Submitting data blobs to Celestia

To submit data to Celestia, users submit blob transactions (BlobTx). Blob transactions contain two components, a standard Cosmos-SDK transaction calledMsgPayForBlobs and one or moreBlob s of data.

Maximum blob size

The maximum total blob size in a transaction is just under2 MiB (1,973,786 bytes) , based on a 64x64 share grid (4096 shares). With one share for the PFB transaction, 4095 shares remain: 1 at 478 bytes and 4094 at 482 bytes each.

This is subject to change based on governance parameters. Learn more orthe Mainnet Beta page under "Maximum bytes" .

It is advisable to submit transactions where the total blob size is significantly smaller than 1.8 MiB (e.g. 500 KiB) in order for your transaction to get included in a block quickly. If a tx contains blobs approaching 1.8 MiB then there will be no room for any other transactions. This means that your transaction will only be included in a block if it has a higher gas price than every other transaction in the mempool.

Fee market and mempool

Celestia makes use of a standard gas-priced prioritized mempool. By default, transactions with gas prices higher than that of other transactions in the mempool will be prioritized by validators.

Fees and gas limits

As of version v1.0.0 of the application (celestia-app), there is no protocol enforced minimum fee (similar to EIP-1559 in Ethereum). Instead, each consensus node running a mempool uses a locally configured gas price threshold that must be met in order for that node to accept a transaction, either directly from a user or gossiped from another node, into its mempool.

As of version v1.0.0 of the application (celestia-app), gas is not refunded. Instead, transaction fees are deducted by a flat fee, originally specified by the user in their tx (where fees = gasLimit * gasPrice). This means that users should use an accurate gas limit value if they do not wish to overpay.

Under the hood, fees are currently handled by specifying and deducting a flat fee. However gas price is often specified by users instead of calculating the flat fee from the gas used and the gas price. Since the state machine does not refund users for unused gas, gas price is calculated by dividing the total fee by the gas limit.

Estimating PFB gas

Generally, the gas used by a PFB transaction involves a static fixed cost and a dynamic cost based on the size of each blob in the transaction.

NOTE

For a general use case of a normal account submitting a PFB, the static costs can be treated as such. However, due to the description above of how gas works in the Cosmos-SDK this is not always the case. Notably, if a vesting account or thefeegrant modules are used, then these static costs change. The fixed cost is an approximation of the gas consumed by operations outside the functionGasToConsume (for example, signature verification, tx size, read access to accounts), which has a default value of 65,000 gas.

NOTE

The first transaction sent by an account (sequence number == 0) has an additional one time gas cost of 10,000 gas. If this is the case, this should be accounted for. Each blob in the PFB contributes to the total gas cost based on its size. The functionGasToConsume calculates the total gas consumed by all the blobs involved in a PFB, where each blob's gas cost is computed by first determining how many shares are needed to store the blob size. Then, it computes the product of the number of shares, the number of bytes per share, and thegasPerByte parameter. Finally, it adds a static amount per blob.

Theblob.GasPerBlobByte andauth.TxSizeCostPerByte are parameters that could potentially be adjusted through the system's governance mechanisms. Hence, actual costs may vary depending on the current state of these parameters.

Gas fee calculation

The total fee for a transaction is calculated as the product of the gas limit for the transaction and the gas price set by the user:

Total Fee

Gas Limit × Gas Price

The gas limit for a transaction is the maximum amount of gas that a user is willing to spend on a transaction. It is determined by both a static fixed cost (FC) and a variable dynamic cost based on the size of each blob involved in the transaction:

Gas Limit

F C + ∑ i = 1 n S S N (B i) × S S × G C P B B

Where:

- F
- C
- = Fixed Cost, is a static value (65,000 gas)
- ∑
- i
- =
- 1
- n
- S
- S
- N
- (
- B
- i
-)
- = SparseSharesNeeded for thei
- th Blob, is the number of shares needed for thei
- th blob in the transaction
- S
- S
- = Share Size, is the size of each share
- G
- C
- P
- B
- B
- = Gas Cost Per Blob Byte, is a parameter that could potentially be adjusted through the system's governance mechanisms.

The gas fee is set by the user when they submit a transaction. The fee is often specified by users directly. The total cost for the transaction is then calculated as the product of the estimated gas limit and the gas price. Since the state machine does not refund users for unused gas, it's important for users to estimate the gas limit accurately to avoid overpaying.

