The zkEVM Architecture

Part V: Unified LXLY (uLXLY)

Work In Progress

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: 4f8b3de8f79deeaa4c17ce3c6fde91b00999fe2d

February 22, 2024

Outline

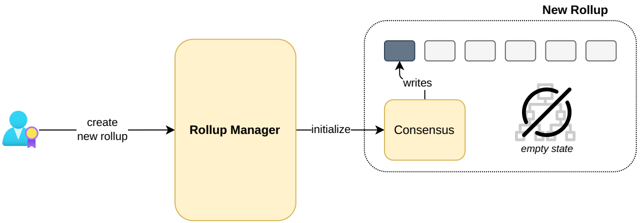
Unified LXLY

- **Unified LXLY** aims to streamline the creation and management of different layers 2 within the Polygon network, including both *rollups* and *validiums* among the Polygon network, ensuring possible exchanges between them.

  Note: While not technically precise, we will refer to both rollups and validiums as *rollups* for simplicity.
- To achieve this goal, a new smart contract called `RollupManager` has been developed to manage de creation of rollups and their state progress through the verification of their batches.

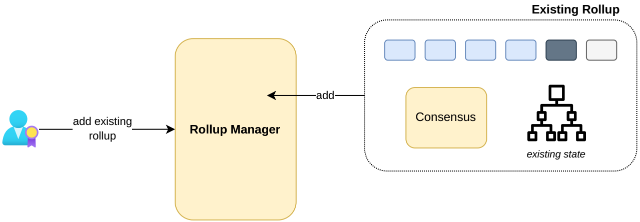
New Rollups and Existing Rollups i

- The first scenario involves newly created rollups, which are not yet initialized and have an empty state.
- When a user triggers a function of the rollup manager to create a new rollup, the rollup manager should populate the configuration parameters and initialize the rollup by generating and writing the genesis block, altogether with the having to sequence the transactions for initializing the Bridge contract attached to the rollup.



New Rollups and Existing Rollups ii

- Otherwise, when an operational rollup **exists** on the Ethereum network (thus having a non-empty state), a user with appropriate permissions can integrate it into the Rollup Manager for unified management
- In this case, the consensus must not initialize anything since the rollup, the genesis block and the corresponding Bridge are set up previously.



Rollup Types

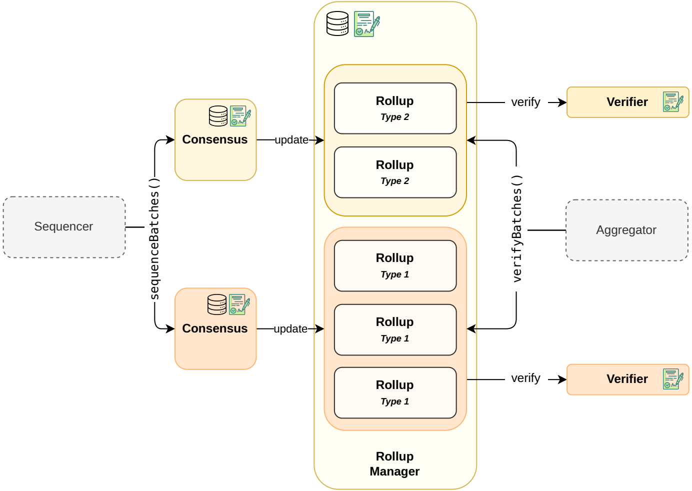
- New rollups have a `RollupType` attached.
- The `RollupType` specifies the following parameters:
  - **The consensus implementation address**, which is the address of the contract responsible for sequencing the batches.
  - **The verifier address**, implementing the `IVerifierRollup` interface, which allows the verification of a proof sent by the **Aggregator**.
  - `The forkID`, for tracking changes in the rollup processing.
  - **A rollup compatibility identifier**, which will be used to prevent compatibility errors when willing to *upgrade* a rollup.
  - **The obsolete flag**, which is a flag for indicating whether the rollup is obsolete or not.
  - **The genesis block**, which is the initial block of the rollup and which can include a small initial state.

