Crypto.com Chain

Welcome to Crypto.com Chain's documentation!

Get Started →

The high-level design vision is documented in

1. whitepaper and
2. technical whitepaper.

For a bit clearer technical vision, see design philosophy.

The documentation in this repository site is meant to provide specifications and implementation details that will be useful to third party developers or contributors to the main repository.

Getting Started

> **Caution**
>
> This page is for the development environment set up only, and it is subject to changes.
>
> For anyone interested in joining the Crypto.com chain testnet, please refer to our **testnet documentation**.

By following this tutorial, you can compile and run the latest version of Crypto.com Chain from scratch. With supported hardware, you can run the chain locally within a cup of coffee ☕ . However, this document aims to provide you with a step-by-step guide to run Crypto.com Chain locally and not a guide for production usage.

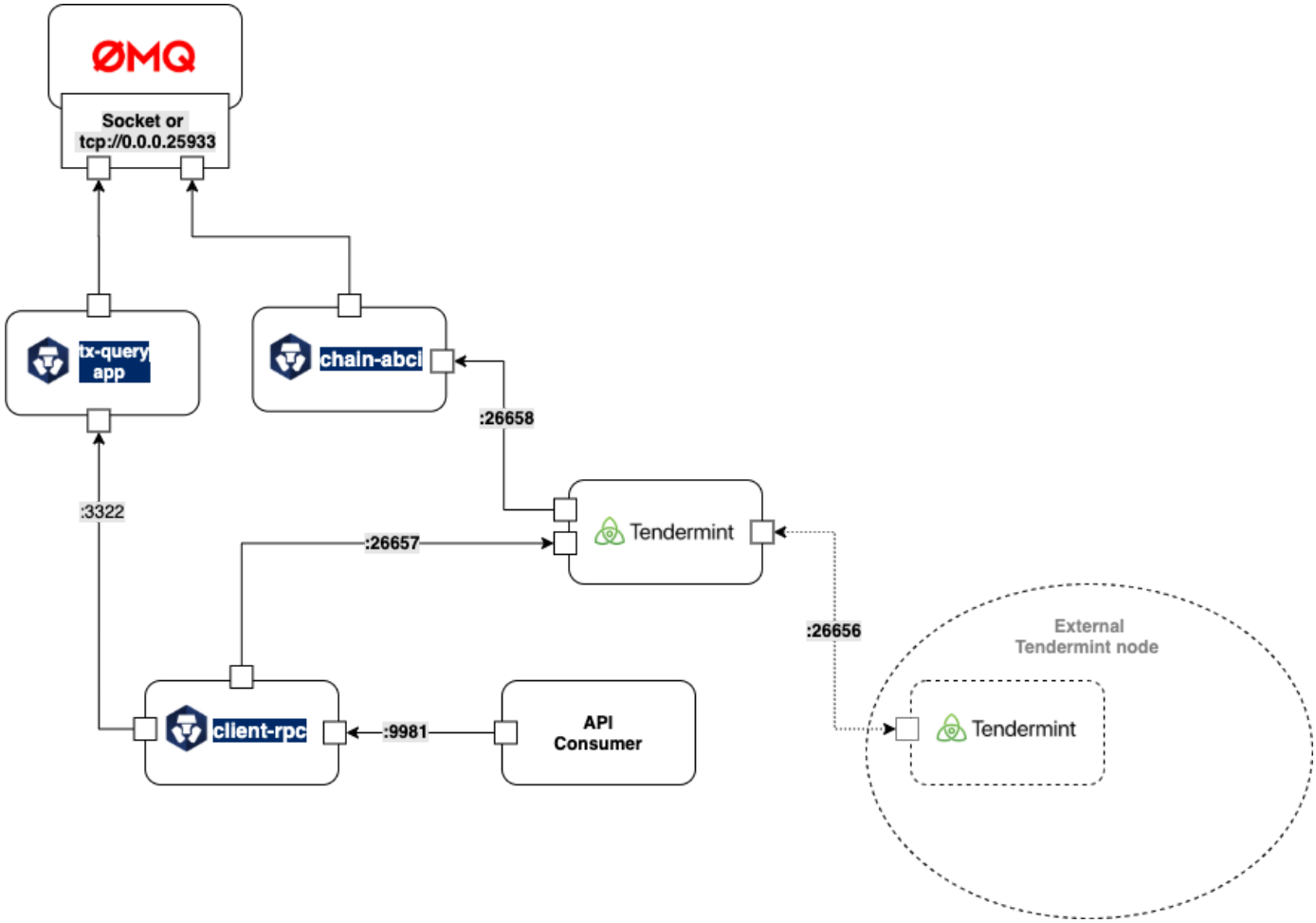
Pre-requisites

Because we utilize the technology of `Intel® Software Guard Extensions (Intel® SGX)` for payment data confidentiality, the pre-requisites are a little more strict than the other chains' setup. A special type of hardware is needed and the reference of SGX-hardware could help you identify if your current hardware supports `Intel® SGX` or not.

