

# The zkEVM Architecture

## Part IV: Exchanging Assets and Messages

---

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 1.1

December 12, 2023

Asset Exchange

Message Exchange

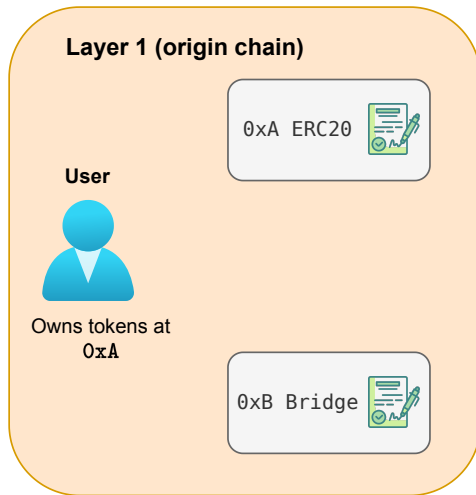
- The bridge allows transferring assets from one chain to another chain.
- Assets are tokens of some type, e.g. [ERC20](#) or [ERC721](#).
- The asset transferring is in fact an "illusion":

*Assets are not transferred, but temporarily locked on the origin chain while the same amount of equivalent assets/cryptocurrency are issued in the destination chain.*

- If the assets go back to the origin chain, they are **unlocked** in the original chain and **burned** in the destination chain.

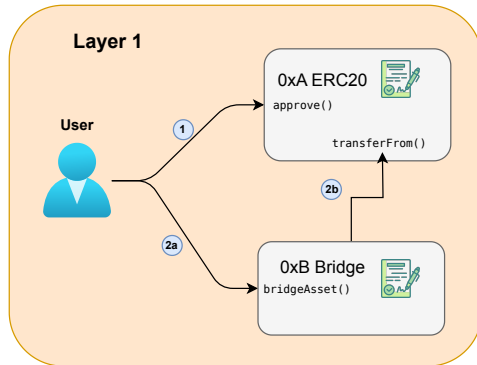
# Bridging ERC20 Tokens

- Let's start describing an **ERC20 token exchange**.
- Let's consider that **0xA** is an instance of an ERC20 token originally in L1 and we want to transfer tokens to another chain.
- For this purpose, first, the tokens have to be transferred to the **Bridge** smart contract (at **0xB**).



# ERC20 Token Transfer to Bridge (with approve)

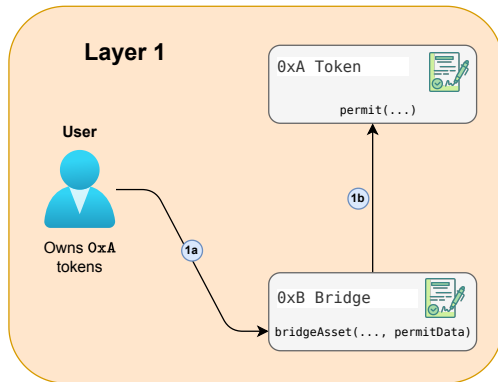
- To transfer the tokens to the bridge, the typical flow is to first send an approve transaction to the ERC20 contract.
- And then, the user sends a transaction to `bridgeAsset()` and the **Bridge** contract transfers the tokens to itself.



# ERC20 Token Transfer to Bridge (with permit)

Alternatively, the **permit** function, if available in the token smart contract, allows users to grant permission and to spend their tokens in a **single transaction**, avoiding the need of `approve+transferFrom`.

- The standard arguments to call `permit()` are the following:
  - **owner**: The user (0xE).
  - **spender**: The bridge (0xB).
  - **value**: Amount of tokens.
  - **deadline**: Deadline to spend the tokens.
  - **signature**: Signature of the owner including antireplay attacks measures (that is, it contains a nonce and the DOMAIN\_SEPARATOR)



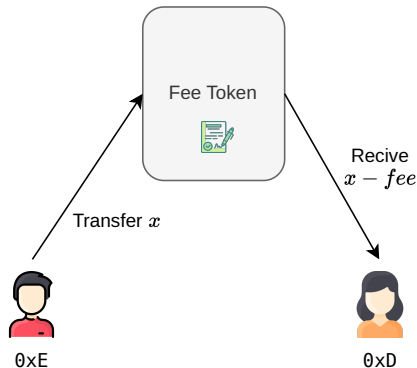
# Fee Tokens

There are a certain kind of tokens (called **fee tokens**) that, upon receiving a transfer request from a user, deduct a fee from the transferred value before the receptor receives the remaining value.

