Manual Execution

note

Read the CCIPmanual execution conceptual page to understand how manual execution works under the hood.

This tutorial is similar to theprogrammable token transfers example . It demonstrates the use of Chainlink CCIP for transferring tokens and arbitrary data between smart contracts on different blockchains. A distinctive feature of this tutorial is that we intentionally set a very low gas limit when using CCIP to send our message. This low gas limit is designed to cause the execution on the destination chain to fail, providing an opportunity to demonstrate the manual execution feature. Here's how you will proceed:

1. Initiate a Transfer: You'll transfer tokens and arbitrary data from your source contract on Avalanche Fuji to a receiver contract on Polygon Mumbai. You will notice that the CCIP message has a very low gas limit, causing the execution on the receiver contract to fail.
2. Failure of CCIP Message Delivery: Once the transaction is finalized on the source chain (Avalanche Fuji), CCIP will deliver your message to the receiver contract on the destination chain (Polygon Mumbai). You can follow the progress of your transaction using theCCIP explorer . Here, you'll observe that the execution on the receiver contract failed due to the low gas limit.
3. Manual Execution via CCIP Explorer: Using theCCIP explorer , you will override the previously set gas limit and retry the execution. This process is referred to asmanual execution.
4. Confirm Successful Execution: After manually executing the transaction with an adequate gas limit, you'll see that the status of your CCIP message is updated to successful. This indicates that the tokens and data were correctly transferred to the receiver contract.

Before you begin

1. You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read thistutorial , which will guide you through using theSolidity programming language , interacting with theMetaMask wallet and working within theRemix Development Environment .
2. Your account must have some ETH and LINK tokens onAvalanche Fujiand MATIC tokens onPolygon Mumbai. Learn how toAcquire testnet LINK .
3. Check theSupported Networks page to confirm that the tokens you will transfer are supported for your lane. In this example, you will transfer tokens fromAvalanche FujitoPolygon Mumbaiso check the list of supported tokenshere .
4. Learn how toacquire CCIP test tokens . Following this guide, you should have CCIP-BnM tokens, and CCIP-BnM should appear in the list of your tokens in MetaMask.
5. Learn how tofund your contract . This guide shows how to fund your contract in LINK, but you can use the same guide for funding your contract with any ERC20 tokens as long as they appear in the list of tokens in MetaMask.
6. Follow the previous tutorial:Transfer Tokens with Data to learn how to make programmable token transfers using CCIP.
7. Create a free account ontenderly . You will use tenderly to investigate the failed execution of the receiver contract.

Tutorial

In this tutorial, you'll send a textstringand CCIP-BnM tokens between smart contracts onAvalanche FujiandPolygon Mumbaiusing CCIP and pay transaction fees in LINK. The tutorial demonstrates setting a deliberately low gas limit in the CCIP message, causing initial execution failure on the receiver contract. You will then:

1. Use theCCIP explorer to increase the gas limit.
2. Manually retry the execution.
3. Observe successful execution after the gas limit adjustment.

