# CosmWasm + ICA

This section contains a tutorial for writing smart contracts that utilize interchain accounts.

## Overview

We are going to learn how to:

1. Install dependencies and import the libraries;
2. Register an Interchain Account;
3. Execute an interchain transaction.

Note: Neutron provides an implementation of an ICA controllermodule , which simplifies the creation and management of interchain accounts for smart contract developers. This module, however, is not your only option; you can use raw IBC packets to imitate theibc-go implementation, or use a framework that already implements the same logic on smart contracts level. Note: this section assumes that you have basic knowledge of CosmWasm and have some experience in writing smart contracts. You can check out CosmWasmdocs andblog posts for entry-level tutorials.

## The complete example

In the snippets below some details might be omitted. Please check out the complete smart contractexample for a complete implementation.

## 1. Install dependencies and import the libraries

In order to start using the Neutron ICA controller module, you need to install some dependencies. Add the following libraries to your dependencies section:

[ dependencies ] cosmwasm-std

=

"1.2.5"

# Other standard dependencies...

# This is a library that simplifies working with IBC response packets (acknowledgments, timeouts),

# contains bindings for the Neutron ICA adapter module (messages, responses, etc.) and provides

# various helper functions.

neutron-sdk

=

"0.5.0"

# Required to marshal skd.Msg values; the marshalled messages will be attached to the IBC packets

# and executed as a transaction on the host chain.

cosmos-sdk-proto

=

{

version

=

"0.14.0" ,

default-features

=

false

} protobuf

=

{

version

=

"3.2.0" ,

features

=

[ "with-bytes" ]

} Now you can import the libraries:

use

neutron_sdk :: { bindings :: { msg :: { IbcFee ,

MsgSubmitTxResponse ,

NeutronMsg } , query :: { NeutronQuery ,

QueryInterchainAccountAddressResponse } , types :: ProtobufAny , } , interchain_txs :: helpers :: { decode_acknowledgement_response , decode_message_response , get_port_id , } , sudo :: msg :: { RequestPacket ,

SudoMsg } , NeutronResult , } ; use

cosmwasm_std :: Coin ;

## 2. Register an Interchain Account

Neutron allows a smart contract to register multiple interchain account for within a single IBC connection. While you can implement interchain account registration in theinstantiate() entrypoint, having a separate handler is probably a better idea:

// Initialize the storage for known interchain accounts. pub

const

INTERCHAIN_ACCOUNTS :

Map < String ,

Option < ( String ,

String )

= Map :: new ( "interchain_accounts" ) ;

# [derive(Serialize, Deserialize, Clone, Debug, PartialEq,

# JsonSchema)]

# [serde(rename_all =

"snake_case" )] pub

enum

ExecuteMsg

{ Register

{ connection_id :

String , interchain_account_id :

String , } }

# [entry_point]

pub

fn

execute ( deps :

DepsMut , env :

Env , _ :

MessageInfo , msg :

ExecuteMsg , )

->

StdResult < Response < NeutronMsg

{ match msg { ExecuteMsg :: Register

{ connection_id , interchain_account_id , }

=>

execute_register_ica ( deps , env , connection_id , interchain_account_id ) , } }

fn

execute_register_ica ( deps :

DepsMut , env :

Env , connection_id :

String , interchain_account_id :

String , )

->

StdResult < Response < NeutronMsg

{ let register = NeutronMsg :: register_interchain_account ( connection_id , interchain_account_id . clone ( ) ) ;

// Get the IBC port identifier generated by Neutron for the new interchain account. let key =

get_port_id ( env . contract . address . as_str ( ) ,

& interchain_account_id ) ;

// Add an incomplete entry for the new account to the storage. INTERCHAIN_ACCOUNTS . save ( deps . storage , key ,

& None ) ? ; Ok ( Response :: new ( ) . add_message ( register ) ) } In the snippet above, we create theExecuteMsg enum that contains theRegister message, and implement a simpleexecute_register_ica() handler for this message. This handler:

1. Creates a message to the Neutronsinterchaintxs
2. module;
3. Uses a helper functionget_port_id()
4. to get the port identifier that Neutron is going to generate for the channel
5. dedicated to this specific interchain account;
6. Initializes the storage for information related to the new interchain account (currently empty).

