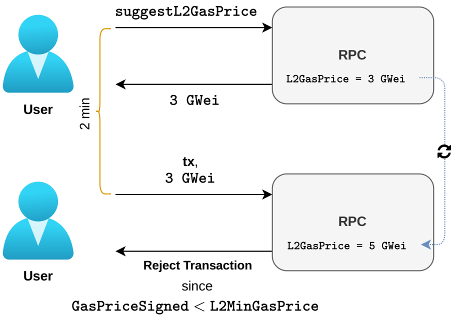

- Now, the user sends the desired L2 transaction with a choice of gas price that we will call **signed gas price**, denoted as `GasPriceSigned`.




- If signed gas price is less than the current `L2GasPrice`, the transaction is automatically **rejected** and not included into the pool (error `ErrGasPrice`).

# Naive Approach: Bad User Experience

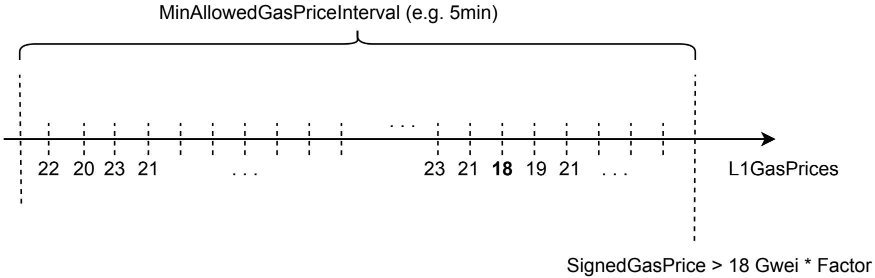
- However, there is a time gap between asking for a suggested gas price and sending the transaction in which the gas price can fluctuate.
- Henceforth, the `L2GasPrice` from at asking for the suggestion can be different from the suggested one at the time of sending the transaction, leading to the following unwanted situation:




- Observe that, since the `L2GasPrice` has been refreshed, the transaction sent by the prover will be rejected even tough it was signed with the exact suggested gas price.

# Gas Price Suggester: Decision Interval Approach

- The solution is to allow transactions from users that have signed any `GasPriceSigned` that is above the minimum L2 gas price recorded during a period of time (called `MinAllowedPriceInterval`).
- This minimum is denoted as `L2MinGasPrice`.



# Pool Config

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

```
[Pool]
...
DefaultMinGasPriceAllowed = 0
MinAllowedGasPriceInterval = "5m"
PollMinAllowedGasPriceInterval = "1s"
IntervalToRefreshGasPrices = "5s"
...
```

https://github.com/0xPolygonHermez/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.

# Multi Gas Provider

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

```
[Etherman]
...MultiGasProvider = false
```

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

# Gas Price Suggester: Final Approach

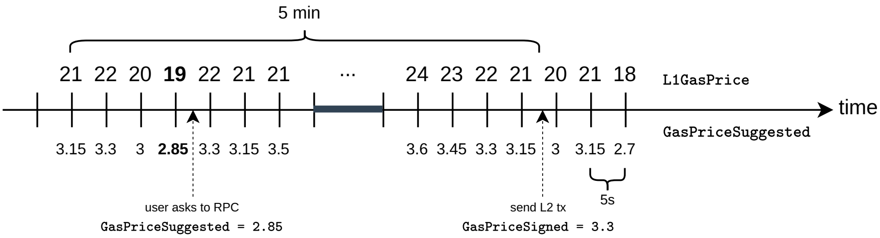
- However, with this particular design, the zkEVM endpoint that provides a suggestion for the gas price that the user has to sign with its transaction (which will be called L2 Gas Price Suggester) has a **big problem design.**
- Recall that the price of posting transactional data to L1 is charged to the zkEVM network to the **full L1 price**.
- Therefore, if we propose a gas price using `L1GasPriceFactor`, representing the measure of computational reduction in L2, there is a risk of running out of Wei reserves for posting data to L1.
- Consequently, we will recommend a slightly higher percentage of the gas price to the user, employing a `SuggesterFactor` of $0.15 \approx 4 \cdot \texttt{L1GasPriceFactor}$:

$$\texttt{GasPriceSuggested} = \texttt{L1GasPrice} \cdot \texttt{SuggestedFactor}.$$