// SPDX-License-Identifier: MITpragmasolidity0.8.19;import{IRouterClient}from"@chainlink/contracts-ccip/src/v0.8/ccip/interfaces/IRouterClient.sol";import{OwnerIsCreator}from"@chainlink/contracts-ccip/src/v0.8/shared/access/OwnerIsCreator.sol";import{Client}from"@chainlink/contracts-ccip/src/v0.8/ccip/libraries/Client.sol";import{CCIPReceiver}from"@chainlink/contracts-ccip/src/v0.8/ccip/applications/CCIPReceiver.sol";import{IERC20}from"@chainlink/contracts-ccip/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/token/ERC20/IERC20.sol";/ *** THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY. * THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE. * DO NOT USE THIS CODE IN PRODUCTION. *//// @title - A simple messenger contract for transferring/receiving tokens and data across chains.contractProgrammableTokenTransfersLowGasLimitisCCIPReceiver,OwnerIsCreator{// Custom errors to provide more descriptive revert messages.errorNotEnoughBalance(uint256currentBalance,uint256calculatedFees);// Used to make sure contract has enough balance to cover the fees.errorNothingToWithdraw();// Used when trying to withdraw Ether but there's nothing to withdraw.errorDestinationChainNotAllowed(uint64destinationChainSelector);// Used when the destination chain has not been allowlisted by the contract owner.errorSourceChainNotAllowed(uint64sourceChainSelector);// Used when the source chain has not been allowlisted by the contract owner.errorSenderNotAllowed(addresssender);// Used when the sender has not been allowlisted by the contract owner.// Event emitted when a message is sent to another chain.eventMessageSent(bytes32indexedmessageId,// The unique ID of the CCIP message.uint64indexeddestinationChainSelector,// The chain selector of the destination chain.addressreceiver,// The address of the receiver on the destination chain.stringtext,// The text being sent.addresstoken,// The token address that was transferred.uint256tokenAmount,// The token amount that was transferred.addressfeeToken,// the token address used to pay CCIP fees.uint256fees// The fees paid for sending the message.);// Event emitted when a message is received from another chain.eventMessageReceived(bytes32indexedmessageId,// The unique ID of the CCIP message.uint64indexedsourceChainSelector,// The chain selector of the source chain.addresssender,// The address of the sender from the source chain.stringtext,// The text that was received.addresstoken,// The token address that was transferred.uint256tokenAmount// The token amount that was transferred.);bytes32privates_lastReceivedMessageId;// Store the last received messageId.addressprivates_lastReceivedTokenAddress;// Store the last received token address.uint256privates_lastReceivedTokenAmount;// Store the last received amount.stringprivates_lastReceivedText;// Store the last received text.// Mapping to keep track of allowlisted destination chains.mapping(uint64=>bool)publicallowlistedDestinationChains;// Mapping to keep track of allowlisted source chains.mapping(uint64=>bool)publicallowlistedSourceChains;// Mapping to keep track of allowlisted senders.mapping(address=>bool)publicallowlistedSenders;IERC20private_linkToken;/// @notice Constructor initializes the contract with the router address.///** @param *router The address of the router contract.*/// @param _link The address of the link contract.constructor(address_router,address_link)CCIPReceiver(_router){s_linkToken=IERC20(_link);}/// @dev Modifier that checks if the chain with the given destinationChainSelector is allowlisted.*/// @param _destinationChainSelector The selector of the destination chain.modifieronlyAllowlistedDestinationChain(uint64_destinationChainSelector){if(!allowlistedDestinationChains[_destinationChainSelector])revertDestinationChainNotAllowed(_destinationChainSelector);_;}/// @dev Modifier that checks if the chain with the given sourceChainSelector is allowlisted and if the sender is allowlisted./// @param *sourceChainSelector The selector of the destination chain.*/// @param _sender The address of the sender.modifieronlyAllowlisted(uint64_sourceChainSelector,address_sender){if(!allowlistedSourceChains[_sourceChainSelector])revertSourceChainNotAllowed(_sourceChainSelector);if(!allowlistedSenders[_sender])revertSenderNotAllowed(_sender);_;}/// @dev Updates the allowlist status of a destination chain for transactions.*/// @notice This function can only be called by the owner.*/// @param _destinationChainSelector The selector of the destination chain to be updated.*/// @param allowed The allowlist status to be set for the destination chain.functionallowlistDestinationChain(uint64_destinationChainSelector,boolallowed)externalonlyOwner{allowlistedDestinationChains[_destinationChainSelector]=allowed;}/// @dev Updates the allowlist status of a source chain*/// @notice This function can only be called by the owner.*/// @param _sourceChainSelector The selector of the source chain to be updated.*/// @param allowed The allowlist status to be set for the source chain.functionallowlistSourceChain(uint64_sourceChainSelector,boolallowed)externalonlyOwner{allowlistedSourceChains[_sourceChainSelector]=allowed;}/// @dev Updates the allowlist status of a sender for transactions.