Introduction

The correctness of the data is a crucial part of decentralized applications (dApps) that rely on data from the real world. Manipulated data can impact the security and the objective of the applications.

Thecw-band introduces a standard for anyone looking to integrate data from Band's oracle into their CosmWasm smart contract on a Cosmos-SDK-based blockchain throughInter-Blockchain Communication (IBC) . The standard itself consists of data types that require for requesting and receiving data.

Workflow

At a high level, the workflow will be as follows. First, the smart contract creates an IBC packet to request data from BandChain. Then, relayers will pick up the IBC packet and relay it on BandChain.

After BandChain processes the request, it will send an acknowledgement message along withrequest_id back. And, when the result of the request is finalized, BandChain will send a new IBC packet that contains the final data back. Relayers will listen and pick up those packets and relay them to the smart contract.

At this point, the smart contract can use the data from BandChain in its smart contract securely. If the smart contract wants to have new or more data, it can always send a new IBC packet and start the process again at any time.

cw-band

Our library (cw-band) provides data types and functions that are necessary for requesting and receiving data from BandChain. It will help developers to easily integrate their CosmWasm smart contract with BandChain.

Here are the data types and functions that the library provides for you.

Version

First, the library provides the version of the IBC channel of the Oracle module.

pub

const

IBC_APP_VERSION :

& str

=

"bandchain-1" ; Packet

The library also provides the types of packets that will be sent through IBC.

// Type of request packet data that will accept

[cw_serde]

pub

struct

OracleRequestPacketData

{ pub client_id :

String ,

// The client_id that you want to send. This will help you to identify the client that request. pub oracle_script_id :

Uint64 ,

// The oracle_script_id that you want to request data from. pub calldata :

Binary ,

// The data that you want to send to your oracle script. pub ask_count :

Uint64 ,

// Number of validators that you want to provide you data. pub min_count :

Uint64 ,

// Minimum number of validators that you will accept the final result. pub fee_limit :

Vec < Coin

,

// The fee limit of the data sources that you are willing to pay for the data. pub prepare_gas :

Uint64 ,

// Prepare gas that you will use in preparation process of the oracle script. pub execute_gas :

Uint64 ,

// Execution gas that you will use in execution process of the oracle script. }

// Type of response packet data from BandChain

[cw_serde]

pub

struct

OracleResponsePacketData

{ pub client_id :

String ,

// The client_id that you send in the OracleRequestPacketData. pub request_id :

Uint64 ,

// The id of the request of this response on BandChain. pub ans_count :

Uint64 ,

// The number of validators that helps provide this result. pub request_time :

Uint64 ,

// The time that this request is accepted on BandChain. pub resolve_time :

Uint64 ,

// The time that this request is resolved on BandChain. pub resolve_status :

ResolveStatus ,

// The status of resolving the request. pub result :

Binary ,

// The result of the request. }

// Enum of possible resolve_status that BandChain will send

[cw_serde]

pub

enum

ResolveStatus

{

[serde(rename =

"RESOLVE_STATUS_OPEN_UNSPECIFIED")] Open ,

// The request is not resolved yet.

[serde(rename =

"RESOLVE_STATUS_SUCCESS")] Success ,

// The request is resolved successfully.

[serde(rename =

"RESOLVE_STATUS_FAILURE")] Failure ,

// The request is failed to solve.

[serde(rename =

"RESOLVE_STATUS_EXPIRED")] Expired ,

// The request is expired before solving. }

// IBC Acknowledgement message (either result or error)

[cw_serde]

pub

enum

AcknowledgementMsg

{ Result (Binary) , Error (String) , }

// create a serialized success message pub

fn

ack_success ( )

->

Binary

{ let res =

AcknowledgementMsg :: Result (b"1" . into ( )) ; to_binary (& res) . unwrap ( ) }

// create a serialized error message pub

fn

ack_fail (err :

String)

->

Binary

{ let res =

AcknowledgementMsg :: Error (err) ; to_binary (& res) . unwrap ( ) }

// Acknowledge message that BandChain will send after receives the request.

[cw_serde]

pub

struct

BandAcknowledgement

{ pub request_id :

Uint64 ,

// The request_id of your request. } Oracle script - Price

This is the example of the Input and Output of the oracle script for getting the rates of symbols.

// Input of the oracle script.

[derive(OBIEncode)]

pub

struct

Input

{ pub symbols :

Vec < String

,

// Symbol list that you want to request. pub minimum_sources :

u8 ,

// Minimum sources }

// Output of the oracle script.

[derive(OBIDecode)]

pub

struct

Output

{ pub responses :

Vec < Response

,

// Arrary of response of each each symbol. }

// Rate of the symbol.

[derive(OBIDecode)]

pub

struct

Response

{ pub symbol :

String ,

// symbol. pub response_code :

u8 ,

// response_code. pub rate :

u64 ,

// the price of the symbol. }

Learn more

- IBC docs
- IBC tutorials Previous Example Use Cases Next Getting started