polygon zkEVM

Knowledge Hub

Architecture

Economics - Users Fees

v.1.0

January 25, 2024

1 Basic Ethereum Fee Schema

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:

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

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:

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

2 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`:

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

For example,

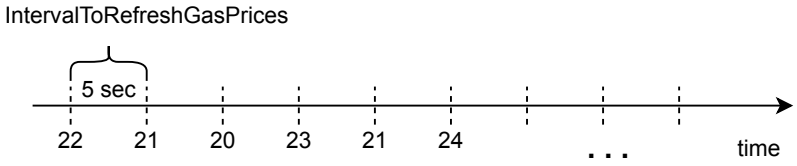
$$\texttt{L1GasPrice} = 20 \text{ Gwei and } \texttt{L1GasPriceFactor} = 0.04 \ (4\% \text{ of L1 } \texttt{gasPrice})$$
$$\text{therefore, } \texttt{L2GasPrice} = 20 \text{ Gwei} \cdot 0.04 = 0.8 \text{ Gwei}$$

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

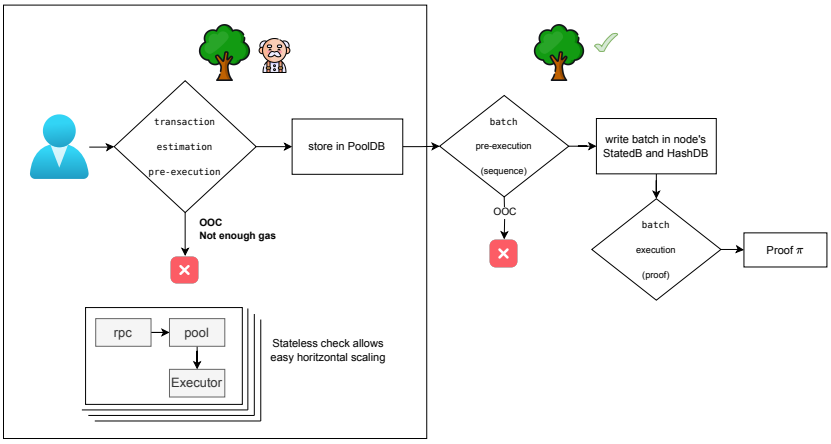
c) The `gas/gasPrice` L1 schema may not be aligned with the actual resources spent by the L2 solution.



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

In the following sections, we will thoroughly examine the significance of fees in Layer 2 and provide detailed answers to the previously mentioned questions.

We will focus first on the first part of the flow: **the RPC transaction pre-execution.**



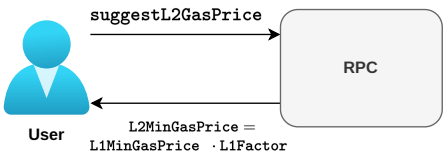
3 Sending a Transaction

This process is divided in two steps: `gasPrice` suggestion and sending the transaction.

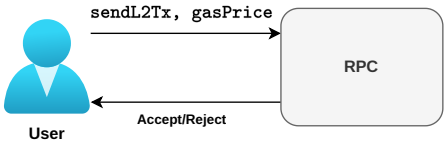
1. `gasPrice` suggestion: the user asks via a RPC call for a suggested `gasPrice`. It is computed the following way:

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

   to get an idea of a price to sign the transaction.

