ICQ Relayer

Overview

Interchain Queries allow smart contracts to make queries to a remote chain. An ICQ Relayer is a required component for making them possible. It acts as a facilitator between the Neutron chain and a querying chain, gathering queries that are needed to be performed from the Neutron, actually performing them, and eventually making the results available for the Neutron's smart contracts. These three main responsibilities are described in details below.

If you are a smart contracts developer and up to develop your dApp on Neutron, you will most likely need your own ICQ Relayer to manage your Interchain Queries.

Queries gathering

All registered Interchain Queries and their parameters are stored in the eponymous module and available by itsquery interface . The Relayer utilises the module's interface in order to initialise the performing list of queries. This is how the Relayer maintains the list of queries to be executed:

- on initialisation, the ICQ moduleRegisteredQueries
- query is executed with theRELAYER_REGISTRY_ADDRESSES
- parameter used for theOwners
- field;
- during the rest of the run, the Relayer listens to the ICQ module'squery_update
- andquery_removed
- events
- and modifies the queries list and parameters correspondingly.

The Relayer also listens to the Neutron'sNewBlockHeader events that are used as a trigger for queries execution. Since each query has its ownupdate_period , the Relayer tracks queries execution height and executes only the queries which update time has come.

Queries execution

When the update time comes for a query, the Relayer runs the specified query on the remote chain:

- in case of a KV-query, the Relayer justreads
- necessary KV-keys from the remote chain's storage withMerkle Proofs
- . Neutron will need these proofs toverify
- validity of KV-results on results submission;
- in case of a TX-query, the Relayer makes a query to the target chain'sTendermint RPC
- to search transactions by message types, events and attributes which were emitted during transactions execution and wereindexed
- by Tendermint. More about Tx query parameters syntaxin the dedicated section
- . When Relayer submits transactions search results to Neutron chain, itDOES NOT
- include events into result (even if events were used for the query), becauseevents are not deterministic
- , therefore they can break blockchain consensus. One more important thing about TX queries is that the Relayer is made the way it only searches for and submits transactions within the trusting period of the Tendermint Light Client. Trusting period is usually calculated as2/3 * unbonding_period
- . Read more about Tendermint Light Client and trusted periodsat this post
- .

Results submission

Relayer submits a query result as the following depending on the Relayer's configuration:

- simply sending it to the Neutron's Interchain Queries module which handles it by storing the result in the blockchain state (KV queries withRELAYER_ALLOW_KV_CALLBACKS
- =false);
- sending it to the Neutron's Interchain Queries module which handles it by storing the result in the blockchain state and passing the result to the owner smart contract (KV queries withRELAYER_ALLOW_KV_CALLBACKS
- =true);
- passing it to the smart contract that has registered the query (TX queries).

This means that it's the Relayer who pays gas for these actions. Note that KV queries submission are straightforward and therefore cheap whereas TX ones and KV callbacks also include smart contract call and their cost may vary significantly.

A bit of technical details about queries

Queries submission

The KV queries are submitted in a fire-and-forget way, i.e. they are submitted once perupdate_period span and never retried forcibly (e.g. on a submission error). The TX queries are a bit more tricky: since they are not stored in the Neutron chain and simply passed to smart contracts, it's needed that each tx is passed and handled by the smart contract only once.

The Relayer uses theBroadcastTxSync messages broadcast type to maintain balance between performance and submission control, but this means that the submission result is not waited for. And here comes an important part related to TX queries. To achieve both submission speed and sequential submission handling, the Relayer fires TX submission messages, remembers the query result as sent, and then in the background retrieves the submission result for the query. If it turns to be a success, the TX is saved as fully processed and will not be sent to the smart contract again. Otherwise, this tx will be marked as failed and will not be sent to the smart contract again during this run. Instead, to prevent repeated submission of transactions which can't be successfully handled by the smart contract, the retry will only be possible on Relayer restart.

As a default when the Relayer submits a TX query result to the Neutron chain and an error occurs in the smart contract during the sudo call, the Relayer will ignore this error and not retry the submission. For all other errors, the Relayer will exit with an error.

This behaviour cased by the fact that the Relayer is not aware of the smart contract's logic and therefore can't know whether the error is recoverable or not. Also, the Relayer should treat all other errors (network/balance/wallet) as fatal, exit and let itself be restarted by the admin/system.

It is strongly recommended to run the Relayer as a daemon to allow easy restart.

If you want to change the behaviour, you can do so by changing the environment variableRELAYER_IGNORE_ERRORS_REGEX .

