From Zero to zkEVM - Communication Layer

v.1.0

December 12, 2023

1 Document Pre-requisites

2 Communication Between Layers

Blockchain interoperability refers to the ability of a blockchain to interchange data with other blockchains. Since the blockchain ecosystem has expanded rapidly in recent years, a large number of networks with different specific properties have emerged and interoperability has become a crucial consideration in blockchain design. Without interoperability, a network risks being isolated from the larger ecosystem and this fact has supposed an incentive to projects to engage the research and development of interoperability solutions. Multiple approaches have been implemented in order to solve the problem, each one of them with particular trade offs and underlying technologies. This document describes the solution implemented by the Polygon team to bring native interoperable properties to the Polygon zkEVM L2 network.

The bridge is an infrastructure component that allows migration of assets and communication between different layers. From the point of view of the user, they should be able to transfer an asset from one network to another without changing its value or its functionality, as well as being able to send data payloads between networks (**cross-chain messaging**).

For sake of simplicity, we will start the bridge description by defining exchanges between L1 and an L2 but our intention is to be general, that is, to enable exchanges between multiple layers LX and LY. This is why we call this subsystem the **LXLY bridge**. For the explanations, we will use three layers denoted as LX, LY and LZ as represented in Figure 1.

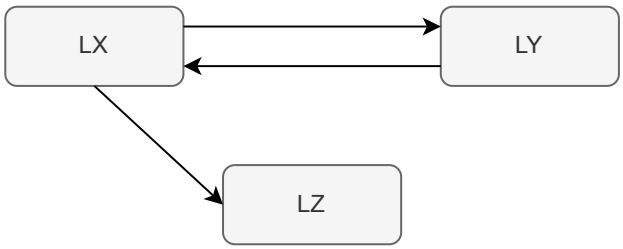

Figure 1: Layered Architecture Example: Illustration of Two Top Layers (LX, LY) and a Bottom Layer (LZ).

The objective of the bridge component is to enable the exchange of both **assets** and **messages** (See Figure 2). On the one hand, the term **asset** refers to any digital representation of value within the Ethereum blockchain. Ethereum supports two types of assets: **Ether (ETH)**, which is the native cryptocurrency of the network, and **Tokens**, including ERC-20 and ERC-721 tokens. On the other hand, a **message** refers to communication or interaction between two smart contracts, involving both data and a value (in ETH) transferred from the origin contract (the one initiating the message) to the specified destination contract. Within the zkEVM context, these messages entail the execution of a function `onMessageReceived` of some existing contract. This is what we call the **messaging mechanism** of the bridge.

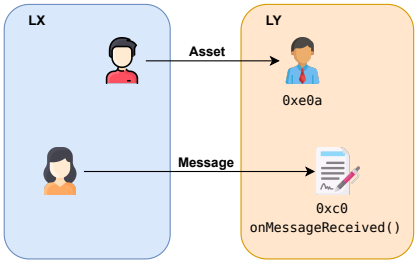

Figure 2: The bridge component should be able to exchange both assets and messages.

The core logic of the LXLY bridge is implemented in smart contracts. In particular, the main contract is called `zkEVMBridge.sol` that is deployed in any layer in which we want exchanging to be enabled. One of the design goals is that this smart contract **is exactly the same in all layers** (See Figure 3). This uniformity is intentional to maintain consistency in the logic whether exchanging assets or messages from a lower layer LY to an upper layer LX, or vice versa and hence externalizing the complexity associated with the layer position distinction from the core contract.

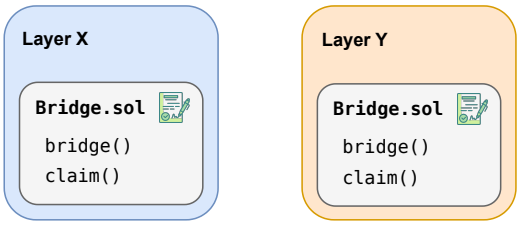

Figure 3: Uniform deployment of `zkEVMBridge.sol` in all layers ensures consistent handling of exchanges in the LXLY bridge's core logic.

The LXLY bridge follows a **bridge-claim model** (See Figure 4). Let us suppose we are in the situation where some LX user want to transfer an asset (or, alternatively, a message) to some LY user. In the origin layer LX, the source user sends a transaction to the `bridge()` function providing the destination network LY. From now on, transactions to the bridge function will also be known as **deposits**. Later on, in the destination layer LY, the user sends a transaction to the `claim()` function providing the origin network LX in order to claim the *transferred* (we will explore the meaning of that later on) asset (or message). To maintain a record of exchanges across layers, within the `zkEVMBridge.sol` smart contracts we need a compact way of storing the information of calls to the bridge function. These data, often referred to as **exits** or **outgoing transmissions**.

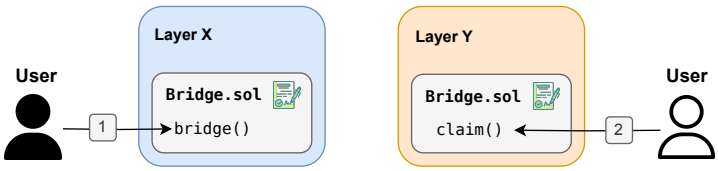

Figure 4: LXLY bridge adopts a bridge-claim model for transferring assets and messages.

