polygon zkEVM

Technical Document

Bridge
v.1.1

February 13, 2023

Contents

1 Introduction

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 L1 and L2. 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).

In L2 rollups as Polygon zkEVM, the management of the L2 State transitions and the data availability of transactions is secured by L1 contracts, hence with a correct design of the L2 architecture we can also synchronize both ends of the a bridge relying only in contract's logic, without the need of off-chain trusted reliers to sync the bridge's ends across networks. It's important to note that this type of bridge must be included into the design of the L2 layer.

As can be seen in figure 1, the bridging interface is a bridge contract instance present in both networks whereby users will be able bridge assets (1), that is, lock an asset in "origin" network, and eventually, claim an asset's representative token that will be minted by the bridge contract in "destination" network. The reverse operation will be possible as well (2) , that is, to burn the asset's representative token and unlock the original asset in "origin" network. Another possible usage is as cross-chain communication channel (3), that is, to send a data payload from L1 to L2 or vice versa.

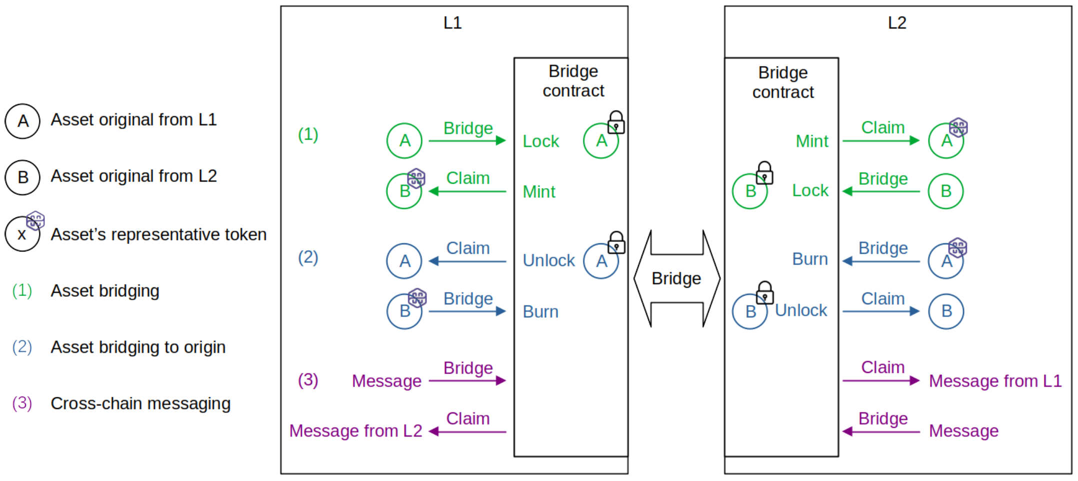


Figure 1: Polygon zkEVM bridge schema.

2 Exit Merkle trees

The bridge system is composed of a Merkle tree known as the Global Exit Merkle Tree (GEMT). In this tree, every leaf node represents the Merkle root of a specific network Exit

Merkle Tree (EMT). The GEMT has only two leaves, one corresponding to the root of the L1 EMT , and the other corresponding to the root of the L2 EMT. Figure 2 shows the structure of the GEMT.

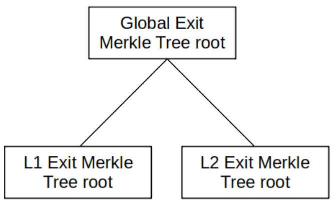


Figure 2: GEMT structure.

While GEMT is a fixed 2 leaves regular Merkle tree, EMT are append only sparse Merkle trees (SMT) with a fixed depth (32 in Polygon zkEVM). SMT are massive Merkle trees that can be handled in an efficient way on-chain (detailed in appendix A).

Every leaf of specific network EMT will represent an intention of bridge an asset (or asset's representative token) or send a message out of that network. Specifically the leaf will be a keccak256 hash of the abi encoded packed structure of the following parameters:

- **uint8 leafType:** [0] asset, [1] message.
- **int32 originNetwork:** Origin Network ID, where the original asset belongs.
- **address originAddress:** If leaf type = 0, Origin network token address, "0x0000...0000" address is reserved for ether. If leaf type = 1 msg.sender of the message.
- **uint32 destinationNetwork:** Bridging destination network ID.
- **address destinationAddress:** Address that will receive the bridged asset in the destination netwok.
- **uint256 amount:** Amount of tokens/ether to brige.
- **bytes32 metadataHash:** Hash of the metadata. Metadata will contain information about asset transfered or the message payload.

Once a leaf has been added to EMT, a new EMT root will be computed, and subsequently, a new GEMT root. The GEMT root will be synced between networks, making it possible to prove leaf inclusion on the second network and complete the bridge operation.

3 Contracts Architecture

Most of the bridge architecture is implemented using smart contracts on both networks. However, to synchronize GEMT among them, a portion of the bridge logic must be integrated with the L2 State management architecture. So that, to provide a comprehensive description of the bridge, one must take into account the off-chain actors involved in L2 state management, including a Sequencer, an Aggregator, and the **PolygonZkEVM.sol** contract.