*/// @notice This function can only be called by the owner.*/// @param _sender The address of the sender to be updated.*/// @param allowed The allowlist status to be set for the sender.functionallowlistSender(address_sender,boolallowed)externalonlyOwner{allowlistedSenders[_sender]=allowed;}/// @notice Sends data and transfer tokens to receiver on the destination chain.*/// @notice Pay for fees in LINK.*/// @notice the gasLimit is set to 20_000 on purpose to force the execution to fail on the destination chain*/// @dev Assumes your contract has sufficient LINK to pay for CCIP fees.*/// @param _destinationChainSelector The identifier (aka selector) for the destination blockchain.*/// @param _receiver The address of the recipient on the destination blockchain.*/// @param _text The string data to be sent.*/// @param _token token address.*/// @param _amount token amount.*/// @return messageId The ID of the CCIP message that was sent.functionsendMessagePayLINK(uint64_destinationChainSelector,address_receiver,stringcalldata_text,address_token,uint256_amount)externalonlyOwneronlyAllowlistedDestinationCh{// Set the token amountsClient.EVMTokenAmount[]memorytokenAmounts=newClient.EVMTokenAmount;tokenAmounts[0]=Client.EVMTokenAmount({token:_token,amount:_amount});// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message// address(linkToken) means fees are paid in LINKClient.EVM2AnyMessagememoryevm2AnyMessage=Client.EVM2AnyMessage({receiver:abi.encode(_receiver),// ABI-encoded receiver addressdata:abi.encode(_text),// ABI-encoded stringtokenAmounts:tokenAmounts,// The amount and type of token being transferredextraArgs:Client._argsToBytes(// gasLimit set to 20_000 on purpose to force the execution to fail on the destination chainClient.EVMExtraArgsV1({gasLimit:20_000})),// Set the feeToken to a LINK token addressfeeToken:address(s_linkToken)});// Initialize a router client instance to interact with cross-chain routerIRouterClient router=IRouterClient(this.getRouter());// Get the fee required to send the CCIP messageuint256fees=router.getFee(_destinationChainSelector,evm2AnyMessage);if(fees>s_linkToken.balanceOf(address(this)))revertNotEnoughBalance(s_linkToken.balanceOf(address(t approve the Router to transfer LINK tokens on contract's behalf. It will spend the fees in LINKs_linkToken.approve(address(router),fees);// approve the Router to spend tokens on contract's behalf. It will spend the amount of the given tokenIERC20(_token).approve(address(router),_amount);// Send the message through the router and store the returned message IDmessageId=router.ccipSend(_destinationChainSelector,evm2AnyMessage);// Emit an event with message detailsemitMessageSent(messageId,_destinationChainSelector,_receiver,_text,_token,_amount,address(s_linkToken),fees);// Return the message IDreturnmessageId;}/ * @notice Returns the details of the last CCIP received message. * @dev This function retrieves the ID, text, token address, and token amount of the last received CCIP message. * @return messageId The ID of the last received CCIP message. * @return text The text of the last received CCIP message. * @return tokenAddress The address of the token in the last CCIP received message. * @return tokenAmount The amount of the token in the last CCIP received message. */functiongetLastReceivedMessageDetails()publicviewreturns(bytes32messageId,stringmemorytext,addresstokenAddress,uint256tokenAmount){return(s_lastReceivedMessageId,s_lastReceivedText,s_lastReceivedTokenAddress,s_lastReceivedTokenAmount);}/// handle a received messagefunction_ccipReceive(Client.Any2EVMMessagememoryany2EvmMessage)internaloverrideonlyAllowlisted(any2EvmMessage.sourceChainSelector,abi.decode(any2EvmMessage.sender,(address)))/// Make sure source chain and sender are allowlisted{s_lastReceivedMessageId=any2EvmMessage.messageId;// fetch the messageIds_lastReceivedText=abi.decode(any2EvmMessage.data,(string));// abi-decoding of the sent text// Expect one token to be transferred at once, but you can transfer several tokens.s_lastReceivedTokenAddress=any2EvmMessage.destTokenAmounts[0].token;s_lastReceivedTokenAmount=any2EvmMessage.destTokenAmounts[0].amount;emitMessageReceived(any2EvmM fetch the source chain identifier (aka selector)abi.decode(any2EvmMessage.sender,(address)),// abi-decoding of the sender address,abi.decode(any2EvmMessage.data,(string)),any2EvmMessage.destTokenAmounts[0].token,any2EvmMessage.destTokenAmounts[0].amount);}/// @notice Allows the owner of the contract to withdraw all tokens of a specific ERC20 token.*/// @dev This function reverts with a 'NothingToWithdraw' error if there are no tokens to withdraw.*/// @param _beneficiary The address to which the tokens will be sent.*/// @param _token The contract address of the ERC20 token to be withdrawn.functionwithdrawToken(address_beneficiary,address_token)publiconlyOwner{// Retrieve the balance of this contractuint256amount=IERC20(_token).balanceOf(address(this));// Revert if there is nothing to withdrawif(amount==0)revertNothingToWithdraw();IERC20(_token).transfer(_beneficiary,amount);}}

