Multicall

Multicall allows you to aggregate multiple contract reads into a single JSON-RPC request, and execute multiple state-changing calls in a single transaction on the FVM.

Multicall3

Multicall3 is a powerful tool that offers batch contract calls to smart contracts on the Filecoin Virtual Machine (FVM).

Multicall3 is deployed on over 100 chains at0xcA11bde05977b3631167028862bE2a173976CA11 . A sortable, searchable list of all chains it's deployed on can be foundhere .

Themulticall3 ABI can be downloaded or copied to the clipboard in various formats, including:

- Solidity interface.
- JSON ABI, prettified.
- JSON ABI, minified.
- ethers.js
- human readable ABI.
- viem
- human readable ABI.

Alternatively, you can:

- Download the ABI from thereleases
- page.
- Copy the ABI fromEtherscan
- .
- InstallFoundry
- and runcast interface 0xcA11bde05977b3631167028862bE2a173976CA11
- .

Contract address

Multicall has the same, precomputed address for all of the networks it is deployed on.

Name Address Mainnet Calibration Multicall - Mainnet 0xcA11bde05977b3631167028862bE2a173976CA11 ✔ ✖Multicall - Calibration 0xcA11bde05977b3631167028862bE2a173976CA11 ✖ ✔

Usage

To use Multicall3 to send batch contract read/write to your smart contract, you will need to:

1. Obtain the Multicall3 contract address for the network you're using (Filecoin mainnet or Calibration testnet).
2. Get the Multicall3 ABI, which can be downloaded or copied from various sources mentioned above.
3. Create an instance of the Multicall3 contract using a web3 library like ethers.js or viem.
4. Prepare your batch calls, including the target contract addresses, function selectors, and input data.
5. Use the appropriate Multicall3 method (e.g.,aggregate3
6. for multiple calls) to execute your batch operations.
7. Process the returned data from the Multicall3 contract.
8.

The steps above differ slightly for integrations using smart contracts, where steps 2 and 3 are replaced with:

1. Import the Multicall3 interface in your smart contract.
2. Create a function that interacts with the Multicall3 contract using the imported interface.
3.

Many libraries and tools such asethers-rs ,viem , andape have native Multicall3 integration which can be used in your projects directly. To learn how to use Multicall3 with these tools, check outMulticall3 examples folder

Batching Contract Reads

Batching contract reads, one of the most common use cases, allows a singleeth_call JSON RPC request to return the results of multiple contract function calls. It has many benefits:

1. Reduced JSON RPC Requests
2. : Multicall reduces the number of separate JSON RPC requests that need to be sent. This is particularly useful when using remote nodes, such as GLIF. By aggregating multiple contract reads into a single JSON-RPC request, Multicall (1) reduces RPC usage and therefore costs, and (2) reduces the number of round trips between the client and the

node, which can significantly improve performance
3. Consistent Data from the Same Block
4. : Multicall guarantees that all values returned are from the same block. This ensures data consistency and reliability, as all the read operations are performed on the same state of the blockchain.
5. Detection of Stale Data
6. : Multicall enables the block number or timestamp to be returned with the read data. This feature helps in detecting stale data, as developers can compare the block number or timestamp with the current state of the blockchain to ensure the data is up-to-date.
7.

When directly interacting with the Multicall3 contract to batch calls, you'll typically use theaggregate3 method. This method allows you to execute multiple contract calls in a single transaction. Here's an explanation of how it works, along with examples:

1. Solidity Implementation: Theaggregate3
2. method is implemented in the Multicall3 contract like this:
3. ```
4. Copy
5. functionaggregate3(Call3[]calldatacalls)publicpayablereturns(Result[]memoryreturnData) {
6. uint256length=calls.length;
7. returnData=newResult;
8. for(uint256i=0; i<length;) {
9. (boolsuccess,bytesmemoryret)=calls[i].target.call(calls[i].callData);
10. if(calls[i].allowFailure) {
11. returnData[i]=Result(success,ret);
12. }else{
13. require(success,"Multicall3: call failed");
14. returnData[i]=Result(true,ret);
15. }
16. unchecked{++i; }
17. }
18. }
19. ```
20. Example of sending multicalls to this smart contract: Here's an example using ethers.js to interact with the Multicall3 contract:
21. ```
22. Copy
23. const{ethers}=require("ethers");
24. constprovider=newethers.providers.JsonRpcProvider("https://api.node.glif.io/rpc/v1");
25. constmulticallAddress="0xcA11bde05977b3631167028862bE2a173976CA11";
26. constmulticallAbi=[/ *Multicall3 ABI* /];
27. constmulticall=newethers.Contract(multicallAddress,multicallAbi,provider);
28. // Example: Batch balance checks for multiple addresses
29. asyncfunctionbatchBalanceChecks(addresses) {
30. constcalls=addresses.map(address=>({
31. target:"0x...",// ERC20 token address
32. allowFailure:false,
33. callData:ethers.utils.id("balanceOf(address)").slice(0,10)+
34. ethers.utils.defaultAbiCoder.encode(["address"],[address]).slice(2)
35. }));
36. constresults=awaitmulticall.aggregate3(calls);
37. returnresults.map(result=>ethers.utils.defaultAbiCoder.decode(["uint256"],result.returnData)[0]);
38. }
39. batchBalanceChecks(["0x123...","0x456...","0x789..."]).then(console.log);
40. ```
41.

This example demonstrates how to use Multicall3 to batch multiplebalanceOf calls for an ERC20 token in a single transaction, significantly reducing the number of separate RPC calls needed.

Batch Contract Writes

⚠ Multicall3, while unaudited, can be safely used for batching on-chain writes when used correctly. As a stateless contract, it should never hold funds after a transaction ends, and users should never approve it to spend tokens.

When using Multicall3, it's crucial to understand two key aspects: the behavior ofmsg.sender in calls versus delegatecalls, and the risks associated withmsg.value in multicalls.

In FVM, there are two types of accounts: Externally Owned Accounts (EOAs) controlled by private keys, and Contract

Accounts controlled by code. Themsg.sender value during contract execution depends on whether a CALL or DELEGATECALL opcode is used. CALL changes the execution context, while DELEGATECALL preserves it.

For EOAs, which can only use CALL, Multicall3's address becomes themsg.sender for subsequent calls. This limits its usefulness from EOAs to scenarios wheremsg.sender is irrelevant . However, contract wallets or other contracts can use either CALL or DELEGATECALL, with the latter preserving the originalmsg.sender .

The handling ofmsg.value in multicalls requires caution. Sincemsg.value doesn't change with delegatecalls, relying on it within a multicall can lead to security vulnerabilities. To learn more about this, seehere andhere .

Hints

Lotus FEVM RPC supports Ethereum batch transactions. The key difference betweenmulticall and batch transactions is thatmulticall aggregates multiple RPC requests into a single call, while batch transactions are simply an array of transactions executed sequentially but sent in one request. For more details, please refer to theEthereum documentation .

Previous Oracles Next Multisig

Last updated6 days ago