In addition to the previously mentioned components, the following components are also included in the bridge architecture:

- **PolygonZkEVMBridge.sol:** bridging interface, allows users to interact with the bridge and perform actions such as bridging and claiming assets or messages. It has an instance in each network and manages its EMT.

- **PolygonZkEVMGlobalExitRoot.sol:** Manages the GEMT, which involves storing the tree and computing new root values every time a new EMT is updated by the PolygonZkEVM.sol contract (updates L2 EMT) or the L1 PolygonZkEVMBridge.sol contract (updates L1 EMT). Acts as GEMT historical repository.

- **PolygonZkEVMGlobalExitRootL2.sol:** Special contract that allows the synchronization of GEMT and L2 EMT roots across networks. Has storage slots to store GEMT roots and L2 EMT root. This contract is special because its storage slots are directly accessed by the low-level zero-knowledge proving/verification system to ensure the validity of GEMT synced from L1 to L2, and the validity of L2 EMT synced from L2 to L1.

3.1 Bridging data flows.

Figure 3 shows the detailed architecture of the bridge and how the components interact with each other to achieve finality in bridge operations. As can be seen, depending on whether the bridge operation is from L1 to L2 (L1 -> L2) or from L2 to L1 (L2 -> L1), there are two possible data flows.

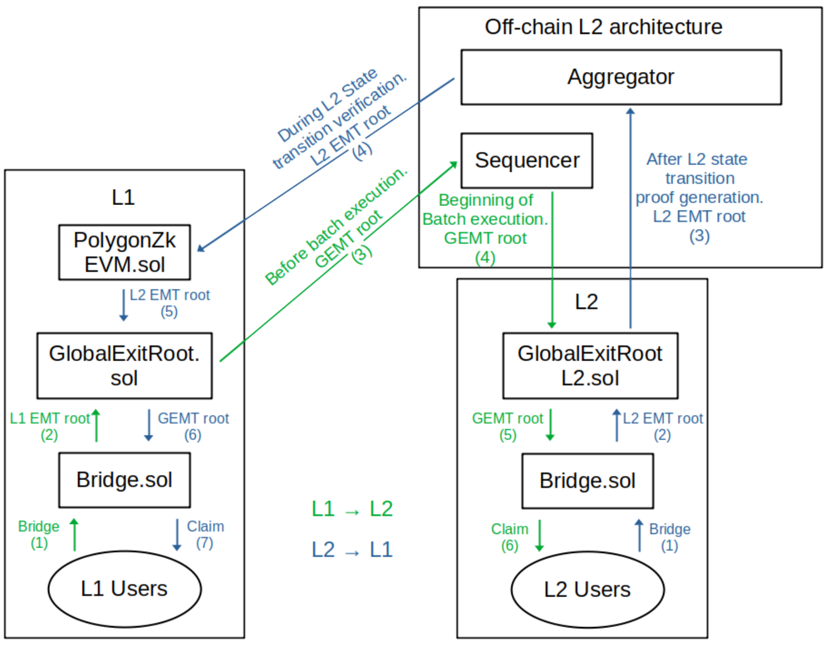


Figure 3: Polygon zkEVM bridge architecture.

The steps involved in each execution flow are enumerated and explained below:
L1 -> L2:

- **(1):** The user have to call the bridging function of the L1 **PolygonZkEVMBridge.sol** contract. Depending on the type of asset to be bridged the contract internals will act different. If the bridging request is valid, the contract will fill the L1 EMT leaf based on the characteristics of the request, add it into the tree, and compute the new L1 EMT root.

- **(2):** In the same L1 transaction the **PolygonZkEVMBridge.sol** contract will call the **PolygonZkEVMGlobalExitRoot.sol** contract to update the new L1 EMT root and consequently compute the new GEMT root.

- **(3):** Eventually, the Sequencer will fetch the new GEMT root to from **PolygonZkEVMGlobalExitRoot.sol** contract.

- **(4):** The new GEMT root will be entered in the special storage slots of the **PolygonZkEVMGlobalExitRootL2.sol** contract at the beginning of transaction batch execution, allowing L2 users to access it.

- **(5) & (6):** To complete the bridging process, users must call the claiming function of the L2 **PolygonZkEVMBridge.sol** contract and provide it with a Merkle inclusion proof of the leaf that was included previously. The contract will retrieve the GEMT root from the **PolygonZkEVMGlobalExitRootL2.sol** contract and verify the validity of the inclusion proof. If the inclusion proof is valid, the contract will finish the bridging process differently depending on the type of asset being bridged. If the inclusion proof is not valid, the transaction will be reverted.

L2 -> L1:

- **(1):** The user have to call the bridging function of the L2 **PolygonZkEVMBridge.sol** contract. Depending on the type of asset to be bridged the contract internals will act different. If the bridging request is valid, the contract will fill the L2 EMT leaf based on the characteristics of the request, add it into the tree, and compute the new L2 EMT root.

- **(2):** In the same L2 transaction the **PolygonZkEVMBridge.sol** contract will call the **PolygonZkEVMGlobalExitRootL2.sol** contract to update the new L2 EMT.

