polygon zkEVM

Knowledge Layer

Architecture

Interacting with the zkEVM Network

v.1.0

March 6, 2024

1 Introduction

In this section, we will address the interaction of a user with the Polygon zkEVM network through a component known as zkEVM **JSON-RPC**. To begin, we need to provide an overview of how blocks are generated in L2. We will also provide a detailed explanation of how transactions are processed when they are sent through JSON-RPC, including the validations that occur before they are stored in the **Pool**. The main components involved in this process are the **JSON-RPC**, **Pool**, **Sequencer**, **Executor**, Prover **HashDB**, and Node **StateDB**.

2 zkEVM JSON-RPC

The **Ethereum JSON-RPC** is a standardized collection of methods used by all Ethereum nodes, acting as the main interface for users to interact with the network. Figure 1 illustrates a user querying the Ethereum JSON-RPC to retrieve information on the latest forged block. The user's retrieved data includes all information related to the block, including its transactions.

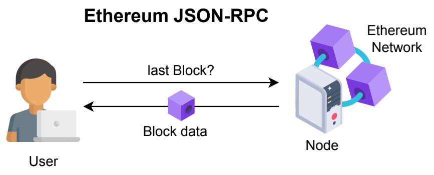

Figure 1: The **user** asks for information for the last block and **Ethereum JSON-RPC** sends it.

For instance, the user can request it as follows:

```
curl -X POST --data '{
  "jsonrpc":"2.0",
  "method":"eth_getBlockByNumber",
  "params":["latest", false],
  "id":1
}' localhost:8545
```

The zkEVM architecture refers to **zkEVM JSON-RPC** as the interface compatible with Ethereum JSON-RPC for any Layer 2 system equal to Ethereum. It replicates the same endpoints and receives similar answers. The zkEVM JSON-RPC incorporates all Ethereum JSON-RPC methods, although some answers from zkEVM JSON-RPC are specifically relevant within the zkEVM context.

The zkEVM JSON-RPC not only includes the methods from the Ethereum JSON-RPC but also offers additional methods to handle zkEVM-specific functionalities. The whole list of available endpoints and their current implementation status can be located in the following link: json-rpc-endpoints.md. The API specification for the zkEVM endpoints is available in the endpoints_zkevm.openrpc.json file, which follows the OpenRPC standard.

Ethereum's JSON-RPC responds with the most recently created L1 block when requesting the last block. We have batches that transition through various states: **trusted**, **virtual**, and **consolidated**. We will now determine the expected response from the zkEVM JSON-RPC when queried for the latest L2 block.

3 zkEVM L2 Blocks and Batches

3.1 L2 Block Definition

The primary purpose when defining an L2 block is to minimize delay. We must select between trusted, virtual, or consolidated batches based on this premise. To minimize latency, it is clear that defining trusted batches as L2 blocks is the best choice. The delay in this case is `close_a_batch_time`, which is the shortest one.

We can keep questioning if the zkEVM can offer a reduced delay. This can be achieved if we generate an L2 block before its associated batch is closed. We can generate L2 blocks that incorporate transactions immediately once their order is determined, meaning once it is established that they will be included in a batch at a specific location. The extreme situation, which is the one that we use in our current design, is that we create an L2 block for each ordered L2 transaction. This minimizes delay by allowing transactions to be processed without waiting for additional transactions to close an L2 block. This delay will be referred to as `order_a_transaction_time`.

3.2 Fork-Dragonfruit

In general, it is important to establish a clear distinction between batches and blocks within the zkEVM framework. An **L2 batch** represents the minimal data unit for which a proof is generated. On the other hand, an **L2 block** is the data structure returned to users through the RPC, and it belongs to L2 batches. L2 blocks, as their associated batches, traverse through all zkEVM states: trusted, virtual, and consolidated. From the user's perspective, the block is the visible entity. Up to the fork-dragonfruit (fork-5) included, each L2 block encapsulates a single transaction, adopting the same approach as Optimism. Consequently, every batch contains as many blocks as transactions, it is illustrated in the provided Figure 2. The formation of a block occurs when the decision is made for the transaction to be included in the trusted state. A **timestamp** is included as part of part of the batch data, and it is shared across all the blocks within the batch.

