Send Arbitrary Data

In this tutorial, you will use Chainlink CCIP to send data between smart contracts on different blockchains. First, you will pay for the CCIP fees on the source blockchain using LINK. Then, you will use the same contract to pay CCIP fees in native gas tokens. For example, you would use ETH on Ethereum or MATIC on Polygon.

Node Operator Rewards

CCIP rewards the oracle node and Risk Management node operators in LINK.

Before you begin

- 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 .
- Your account must have some ETH tokens onEthereum Sepoliaand MATIC tokens onPolygon Mumbai.
- Learn how toAcquire testnet LINK andFund your contract with LINK .

Tutorial

In this tutorial, you will send astringtext between smart contracts onEthereum SepoliaandPolygon Mumbaiusing CCIP. First, you will payCCIP fees in LINK , then you will payCCIP fees in native gas .

// 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 sending/receving string data across chains.contractMessengerisCCIPReceiver,OwnerIsCreator{// Custom errors to provide more descriptive revert messages.errorNotEnoughBalance(uint256currentBalance,uint256calculatedFees);// Used to make sure contract has enough balance.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.errorSourceChainNotAllowlisted(uint64sourceChainSelector);// Used when the source chain has not been allowlisted by the contract owner.errorSenderNotAllowlisted(addresssender);// Used when the sender has not been allowlisted by the contract owner.errorInvalidReceiverAddress();// Used when the receiver address is 0.// 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.addressfeeToken,// the token address used to pay CCIP fees.uint256fees// The fees paid for sending the CCIP 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.);bytes32private*lastReceivedMessageId;// Store the last received messageId.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;/// @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])revertSourceChainNotAllowlisted(_sourceChainSelector);if(!allowlistedSenders[_sender])revertSenderNotAllowlisted(_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 Updates the allowlist status of a destination chain for transactions.functionallowlistDestinationChain(uint64_destinationChainSelector,boolallowed)externalonlyOwner{allowlistedDestinationChains[_destinationChainSelector]=allowed;}/// @dev Updates the allowlist status of a source chain for transactions.functionallowlistSourceChain(uint64_sourceChainSelector,boolallowed)externalonlyOwner{allowlistedSourceChains[_sourceChainSelector]=allowed;}/// @dev Updates the allowlist status of a sender for transactions.functionallowlistSender(address_sender,boolallowed)externalonlyOwner{allowlistedSenders[_sender]=allowed;}/// @notice Sends data to receiver on the destination chain./// @notice Pay for fees in LINK./// @dev Assumes your contract has sufficient LINK./// @param _destinationChainSelector The identifier (aka selector) for the destination blockchain./// @param _receiver The address of the recipient on the destination blockchain./// @param _text The text to be sent./// @return messageId The ID of the CCIP message that was sent.functionsendMessagePayLINK(uint64_destinationChainSelector,address_receiver,stringcalldata_text)externalonlyOwneronlyAllowlistedDestinationChain(_destinationChainSelector)validateReceiver{// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain messageClient.EVM2AnyMessagememoryevm2AnyMessage=_buildCCIPMessage(_receiver,_text,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(this)),fees);// approve the Router to transfer LINK tokens on contract's behalf. It will spend the fees in LINKs_linkToken.approve(address(router),fees);// Send the CCIP message through the router and store the returned CCIP message IDmessageId=router.ccipSend(_destinationChainSelector,evm2AnyMessage);// Emit an event with message detailsemitMessageSent(messageId,_destinationChainSelector,_receiver,_text,address(s_linkToken),fees);// Return the CCIP message IDreturnmessageId;}/// @notice Sends data to receiver on the destination chain./// @notice Pay for fees in native gas./// @dev Assumes your contract has sufficient native gas tokens./// @param _destinationChainSelector The identifier (aka selector) for the destination blockchain./// @param _receiver The address of the recipient on the destination blockchain./// @param _text The text to be sent./// @return messageId The ID of the CCIP message that was sent.functionsendMessagePayNative(uint64_destinationChainSelector,address_receiver,stringcalldata_text)externalonlyOwneronlyAllowlistedDestinationChain(_destinationChainSelector)validateReceiver{// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain messageClient.EVM2AnyMessagememoryevm2AnyMessage=_buildCCIPMessage(_receiver,_text,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);// Send the CCIP message through the router and store the returned CCIP message IDmessageId=router.ccipSend{value:fees}(_destinationChainSelector,evm2AnyMessage);// Emit an event with message detailsemitMessageSent(messageId,_destinationChainSelector,_receiver,_text,address(0),fees);// Return the CCIP message IDreturnmessageId;}/// 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 textemitMessageReceived(any2EvmMessage.messageId,any2EvmMessage.sourceChainSelector,// fetch the source chain identifier (aka selector)abi.decode(any2EvmMessage.sender,(address)),// abi-decoding of the sender address,abi.decode(any2EvmMessage.data,(string)));}/// @notice Construct a CCIP message./// @dev This function will create an EVM2AnyMessage struct with all the necessary information for sending a text./// @param _receiver The address of the receiver./// @param _text The string data to be sent./// @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_feeTokenAddress)privatepurereturns(Client.EVM2AnyMessagememory){// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain messagereturnClient.EVM2AnyMessage({receiver:abi.encode(_receiver),// ABI-encoded receiver addressdata:abi.encode(_text),// ABI-encoded stringtokenAmounts:newClient.EVMTokenAmount,// Empty array aas no tokens are transferredextraArgs:Client._argsToBytes(// Additional arguments, setting gas limitClient.EVMExtraArgsV1({gasLimit:200_000})),// Set the feeToken to a feeTokenAddress, indicating specific asset will be used for feesfeeToken:_feeTokenAddress});}/// @notice Fetches the details of the last received message./// @return messageId The ID of the last received message./// @return text The last received text.functiongetLastReceivedMessageDetails()externalviewreturns(bytes32messageId,stringmemorytext){return(s_lastReceivedMessageId,s_lastReceivedText);}/// @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).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 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 the router address and the link address for your network. You can find the router address on thesupported networks page and the LINK token address on theLINK Token contracts page .