If your development machine does not support SGX, we recommend spinning up a cloud instance listed in the above reference. In this guide, we will walk through the process of setting it up on Azure Confidential Compute VM.

A bird's-eye view

Before diving into details, we would like to introduce you the big picture of Crypto.com Chain's main components as following:

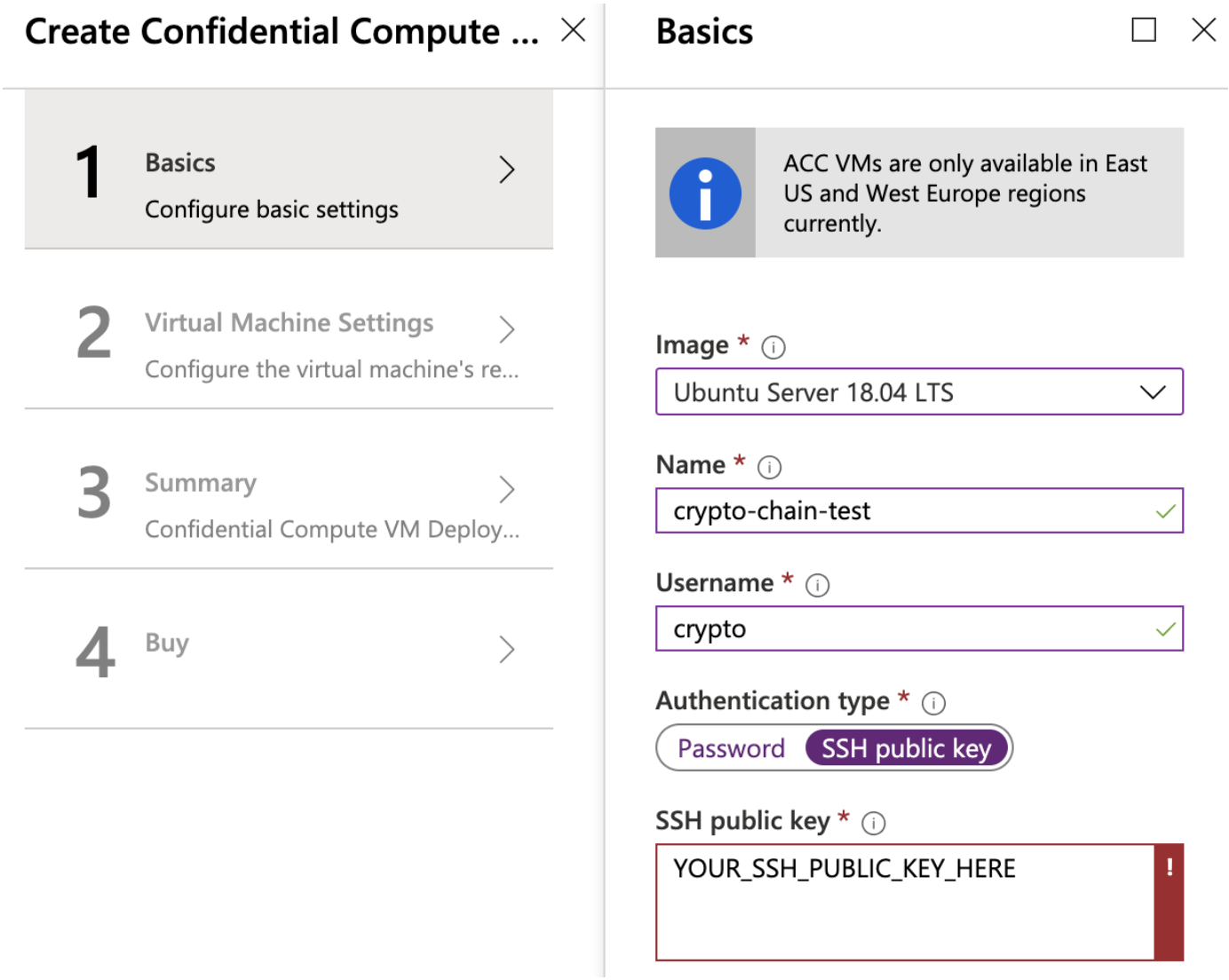


At the end of this getting-start document, you will be running four components:

- `chain-abci` as main chain process.
- `client-rpc` as rpc server for client's interactions.
- `tendermint` for consensus.
- `tx-query-app` allows semi-trusted client querying of sealed tx payloads.

