Data Availability (DA) using Celestia

Introduction

Rollup reduces transaction costs and increases Ethereum's throughput by processing transactions off-chain. Rollup transactions are compressed and posted on L1 in batches. Batches represent thousands of individual off-chain transactions in a single transaction on L1. This reduces congestion on the base layer and reduces fees for users. However, it is only possible to trust the 'summary' transactions posted to L1 if the state change proposed can be independently verified and confirmed to be the result of applying all the individual off-chain transactions. If rollup operators do not make the transaction data available for this verification, then they could send incorrect data to L1. This problem is referred to as the data availability problem . In fact, if it is possible to verify that the data corresponding to a rollup block is available directly on the L1, then anyone can take the transaction data and recompute the correct rollup state. The data availability problem is particularly interesting for rollups as the rollup itself operates as an off-chain database and if the rollup operators do not post the data on a Data Availability layer such as the L1, then it becomes impossible to verify the rollup state.

A cheaper and more scalable rollup using Celestia as Data availability layer

The most obvious DA option is to use a base monolithic chain such as Ethereum. Full nodes in a monolithic network share and hold raw transaction data that gets included in each block by a validator. With that data, a full node can recompute the chain's state and compare it with any state root committed by a network validator. As a result, even in the absolute worst-case when the validator network is compromised and produces an invalid block, a full node can reject the block at the social layer. However, any monolithic chain that performs execution, consensus and ensures data availability all on one single P2P layer is often not optimised for either and hence can be costly if used for data availability needs. For context, Arbitrum One pays about USD 112k per day to Ethereum for its DA needs. This is about USD 0.15 per transaction on average, considering the average daily transaction volume on Arbitrum One. There are many applications, such as on-chain gaming, for which a transaction cost of USD 0.15 would be completely unacceptable. As a result, new types of networks optimized for data availability are being built. One such solution is Celestia. Celestia is a data availability (DA) layer that provides a scalable solution to the data availability problem . Two key features of Celestia's DA layer aredata availability sampling (DAS) and Namespaced Merkle trees (NMTs). Both features are novel blockchain scaling solutions: DAS enables light nodes to verify data availability without downloading an entire block; NMTs enable execution and settlement layers on Celestia to download transactions that are only relevant to them.

Integrating Celestia into AltLayer

At AltLayer, we are creating modular rollups. One example will be using Celestia as the data availability layer for rollup created by AltLayer rollups-as-a-service platform to lower the costs required for data availability.

Workflow

We have developed a service that performs the following workflow

1. Initialisation

The service first retrieves the genesis information and node version of the rollup. It posts both information into Celestiab (similar to step 4) and updates the smart contract with the block height at which the data is stored.

2. Retrieval of blocks data from L2 rollup

The goal here is to retrieve sufficient data from the L2 to commit to DA layer such that the L2 rollup can be reconstructed in a disastrous event. To do that, the data availability service will constantly fetch blocks and proof of validity in batches.

3. Compressing blocks data

To minimize data cost, block data are compressed using zlib compression library. To achieve better data compression efficiency, batch size can be increased. For example, compressing a batch of 2 blocks will result in a 48% data size reduction. If the batch size is increased to 100 blocks, the reduction increases to 67%.

4. Posting data to Celestia

To post data, * Set up a * Celestia light node * . Light nodes ensure data availability in particular, Light nodes ensure data availability by * Listens for new block headers and relevant DA metadata * Data availability sampling on the received headers * Specify a namespace, version and commitment * Post your data to Celestia DA using * blob.Submit execute (logger ,

async

(data)

=>

{ const blobArr =

[{ namespace :

CELESTIA_NODE_NAMESPACE , data :

"0x"

+ data . toString ("hex"), share_version :

CELESTIA_NODE_SHARE_VERSION , commitment :

CELESTIA_NODE_COMMITMENT , },]; const height =

await

jsonRPC (logger , CELESTIA_NODE_URL , "blob.Submit" , [blobArr], CELESTIA_NODE_AUTH_TOKEN); return height ; }). catch (logger . error); Sample query: curl

-X POST -H

"Content-Type: application/json"

-H

"Authorization: Bearer "

-d

'{ "id": 1, "jsonrpc": "2.0", "method": "blob.Submit", "params": [[{ "namespace": "AAAAAAAAAAAAAAAAAAAAAAAAAEJpDCBNOWAP3dM=", "data": "8fIMqAB+kQo7+LLmHaDya8oH73hxem6lQWX1", "share_version": 0, "commitment": "R4DRTAENQ7fGawPbt8aMse1+YReCZYg87xZIZf2fAxc=" }]] }' http://localhost:26658 * Celestia API will return the block height at which the data is posted

5. Tracking block height where data is stored

To track which L2 block number corresponds to the block height at which the data is posted to the smart contract will be deployed on L1. The smart contract will maintain a block number mapping to Celestia block height. To do that, the service calls set() on the smart contract and populate it with the related data. * key * : Start block number for this batch * value * : block number at which the data is stored into celetia * metadata_ * : End block number for this batch function

set (uint256 key , uint256 value , bytes

calldata metadata_)

external onlyOperator { emit

Set (key , value , metadata_ ,

_msgSender ()); // slither-disable-next-line unused-return _map . set (key , value); metadata [key]