Remarks About Rollup Types

Some remarks about Rollup Types:

- Note that there can be several rollups having the same `RollupType`, which means that they all share the smart contracts for consensus and batch verification.
- In the `RollupManager` contract, there are functions designed to add (`addNewRollupType()`) and to obsolete (`obsoleteRollupType`) rollup types.
- It is not possible to create rollups having an obsolete rollup type.

uLXLY Bird's View



Rollup Data

Each rollup, apart from having a `RollupType` attached, should store some important state data, which is included in a struct called `RollupData`.

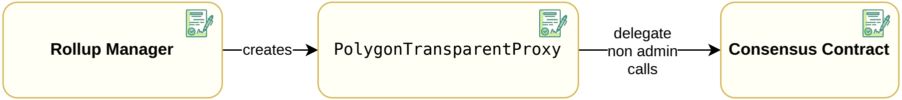
This struct contains information from the current **state** of the rollup (for example, the current batch being sequenced or verified, the states root for each batch, etc.), information of the **bridge** within the rollup (such as the current local exit root) and **forced batches** data, which will be explained in another document.

```
struct RollupData {

    IPolygonRollupBase rollupContract;
    uint64 chainID;
    IVerifierRollup verifier;
    uint64 forkID;

    mapping(uint64 batchNum => bytes32)            batchNumToStateRoot;
    mapping(uint64 batchNum => SequencedBatchData)  sequencedBatches;
    mapping(uint256 pendingStateNum => PendingState) pendingStateTransitions;

    bytes32 lastLocalExitRoot;
    uint64  lastBatchSequenced;
    uint64  lastVerifiedBatch;
    uint64  lastPendingState;
    uint64  lastPendingStateConsolidated;
    uint64  lastVerifiedBatchBeforeUpgrade;
    uint64  rollupTypeID;
    uint8   rollupCompatibilityID;

  }
```

Creating a New Rollup i

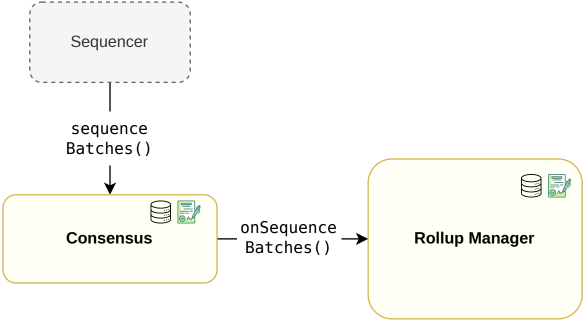
- Each rollup is associated with either none or a single rollup type.
- In order to create a rollup of a certain rollup type, we can use the function `createNewRollup()` by specifying:
    - The associated non obsolete rollup type identifier, which should exist.
    - The `chainID` of the rollup among the Polygon network, which should be new.
    - The address of the **admin** of the rollup, which will be able to update several parameters of the consensus contract (such that setting a trusted sequencer or a force batches address).
    - The address of the **trusted sequencer**, which will be the one responsible for sending the transaction to execute the `sequenceBatches()` function.
    - The address of the token address that will be used to pay gas fees in the newly created rollup (more info on this later on).

- When creating a new rollup, we employ the transparent proxy pattern, by generating an instance of the `PolygonTransparentProxy` contract, with the consensus contract specified by the rollup type serving as its implementation.
- The `RollupData` is *partially* filled (because the rollup is not currently initialized) and stored in the `rollupIDToRollupData` mapping within the contract's storage.
- To end up, the rollup creation calls the `initialize()` function of the consensus, which is in charge of setting the previously specified addresses in the consensus contract.



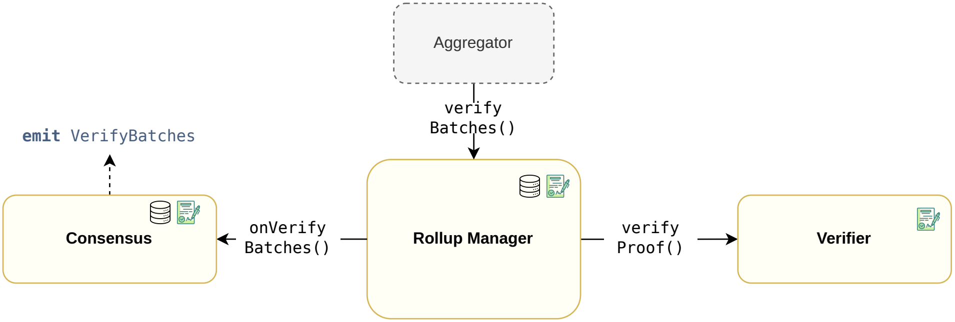
`RollupManager`: Sequencing Flow

- First of all, the **Sequencer** invokes the **`sequenceBatches()`** function within the Consensus contract to send the batches to be sequenced.
- Additionally, because the state information **must be stored** within the `RollupManager` contract, a callback function called `onSequenceBatches()` is triggered to store this data in the corresponding `RollupData` struct.

