Transfer Tokens with Data - Defensive Example

This tutorial extends theprogrammable token transfers example . It uses Chainlink CCIP to transfer tokens and arbitrary data between smart contracts on different blockchains, and focuses on defensive coding in the receiver contract. In the event of a specified error during the CCIP message reception, the contract locks the tokens. Locking the tokens allows the owner to recover and redirect them as needed. Defensive coding is crucial as it enables the recovery of locked tokens and ensures the protection of your users' assets.

Transferring tokens

This tutorial uses the term "transferring tokens" even though the tokens are not technically transferred. Instead, they are locked or burned on the source chain and then unlocked or minted on the destination chain. Read theToken Pools section to understand the various mechanisms that are used to transfer value across chains.

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 onEthereum Sepoliaand 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 fromEthereum SepoliatoPolygon 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.

Tutorial

In this guide, you'll initiate a transaction from a smart contract onEthereum Sepolia, sending astringtext and CCIP-BnM tokens to another smart contract onPolygon Mumbaiusing CCIP. However, a deliberate failure in the processing logic will occur upon reaching the receiver contract. This tutorial will demonstrate a graceful error-handling approach, allowing the contract owner to recover the locked tokens.

Correctly estimate your gas limit

It is crucial to thoroughly test all scenarios to accurately estimate the required gas limit, including for failure scenarios. Be aware that the gas used to execute the error-handling logic for failure scenarios may be higher than that for successful scenarios.