Theinterchain_account_id is just a string name for your new account that you can use to distinguish between multiple accounts created within a single IBC connection.

Note: in a real-world scenario you wouldn't want just anyone to be able to make your contract register interchain accounts, so it might make sense to check the handler After executing theexecute_register_ica() handler you need to have a way to know whether the account was registered properly. As with all IBC-related events (acknowledgements, timeouts),OnChanOpenAck messages are dispatched by Neutron to respective contracts viawasm.Sudo() . So, in order to process this type of events, you need to implement thesudo() entrypoint for your contract and process the message dispatched by Neutron:

# [cfg_attr(not(feature =

"library" ), entry_point)] pub

fn

sudo ( deps :

DepsMut , env :

Env , msg :

SudoMsg )

->

StdResult < Response

{ match msg { SudoMsg :: OpenAck

{ port_id , channel_id , counterparty_channel_id , counterparty_version , }

=>

sudo_open_ack ( deps , env , port_id , channel_id , counterparty_channel_id , counterparty_version , ) , _ =>

Ok ( Response :: default ( ) ) , } }

fn

sudo_open_ack ( deps :

DepsMut , _env :

Env , port_id :

String , _channel_id :

String , _counterparty_channel_id :

String , counterparty_version :

String , )

->

StdResult < Response

{ // The version variable contains a JSON value with multiple fields, // including the generated account address. let

parsed_version :

Result < OpenAckVersion , _

= serde_json_wasm :: from_str ( counterparty_version . as_str ( ) ) ;

// Update the storage record associated with the interchain account. if

let

Ok ( parsed_version )

= parsed_version { INTERCHAIN_ACCOUNTS . save ( deps . storage , port_id , & Some ( ( parsed_version . address , parsed_version . controller_connection_id , ) ) , ) ? ; return

Ok ( Response :: default ( ) ) ; }

Err ( StdError :: generic_err ( "Can't parse counterparty_version" ) ) } 1. All possible message types that can come from Neutron are listed in theSudoMsg 2. enum. Here we implement a handler 3. just for one element of this enum,SudoMsg::OpenAck 4. ; 5. If the interchain account was successfully created, you might want to know what account address was generated for you 6. on the host zone. This information is contained in 7. thecounterparty_version 8. variable (see the structure 9. ) 10. , which we need to parse. If we are able to parse it successfully, we save the remote address and the connection 11. identifier to the previously created entry in theINTERCHAIN_ACCOUNTS 12. storage.

Note: it is required that you implement asudo() handler in your contract if you are using the interchain transactions module, even if for some reason you don't want to implement any specific logic for IBC events. Note: you can organise yourINTERCHAIN_ACCOUNTS storage in any way that suits your needs. for example, you can also save theinterchain_account_id value there to have easy access to it from inside your contract. After your contract successfully processed theSudoMsg::OpenAck event sent by Neutron, you can start using the Interchain Account that was created for you.

## 3. Execute an interchain transaction

### Sending the transaction

use

cosmos_sdk_proto :: cosmos :: staking :: v1beta1 :: { MsgDelegate ,

MsgDelegateResponse } ;

// Default timeout for SubmitTX is two weeks const

DEFAULT_TIMEOUT_SECONDS :

u64

=

60

*

60

*

24

*

7

*

2 ;

/// SudoPayload is a type that stores information about a transaction that we try to execute /// on the host chain. This is a type introduced for our convenience.

# [derive(Serialize, Deserialize, Clone, Debug, PartialEq,

# JsonSchema)]

# [serde(rename_all =

"snake_case" )] pub

struct

SudoPayload

{ pub message :

String , pub port_id :

String , }

# [derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]

# [serde(rename_all =

"snake_case" )] pub

enum

ExecuteMsg

{ Register

{ connection_id :

String , interchain_account_id :

String , } , Delegate

{ interchain_account_id :

String , validator :

String , amount :

u128 , denom :

String , timeout :

Option < u64

, } }

# [entry_point]

pub

fn

execute ( deps :

DepsMut , env :

Env , _ :

MessageInfo , msg :

ExecuteMsg , )