- In our case, we must store the correct amount ( $x - \text{fee}$ ) in the leaves of all our Exit Trees:

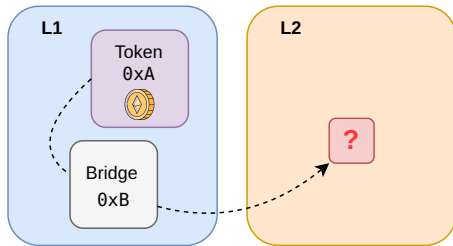
$$\text{leafAmount} = \text{balanceAfter} - \text{balanceBefore}$$

- In the case of non-fee tokens, **leafAmount** should equal **amount**.



# Transfer the Tokens to Destination Layer

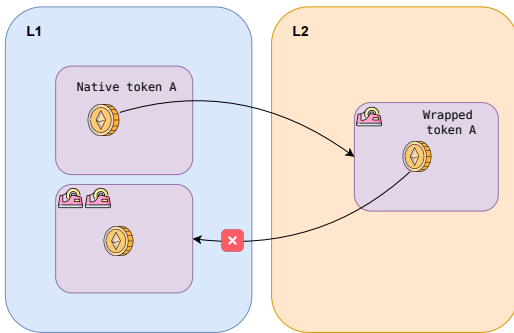
- Recall that tokens live in a instance of a smart contract deployed in its corresponding Layer.
- Therefore, when transferring tokens from one layer to another one, we might need to create an instance of a token contract in the destination layer to hold the tokens (if this instance does not already exist).
- These contracts are called contracts of **wrapped tokens**.





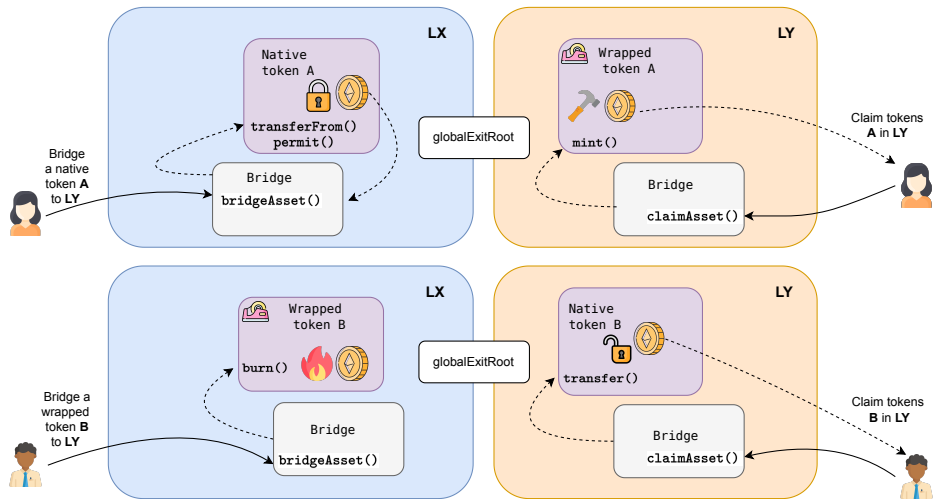
# Avoiding Loops of Wrappings

- Another thing that we need to manage is the possible creation of *loops of wrappings*.
- That is to say, creating a new wrapped token smart contract in the origin Layer.