For more details on how gas is calculated per blob, refer to thePayForBlobs function that consumes gas based on the blob sizes. This function uses theGasToConsume function to calculate the extra gas charged to pay for a set of blobs in aMsgPayForBlobs transaction. This function calculates the total shares used by all blobs and multiplies it by theShareSize andgasPerByte to get the total gas to consume.

For estimating the total gas required for a set of blobs, refer to theEstimateGas function . This function estimates the gas based on a linear model that is dependent on the governance parameters:gasPerByte andtxSizeCost . It assumes other variables are constant, including the assumption that theMsgPayForBlobs is the only message in the transaction. TheDefaultEstimateGas function runsEstimateGas with the system defaults.

Estimating gas programmatically

Users can estimate an efficient gas limit by using this function:

go import (blobtypes

" github.com/celestiaorg/celestia-app/x/blob/types ") gasLimit := blobtypes. DefaultEstimateGas ([] uint32 { uint32 (sizeOfDataInBytes)}) import (blobtypes

" github.com/celestiaorg/celestia-app/x/blob/types ") gasLimit := blobtypes. DefaultEstimateGas ([] uint32 { uint32 (sizeOfDataInBytes)}) If using a celestia-node light client, then this function is automatically called for you when submitting a

blob. This function works by breaking down the components of calculating gas for a blob transaction. These components consist of a flat costs for all PFBs, the size of each blob and how many shares each uses and the parameter for gas used per byte. More information about how gas is used can be found in thegas specs and the exact formula can be found in theblob module .

Submitting multiple transactions in one block from the same account

The mempool Celestia uses works by maintaining a fork of the canonical state each block. This means that each time we submit a transaction to it, it will update the sequence number (aka nonce) for the account that submitted the transaction. If users wish to submit a second transaction, they can, but must specify the nonce manually. If this is not done, the new transactions will not be able to be submitted until the first transaction is reaped from the mempool (i.e. included in a block), or dropped due to timing out.

By default, nodes will drop a transaction if it does not get included in 5 blocks (roughly 1 minute). At this point, the user must resubmit their transaction if they want it to eventually be included.

As of v1.0.0 of the application (celestia-app), users are unable to replace an existing transaction with a different one with higher fees. They must instead wait 5 blocks from the original submitted time and then resubmit the transaction. Again, community members have already suggested solutions and a willingness to accept changes to fix this issue.

API

Users can currently create and submitBlobTx s in six ways.

The celestia-app consensus node CLI

bash celestia-appd

tx

blob

PayForBlobs

< hex-encoded

namespac e

< hex-encoded

dat a

[flags] celestia-appd

tx

blob

PayForBlobs

< hex-encoded

namespac e

< hex-encoded

dat a

[flags]

The celestia-node light node CLI

Usingblob.Submit :

bash celestia

blob

submit

< hex-encoded

namespac e

< hex-encoded

dat a

celestia

blob

submit

< hex-encoded

namespac e

< hex-encoded

dat a

Learn more in thenode tutorial .

The celestia-node API golang client

For more celestia-node API golang examples, refer to thegolang client tutorial .

go import (" bytes " " context " " fmt "

client

" github.com/celestiaorg/celestia-openrpc " " github.com/celestiaorg/celestia-openrpc/types/blob " " github.com/celestiaorg/celestia-openrpc/types/share ")

// SubmitBlob submits a blob containing "Hello, World!" to the 0xDEADBEEF namespace. It uses the default signer on the running node. func

SubmitBlob (ctx context.Context, url string , token string) error { client, err := client. NewClient (ctx, url, token) if err !=

nil { return err }

// let's post to 0xDEADBEEF namespace namespace, err := share. NewBlobNamespaceV0 ([] byte { 0x DE , 0x AD , 0x BE , 0x EF }) if err !=

nil { return err }

// create a blob helloWorldBlob, err := blob. NewBlobV0 (namespace, [] byte ("Hello, World!")) if err !=

nil { return err }

// submit the blob to the network height, err := client.Blob. Submit (ctx, [] * blob.Blob{helloWorldBlob}, blob. DefaultGasPrice ()) if err !=

nil { return err }

fmt. Printf ("Blob was included at height %d\n " , height)

// bonus: fetch the blob back from the network retrievedBlobs, err := client.Blob. GetAll (ctx, height, []share.Namespace{namespace}) if err !=

nil { return err }

fmt. Printf ("Blobs are equal? %v\n " , bytes. Equal (helloWorldBlob.Commitment, retrievedBlobs[0].Commitment)) return

nil }} import (" bytes " " context " " fmt "

client

" github.com/celestiaorg/celestia-openrpc " " github.com/celestiaorg/celestia-openrpc/types/blob " " github.com/celestiaorg/celestia-openrpc/types/share ")