ForEthereum Sepolia, the router address is0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59and the LINK contract address is0x779877A7B0D9E8603169DdbD7836e478b4624789.
7. Click ontransact. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
8. 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.
9. Call theallowlistDestinationChainwith12532609583862916517as the destination chain selector, andtrueas allowed. Each chain selector is found on thesupported networks page .
10. Deploy your receiver contract onPolygon Mumbaiand enable receiving messages from your sender contract:
11. Open MetaMask and select the networkPolygon Mumbai.
12. In Remix IDE, underDeploy & Run Transactions, make sure the environment is stillInjected Provider - MetaMask.
13. Fill in the router address and the LINK address for your network. You can find the router address 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.
14. Click ontransact. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
15. 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.
16. Call theallowlistSourceChainwith16015286601757825753as the source chain selector, andtrueas allowed. Each chain selector is found on thesupported networks page .
17. 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.
18. Call theallowlistSenderwith the contract address of the contract that you deployed onEthereum Sepolia, andtrueas allowed.

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

Send data and pay in LINK

You will use CCIP to send a text. The CCIP fees for using CCIP will be paid in LINK. Read thisexplanation for a detailed description of the code example.

1. Open MetaMask and connect toEthereum Sepolia. Fund your contract with LINK tokens. You can transfer0.1LINKto your contract. In this example, LINK is used to pay the CCIP fees.
2. Send "Hello World!" 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:

ArgumentDescriptionValue (Polygon Mumbai)_destinationChainSelectorCCIP Chain identifier of the target blockchain. You can find each network's chain selector on thesupported networks page12532609583862916517_receiverThe destination smart contract addressYour deployed receiver contract address_textanystringHello World! 4. Click ontransactand confirm the transaction on MetaMask. 5. Once the transaction is successful, note the transaction hash. Here is anexample of a transaction onEthereum Sepolia.

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. 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".Note: In this example, the CCIP message ID is0x223b73f2e7dfb65cf317661ed9c5ba6b9f0bd8d61170a95c801b707d3526070a. 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 thegetLastReceivedMessageDetails.
4. Notice the received text is the one you sent, "Hello World!" and the message ID is the one you expect0x223b73f2e7dfb65cf317661ed9c5ba6b9f0bd8d61170a95c801b707d3526070a.

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

Send data and pay in native

You will use CCIP to send a text. The CCIP fees for using CCIP will be paid in native gas. Read thisexplanation for a detailed description of the code example.

1. Open MetaMask and connect toEthereum Sepolia. Fund your contract with ETH. You can transfer0.01ETHto your contract. In this example, ETH is used to pay the CCIP fees.
2. Send "Hello World!" 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 thesendMessagePayNativefunction:

ArgumentDescriptionValue (Polygon Mumbai)_destinationChainSelectorCCIP Chain identifier of the target blockchain. You can find each network's chain selector on thesupported networks page12532609583862916517_receiverThe destination smart contract addressYour deployed receiver contract address_textanystringHello World! 4. Click ontransactand confirm the transaction on MetaMask. 5. Once the transaction is successful, note the transaction hash. Here is anexample of a transaction onEthereum Sepolia.

note

During gas price spikes, your transaction might fail, requiring more than0.01 ETHto proceed. If your transaction fails, fund your contract with moreETHand 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 is0x54862fd17ca9718b55e3e2d34c84f26ef1e71a20cc2398f76974e40aff378838. Note that CCIP fees are denominated in LINK. Even if CCIP fees are paid using native gas tokens, node operators will be paid in LINK. 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 thegetLastReceivedMessageDetails.
4. Notice the received text is the one you sent, "Hello World!" and the message ID is the one you expect0x54862fd17ca9718b55e3e2d34c84f26ef1e71a20cc2398f76974e40aff378838.