Beacons in TX queries

Transactions for a TX query are retrieved from a target chain in ascending order. Since the TX query results aren't submitted to the Neutron chain storage (they are processed by smart contracts via Sudo calls right away) there is no way to get the last processed height from the Neutron for a TX query. In order to keep a TX query progress in terms of already processed heights (make further queries, or restart the Relayer and start from the point where the Relayer stopped during the previous run) the Relayer saves progress for each TX query in its own storage. One of the things it stores is the remote chain height, and it gets updated when all transactions from the given height have been submitted to the chain (i.e. submission messages with these transactions have been broadcast). When the next time to execute the query comes, or when the Relayer restarts, this height will be used to retrieve the next batch of transactions. TheRELAYER_INITIAL_TX_SEARCH_OFFSET config parameter is tightly coupled with this part of documentation. Read more about it in theRelayer application configuration section .

Configuration

This section contains description for all the possible config values that the Relayer supports. For example values see the.env.example file in the Relayer's repository.

Neutron chain node settings

- RELAYER_NEUTRON_CHAIN_RPC_ADDR
- — RPC address of a Neutron node to interact with (e.g. get events and to submit results);
- RELAYER_NEUTRON_CHAIN_REST_ADDR
- — REST address of a Neutron node to interact with (e.g. get registered queries list);
- RELAYER_NEUTRON_CHAIN_HOME_DIR
- — path to keys directory;
- RELAYER_NEUTRON_CHAIN_SIGN_KEY_NAME
- — name of the key pair to be used by the Relayer;
- RELAYER_NEUTRON_CHAIN_TIMEOUT
- — timeout for Neutron RPC and REST calls;
- RELAYER_NEUTRON_CHAIN_GAS_PRICES
- — the price for a unit of gas used by the Relayer;
- RELAYER_NEUTRON_CHAIN_GAS_LIMIT
- — the maximum price to be paid for a single submission;
- RELAYER_NEUTRON_CHAIN_GAS_ADJUSTMENT
- — gas multiplier used in order to avoid underestimating;
- RELAYER_NEUTRON_CHAIN_CONNECTION_ID
- — Neutron chain connection ID; Relayer will only relay events for this connection;
- RELAYER_NEUTRON_CHAIN_DEBUG
- — flag to run neutron chain provider in debug mode;

- RELAYER_NEUTRON_CHAIN_KEYRING_BACKEND
- — describedhere
- ;
- RELAYER_NEUTRON_CHAIN_OUTPUT_FORMAT
- — Neutron chain provider output format;
- RELAYER_NEUTRON_CHAIN_SIGN_MODE_STR
- — describedhere
- , also consider use short variation, e.g.direct
- .

Target chain node settings

- RELAYER_TARGET_CHAIN_RPC_ADDR
- — RPC address of a target chain node to interact with (e.g. send queries);
- RELAYER_TARGET_CHAIN_ACCOUNT_PREFIX
- — target chain account prefix;
- RELAYER_TARGET_CHAIN_VALIDATOR_ACCOUNT_PREFIX
- — target chain validator account prefix;
- RELAYER_TARGET_CHAIN_TIMEOUT
- — timeout for target chain RPC calls;
- RELAYER_TARGET_CHAIN_DEBUG
- — flag to run neutron chain provider in debug mode;
- RELAYER_TARGET_CHAIN_OUTPUT_FORMAT
- — target chain provider output format.

Relayer application settings

- RELAYER_REGISTRY_ADDRESSES
- — a list of comma-separated smart-contract addresses (registered query owners) for which the Relayer processes interchain queries. If empty, literally all registered queries are processed which is usable if you are up to deploy a public Relayer;
- RELAYER_ALLOW_TX_QUERIES
- — if true, Relayer will process tx queries (iffalse
- , Relayer will ignore them). A true value here is mostly usable for a private Relayer because TX queries submission is quite expensive;
- RELAYER_ALLOW_KV_CALLBACKS
- — iftrue
- , will pass proofs as sudo callbacks to contracts. A true value here is mostly usable for a private Relayer because KV query callbacks execution is quite expensive. If false, results will simply be submitted to Neutron and become available for smart contracts retrieval;
- RELAYER_MIN_KV_UPDATE_PERIOD
- — minimal period of queries execution and submission. This value is usable for a public Relayer as a rate limiter because it roughly overrides the queriesupdate_period
- and force queries execution not more often thanN
- blocks;
- RELAYER_STORAGE_PATH
- — path to leveldb storage, will be created on the given path if it doesn't exist. It is required ifRELAYER_ALLOW_TX_QUERIES
- istrue
- ;
- RELAYER_QUERIES_TASK_QUEUE_CAPACITY
- — capacity of the channel that is used to send messages from subscriber to Relayer. Better set to a higher value to avoid problems with Tendermint websocket subscriptions;
- RELAYER_CHECK_SUBMITTED_TX_STATUS_DELAY
- — delay in seconds between TX query submission and the result handling checking (more about this in theTX submission section
-);
- RELAYER_INITIAL_TX_SEARCH_OFFSET
-
    - Only for transaction queries. If set to non zero and no prior search height exists, it will initially set search height to (last_height - X). One example of usage of it will be if you have lots of old tx's on first start you don't need. Keep in mind that it will affect each newly created transaction query. To get a better understanding about how this works read thededicated section
- ;
- RELAYER_WEBSERVER_PORT
- — listener address for webserver json api you can query and prometheus metrics;
- RELAYER_IGNORE_ERRORS_REGEX
-