- **(3):** Eventually, the Aggregator will generate the Zero-Knowledge proof of computational integrity of the execution of the sequence of batches in which the bridging transaction is included. The new L2 EMT will be obtained from the L2 state that results from the execution.

- **(4):** The Aggregator will submit the resulting L2 EMT, along with the Zero-Knowledge proof to the L1 **PolygonZkEVM.sol** contract.

- **(5): PolygonZkEVM.sol** contract will verify the validity of the Zero-Knowledge proof and, if it is valid, the contract will call **PolygonZkEVMGlobalExitRoot.sol** contract to update the new L2 EMT root and consequently compute the new GEMT root.

- **(6) & (7):** To complete the bridging process, users must call the claiming function of the L1 **PolygonZkEVMBridge.sol** contract and provide it with a Merkle inclusion proof of the leaf that was included previously. The contract will retrieve the GEMT root from the **PolygonZkEVMGlibalExitRoot.sol** contract and verify the validity of the inclusion proof. If the inclusion proof is valid, the contract will finish the bridging process differently depending on the type of asset being bridged. If the inclusion proof is not valid, the transaction will be reverted.

3.2 PolygonZkEVMBridge.sol

PolygonZkEVMBridge.sol is the contract that acts as a bridging interface for users in a specific network, hence has an instance in each network. Has the storage slots required to hold the EMT of each network and all necessary functions to interact with it.

3.2.1 Bridging functions

bridgeAsset is the function used to bridge assets out to another network:

```
  function bridgeAsset(
        address token,
        uint32 destinationNetwork,
        address destinationAddress,
        uint256 amount,
        bytes calldata permitData
    )
```

bridgeAsset function arguments:

- **token:** ERC20 token address in origin network, if its "0x0000...0000" means that the user wants to bridge ether.
- **destinationNetwork:** Network ID of the destination network, must be different of the network ID in which is called the function, otherwise the transaction will revert.
- **destinationAddress:** Address that will receive the bridged tokens in the destination netwok.
- **amount:** Amount of tokens to bridge
- **permitData:** Signed permit data for ERC-20 tokens with EIP-2612 Permit extension, used to change an account's ERC-20 allowance, and permit the brige contract to transfer to himself the tokens to be bridged.

As can be seen in figure 4, since there are three different types of assets that can be bridged, there are three possible execution flows for the **bridgeAsset** function:

- **(1) The asset to be bridged is ether.**
- **(2) The asset is a representative ERC-20 token of an ERC-20 token from another network.**
- **(3) The asset is an ERC-20 token original from this network.**

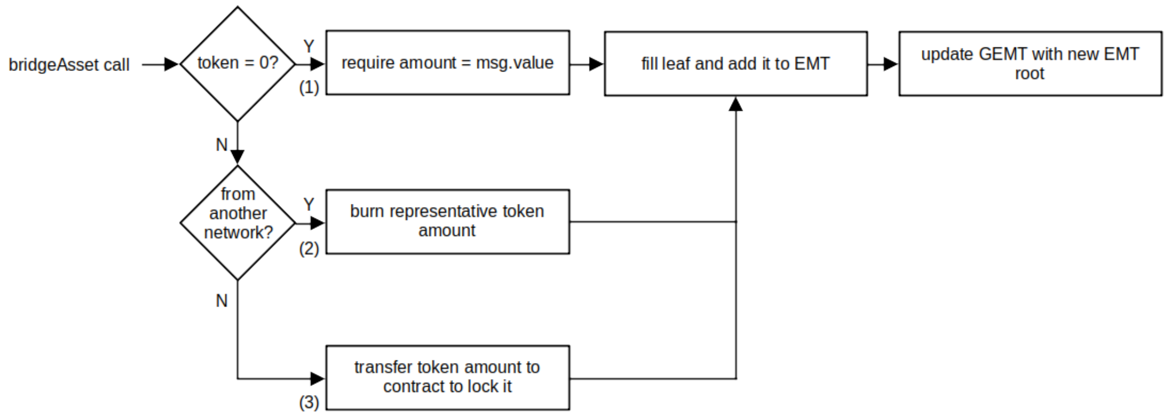

bridgeAsset call
token = 0?
Y
(1)
require amount = msg.value
fill leaf and add it to EMT
update GEMT with new EMT root
N
from another network?
Y
(2)
burn representative token amount
N
(3)
transfer token amount to contract to lock it

Figure 4: Possibles execution flows of **bridgeAsset** function depending on the type of the asset.

(1) The asset to be bridged is ether:

If the **token** argument equals to "0x0000...0000" means that the asset to bridge is ether, and the msg.value of the transaction shall match with the **amount** argument. The ether amount will be locked in the contract in order to proceed with the bridging process. Note that the ether in L1 and in L2 is treated in the same way, indeed, it is bridged with 1:1 exchange ratio and will be the gas payment (native) token in both networks. Since ether is original from L1 the leaf's parameter **originNetwork** will be set to L1 network ID.