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

Explanation

The smart contract featured in this tutorial is designed to interact with CCIP to send and receive messages. The contract code contains supporting comments clarifying the functions, events, and underlying logic. Here we will further explain initializing the contract and sending and receiving data.

Initializing of the contract

When deploying the contract, we define the router address and LINK contract address of the blockchain we deploy the contract on. Defining the router address is useful for the following:

- Sender part:
- Calls the router'sgetFeefunction to estimate the CCIP fees.
- Calls the router'sccipSendfunction to send CCIP messages.
- Receiver part:
- The contract inherits fromCCIPReceiver , which serves as a base contract for receiver contracts. This contract requires that child contracts implement the_ccipReceivefunction ._ccipReceiveis called by theccipReceivefunction , which ensures that only the router can deliver CCIP messages to the receiver contract.

Sending data and pay in LINK

ThesendMessagePayLINKfunction undertakes five primary operations:

1. Call the_buildCCIPMessageprivate function to construct a CCIP-compatible message using theEVM2AnyMessagestruct :

2. The_receiveraddress is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved throughabi.encode .

3. Thedatais encoded from astringtobytesusingabi.encode .
4. ThetokenAmountsis an emptyEVMTokenAmountstruct array as no tokens are transferred.
5. TheextraArgsspecifies thegasLimitfor relaying the message to the recipient contract on the destination blockchain. In this example, thegasLimitis set to200000.
6. The_feeTokenAddressdesignates the token address used for CCIP fees. Here,address(linkToken)signifies payment in LINK.

Do not hardcode extraArgs

To simplify this example,extraArgsare hardcoded in the contract. For production deployments, make sure thatextraArgsis mutable. This allows you to build it offchain and pass it in a call to a function or store it in a variable that you can update on-demand. This makesextraArgscompatible with future CCIP upgrades. 2. Computes the message fees by invoking the router'sgetFeefunction . 3. Ensures your contract balance in LINK is enough to cover the fees. 4. Grants the router contract permission to deduct the fees from the contract's LINK balance. 5. Dispatches the CCIP message to the destination chain by executing the router'sccipSendfunction .

Note: As a security measure, thesendMessagePayLINKfunction is protected by theonlyAllowlistedDestinationChain, ensuring the contract owner has allowlisted a destination chain.

Sending data and pay in native

ThesendMessagePayNativefunction undertakes four primary operations:

1. Call the_buildCCIPMessageprivate function to construct a CCIP-compatible message using theEVM2AnyMessagestruct :

2. The_receiveraddress is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved throughabi.encode .

3. Thedatais encoded from astringtobytesusingabi.encode .
4. ThetokenAmountsis an emptyEVMTokenAmountstruct array as no tokens are transferred.
5. TheextraArgsspecifies thegasLimitfor relaying the message to the recipient contract on the destination blockchain. In this example, thegasLimitis set to200000.
6. The_feeTokenAddressdesignates the token address used for CCIP fees. Here,address(0)signifies payment in native gas tokens (ETH).

Do not hardcode extraArgs

To simplify this example,extraArgsare hardcoded in the contract. For production deployments, make sure thatextraArgsis mutable. This allows you to build it offchain and pass it in a call to a function or store it in a variable that you can update on-demand. This makesextraArgscompatible with future CCIP upgrades. 2. Computes the message fees by invoking the router'sgetFeefunction . 3. Ensures your contract balance in native gas is enough to cover the fees. 4. Dispatches the CCIP message to the destination chain by executing the router'sccipSendfunction .Note:msg.valueis set because you pay in native gas.

Note: As a security measure, thesendMessagePayNativefunction is protected by theonlyAllowlistedDestinationChain, ensuring the contract owner has allowlisted a destination chain.

Receiving data

On the destination blockchain, the router invokes theccipReceivefunction which expects anAny2EVMMessagestruct that contains:

- The CCIPmessageId.
- ThesourceChainSelector.
- Thesenderaddress in bytes format. Given that the sender is known to be a contract deployed on an EVM-compatible blockchain, the address is decoded from bytes to an Ethereum address using theABI specifications .
- Thedata, which is also in bytes format. Given astringis expected, the data is decoded from bytes to a string using theABI specifications .

This example applies three important security measures:

- _ccipReceiveis called by theccipReceivefunction , which ensures that only the router can deliver CCIP messages to the receiver contract. See theonlyRoutermodifier for more information.
- The modifieronlyAllowlistedensures that only a call from an allowlisted source chain and sender is accepted.