polygon zkEVM

Knowledge Layer

Architecture

Unified LXLY (uLXLY)

v.1.0

February 22, 2024

1 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.

2 The Rollup Manager

2.1 New vs Existing Rollups

New Rollups 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 (See Figure 1).

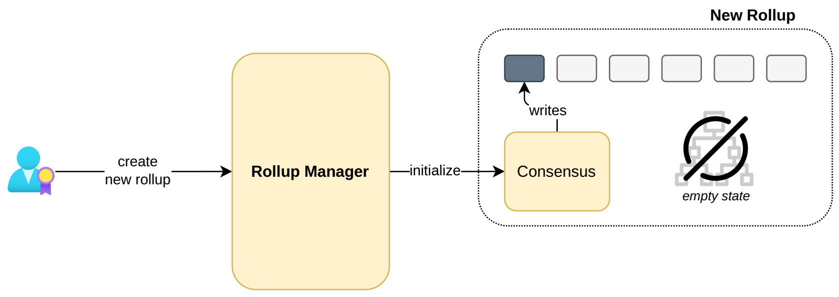

Figure 1: Diagram illustrating the process of creating and initializing a new rollup instance. Observe that the state tree is empty in this situation.

Existing Rollups When an operational rollup is present on the Ethereum network, a user with the necessary rights can incorporate it into the Rollup Manager for centralized management. In this scenario, the consensus should not be initialized as the rollup, genesis block, and corresponding Bridge have already been established previously (See Figure 2).

2.2 Rollup Type and Rollup Data

Rollup Type 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**.

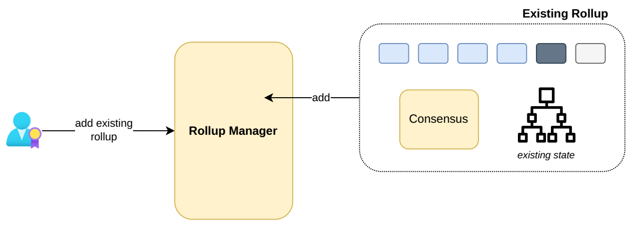

Figure 2: Diagram depicting the integration of an existing operational rollup into the Rollup Manager, showcasing the process without the need for initialization as the rollup is already established. In this case, the state tree is filled.

- 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.

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.

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;

