Debugging Messages

The V2 protocol now splits the verification and contract logic execution of messages into two separate, distinct phases:

Verified : the destination chain has received verification from all configuredDVNs and the message nonce has been committed to theEndpoint 's messaging channel.

Delivered : the message has been successfully executed by theExecutor .

Because verification and execution are separate, LayerZero can provide specific error handling for each message state.

Message Execution

When your message is successfully delivered to the destination chain, the protocol attempts to execute the message with the execution parameters defined by the sender. Message execution can result in two possible states:

- Success
- : If the execution is successful, an event (PacketReceived
-) is emitted.
- Failure
- : If the execution fails, the contract reverses the clearing of the payload (re-inserts the payload) and emits an event (LzReceiveAlert
-) to signal the failure.
-
  - Out of Gas
-
  - : The message fails because the transaction that contains the message doesn't provide enough gas for execution.
-
  - Logic Error
-
  - : There's an error in either the contract code or the message parameters passed that prevents the message from being executed correctly.

Retry Message

Because LayerZero separates the verification of a message from its execution, if a message fails to execute due to either of the reasons above, the message can be retried without having to resend it from the origin chain.

This is possible because the message has already been confirmed by the DVNs as a valid message packet, meaning execution can be retried at anytime, by anyone.

Here's how an OApp contract owner or user can retry a message:

- Using LayerZero Scan
- : For users that want a simple frontend interface to interact with, LayerZero Scan provides both message failure detection and in-browser message retrying.
- CallinglzReceive
- Directly
- : If message execution fails, any user can retry the call on the Endpoint'slzReceive
- function via the block explorer or any popular library for interacting with the blockchain likeethers
- ,viem
- , etc.

function

lzReceive (Origin calldata _origin , address _receiver , bytes32 _guid , bytes

calldata _message , bytes

calldata _extraData)

external

payable

{ // clear the payload first to prevent reentrancy, and then execute the message _clearPayload (_receiver , _origin . srcEid , _origin . sender , _origin . nonce , abi . encodePacked (_guid , _message)) ; ILayerZeroReceiver (_receiver) . lzReceive { value : msg . value } (_origin , _guid , _message , msg . sender , _extraData) ; emit

PacketDelivered (_origin , _receiver) ; }

Skipping Nonce

Occasionally, anOApp delegate may want to cancel the verification of an in-flight message. This might be due to a variety of reasons, such as:

- Race Conditions
- : conditions where multiple transactions are being processed in parallel, and some might become invalid or redundant before they are processed.
- Error Handling
- : In scenarios where a message cannot be delivered (for example, due to being invalid or because prerequisites are not met), the skip function provides a way to bypass it and continue with subsequent messages.

By allowing the OApp to skip the problematic message, the OApp can maintain efficiency and avoid getting stuck by a single bottleneck.

caution Theskip function should be used only in instances where either messageverification fails or must be stopped, not messageexecution . LayerZero provides separate handling for retrying or removing messages that have successfully been verified, but fail to execute. danger It is crucial to use this function with caution because once a payload is skipped, it cannot be recovered.

An OApp's delegate can call theskip method via the Endpoint to stop message delivery:

/// @dev the caller must provide _nonce to prevent skipping the unintended nonce /// @dev it could happen in some race conditions, e.g. intent to skip nonce 3, but nonce 3 was consumed before this transaction was included in the block /// @dev NOTE: only allows skipping the next of the effective inbound nonce (from the inboundNonce() function). if the Oapp wants to skips a delivered message, it should call the clear() function and ignore the payload instead /// @dev after skipping, the lazyInboundNonce is set to the provided nonce, which makes the inboundNonce also the provided nonce

function skip (address _oapp ,

//the Oapp address uint32 _srcEid ,

//source chain endpoint id bytes32 _sender ,

//the byte32 format of sender address uint64 _nonce // the message nonce you wish to skip to)

external

{ _assertAuthorized (_oapp) ;

if

(_nonce !=

inboundNonce (_oapp , _srcEid , _sender)