Open in Remix What is Remix?

Deploy your contracts

To use this contract:

1. Open the contract in Remix .
2. Compile your contract.

3. Deploy, fund your sender contract onAvalanche Fujiand enable sending messages toPolygon Mumbai:

4. Open MetaMask and select the networkAvalanche Fuji.

5. In Remix IDE, click onDeploy & Run Transactionsand selectInjected Provider - MetaMaskfrom the environment list. Remix will then interact with your MetaMask wallet to communicate withAvalanche Fuji.
6. Fill in your blockchain's router and LINK contract addresses. The router address can be found on thesupported networks page and the LINK contract address on theLINK token contracts page . ForAvalanche Fuji, the router address is0xF694E193200268f9a4868e4Aa017A0118C9a8177and the LINK contract address is0x0b9d5D9136855f6FEc3c0993feE6E9CE8a297846.
7. Click thetransactbutton. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
8. Open MetaMask and fund your contract with CCIP-BnM tokens. You can transfer0.002CCIP-BnMto your contract.
9. Open MetaMask and fund your contract with LINK tokens. You can transfer0.1LINKto your contract. In this example, LINK is used to pay the CCIP fees.
10. Enable your contract to send CCIP messages toPolygon Mumbai:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onAvalanche Fuji.
11. Call theallowlistDestinationChainwith12532609583862916517as the destination chain selector, andtrueas allowed. Each chain selector is found on thesupported networks page .

12. Deploy your receiver contract onPolygon Mumbaiand enable receiving messages from your sender contract:

13. Open MetaMask and select the networkPolygon Mumbai.

14. In Remix IDE, underDeploy & Run Transactions, make sure the environment is stillInjected Provider - MetaMask.
15. Fill in your blockchain's router and LINK contract addresses. The router address can be found on thesupported networks page and the LINK contract address on theLINK token contracts page . ForPolygon Mumbai, the router address is0x1035CabC275068e0F4b745A29CEDf38E13aF41b1and the LINK contract address is0x326C977E6efc84E512bB9C30f76E30c160eD06FB.
16. Click thetransactbutton. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
17. Enable your contract to receive CCIP messages fromAvalanche Fuji:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
18. Call theallowlistSourceChainwith14767482510784806043as the source chain selector, andtrueas allowed. Each chain selector is found on thesupported networks page .
19. Enable your contract to receive CCIP messages from the contract that you deployed onAvalanche Fuji:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
20. Call theallowlistSenderwith the contract address of the contract that you deployed onAvalanche Fuji, andtrueas allowed.

At this point, you have onesendercontract onAvalanche Fujiand onereceivercontract onPolygon Mumbai. As security measures, you enabled the sender contract to send CCIP messages toPolygon Mumbaiand the receiver contract to receive CCIP messages from the sender andAvalanche Fuji.

Transfer and Receive tokens and data and pay in LINK

You will transfer0.001 CCIP-BnMand a text. The CCIP fees for using CCIP will be paid in LINK.