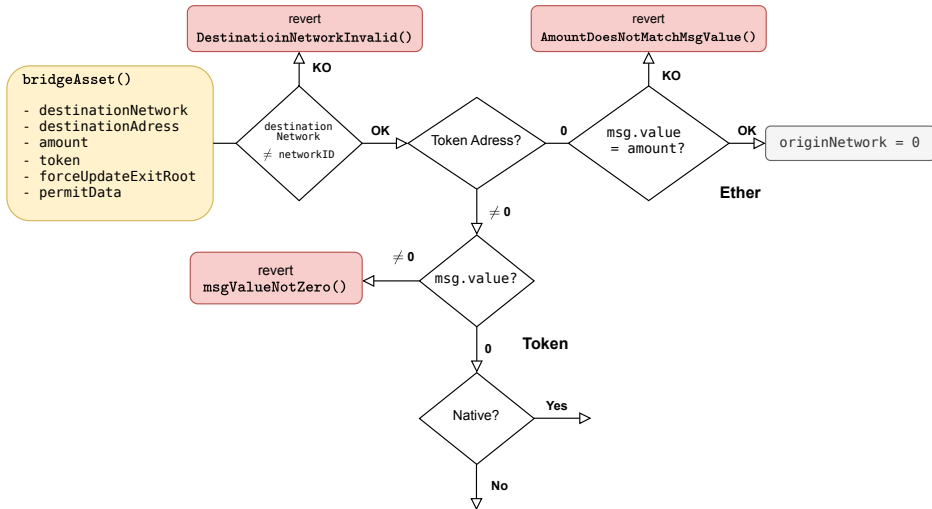


- To achieve that, we always set **originNetwork** to be the network of the native token.
- In the case of Ether, the origin network is L1.

# Overview of Asset Transfer with Bridge-Claim



# Flow of bridgeAsset(): Part I



# Native Tokens vs Wrapped Tokens i

- In the **Bridge** smart contract, we have a wrapped token information struct, which contains the following data:

```
1 struct TokenInformation{  
2     uint32 originNetwork;  
3     address originTokenAddress;  
4 }
```

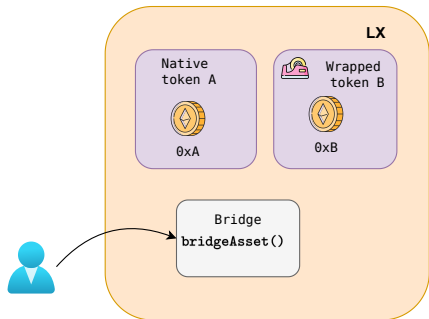
- We use **TokenInformation** as the value of the **wrappedTokenToTokenInfo** mapping, keyed by the **wrappedTokenAddress**.



# Native Tokens vs Wrapped Tokens ii

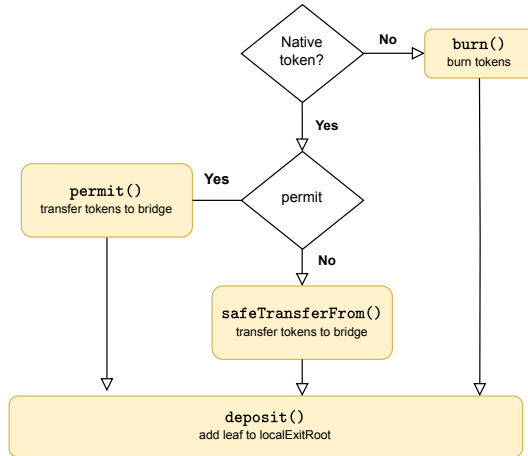
Using the token address and the `wrappedTokenToTokenInfo` mapping, we check if the token is native or wrapped.

`TokenInformation.originTokenAddress` must be zero if it is native and non-zero if is wrapped.

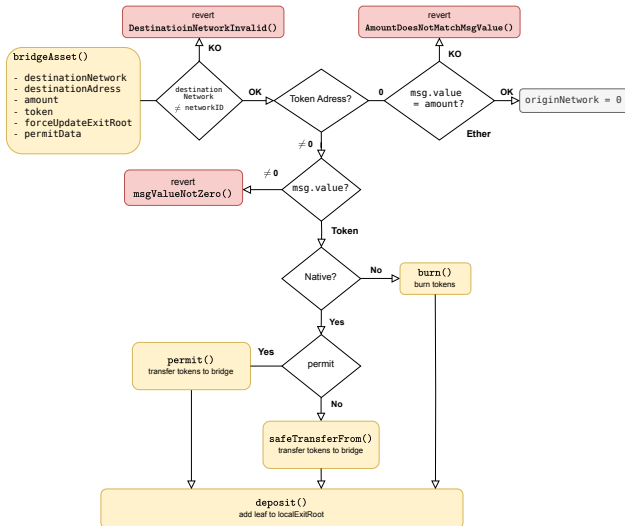
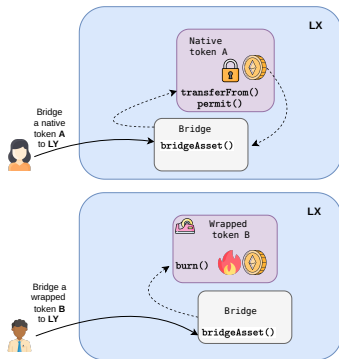