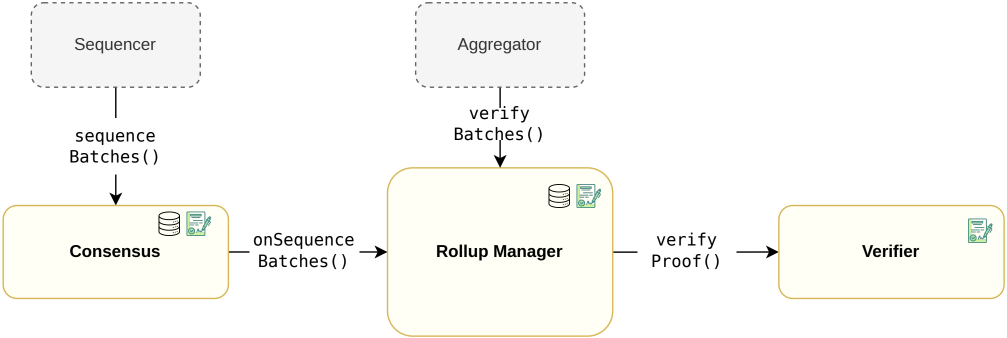


- Once the Aggregator has constructed the corresponding proof to validate the processing of a specific set of batches, it transmits the proof for verification to the `RollupManager` by invoking the `verifyBatches()` function.
- Then, the `RollupManager` invokes the `veriftyProof()` function at the verifier's contract.
- The previous function, either validates the proof or reverts if the proof is invalid.
- Upon successful verification of a proof, a callback function called `onVerifyBatches()` is called in the Consensus contract.
- The previous function emits the `VerifyBatches` event containing important details of the processed batch such as the last verified batch.

RollupManager: Verifying Flow ii



Sequencing and Verifying Batches Summary



Updating a Rollup: the `rollupCompatibilityID`

- This function provides upgradeability to the rollups.
- More specifically, a user with correct rights can change the consensus implementation and the rollup type of a certain rollup to modify its sequencing and/or verification procedures.
- In order to change the consensus, the function `UpdateRollup()` needs to change the transparent proxy implementation.
- In the upgrading procedure the `rollupCompatibilityID` comes into play: **in order to avoid errors, we can only upgrade to a rollup type having the same compatibility identifier as the original one**.
- If this is not the case, the transaction is reverted rising the `UpdateNotCompatible` error.

Existing Rollups i

- Rollups that are already deployed and already working does not follow any rollup type and are added to the `RollupManager` via the `addExistingRollup` function, specifying its current address.
- Meanwhile the verifier implements the `IVerifierRollup` interface we only request the raw consensus contract address, as it will not be used directly, but through a proxy to allow upgradeability options.
- As we have said before, we can add rollups that are deployed and already working to the `RollupManager` to allow unified management.
- In this case, we must call the function `addExistingRollup`.

Existing Rollups ii

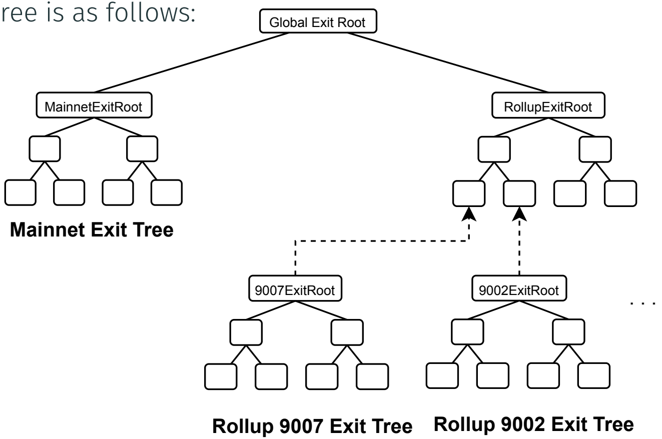
- Since the rollup has been previously initialized, we should only provide the following information:
  - The consensus contract, implementing the `IPolygonRollupBase` interface.
  - The verifier contract, implementing the `IVerifierRollup` interface.
  - The **`forkID`** of the existent rollup.
  - The **`chainID`** of the existent rollup.
  - The genesis block of the rollup.
  - The `rollupCompatibilityID`.
- Observe that most of these parameters were actually provided by the `RollupType`, but existent rollups `RollupData` is constructed by hand, since they do not follow any rollup type.