}
```

Figure 3: The structure definition of `RollupData` includes type information, necessary state data, bridge details, and forced batches data for a specific rollup instance.

2.3 Creating a Rollup

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 (See Figure 4), 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.

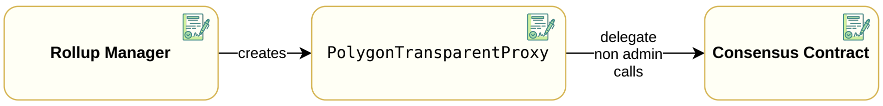

Figure 4: Schematic representation of the transparent proxy pattern within the Rollup Manager context. Proxies are frequently utilized in Ethereum for upgradability, albeit their management can pose difficulties, which can be solved by using a transparent proxy pattern that distinguishes calls based on the caller before the function selector. This ensures that interactions with the proxy are indistinguishable from interactions with the logic contract for all users except the admin.

2.4 Sequencing and Verifying Flow

2.4.1 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. An illustration of the sequencing flow can be found in Figure 5.

2.4.2 Verifying Flow

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

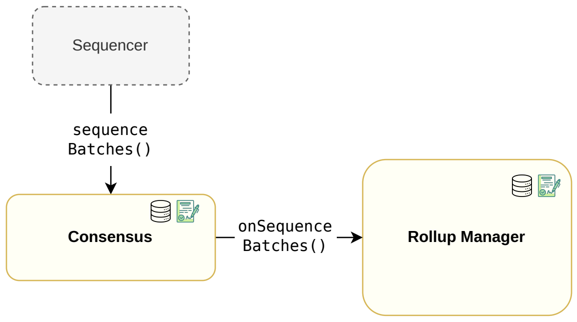

Figure 5: Depiction of the sequencing flow within the **Rollup Manager** component, which starts when the **Sequencer** calls the `sequenceBatches()` function. This function, in turn, invokes a callback function `onSequenceBatches()` on the **Rollup Manager** that stores the sequence data on the `RollupData` struct.

emits the `VerifyBatches` event containing important details of the processed batch such as the last verified batch. An illustration of the verifying flow can be found in Figure 6.

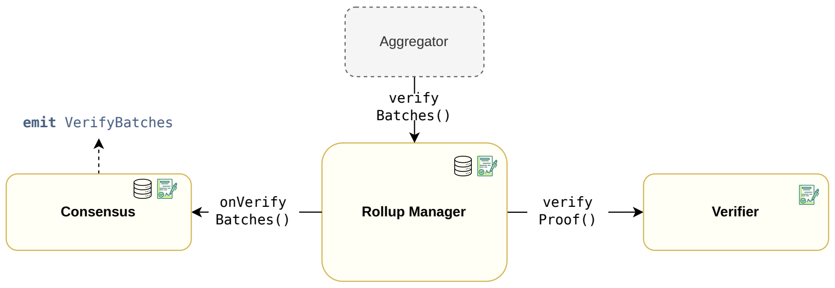

Figure 6: Depiction of the verification flow within the **Rollup Manager** component, which starts when the **Aggregator** calls `verifyBatches()` function. Observe that the verification flow involves a secondary stateless **Verifier** contract, which is used in order to invoke the function `verifyProof()`. At the end of the process, the consensus emit the `verifyBatches` event.

2.5 Updating a Rollup

This functionality 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.

2.6 Adding Existing Rollups

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`.

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.

3 Bridge within Layers

3.1 New Global Exit Tree

In uLXLY, due to the presence of multiple layers, it becomes necessary to adjust the global exit tree to accommodate exits across all these layers. The design of the updated global exit tree is depicted in Figure 7.

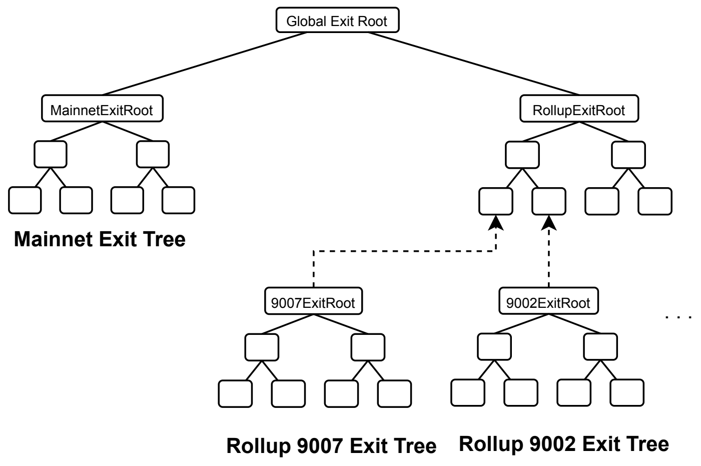

Figure 7: As can be observed, the global exit tree has been modified to introduce more than one rollup. Now, there are two main branches, one containing the root of the mainnet exit tree and the other containing the root summarizingthe exit trees of all the rollups, which we call the **rollup exit tree**. This tree has as its leaves all the local exit roots of the different rollups.

- 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` is a unique identifier that distinguishes the rollup from other chains in the Ethereum ecosystem and is crucial for preventing anti-replay attacks. See the chainlist to see the chain Ids of the different networks.
- The `networkID` identifier defines the rollup in the Polygon ecosystem, allowing network participants to uniquely identify and interact with them. 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`, which 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`.

Global Index. 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. Figure 8 exemplifies the usage of `globalIndex` to locate leafs on the tree.

- **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.

3.2 Node Configuration

Node configuration for a rollup/validium network typically involves specifying various parameters and addresses crucial for its operation. Figure 9 provides a detailed breakdown of the parameters within the provided TOML configuration snippet.