->

StdResult < Response < NeutronMsg

{ deps . api . debug ( format! ( "WASMDEBUG: execute: received msg: {:?}" , msg ) . as_str ( ) ) ; match msg { ExecuteMsg :: Register

{ connection_id , interchain_account_id , }

=>

execute_register_ica ( deps , env , connection_id , interchain_account_id ) , ExecuteMsg :: Delegate

{ validator , interchain_account_id , amount , denom , timeout , }

=>

execute_delegate ( deps , env , interchain_account_id , validator , amount , denom , timeout , ) , } }

fn

execute_delegate ( mut deps :

DepsMut , env :

Env , interchain_account_id :

String , validator :

String , amount :

u128 , denom :

String , timeout :

Option < u64

, )

->

StdResult < Response < NeutronMsg

{ // Get the delegator address from the storage & form the Delegate message. let

( delegator , connection_id )

=

get_ica ( deps . as_ref ( ) ,

& env ,

& interchain_account_id ) ? ; let delegate_msg =

MsgDelegate

{ delegator_address : delegator , validator_address : validator , amount :

Some ( Coin

{ denom , amount : amount . to_string ( ) , } ) , } ;

// Serialize the Delegate message. let

mut buf =

Vec :: new ( ) ; buf . reserve ( delegate_msg . encoded_len ( ) ) ;

if

let

Err ( e )

= delegate_msg . encode ( & mut buf )

{ return

Err ( StdError :: generic_err ( format! ( "Encode error: {}" , e ) ) ) ; }

// Put the serialized Delegate message to a types.Any protobuf message. let any_msg =

ProtobufAny

{ type_url :

"/cosmos.staking.v1beta1.MsgDelegate" . to_string ( ) , value :

Binary :: from ( buf ) , } ;

// specify fees to refund relayers for submission of ack and timeout messages // // The contract MUST HAVE recv_fee + ack_fee + timeout_fee coins on its balance! // See more info about fees here: https://docs.neutron.org/neutron/modules/interchain-txs/messages#msgsubmittx // and here: https://docs.neutron.org/neutron/modules/feerefunder/overview let fee =

IbcFee

{ recv_fee :

vec! [ ] ,

// must be empty ack_fee :

vec! [ CosmosCoin :: new ( 100000u128 ,

"untrn" ) ] , timeout_fee :

vec! [ CosmosCoin :: new ( 100000u128 ,

"untrn" ) ] , } ;

// Form the neutron SubmitTx message containing the binary Delegate message. let cosmos_msg =

NeutronMsg :: submit_tx ( connection_id , interchain_account_id . clone ( ) , vec! [ any_msg ] , "" . to_string ( ) , timeout . unwrap_or ( DEFAULT_TIMEOUT_SECONDS ) , fee ) ;

// We use a submessage here because we need the process message reply to save // the outgoing IBC packet identifier for later. let submsg =

msg_with_sudo_callback ( deps . branch ( ) , cosmos_msg , SudoPayload

{ port_id :

get_port_id ( env . contract . address . to_string ( ) ,

& interchain_account_id ) , // Here you can store some information about the transaction to help you parse // the acknowledgement later. message :

"interchain_delegate" . to_string ( ) , } , ) ? ;

Ok ( Response :: default ( ) . add_submessages ( vec! [ submsg ] ) ) }

fn

msg_with_sudo_callback < C :

Into < CosmosMsg < T

,

T

( deps :

DepsMut , msg :

C , payload :

SudoPayload , )

->

StdResult < SubMsg < T

{ save_reply_payload ( deps . storage , payload ) ? ; Ok ( SubMsg :: reply_on_success ( msg ,

SUDO_PAYLOAD_REPLY_ID ) ) }

// Add storage for reply ids pub

const

REPLY_ID_STORAGE :

Item < Vec < u8

=

Item :: new ( "reply_queue_id" ) ;

pub

fn

save_reply_payload ( store :

& mut

dyn

Storage , payload :

SudoPayload )

->

StdResult < ( )

{ REPLY_ID_STORAGE . save ( store ,

& to_vec ( & payload ) ? ) } 1. First we need to import theMsgDelegate 2. type from thecosmos_sdk_proto 3. library. This is required to marshal the 4. message and put it to the IBC packet sent by the ICA module; 5. Then we implement a handler for the newExecuteMsg::Delegate 6. handler,execute_delegate() 7. , and add it to 8. ourexecute() 9. entrypoint; 10. Inside theexecute_delegate() 11. handler, we get the interchain account address from the storage, form aDelegate 12. message, form anIBCFee 13. structure that specifies fees to refund relayers for submission ofack 14. andtimeout 15. messages, put it and the formedDelegate 16. message inside Neutron'sSubmitTx 17. message and execute it as a submessage. Inside 18. themsg_with_sudo_callback() 19. function, we set up the reply payload using theSUDO_PAYLOAD_REPLY_ID 20. value.