Azure VM creation

Get into the portal of Azure computing and create a new Azure Confidential Compute VM as following config. Noted that `Ubuntu Server 18.04 LTS` is recommended.



Make sure include the Open Enclave SDK:

Then choose your desirable VM `location`, `size`, `storage` and `network configs`, or you can leave them as default.

VM environment setup

SSH to the Azure VM, and start the environment setup for Crypto.com Chain.

- Install `Docker`: you can refer to following document How To Install and Use Docker on Ubuntu 18.04

Make sure you have complete the part of `Executing the Docker Command Without Sudo` by:

```
sudo usermod -aG docker ${USER}
```

- Clone the main chain repo

```
git clone https://github.com/crypto-com/chain.git
```

Build binary and Docker images

1. Build the Crypto.com Chain binary files:

```
$ cd chain/
$ make build
```

It will take you several minutes, and check the binary files share object files in following directory:

```
$ ls target/debug/

chain-abci client-rpc client-cli tx-query-app ...
```

2. Build the docker image with local binary files using following command:

```
$ make image
```

Check the current built image with:

```
$ docker images

REPOSITORY              TAG                 IMAGE ID            CREATED
crypto-chain            develop             817f6c7c7a76        - seconds ago
```

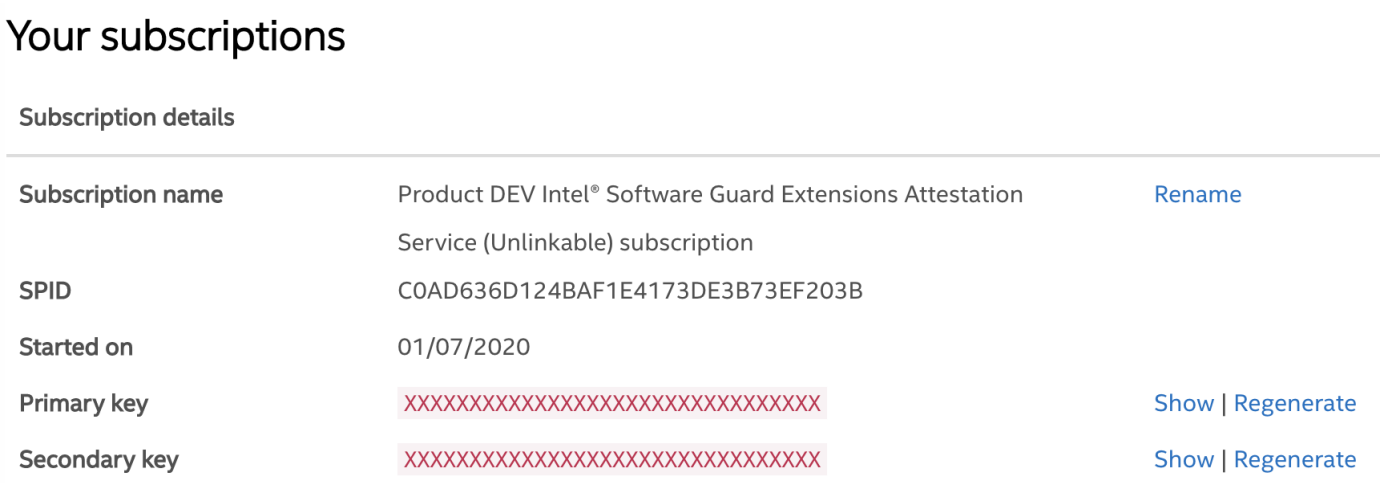
Prepare SPID & KEY

Before kicking off all the components, there is one more step to go, which is registering your own accessing ID and KEY for Intel SGX attestation service.

