# 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: 3b678525fe19a5156a45912a7b06ea20bef3b474

March 4, 2024

# Outline

# 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:

a) `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`.

b) `GasLimit`:
- It is the maximum amount of gas units that a user enables to be consumed by the transaction.

# Basic Ethereum Fee Schema ii

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

$$\texttt{GasLimit}\ (\textit{gas}) \cdot \texttt{GasPrice}\ (\textit{Gwei}/\textit{gas}).$$

- Then,
  - If $\texttt{GasUsed} > \texttt{GasLimit}$, the transaction is reverted and all the subtracted amount is consumed from the user account.
  - 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:

  `L2GasPrice` = `L1GasPrice` · `L1GasPriceFactor`.

- For example:

  `L1GasPrice` = 21 Gwei/gas
  `L1GasPriceFactor` = 0.04 (4% of L1 `GasPrice`)
  `L2GasPrice` = 21 Gwei/gas · 0.04 = 0.84 Gwei/gas

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

# L2 User Fee Strategies are More Complex

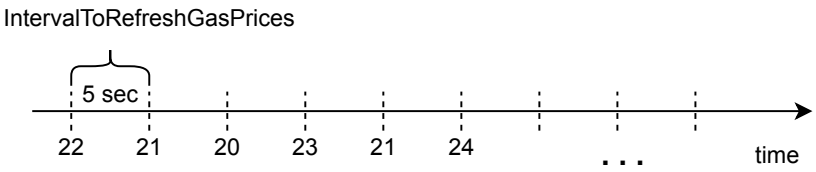
However, this is not as easy as it may seem and there are additional aspects to consider:

a) How is taken into account the fact that the gas price in L1 varies with time?

b) How is it managed the fact that the L1 gas schema may not be aligned with the actual resources spent by the L2 solution?

c) In L1, different gas price values can be used to prioritize transactions but how are these priorities managed by the L2 solution?

# Obtaining L1 GasPrices



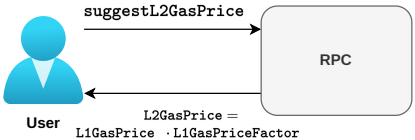

- As shown, gas prices vary with time.
- In the example, we poll for the L1 gas price every 5 seconds.

# Gas Price Suggester: Naive Approach

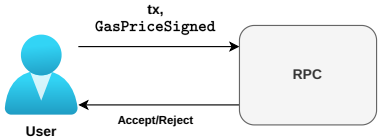
The system operates in two steps: **GasPrice suggestion** and **transaction sending**.

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

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



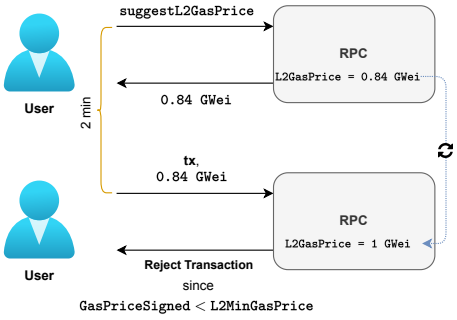

- Second, the user sends the desired L2 transaction with a choice of gas price that we will denoted as `GasPriceSigned`.




- In principle, the user should sign the transaction using the suggested gas price, but actually, users are free to sign any price.
- The node (RPC) will accept the `GasPriceSigned` if it is bigger than the current `L2GasPrice` suggested.

# Naive Approach: Bad User Experience

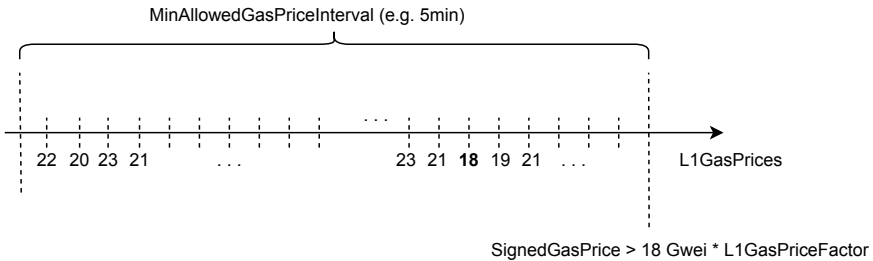
- However, notice that there is a time gap between asking for a suggested gas price and sending the transaction in which the gas price can fluctuate.
- Henceforth, the `L2GasPrice` from at asking for the suggestion can be different from the suggested one at the time of sending the transaction, leading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