We need to execute theSubmitTx message as a submessage because Neutron returns the outgoing IBC packet identifier for us as a message reply. This IBC packet identifier is necessary to later determine which To process it, we need to implement thereply() handler:

# [entry_point]

pub

fn

reply ( deps :

DepsMut , env :

Env , msg :

Reply )

->

StdResult < Response

{ match msg . id { SUDO_PAYLOAD_REPLY_ID

=>

prepare_sudo_payload ( deps , env , msg ) , _ =>

Err ( StdError :: generic_err ( format! ( "unsupported reply message id {}" , msg . id ) ) ) , } }

fn

prepare_sudo_payload ( mut deps :

DepsMut , _env :

Env , msg :

Reply )

->

StdResult < Response

{ let payload =

read_reply_payload ( deps . storage ) ? ; let resp :

MsgSubmitTxResponse

=

serde_json_wasm :: from_slice ( msg . result . into_result ( ) . map_err ( StdError :: generic_err ) ? . data . ok_or_else ( | |

StdError :: generic_err ( "no result" ) ) ? . as_slice ( ) , ) . map_err ( | e |

StdError :: generic_err ( format! ( "failed to parse response: {:?}" , e ) ) ) ? ; deps . api . debug ( format! ( "WASMDEBUG: reply msg: {:?}" , resp ) . as_str ( ) ) ; let seq_id = resp . sequence_id ; let channel_id = resp . channel ; save_sudo_payload ( deps . branch ( ) . storage , channel_id , seq_id , payload ) ? ; Ok ( Response :: new ( ) ) }

pub

fn

read_reply_payload ( store :

& mut

dyn

Storage )

->

StdResult < SudoPayload

{ let data =

REPLY_ID_STORAGE . load ( store ) ? ; from_binary ( & Binary ( data ) ) }

pub

fn

save_sudo_payload ( store :

& mut

dyn

Storage , channel_id :

String , seq_id :

u64 , payload :

SudoPayload , )

->

StdResult < ( )

{ SUDO_PAYLOAD . save ( store ,

( channel_id , seq_id ) ,

& to_vec ( & payload ) ? ) }

### IBC Events

After we saved the IBC packet identifier, we are ready for processing the IBC events that can be triggered by an IBC relayer: an acknowledgement or a timeout. In order to process them, we need to add a couple of new handlers to thesudo() entrypoint:

# [cfg_attr(not(feature =

"library" ), entry_point)] pub

fn

sudo ( deps :

DepsMut , env :

Env , msg :

SudoMsg )

->

StdResult < Response

{ match msg { // For handling successful (non-error) acknowledgements. SudoMsg :: Response

{ request , data }

=>

sudo_response ( deps , request , data ) ,

// For handling error acknowledgements. SudoMsg :: Error

{ request , details }

=>

sudo_error ( deps , request , details ) ,

// For handling error timeouts. SudoMsg :: Timeout

{ request }

=>

sudo_timeout ( deps , env , request ) ,

SudoMsg :: OpenAck

{ port_id , channel_id , counterparty_channel_id , counterparty_version , }

=>

sudo_open_ack ( deps , env , port_id , channel_id , counterparty_channel_id , counterparty_version , ) , _ =>

Ok ( Response :: default ( ) ) , } }

// Interchain transaction responses - here we just save ack/err/timeout state pub

const

ACKNOWLEDGEMENT_RESULTS :

Map < ( String ,

u64 ) ,

AcknowledgementResult

= Map :: new ( "acknowledgement_results" ) ;