> **Enhanced Privacy ID (EPID)**
>
> The Intel SGX attestation service is a public web service operated by Intel for client-based privacy focused usages on PCs or workstations. The primary responsibility of the Intel SGX attestation service is to verify attestation evidence submitted by relying parties. The Intel SGX attestation service utilizes Enhanced Privacy ID (EPID) provisioning, in which an Intel processor is given a unique signing key belonging to an EPID group. During attestation, the quote containing the processor's provisioned EPID signature is validated, establishing that it was signed by a member of a valid EPID group. A commercial use license is required for any SGX application running in production mode accessing the Intel SGX attestation service.

In short, you should go to Intel Portal to sign up for the ID and KEY. It won't take you more than 5 minutes.

Make sure what your subscription is DEV Intel® Software Guard Extensions Attestation Service (Unlinkable). Your `SPID` and `Primary key` will be shown on the portal as below:



After you obtained your `SPID` and `Primary key` from Intel, you should embed them to your `.profile` file as environment variables with the other two variables (`SGX_MODE` , `NETWORK_HEX_ID`) we mentioned above. So, make sure append the following lines in your `.profile` file.

```
export SPID={YOUR_SPID}
export IAS_API_KEY={YOUR_PRIMARY_KEY}
```

Surely, remember to source the new `.profile` file:

```
$ source ~/.profile
```

Prepare environment to run the chain

Prepare initial chain data and try to install Intel SGX if the SGX device is not ready.

```
$ make prepare
```

Run chain components

Run all the components of Cryto.com Chain with following command:

```
$ make run
```

Then you can check if all containers are running normally:

```
$ docker ps

CONTAINER ID     IMAGE                              COMMAND                  CREATED
fc51af59593b     crypto-chain:develop               "client-rpc --port=2…"   -
bc586070744b     crypto-chain:develop               "chain-abci --chain_…"   -
ade1db657cd8     tendermint/tendermint:v0.32.8      "/usr/bin/tendermint…"   -
800f173dccc7     crypto-chain:develop               "bash ./run_tx_query…"   -
1c5c71c4047b     crypto-chain:develop               "bash ./run_tx_valid…"   -
```

Besides, you can check the chain-abci and Tendermint status by following commands:

```
$ docker logs -f chain-abci
[-T08:50:02Z INFO  chain_abci::app] received beginblock request
[-T08:50:02Z INFO  chain_abci::app] received endblock request
[-T08:50:02Z INFO  chain_abci::app] received commit request
[-T08:50:03Z INFO  chain_abci::app] received beginblock request
[-T08:50:03Z INFO  chain_abci::app] received endblock request
[-T08:50:03Z INFO  chain_abci::app] received commit request
...
```

```
$ curl 'http://localhost:26657/health'
{
  "jsonrpc": "2.0",
  "id": "",
  "result": {}
}
```

Congratulations! Crypto.com Chain is now running on your machine!

Re-initialize a Crypto.com Chain

If you need to stop the processes and initialize a new chain, firstly you should stop all service and remove docker containers by:

```
$ make stop-all
$ make rm-all
```

And then clean all the storage used by Tendermint, Cryto.com Chain by:

```
$ make clean-data
```

Finally you can initialize a new chain by:

```
$ make prepare
$ make run
```

If no error Check all containers

Congratulations

Congratulations, now the environment to run Crypto.com Chain is all set. Let's on on and start sending your first transaction.

Send Your First Transaction →

Chain ID and Network ID

Chain ID

Crypto.com Chain has different Chain ID to distinguish between *devnet*, *testnet* and *mainnet*. When running the Crypto.com Chain in your local environment, you will also need to decide your own Chain ID.

Different chain has different address prefixes for its corresponding network types, these prefixes are:

Mainnet	Testnet	Regtest
`cro`	`tcro`	`dcro`

For example, our testnet Chain ID is `testnet-thaler-crypto-com-chain-42`.

Network ID

Network ID is the last two hex characters of the Chain ID. Using our testnet Chain ID `testnet-thaler-crypto-com-chain-42` as an example, the network ID would be `42`.

← Genesis

Transaction Accounting Model →

Consensus

Crypto.com Chain prototype uses Tendermint Core as its consensus algorithm. It utilizes the ABCI (Application BlockChain Interface), which allows applications written in different languages to interact with the blockchain.

This interface allows a "plugging" custom applications with Tendermint. Specifically, if the application is written in Go, it can be linked directly with the chain; otherwise, a connection can be established via 3 TCP or Unix sockets. The details of this interface can be found here.

As Crypto.com Chain Core code is written in Rust, we utilize (and aim to continually improve) the rust-abci library.

