Integrate with Blobstream client

Blobstream demo rollup

Rollups can use Blobstream for DA by posting their data to Celestia and then proving that it was posted on Ethereum. This is done identically to how any rollup or user would post data to Celestia. Then, a zero-knowledge proof that Celestia validators have come to consensus on Celestia block headers is generated, and subsequently relayed to Ethereum to the Blobstream X smart contract.

This demo rollup will outline (the outline is not an implementation! please do not expect to copy and paste this code) a Blobstream rollup to illustrate at a high level what this could look like.

Defining a chain

The first step to starting a new chain is to define the structure of the commitments that each block consists of.

go type

Block

struct { // Data is the data of a block that is submitted to Celestia. Data json:"Data" // Header is the set of commitments over a block that is submitted to // Ethereum. Header json:"Header" }

// Data is the data of a block that is submitted to Celestia. type

Data

struct { Txs []json.RawMessage json:"txs" }

// Header is the set of commitments over a block that is submitted to Ethereum. type

Header

struct { Height uint64

json:"height" Namespace [] byte

json:"namespace" PreviousHash [] byte

json:"previous_hash" Span Span json:"span" SequencerSignature Signature json:"sequencer_signature,omitempty" } type

Block

struct { // Data is the data of a block that is submitted to Celestia. Data json:"Data" // Header is the set of commitments over a block that is submitted to // Ethereum. Header json:"Header" }

// Data is the data of a block that is submitted to Celestia. type

Data

struct { Txs []json.RawMessage json:"txs" }

// Header is the set of commitments over a block that is submitted to Ethereum. type

Header

struct { Height uint64

json:"height" Namespace [] byte

json:"namespace" PreviousHash [] byte

json:"previous_hash" Span Span json:"span" SequencerSignature Signature json:"sequencer_signature,omitempty" } Note the Celestia-specific structures in the header such as theNamespace and the Blobstream-specific structure called theSpan . The goal of these structures are to locate the data in the Celestia block so that we can prove that data's inclusion via Blobstream if needed. Read more in thenamespace specifications , and you can think of this like a chain ID. Learn more information aboutshares , which are small pieces of the encoded Celestia block. We use the same encoding here so that the commitments to the rollup block match those committed to by validators in the Celestia data root.

TheSpan could take many forms, but in this demo, we will use the following:

go // Span describes the location of the rollup block data that is posted to // Celestia. This is important for other nodes to be able to prove that data in // the Celestia block. This can be thought of as a pointer to some data in the // Celestia block. type

Span

struct { // CelestiaHeight is the height of the Celestia block that contains the // rollup block data. CelestiaHeight uint64

json:"celestia_height" // DataShareStart is the index of the first share of the rollup block data. DataShareStart uint64

json:"share_start" // DataShareLen is length in shares of the rollup block data. This is used // to identify all of the rollup block data in a Celestia block. DataShareLen uint64

json:"share_end" } // Span describes the location of the rollup block data that is posted to // Celestia. This is important for other nodes to be able to prove that data in // the Celestia block. This can be thought of as a pointer to some data in the // Celestia block. type

Span

struct { // CelestiaHeight is the height of the Celestia block that contains the // rollup block data. CelestiaHeight uint64

json:"celestia_height" // DataShareStart is the index of the first share of the rollup block data. DataShareStart uint64

json:"share_start" // DataShareLen is length in shares of the rollup block data. This is used // to identify all of the rollup block data in a Celestia block. DataShareLen uint64

json:"share_end" } We can then define the blockchain as a collection of blocks and some additional information about the chain such as the sequencer address.

go type

Blockchain

struct { Blocks []Block SequencerAddress [] byte Namespace [] byte } type

Blockchain

struct { Blocks []Block SequencerAddress [] byte Namespace [] byte }

Rollup sequencer

The rollup sequencer is responsible for creating blocks and, in this demo, writing that data to Celestia and Ethereum. The rollup full node is responsible for reading that data from Celestia and Ethereum and verifying that it follows the protocol rules of that rollup.

Therefore, we can start by first defining the reading and writing interactions rollup nodes will have with both the Celestia and Ethereum networks. The actual implementations of these interfaces are left as exercises to the reader ( ). Assume that those implementations of these interfaces are verifying the respective chain. For the connection to Celestia, this would likely mean connecting to a Celestia light node, which can detect faults in consensus such as hidden data. For the connection to Ethereum, this would likely mean running and connecting to a full node. More information on the RPC that is exposed by a Celestia light node can be foundin the RPC documentation . Additionally, if you need more information on how to run a light node, you canheck out the documentation .

go // CelestiaLightNodeClient summarizes the actions that a rollup that uses // Blobstream for DA would need from a Celestia light node. Note that the actual // connection to this light node is arbitrary, but would likely involve an RPC // connection to a Celestia light node. type