/// Serves for storing acknowledgement calls for interchain transactions

# [derive(Serialize, Deserialize, Clone, PartialEq, Eq, JsonSchema, Debug)]

# [serde(rename_all =

"snake_case" )] pub

enum

AcknowledgementResult

{ /// Success - Got success acknowledgement in sudo with array of message item types in it Success ( Vec < String

) , /// Error - Got error acknowledgement in sudo with payload message in it and error details Error ( ( String ,

String ) ) , /// Timeout - Got timeout acknowledgement in sudo with payload message in it Timeout ( String ) , }

**Successful Response**

Let's have a look at how to handle a successful Response event (a non-error IBC Acknowledgement):

fn

sudo_response ( deps :

DepsMut , request :

RequestPacket , data :

Binary )

->

StdResult < Response

{ deps . api . debug ( format! ( "WASMDEBUG: sudo_response: sudo received: {:?} {:?}" , request , data ) . as_str ( ) , ) ;

// Get the channel identifier and the sequence identifier to be able to understand // which transaction is acknowledged by this packet, and for which Interchain Account. // // WARNING: RETURNING THIS ERROR CLOSES THE CHANNEL. // AN ALTERNATIVE IS TO MAINTAIN AN ERRORS QUEUE AND PUT THE FAILED REQUEST THERE // FOR LATER INSPECTION. // In this particular case, we return an error because not having the sequence id // in the request value implies that a fatal error occurred on Neutron side. let seq_id = request . sequence . ok_or_else ( | |

StdError :: generic_err ( "sequence not found" ) ) ? ; // WARNING: RETURNING THIS ERROR CLOSES THE CHANNEL. // AN ALTERNATIVE IS TO MAINTAIN AN ERRORS QUEUE AND PUT THE FAILED REQUEST THERE // FOR LATER INSPECTION. // In this particular case, we return an error because not having the sequence id // in the request value implies that a fatal error occurred on Neutron side. let channel_id = request . source_channel . ok_or_else ( | |

StdError :: generic_err ( "channel_id not found" ) ) ? ;

// Read the information about the transaction that we previously executed and saved to state. // // NOTE: NO ERROR IS RETURNED HERE. THE CHANNEL LIVES ON. // In this particular example, this is a matter of developer's choice. Not being able to read // the payload here means that there was a problem with the contract while submitting an // interchain transaction. You can decide that this is not worth killing the channel, // write an error log and / or save the acknowledgement to an errors queue for later manual // processing. The decision is based purely on your application logic. let payload =

read_sudo_payload ( deps . storage , channel_id , seq_id ) . ok ( ) ; if payload . is_none ( )

{ let error_msg =

"WASMDEBUG: Error: Unable to read sudo payload" ; deps . api . debug ( error_msg ) ; add_error_to_queue ( deps . storage , error_msg . to_string ( ) ) ; return

Ok ( Response :: default ( ) ) ; }

deps . api . debug ( format! ( "WASMDEBUG: sudo_response: sudo payload: {:?}" , payload ) . as_str ( ) ) ;

// Parse the response to Vec. // // WARNING: RETURNING THIS ERROR CLOSES THE CHANNEL. // AN ALTERNATIVE IS TO MAINTAIN AN ERRORS QUEUE AND PUT THE FAILED REQUEST THERE // FOR LATER INSPECTION. // In this particular case, we return an error because not being able to parse this data // that a fatal error occurred on Neutron side, or that the remote chain sent us unexpected data. // Both cases require immediate attention. let parsed_data =

decode_acknowledgement_response ( data ) ? ;

// Iterate over the messages, parse them depending on their type & process them. let

mut item_types =

vec! [ ] ; for item in parsed_data { let item_type = item . msg_type . as_str ( ) ; item_types . push ( item_type . to_string ( ) ) ; match item_type { "/cosmos.staking.v1beta1.MsgDelegate"

=>

{ // WARNING: RETURNING THIS ERROR CLOSES THE CHANNEL. // AN ALTERNATIVE IS TO MAINTAIN AN ERRORS QUEUE AND PUT THE FAILED REQUEST THERE // FOR LATER INSPECTION. // In this particular case, a mismatch between the string message type and the // serialised data layout looks like a fatal error that has to be investigated. let _out :