(2) The asset is a representative ERC-20 token of an ERC-20 token from another network:

Representative ERC-20 tokens are managed by the **PolygonZkEVMBridge.sol** contract, which is responsible for minting and burning them. Additionally, the contract has a mapping named **wrappedTokenToTokenInfo** that acts as a list of representative ERC-20 token contracts deployed on this network. For each contract representative token deployed, this mapping will have an entry following the **TokenInformation** struct, with the address of representative token contract as the key.

```
struct TokenInformation {
  uint32 originNetwork;
  address originTokenAddress;
}
```

If the **token** argument value is present as a key in the **wrappedTokenToTokenInfo** mapping, it means that the token being bridged is a representative ERC-20 token of an ERC-20 token from another network. In order to proceed with the bridging process, a quantity of token corresponding to the **amount** argument will be burned, the bridge has the rights to burn the tokens without the user's allowance. Leaf's parameters, **originAddress** and **originNetwork**, will be taken from the **wrappedTokenToTokenInfo** mapping's corresponding entry.

(3) The asset is an ERC-20 token original from this network:

The last possible flow occurs when the assets to be bridged are ERC20 tokens originals from this network. In this case, in order to proceed with the bridging process, a quantity of token corresponding to the **amount** argument will be locked in the contract. Note that to successfully transfer the tokens to himself, the contract must have allowance over at least the quantity of tokens that the user wants to bridge. If the token contract supports EIP-2612 permit extension the allowance can be also done in the same transaction giving the signed **permitData** argument. Leaf's parameters, **originNetwork** and **originTokenAddress**, will be set to the current network ID and ERC-20 token contract address, respectively. The **metadataHash** parameter of the leaf will also be computed as follows. This parameter will ensure that the representative ERC-20 token contract is deployed with exactly the same ERC-20 token parameters in the destination network.

```
 metadataHash = keccak256(
  abi.encode(
    IERC20MetadataUpgradeable(token).name(),
    IERC20MetadataUpgradeable(token).symbol(),
    IERC20MetadataUpgradeable(token).decimals()
  )
);
```

The last execution step is common regardless the type of asset. The remaining leaf's parameters, **leafType**, will be set to 0 (type asset) and **destinationNetwork** and **destinationAddress** will be set according to the arguments of the **bridgeAsset** function call. First, a **BridgeEvent** event that contains all the information of the leaf will be emitted. Then, the new leaf will be included in the EMT. Finally, the GEMT contract will be called to update the new EMT root.

bridgeMessage is the function used to bridge a message to another network:

```
function bridgeMessage(
      uint32 destinationNetwork,
      address destinationAddress,
      bytes memory metadata
  )
```

bridgeMessage function arguments:

- **destinationNetwork:** Network ID of the destination network, must be different of the network ID in which is called the function, otherwise the transaction will revert.
- **destinationAddress:** Address in the destination network that will receive the bridged message.
- **metadata:** Message payload.

bridgeMessage function will directly emit a **BridgeEvent** event, will add a new leaf to the EMT and will call GEMT to update the new EMT root as **bridgeAsset** function does. The main difference is that the leaf it creates is of type message (leafType = 1), hence, **orginAddress** and **metadataHash** leaf's values will be the msg.sender and the hash of the message payload respectively. The user can bridge ether along with the message by simply adding the amount as the msg.value in the **bridgeMessage** function call transaction, this amount will be sent as value when **destinationAddress** will be called to receive the message in the destination network.

3.2.2 Claiming functions

Since L2 accounts, by default, do not have ether to pay transaction fees, when claiming bridged assets or messages from L1, L2 claiming transactions that call bridge claiming functions are funded by the protocol and will not require the payment of gas fees.

claimAsset is the function used to claim assets bridged from other networks:

```
function claimAsset(
      bytes32[] memory smtProof,
      uint32 index,
      bytes32 mainnetExitRoot,
      bytes32 rollupExitRoot,
      uint32 originNetwork,
      address originTokenAddress,
      uint32 destinationNetwork,
      address destinationAddress,
      uint256 amount,
      bytes memory metadata
  )
```

claimAsset function arguments:

- **smtProof:** Merkle proof, that is, the sibling nodes array needed to verify the leaf.
- **index:** Index of the leaf.
- **mainnetExitRoot:** L1 EMT root when the leaf has been included.
- **rollupExitRoot:** L2 EMT root when the leaf has been included.
- **originNetwork:** Origin Network ID, where the original asset belongs.

- **originTokenAddress:** ERC20 token address in origin network, if its 0x0000..0000 means that ether is being claimed.
- **destinationNetwork:** Network ID of the destination network, that is, the network where the call is being made.
- **destinationAddress:** Address that will receive the bridged tokens.
- **amount:** Amount of tokens that are claimed.
- **metadata:** If the asset being claimed is not original from the network where the call is being made the ABI encoded metadata of the original ERC-20 token (name, symbol, decimals) will be used as metadata. If is either Ether or a token original from the network where the call is being made, the value would be 0.

The **claimAsset** function will verify the validity of the leaf that corresponds to the arguments provided by the user.

To prevent replay attacks, measures should be taken to ensure that a given leaf can only be successfully verified once. **PolygonZkEVMBridge.sol** contract has a **claimedBitMap** mapping to store a nullifier bit for each leaf index that has already been successfully verified. As can be seen in figure 5, in order to optimize storage slots usage, each entry in the mapping will hold 256 nullifier bits for 256 already verified leaves.