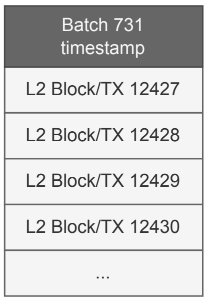

Figure 2: A batch with several blocks. Approach fork-dragonfruit (fork-5).

3.3 Fork-Etrog

Fork-dragonfruit approach assigns each transaction to a block, reducing delay to a minimum that cannot be further decreased beyond `order_a_transaction_time`. However, this approach has drawbacks as it results in a significant amount of data in the database due to the large number of L2 blocks created. Additionally, it does not offer a method for assigning unique timestamps to each block within a batch, causing issues for Dapps that rely on this parameter to determine the timing of actions performed by smart contracts.

We address these difficulties in the fork-etrog (fork-6), which allows blocks with multiple transactions by implementing a short timeout period when constructing the block to wait for transactions. The sequencer can modify the timestamp of various blocks in a batch by utilizing a specific transaction or marker known as `changeL2Block`. This transaction is included in the batch to indicate anytime there is a change in blocks and is responsible for modifying the timestamp and the L2 block number. Figure 3 demonstrates the functionality of this new type of transaction.

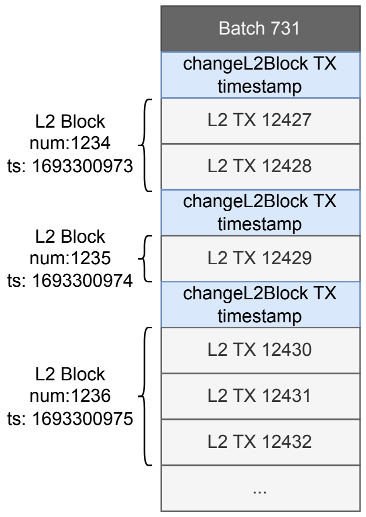
Batch 731
changeL2Block TX timestamp
L2 Block num:1234 ts: 1693300973
L2 TX 12427
L2 TX 12428
changeL2Block TX timestamp
L2 Block num:1235 ts: 1693300974
L2 TX 12429
changeL2Block TX timestamp
L2 Block num:1236 ts: 1693300975
L2 TX 12430
L2 TX 12431
L2 TX 12432
...

Figure 3: Fork-6 new approach including the `changeL2Block` transaction.

4 Custom zkEVM Endpoints

When a user requests information from the JSON-RPC regarding the latest block, the response will show the most recent L2 block. This L2 block contains the most recent transactions approved by the trustworthy sequencer to be arranged within the system. We retrieve this information using the identical RPC call as in Ethereum.

Perhaps as users we are not concerned with the status of our transaction being in a block, but rather with when the batch transitions to a new state. The zkEVM protocol has additional endpoints for these circumstances to offer more information regarding the status of the L2 block. For example, the user may ask whether a block is virtualized using a specific endpoint as shown in Figure 4.

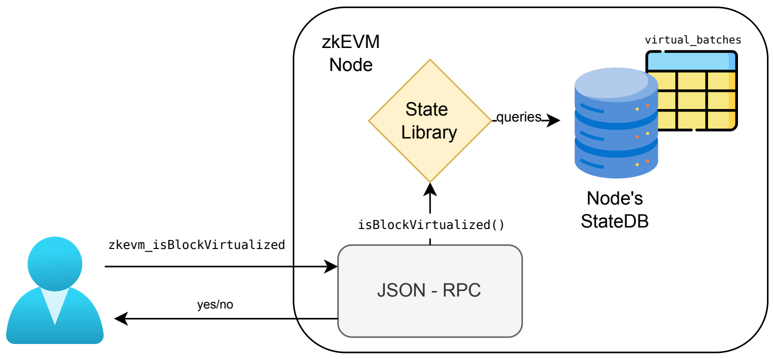

Figure 4: Request from the user for knowing if the block is virtualized.

