Sending Data Between L1 and L2

Smart contracts on L1 (Ethereum) can interact with smart contracts on L2 (OP Mainnet) through a process called "bridging". This page explains how bridging works, how to use it, and what to watch out for.

This is a high-level overview of the bridging process. For a step-by-step tutorial on how to send data between L1 and L2, check out theSolidity tutorial .

Understanding Contract Calls

It can be easier to understand bridging if you first have a basic understanding of how contracts on EVM-based blockchains like OP Mainnet and Ethereum communicate within thesame network. The interface for sending messagesbetween Ethereum and OP Mainnet is designed to mimic the standard contract communication interface as much as possible.

Here's how a contract on Ethereum might trigger a function within another contract on Ethereum:

contract MyContract { function

doTheThing (address myContractAddress ,

uint256 myFunctionParam) public { MyOtherContract (myContractAddress). doSomething (myFunctionParam); } } Here,MyContract.doTheThing triggers a call toMyOtherContract.doSomething . Under the hood, Solidity is triggering the code forMyOtherContract by sending anABI encoded(opens in a new tab) call for thedoSomething function. A lot of this complexity is abstracted away to simplify the developer experience. Solidity also has manual encoding tools that allow us to demonstrate the same process in a more verbose way.

Here's how you might manually encode the same call:

contract MyContract { function

doTheThing (address myContractAddress ,

uint256 myFunctionParam) public { myContractAddress. call (abi. encodeCall (MyOtherContract.doSomething , (myFunctionParam))); } } Here you're using thelow-level "call" function(opens in a new tab) and one of theABI encoding functions built into Solidity(opens in a new tab) . Although these two code snippets look a bit different, they're doing the exact same thing. Because of limitations of Solidity,the OP Stack's bridging interface is designed to look like the second code snippet .

Basics of Communication Between Layers

At a high level, the process for sending data between L1 and L2 is pretty similar to the process for sending data between two contracts on Ethereum (with a few caveats). Communication between L1 and L2 is made possible by a pair of special smart contracts called the "messenger" contracts. Each layer has its own messenger contract which serves to abstract away some lower-level communication details, a lot like how HTTP libraries abstract away physical network connections.

We won't get intotoo much detail about these contracts here. The most important thing that you need to know is that each messenger contract has asendMessage function that allows you to send a message to a contract on the other layer.

function

sendMessage (address

_target , bytes

memory

_message , uint32

_minGasLimit) public ; ThesendMessage function has three parameters:

1. Theaddress _target
2. of the contract to call on the other layer.
3. Thebytes memory _message
4. calldata to send to the contract on the other layer.
5. Theuint32 _minGasLimit
6. minimum gas limit that can be used when executing the message on the other layer.

This is basically equivalent to:

address (_target).call{gas : _gasLimit}(_message); Except, of course, that you're calling a contract on a completely different network.

This is glossing over a lot of the technical details that make this whole thing work under the hood, but this should be enough to get you started. Want to call a contract on OP Mainnet from a contract on Ethereum? It's dead simple:

// Pretend this is on L2 contract MyOptimisticContract { function

doSomething (uint256 myFunctionParam) public { // ... some sort of code goes here } }

// And pretend this is on L1 contract MyContract { function

doTheThing (address myOptimisticContractAddress ,

uint256 myFunctionParam) public { messenger. sendMessage (myOptimisticContractAddress , abi. encodeCall (MyOptimisticContract.doSomething , (myFunctionParam)) , 1000000

// or use whatever gas limit you want) } } You can find the addresses of theL1CrossDomainMessenger and theL2CrossDomainMessenger contracts on OP Mainnet and OP Sepolia on theContract Addresses page.

Communication Speed

Unlike calls between contracts on the same blockchain, calls between Ethereum and OP Mainnet arenot instantaneous. The speed of a cross-chain transaction depends on the direction in which the transaction is sent.

For L1 to L2 Transactions

Transactions sent from L1 to L2 takeapproximately 1-3 minutes to get from Ethereum to OP Mainnet, or from Sepolia to OP Sepolia. This is because the Sequencer waits for a certain number of L1 blocks to be created before including L1 to L2 transactions in order to avoid potentially annoyingreorgs(opens in a new tab) .

For L2 to L1 Transactions

Transactions sent from L2 to L1 takeapproximately 7 days to get from OP Mainnet to Ethereum, or from OP Sepolia to Sepolia. This is because the bridge contract on L1 must wait for the L2 state to beproven to the L1 chain before it can relay the message.