# Gas Price Suggester

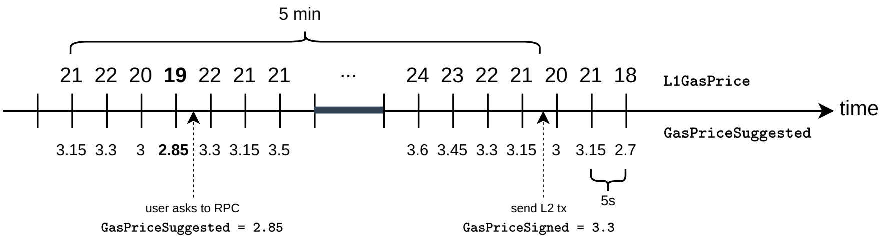
```toml
[L2GasPriceSuggester]
Type = "follower"
UpdatePeriod = "3s"
Factor = 0.12
DefaultGasPriceWei = 0
MaxGasPriceWei = 0
CleanHistoryPeriod = "1h"
CleanHistoryTimeRetention = "5m"
```

# Numerical Example: Minimum Gas Price i




- Observe that, when the user queries the suggested gas price through the RPC, the networks responds the current suggested gas price computed as $0.15 \cdot 19$, which is the current L1 gas price updated every 5 seconds.

# Numerical Example: Minimum Gas Price ii




- However, at the time of sending the transaction, the RPC will only accept the transaction if `GasPriceSigned` is strictly higher than the minimum suggested gas price from 5 minutes ago (highlighted in **bold** in the figure), which in this instance is $19 \cdot 0.15 = 2.85$.
- In order to get his transaction accepted, the user sets the gas price of the transaction to `GasPriceSigned` $= 3.3 > 2.85 =$ `L2MinGasPrice`.

# 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 priorize their transaction by increasing signed gas price when sending it.
- 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 `gasUsed` · `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 gas price.
  - Moreover, we have to take into account that L2 users should be able to prioritize its transactions also increasing gas price, 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 gas price according to the amount of resources used.
- To do so, the **sequencer** provides a single byte $\texttt{EffectivePercentageByte} \in \{0, 1, \ldots, 255\}$ (1 Byte), which will be used to compute a ratio called $\texttt{EffectivePercentage}$

$$\texttt{EffectivePercentage} = \frac{1 + \texttt{EffectivePercentageByte}}{256}.$$

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

$$\texttt{TxGasPrice} = \left\lfloor \texttt{GasPriceSigned} \cdot \frac{1 + \texttt{EffectivePercentageByte}}{256} \right\rfloor.$$

# Effective Gas Price Overview ii

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

$$\mathtt{TxGasPrice} = \mathtt{GasPriceSigned}.$$

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

$$\mathtt{TxGasPrice} = \frac{\mathtt{GasPriceSigned}}{2}.$$

- 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 trough 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 gas price:

$$\texttt{EffectivePercentage} = \frac{\texttt{GasPriceFinal}}{\texttt{GasPriceSigned}}$$

- Observe that, by modifying `GasPriceFinal`, which is the gas price charged at the end of the whole processing by the sequencer, we can modify the amount of Wei that we will charge to the user in order to process the sent transaction.

## Introduction of the `BreakEvenGasPrice`

- Our goal as service providers is to **not accept transactions in which we loose money**.
- To attain this goal, we will determine the `BreakEvenGasPrice`, representing the lowest gas price at which we do not incur losses.
- 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

$$\texttt{DataCost} \cdot \texttt{L1GasPrice},$$

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

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

- 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 data always appearing in a transaction and are not included in the RLP:
  - The **signature**, consisting on 65 bytes.
  - The previously defined `EffectivePercentageByte`, which consists in a single byte.
- This results in a total of 66 constantly present bytes.
- Taking all in consideration, `DataCost` can be computed as:

$$(\texttt{TxConstBytes} + \texttt{TxNonZeroBytes}) \cdot \texttt{NonZeroByteGasCost} + \texttt{TxZeroBytes} \cdot \texttt{ZeroByteGasCost},$$

  where `TxZeroBytes` (resp. `TxNonZeroBytes`) represents the count of zero bytes (resp. non-zero bytes) in the raw transaction sent by the user.