- Observe that the `L2GasPrice` has been refreshed, in particular increased, and thus, the transaction sent by the prover will be rejected (error `ErrGasPrice`) by the node even tough it was signed with the exact suggested `L2GasPrice`.

# Gas Price Suggester: 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"
IntervalToRefreshGasPrices = "5s"
PollMinAllowedGasPriceInterval = "15s"
...
```

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.
- `IntervalToRefreshGasPrices:` It is the interval to refresh L2 gas prices.

There is another parameter related with an implementation detail of the node:

- `PollMinAllowedGasPriceInterval:` It is the interval to poll the database and get the `MinAllowedGasPrice` to build a cache for the different threads.

# Multi Gas Provider

- When computing the L1 gas price, we can activate the `MultiGasProvider`.
- When enabled, it allows using multiples sources for obtaining the L1 gas price.
- By default, it is disabled:

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

# L1/L2 Operation Cost Alignment Issue

- In Ethereum L1, users are used to prioritize their transactions by increasing the gas price.
- In the zkEVM:
  - There can be operations that consume low gas in Ethereum (L1) but that represent a major cost for the zkEVM proving system (L2).
  - An (old) approach used in the zkEVM was to use the amount of resources used by the proving system for a transaction as a way to prioritize some transactions in front of others (this algorithm was called "efficiency").
  - However, the efficiency mechanism was not clear for users who saw that transactions with less gas price where preferred by the system.
  - This approach was discarded and currently, following the Ethereum paradigm, **we only use the gas and the gas price to accept and prioritize transactions**.

# Data Availability Issue

- The Ethereum transaction gas scheme takes into account:
  - **Data availability**: the transaction bytes (size).
  - **Execution**: CPU, memory, storage and logs.
- The data availability cost:
  - Are fixed for a transaction of a certain size and they are paid at L1 gas price by L2's.
- The execution cost:
  - It is variable because executions depend on the network state.
  - It is what the L2 can optimize regarding the execution cost at L1 (but data availability has the same cost as in L1 in a rollup).
- Henceforth, L2 transactions having a **high data availability cost** and a **small execution cost** are **problematic** for L2's because there is little room for optimization and for making profit.
- Next, we show a numerical example.

# Data Availability Issue: Numerical Example

- **Non-zero bytes** cost 16 gas meanwhile **zero bytes** cost 4 gas.
- Assume that:
  - `L1GasPrice` = 21 Gwei/gas
  - `L1GasPriceFactor` = 0.04 (4% of L1 `GasPrice`)
  - `L2GasPrice` = 21 Gwei/gas $\cdot$ 0.04 = 0.84 Gwei
- Suppose the user sends a transaction having:
  - 200 non-zero bytes, including the constant ones.
  - 100 zero bytes.
- Then:
  - The data availability cost is $200 \cdot 16 + 100 \cdot 4 = 3600$ Gas.