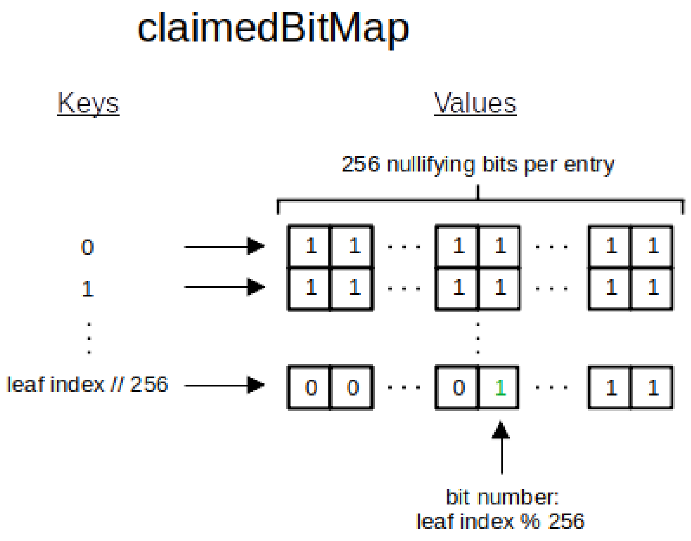


Figure 5: claimedBitMap nullifier mapping structure.

In order for a Merkle proof of a specific leaf to be considered valid, the following conditions must be met:

- The leaf's nullifying bit must be unset in **claimedBitMap** mapping.
- The leaf's **destinationNetwork** parameter must match the network ID in which the **claimAsset** function is being called.
- The GEMT root resulting of hashing **mainnetExitRoot** and **rollupExitRoot** arguments must already exist in the **PolygonZkEVMGlobalExitRoot.sol** contract.
- The Merkle proof must be valid, which means it should produce the expected GEMT root.

If the leaf is successfully verified, the leaf index will be nullified by setting its corresponding bit in **claimedBitMap** mapping, and then, as can be seen in figure 6, since there are three different types of assets that can be claimed, there are three possible execution flows for the **claimAsset** function:

- **(1) The asset to be claimed is ether.**
- **(2) The asset is an ERC-20 token original from this network.**
- **(3) The asset is a representative ERC-20 token of an ERC-20 token from another network.**

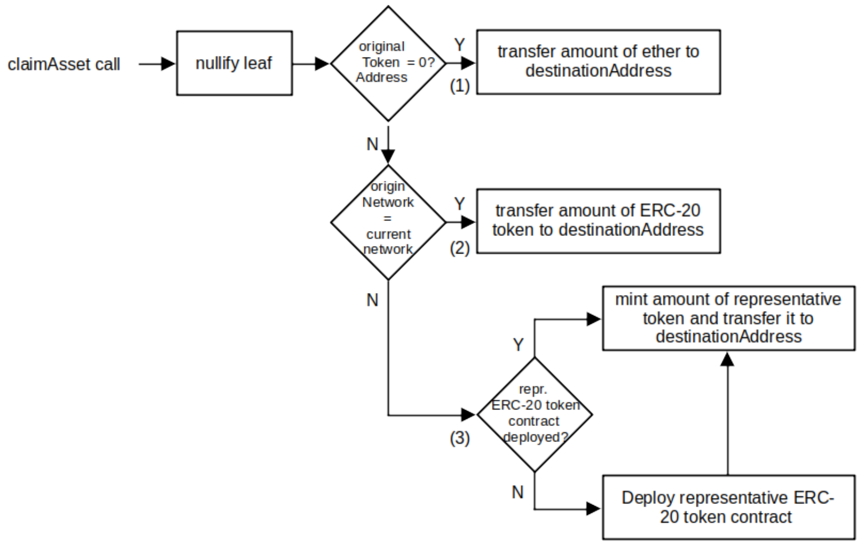

claimAsset call
nullify leaf
original Token Address = 0?
Y
(1)
transfer amount of ether to destinationAddress
N
origin Network = current network
Y
(2)
transfer amount of ERC-20 token to destinationAddress
N
mint amount of representative token and transfer it to destinationAddress
Y
repr. ERC-20 token contract deployed?
(3)
N
Deploy representative ERC-20 token contract

Figure 6: Possibles execution flows of **bridgeAsset** function depending on the type of the asset.

(1) The asset to be claimed is ether:
if the **originTokenAddress** argument equals to "0x0000...0000" means that the asset that is being claimed is ether, hence a transaction with the value of **amount** argument quantity of ether will be sent to **destinationAddress** argument address. Since ether cannot be minted on demand, **PolygonZkEVMBridge.sol** deployed on L2 has a balance of preminted 100000000000 ether as ether bridging liquidity. It is assumed that all ether in L2 is bridged form L1 so, L1's **PolygonZkEVMBridge.sol** does not need to have ether balance beforehand, every ether wei in L2 will have a backing ether wei blocked in L1 contract. Note that the preminted liquidity of L2 **PolygonZkEVMBridge.sol** does not have any inflationary effect on ether.