MsgDelegateResponse

=

decode_message_response ( & item . data ) ? ; } _ =>

{ deps . api . debug ( format! ( "This type of acknowledgement is not implemented: {:?}" , payload ) . as_str ( ) , ) ; } } }

if

let

Some ( payload )

= payload { // update but also check that we don't update same seq_id twice ACKNOWLEDGEMENT_RESULTS . update ( deps . storage , ( payload . port_id , seq_id ) , | maybe_ack |

->

StdResult < AcknowledgementResult

{ match maybe_ack { Some ( _ack )

=>

Err ( StdError :: generic_err ( "trying to update same seq_id" ) ) , None

=>

Ok ( AcknowledgementResult :: Success ( item_types ) ) , } } , ) ? ; }

Ok ( Response :: default ( ) ) }

pub

fn

read_sudo_payload ( store :

& mut

dyn

Storage , channel_id :

String , seq_id :

u64 , )

->

StdResult < SudoPayload

{ let data =

SUDO_PAYLOAD . load ( store ,

( channel_id , seq_id ) ) ? ; from_binary ( & Binary ( data ) ) }

pub

fn

add_error_to_queue ( store :

& mut

dyn

Storage , error_msg :

String )

->

Option < ( )

{ let result =

ERRORS_QUEUE . keys ( store ,

None ,

None ,

Order :: Descending ) . next ( ) . and_then ( | data | data . ok ( ) ) . map ( | c | c +

1 ) . or ( Some ( 0 ) ) ;

result . and_then ( | idx |

ERRORS_QUEUE . save ( store , idx ,

& error_msg ) . ok ( ) ) }

pub

const

ERRORS_QUEUE :

Map < u32 ,

String

=

Map :: new ( "errors_queue" ) ; 1. We get the sequence and channel identifiers to retrieve the information about the interchain transaction from our 2. local storage.Note 3. : we could instead parse the raw data from theRequestPacket 4. , but it feels more natural to 5. save 6. the required information when sending the transaction and to retrieve it from the state when processing the response; 7. We parse the response data and start iterating over the message responses, determining the message type for each of 8. them and (potentially) executing custom logic for each message.

Note : if your Sudo handler fails, the acknowledgment won't be marked as processed inside the IBC module. This will make most IBC relayers try to submit the acknowledgment over and over again. And since the ICA channels areORDERED , ACKs must be processed in the same order as corresponding transactions were sent, meaning no further acknowledgments will be process until the previous one processed successfully.

We strongly recommend developers to write Sudo handlers very carefully and keep them as simple as possible. If you do

want to have elaborate logic in your handler, you should verify the acknowledgement data before making any state changes; that way you can, if the data received with the acknowledgement is incompatible with executing the handler logic normally, return anOk() response immediately, which will prevent the acknowledgement from being resubmitted.

**Error**

fn

sudo_error ( deps :

DepsMut , request :

RequestPacket , details :

String )

->

StdResult < Response

{ deps . api . debug ( format! ( "WASMDEBUG: sudo error: {}" , details ) . as_str ( ) ) ; deps . api . debug ( format! ( "WASMDEBUG: request packet: {:?}" , request ) . as_str ( ) ) ;

// WARNING: RETURNING THIS ERROR CLOSES THE CHANNEL. // AN ALTERNATIVE IS TO MAINTAIN AN ERRORS QUEUE AND PUT THE FAILED REQUEST THERE // FOR LATER INSPECTION. // In this particular case, we return an error because not having the sequence id // in the request value implies that a fatal error occurred on Neutron side. let seq_id = request . sequence . ok_or_else ( | |

StdError :: generic_err ( "sequence not found" ) ) ? ;

// WARNING: RETURNING THIS ERROR CLOSES THE CHANNEL. // AN ALTERNATIVE IS TO MAINTAIN AN ERRORS QUEUE AND PUT THE FAILED REQUEST THERE // FOR LATER INSPECTION. // In this particular case, we return an error because not having the sequence id // in the request value implies that a fatal error occurred on Neutron side. let channel_id = request . source_channel . ok_or_else ( | |

StdError :: generic_err ( "channel_id not found" ) ) ? ; let payload =

read_sudo_payload ( deps . storage , channel_id , seq_id ) . ok ( ) ;

if

let

Some ( payload )