- If bridged token is `0xA` then  
`wrappedTokenToTokenInfo[0xA].originTokenAddress = 0.`
- If bridged token `0xB` then  
`wrappedTokenToTokenInfo[0xB].originTokenAddress  $\neq$  0.`

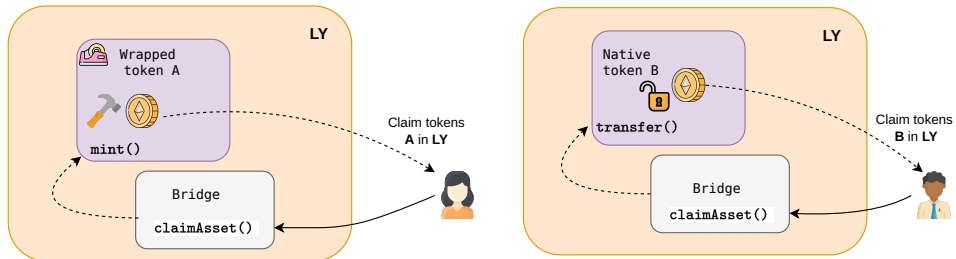
## Flow of `bridgeAsset()`: Part II



# Complete Flow of bridgeAsset()

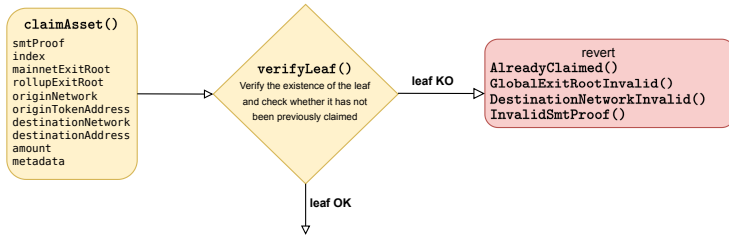


# Claim Assets





# Flow of `claimAsset()`: Part I



## Storing binary-valued maps: BitMap Optimization i

- Recall that, when nullifying claims, we need to store an associative map `uint256 → bool`.
- For example: `3 ↦ true`, `7 ↦ false`, etc.
- We want to do it in the most efficient way in Solidity.
- As a first try, we could use an `uint256 → bool` mapping directly, but in Solidity this uses 256 bits for each value in the mapping (despite being a boolean!).
- An optimization would be using a bit map, which consists in grouping 256 indexes and use its bit-wise values to construct a `uint256`.
- For this purpose, the first step is to change the mapping definition to `uint256 → uint256`.
- Then, each index is divided in a word position (`wordPos`) and a bit position (`bitPos`).

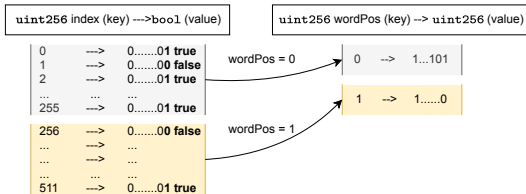
# Storing binary-valued maps: BitMap Optimization ii

- **Word positions** of each index specifies which key of the new mapping contains the wanted value

$$\text{wordPos} = \lfloor \text{index} / 256 \rfloor.$$

- On the other side, the **bit positions** specifies at which bit position of the corresponding `uint256` value (uniquely determined by `wordPos`) we can find the wanted value.

$$\text{bitPos} = \text{index} \bmod 256.$$



# BitMap Optimization Example

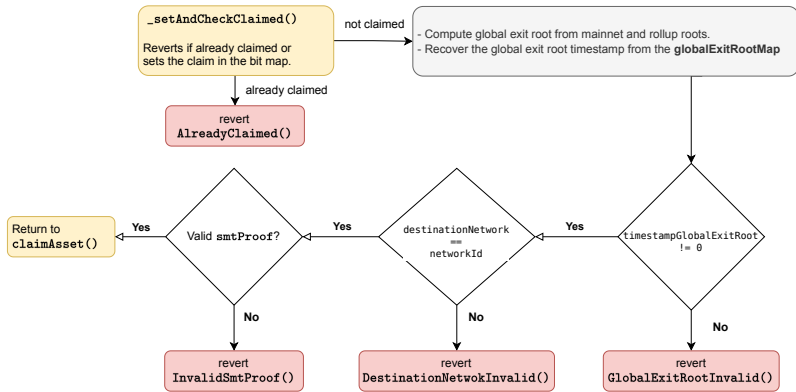
- For example, consider the mapping

$0 \mapsto \text{false}, 1 \mapsto \text{true}, \dots, 519 \mapsto \text{true}, 520 \mapsto \text{false}, 521 \mapsto \text{false}, \dots$