// SubmitBlob submits a blob containing "Hello, World!" to the 0xDEADBEEF namespace. It uses the default signer on the running node. func

SubmitBlob (ctx context.Context, url string , token string) error { client, err := client. NewClient (ctx, url, token) if err !=

nil { return err }

// let's post to 0xDEADBEEF namespace namespace, err := share. NewBlobNamespaceV0 ([] byte { 0x DE , 0x AD , 0x BE , 0x EF }) if err !=

nil { return err }

// create a blob helloWorldBlob, err := blob. NewBlobV0 (namespace, [] byte ("Hello, World!")) if err !=

nil { return err }

// submit the blob to the network height, err := client.Blob. Submit (ctx, [] * blob.Blob{helloWorldBlob}, blob. DefaultGasPrice ()) if err !=

nil { return err }

fmt. Printf ("Blob was included at height %d\n " , height)

// bonus: fetch the blob back from the network retrievedBlobs, err := client.Blob. GetAll (ctx, height, []share.Namespace{namespace}) if err !=

nil { return err }

fmt. Printf ("Blobs are equal? %v\n " , bytes. Equal (helloWorldBlob.Commitment, retrievedBlobs[0].Commitment)) return

nil }}

GRPC to a consensus node via theuser

package

go import (" context " " fmt "

" github.com/celestiaorg/celestia-app/app " " github.com/celestiaorg/celestia-app/app/encoding " " github.com/celestiaorg/celestia-app/pkg/appconsts " " github.com/celestiaorg/celestia-app/pkg/namespace " " github.com/celestiaorg/celestia-app/pkg/user " blobtypes

" github.com/celestiaorg/celestia-app/x/blob/types " " github.com/cosmos/cosmos-sdk/crypto/keyring " tmproto

" github.com/tendermint/tendermint/proto/tendermint/types " " google.golang.org/grpc " " google.golang.org/grpc/credentials/insecure ")

// SubmitData is a demo function that shows how to use the signer to submit data // to the blockchain directly via a celestia node. We can manage this keyring // using the celestia-appd keys or celestia keys sub commands and load this // keyring from a file and use it to programmatically sign transactions. func

DemoSubmitData (grpcAddr string , kr keyring.Keyring) error { // create an encoding config that can decode and encode all celestia-app // data structures. ecfg := encoding. MakeConfig (app.ModuleEncodingRegisters ...)

// create a connection to the grpc server on the consensus node. conn, err := grpc. Dial (grpcAddr, grpc. WithTransportCredentials (insecure. NewCredentials ())) if err !=

nil { return err } defer conn. Close ()

// get the address of the account we want to use to sign transactions. rec, err := kr. Key ("accountName") if err !=

nil { return err }

addr, err := rec. GetAddress () if err !=

nil { return err }

// Setup the signer. This function will automatically query the relevant // account information such as sequence (nonce) and account number. signer, err := user. SetupSigner (context. TODO (), kr, conn, addr, ecfg) if err !=

nil { return err }

ns := namespace. MustNewV0 ([] byte ("1234567890"))

fmt. Println ("namespace" , len (ns. Bytes ()))

blob, err := blobtypes. NewBlob (ns, [] byte ("some data"), appconsts.ShareVersionZero) if err !=

nil { return err }

gasLimit := blobtypes. DefaultEstimateGas ([] uint32 { uint32 (len (blob.Data))})

options := []user.TxOption{ // here we're setting estimating the gas limit from the above estimated // function, and then setting the gas price to 0.1utia per unit of gas. user. SetGasLimitAndFee (gasLimit, 0.1), }

// this function will submit the transaction and block until a timeout is // reached or the transaction is committed. resp, err := signer. SubmitPayForBlob (context. TODO (), [] * tmproto.Blob{blob}, options ...) if err !=

nil { return err }

// check the response code to see if the transaction was successful. if resp.Code !=

0 { // handle code fmt. Println (resp.Code, resp.Codespace, resp.RawLog) }

// if we don't want to wait for the transaction to be confirmed, we can // manually sign and submit the transaction using the same package. blobTx, err := signer. CreatePayForBlob ([] * tmproto.Blob{blob}, options ...) if err !=

nil { return err }

resp, err

signer. BroadcastTx (context. TODO (), blobTx) if err !=

nil { return err }

// check the response code to see if the transaction was successful. Note // that this time we're not waiting for the transaction to be committed. // Therefore the code here is only from the consensus node's mempool. if resp.Code !=