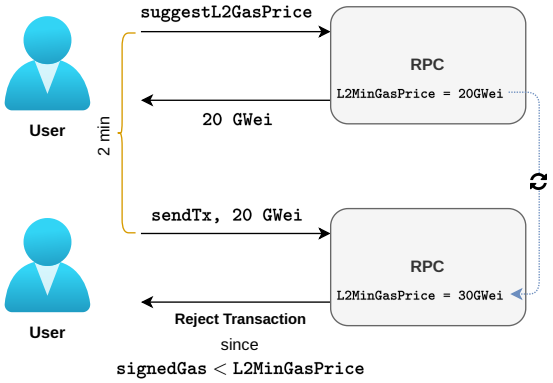


2. Transaction sending: the user sends the desired L2 transaction with the decided `gasPrice`.



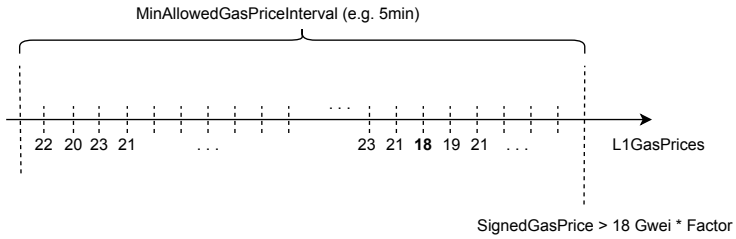
In case the `gasPrice` decided by the user is less than the current `L2GasPrice`, the transaction is automatically rejected and not included into the pool. This error is named `ErrGasPrice`.

The previous framework has a limitation as there is a time gap between step one and step two during which the gas price can fluctuate. This leads to the following unwanted situation:



In the previous figure we can observe that although the user has sent a `gasPrice` according to the suggested price, after the time lapse between one step and the other, the price has increased and the transaction is rejected.

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



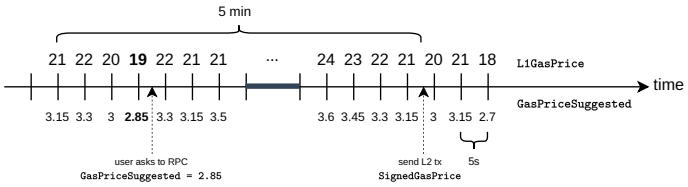
This previous parameters can be configured in the Polygon zkEVM node:

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 $0.15 \approx 4 \cdot$ `L1GasPriceFactor`:

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

Let's see a numerical example of how is the `GasPriceSuggested` computed:



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. 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 $\texttt{GasPriceSigned} = 3.3 > 2.85 = \texttt{L2MinGasPrice}$. The user has signed at a higher `gasPrice` than the suggested to ensure that the transaction is executed.

4 Cost issues and strategies L1/L2

In Ethereum, gas accounts for the resources used by a transaction, considering two elements in particular: **Data availability** (transaction bytes) and **Processing resources** like CPU, memory, and storage. Ethereum users commonly prioritize their transactions by increasing the `gasPrice`. A notable challenge arises when certain operations consume low gas in Layer 1 but represent a significant cost in Layer 2.

The costs associated with data availability are fixed once the transaction is known, and they are directly proportional to L1 data availability costs. On the other hand, L2 execution costs are variable, depending on the state, and typically offer a smaller cost per gas. Consequently, in our pricing schema, L2 transactions with high data availability costs and small execution costs are a significant challenge.

Recall that the Ethereum fee is computed as $\texttt{gasUsed} \cdot \texttt{gasPrice}$, giving us two ways of solving the misalignment problem when certain operations consume low gas in L1 but represent a significant cost in L2:

(A) **Arbitrum Approach. Increase `gasUsed`.** This approach involves modifying the gas schema to elevate the Gas costs associated with data availability. While this strategy is relatively straightforward to implement and comprehend, it comes with a notable implication: **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 aim to avoid modifying the gas, the alternative is to increase the `gasPrice` to cover the costs. Unlike the previous approach, this doesn't alter the Ethereum specifications. However, determining a fair `gasPrice` becomes a complex task. 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.

We will now develop how the **Effective Gas Price Approach** works.

First, the user signs a relatively high gas price (`GasPriceSigned`) 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 $\texttt{EffectivePercentageByte} \in \{0, 1, \dots, 255\}$ (1 Byte), which will be used to compute a ratio called `effectivePercentage`:

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

This percentage will be used in order to compute the factor of the signed transaction's `gasPrice` which should be charged to the user

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

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:

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

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

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

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`.

To calculate the `EffectivePercentage`, one option is to consider the pricing resources based on the number of **consumed counters** within our proving system. However, understanding this metric can be challenging for users because stating the efficiency trough counters is not intuitive at the time of prioritizing their transactions. As we want to prioritize a positive user experience, we will consider an alternative where gas is used for calculations, as it is more user-friendly. So, our primary objective is to compute `EffectivePercentage` exclusively using Gas, allowing users to prioritize their transactions through the use of `gasPrice` without the need for intricate counter-based considerations.

5 Introducing `BreakEvenGasPrice`

As service providers, our primary goal is to **avoid accepting transactions that result in financial losses**. To attain this objective, 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.