- We want to retrieve in the new mapping the value for the index 520, which is **false** in the original mapping.
- We compute **wordPos** and **bitPos** as specified:
  - **wordPos**:  $\lfloor 520/256 \rfloor = 2$ .
  - **bitPos**:  $520 \bmod 256 = 8$ .
- Hence, in order to look for the corresponding boolean of index 520, we must use 2 as the key of the mapping and then get the 8th bit of the returned value (which will be 0, corresponding to **false**).

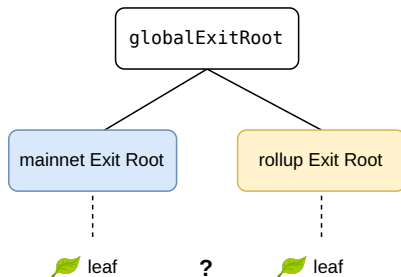
$0 \mapsto 1\dots 110, 1 \mapsto 1\dots 000, 2 \mapsto 1\dots 1001011011.$

# Flow of `verifyLeaf()`

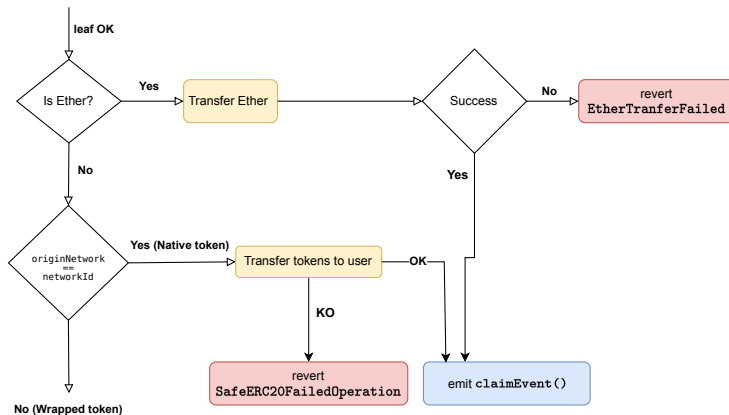


# Verifying Leaves of Exit Trees

With the Merkle proof (the `smtProof` parameter), we need to check that the leaf is included in the appropriate Exit Tree:



## Flow of `claimAsset()`: Part II



Note. The bridge smart contract in L2 has a big amount of ETH to do the transfers.

Let's consider that we are in a situation in which claimed tokens must be transferred to a wrapped token smart contract.

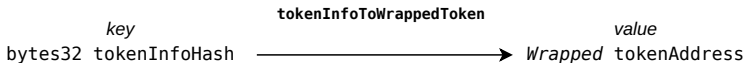
In this case:

- a) We need to check whether the corresponding contract instance exists or not.
- b) If the instance for the wrapped token does not exist, we must create it.



## Wrapped Tokens ii

- For a), we use a mapping to store the information of wrapped token instances:



```
1 bytes32 tokenInfoHash = keccak256(  
2     abi.encodePacked(originNetwork, originTokenAddress)  
3 );
```

- For b), creating the instances of tokens for wrapped tokens, we use CREATE2 to be able to predict the deployment address.

## CREATE vs. CREATE2

- With the CREATE opcode the address of the new smart contracts is calculated as follows:

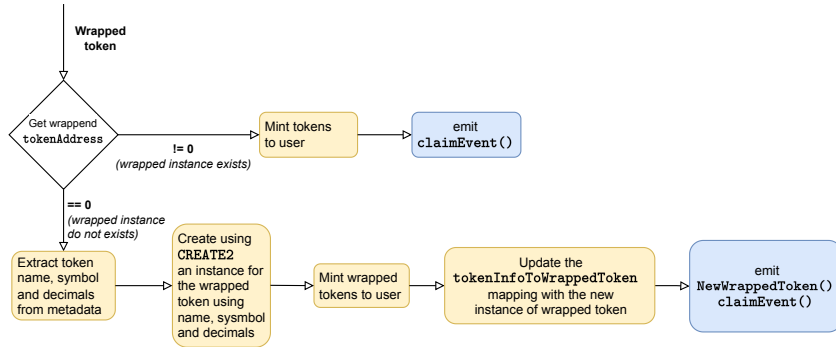
$$\text{new\_address} = \text{hash}(\text{sender}, \text{nonce})$$