```
{
    "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": [...]
}
```

Figure 9: TOML file for the node configuration of a rollup.

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.

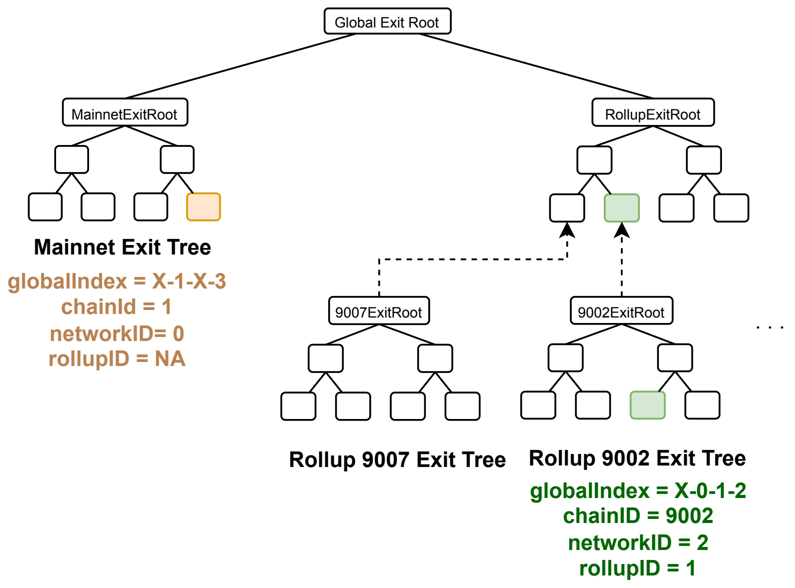

Figure 8: This figure shows how the `globalIndex` arranges the leaves of the tree. The `globalIndex` on the left has the mainnet flag set to 1, indicating that the bits associated with the rollup index are not utilized. Given a local root index of 3, we point the fourth leaf in the tree. In the right example, the mainnet flag is set to 0. Therefore, we utilize the rollup index to find the local exit tree associated with 1 as rollup index, which corresponds to the second leaf of the tree of rollups. Subsequently, we identify the leaf with index 2, which is the third leaf within the corresponding local exit tree.

3.3 Gas Tokens and Inter-Layer Exchanges

The native currency to pay the **gas** at a certain layer can be:

- Any **ERC20 token** instance on any other layer.
- 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**, which is just another **ERC20** instance. 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).

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. Figure 10 illustrates various inter-layer exchanges scenarios which can occur within the system.

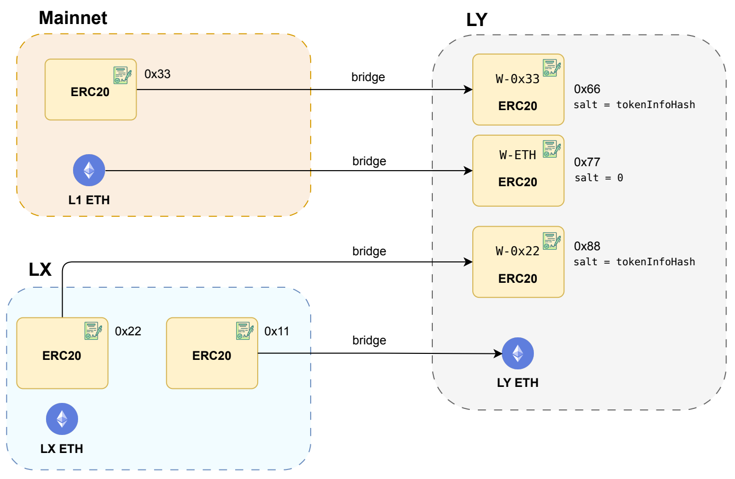

Figure 10: Diagram illustrating the interchange of assets between layers, focusing on **LY** as a layer of interest. It depicts several scenarios, such as bridging a ERC20 token from mainnet to another ERC20 token in LY, bridging L1 ETH to the LY gas token or bridging a wrapped ERC20 token living on LX to LY ETH.

Issue with the Upgradable CREATE2 Factory. 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:

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

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.

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 the second solution 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.