+

1)

revert Errors . InvalidNonce (_nonce) ;

//Skipping ahead of this nonce. lazyInboundNonce [_oapp] [_srcEid] [_sender]

= _nonce ; emit

InboundNonceSkipped (_srcEid , _sender , _oapp , _nonce) ; } Example for callingskip

1. Set up Dependencies and Define the ABI

// using ethers v5 const

{ ethers }

=

require ('ethers') ; const skipFunctionABI =

['function skip(address _oapp,uint32 _srcEid, bytes32 _sender, uint64 _nonce)' ,] ; 1. Configure the Contract Instance

// Example Endpoint Address const

ENDPOINT_CONTRACT_ADDRESS

=

'0xb6319cC6c8c27A8F5dAF0dD3DF91EA35C4720dd7' ;

const provider =

new

ethers . providers . JsonRpcProvider (YOUR_RPC_URL) ; const signer =

new

ethers . Wallet (YOUR_PRIVATE_KEY , provider) ; const endpointContract =

new

ethers . Contract (ENDPOINT_CONTRACT_ADDRESS , skipFunctionABI , signer) ; 1. Prepare Function Parameters

// Example Oapp Address const oAppAddress =

'0x123123123678afecb367f032d93F642f64180aa3' ;

// Parameters for the skip function const srcEid =

50121 ;

// srcEid example

// padding an example address to bytes32 const sender = ethers . zeroPadValue (0x5FbDB2315678afecb367f032d93F642f64180aa3 ,

32) ; const nonce =

3 ;

// uint64 nonce example

const tx =

await endpointContract . skip (oAppAddress , srcEid , sender , nonce) ; 1. Send the Transaction

const tx =

await endpointContract . skip (oAppAddress , srcEid , sender , nonce) ; await tx . wait ( ) ;

Clearing Message

As a last resort, an OApp contract owner may want to force eject a message packet, either due to an unrecoverable error or to prevent a malicious packet from being executed:

- When logic errors exist and the message can't be retried successfully.
- When a malicious message needs to be avoided.

Using theclear Function : This function exists on the Endpoint and allows an OApp contract delegate to burn the message payload so it can never be retried again.

danger It is crucial to use this function with caution because once a payload is cleared, it cannot be recovered. /// @dev Oapp uses this interface to clear a message. /// @dev this is a PULL mode versus the PUSH mode of lzReceive /// @dev the cleared message can be ignored by the app (effectively burnt) /// @dev authenticated by oapp /// @param _origin the origin of the message /// @param _guid the guid of the message /// @param _message the message

function

clear (address _oapp ,

//the Oapp address Origin calldata _origin ,

// The Origin struct of the message. bytes32 _guid ,

// The unique identifier of the message. This can be fetched from the arguments oLzReceive. bytes

calldata _message // The bytes message you sent on the source chain. This can be fetched from the arguments oLzReceive.)

external

{ _assertAuthorized (_oapp) ;

bytes

memory payload = abi . encodePacked (_guid , _message) ; _clearPayload (_oapp , _origin . srcEid , _origin . sender , _origin . nonce , payload) ; emit

PacketDelivered (_origin , _oapp) ; } Example for callingclear

1. Set up Dependencies and Define the ABI

// using ethers v5 const

{ ethers }

=

require ('ethers') ; const clearFunctionABI =

[{ inputs :

[{ components :

[{ internalType :

'uint32' ,

name :

'srcEid' ,

type :

'uint32' } , { internalType :

'bytes32' ,

name :

'sender' ,

type :

'bytes32' } , { internalType :

'uint64' ,

name :

'nonce' ,

type :

'uint64' } ,] , internalType :

'struct Origin' , name :

'_origin' , type :

'tuple' , } , { internalType :

'bytes32' ,

name :

'_guid' ,

type :