## `BreakEvenGasPrice`: Computational Costs

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

$$\texttt{GasUsed} \cdot \texttt{L2GasPrice},$$

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

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

- In particular, we will choose a factor of 0.04

$$\texttt{L1GasPriceFactor} = 0.04.$$

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

# `BreakEvenGasPrice` Formula

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

$$\texttt{TotalTxPrice} = \texttt{DataCost} \cdot \texttt{L1GasPrice} + \texttt{GasUsed} \cdot \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor}.$$

- We can compute `BreakEvenGasPrice` as the following ratio:

$$\texttt{BreakEvenGasPrice} = \frac{\texttt{TotalTxPrice}}{\texttt{GasUsed}}.$$

- This calculation helps to establish the gas price at which the total transaction cost is covered.
- Additionally, we incorporate a factor $\texttt{NetProfit} \geq 1$ that allows us to achieve a slight profit margin:

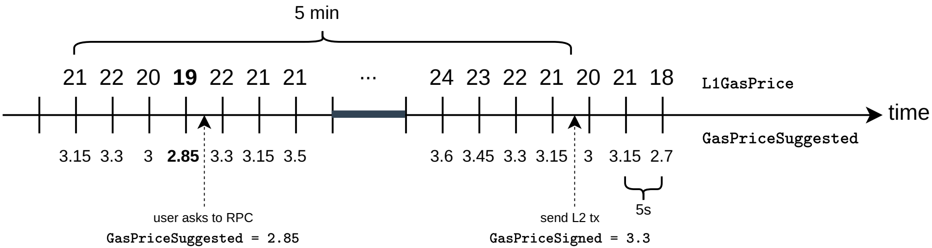
$$\texttt{BreakEvenGasPrice} = \frac{\texttt{TotalTxPrice}}{\texttt{GasUsed}} \cdot \texttt{NetProfit}.$$

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

# Numerical Example: `BreakEvenGasPrice` i

- Recall the example proposed before, where the user ended up by setting `GasPriceSigned` to 2.85.
- Suppose the user sends a transaction having:
    - 200 non-zero bytes, including the constant ones.
    - 100 zero bytes.
- Moreover, at the time of pre-executing the transaction (without getting an OOC error), $60,000$ Gas is consumed (recall that, since we are using a *wrong* state root, this gas is only an estimation).

# Numerical Example: **BreakEvenGasPrice** ii




- Hence, the total transaction cost is of

$$(200 \cdot 16 + 100 \cdot 4) \cdot 21 + 60,000 \cdot 21 \cdot 0.04 = 126,000 \text{ GWei}.$$

- Observe that 21 is the `L1GasPrice` at the time of sending the transaction.

- Now, we are able to compute the `BreakEvenGasPrice` as

$$\texttt{BreakEvenGasPrice} = \frac{\texttt{TotalTxPrice}}{\texttt{GasUsed}} = \frac{126,000 \text{ GWei}}{60,000 \text{ Gas}} \cdot 1,2 = 2.52 \text{ GWei/Gas}.$$

- Observe that we have introduced a `NetProfit` value of 1.2, indicating a target of a 20% gain in this process.
- At a first glance, we might conclude acceptance since $\texttt{GasPriceSigned} = 3.3 > 2.52$ but, recall that this is only an estimation, gas consumed with the correct state root can differ.
- Therefore, we introduce a `BreakEvenFactor` of 30% to account for estimation uncertainties:

$$\texttt{GasPriceSigned} = 3.3 > 3.276 = 2.52 \cdot 1.3 = \texttt{BreakEvenGasPrice} \cdot \texttt{BreakEvenFactor}.$$

- Consequently, we decide to **accept the transaction**.

# Numerical Example: `BreakEvenFactor` i

- Imagine we disable the `BreakEvenFactor` setting it to 1.
- Our original transaction's pre-execution consumed 60k Gas, $\texttt{GasUsedRPC} = 60\text{k}$.
- However, imagine that the correct execution at the time of sequencing consumes 35k Gas.
- If we recompute `BreakEvenGasPrice` using this updated used gas, we get 3.6 GWei/Gas, which is way higher than the original one.
- That means that, we should have charged the user with a higher gas price in order to cover the whole transaction cost, which now is of $105,000$ GWei.
- But, since we are accepting all the transactions signing more than 2.85 of gas price, we do not have margin to increase more.

## Numerical Example: `BreakEvenFactor` ii

- In the worst case we are loosing

$$105,000 - 35,000 \cdot 2.85 = 5,250 \text{ GWei}.$$

- Introducing `BreakEvenFactor` we are limiting the accepted transactions to the ones having

$$\texttt{GasPriceSigned} \geq 3.27,$$

in order to compensate such losses.