For the overall architecture of Tendermint Core consensus engine with the ABCI, please refer to this infographic and this guide.

Client: Interacting with the blockchain

To query a blockchain or submit a transaction, one can use the Tendermint RPC for that. Details of this RPC and its mechanism can be found in here.

Currently, it supports 3 methods:

1. URI over HTTP
2. JSON-RPC over HTTP
3. JSON-RPC over WebSockets

The RPC HTTP server is executed on every full node. The RPC methods are equivalent, but WebSockets allow realtime subscription to different events.

> **NOTE**
>
> Tendermint RPC is for internal use only, as it doesn't support rate-limiting, authentication etc., so it shouldn't be directly exposed to the internet.

At this moment, it's also recommended to use `tendermint lite` as a local proxy for the Chain client libraries when connecting to remote full nodes.

The Chain client interaction with Tendermint is currently done via a custom `client-rpc` crate.

The main RPC methods are used `broadcast_tx_(a)sync` and `abci_query` .

broadcast_tx_(a)sync

This method takes "tx" parameter which is application-specific binary data (see transaction serialization for details on Chain binary format). The transaction binary payload is either hex-encoded (when called with the URI method) or base64-encoded (when called with JSON-RPC).

abci_query

Currently the main usage is that given a path "account", one can query the current "staked state" of some address (which is provided as the "data" field).

Application hash

The **"application hash"** is a compact representation of the overall ABCI application state. In Tendermint, ABCI applications are expected to be deterministic. Therefore, given the same input (block/consensus events + transaction data), one can expect that the applications will update its state in the same way. Eventually, we can compare the hash value of its application state with the others and verify the consistency.

In Crypto.com Chain, the application hash is a Blake3 hash of several components:

- Root of a Merkle tree of a valid transaction in a given block;
- Root of a sparse Merkle trie of staked states (see accounting details);
- Binary serialized state of rewards pool (see serialization for details on Chain binary format and genesis for details on "state");
- Serialised network parameters.

Conventions

As genesis information is taken from the Ethereum network, the same address format is used (i.e. hexadecimal encoding of 20-bytes from a keccak-256 hash of a secp256k1 public key).

For Tendermint data, its conventions must be followed:

- *Validator Key pair*: base64-encoded Ed25519 key-pair;
- *Addresses*: the first 20 bytes of SHA256 of the raw public key bytes.

For Crypto.com Chain, it has the following conventions:

- *Chain-ID*: this is a string in Tendermint's genesis.json. In Crypto.com Chain, it should end with two hex digits;
- Network-ID: a single byte determined by the two last hex digits of Chain-ID. It is included in metadata of every transaction to specify the network;
- Transactions, addresses etc.: Please refer to transaction binary serialization, accounting model, addresses / witness and format / types.

← Local Full Node Development

Genesis →

Client Flow

Block-level filtering

The full node "tags" each block with a probabilistic filter (tagging is done using the Events in Tendermint). This filter is then used by the light client code to determine which blocks are of any interest, requests these blocks and processes them. This current mechanism is similar to BIP 157.

Currently, the used filter is the Bloom filter used in Ethereum defined with the following parameters: m = 2048 (bits; keccak-256 hash function), k = 3 (yellowpaper page 5)

The full node inserts the following data into the filter at the moment:

1. "view keys" (secp256k1 public keys allowed to view the content) in the case of transactions producing UTXOs (see transaction types and transaction privacy mechanism for more details).
2. Staked state addresses in the case of transactions manipulating accounts (see accounting details).

Client knows the transaction data

Each block header includes an application hash ("APP_HASH", see consensus for details). As a part of it is a root of a Merkle tree of valid transactions, a client can check whether its known transaction was included in a block:

1. Get the block's application hash / header information.
2. Compute the transaction ID from transaction data (see transaction).
3. Check a Merkle inclusion proof where the transaction ID is one of the leaves, and a part of the application hash is the root.

Client doesn't know the transaction data

If the client doesn't know transaction data, it can collect valid transaction identifiers from blocks that matched its data using the block-level filter. If the transaction data was transparent (staked state operations), the client can decode transaction directly by requesting the full block data. If the transaction data was obfuscated (payments), the client needs to contact an enclave and prove they

can access transaction data using view key signatures (see transaction privacy and enclave architecture for more details).

Clients are responsible for their own confidential data treatment – the use of view keys (whether reused or not), requesting transaction data (e.g. from multiple enclaves when not the client isn't running their own full node) etc.