// 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";import{SafeERC20}from"@chainlink/contracts-ccip/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/token/ERC20/utils/SafeERC20.sol";import{EnumerableMap}from"@chainlink/contracts-ccip/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/utils/structs/EnumerableMap.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.//// @dev - This example shows how to recover tokens in case of revertcontractProgrammableDefensiveTokenTransfersisCCIPReceiver,OwnerIsCreator{usingEnumerableMapforEnumerableMap.Bytes32ToUintMap;usingSafeERC20forIERC20;// 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.errorFailedToWithdrawEth(addressowner,addresstarget,uint256value);// Used when the withdrawal of Ether fails.errorDestinationChainNotAllowlisted(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.errorInvalidReceiverAddress();// Used when the receiver address is 0.errorOnlySelf();// Used when a function is called outside of the contract itself.errorErrorCase();// Used when simulating a revert during message processing.errorMessageNotFailed(bytes32messageId);// Example error code, could have many different error codes.enumErrorCode{// RESOLVED is first so that the default value is resolved.RESOLVED,// Could have any number of error codes here.FAILED}structFailedMessage{bytes32messageId;ErrorCode errorCode;}// 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.);eventMessageFailed(bytes32indexedmessageId,bytesreason);eventMessageRecovered(bytes32indexedmessageId);bytes32private_lastReceivedMessageId;// Store the last received messageId.addressprivate_lastReceivedTokenAddress;// Store the last received token address.uint256private_lastReceivedTokenAmount;// Store the last received amount.stringprivate_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;// The message contents of failed messages are stored here.mapping(bytes32messageId=>Client.Any2EVMMessage contents)public_messageContents;// Contains failed messages and their state.EnumerableMap.Bytes32ToUintMapinternal_failedMessages;// This is used to simulate a revert in the processMessage function.boolinternal_simRevert=false;/// @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])revertDestinationChainNotAllowlisted(_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 Modifier that checks the receiver address is not 0./// @param *receiver The receiver address.*modifiervalidateReceiver(address_receiver) *{if(_receiver==address(0))revertInvalidReceiverAddress();;}*/// @dev Modifier to allow only the contract itself to execute a function./// Throws an exception if called by any account other than the contract itself.modifieronlySelf(){if(msg.sender!=address(this))revertOnlySelf();_;}/// @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./// @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 {// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message// address(linkToken) means fees are paid in LINKClient.EVM2AnyMessagememoryevm2AnyMessage=_buildCCIPMessage(_receiver,_text,_token,_amount,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(tl 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 Sends data and transfer tokens to receiver on the destination chain./// @notice Pay for fees in native gas./// @dev Assumes your contract has sufficient native gas like ETH on Ethereum or MATIC on Polygon./// @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.functionsendMessagePayNative(uint64_destinationChainSelector,address_receiver,stringcalldata_text,address_token,uint256_amount)externalonlyOwneronlyAllowlistedDestinationC {// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message// address(0) means fees are paid in native gasClient.EVM2AnyMessagememoryevm2AnyMessage=_buildCCIPMessage(_receiver,_text,_token,_amount,address(0));// 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>address(this).balance)revertNotEnoughBalance(address(this).balance,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{value:fees}(_destinationChainSelector,evm2AnyMessage);// Emit an event with message detailsemitMessageSent(messageId,_destinationChainSelector,_receiver,_text,_token,_amount,address(0),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);}/ * @notice Retrieves a paginated list of failed messages. * @dev This function returns a subset of failed messages defined by offset and limit parameters. It ensures that the pagination parameters are within the bounds of the available data set. * @param offset The index of the first failed message to return, enabling pagination by skipping a specified number of messages from the start of the dataset. * @param limit The maximum number of failed messages to return, restricting the size of the returned array. * @return failedMessages An array of FailedMessage struct, each containing a messageId and an errorCode (RESOLVED or FAILED), representing the requested subset of failed messages. The length of the returned array is determined by the limit and the total number of failed messages.*
/functiongetFailedMessages(uint256offset,uint256limit)externalviewreturns(FailedMessage[]memory){uint256length=s_failedMessages.length();// Calculate the actual number of items to return (can't exceed total length or requested limit)uint256returnLength=(offset+limit>length)?length-offset:limit;FailedMessage[]memoryfailedMessages=newFailedMessage;// Adjust loop to respect pagination (start at offset, end at offset + limit or total length)for(uint256i=0;i<returnLength;i++)
{(bytes32messageId,uint256errorCode)=s_failedMessages.at(offset+i);failedMessages[i]=FailedMessage(messageId,ErrorCode(errorCode));}returnfailedMessages;}/// @notice The entrypoint for the CCIP router to call. This function should/// never revert, all errors should be handled internally in this contract./// @param any2EvmMessage The message to process./// @dev Extremely important to ensure only router calls this.functionccipReceive(Client.Any2EVMMessagecalldataany2EvmMessage)externaloverride
onlyRouteronlyAllowlisted(any2EvmMessage.sourceChainSelector,abi.decode(any2EvmMessage.sender,(address)))// Make sure the source chain and sender are allowlisted{/ *solhint-disable no-empty-blocks* /trythis.processMessage(any2EvmMessage){// Intentionally empty in this example; no action needed if processMessage succeeds}catch(bytesmemoryerr){// Could set different error codes based on the caught error. Each could be// handled
differently.s_failedMessages.set(any2EvmMessage.messageId,uint256(ErrorCode.FAILED));s_messageContents[any2EvmMessage.messageId]=any2EvmMessage;// Don't revert so CCIP doesn't revert. Emit event instead.// The message can be retried later without having to do manual execution of CCIP.emitMessageFailed(any2EvmMessage.messageId,err);return;}}/// @notice Serves as the entry point for this contract to process incoming messages./// @param any2EvmMessage Received CCIP message./// @dev Transfers specified token amounts to the owner of this contract. This function/// must be external because of the try/catch for error handling./// It uses the onlySelf: can only be called from the