- In this case, we have the flexibility to avoid losses and adjust both user and our benefits since

$$105,000 - 35,000 \cdot 3.27 < 0.$$

- **Final Note**: In the example, even though we assumed that the decrease in `BreakEvenGasPrice` is a result of executing with a correct state root, it can also decrease significantly due to a substantial reduction in `L1GasPrice`.

# Introducing Priority i

- Prioritization of transactions in Ethereum is determined by `GasPriceSigned`; higher values result in higher priority.
- To implement this, consider that users are only aware of two gas price values: the one signed with the transaction, called `GasPriceSigned`, the current `GasPriceSuggested`, which is the one that provides the RPC.

# Introducing Priority ii

- At the time of sequencing a transaction, we should prioritize ones among the others, depending basically on both $\texttt{GasPriceSigned}$ and current $\texttt{GasPriceSuggested}$.
- In the case that $\texttt{GasPriceSigned} > \texttt{GasPriceSuggested}$, we establish a priority ratio as follows:

$$\texttt{PriorityRatio} = \frac{\texttt{GasPriceSigned}}{\texttt{GasPriceSuggested}} - 1.$$

- If $\texttt{GasPriceSigned} \leq \texttt{GasPriceSuggested}$, the user has chosen not to proritize its transaction (and maybe we can reject the transaction due to low gas price).
- In this case, we establish a priority ratio to be 0.
- The $\texttt{EffectiveGasPrice}$ will be computed as:

$$\texttt{EffectiveGasPrice} = \texttt{BreakEvenGasPrice} \cdot (1 + \texttt{PriorityRatio}).$$

## Numerical Example: `EffectiveGasPrice`

- Recall that, in the example, we were signing a gas price of 3.3 at the time of sending the transaction.
- Suppose that, at the time of sequencing a transaction, the suggested gas price is

$$\texttt{GasPriceSuggested} = 3.$$

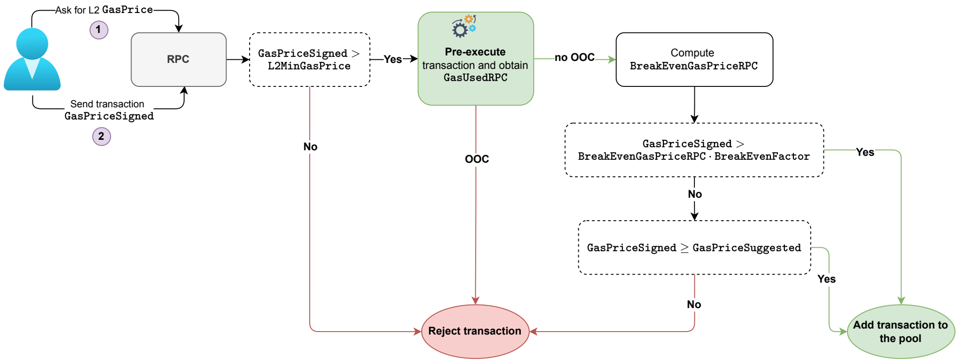
- The difference between both is taken into account in the priority ratio:

$$\texttt{PriorityRatio} = \frac{3.3}{3} - 1 = 0.1.$$

- Henceforth, the estimated `EffectiveGasPrice` (that is, the one using the RPC gas usage estimations) will be

$$\texttt{EffectiveGasPrice} = 2.52 \cdot (1 + 0.1) = 2.772.$$

# gasPrice Flows: RPC i



# `gasPrice` Flows: RPC ii

1. The user asks to the RPC for `GasPriceSuggested`.
2. The users sends the transaction together with a selected `GasPriceSigned`.
3. If `GasPriceSigned` $\leq$ `MinL2GasPrice`, we reject the transaction.
4. If the transaction was not rejected, the RPC pre-executes the transaction (`important`, using a wrong state root) to obtain `GasUsedRPC`, which is used in order to compute the `BreakEvenGasPriceRPC`.
5. We have two cases scenarios:
    - 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.