  - Consider that the execution cost is 60000 Gas.
  - The L2 layer will have to pay $3600 * 21 = 75600$ GWei plus the proving cost.
  - The L2 layer will charge the user $63600 * 0.84 = 53424$ Gwei.
  - So, the L2 layer is loosing money with this transaction.

# Strategies for Fixing the Data Availability Issue i

Recall that the Ethereum fee is computed as `GasUsed` · `GasPrice`, giving us two ways of solving the data availability problem:

**1) Arbitrum Approach:** Increase the cost (`GasUsed`) for data availability.

- 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**.
- The EVM has the `GAS Opcode` that provides the remaining gas after executing the instruction so, an L1 Ethereum transaction may execute different when compared to the same transaction executed in L2.

# Strategies for Fixing the Data Availability Issue ii

**2) Effective Gas Price Approach:** Increase the `GasPrice` signed by the user and after processing the transaction, adjust the fee according to the final cost.

- 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.

# L1GasPriceFactor vs. SuggesterFactor

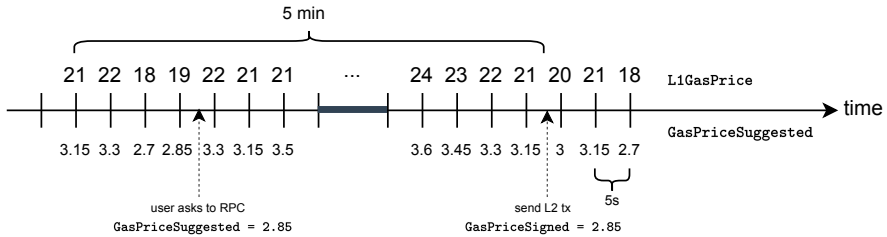
- The user signs a relatively high gas price at the time of sending the L2 transaction.
- Later on, the **sequencer** establishes a fair gas price according to the amount of resources used by the transaction.
- As a result, we have two different factors:
  a) `L1GasPriceFactor`, which represents the portion of the fee that the L2 wants to earn for processing the transaction.
  b) `SuggesterFactor`, which is used to suggest the L2 `GasPrice`: $\texttt{GasPriceSuggested} = \texttt{L1GasPrice} \cdot \texttt{SuggestedFactor}$.
- We have $\texttt{SuggesterFactor} >> \texttt{L1GasPriceFactor}$.

# L1GasPriceFactor vs. SuggesterFactor ii

- Example configuration:
  - L1GasPriceFactor $= 0.04$.
  - SuggesterFactor $= 0.15 \approx 4 \cdot$ L1GasPriceFactor.
- The configuration in the node of the SuggesterFactor can be found in the section about the L2GasPriceSuggester:

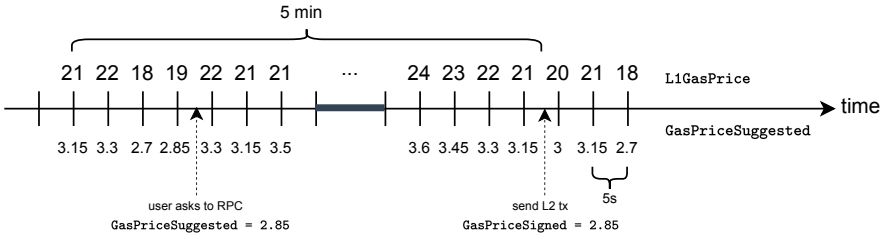
```
[L2GasPriceSuggester]
Type = "follower"
UpdatePeriod = "3s"
Factor = 0.15
DefaultGasPriceWei = 0
MaxGasPriceWei = 0
CleanHistoryPeriod = "1h"
CleanHistoryTimeRetention = "5m"
```

# 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 $2.85 = 0.15 \cdot 19$, which is the current L1 gas price (recall that the price is 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 equal or higher than the minimum suggested gas price from 5 minutes ago, which in this example is $18 \cdot 0.15 = 2.7$.
- In order to get his transaction accepted, the user sets the gas price of the transaction to $\texttt{GasPriceSigned} = 2.85 \geq 2.7 = \texttt{L2MinGasPrice}$.

# Effective Percentage i

- For each transaction, the **sequencer** provides a byte called $\texttt{EffectivePercentageByte} \in \{0, 1, \ldots, 255\}$, which will be used to provide a ratio called $\texttt{EffectivePercentage}$

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

- Observe that, in this schema, users **must trust the sequencer**.