(2) The asset is an ERC-20 token original from this network:
If the **originNetwork** argument equals to the network ID where the call is being made, it means that the asset being claimed is original from that network and should already be locked in the balance of **PolygonZkEVMBridge.sol** contract. Hence, the **amount** argument quantity of ERC-20 tokens will be sent to **destinationAddress** argument address.

(3) The asset is a representative ERC-20 token of an ERC-20 token from another network:
The last possible flow occurs when the assets that is being claimed is an ERC20 token original from another network.

The **PolygonZkEVMBridge.sol** contract has a mapping called **tokenInfoToWrappedToken** which stores the addresses of representative ERC-20 token contracts deployed on the network. The salt used during the deployment of the contracts using the create2 function is used as a key in each entry of the mapping.

The salt is computed as follows from the **originNetwork** and **originTokeAddress** arguments:

```
  keccak256(
    abi.encodePacked(originNetwork, originTokenAddress)
  );
```

The contract will check if the representative ERC-20 token contract for the asset being claimed is present in the **tokenInfoToWrappedToken** mapping. If it is, it means that the representative ERC-20 token contract has already been deployed, and the quantity specified in the **amount** argument will be minted to the address specified in the **destinationAddress** argument.

If it is not present, a new representative ERC-20 token contract will be deployed using the create2 function and the previously computed salt. The use of the create2 function and this specific salt will deterministically bind the address of the representative token contract to the address of the original token contract on origin network. Then, the quantity specified in the **amount** argument will be minted to the address specified in the **destinationAddress** argument, and a **NewWrappedToken** event will be emitted.

A new entry in **tokenInfoToWrappedToken** and **wrappedTokenToTokenInfo** mappings will be added for the new representative ERC-20 token contract.

Finally, regardless of the kind of asset being claimed, a **ClaimEvent** event will be emitted.

claimMessage is the function used to claim messages bridged from other networks:

```
  function claimMessage(
        bytes32[] memory smtProof,
        uint32 index,
        bytes32 mainnetExitRoot,
        bytes32 rollupExitRoot,
        uint32 originNetwork,
        address originAddress,
        uint32 destinationNetwork,
        address destinationAddress,
        uint256 amount,
        bytes memory metadata
    )
```

claimMessage, as **claimAsset** function does, will try to verify the leaf given by the user, since the leaf format is the same, both functions will take the same arguments. And again, as **claimAsset** function does, if the leaf is successfully verified the leaf index will be nullified by setting its corresponding bit in **claimedBitMap** mapping. Then, low level call will be made to **destinationAddress** argument as the follows:

```
// Execute message
// Transfer ether
/* solhint-disable avoid-low-level-calls */
(bool success, ) = destinationAddress.call{value: amount}(
  abi.encodeCall(
    IBridgeMessageReceiver.onMessageReceived,
    (originAddress, originNetwork, metadata)
  )
);
```

As can be seen, call data is set in order to call a function **onMessageReceived** passing **originAddress**, **originNetwork**, **metadata** arguments, if the message contains ether, call value will be set to **amount** argument. **metadata** will be the message payload. Note that messaging service can be used to transfer ether to Externally owned accounts (EOAs), however the message cannot be interpreted by them hence the message payload will be unusable.

Finally, if the message sending succeeds, a **ClaimEvent** event will be emitted.

3.3 PolygonZkEVMGlobalExitRoot.sol

PolygonZkEVMGlobalExitRoot.sol is the L1 contract that computes and store each new GEMT root. In order for any added leaves to be verifiable in the future, every GEMT root needs to be stored. A mapping named **globalExitRootMap** holds all computed GEMT roots. L1 **PolygonZkEVMBridge.sol** will fetch the GETM roots form this mapping during the leaf verification process.

As can be seen in figure 7, the **updateExitRoot** function is used to update the values of EMT roots and compute a new GEMT root. If it is called by the **PolygonZkEVM.sol** contract, the L2 EMT root will be updated. If it is called by the **PolygonZkEVM-Bridge.sol** contract, the L1 EMT root will be updated. This function will accept only cal ls coming from **PolygonZkEVM.sol** or from **L1 PolygonZkEVMBridge.sol** contract.

```
function updateExitRoot(bytes32 newRoot) external
```

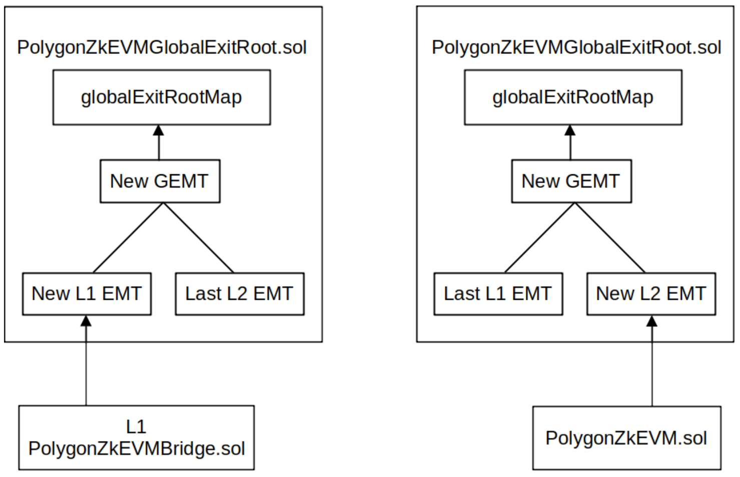


Figure 7: Two execution flows of **updateExitRoot** function.