contract.functionprocessMessage(Client.Any2EVMMessagecalldataany2EvmMessage)externalonlySelfonlyAllowlisted(any2EvmMessage.sourceChainSelector,abi.decode(any2EvmMessage.sender,(address)))// Make sure the source chain and sender are allowlisted{// Simulate a revert for testing purposesif(s_simRevert)revertErrorCase();_ccipReceive(any2EvmMessage);// process the message - may revert as well}/// @notice Allows the owner to retry a failed message in order to unblock the associated tokens./// @param messageId The unique identifier of the failed message./// @param tokenReceiver The address to which the tokens will be sent./// @dev This function is only callable by the contract owner. It changes the status of the message/// from 'failed' to 'resolved' to prevent reentry and multiple retries of the same message.functionretryFailedMessage(bytes32messageId,addresstokenReceiver)externalonlyOwner{// Check if the message has failed; if not, revert the transaction.if(s_failedMessages.get(messageId)!=uint256(ErrorCode.FAILED))revertMessageNotFailed(messageId);// Set the error code to RESOLVED to disallow reentry and multiple retries of the same failed message.s_failedMessages.set(messageId,uint256(ErrorCode.RESOLVED));// Retrieve the content of the failed
message.Client.Any2EVMMessagememorymessage=s_messageContents[messageId];// This example expects one token to have been sent, but you can handle multiple tokens.// Transfer the associated tokens to the specified receiver as an escape hatch.IERC20(message.destTokenAmounts[0].token).safeTransfer(tokenReceiver,message.destTokenAmounts[0].amount);// Emit an event indicating that the message has been recovered.emitMessageRecovered(messageId);}/// @notice Allows the owner to toggle simulation of reversion for testing purposes./// @param simRevert If true, simulates a revert condition; if false, disables the simulation./// @dev This function is only callable by the contract
owner.functionsetSimRevert(boolsimRevert)externalonlyOwner{s_simRevert=simRevert;}function_ccipReceive(Client.Any2EVMMessagememoryany2EvmMessage)internaloverride{s_lastReceivedMess
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 Construct a CCIP message./// @dev This function will create an EVM2AnyMessage struct with all the necessary information for programmable tokens transfer./// @param _receiver The address of the receiver./// @param _text The string data to be sent./// @param _token The token to be transferred./// @param _amount The amount of the token to be transferred./// @param _feeTokenAddress The address of the token used for fees. Set address(0) for native gas./// @return Client.EVM2AnyMessage Returns an EVM2AnyMessage struct which contains information for sending a CCIP
message.function_buildCCIPMessage(address_receiver,stringcalldata_text,address_token,uint256_amount,address_feeTokenAddress)privatepurereturns(Client.EVM2AnyMessagememory){// Set the token
amountsClient.EVMTokenAmount[]memorytokenAmounts=newClient.EVMTokenAmount;Client.EVMTokenAmountmemorytokenAmount=Client.EVMTokenAmount({token:_token,amount:_amount});toke
Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain
messageClient.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(// Additional arguments, setting gas limitClient.EVMExtraArgsV1({gasLimit:400_000})),// Set the feeToken to a feeTokenAddress, indicating specific asset will be used for feesfeeToken:_feeTokenAddress});returnevm2AnyMessage;}/// @notice Fallback function to allow the contract to receive Ether./// @dev This function has no function body, making it a default function for receiving Ether./// It is automatically called when Ether is sent to the contract without any data.receive()externalpayable{}/// @notice Allows the contract owner to withdraw the entire balance of Ether from the contract./// @dev This function reverts if there are no funds to withdraw or if the transfer fails./// It should only be callable by the owner of the contract./// @param _beneficiary The address to which the Ether should be sent.functionwithdraw(address_beneficiary)publiconlyOwner{// Retrieve the balance of this contractuint256amount=address(this).balance;// Revert if there is nothing to withdrawif(amount==0)revertNothingToWithdraw();// Attempt to send the funds, capturing the success status and discarding any return data(boolsent,)=_beneficiary.call{value:amount}("");// Revert if the send failed, with information about the attempted transferif(!sent)revertFailedToWithdrawEth(msg.sender,_beneficiary,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).safeTransfer(_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 onEthereum Sepoliaand enable sending messages toPolygon Mumbai:
4. Open MetaMask and select the networkEthereum Sepolia.
5. In Remix IDE, click onDeploy & Run Transactionsand selectInjected Provider - MetaMaskfrom the environment list. Remix will then interact with your MetaMask wallet to communicate withEthereum Sepolia.
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 . ForEthereum Sepolia, the router address is0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59and the LINK contract address is0x779877A7B0D9E8603169DdbD7836e478b4624789.
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. 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 onEthereum Sepolia.
10. Call theallowlistDestinationChainwith12532609583862916517as the destination chain selector, andtrueas allowed. Each chain selector is found on thesupported networks page .
11. Deploy your receiver contract onPolygon Mumbaiand enable receiving messages from your sender contract:
12. Open MetaMask and select the networkPolygon Mumbai.
13. In Remix IDE, underDeploy & Run Transactions, make sure the environment is stillInjected Provider - MetaMask.
14. 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.
15. Click thetransactbutton. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
16. Enable your contract to receive CCIP messages fromEthereum Sepolia:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
17. Call theallowlistSourceChainwith16015286601757825753as the source chain selector, andtrueas allowed. Each chain selector is found on thesupported networks page .
18. Enable your contract to receive CCIP messages from the contract that you deployed onEthereum Sepolia:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
19. Call theallowlistSenderwith the contract address of the contract that you deployed onEthereum Sepolia, andtrueas allowed.
20. Call thesetSimRevertfunction, passingtrueas a parameter, then wait for the transaction to confirm. Settings_simRevertto true simulates a failure when processing the received message. Read theexplanation section for more details.