- The **effectivePercentage** will be used in order to compute the factor of the signed transaction's $\texttt{GasPrice}$ which should be charged to the user:

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

# Effective Percentage 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:

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

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

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

- As having `EffectivePercentage` implies having `EffectivePercentageByte`, and vice versa, we will abuse of notation and use them interchangeably as `EffectivePercentage`.

# The `BreakEvenGasPrice`

- The next question is how to compute the `EffectivePercentage`.
- Our goal, as service providers, is to **not accept transactions in which we loose money** or at least, try to minimize this situation.
- 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 on one hand the costs associated with data availability and on the other hand the costs associated with transaction processing.

# **BreakEvenGasPrice**: Data Availability Cost

- The cost associated with data availability will be denoted as **DataCost** and it is measured in gas.
- Recall that in Ethereum:

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

- Additionally, 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:

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

## **BreakEvenGasPrice**: Execution Cost

- The cost associated with transaction execution will be denoted as `ExecutionCost` and it is measured in gas.
- Observe that, unlike data costs, to obtain the execution cost, we will need to **execute** the transaction.
- Then, $\texttt{GasUsed} = \texttt{DataCost} + \texttt{ExecutionCost}$.
- The total fee that the L2 will receive is $\texttt{GasUsed} \cdot \texttt{L2GasPrice}$.
- Where recall that $\texttt{L2GasPrice} = \texttt{L1GasPrice} \cdot \texttt{L1GasPriceFactor}$.

# **BreakEvenGasPrice** Formula

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

$$\texttt{TotalTxPrice} = \texttt{DataCost} \cdot \texttt{L1GasPrice} + \texttt{GasUsed} \cdot \texttt{L2GasPrice}.$$

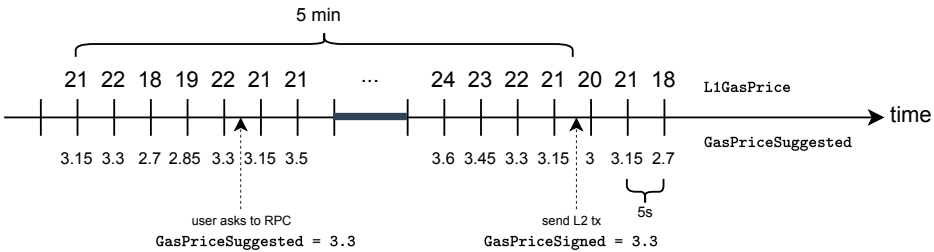
- We can compute `BreakEvenGasPrice` as the following ratio:

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

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

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

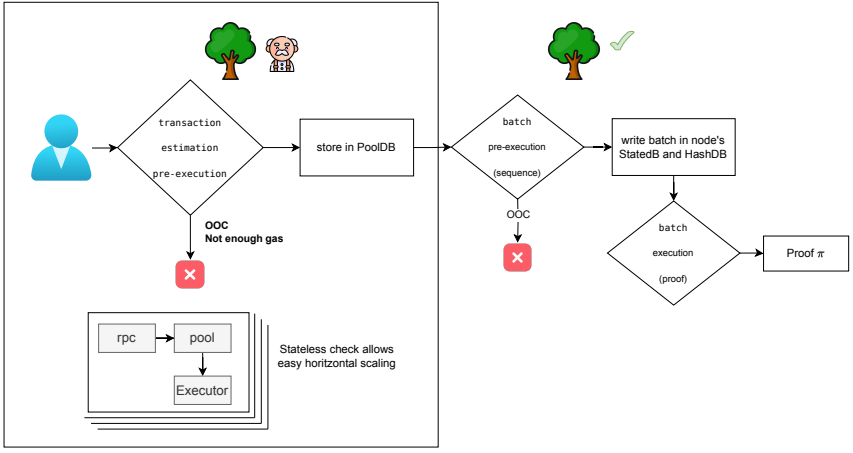
# Numerical Example: **BreakEvenGasPrice** i




Let's consider that the user sends a transaction with **GasPriceSigned** $= 3.3$.

The first step at the node (RPC) is do a pre-execution of the transaction to obtain an estimation of the **ExecutionCost**.

# Numerical Example: **BreakEvenGasPrice** ii



# Numerical Example: `BreakEvenGasPrice` iii

- Suppose the user sends a transaction having:
  - 200 non-zero bytes, including the constant ones.
  - 100 zero bytes.
- Moreover, let's consider that the pre-execution of the transaction yields $60,000$ of `GasUsed` (availability+execution) and that we do not get an **OOC** error.
- Recall that, since we are not using the actual state root, so this gas is only an estimation.