- With CREATE2, the address of the new smart contracts is calculated as follows:

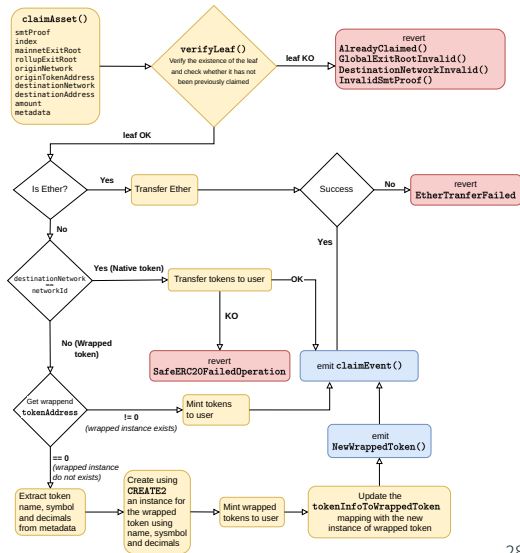
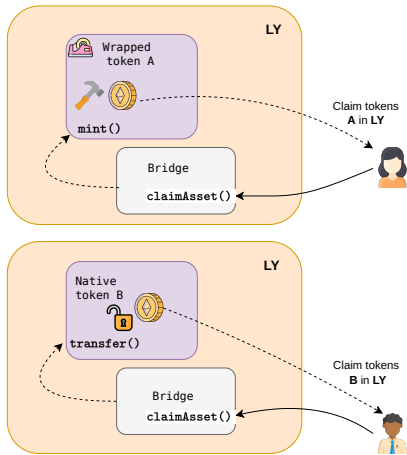
$$\text{new\_address} = \text{hash}(\text{0xFF}, \text{sender}, \text{salt}, \text{creationBytecode}, [\text{args}])$$

<https://docs.soliditylang.org/en/latest/control-structures.html#salted-contract-creations-create2>

## Flow of ClaimAsset() Part III



# Complete Flow of `claimAsset()`



## Related **view** Functions

- **preCalculateWrappedAddress()**: returns the precalculated address of a wrapper using the token information.

```
1 function precalculatedWrapperAddress(  
2     uint32 originNetwork,  
3     address originTokenAddress,  
4     string calldata name,  
5     string calldata symbol,  
6     uint8 decimals  
7 ) external view returns (address);
```

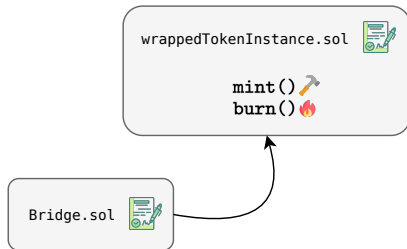
Note. These addresses can be pre-calculated thanks to using **CREATE2**.

- **getTokenWrappedAddress()**: returns the address of a wrapper contract using the token information if already exists.

```
1 function getTokenWrappedAddress(  
2     uint32 originNetwork,  
3     address originTokenAddress  
4 ) external view returns (address);
```

# Wrapped Token Instances

- Our wrapped token contracts are ERC20 contracts.
- The code is in [lib/TokenWrapped.sol](#).
- Mint and burn are only allowed to be called from the bridge contract.



# Permit in TokenWrapped i

- The token contract implements the permit function of [EIP2612](#), which performs a token approval with the provided permit data.
- In an **EIP2612** contract, we have to include:
  - The hash of the type of the domain used.
  - The hash of the signature of the permit function used.

TokenWrapped.sol



→ `permit(owner,spender,value,deadline,v,r,s)`

- The signature  $\sigma = (v, r, s)$  is performed over the hash of the following data:

`"\x19\x01" + DOMAIN_SEPARATOR + dataHash`

```
1 bytes32 digest = keccak256(abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR(), hashStruct));
```

- The first string `"\x19\x01"` is used to indicate that we **are not signing a transaction**.
- The `DOMAIN_SEPARATOR` is defined according [EIP712](#) and it should be unique to the contract and chain to prevent replay attacks from other domains.