The process of sending transactions from L2 to L1 involves four distinct steps:

1. The L2 transaction that sends a message to L1 is sent to the Sequencer.
2. This is just like any other L2 transaction and takes just a few seconds to be confirmed by the Sequencer.
3. The block containing the L2 transaction is proposed to the L1.
4. This typically takes approximately 20 minutes.
5. A proof of the transaction is submitted to theOptimismPortal (opens in a new tab)
6. contract on L1.
7. This can be done any time after step 2 is complete.
8. The transaction is finalized on L1.
9. This canonly
10. be done after thefault challenge period
11. has elapsed.
12. This period is 7 days on Ethereum and a few seconds on Sepolia.
13. This waiting period is a core part of the security model of the OP Stack and cannot be circumvented.

Accessingmsg.sender

Contracts frequently make use ofmsg.sender to make decisions based on the calling address. For example, many contracts will use theOwnable(opens in a new tab) pattern to selectively restrict access to certain functions. Because messages are essentially shuttled between L1 and L2 by the messenger contracts,themsg.sender you'll see when receiving one of these messages will be the messenger contract corresponding to the layer you're on.

In order to get around this, you can find axDomainMessageSender function to each messenger:

function

xDomainMessageSender () public

returns (address); If your contract has been called by one of the messenger contracts, you can use this function to see who'sactually sending this message. Here's how you might implement anonlyOwner modifier on L2:

modifier

onlyOwner () { require (msg.sender ==

address (messenger) && messenger. xDomainMessageSender () == owner); _; }

Fees For Sending Data Between L1 and L2

For L1 to L2 Transactions

The majority of the cost of an L1 to L2 transaction comes from the smart contract execution on L1. When sending an L1 to L2 transaction, you send to the L1CrossDomainMessenger (opens in a new tab) contract, which then sends a call to the OptimismPortal (opens in a new tab) contract. This involves some execution on L1, which costs gas. The total cost is ultimately determined by gas prices on Ethereum when you're sending the cross-chain transaction.

L1 to L2 execution also triggers contract execution on L2. TheOptimismPortal contract charges you for this L2 execution by burning a dynamic amount of L1 gas during your L1 to L2 transaction, depending on the gas limit you requested on L2. The amount of L1 gas charged increases when more people are sending L1 to L2 transactions (and decreases when fewer people are sending L1 to L2 transactions).

Since the gas amount charged is dynamic, the gas burn can change from block to block. You should always add a buffer of at least 20% to the gas limit for your L1 to L2 transaction to avoid running out of gas.

For L2 to L1 Transactions

Each message from L2 to L1 requires three transactions:

1. An L2 transaction thatinitiates
2. the transaction, which is priced the same as any other transaction made on OP Mainnet.
3. An L1 transaction thatproves
4. the transaction.
5. This transaction can only be submitted after L2 block including your L2 transaction is proposed on L1.
6. This transaction is expensive because it includes verifying a Merkle trie
7. inclusion proof on L1.
8. An L1 transaction thatfinalizes
9. the transaction.
10. This transaction can only be submitted after the transaction challenge period (7 days on mainnet) has passed.

The total cost of an L2 to L1 transaction is therefore the combined cost of the L2 initialization transaction and the two L1 transactions. The L1 proof and finalization transactions are typically significantly more expensive than the L2 initialization transaction.

Understanding the Challenge Period

One of the most important things to understand about L1 ⇔ L2 interaction is thatmainnet messages sent from Layer 2 to Layer 1 cannot be relayed for at least 7 days . This means that any messages you send from Layer 2 will only be received on Layer 1 after this one week period has elapsed. We call this period of time the "challenge period" because it is the time during which a transaction can be challenged with a fault proof .

Optimistic Rollups are "optimistic" because they're based around the idea of publishing theresult of a transaction to Ethereum without actually executing the transaction on Ethereum. In the "optimistic" case, this transaction result is correct and one can completely avoid the need to perform complicated (and expensive) logic on Ethereum.

However, one still needs some way to prevent incorrect transaction results from being published in place of correct ones. Here's where the "fault proof" comes into play. Whenever a transaction result is published, it's considered "pending" for a period of time known as the challenge period. During this period of time, anyone may re-execute the transactionon Ethereum in an attempt to demonstrate that the published result was incorrect.

If someone is able prove that a transaction result is faulty, then the result is scrubbed from existence and anyone can publish another result in its place (hopefully the correct one this time, financial punishments make faulty resultsvery costly for their publishers). Once the window for a given transaction result has fully passed without a challenge the result can be considered fully valid (or else someone would've challenged it).

Anyway, the point here is thatyou don't want to be making decisions about Layer 2 transaction results from inside a smart contract on Layer 1 until this challenge period has elapsed . Otherwise you might be making decisions based on an invalid transaction result. As a result, L2 ⇒ L1 messages sent using the standard messenger contracts cannot be relayed until they've waited out the full challenge period.

Custom Token Bridges Open Source Code Repo