- Hence, the total transaction fee charged to the user should be:

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

- Observe that 21 is the `L1GasPrice` at the time the node receives the transaction.

# Numerical Example: **BreakEvenGasPrice** iv

- 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 uncertainties since the execution cost is estimated:

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

- 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, `GasUsedRPC` $= 60$k.
- However, assume that the 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 (since there is less execution cost and thus, data availability represents a bigger portion in the required fee).
- That means that we should have charged the user with a higher gas price to cover the whole transaction cost at 3.6 GWei/gas, which now is of $105,000$ GWei.
- But, since we were accepting all the transactions signing more than 2.85 GWei/gas at the time the transaction was sent, we do not have margin to increase the gas price and this transaction will represent a loss.

# Numerical Example: `BreakEvenFactor` ii

- In the worst case we are loosing

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

- Introducing `BreakEvenFactor` = 1.3, 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

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

# Numerical Example: **BreakEvenFactor** iii

Note:

- In the example, we assumed that the increase in `BreakEvenGasPrice` is a result of have less `GasUsed` after executing the transaction with the correct state root.
- However, the `BreakEvenGasPrice` increase can also be due to an increase of the `L1GasPrice`.

# Introducing the Priority

Prioritization of transactions in Ethereum:

- It is done when building the block.
- It is determined by the signed gas price.
- The list of pending transactions are ordered by their priority (signed gas price).

In the zkEVM:

- We prioritize transactions at the sequencing time.
- We follow exactly the same principle as Ethereum:
  We order the list of transactions waiting to be sequenced at the pool by their signed gas price.

https://github.com/0xPolygonHermez/zkevm-node/blob/b7c17983d56b3404d65da2f14db980190c190034/sequencer/txsorted_list.go#L11

# **PriorityRatio** and **EffectiveGasPrice**

- Users have to pay for prioritizing their transactions, in other words, the priority has an impact over the **EffectiveGasPrice** charged to the user.
- We define the **PriorityRatio** as follows:
  - If $\texttt{GasPriceSigned} \leq \texttt{GasPriceSuggested}$, then,

$$\texttt{PriorityRatio} = 1$$

  - If $\texttt{GasPriceSigned} > \texttt{GasPriceSuggested}$, then,

$$\texttt{PriorityRatio} = \frac{\texttt{GasPriceSigned}}{\texttt{GasPriceSuggested}}$$

- The **EffectiveGasPrice** takes into account the priority and it is computed as:

$$\texttt{EffectiveGasPrice} = \texttt{BreakEvenGasPrice} \cdot \texttt{PriorityRatio}$$

https://github.com/0xPolygonHermez/zkevm-node/blob/b7c17983d56b3404d65da2f14db980190c190034/pool/effectivegasprice.go#L112