Payment transaction submission

When the client wishes to submit a payment transaction (UTXO-based), they will construct a plain signed transaction and submit it to one of the full node's enclaves over a secure channel (see enclave architecture).

← Signature Schemes

Enclave Architecture →

Transaction Accounting Model

Crypto.com Chain uses both Unspent Transaction Outputs (UTXOs) and (a form of) accounts for its accounting model, inspired by the work on chimeric ledgers.

Motivation

The native token used in Crypto.com Chain serves two main purposes:

1. Payments
2. Network operation (staking etc.)

These two realms have different rules and properties. These differences are highlighted in the table below:

	Transaction Volume	Visibility	State Changes	Value Transfer Rules
Payments	High	Minimal / confidentiality / data protection is desired	UTXO set is the only "state": changes only by transactions	Flexible: new address per invoice, ad-hoc n-of-m address formations (e.g. for escrows); encoding extra information for atomic swaps etc.
Network operation (staking etc.)	Low	Maximal transparency is desired	Both by transactions and network events (e.g. a validator not following the protocol)	Self-contained: same account changes (reward payouts, unbonding...)

UTXO+Accounts model

For this reason, we chose the mixed model where:

- UTXO is used for payments / value transfers
- Account-model is used for network operations ("staked state")

Different types of transactions and how they relate to these accounting are described in transactions.

Staked state

The current account usage is self-contained limited. Each account ("staked state") contains two balances:

- bonded amount
- unbonded amount

and its slashing related information.

For example, by using client-cli, one can check the staking stake of a *Staking* type address and obtain the following:

```sh
##### EXAMPLE: Staking state #####
+-----------------+----------------------------+
| Nonce           |                          2 |
+-----------------+----------------------------+
| Bonded          |          2000000.00000000 |
+-----------------+----------------------------+
| Unbonded        |             3000.00000000 |
+-----------------+----------------------------+
| Unbonded From   | 2020-02-02 08:28:16 +08:00 |
+-----------------+----------------------------+
| Jailed Until    |                Not jailed |
+-----------------+----------------------------+
| Punishment Type |              Not punished |
+-----------------+----------------------------+
| Slash Amount    |              Not punished |
+-----------------+----------------------------+
```

- The `Nonce` is the number of transactions that have the witness of the staking address.
- The `Bonded` amount is the amount used to check against minimal staking requirements and used to calculate the Tendermint validator voting power (in case of council nodes).

  As it may take time for the network evidence of malicious activity (e.g. double signing) to appear, the stake cannot be withdrawn immediately and is first moved to the "unbonded" balance.

- The `Unbonded` balance can be withdrawn (into transaction outputs) after `Unbonded From` time if the account was not jailed / slashed (see staking).
- For slashing related information:
  - `Jailed Until` is the time until which current account is jailed;
  - `Punishment Type` represents the type of punishment that will be imposed on the account (if any);
  - `Slash Amount` is the amount of penalty.

Staked State Storage

The account state is currently stored in a sparse Merkle trie structure (currently MerkleBIT backed by RocksDB, but it may be migrated to a more robust / better understood structure, e.g. a Patricia Merkle trie or IAVL+).

← Chain ID and Network ID

Transaction →

Enclave Architecture

The primary initial use of Trusted Execution Environments (TEE) is for enforcing payment data confidentiality (see transaction privacy), while maintaining flexibility and auditability. Other use cases may be developed in the long term.

Technology

The initial version is based on Intel SGX, but in the long-term, it would be desirable to support other TEE technology stacks, such as Arm TrustZone or RISC-V Keystone.

Enclave Infrastructure Overview

There are *three* enclaves planned in Crypto.com Chain's transaction data confidentiality implementation:

1. *Transaction validation enclave* (**TVE**) responsible for
    - validating transactions;
    - persisting previously valid transactions (sealed to a local machine); and
    - holding the current key (used for obfuscating or de-obfuscating transaction data).
2. *Transaction query enclave* (**TQE**) responsible for
    - serving encryption and decryption requests from wallets / clients.
3. *Transaction data bootstrapping enclave* (**TDBE**) responsible for
    - fetching current UTXO set transaction data; and
    - handling periodic key generation operations.

For a detailed description of how these enclaves work together, please refer to the implementation plan of transaction data confidentiality.