5.1 Costs Associated with Data Availability

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

Also recall that, when computing non-zero bytes cost we should take into account some constant data always appearing in a transaction: the **signature**, consisting on 65 bytes and the **EffectivePercentageBytesLength**, consisting on 1 byte 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:

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

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

5.2 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.

In contrast to data availability costs, to compute computational costs we will need to **execute** the transaction.

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

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

To establish the gas price at which the total transaction cost is covered we can compute `BreakEvenGasPrice` as the following ratio:

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

Additionally, we incorporate a factor $\texttt{NetProfit}\geq 1$ that allows us to achieve a slight profit margin:

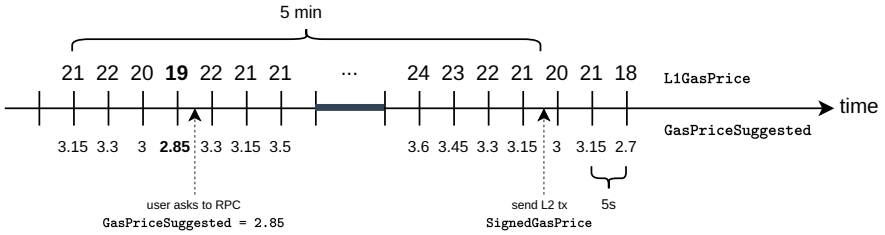
$$\texttt{BreakEvenGasPrice}=\frac{\texttt{TotalTxPrice}}{\texttt{GasUsed}}\cdot\texttt{NetProfit}.$$

Observe that we still need to introduce here `gasPrice` prioritization, which will be covered later on.

5.3 Numerical Examples

1. About how to calculate the `BreakEvenGasPrice`:

   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 and 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).



Hence, following the formulas previously explained, 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

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

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. To avoid this issue, 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**.

2. About how to calculate the `BreakEvenFactor`:

   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. 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`.

6 Introducing Priority

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

At the time of sequencing a transaction, we should prioritize ones among the others, depending basically on both `GasPriceSigned` and current `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 `EffectiveGasPrice` will be computed as:

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

Let's see it with an example:

Recall that, in the previous 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 3.

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

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

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

7 User Fees Flows

In this section, we will delve into the economic process that starts when a user wants to send some transaction to the L2 network. Our discussion will be split into two parts: firstly, the transaction flow done by the **RPC** until it is dispatched by the user, concluding when it is stored in the **Pool**. Secondly, we will examine the sequence transaction flow managed by the **Sequencer**, ending in some execution.

7.1 RPC Flow

The **RPC** is the zkEVM component that, in a high level, handles the acceptance or rejection of incoming transactions and saves the approved transactions in the **Pool**. In Figure 1, the transaction progression within the **RPC** component is shown, starting from the moment a user is willing to send a transaction to the network to the point where it becomes either stored in the **Pool** or rejected. Let's examine the Figure 1 in detail.

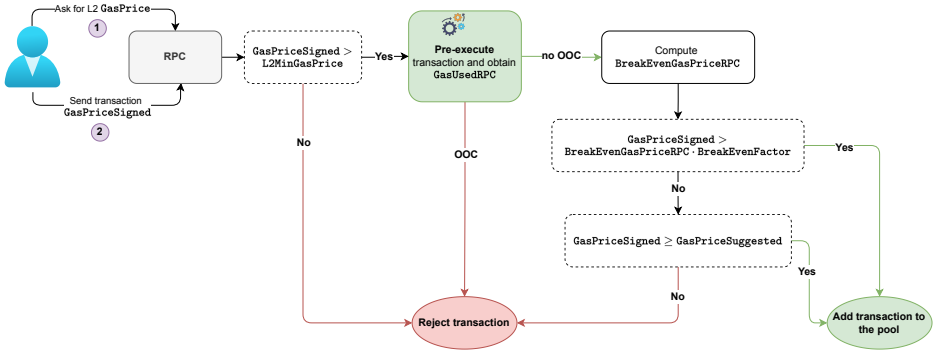


Figure 1: Flow of a transaction within the **RPC** component. As a result of this flow, a transaction can be either included in the **Pool** for future sequencing or rejected from the network.