Here is a collection of the main custom zkEVM endpoints, each accompanied by a brief description.

- `zkevm_consolidatedBlockNumber`: Returns the latest block number within the last verified batch.
- `zkevm_isBlockVirtualized`: Returns true if the provided block number is in a virtualized batch.
- `zkevm_isBlockConsolidated`: Returns true if the provided block number is in a consolidated batch.
- `zkevm_batchNumber`: Returns the latest batch number.
- `zkevm_virtualBatchNumber`: Returns the latest virtual batch number.
- `zkevm_verifiedBatchNumber`: Returns the latest verified batch number.
- `zkevm_batchNumberByBlockNumber`: Returns the batch number of the batch containing the given block.
- `zkevm_getBatchByNumber`: Gets the info of a batch given its number.

5 Sending Raw Transactions

Figure 5 illustrates the process when a user invokes the `eth_sendRawTransaction` endpoint to submit a transaction to the **PoolDB**.

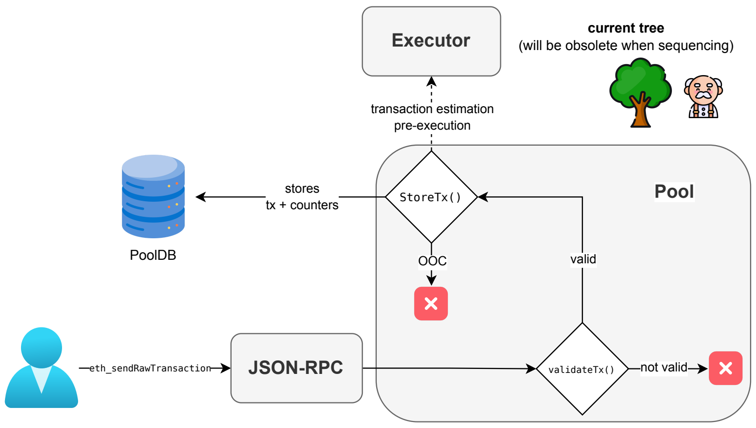

Figure 5: Sending a transaction to the system.

When an L2 transaction is received, **JSON-RPC** sends it to the **Pool** component. The **Pool** component is responsible for, among other things, adding transactions into the **PoolDB**. The Pool component conducts initial validations on transactions. If any of these validations fail, an error is sent to the JSON-RPC component, which then forwards the error to the user. It also conducts a transaction estimation pre-execution using the current state root, which may differ from the state when the transaction is ultimately ordered.

The first process that takes place in the Pool, as we can see in Figure 5, is the `validateTx()`. In this function take place the preliminary validations:

1. The transaction IP address has a valid format.

2. The transaction fields are properly signed (in both current and pre-EIP-155). EIP-155 states that we must include the chainID in the hash of the to be signed data (which is an anti replay attack protection).

3. the transaction's `chainID` is the same as the pool's `chainID` (which is the `chainID` of the L2 Network) whenever `chainID` is not zero.

4. The transaction string has an encoding that is accepted by the zkEVM (more in Section 5.1).

5. The transaction sender's address can be correctly retrieved from the transaction, using the *ecRecover algorithm*.

6. The transaction size is not too large (more specifically, larger than 100132 bytes), to prevent DOS attacks.

7. The transaction's value is not negative (which can be the case since we are passing parameters over an API).

8. The transaction sender's address is not present in the **black list** (that is, is not blocked by the zkEVM network).

9. The transaction preserves the nonce ordering of the account.

10. The transaction sender account has not exceeded the maximum of transactions to have on the pool per user.

11. The pool is not full (currently, the maximum amount of elements in the queue of pending transactions is 1024).

12. The gas price is not lower than the set **minimum gas price** (this will be explained in more detail later in a section devoted to *economics*).

13. The transaction sender account has enough funds to cover the costs ($\texttt{value}+\texttt{GasPrice}\cdot\texttt{GasLimit}$).

14. The intrinsic gas of a transaction measures the required gas by the amount of transactional data plus the starting gas for the raw transaction (which is currently of 21000 or 53000 in case of being a contract creation transaction). If the provided gas is less than the computed intrinsic gas, the check is not passed.