Transaction validation

The validation of transactions that involve payment obfuscated transaction outputs (see transaction types and accounting model) need to happen inside enclaves. Detailed transaction processing can found here

For the ease of development, the transaction validation happens in a separate process. The Chain ABCI application process then communicates with this process using a simple request-reply protocol over a 0MQ socket:

```
+-----------+ REQ +--------+
|Chain ABCI +-----+ TX val.|
|           | REP | enc.   |
+-----------+     +--------+
```

In production deployment, both of these processes should be on the same machine and hence use IPC as the underlying transport for the 0MQ messages. In development, other transport mechanisms (e.g. TCP) can be used and processes could be in different locations, for example:

- Chain ABCI is executed on the developer laptop (any operating system), and the transaction validation enclave runs inside a Docker container (using the software-simulation mode).
- Chain ABCI is executed on the developer laptop (any operating system), and the transaction validation enclave runs on a remote Linux machine (using the hardware mode).

For further details, please refer to the transaction flow chat between Tendermint, ABCI and transaction validation enclave.

Data sealing

As previous transaction data is needed for transaction validation, it needs to be persisted locally. The enclave uses the process of "data sealing" for this purpose. To make the data accessible for future upgrades and other enclaves, it should be sealed with MRSIGNER-derived keys. For further details, please refer to TEE primitives.

Transaction data bootstrapping

As old payment data becomes inaccessible due to the periodic key rotation (see transaction privacy), newly joined nodes (or nodes that went offline for some time) would need a way to bootstrap the old transaction data by connecting to enclaves of remote nodes and requesting transaction data that the other nodes have locally sealed.

For further details, please refer to this implementation plan of **transaction data bootstrapping enclave**.

Lite client inside enclaves

Each enclave should internally run a lite client that would keep track of the validator set, so that it can safely store the latest "app hash" (see consensus).

Mutual attestation

The core of the bootstrapping process lies in establishing a secure channel between two enclaves – e.g. TLS, see Integrating Remote Attestation with Transport Layer Security. During the connection establishment, both parties present attestation reports that they cross-verify. This will initially utilize the Intel Attestation Service (IAS) where each full node is expected to run an IAS proxy that contains the required credentials (Key and SPID) that can be obtained on Intel portal.

In the future, the support for Data Center Attestation Primitives (DCAP) will be developed, so that each full node operator can run its own attestation service (rather than relying on IAS).

Beyond the mutual attestation, enclaves should perform additional checks against the stored app hash, e.g. if the staked state-associated with the node's enclave is valid (or check the whitelist entry in the case of higher tier nodes).

Transaction querying

As mentioned in client flows, clients may not know their transaction data and would need to submit blind queries requesting data of some payment transactions.

For this purpose, there needs to be an enclave that can unseal the previously stored transaction data, verify the client query and return the matching transactions.

This process would again require establishing a secure connection channel between the client and the enclave (if it is remote) as in the transaction data bootstrapping – the difference is that it may only be one-side attested, as the client may not have access to the enclave architecture.

For further details, please refer to this documentation on **transaction query enclave**.

Transaction creation

When the client wishes to broadcast a payment transaction, they first need to obfuscate its content which can only be done within enclaves (that have been up-to-date with the network).

For this purpose, there needs to be an enclave that can access the current random symmetric key obtained from other enclaves (similarly to the transaction validation enclave that needs it for decryption), so that it can encrypt the payment transaction content.

Enclave breaches

If enclaves were breached, it would lead to reduced confidentiality – there would still be a level of confidentiality, as the multi-signature scheme in Chain records only limited information in the blockchain (see signature schemes). It would only affect transactions that were obfuscated with the breached key, as the key would be periodically rotated (see transaction privacy).

Note that the breach wouldn't lead to the loss of ledger integrity, as that is preserved by the consensus algorithm.

← Client Flow

Transaction Privacy →

Join Crypto.com chain as a validator

Anyone who wishes to become a validator can submit a `NodeJoinTx`. This document will guide you through how to join Crypto.com chain as a validator by using the client-cli.

Pre-requisites

- Fully functional validation node;
- Sufficient amount of CRO for staking;
- Wallet with transfer and staking addresses.

Querying the staking related parameters

First of all, we can query the current staking related parameters of Crypto.com chain by using the tendermint rpc. In particular, one would want to check the value of `required_council_node_stake`, which is the minimum staking amount required to become a validator:

```sh
curl -s 'http://localhost:26657/abci_info' | grep required_council_node_stake
```

Example

```json
"required_council_node_stake\":\"625000000000000000\"
```

implies that you would need at least `625000000000000000` basic unit of CRO to become a validator, which is equivalent to `6250000000` CRO.