1. First of all, the users asks to the **RPC** for the current `GasPriceSuggested`, which recall is a factor of the current L1 `GasPrice`. More concretely,

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

   where `SuggestedFactor` (which is currently of 0.15) satisfies

$$\texttt{SuggestedFactor} > \texttt{L1GasPriceFactor}$$

   in order to be able to cover data availability costs. Observe that the suggested gas price varies over time as `L1GasPrice` also does.

2. Now, the user sends the transaction selecting a `GasPriceSigned`, the same as in L1. Observe that, from asking for a suggested gas price to sending the transaction has passed some unbounded period of time, where the L1 Gas Price could have increased. Henceforth, rejecting a transaction if $\texttt{GasPriceSigned} < \texttt{GasPriceSuggested}$ does not offer a good user experience. In contrast, we give a margin of 5 minutes (controlled by the `MinAllowedGasPriceInterval` parameter). If the signed gas price does not strictly exceed `MinL2GasPrice`, which is the minumum suggested gas price of 5 minutes before sending the transaction refreshed every 5 seconds, called (controlled by the `IntervalToRefreshGasPrices` parameter),

$$\texttt{GasPriceSigned} > \texttt{L2MinGasPrice},$$

   we automatically **reject** the transaction, since it will not be possible to cover costs.

3. If the transaction was not rejected in the previous step, the RPC uses a cloud executor in order to pre-execute the transaction. Observe that this pre-execution is only an estimated execution since the used state root is not the correct one, as the transaction has not been sequenced yet. Recall that once a transaction is added into the **Pool**, it is mandatory that it is eventually sequenced. Due to this fact, the purpose of this is to filter transactions according and estimate a fair gas price in an early processing stage, to avoid future losses. As a result of this pre-execution it is obtained and estimated amount of used Gas, which will be called `GasUsedRPC`. If by pre-executing the transaction we run out of counters, we immediately **reject** the transaction.

4. If the transaction was not reverted due to **OOC error**, we compute the current break even gas price, which we will call `BreakEvenGasPriceRPC`. Recall that it depends on the current `L1GasPrice`, the transaction size, the `GasUsed RPC` and a parameter `NetProfit` that is present in order to include the network's profit for the whole transaction's processing.

5. Now, we have two different paths:

   - If the gas price signed by the user at the time of sending the transaction is higher than the `BreakEvenGasPriceRPC`, increased by a factor `BreakEvenFactor` ≥ 1 (and currently set at 1.3) used to protect the network against bad Gas usage estimations in the RPC

     $$\texttt{GasPriceSigned} > \texttt{BreakEvenGasPriceRPC} \cdot \texttt{BreakEvenFactor}$$

     then the transaction is immediately **accepted** and stored in the pool.

   - Otherwise, if

     $$\texttt{GasPriceSigned} \leq \texttt{BreakEvenGasPriceRPC} \cdot \texttt{BreakEvenFactor}$$

     we are in dangerous zone because we may be facing losings due high data availability costs or to fluctuation between future computations, so we **should reject** the transaction. However, we are currently not directly rejecting transactions in this case.

6. In the later case, we check if the gas price signed with the transaction exceeds the current suggested gas price:

   $$\texttt{GasPriceSigned} \geq \texttt{GasPriceSuggested}.$$

   In this case, we take the risk of possible losses, sponsoring the difference if necessary and we introduce the transaction into the **Pool**. However, if $\texttt{GasPriceSigned} < \texttt{GasPriceSuggested}$ we assume that is highly probable that we face losing and we immediately reject the transaction.

Final Considerations It is important to remark that, as we have said before, once a transaction is included into the pool, we should actually sequence it, that is, we should include it into a block. Hence, if something goes bad in later steps and the gas consumption deviates significantly from the initial estimate, we risk incurring losses with no means to rectify the situation. On the contrary, if the process goes as estimated and the consumed gas is similar to the estimated one, we can reward the user by modifying the previously introduced `effectivePercentage`. Additionally, its important to observe that, among all the transactions stored in the Pool, the ones prioritized at the time of sequencing are the ones with higher `effectiveGasPrice`, due to the prioritization introduced with `ratioPriority`. Observe that `effectiveGasPrice` **is not** computed in the **RPC** but in the **Sequencer**, so it possible that the suggested gas price at this moment differs from the suggested one when user sent the transaction.