At this point, you have onesendercontract onEthereum Sepoliaand 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 andEthereum Sepolia. The receiver contract cannot process the message, and therefore, instead of throwing an exception, it will lock the received tokens, enabling the owner to recover them.

Note: Another security measure enforces that only the router can call the_ccipReceivefunction. Read theexplanation section for more details.

Recover the locked tokens

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

1. Open MetaMask and connect toEthereum Sepolia. Fund your contract with LINK tokens. You can transfer0.5LINKto your contract. In this example, LINK is used to pay the CCIP fees.
2. Send a string data with tokens fromEthereum Sepolia:
3. Open MetaMask and select the networkEthereum Sepolia.

4. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onEthereum Sepolia.
5. 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_token0xFd57b4ddBf88a4e07fF4e34C487b99af2Fe82a05TheCCIP-BnMcontract address at the source chain (Ethereum Sepoliain 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 onEthereum Sepolia.

note

During gas price spikes, your transaction might fail, requiring more than0.5 LINKto proceed. If your transaction fails, fund your contract with moreLINKtokens and try again. 3. Open theCCIP explorer and search your cross-chain transaction using the transaction hash. 4. The CCIP transaction is completed once the status is marked as "Success". In this example, the CCIP message ID is0x2fb721d506350a75439b16f7dab551cf518c6dcc473b4e9271218f8f470533f8. 5. Check the receiver contract on the destination chain:

1. Open MetaMask and select the networkPolygon Mumbai.
2. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
3. Call thegetFailedMessagesfunction with anoffsetof0and alimitof1to retrieve the first failed message.
4. Notice the returned values are:0x2fb721d506350a75439b16f7dab551cf518c6dcc473b4e9271218f8f470533f8(the message ID) and1(the error code indicating failure).
5. To recover the locked tokens, call theretryFailedMessagefunction:

ArgumentDescriptionmessageIdThe unique identifier of the failed message.tokenReceiverThe address to which the tokens will be sent. 7. After confirming the transaction, you can open it in a block explorer. Notice that the locked funds were transferred to thetokenReceiveraddress. 8. Call again thegetFailedMessagesfunction with anoffsetof0and alimitof1to retrieve the first failed message. Notice that the error code is now0, indicating that the message was resolved.

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to transfer tokens with data fromEthereum SepoliatoPolygon Mumbaiand fromPolygon Mumbaiback toEthereum Sepolia.

Explanation

The smart contract featured in this tutorial is designed to interact with CCIP to transfer and receive tokens and data. The contract code is similar to theTransfer Tokens with Data tutorial. Hence, you can refer to itscode explanation . We will only explain the main differences.

Sending messages

ThesendMessagePayLINKfunction is similar to thesendMessagePayLINKfunction in theTransfer Tokens with Data tutorial. The main difference is the increased gas limit to account for the additional gas required to process the error-handling logic.

Receiving and processing messages

Upon receiving a message on the destination blockchain, theccipReceivefunction is called by the CCIP router. This function serves as the entry point to the contract for processing incoming CCIP messages, enforcing crucial security checks through theonlyRouter, andonlyAllowlistedmodifiers.

Here's the step-by-step breakdown of the process:

1. Entrance throughccipReceive:
2. TheccipReceivefunction is invoked with anAny2EVMMessagestruct containing the message to be processed.
3. Security checks ensure the call is from the authorized router, an allowlisted source chain, and an allowlisted sender.
4. Processing Message:
5. ccipReceivecalls theprocessMessagefunction, which is external to leverage Solidity's try/catch error handling mechanism.Note: TheonlySelfmodifier ensures that only the contract can call this function.
6. InsideprocessMessage, a check is performed for a simulated revert condition using thes_simRevertstate variable. This simulation is toggled by thesetSimRevertfunction, callable only by the contract owner.
7. Ifs_simRevertis false,processMessagecalls the_ccipReceivefunction for further message processing.
8. Message Processing in_ccipReceive:
9. _ccipReceiveextracts and stores various information from the message, such as themessageId, decodedsenderaddress, token amounts, and data.
10. It then emits aMessageReceivedevent, signaling the successful processing of the message.
11. Error Handling:
12. If an error occurs during the processing (or a simulated revert is triggered), the catch block withinccipReceiveis executed.
13. ThemessageIdof the failed message is added tos_failedMessages, and the message content is stored ins_messageContents.
14. AMessageFailedevent is emitted, which allows for later identification and reprocessing of failed messages.

Reprocessing of failed messages

TheretryFailedMessagefunction provides a mechanism to recover assets if a CCIP message processing fails. It's specifically designed to handle scenarios where message data issues prevent entire processing yet allow for token recovery:

1. Initiation:
2. Only the contract owner can call this function, providing themessageIdof the failed message and thetokenReceiveraddress for token recovery.
3. Validation:
4. It checks if the message has failed usings_failedMessages.get(messageId). If not, it reverts the transaction.
5. Status Update:
6. The error code for the message is updated toRESOLVEDto prevent reentry and multiple retries.
7. Token Recovery:
8. Retrieves the failed message content usings_messageContents[messageId].
9. Transfers the locked tokens associated with the failed message to the specifiedtokenReceiveras an escape hatch without processing the entire message again.
10. Event Emission:
11. An eventMessageRecoveredis emitted to signal the successful recovery of the tokens.

This function showcases a graceful asset recovery solution, protecting user values even when message processing encounters issues.