15. The current transaction gasprice is higher than the other transactions in the **PoolDB** with the same `nonce` and `from`. This is because you cannot replace a transaction with another with less gasprice.

16. The sizes of the transaction's fields are compatible with the **Executor** needs, more specifically:

    - `Data` size: 30000 bytes.
    - `GasLimit, GasPrice` and `Value`: 256 bits.
    - `Nonce` and `chainID`: 64 bits.
    - `To`: 160 bits.

5.1 zkEVM Transaction Custom Encoding

Returning to point 4 of the `validateTx()` validations explained before, let's analyze the zkEVM transaction custom encoding. The zkEVM currently only supports **non-typed transactions**. You can refer to standard transaction encodings in Ethereum for further information. The `to-be-signed-hash` remains consistent between pre-EIP155 and EIP155 transactions, resulting in identical signatures for these transactions.

$$\texttt{to-be-signed-hash}_{\texttt{pre-EIP155}} = \texttt{keccak}(\texttt{rlp}(\texttt{nonce}, \texttt{gasPrice}, \texttt{startGas}, \texttt{to}, \texttt{value}, \texttt{data}))$$

$$\texttt{to-be-signed-hash}_{\texttt{EIP155}} = \texttt{keccak}(\texttt{rlp}(\texttt{nonce}, \texttt{gasPrice}, \texttt{startGas}, \texttt{to}, \texttt{value}, \texttt{data}, \texttt{chainID}, 0, 0))$$

The transaction string used for L2 transactions in the batches is a slightly modified version of the regular transaction string. The goal is to streamline the proving system's processing of L2 transactions.

Let's analyze the handling of a typical EIP155 transaction on our chain with a chain ID of 1101 and a parity of 1. We would receive the following transaction string:

$$\texttt{rlp}(\texttt{nonce}, \texttt{gasPrice}, \texttt{startGas}, \texttt{to}, \texttt{value}, \texttt{data}, \texttt{r}, \texttt{s}, \texttt{v} = 2238)$$

To check the signature, we have to rlp-decode the previous string to extract `nonce`, `gaspPrice`, `startGas`, `to`, `value`, `data`, `r`, `s` and `v`. Then, we have to compute the chain identifier and the parity from the `v`, in our example:

$$\texttt{chainID} = \lceil(\texttt{v} - 35)/2\rceil = \lceil(2238 - 35)/2\rceil = 1101,$$
$$\texttt{parity} = \texttt{v} - (2 \cdot \texttt{chainID}) - 35 = 2238 - (2 \cdot 1101) - 35 = 1.$$

Then, we have to compute rlp-encoding of the following parameters

$$\texttt{rlp}(\texttt{nonce}, \texttt{gasPrice}, \texttt{startGas}, \texttt{to}, \texttt{value}, \texttt{data}, 1101, 0, 0),$$

and compute its Keccak hash:

$$\texttt{to-be-signed-hash} = \texttt{keccak}(\texttt{rlp}(\texttt{nonce}, \texttt{gasPrice}, \texttt{startGas}, \texttt{to}, \texttt{value}, \texttt{data}, 1101, 0, 0)).$$

Finally, we have to validate the ECDSA signature over `to-be-signed-hash` with `r`, `s` and the `parity`.

As you may notice, there is a rlp-decoding and a rlp-encoding of the transaction parameters. To just do the decoding once, we can just create the transaction string as the `to-be-signed-hash` and concatenate the `r`, `s` and `parity` which are the only extra parameters required to verify the signature. According to this reasoning, we use the following **raw transaction string** for the zkEVM L2 transactions:

$$\texttt{rlp}(\texttt{nonce}, \texttt{gasPrice}, \texttt{gasLimit}, \texttt{to}, \texttt{value}, \texttt{data}, \texttt{chainID}, 0, 0) \mid \texttt{r} \mid \texttt{s} \mid \texttt{v} \mid \texttt{effectivePercentage}.$$

The `parity` is expressed in the one-byte `v` parameter that can take the pre-EIP155 `parity` values (27 and 28). The parameter `effectivePercentage` which is a specific parameter of the zkEVM related to the fee system. Since the transaction signature is the same, it is easy to transform standard transactions into our custom format and vice-versa (here you JS utilities for doing these transformations).