When each L2 state transition is consolidated by the **PolygonZkEVM.sol** contract (through ZK proof verification given by the Aggregator), a new L2 EMT will be updated in the **PolygonZkEVMGlobalExitRoot.sol** contract through a call to the **updateEx-itRoot** function.

With each new leaf added to L1 EMT **PolygonZkEVMBridge.sol** contract, a new L1 EMT root will need updated in the **PolygonZkEVMGlobalExitRoot.sol** contract, so on **PolygonZkEVMBridge.sol** will call **updateExitRoot** function in order to update the new L1 EMT root.

Finally an event **UpdateGlobalExitRoot** will be emitted.

3.4 PolygonZkEVMGlobalExitRootL2.sol

PolygonZkEVMGlobalExitRootL2.sol is a "special" contract deployed in L2 that allows to L2 **PolygonZkEVMBridge.sol** instance to have access to GEMT roots history. Note that L2 **PolygonZkEVMBridge.sol** instance cannot directly call L1 **PolygonZkEVMGlobalExitRoot.sol** contract to query GEMT roots since it is deployed in another network.

PolygonZkEVMGlobalExitRootL2.sol as **PolygonZkEVMGlobalExitRoot.sol** has the mapping **globalExitRootMap** to store the GEMT roots, but instead of compute them by itself the **globalExitRootMap** mapping is synchronized between networks regarding the mapping in L1 **PolygonZkEVMGlobalExitRoot.sol** contract.

PolygonZkEVMGlobalExitRootL2.sol stores last L2 EMT root value updated by L2 **PolygonZkEVMBridge.sol** instance every time that a new leaf is added to L2 EMT.

PolygonZkEVMGlobalExitRootL2.sol is "special" because **globalExitRootMap** and **lastRollupExitRoot** storage slots are directly accessed by the zkEVM node SW during the transactions batches execution.

In order to allow to the L2 users to claim assets bridged, **globalExitRootMap** must be synchronized by zkEVM node at the beginning of the batch execution, then L2 **PolygonZkEVMBridge.sol** contract will have access to the valid GEMT roots to verify users claim transactions contained in the batch.

L2 EMT root is queried form **lastRollupExitRoot** variable by zkEVM node that acts as Aggregator after the execution of the batches. Then the L2 EMT root will be sent along with the L2 State transition proof to L1 **PolygonZkEVM.sol** contract, and if the proof verification succeeds, it will update the new L2 EMT root in L1 **PolygonZkEVMGlobalExitRoot.sol** contract and the new GEMT will be computed to allow to L1 **PolygonZkEVMBridge.sol** contract to have access to the valid GEMT roots to verify users claim transactions contained in the aggregated batches.

A Gas efficient append only sparse Merkle tree

A sparse Merkle tree is a Merkle tree of an intractable size that can be handled in an efficient way, making the assumption that it is almost empty, indeed, initially it is empty. It is considered empty when all leaves in it have the same zero (empty) value, thanks to this assumption it is possible to calculate the root by computing $log_2(n)$ hash operations where n is the number of leaves on the tree. Note that in a non-sparse tree we will need to compute $2n - 1$ hash operations to compute the root. During the computation of the empty tree root, in each level all nodes will take the same value, hence it is not needed to compute all subtrees in every level. Every value that takes a node in a specific level of the tree when it is empty is named Zero Hash and will represent a subtree of x zero leaves.

For example, for an empty sparce Merkle of 3 levels (8 leaves), the Zero Hash list will be as follows:

- level 0 node = 0
- level 1 node = H(0,0) = ZH_1
- level 2 node = H(ZH_1,ZH_1) = ZH_2
- level 3 node = H(ZH_2,ZH_2) = ZH_3 = Merkle root

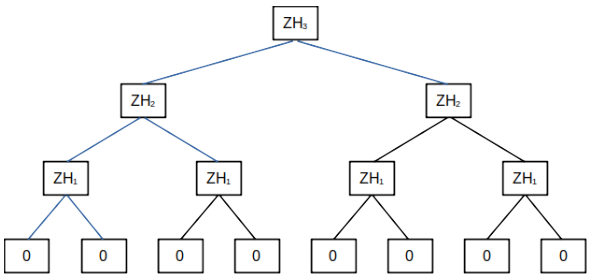


Figure 8: 8 leaves empty sparse Merkle tree.

To add new values to the tree we will use only $1 + log_2(n)$ hash operations, in addition to ZH evaluations if we do not have them precomputed. Indeed compute ZH on the fly is more gas efficient than reading storage slots with the precomputed values. Every new value added will fill one empty leaf of the tree.