CelestiaLightNodeClient

interface { GetBlockData (Span) (Data, error) SubmitBlockData (Data) (Span, error) }

// EthereumClient summarizes the actions that a rollup that uses Blobstream for // DA would need from an Ethereum client. type

EthereumClient

interface { // GetLatestRollupHeight returns the height of the latest rollup block by // querying the appropriate contract on Ethereum. LatestRollupHeight () (uint64 , error) // GetHeader returns the rollup header of a specific height. GetHeader (uint64) (Header, error) // SubmitHeader submits a header to the rollup bridge contract on Ethereum. SubmitHeader (Header) error } // CelestiaLightNodeClient summarizes the actions that a rollup that uses // Blobstream for DA would need from a Celestia light node. Note that the actual // connection to this light node is arbitrary, but would likely involve an RPC // connection to a Celestia light node. type

CelestiaLightNodeClient

interface { GetBlockData (Span) (Data, error) SubmitBlockData (Data) (Span, error) }

// EthereumClient summarizes the actions that a rollup that uses Blobstream for // DA would need from an Ethereum client. type

EthereumClient

interface { // GetLatestRollupHeight returns the height of the latest rollup block by // querying the appropriate contract on Ethereum. LatestRollupHeight () (uint64 , error) // GetHeader returns the rollup header of a specific height. GetHeader (uint64) (Header, error) // SubmitHeader submits a header to the rollup bridge contract on Ethereum. SubmitHeader (Header) error } Note that here we are waiting for the head to be posted to Ethereum, however, it would likely be better to simply download that header from the p2p network or directly from the sequencer instead.

For the purposes of this demo, we will be using a single centralized sequencer, which can be defined by simply wrapping the full node to isolate the logic to create blocks.

A rollup full node will just consist of some representation of a blockchain along with clients to read from with Celestia and Ethereum.

go type

Fullnode

struct { Blockchain CelestiaLightNodeClient EthereumClient }

// Sequencer wraps the demo Fullnode struct to add specific functionality for // producing blocks. type

Sequencer

struct { Fullnode } type

Fullnode

struct { Blockchain CelestiaLightNodeClient EthereumClient }

// Sequencer wraps the demo Fullnode struct to add specific functionality for // producing blocks. type

Sequencer

struct { Fullnode }

Committing to data

Typical blockchains commit to the transactions included in each block using a Merkle root. Rollups that use Blobstream for DA need to use the commitments that are relayed to the Blobstream contracts.

For optimistic rollups, this could be as simple as referencing the data in the Celestia block, not unlike using a pointer in memory. This is what is done below via aSpan in thecreating blocks section. We keep track of where the data is located in the Celestia block and the sequencer signs over that location in the header. If the sequencer commits to non-existent data or an invalid state root, then the invalid transaction is first proved to be included in theSpan before the rest of the fraud proof process is followed. Find more informationin the inclusion proofs documentation .

For zk rollups, this would involve creating an inclusion proof to the data root tuple root in the Blobstream contracts and then verifying that proof in the zk proof used to verify state. Find more information in thedata root inclusion proof documentation .

Also, see the documentation for thedata square layout and theshares of the Celestia block to see how the data is encoded in Celestia.

Creating blocks

The first step in creating a block is to post the block data to Celestia. Upon confirmation of the data being included in a block, the actual location of the data in Celestia can be determined. This data is used to create aSpan which is included in the header and signed over by the sequencer. ThisSpan can be used by contracts on Ethereum that use the Blobstream contracts to prove some specific data was included.

go func (s * Sequencer) ProduceBlock (txs []json.RawMessage) (Block, error) { data := Data{Txs: txs}

span, err := s.CelestiaLightNodeClient. SubmitBlockData (data) if err !=

nil { return Block{}, err }

var lastBlock Block if

len (s.Blocks)

0 { lastBlock = s.Blocks[len (s.Blocks) - 1] }

header := Header{ Height: uint64 (len (s.Blocks) +

1), PreviousHash: lastBlock.Header. Hash (), Namespce: s.Namespace, Span: span, }

signature := s.key. Sign (header. SignBytes ())

header.SequencerSignature

signature

block := Block{ Data: data, Header: header, }

s. AddBlock (block)

return block, nil } func (s * Sequencer) ProduceBlock (txs []json.RawMessage) (Block, error) { data := Data{Txs: txs}

span, err := s.CelestiaLightNodeClient. SubmitBlockData (data) if err !=

nil { return Block{}, err }

var lastBlock Block if

len (s.Blocks)