7.2 Numerical Example: RPC Flow

Let us continue the numerical example that has been carried during the whole document. In Figure 2 we represent, at the top of the timeline, the current `L1GasPrice` and, at the bottom, the associated $\texttt{GasPriceSuggested} = 0.15 \cdot \texttt{L1GasPrice}$.

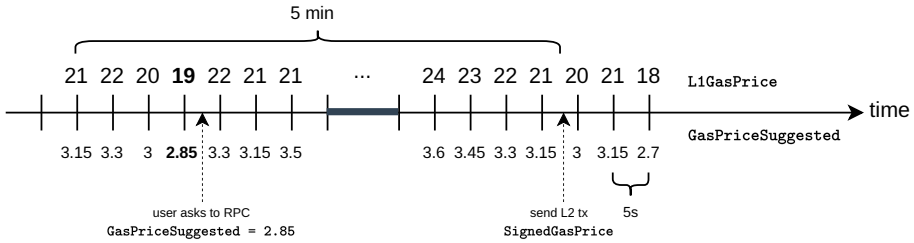


Figure 2: Timeline displaying the current `L1GasPrice` and its associated suggested gas price.

1. At the time marked with the left arrow, the user queries the RPC for the suggested gas price when `L1GasPrice` is 19, and receives a as response the value of 2.85 GWei/-Gas:
$$\texttt{GasPriceSuggested} = 0.15 \cdot 19 = 2.85 \text{ GWei/Gas}.$$

2. Let us suppose that the users sends a transaction signed with a gas price of 3
$$\texttt{GasPriceSigned} = 3.$$
Observe that the signed gas price is strictly lower than the current suggested gas price, which is $3.15 = 21 \cdot 0.15$. However, recall that at this precise step we are allowing all the transactions with a signed gas price exceeding the minimum suggested gas price 5 minutes before sending the transaction refreshed every 5 seconds. Henceforth, since
$$\texttt{GasPriceSigned} = 3 > 2.85 = \texttt{L2MinGasPrice},$$
we accept the transaction at this point.

3. At this point, the RPC asks for a pre-execution, getting an estimation for the gas used, computed with a state root that will differ from the one used when sequencing the transaction. In this case, we get a gas used estimation of
$$\texttt{GasUsedRPC} = 60,000 \text{ Gas},$$
without running out of counters.

4. Since we have not run out of counters, we compute `BreakEvenGasPriceRPC` supposing same transaction sizes as before and getting
$$\texttt{BreakEvenGasPriceRPC} = 2.52 \text{ GWei/Gas}.$$

5. Notice that, in this particular scenario, despite having
$$\texttt{GasPriceSigned} < \texttt{BreakEvenGasPriceRPC},$$
the introduction of the `BreakEvenFactor` (which acts as a protective measure as previously discussed) results in the next check failure:
$$\texttt{GasPriceSigned} < 3.276 = 2.52 \cdot 1.3 = \texttt{BreakEvenGasPrice} \cdot \texttt{BreakEvenFactor}.$$

6. However, recall that but we are currently sponsoring and accepting all transactions as long as
$$\texttt{GasPriceSigned} = 3 \geq 2.85 = \texttt{GasPriceSuggested},$$
which is the current case. Henceforth, we accept the transaction and store it into the **Pool**.

7.3 Sequencer Flow

The **Sequencer** is the zkEVM component that is responsible for fetching transactions from the **Pool** and assembling some of them into a **batch**. The sequencer submits a sequence of batches to the L1 which will be then proved by the **Aggregator**. In Figure 3, the transaction progression within the **Sequencer** component is shown, starting from the moment that a transaction is fetched from the **Pool** until it is executed by the **Executor**. Let's examine the Figure 3 in detail.