Continuing with the previous example, to add first value (L_0) we will compute the following hash operations:

- $H(L_0) = B_0$
- $H(B_0,0) = B_1$
- $H(B_1,ZH_1) = B_2$
- $H(B_2,ZH_2) =$ Merkle root

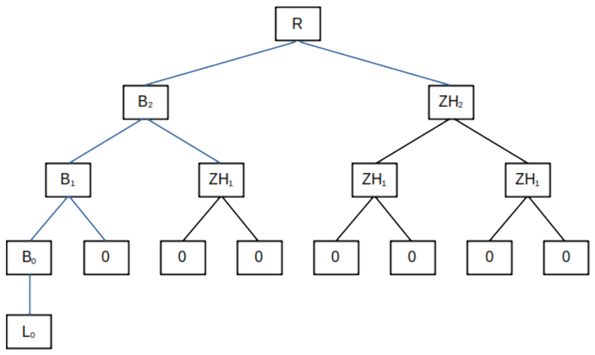


Figure 9: 8 leaves sparse Merkle tree first leave inclusion.

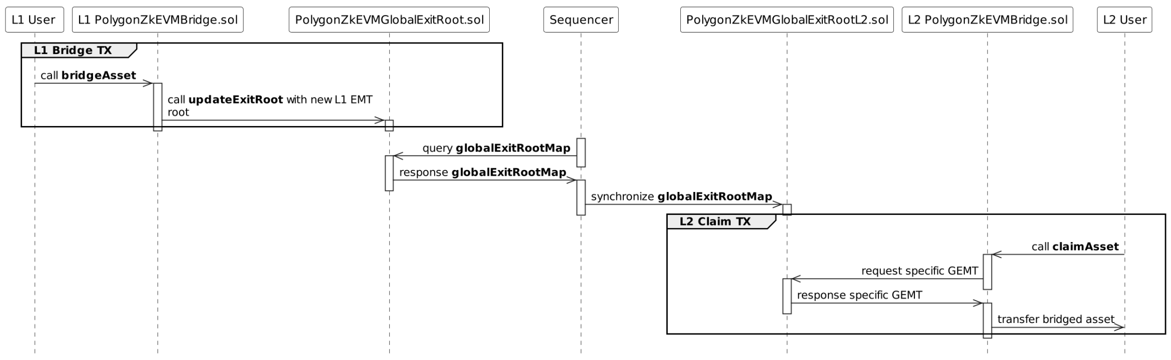
Note that we are computing a n leaves Merkle root, using only $1 + log_2(n)$ hash operations, and we will have the same situation for any leaf added consecutively. B_x (Branch) are all the values of the already computed subtrees, and it will be the storage slots needed to store the tree on-chain. So the whole storage slots needed to implement an incremental Sparce Merkle tree on-chain will be $1 + log_2(n)$.

Regarding the inclusion proof verification operations there is no difference between a regular Merkle tree and a sparse one, it can continue to be done efficiently using $1+log_2(n)$ hash operations. In addition, to generate the inlcusion proof, it will be nedded to have access to all the already included leves of the tree, nevertheless it is not needed to use storage slots since we have it available in the inclusion transaction calldata or by the emission of an event during the inclusion.

To sum up, SMT are massive Merkle trees that can be handled in an efficient way, and can be stored using a few storage slots. Efficiency in terms of on-chain computation means less transactions gas cost as well as low storage usage.

B Bridging flows

B.1 L1 -> L2



L1 User
L1 PolygonZkEVMBridge.sol
PolygonZkEVMGlobalExitRoot.sol
Sequencer
PolygonZkEVMGlobalExitRootL2.sol
L2 PolygonZkEVMBridge.sol
L2 User
L1 Bridge TX
call bridgeAsset
call updateExitRoot with new L1 EMT root
query globalExitRootMap
response globalExitRootMap
synchronize globalExitRootMap
L2 Claim TX
call claimAsset
request specific GEMT
response specific GEMT
transfer bridged asset

Figure 10: L1 -> L2 bridging flow.

B.2 L2 -> L1

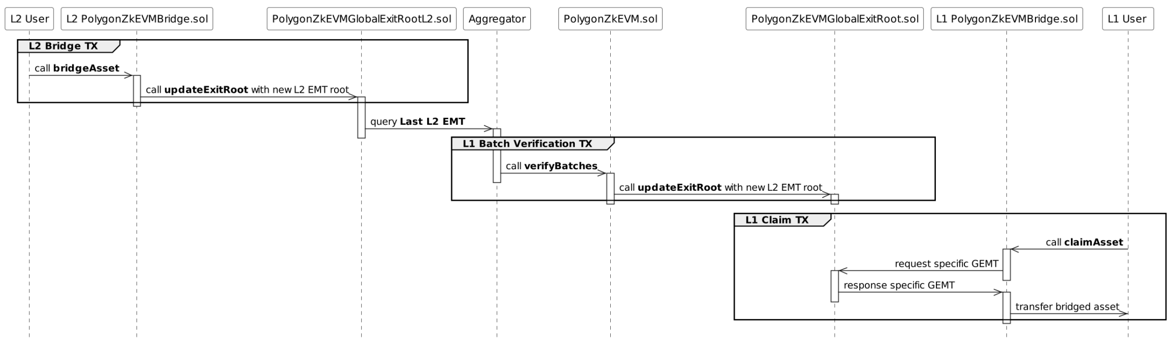


Figure 11: L2 -> L1 bridging flow.