5.2 Storing Transactions in the PoolDB

Once the first checks are completed without any errors, the function `StoreTx()` is executed. The function performs a **transaction estimation pre-execution**, which involves estimating **zkCounters** for pricing and prioritization purposes, even if it is based on a potentially incorrect state root. Further details on this will be discussed in another document. An error will be raised and the transaction will not be added to the **PoolDB** if we get an OCC (out-of-counters) error at this moment.

After the transactions are stored in the **PoolDB**, the **Sequencer** chooses certain transactions, attempts to close a batch from them, and then inserts them all into a designated table within the node's **StateDB**, which monitors the L2 blocks. The **Sequencer** also updates the **HashDB** with the L2 state data obtained by pre-executing the batch again. Figure 6 illustrates the process. This second batch pre-execution is expected to be correct at this stage of the processing as it uses the current tree to calculate the exact Gas and zkCounters needed.

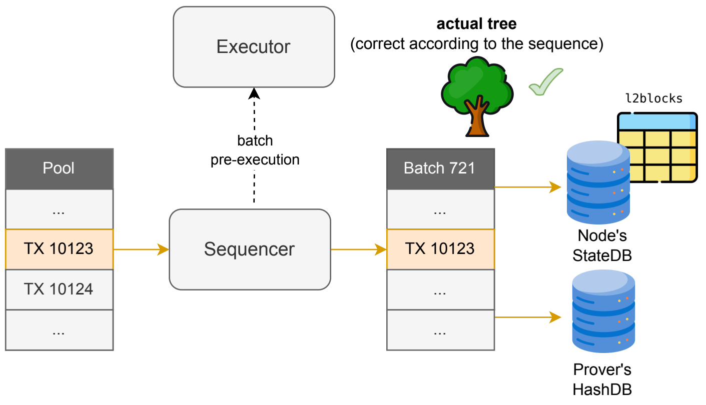

Figure 6: A second pre-execution takes place once the Sequencer sequence a certain transaction and includes it in a certain batch. Observe that, in this case, the state root is the correct one. During this second pre-execution, sequencer updates the **HashDB** accordingly.

After this process, the **Executor** performs the last execution known as batch execution during which the **Prover** can generate the corresponding proof. If all functions correctly, the second pre-execution and the final execution should be identical. The whole process is summarized in Figure 7.

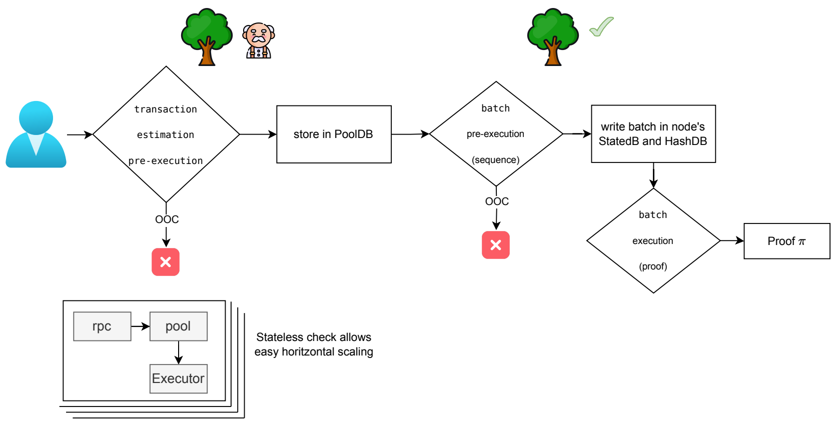

Figure 7: Diagram illustrating the three executions occurring between the user sending a transaction and the generation of a proof.

Figure 7 shows that the counters check is performed twice. The first check is an estimation, while the second one provides the actual values. If all goes as planned, the two final checks should be identical, as previously mentioned.