3 Exit Trees

In order to store all the exits associated to a certain layer, the bridge contract builds an **append-only Merkle tree** having single exits, i.e., each call to the `bridge()` function, as its leaves. The Merkle tree with all the exits of a layer is called its **Local Exit Tree** (Figure 5) and, its root is called **Local Exit Root (LER)**. It is important to remark that an exit tree is a different object and has a different structure than the tree that stores the L2 state.

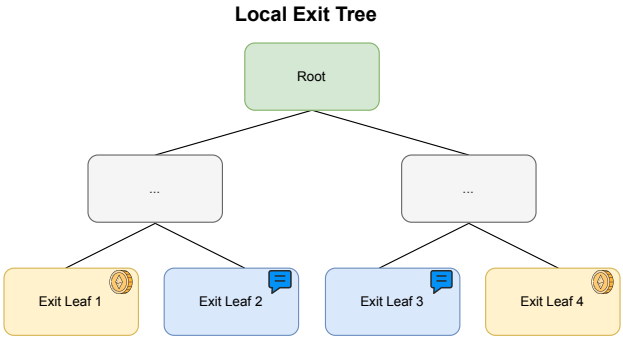

Figure 5: Illustration of the Local Exit Tree containing 4 exits (2 assets and 2 messages).

The following listing provides a detailed breakdown of the data structure encapsulated within each leaf of a Local Exit Tree:

```
uint8 leafType,
uint32 originNetwork,
address originAddress,
uint32 destinationNetwork,
address destinationAddress,
uint256 amount,
bytes32 metadataHash
```

- `leafType`: This parameter serves as an identifier, distinguishing whether the leaf corresponds to an asset or a message. A value of 0 designates an asset, while a value of 1 signifies a message.
- `originNetwork`: The identifier (`chainId`) of origin layer of the exchange.
- `originAddress`: For asset exchanges, it represents the address of the token contract. For message exchanges, it denotes the source address of the bridge call.
- `destinationNetwork`: The identifier of the destination layer (`chainId`) of the exchange.
- `destinationAddress`: Denotes the account receiving the asset or the address of the smart contract in the case of a message exchange.
- `leafAmount`: The amount of asset exchanged, encompassing both Ether or Tokens.
- `metadataHash`: This field captures the hash of the metadata associated with the exit. For asset exchanges, the metadata includes details such as the token's name, symbol, and decimals. Conversely, for message exchanges, the metadata comprises the calldata necessary for invoking the `onMessageReceived()` function, encapsulating the functional payload of the transmitted message.

In each layer, the corresponding bridge smart contract needs to be able to both **write** to and **read** from Local Exit Trees. On the one hand, the bridge contract will need to write in its Local Exit Tree each new exit resulting from a call to its `bridge()` function. On the other hand, the bridge contract needs to read the Exit Trees of other layers in order to process calls to the `claim()` function, since it should not be possible to claim something that has not been bridged, for example.

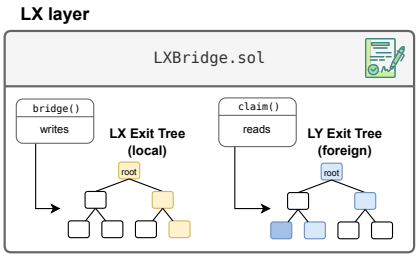

Figure 6: Bridge contracts in each layer needs to both read and write to Local Exit Trees for a correct tracking within the bridge-claim model.

Moreover, in each layer, the corresponding bridge smart contract needs to additionally nullify the claim. Each claim must be locally nullified in the bridge contract to avoid double claimings. The nullify process to avoid claiming transactions that have already been processed uses an efficient mapping known as `claimedBitMap` (we will explain how this bitmap works later in more detail).

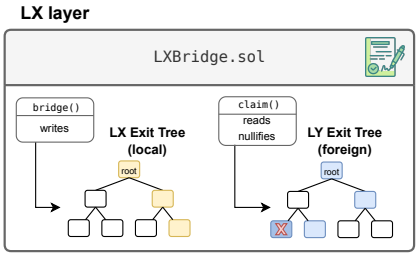

Figure 7: Bridge contracts locally nullify claims to prevent duplicate claimings.

4 Append Only Merkle Trees

This section entails the design of an append-only sparse binary Merkle tree whose successive roots can be computed with a minimal amount of persistent data. This kind of trees are designed for scenarios where the tree represents an ordered list of data elements which can only be modified by appending new ones. Rather than storing the whole the tree, only two key elements are stored:

- An array of the size of the tree depth, denoted to as `branch`.
- The last appended element's index, denoted to as `lastElementIndex`.

Importantly, the depth of the tree, representing its maximum capacity, is known a priori. This drops the necessity to use markers to distinguish between branches and leaves. Furthermore, note that the tree is inherently balanced.

By convention, for our append-only tree we are going to use 0s as default value for empty leaves. Hence, when the list represented using theincremental Merkle tree is empty, the following holds:

$$
\begin{aligned}
h_i &= 0, \\
h_{j,k} &= h(0,0) := h^{(00)}, \\
h_{m,n,l} &= h(h^{(00)}, h^{(00)}) := h^{(0000)}, \\
&\dots
\end{aligned}
$$

Here, h denotes the employed hash function, and the notation $\mathbf{,}$ signifies juxtaposition. Observe that, in this scenario, each of the hashes stored **do only depend on the level being stored**. Hence, we just need to compute a different hash value per level.

A Toy Example For sake of simplicity, lets consider as a toy example a small incremental Merkle tree of a maximum capacity of 8 leaves (hence, the having maximum depth of $d = \log_2(8) = 3$).