The zkEVM Architecture

Part VI: 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: f4cca7a4cea2e51a2596449f58d3aa0bf369ac13

February 15, 2024

Outline

Unified LXLY Introduction

- **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

All new rollups will have a `RollupType` attached, which will specify the following parameters:

FALTA DIBUJO

Rollup Types i

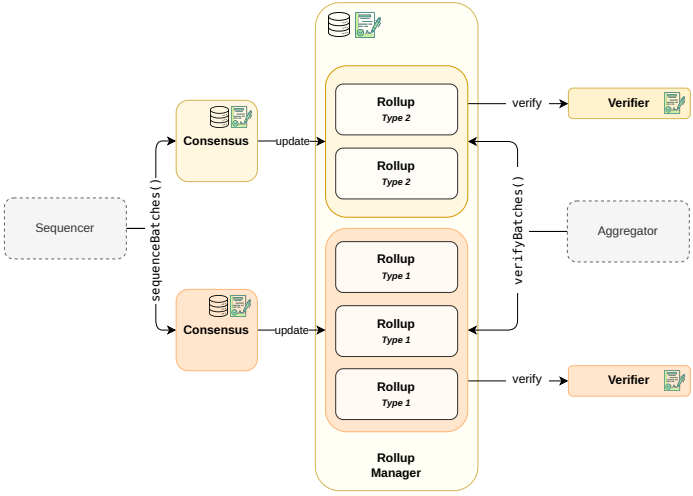
- New rollups have a `RollupType` attached.
- The `RollupType` specifies the following parameters:
  - **The consensus implementation address**, which is the address of the contract responsible for sequencing the batches.
  - **The verifier address**, implementing the `IVerifierRollup` interface, which 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  - **The obsolete flag**, which is a flag for indicating whether the rollup is obsolete or not.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.

Rollup Types ii

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.

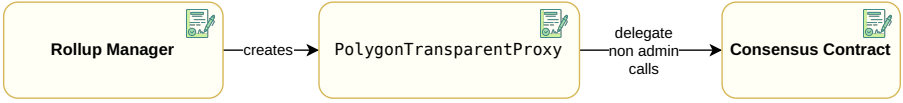
```
struct RollupData {

    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).

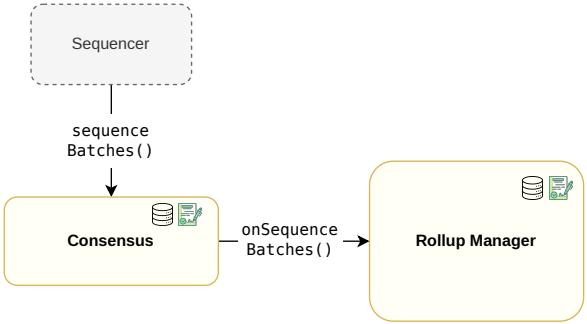
Creating a New Rollup ii

- 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

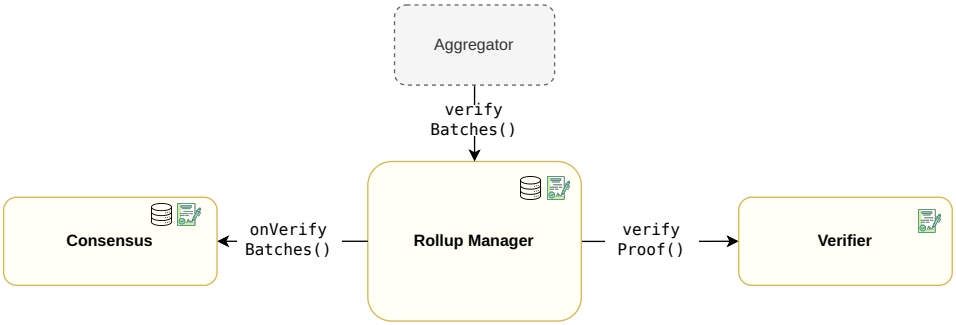
- First of all, the **Sequencer** invokes the `sequenceBatches()` function within the Consensus contract to send the batches to be sequenced.
- Additionally, because the state information **must be stored** within the `RollupManager` contract, a callback function called `onSequenceBatches()` is triggered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



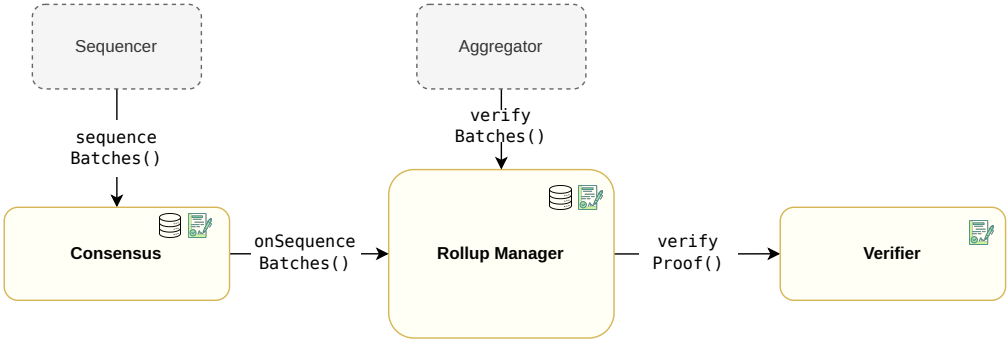
`RollupManager`: Verifying Flow i

- 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 procedure by means of upgrading 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.

zkEVM Node Configuration

Node configuration of a rollup/validium:

```
{
    "l1Config": {
        "chainId": 1,
        "polygonZkEVMAddress": "Address of the consensus contract",
        "polygonRollupManagerAddress": "Rollup Manager SC",
        "polTokenAddress": "polTokenAddress",
        "polygonZkEVMGlobalExitRootAddress": "GlobalExitRootAddress"
    },
    "genesisBlockNumber": X,
    "root": "Initial Root of the L2 Genesis",
    "genesis": [...]
}
```

Notes:

- The `chainId` is the chain identifier of the base layer (Ethereum mainnet in this case).
- The `genesisBlockNumber` is the L1 block number in which the rollup/validium is created.

Gas Tokens i

Upgradability and ERC20 CREATE2 Issue

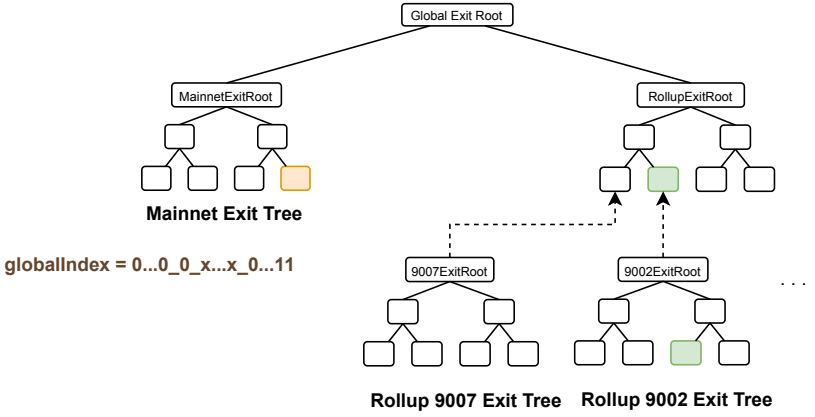
- Non predictable addresses depending on the compiled: `BASE_INIT_BYTECODE_WRAPPED_TOKEN` code is here to avoid different versions when deploying the token.

New Global Exit Tree i

- Each rollup and mainnet have a tree of 32 levels.
- There is a tree of rollups also of 32 levels.
- The globalIndex allows to compute the proofs in the new Global Exit Tree.
- The globalIndex Global index is a string of 256 bits starting from the msb are defined as:
  - **191 bits (unused):** must be set to 0s.
  - **1 bit (mainnet flag):** 0 for an exit not belonging to a rollup and 1 for an exit belonging to mainnet.
  - **32 bits (rollupIndex):**
  - **32 bits (localRootIndex):**

networkID 1 is leaf 0

New Global Exit Tree ii



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.