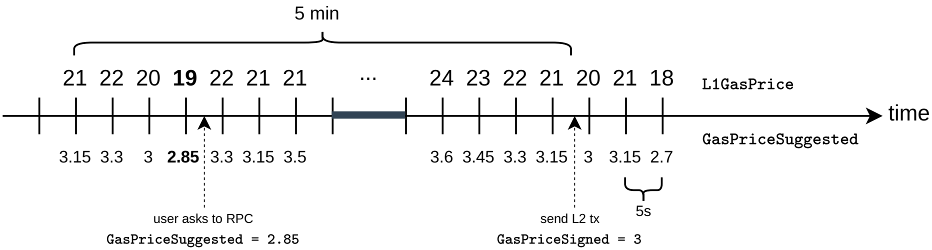
6. Now, we have two options:
   - If $\texttt{GasPriceSigned} > \texttt{BreakEvenGasPriceRPC} \cdot \texttt{BreakEvenFactor}$, we immediately accept the transaction, storing it in the Pool.
   - Otherwise $\texttt{GasPriceSigned} \leq \texttt{BreakEvenGasPriceRPC} \cdot \texttt{BreakEvenFactor}$, we are in dangerous zone because we may be facing losings due high data availability costs or to fluctuation between future computations.
7. In the dangerous path, we allow two options:
   - If $\texttt{GasPriceSigned} \geq \texttt{GasPriceSuggested}$, we take the risk of possible losses, sponsoring the difference if necessary and we introduce the transaction into the Pool.
   - Otherwise $\texttt{GasPriceSigned} < \texttt{GasPriceSuggested}$, 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 gas consumption deviates significantly from the initial estimate, we will **loose money having no possibility to overcome that situation**.
- On the contrary, if the process goes well and the consumed gas is similar to the estimated one, we can reward the user, modifying the previously introduced `EffectivePercentage`.
- Additionally, it's important to observe that, among all the transactions stored in the **Pool**, the ones prioritized for sequencing are the ones with higher `EffectiveGasPrice`.

# Numerical Example: RPC Flow i

- Recall the previous scenario:




1. The users asks to the RPC and gets

$$\texttt{GasPriceSuggested} = 0.15 \cdot 19 = 2.85.$$

2. Imagine the users sends a transaction signed with a gas price of 3:

$$\texttt{GasPriceSigned} = 3.$$

3. Now, the RPC pre-executes the transaction and gets $\texttt{GasUsedRPC} = 60\text{k Gas}$.
4. Imagine that the pre-execution does not ended running out of counters, so we compute `BreakEvenGasPrice` supposing same conditions as before, getting:

$$\texttt{BreakEvenGasPrice} = 2.52.$$

5. In this case,

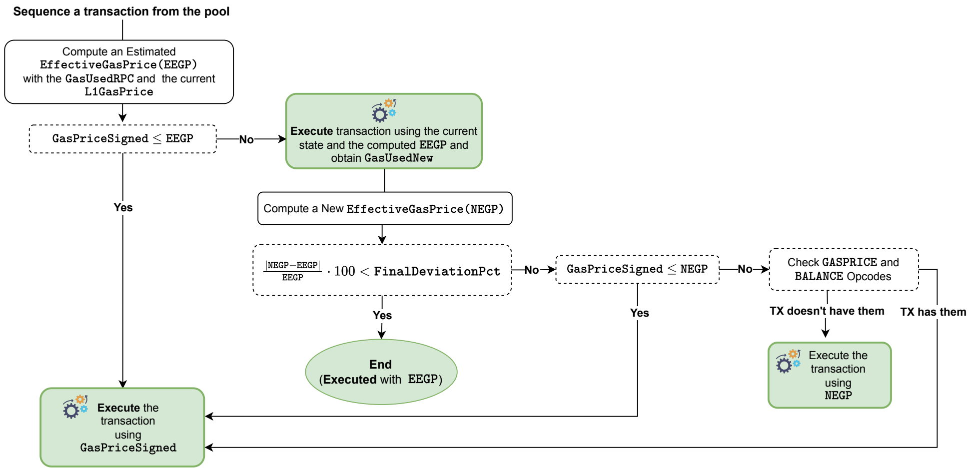
$$\texttt{GasPriceSigned} = 3 < 3.276 = \texttt{BreakEvenGasPrice} \cdot \texttt{BreakEvenFactor}.$$

6. At this moment, we should **reject transaction** but we are currently sponsoring them and we accept it as long as

$$\texttt{GasPriceSigned} = 3 \geq 2.85 = \texttt{GasPriceSuggested},$$

which is satisfied.

# gasPrice Flows: Sequencer i



## `gasPrice` Flows: Sequencer ii

1. The sequencer computes the estimated `EffectiveGasPrice` (which we will call `EEGP`) using the `GasUsedRPC` stored by the RPC and the current `L1GasPrice` for all the transactions of the `Pool` and sequence the one having higher `EEGP`.
2. At this point, we have two options:
   - If `GasPriceSigned` $\leq$ `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 `GasPriceSigned` $>$ `EEGP`, there is room for further adjustment of the user's gas price.
3. In this case, we execute the transaction using the **correct state root** and the computed `EEGP` to obtain the correct amount of gas used `GasUsedNew`.
4. Now we compute a new `EffectiveGasPrice` (which we will call `NEGP`) with the execution-related data computed from the current state.