0 { // handle code fmt. Println (resp.Code, resp.Codespace, resp.RawLog) }

return err } import (" context " " fmt "

" github.com/celestiaorg/celestia-app/app " " github.com/celestiaorg/celestia-app/app/encoding " " github.com/celestiaorg/celestia-app/pkg/appconsts " " github.com/celestiaorg/celestia-app/pkg/namespace " " github.com/celestiaorg/celestia-app/pkg/user " blobtypes

" github.com/celestiaorg/celestia-app/x/blob/types " " github.com/cosmos/cosmos-sdk/crypto/keyring " tmproto

" github.com/tendermint/tendermint/proto/tendermint/types " " google.golang.org/grpc " " google.golang.org/grpc/credentials/insecure ")

// SubmitData is a demo function that shows how to use the signer to submit data // to the blockchain directly via a celestia node. We can manage this keyring // using the celestia-appd keys or celestia keys sub commands and load this // keyring from a file and use it to programmatically sign transactions. func

DemoSubmitData (grpcAddr string , kr keyring.Keyring) error { // create an encoding config that can decode and encode all celestia-app // data structures. ecfg := encoding. MakeConfig (app.ModuleEncodingRegisters ...)

// create a connection to the grpc server on the consensus node. conn, err := grpc. Dial (grpcAddr, grpc. WithTransportCredentials (insecure. NewCredentials ())) if err !=

nil { return err } defer conn. Close ()

// get the address of the account we want to use to sign transactions. rec, err := kr. Key ("accountName") if err !=

nil { return err }

addr, err := rec. GetAddress () if err !=

nil { return err }

// Setup the signer. This function will automatically query the relevant // account information such as sequence (nonce) and account number. signer, err := user. SetupSigner (context. TODO (), kr, conn, addr, ecfg) if err !=

nil { return err }

ns := namespace. MustNewV0 ([] byte ("1234567890"))

fmt. Println ("namespace" , len (ns. Bytes ()))

blob, err := blobtypes. NewBlob (ns, [] byte ("some data"), appconsts.ShareVersionZero) if err !=

nil { return err }

gasLimit := blobtypes. DefaultEstimateGas ([] uint32 { uint32 (len (blob.Data))})

options := []user.TxOption{ // here we're setting estimating the gas limit from the above estimated // function, and then setting the gas price to 0.1utia per unit of gas. user. SetGasLimitAndFee (gasLimit, 0.1), }

// this function will submit the transaction and block until a timeout is // reached or the transaction is committed. resp, err := signer. SubmitPayForBlob (context. TODO (), [] * tmproto.Blob{blob}, options ...) if err !=

nil { return err }

// check the response code to see if the transaction was successful. if resp.Code !=

0 { // handle code fmt. Println (resp.Code, resp.Codespace, resp.RawLog) }

// if we don't want to wait for the transaction to be confirmed, we can // manually sign and submit the transaction using the same package. blobTx, err := signer. CreatePayForBlob ([] * tmproto.Blob{blob}, options ...) if err !=

nil { return err }

resp, err

signer. BroadcastTx (context. TODO (), blobTx) if err !=

nil { return err }

// check the response code to see if the transaction was successful. Note // that this time we're not waiting for the transaction to be committed. // Therefore the code here is only from the consensus node's mempool. if resp.Code !=

0 { // handle code fmt. Println (resp.Code, resp.Codespace, resp.RawLog) }

return err }

RPC to a celestia-node

Using the JSON RPC API, submit data using the following methods:

- blob.Submit
- state.SubmitPayForBlob

Learn more in thecelestia-node API docs .

Post a blob directly from Celenium

Celenium provides a user-friendly interface to view and interact with data on Celestia, and allows for submitting blobs directly from the explorer interface.

To submit a blob from Celenium, follow these steps:

1. Navigate to theCelenium explorer
2. and connect your wallet.
3. Click on the terminal button in the top right corner of the screen and select the "Submit data blob" option.
4. Next, ensure the file you are submitting is in a supported format and upload it.
5. In the "Namespace" field, input the namespace you want to use for the blob in hex format.
6. Finally, click on the "Continue" button to submit your blob, then approve the transaction in your wallet.

Once the blob is submitted, you will see its hash on the screen. You can also use Celenium’s search bar to search for the blob's hash and view its details. [][Edit this page on GitHub] Last updated: Previous page Blobstream rollups Next page FeeGrant module for blobs submission []