# Numerical Example: **PriorityRatio** and **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

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

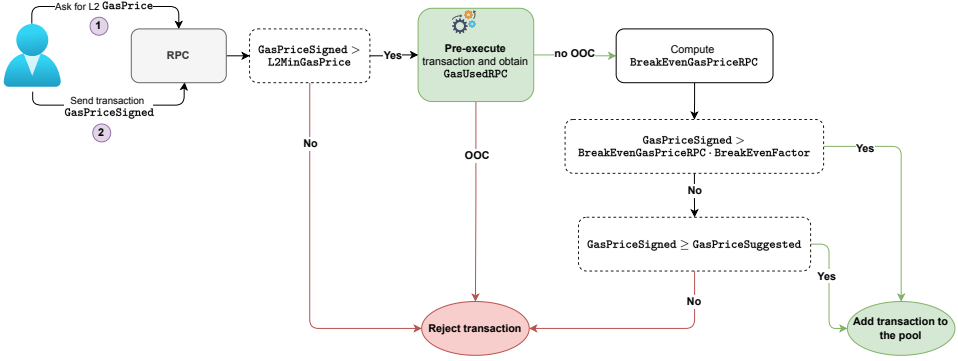
- The difference between both is taken into account in the priority ratio:

$$\texttt{PriorityRatio} = \frac{3.3}{3} = 1.1$$

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

$$\texttt{EffectiveGasPrice} = 2.52 \cdot (1.1) = 2.772.$$

# RPC Flow i


Ask for L2 GasPrice
1
RPC
Send transaction
GasPriceSigned
2
GasPriceSigned >
L2MinGasPrice
Yes
No
Pre-execute
transaction and obtain
GasUsedRPC
OOC
no OOC
Compute
BreakEvenGasPriceRPC
GasPriceSigned >
BreakEvenGasPriceRPC · BreakEvenFactor
Yes
No
GasPriceSigned ≥ GasPriceSuggested
Yes
No
Reject transaction
Add transaction to
the pool

# RPC Flow ii

1. The user asks to the RPC for **GasPriceSuggested**.
2. The user sends the transaction with a **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 the current state root that will differ from the one finally used when sequencing) to obtain **GasUsedRPC**, which is used in order to compute the **BreakEvenGasPriceRPC** at this moment.
5. We have two cases:
   a) If the transaction pre-execution runs out of counters (**OCC** error), we immediately reject the transaction.
   b) If not, the RPC computes the **BreakEvenGasPriceRPC** and we continue the flow.

# RPC Flow iii

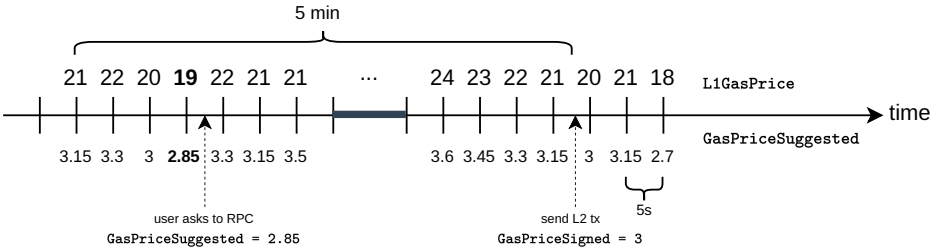
6. Now, we have two options:
   a) If `GasPriceSigned` $>$ `BreakEvenGasPriceRPC` $\cdot$ `BreakEvenFactor`, we immediately accept the transaction, storing it in the Pool.
   b) Otherwise `GasPriceSigned` $\leq$ `BreakEvenGasPriceRPC` $\cdot$ `BreakEvenFactor`, we are in dangerous zone because we may be facing losings due high data availability costs or to fluctuation between future computations.
7. In the dangerous path, we allow two options:
   a) If `GasPriceSigned` $\geq$ `GasPriceSuggested`, we take the risk of possible losses, sponsoring the difference if necessary and we introduce the transaction into the Pool.
   b) Otherwise `GasPriceSigned` $<$ `GasPriceSuggested`, we immediately reject the transaction because its **highly probable that we face losings**.

# Some Considerations about the RPC Flow

- 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 compensate 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 `SignedGasPrice`.

# Numerical Example: RPC Flow i

Following our previous example:




1. The user asks to the RPC and gets $\texttt{GasPriceSuggested} = 0.15 \cdot 19 = 2.85$.
2. Assume that the user sends a transaction signed with $\texttt{GasPriceSigned} = 3$ to prioritize a little his transaction (since $\texttt{GasPriceSigned} > \texttt{GasPriceSuggested}$).

# Numerical Example: RPC Flow ii

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

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

5. In this case, with $\texttt{BreakEvenFactor} = 1.3$:

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

6. At this moment, the transaction is not profitable but we sponsor it and we accept it as long as

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