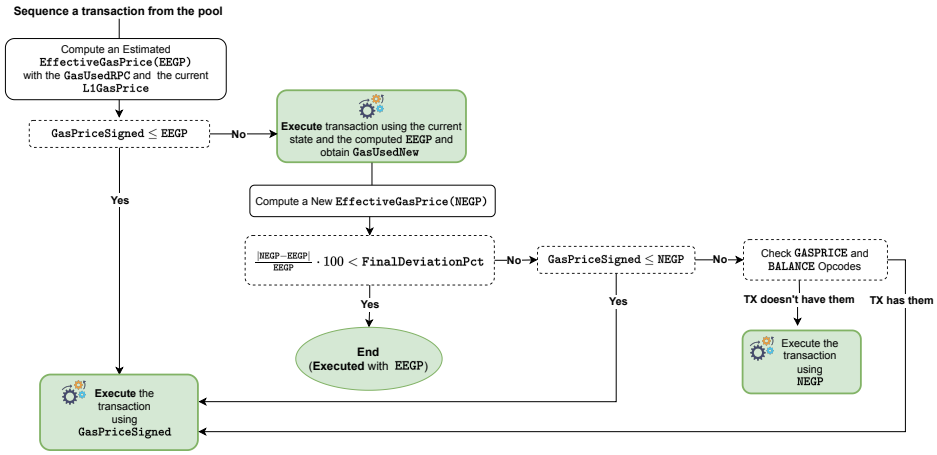


Figure 3: Flow of a transaction within the **Sequencer** component. Observe that we can not reject transactions at this point, so all of them should be included in the L1 State.

1. The **Sequencer** computes the estimated `EffectiveGasPrice` (which we will call `EEGP`) using the `GasUsedRPC` obtained in the **RPC** pre-execution using a previous state root (which has now changed) and the current `L1GasPrice` (which also may differ from the one used when sending the transaction to the **RPC**) for all the transactions stored in the **Pool** and sequence the one having higher `EEGP`. It is important to note that this is done in this precise order. We could have calculated the `EEGP` just before storing the transactions in the **Pool** and sorting it by `EEGP`, but this would not yield the same result because the `L1GasPrice` at that moment is different from the one at the time of sequencing a transaction, potentially changing the `EEGP` as well as the prioritization order of transactions.

2. At this point, we have two options:

   - If `GasPriceSigned` $\leq$ `EEGP`, even with only an estimated effective gas price, there is a significant risk of loss. In such cases, we opt not to further adjust the gas price in order to reduce the number of executions needed to do so. Henceforth, the user is charged the full `GasPriceSigned`, so the **Executor** will execute the transaction using it, concluding the sequencing process.
   - Conversely, if `GasPriceSigned` $>$ `EEGP`, there is a potential room for further adjustment for the gas price that will be charged to the user.

3. In the previous case, it is necessary to compute a more precise effective gas price based on the accurate amount of gas, denoted as `GasUsedNew`, obtained during the transaction's execution using the correct state root at the sequencing time (which was not known earlier for straightforward reasons). Henceforth, the **Executor** executes the transaction using `EEGP`, obtaining `GasUsedNew` which the sequencer utilizes to compute a new effective gas price referred to as `NEGP`.

4. We have to paths:

   - If the percentage deviation between `EEGP` and `NEGP` is higher than a fixed deviation parameter `FinalDeviationPct` (which is 10 in the actual configuration), that is
   $$\frac{|\texttt{NEGP} - \texttt{EEGP}|}{\texttt{EEGP}} \cdot 100 < \texttt{FinalDeviationParameter},$$
   indicates that there is minimal distinction between charging the user with `NEGP` compared to `EEGP`. Therefore, despite potential (but quite small) losses to the network or the user, we end up the flow just to avoid re-executions and save execution resources and we charge the user with `EEGP`.
   - On the contrary, if the percentage deviation equals or exceeds the deviation parameter
   $$\frac{|\texttt{NEGP} - \texttt{EEGP}|}{\texttt{EEGP}} \cdot 100 \geq \texttt{FinalDeviationParameter},$$
   there is a big difference between executions and we may better adjust gas price to potential (and quite big) losses to the network or the user.

5. In the later case, two options arise:

   - If the gas price signed is less or equal than the accurate effective gas price computed with the correct state root
   $$\texttt{GasPriceSigned} \leq \texttt{NEGP},$$
   the network suffers again a risk of loss. Henceforth, the user is charged the full `GasPriceSigned`, so the **Executor** will execute the transaction using it, concluding the sequencing process.
   - Otherwise, if
   $$\texttt{GasPriceSigned} > \texttt{NEGP},$$
   means that we have margin to adjust the gas price that is charged to the user. However, in order to save executions and end up the adjustment process in this iteration, we will conclude the flow using a trick explained in the next point.