= metadata_ ; latestKey = key ; } Full contract code: // SPDX-License-Identifier: UNLICENSED // SEE LICENSE IN https://files.altlayer.io/Alt-Research-License-1.md // Copyright Alt Research Ltd. 2023. All rights reserved. // // You acknowledge and agree that Alt Research Ltd. ("Alt Research") (or Alt // Research's licensors) own all legal rights, titles and interests in and to the // work, software, application, source code, documentation and any other documents pragma

solidity

^ 0.8.18 ; import

{ AccessControlUpgradeable }

from

"@openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol" ; import

{ EnumerableMapUpgradeable }

from

"@openzeppelin/contracts-upgradeable/utils/structs/EnumerableMapUpgradeable.sol" ; /// @title UintToUintMap /// @notice This contract allows mapping between uint256 keys and values using EnumerableMapUpgradeable utility from OpenZeppelin. /// @dev Contract also includes a role based access system. contract

UintToUintMap

is AccessControlUpgradeable { using

EnumerableMapUpgradeable

for EnumerableMapUpgradeable . UintToUintMap ; /// @notice Role identifier for operators. bytes32

public

constant OPERATOR_ROLE =

keccak256 ("OPERATOR_ROLE"); /// @dev Internal storage for the map. EnumerableMapUpgradeable . UintToUintMap private _map ; uint256

public latestKey ; uint256

public globalMetadata ; /// @notice Public metadata storage associated with each key. mapping (uint256

=>

bytes)

public metadata ; /// @dev This error is thrown when the lengths of the keys and values arrays are not equal in the set function. error LengthMismatch (); /// @dev This error is thrown when caller is not an operator. error NotOperator (); /// @notice Emitted when global metadata is set. event

SetGlobalMetadata (uint256 value ,

address sender); /// @notice Emitted when a key-value pair is set. event

Set (uint256

indexed key , uint256 value , bytes metadata , address sender); /// @custom:oz-upgrades-unsafe-allow constructor constructor ()

{ _disableInitializers (); } /// @dev Ensures only operators can call a function. modifier

onlyOperator ()

{ if

(! hasRole (OPERATOR_ROLE ,

_msgSender ()))

{ revert

NotOperator (); } _ ; } /// @notice Initializes the contract, sets roles of default admin and operator. /// @param initialDefaultAdmin Address of the first default admin. /// @param initialOperator Address of the first operator. function

initialize (address initialDefaultAdmin , address initialOperator)

external initializer { _grantRole (DEFAULT_ADMIN_ROLE , initialDefaultAdmin); _grantRole (OPERATOR_ROLE , initialOperator); } /// @notice Sets global metadata. /// @param value global metadata ID. function

setGlobalMetadata (uint256 value)

external onlyOperator { emit

SetGlobalMetadata (value ,

msgSender ()); globalMetadata = value ; } /// @notice Inserts a key-value pair into the map and emits a Set event. /// @param key The key to insert. /// @param value The value to insert. /// @param metadata The metadata associated with the key. function

set (uint256 key , uint256 value , bytes

calldata metadata_)

external onlyOperator { emit

Set (key , value , metadata_ ,

_msgSender ()); // slither-disable-next-line unused-return _map . set (key , value); metadata [key]

= metadata_ ; latestKey = key ; } /// @notice Returns the key-value pair stored at a given index in the map. /// @param index The index to retrieve the key-value pair from. /// @return The key and value at the given index. function

at (uint256 index)

external

view

returns

(uint256 ,

uint256)

{ return _map . at (index); } /// @notice Retrieves the value associated with a given key from the map. /// @param key The key to retrieve the value for. /// @return The value associated with the key. function

get (uint256 key)

external

view

returns

(uint256)

{ return _map . get (key); } /// @notice Returns the total number of key-value pairs in the map. /// @return The total number of key-value pairs. function

total ()

external

view

returns

(uint256)

{ return _map . length (); } }

6. Reconstruction of chain

From time to time, there may be a need to reconstruct back the chain. It can be due to * disastrous events causing the node operator to be down indefinitely * malicious node denying or returning invalid query results to the challenger In such a case, the rollup can be reconstructed 1. 1. 2. Retrieve genesis information and node version by calling 3. blob.GetAll 4. with the right namespace and block height curl

-X POST -H

"Content-Type: application/json"

-H

"Authorization: "

-d

'{ "id": 1, "jsonrpc": "2.0", "method": "blob.GetAll", "params": [110684, ["AAAAAAAAAAAAAAAAAAAAAAAAAEJpDCBNOWAP3dM="]] }' http://localhost:26658/ 1. 1. 2. Run the node with the correct version 3. 2. 4. Retrieve the mapping of the rollup block number to Celestia block height from the L1 smart contract 5. 3. 6. Download block data from Celestia by calling 7. blob.GetAll 8. with the right namespace and block height 9. 4. 10. Decompress the block data 11. 5. 12. import block data into the node 13. 6. 14. Repeat till the node is fully synced 15. 7. 16. Start the node with the required information External Integrations -Previous Account Abstraction using Biconomy Next-External Integrations Enabling permissionless interoperability on AltLayer Rollup with Hyperlane Last modified3mo ago On this page Introduction A cheaper and more scalable rollup using Celestia as Data availability layer Integrating Celestia into AltLayer Workflow