Messages

MsgRegisterInterchainAccount

Attempts to register an interchain account by sending an IBC packet over an IBC connection.

message

MsgRegisterInterchainAccount

{ option

(gogoproto . equal)

=

false ; option

(gogoproto . goproto_getters)

=

false ;

string from_address =

1 ; string connection_id =

2

[

(gogoproto . moretags)

=

"yaml:\"connection_id\""

] ; string interchain_account_id =

3

[

(gogoproto . moretags)

=

"yaml:\"interchain_account_id\""

] ; repeated

cosmos . base . v1beta1 . Coin register_fee =

4 ; } * from_address * must be a smart contract address, otherwise the message will fail; * connection_id * must be the identifier of a valid IBC connection, otherwise the message will fail; * interchain_account_id * is used to generate theowner * parameter for ICA'sRegisterInterchainAccount() * call, which is later used for port identifier generation (see below). Maximum allowed length ofinterchain_account_id * is 47 characters. * register_fee * fee is required to be paid in favor of the community (payee address is the treasury) to register an interchain account. Minimal amount of fee is contolled by the module's paramRegisterFee

IBC ports naming / Interchain Account address derivation If a contract with the addressneutron14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9s5c2epq sends anMsgRegisterInterchainAccount withinterchain_account_id set tohub/1 , the generated ICA owner will look likeneutron14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9s5c2epq.hub/1 , and the IBC port generated by the ICA app will be equal toicacontroller-neutron14hj2tavq8fpesdwxxcu44rty3hh90vhujrvcmstl4zr3txmfvw9s5c2epq.hub/1 . Neutron V2 update: Fee Implementation for ICA Registration As ofNeutron V2 , we have introduced a new fee structure for the registration of Interchain Accounts (ICAs). Please be aware of the following updates:

- Minimum Fee (minFee)
- :

- A minimum fee is now required for all new ICA registrations. This fee goes directly to theFeeCollector
- with purpose of preventing spam.
- FeeCollector Beneficiary
- :
- TheFeeCollector
- is the designated recipient of the new registration fees, ensuring the economic sustainability of the network. This is the Neutron DAO at the moment.
- Backwards Compatibility Assurance:
-
    - Contracts and ICAs established with contracts, stored on Neutron beforeV2
-
    - willnot
-
    - incur the new registration fee.
-
    - This update is fully compatible with previousNeutron V1 version
-
    - , guaranteeing a smooth transition and no disruption to existing contracts and services.

ICA's remote address generation concatenates connection identifier and port identifier to use them as the derivation key for the new account:

// GenerateAddress returns an sdk.AccAddress derived using the provided module account address and connection and port identifiers. // The sdk.AccAddress returned is a sub-address of the module account, using the host chain connection ID and controller chain's port ID as the derivation key func

GenerateAddress (moduleAccAddr sdk . AccAddress , connectionID , portID string) sdk . AccAddress { return sdk . AccAddress (sdkaddress . Derive (moduleAccAddr ,

[ ] byte (connectionID + portID))) } Note: your contract needs to implement thesudo() entrypoint on order to successfully process the IBC events associated with this message. You can find an example in theneutron-sdk repository.

Response

message

MsgRegisterInterchainAccountResponse

{ }

IBC Events

type MessageOnChanOpenAck struct

{ OpenAck OpenAckDetails json:"open_ack" }

type OpenAckDetails struct

{ PortID string

json:"port_id" ChannelID string

json:"channel_id" CounterpartyChannelId string

json:"counterparty_channel_id" CounterpartyVersion string

json:"counterparty_version" } The data from anOnChanOpenAck event is passed to the contract using a Sudo() call . You can have a look at an example handler implementation in theneutron-sdk repository.

Note: you can find the interchain account address in the stored in theCounterpartyVersion field as part ofmetadata .

State modifications

None.

MsgSubmitTx

Attempts to execute a transaction on a remote chain.

message

MsgSubmitTx

{ option

(gogoproto . equal)

=

false ; option

(gogoproto . goproto_getters)

=

false ;

string from_address =

1 ; string interchain_account_id =

2 ; string connection_id =

3 ; repeated

google . protobuf . Any msgs =

4 ; string memo =

5 ; uint64 timeout =

6 ;

neutron . feerefunder . Fee fee =

7

[

(gogoproto . nullable)

=

false

] ; } * from_address * must be a smart contract address, otherwise the message will fail; * interchain_account_id * is identical toMsgRegisterInterchainAccount.interchain_account_id * ; * connection_id * must be the identifier of a valid IBC connection, otherwise the message will fail; * msgs * must contain not more than it is defined in the module params; * memo * is the transactionmemo * ; * timeout * is a timeout in seconds after which the packet times out; * fee * is a fee amount to refund relayer forack * andtimeout * messages submission.

Note: your smart-contractmust have fee.ack_fee + fee.timeout_fee + fee.recv_fee coins on its balance, otherwise the message fails. See more info about fee refunding mechanismhere . Note: most networks reject memos longer than 256 bytes. Note: your contract needs to implement thesudo() entrypoint on order to successfully process the IBC events associated with this message. You can find an example in theneutron-sdk repository. Note: to see the currently available messages amount in a single MsgSubmitTx, query the module parameters: neutrond query interchaintxs params

params: msg_submit_tx_max_messages: "16"

Response

message

MsgSubmitTxResponse

{ uint64 sequence_id =

1 ; string channel =

2 ; } * sequence_id * is a channel's sequence_id for outgoing ibc packet. Unique per a channel; * channel * is the src channel name on neutron's side trasaction was submitted from;

IBC Events

// MessageSudoCallback is passed to a contract's sudo() entrypoint when an interchain // transaction failed. type MessageSudoCallback struct

{ Response * ResponseSudoPayload json:"response,omitempty" Error * ErrorSudoPayload json:"error,omitempty" Timeout * TimeoutPayload json:"timeout,omitempty" }

type ResponseSudoPayload struct

{ Request channeltypes . Packet json:"request" Data [ ] byte

json:"data"

// Message data }

type ErrorSudoPayload struct

{ Request channeltypes . Packet json:"request" Details string

json:"details" }

type TimeoutPayload struct

{ Request channeltypes . Packet json:"request" } While trying to execute an interchain transaction, you can receive an IBCTimeout or an IBCAcknowledgement , and the latter can contain either a valid response or an error. These three types of transaction results are passed to the contract as distinct messages using aSudo() call . You can have a look at an example handler implementation in theneutron-sdk repository.

You can more find info, recommendations and examples about how process acknowledgementshere .

State modifications

None. Previous Overview Next Client