6. In the later case, 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.

   We have two cases:

   - If the transaction contains the aforementioned opcodes, we impose a penalty on the user for security reasons. In such cases, we simply proceed with executing the transaction using the entire `GasPriceSigned` to minimize potential losses and conclude the flow, as mentioned earlier. This precaution 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.
   - If the transaction does not make use of the gas price-related opcodes, the **Executor** executes the transaction with the more adjusted gas price up to this point which is `NEGP` and end up the sequencing process.

7.4 Numerical Example: Sequencer Flow

Let us continue the numerical example that has been carried during the whole document. In Figure 2 we represent, at the top of the timeline, the current `L1GasPrice` and, at the bottom, the associated `GasPriceSuggested` $= 0.15 \cdot$`L1GasPrice` at the time of sequencing the transaction. Recall that in Section 7.2 we end up computing the `BreakEvenGasPriceRPC` of 2.52 GWei/Gas.

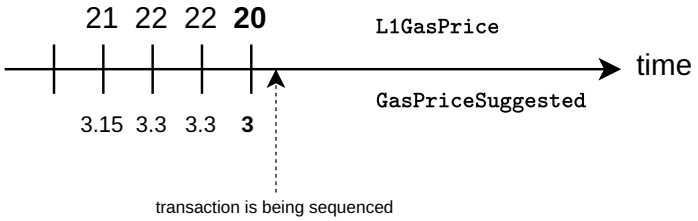


Figure 4: Timeline displaying the current `L1GasPrice` and its associated suggested gas price at the time of sequencing the transaction.

1. Imagine that the user signed a gas price 3.3 GWei/Gas. As illustrated in Figure 4, the network recommends a gas price of 3 at the sequencing moment, equivalent to an L1 gas price of 20. This results in an `EEGP` of

$$\texttt{EEGP} = 2.722 \text{ GWei/Gas},$$

   where the 10% increase attributed to the prioritization carried out by `PriorityRatio` set at 0.1.

2. Since the signed gas price is bigger than the estimated effective gas price

$$\texttt{GasPriceSigned} = 3.3 > 2.772 = \texttt{EEGP},$$

   we execute the transaction using the current (and correct) state and the computed `EEGP` in order to obtain an accurate measure of the Gas used, which we call `GasUsedNew`. Suppose that, in this case, we obtain

$$\texttt{GasUsedNew} = 95,000 \text{ Gas},$$

   which is bigger than the estimated gas of $60,000$ at the **RPC** pre-execution.

3. By using `GasUsedNew`, we can compute and adjusted effective gas price called `NEGP` as follows:

$$\texttt{TxCostNew} = (200 \cdot 16 + 100 \cdot 4) \cdot 20 + 95,000 \cdot 20 \cdot 0.04 = 148,000 \text{ GWei},$$
$$\texttt{BreakEvenGasPriceNew} = \frac{148,000}{95,000} \cdot 1.2 = 1.869 \text{ GWei/Gas},$$
$$\texttt{NEGP} = 1.869 \cdot 1.1 = 2.056 \text{ GWei/Gas}.$$

   Observe that the transaction cost is way higher than the estimated one of $126,000$ even when the L1 Gas Price has decreased from 21 to 20 due to a huge increase in Gas.

4. Observe that there is a significative deviation between both effective gas prices:

$$\frac{|\texttt{NEGP} - \texttt{EEGP}|}{\texttt{EEGP}} \cdot 100 = 25.82 > 10.$$

This deviation penalizes the user a lot since

$$\texttt{GasPriceSigned} = 3 > 2.52 = \texttt{EEGP} \gg 2.056 = \texttt{NEGP},$$

so we try to charge `NEGP` to the user to further adjust the gas price.

5. In this case, suppose that the transaction does not have neither `GASPRICE` nor `BALANCE` (from the source address) opcodes, so we will execute the transaction with

$$\texttt{GasPriceFinal} = \texttt{NEGP} = 2.056 \text{ GWei/Gas}.$$

Observe that `GasUsedFinal` should be the same as $\texttt{GasUsedNew} = 95,000$. We can now compute `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.