which is satisfied.

# Design of the **EffectivePercentage** Adjustment

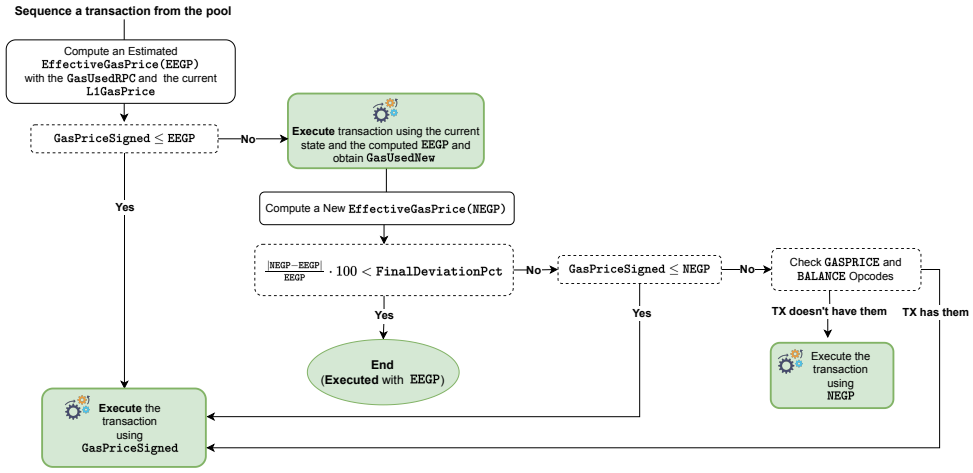
The sequencer is who finally computes the effective percentage.

Regarding the effective percentage adjustment process we have to take into account that:

- In general, the transaction execution cost depends on:
  a) **The state when the transaction is finally executed**, i.e., when the transaction is definitively ordered (in our case, when the transaction is sequenced).
  b) **The gas price used for the execution**. In particular, there are two opcodes that are directly affected by the gas price: the `GASPRICE` opcode and the `BALANCE` opcode when applied to the transaction source address.
- Also, we need to take into account that each transaction execution consumes resources of the L2 provider.

So, the effective percentage is **determined through a series of iterative refinements aimed at achieving a fair gas price for both the user and the L2 provider with a minimum number of transaction executions**.

# Sequencer Flow i



# Sequencer Flow 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, sequences the one having higher `EEGP`.
2. At this point, we have two options:
   a) 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.
   b) 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.

# Sequencer Flow iii

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):

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

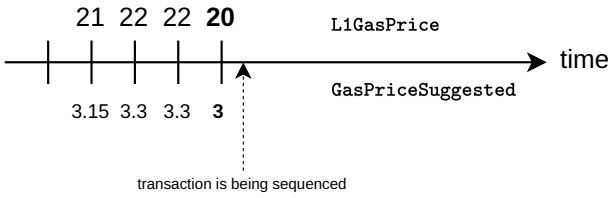
     we end up the flow just to avoid re-executions and save execution resources.
   - On the contrary, if the difference equals or exceeds the deviation parameter, there is a big difference between executions and we may better adjust gas price.
6. In the later case, two options arise:
   - If $\mathbf{GasPriceSigned} \leq \mathbf{NEGP}$ 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 $\mathbf{GasPriceSigned} > \mathbf{NEGP}$, 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.

# Sequencer Flow iv

7. We check if the previous transaction processing includes the two opcodes that use the gas price:
   - The `GASPRICE` opcode.
   - The `BALANCE` opcode from the source address.
8. If the transaction uses these opcodes, we execute the transaction using the full `GasPriceSigned` to ensure we minimize potential losses and we end up the flow.

   This approach is employed to mitigate potential vulnerabilities by contract executions that can create a condition based on the gas price to manipulate execution costs.
9. If the transaction does not use these opcodes, we do the final adjustment 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 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 `GasPriceFinal` = `NEGP` = 2.056 GWei/Gas.
8. Observe that `GasUsedFinal` should be `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

- `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.

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

- `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).

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