0 { lastBlock = s.Blocks[len (s.Blocks) - 1] }

header := Header{ Height: uint64 (len (s.Blocks) +

1), PreviousHash: lastBlock.Header. Hash (), Namespce: s.Namespace, Span: span, }

signature := s.key. Sign (header. SignBytes ())

header.SequencerSignature

signature

block := Block{ Data: data, Header: header, }

s. AddBlock (block)

return block, nil } Note that the sequencer here is not yet posting headers to Ethereum. This is because the sequencer is waiting for the commitments from the Celestia validator set (the data root tuple roots) to be relayed to the contracts. Once the contracts are updated, the sequencer can post the header to Ethereum.

go func (s * Sequencer) UpdateHeaders () error { latestRollupHeight, err := s.EthereumClient. LatestRollupHeight () if err !=

nil { return err }

for i := latestRollupHeight; i <=

uint64 (len (s.Blocks) + 1); i ++ { err := s.EthereumClient. SubmitHeader (s.Blocks[i].Header) if err !=

nil { return err } }

return

nil } func (s * Sequencer) UpdateHeaders () error { latestRollupHeight, err := s.EthereumClient. LatestRollupHeight () if err !=

nil { return err }

for i := latestRollupHeight; i <=

uint64 (len (s.Blocks) + 1); i ++ { err := s.EthereumClient. SubmitHeader (s.Blocks[i].Header) if err !=

nil { return err } }

return

nil }

Rollup full node

Downloading the block

There are a few different mechanisms that could be used to download blocks. The simplest solution and what is outlined above is forFullnodes to wait until the blocks and the headers are posted to the respective chains, and then download each as they are posted. It would also be possible to gossip the headers ahead of time and download the rollup blocks from Celestia instead of waiting for the headers to be posted to Ethereum. It's also possible to download the headers and the block data like a normal blockchain via a gossiping network and only fall back to downloading the data and headers from Celestia and Ethereum if the gossiping network is unavailable or the sequencer is malicious.

go func (f * Fullnode) AddBlock (b Block) error { // Perform validation of the block if b.Header.Height !=

uint64 (len (f.Blocks) + 1) { return fmt. Errorf ("failure to add block: expected block height %d , got %d " , len (f.Blocks) + 1 , b.Header.Height) } // Check the sequencer's signature if

! b.Header.SequencerSignature. IsValid (f.SequencerAddress) { return fmt. Errorf ("failure to add block: invalid sequencer signature") }

f.Blocks

append (f.Blocks, b) return

nil }

func (f * Fullnode) GetLatestBlock () error { nextHeight :=

uint64 (len (f.Blocks) +

1)

// Download the next header from etheruem before we download the block data // from Celestia. Note that we could alternatievely download the header // directly from the sequencer instead of waiting. header, err := f.EthereumClient. GetHeader (nextHeight) if err !=

nil { return err }

data, err := f.CelestiaLightNodeClient. GetBlockData (header.Span) if err !=

nil { return err }

return f. AddBlock (Block{ Data: data, Header: header, },) } func (f * Fullnode) AddBlock (b Block) error { // Perform validation of the block if b.Header.Height !=

uint64 (len (f.Blocks) + 1) { return fmt. Errorf ("failure to add block: expected block height %d , got %d " , len (f.Blocks) + 1 , b.Header.Height) } // Check the sequencer's signature if

! b.Header.SequencerSignature. IsValid (f.SequencerAddress) { return fmt. Errorf ("failure to add block: invalid sequencer signature") }

f.Blocks

append (f.Blocks, b) return

nil }

func (f * Fullnode) GetLatestBlock () error { nextHeight :=

uint64 (len (f.Blocks) +

1)

// Download the next header from etheruem before we download the block data // from Celestia. Note that we could alternatievely download the header // directly from the sequencer instead of waiting. header, err := f.EthereumClient. GetHeader (nextHeight) if err !=

nil { return err }

data, err := f.CelestiaLightNodeClient. GetBlockData (header.Span) if err !=

nil { return err }

return f. AddBlock (Block{ Data: data, Header: header, },) } This outline of a Blobstream rollup isn't doing execution or state transitions induced by the transactions, however, that step would occur here. If fraud is detected, the fraud proof process will begin. The only difference between the fraud proof process of a normal optimistic rollup and a rollup that uses Blobstream for DA is that the full node would first prove the fraudulent transaction was committed to by the Sequencer using theSpan in the header and before proceeding with the normal process.

More documentation

Proving inclusion via Blobstream

Blobstream inclusion proof docs and theverifier helper contracts.

Submitting block data to Celestia via light node

As linked above, use theCelestia light node RPC to submit the data to Celestia.

Posting headers to Ethereum

How headers are posted to Ethereum is entirely dependent upon how the rollup light client contracts work. For examples of interacting with the Ethereum blockchain programmatically, please see thego-ethereum book or one of the many other resources forsubmitting transactions orwriting contracts . [][Edit this page on GitHub] Last updated: Previous page Integrate with Blobstream contracts Next page Querying the Blobstream proofs []