# The EIP 712 Domain Separator

- The usual way to compute the DOMAIN\_SEPARATOR is as a hash of a name, version, chainId and a verifyingContract:

```
1 // Domain typehash
2 bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)");
```

- There exists a view function to retrieve the DOMAIN\_SEPARATOR for the corresponding chainID:

```
1 function DOMAIN_SEPARATOR() public view returns (bytes32) {
2     return block.chainid == deploymentChainId ? _DEPLOYMENT_DOMAIN_SEPARATOR : _calculateDomainSeparator(block.chainid);
3 }
4 function _calculateDomainSeparator(uint256 chainId) private view returns (bytes32) {
5     return keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name())), keccak256(bytes(VERSION)), chainId, address(this)));
6 }
```

# The Data Signed for the Permit Function

- The `hashStruct` is the hash of the arguments of the permit function together with a hash that identifies the signature of the permit function used:

```
1  // Permit typehash
2  bytes32 public constant PERMIT_TYPEHASH = keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256 deadline)");
3
4  bytes32 hashStruct = keccak256(abi.encode(
5      PERMIT_TYPEHASH,
6      owner,
7      spender,
8      value,
9      nonces[owner]++,
10     deadline
11 ));
12
```

- The `nonces` mapping is used to avoid attacks that try to replay the same permit message of the same owner.
- Finally, with an `ecrecover`, the permit function checks that the signature provided in the arguments corresponds to the signed hashed data.

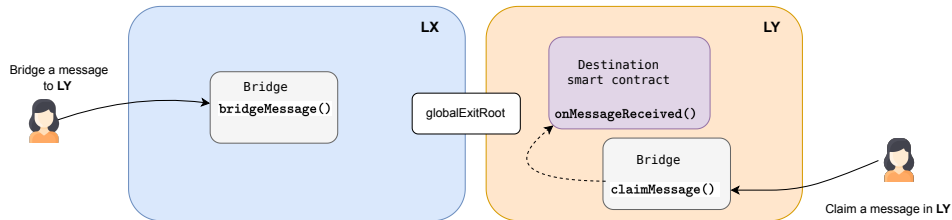
Asset Exchange

Message Exchange

# Leaves of Exit Trees (Review)

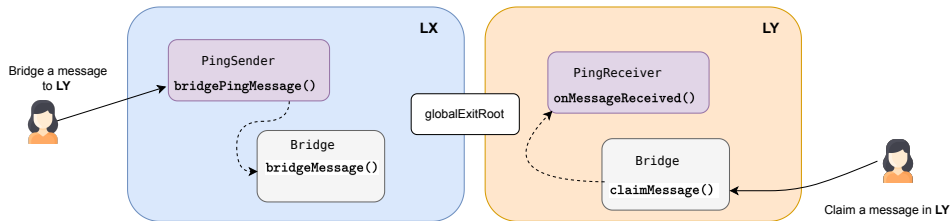
- **leafType** used to identify whether the leaf is an asset or a message:
  - Asset: value is 0.
  - Message: value is 1.
- **originNetwork**: the identifier (**chainId**) of origin layer of the exchange.
- **originAddress**: if it is an asset exchange, it is the address of the token contract.  
If it is a message exchange, it is the source address of the bridge call.
- **destinationNetwork**: the identifier of the destination layer (**chainId**) of the exchange.
- **destinationAddress**: is the account receiving the asset or the address of the smart contract if it is a message exchange.
- **leafAmount**: amount of asset exchanged (Ether or Tokens).
- **bytes32 metadataHash**: the hash of the metadata.
  - Asset: the metadata is the name, symbol and decimals of the token.
  - Message: the metadata is the **calldata** for calling the **onMessageReceived()** function.

# Bridge-Claim for Messages



Unlike bridging assets, which is closely managed by the **Bridge** smart contract, **messages** are a more low level primitive which you can use to **create any logic that you want**.

# Example of Bridge-Claim for Messages: Ping/Pong



<https://github.com/0xPolygonHermes/code-examples/tree/main/pingPongExample>

## More Examples of Bridge-Claim for Messages

- You can find more examples, e.g. an implementation for custom ERC20 or NFT bridging, at:

<https://github.com/0xPolygonHermes/code-examples>

- The DAI project has a custom implementation for bridging ERC20:

<https://github.com/pyk/zkevm-dai>

- Also the USDC project has a custom implementation for bridging ERC20 in which the main idea is to be able to manage white/black lists of addresses.