'bytes32' } , { internalType :

'bytes' ,

name :

'_message' ,

type :

'bytes' } ,] , name :

'clear' , outputs :

[ ] , stateMutability :

'nonpayable' , type :

'function' , } ,] ; 1. Configure the Contract Instance

// Example Endpoint Address const

ENDPOINT_CONTRACT_ADDRESS

=

'0xb6319cC6c8c27A8F5dAF0dD3DF91EA35C4720dd7' ;

const provider =

new

ethers . providers . JsonRpcProvider (YOUR_RPC_URL) ; const signer =

new

ethers . Wallet (YOUR_PRIVATE_KEY , provider) ; const endpointContract =

new

ethers . Contract (ENDPOINT_CONTRACT_ADDRESS , clearFunctionABI , signer) ; 1. Prepare Function Parameters

// Example Oapp Address const oAppAddress =

'0x123123123678afecb367f032d93F642f64180aa3' ;

// Parameters for the skip function const origin =

{ srcEid :

10111 ,

// example source chain endpoint Id sender : ethers . zeroPadValue (0x5FbDB2315678afecb367f032d93F642f64180aa3 ,

32) ,

// bytes32 representation of an address nonce :

3 ,

// example nonce } ;

const _guid =

'0x0af522cbed56c0e67988a3eab0e83fc576d501659ffe7743ffa4a0a76b40419d' ;

// example _guid const _message =
'0x0000000000000000000000000000000000000000000000000000000000000001000000000000000000000000000000000000000000000000000000000000000400000000000000000000000000000000
;

//example _message 1. Send the Transaction

const tx =

await endpointContract . clear (oAppAddress , origin , _guid , _message) ; await tx . wait ( ) ;

Nilify and Burn

These two functions exist in the Endpoint contract and are used in very specific cases to avoid malicious acts by DVNs. These two functions are infrequently utilized and serve as precautionary design measures.

tip nilify andburn are called similarly toclear andskip , refer to those examples if needed.

nilify

/// @dev Marks a packet as verified, but disallows execution until it is re-verified. /// @dev Reverts if the provided _payloadHash does not match the currently verified payload hash. /// @dev A non-verified nonce can be nilified by passing EMPTY_PAYLOAD_HASH for _payloadHash. /// @dev Assumes the computational intractability of finding a payload that hashes to bytes32.max. /// @dev Authenticated by the caller function

nilify (address _oapp ,

// The Oapp address uint32 _srcEid ,

// The source Endpoint Id bytes32 _sender ,

// The bytes32 representation of the source chain's Oapp address uint64 _nonce ,

// The nonce you want to nilify bytes32 _payloadHash // The targeted payload hash)

external Thenilify function is designed to transform a non-executed payload hash into NIL value (0xFFFFFF...). This transformation enables the resubmission of these NIL packets via the MessageLib back into the endpoint, providing a recovery mechanism from disruptions caused by malicious DVNs.

burn

/// @dev Marks a nonce as unexecutable and un-verifiable. The nonce can never be re-verified or executed. /// @dev Reverts if the provided _payloadHash does not match the currently verified payload hash. /// @dev Only packets with nonces less than or equal to the lazy inbound nonce can be burned. /// @dev Reverts if the nonce has already been executed. /// @dev Authenticated by the caller function

burn (address _oapp ,

// The Oapp address uint32 _srcEid ,

// The source Endpoint Id bytes32 _sender ,

// The bytes32 representation of the source chain's Oapp address uint64 _nonce ,

// The nonce you want to nilify bytes32 _payloadHash // The targeted payload hash)

external Theburn function operates similarly to theclear function with two key distinctions:

1. The OApp is not required to be aware of the original payload
2. The nonce designated for burning must be less than thelazyInboundNonce

This function exists to avoid malicious DVNs from hiding the original payload to avoid the message from being clearedEdit this page

Previous LayerZero Scan Next Error Codes