New Global Exit Tree

In uLXLY, we have multiple layers, so, we need to adapt the global exit tree for storing the exits on all these layers.

The design of the new global exit tree is as follows:

- Mainnet has a local exit tree built as an append-only tree of 32 levels.
- Each rollup, also has a local exit tree built as an append-only tree of 32 levels.
- Rollups are grouped in a tree of rollups, that again, is built as an append-only tree of 32 levels.



Rollup Identifiers

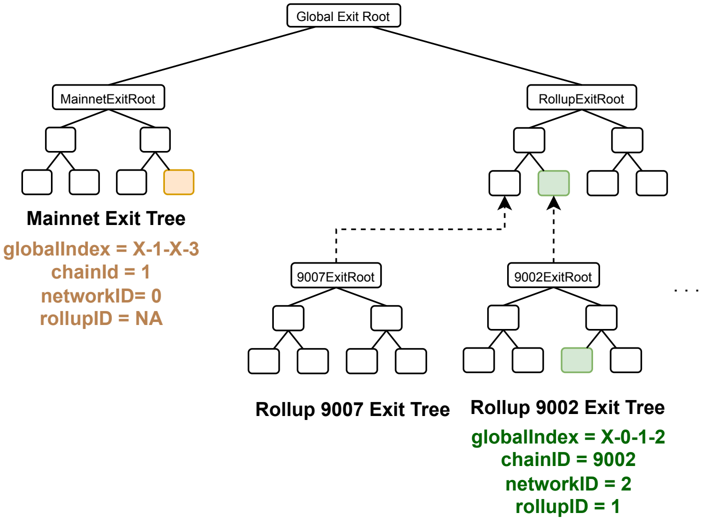
Every rollup has a set of distinct identifiers that are essential for its functioning and interaction within the larger network ecosystem:

- The `chainID`:
  - It is a unique identifier that distinguishes the rollup from other chains in the Ethereum ecosystem and it is crucial for preventing anti-replay attacks.
  - See the chainlist to see the chain Ids of the different networks.
- The `networkID`:
  - It is an identifier defines the networks that interact in the Polygon ecosystem.
  - The `networkID = 0` is used for Ethereum mainnet, while the `networkID = 1` is used for the zkEVM and so on for the rest of the networks.
- The `rollupIndex`:
  - It is an identifier used to identify a rollup within the rollup tree.
  - The first rollup (the zkEVM) has `rollupIndex` = 0 and in general: `rollupIndex` = `networkID - 1`.

`globalIndex` i

- To create and verify the proofs, we use an index called `globalIndex` that allows to uniquely locate a leaf in the new global exit tree.
- The `globalIndex` consists of a string of 256 bits, with its definition starting from the most significant bit as follows:
  - **Unused bits, 191 bits:** These bits are unused and can be filled with any value being the best option to fill them with zeros (since zeros are cheaper).
  - **Mainnet Flag, 1 bit:** This single bit serves as a flag indicating whether an exit pertains to a rollup (represented by 0) or to the mainnet (indicated by 1).
  - **Rollup Index, 32 bits:** These bits indicate the specific rollup we are pointing to within the rollup exit tree. This bits are only used whenever mainnet flag is 0.
  - **Local Root Index, 32 bits:** These bits indicate the specific index we are pointing to within each of the local exit trees of each rollup.

globalIndex ii



zkEVM Node Configuration

Node configuration of a rollup/validium:

```
{
    "l1Config": {
        "chainId": 1,
        "polygonZkEVMAddress": "Address of the consensus contract",
        "polygonRollupManagerAddress": "Address of the Rollup Manager contract",
        "polTokenAddress": "Address of the POL token contract",
        "polygonZkEVMGlobalExitRootAddress": "Address of the GlobalExitRoot contract"
    },
    "genesisBlockNumber": X,
    "root": "Initial Root hash after applying the L2 Genesis",
    "genesis": [...]
}
```

Notes:

- The `chainId` is the chain identifier of the base layer (Ethereum mainnet in this case).
- The `genesisBlockNumber` is block number at the base layer (Ethereum mainnet) in which the rollup/validium is created.