1. Send a string data with tokens fromAvalanche Fuji:

2. Open MetaMask and select the networkAvalanche Fuji.

3. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onAvalanche Fuji.
4. Fill in the arguments of thesendMessagePayLINKfunction:

ArgumentValue and Description_destinationChainSelector12532609583862916517CCIP Chain identifier of the destination blockchain (Polygon Mumbaiin this example). You can find each chain selector on thesupported networks page ._receiverYour receiver contract address atPolygon Mumbai.The destination contract address._textHello World!Anystring_token0xD21341536c5cF5EB1bcb58f6723cE26e8D8E90e4TheCCIP-BnMcontract address at the source chain (Avalanche Fujiin this example). You can find all the addresses for each supported blockchain on thesupported networks page ._amount1000000000000000The token amount (0.001 CCIP-BnM). 4. Click ontransactand confirm the transaction on MetaMask. 5. After the transaction is successful, record the transaction hash. Here is anexample of a transaction onAvalanche Fuji.

note

During gas price spikes, your transaction might fail, requiring more than0.1 LINKto proceed. If your transaction fails, fund your contract with moreLINKtokens and try again.

1. Open theCCIP explorer and search your cross-chain transaction using the transaction hash. Note that theGas Limitis20000. In this example, the CCIP message ID is0x21c3b177dd118a7347e744e0ac64cea69ce85d0a207e5a14b74867b1f911622a.
2. After a few minutes, the status will be updated toReady for manual executionindicating that CCIP could not successfully deliver the message due to the initial low gas limit. At this stage, you have the option to override the gas limit.

3. You can also confirm that the CCIP message was not delivered to the receiver contract on the destination chain:

4. Open MetaMask and select the networkPolygon Mumbai.

5. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
6. Call thegetLastReceivedMessageDetailsfunction.
7. Observe that the returned data is empty: the received messageId is0x0000000000000000000000000000000000000000000000000000000000000000, indicating no message was received. Additionally, the received text field is empty, the token address is the default0x0000000000000000000000000000000000000000, and the token amount shows as0.

Manual execution

Investigate the root cause of receiver contract execution failure

To determine if a low gas limit is causing the failure in the receiver contract's execution, consider the following methods:

- Error analysis: Examine the error description in the CCIP explorer. An error labeledReceiverError. This may be due to an out of gas error on the destination chain. Error code: 0x, often indicates a low gas issue
- Advanced Investigation Tool: For a comprehensive analysis, employ a sophisticated tool liketenderly . Tenderly can provide detailed insights into the transaction processes, helping to pinpoint the exact cause of the failure.

To usetenderly :

1. Copy the destination transaction hash from the CCIP explorer. In this example, the destination transaction hash is0x06cb1c7d92483e67382a932e99411c4525e2c3aca6e46498c2ba64bf7eb08aba.
2. Open tenderly and search for your transaction. You should see an interface similar to the following:
3. EnableFull Tracethen click onReverts.
4. Notice theout of gaserror in the receiver contract. In this example, the receiver contract is0x4314123b4E8739f5cb1eE176C33Bd45f8573c41C.

Trigger manual execution

You will increase the gas limit and trigger manual execution:

1. In theCCIP explorer , set theGas limit overrideto200000then click onTrigger Manual Execution.
2. After you confirm the transaction on Metamask, the CCIP explorer shows you a confirmation screen.
3. Click on theClosebutton and observe the status marked asSuccess.

4. Check the receiver contract on the destination chain:

5. Open MetaMask and select the networkPolygon Mumbai.

6. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
7. Call thegetLastReceivedMessageDetailsfunction.
8. Notice the received messageId is0x21c3b177dd118a7347e744e0ac64cea69ce85d0a207e5a14b74867b1f911622a, the received text isHello World!, the token address is0xf1E3A5842EeEF51F2967b3F05D45DD4f4205FF40(CCIP-BnM token address onPolygon Mumbai) and the token amount is 1000000000000000 (0.001 CCIP-BnM).

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to transfer tokens and data fromAvalanche FujitoPolygon Mumbaiand fromPolygon Mumbaiback toAvalanche Fuji.

Explanation

The smart contract used in this tutorial is configured to use CCIP for transferring and receiving tokens with data, similar to the contract in the Transfer Tokens with Data tutorial. For a detailed understanding of the contract code, refer to the code explanation section of that tutorial.

A key distinction in this tutorial is the intentional setup of a low gas limit of 20,000 for building the CCIP message. This specific gas limit setting is expected to fail the message delivery on the receiver contract in the destination chain:

```
Client._argsToBytes(Client.EVMExtraArgsV1({gasLimit:20_000}))
```