- regular expression to match errors that should be ignored. If the error matches the regex, the Relayer will ignore it and will not retry the submission. For any other errors, the Relayer will exit with an error.

Logger configuration

As it is said in the Relayer's readme , the Relayer uses a little bit modified version of Uber's zap.Logger . This modification allows logger configuration via env parameters. See the logger configuration guide readme for more information.

Prerequisites

Before running the Relayer application for production purposes, you need to create a wallet for the Relayer, top it up, and set up the configuration (refer to the Configuration section). Also you will most likely need to deploy your own RPC nodes of Neutron and the chain of interest.

- How to deploy your own Neutron RPC node
- ;
- How to prepare target chain RPC node for Relayer's usage
- .

Setting up Relayer wallet

1. The keyring folder for Relayer's usage is configured by the RELAYER_NEUTRON_CHAIN_HOME_DIR
2. variable. The easiest way is to run neutrond keys
3. from the cloned neutron repository
4. and get the default value from the --keyring-dir
5. flag:

neutrond keys Keyring management commands. These keys may be in any format supported by the Tendermint crypto library and can be used by light-clients, full nodes, or any other application that needs to sign with a private key. ... Flags: --keyring-dir string The client Keyring directory; if omitted, the default 'home' directory will be used ... Global Flags: --home string directory for config and data (default "/Users/your-user/.neutrond") 1. Then execute neutrond keys add relayer --keyring-backend test 2. to create an account in the default keyring directory; 3. User relayer 4. as the RELAYER_NEUTRON_CHAIN_SIGN_KEY_NAME 5. , test 6. as the RELAYER_NEUTRON_CHAIN_KEYRING_BACKEND 7. , and pass the keyring directory as a volume to the Relayer's docker container using the keyring path in the container as the RELAYER_NEUTRON_CHAIN_HOME_DIR 8. ; 9. Get the Relayer's wallet address and top its balance up. If you're running the Relayer on the testnet, use the official Neutron faucet. For the mainnet, get some NTRN for the address.

Running the Relayer

1. Make sure you've finished the Configuration
2. part;
3. Build Relayer's docker image from the Relayer's folder:

make build-docker 1. Run Relayer in a docker container way:

docker run --env-file .env.example -p 9999:9999 neutron-org/neutron-query-relayer Notes:

- -p 9999:9999
- exposes the port that allows access to the webserver json api and Relayer's metrics powered using Prometheus. The container's port will be the same as the RELAYER_LISTEN_ADDR
- value that is 9999
- by default. Use another value if you are up to use a different port;
- add keyring passing to the volumes list. For example, assign RELAYER_NEUTRON_CHAIN_HOME_DIR=/keyring
- and run the app as:

docker run --env-file .env.example -v /Users/your-user/.neutrond:/keyring -p 9999:9999 neutron-org/neutron-query-relayer

Webserver API

Relayer serves it's own JSON API and provides commands for querying info about it.

It listens on port that is set in RELAYER_LISTEN_ADDR env.

Commands:

- Print available queries:

go run ./cmd/neutron_query_relayer query

- Resubmit failed transactions:

go run ./cmd/neutron_query_relayer exec resubmit-tx

Shutting the Relayer down

During the execution the Neutron ICQ Relayer receives events from Neutron, reads remote chain's state, and modifies its own state and the Neutron' one. In order to reach a reliable and consistent flow the Relayer is designed the way it finishes initialised interactions with its local storage on receivedSIGINT andSIGTERM . It usually takes a fraction of a second.
Previous IBC Relayer Next Prepare target chain RPC node for Relayer's usage