= payload { // update but also check that we don't update same seq_id twice ACKNOWLEDGEMENT_RESULTS . update ( deps . storage , ( payload . port_id , seq_id ) , | maybe_ack |

->

StdResult < AcknowledgementResult

{ match maybe_ack { Some ( _ack )

=>

Err ( StdError :: generic_err ( "trying to update same seq_id" ) ) , None

=>

Ok ( AcknowledgementResult :: Error ( ( payload . message , details ) ) ) , } } , ) ? ; }

else

{ let error_msg =

"WASMDEBUG: Error: Unable to read sudo payload" ; deps . api . debug ( error_msg ) ; add_error_to_queue ( deps . storage , error_msg . to_string ( ) ) ; }

Ok ( Response :: default ( ) ) } This handler is very similar tosudo_response() . Unfortunately, current ICA implementation does not allow you to get the exact error string that was returned by the host chain; your controller code can only know that something went wrong on the other side.

**Timeout**

fn

sudo_timeout ( deps :

DepsMut , _env :

Env , request :

RequestPacket )

->

StdResult < Response

{ deps . api . debug ( format! ( "WASMDEBUG: sudo timeout request: {:?}" , request ) . as_str ( ) ) ;

// WARNING: RETURNING THIS ERROR CLOSES THE CHANNEL. // AN ALTERNATIVE IS TO MAINTAIN AN ERRORS QUEUE AND PUT THE FAILED REQUEST THERE // FOR LATER INSPECTION. // In this particular case, we return an error because not having the sequence id // in the request value implies that a fatal error occurred on Neutron side. let seq_id = request . sequence . ok_or_else ( | |

StdError :: generic_err ( "sequence not found" ) ) ? ;

// WARNING: RETURNING THIS ERROR CLOSES THE CHANNEL. // AN ALTERNATIVE IS TO MAINTAIN AN ERRORS QUEUE AND PUT THE FAILED REQUEST THERE // FOR LATER INSPECTION. // In this particular case, we return an error because not having the sequence id // in the request value implies that a fatal error occurred on Neutron side. let channel_id = request . source_channel . ok_or_else ( | |

StdError :: generic_err ( "channel_id not found" ) ) ? ;

// update but also check that we don't update same seq_id twice // NOTE: NO ERROR IS RETURNED HERE. THE CHANNEL LIVES ON. // In this particular example, this is a matter of developer's choice. Not being able to read // the payload here means that there was a problem with the contract while submitting an // interchain transaction. You can decide that this is not worth killing the channel, // write an error log and / or save the acknowledgement to an errors queue for later manual // processing. The decision is based purely on your application logic. // Please be careful because it may lead to an unexpected state changes because state might // has been changed before this call and will not be reverted because of suppressed error. let payload =

read_sudo_payload ( deps . storage , channel_id , seq_id ) . ok ( ) ; if

let

Some ( payload )

= payload { // update but also check that we don't update same seq_id twice ACKNOWLEDGEMENT_RESULTS . update ( deps . storage , ( payload . port_id , seq_id ) , | maybe_ack |

->

StdResult < AcknowledgementResult

{ match maybe_ack { Some ( _ack )

=>

Err ( StdError :: generic_err ( "trying to update same seq_id" ) ) , None

=>

Ok ( AcknowledgementResult :: Timeout ( payload . message ) ) , } } , ) ? ; }

else

{ let error_msg =

"WASMDEBUG: Error: Unable to read sudo payload" ; deps . api . debug ( error_msg ) ; add_error_to_queue ( deps . storage , error_msg . to_string ( ) ) ; }

Ok ( Response :: default ( ) ) } This handler looks exactly the same as the previous one. TheTimeout event, however, should be treated with extra attention. There is no dedicated event for a closed channel (ICA disables all messages related to closing the channels). Your channel, however, can still be closed if a packet timeout occurs. This means that if you are

notified about a packet timeout,you can be sure that the affected channel was closed .

Note: it is generally a good practice to set the packet timeout for your interchain transactions to a really large value. If the timeout occurs anyway, you can just executeRegisterInterchainAccount message again to recover access to your interchain account. Previous CosmWasm + WasmKit Next CosmWasm + ICQ