5. 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):

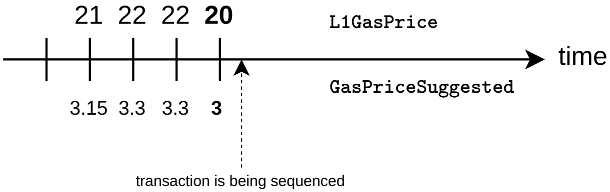
$$\frac{|\texttt{NEGP} - \texttt{EEGP}|}{\texttt{EEGP}} \cdot 100 < \texttt{FinalDeviationPct},$$

     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.
6. 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.

## `gasPrice` Flows: Sequencer iv

7. 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.
8. 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.
   This approach is employed to mitigate potential vulnerabilities in deployed Smart Contracts, that arise from creating a specific condition based on the gas price, for example, to manipulate execution costs.
9. If not, and with the intention of optimizing an execution while making a slight adjustment to the gas price, we proceed by executing the transaction using the `NEGP`.

# Numerical Example: Sequencer Flow i




- In our example, we end up computing `BreakEvenGasPrice` of 2.52 Gwei/Gas.
    1. Imagine that the user signed a gas price of 3.3 Gwei/Gas and, at the time of sequencing the transaction, the network suggests a gas price of 3 (corresponding to a L1 gas price of 20), leading to

$$\texttt{EEGP} = 2.772 \text{ Gwei/Gas.}$$

# Numerical Example: Sequencer Flow ii

2. Since $\texttt{GasPriceSigned} = 3.3 > 2.772 = \texttt{EEGP}$, we execute the transaction using the current (and correct) state and the computed $\texttt{EEGP}$ in order to obtain $\texttt{GasUsedNew}$, which in this case we suppose is
$$\texttt{GasUsedNew} = 95,000 \text{ Gas}.$$
3. Using $\texttt{GasUsedNew}$, we can compute an adjusted $\texttt{NEGP}$:
$$\texttt{TxCostNew} = (200 \cdot 16 + 100 \cdot 4) \cdot 20 + 95,000 \cdot 20 \cdot 0.04 = 148,000 \text{ GWei}.$$
$$\texttt{BreakEvenGasPriceNew} = \frac{148,000}{95,000} \cdot 1.2 = 1.869 \text{ GWei/Gas}.$$
$$\texttt{NEGP} = 1.869 \cdot 1.1 = 2.056 \text{ GWei/Gas}.$$
4. Observe that there is a significative deviation between both effective gas prices:
$$\frac{|\texttt{NEGP} - \texttt{EEGP}|}{\texttt{EEGP}} \cdot 100 = 25.82 > 10.$$

# Numerical Example: Sequencer Flow iii

5. Observe that this deviation penalizes the user a lot, since

$$\texttt{GasPriceSigned} = 3.3 \gg 2.056 = \texttt{NEGP},$$

   so we try to further adjust the charged gas price.
6. Suppose that the transaction does not have neither `GASPRICE` nor `BALANCE` opcodes.
7. In this case, we execute the transaction with $\texttt{GasPriceFinal} = \texttt{NEGP} = 2.056$ GWei/Gas.
8. Observe that `GasUsedFinal` should be $\texttt{GasUsedNew} = 95,000$ Gas.
9. We can compute now `EffectivePercentage` and `EffectivePercentageByte` as follows:

$$\texttt{EffectivePercentage} = \frac{\texttt{GasPriceFinal}}{\texttt{GasPriceSigned}} = \frac{2.056}{3.3} = 0.623.$$

$$\texttt{EffectivePercentageByte} = \texttt{EffectivePercentage} \cdot 256 - 1 = 148.$$

   Observe that the user has been charged with the 62.3% of the gas price he/she signed at the time of sending the transaction.

# Pool Effective Gas Price

```
...
    [Pool.EffectiveGasPrice]
    Enabled = false
    L1GasPriceFactor = 0.04
    ByteGasCost = 16
    ZeroByteGasCost = 4
    NetProfit = 1.2
    BreakEvenFactor = 1.3
    FinalDeviationPct = 10
    L2GasPriceSuggesterFactor = 0.3
```

https://github.com/0xPolygonHermez/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)