Rollup Synchronization

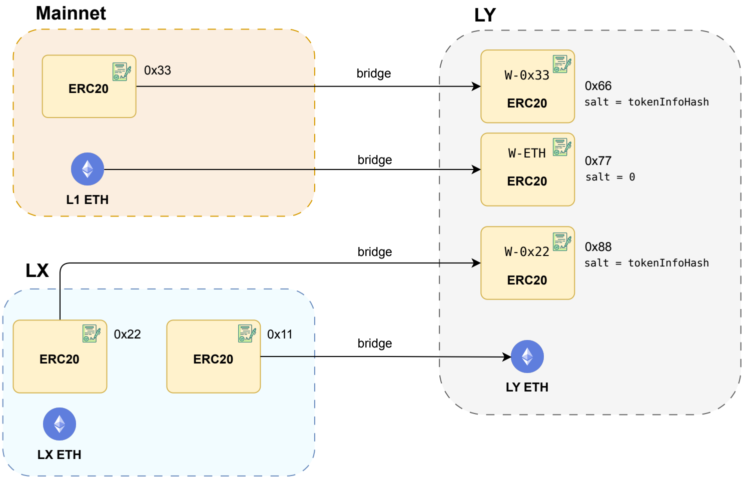
the l1InfoRoot mismatch: Nodes have to sync events from GER SC since rollup manager creation, but they only do so since rollup creation

This leads to missing events -> missing leaves -> root doesnt match

However, `genesisBlockNumber` seems useless (sync needs to go from the begginning of the deployment of the Rollup manager.

Rollups need to have the Global Exit Tree.

- The native currency to pay the **gas** at a certain layer can be:
  a) Any **ERC20 token** instance on any other layer.
  b) L1 ETH.
- If we use a token to pay the gas at a layer, we call this token, the **"gas token"** for the layer.



- If we are using a gas token at a layer, it is still possible to send **L1 ETH** to the layer.
- In this case, the ETH gets accounted in an ERC20 contract called **W-ETH**[1], which is just another **ERC20** instance.
- Regarding the creation of the ERC20 tokens with `CREATE2`:
  - We use `salt = 0` to create the `W-ETH` contract.
  - We use `salt = tokenInfoHash` for the rest of the wrapped tokens of the layer with `tokenInfoHash` defined as the following hash:

$$\texttt{tokenInfoHash} = \texttt{keccak256}(\texttt{originNetwork}, \texttt{originTokenAddress}).$$

- Finally, we would like to remark that L1 ETH is the only native currency that can be used as a native currency in another layer.

[1] Note that `W-ETH` is different from the contract `WETH` (a contract for converting ETH into an ERC20 token that runs at L1 at 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2).

Issue with the "Upgradable CREATE2 Factory" i

- Note that the Bridge contract is a factory of ERC20 token instances created with `CREATE2`.
- Recall that `CREATE2` uses the following formula to compute the address of the instances:

$$instance\_address = hash(0xFF, sender, salt, creationBytecode, [args])$$

- Recall also that in the Bridge contract, the mapping `tokenInfoToWrappedToken` stores the addresses of all the wrapped ERC20 tokens of the layer.
- The **problem** is that if we **change** the `creationBytecode` of the ERC20 token contract, this will change all the addresses of the contract instances and therefore, this breaks the data of the mapping.

Issue with the ”Upgradable CREATE2 Factory” ii

- The `creationBytecode` will change with high probability if we compile the factory (in our case the Bridge) with another version of the Solidity compiler.
- In this case, we had to options:
  a) Freeze the Solidity compiler version for the development of the whole Bridge contract.
  b) Freeze the `creationBytecode` of the ERC20 token contract.
- We opted for b) because the ERC20 code is not prone to change so much, while freezing the compiler (and the language) for the whole Bridge could constrain its future development.
- Taking this approach, in the `BASE_INIT_BYTECODE_WRAPPED_TOKEN` variable of the Bridge contract you can find the pre-compiled `creationBytecode` of our ERC20 token contract.