The minimum staking amount is the first requirement for being a validator. Besides this, note that only the top `MAX_VALIDATORS` with the most `bonded` amount would be considered as an *active* validator (see here for details). This network parameter can be found by:

```sh
curl -s 'http://localhost:26657/abci_info' | grep max_validators
```

Checking the current validators set

You can check the current validator set and their voting power by

```sh
curl -s 'http://localhost:26657/validators'
```

Staking the funds

Deposit required funds from your transfer address to your staking address.

Sending a `NodeJoinTx`

Once the funds have been sent to the staking address, we can submit a `node-join` transaction and join as a validator by

```sh
./bin/client-cli transaction new --name <NAME_OF_YOUR_WALLET> --type node-join
```

You will be requested to unlock your wallet and fill in the following fields:

- Staking address with the required funds;
- Name for the validator node;
- base64 encoded public key of the validator. It can be found under `~/.tendermint/config/priv_validator_key.json`.

Once the transaction has been submitted, you can then check the current validator set to confirm your validator is running.

Remarks

- Validators are responsible for signing or proposing block at each consensus round. It is important that they maintain excellent availability and network connectivity to perform these tasks. A penalty performed by the slashing module is imposed on validators' misbehaviour or unavailability to reinforce this.
- A `NodeJoinTx` is considered to be invalid if your staking amount is less then `required_council_node_stake`:

  ```sh
  Error: Tendermint RPC error: verification failed: staked state bonded amount is
  ```

← Reward & Punishment

List of network parameters →

Local Full Node Development

We will need the following to run a local full node:

- Genesis: Defining the initial state of the blockchain;
- Tendermint Node: Performing consensus operations;
- Transaction Enclave: Validating transactions;
- Application BlockChain Interface (*ABCI*): Connecting between Tendermint and applications.

Step 1. Generate Genesis

Genesis describes the initial funding distributions as well as other configurations such as validators setup. Firstly, we will need a wallet to receive the genesis funds.

Step 1a) Create a Wallet

To create a wallet, currently, we have client-rpc and client-cli available for this purpose. We will be using client-cli in this guide.

- Create a new *basic* wallet with the name "Default" by running

  ```sh
  $ ./target/debug/client-cli wallet new --name Default --type basic
  ```

  You will be prompted to enter a passphrase.

  Note: The client-cli also supports HD wallet with mnemonic seed. Kindly follow this instruction to create your HD wallet.

- Generate a staking address for the wallet to receive genesis funds. You will be prompted to enter the wallet passphrase again to verify.

  ```sh
  $ ./target/debug/client-cli address new --name Default --type Staking
  Enter authentication token: ## Insert your authentication token ##
  New address: 0x3a102b53a12334e984ef51fda0baab1768116363
  ```

We will be distributing funds to our newly-created wallet address `0x3a102...8116363`.

Step 1b) Initialize Tendermint

- Initialise the Tendermint root directory by running:

```sh
tendermint init
```

It initialises a fresh Tendermint Core data directory to be used by a full node. Specifically, it creates a new validator private key (`priv_validator.json`), and a genesis file (`genesis.json`).

- If you have previously initialized a Tendermint node, you may need to reset it by running:

```sh
tendermint unsafe_reset_all
```

Caution: Only use this rest command in development as it removes the data directory, and all blockchain data will be lost.

Details and field definitions of `genesis.json` can be found here. Note that as in the sample genesis, `app_hash` is initially left as empty. The missing configuration will be completed in the next step.

Step 1c) Create a Genesis configuration

We will use the development tool dev-utils to generate the completed `genesis.json`:

- Create a configuration file `dev-conf.json` in `~/chain/dev-utils/`
- Obtain the following information for generating the configuration of our Genesis file:
  - **Address to Receive Genesis Funds**: We have just created one in the previous step
  - **Genesis Time**: Copy the `genesis_time` from `~/.tendermint/config/genesis.json`
  - **Validator Pub Key**: Copy the `pub_key.value` from `~/.tendermint/config/priv_validator_key.json`
- Replace `{WALLET_ADDRESS}`, `{PUB_KEY}` and `{GENESIS_TIME}` with the information obtained above.

```json
{
"distribution": {
```

```
        "{WALLET_ADDRESS}": "2500000000000000000",
        "0x3ae55c16800dc4bd0e3397a9d7806fb1f11639de": "1250000000000000000"
    },
    "unbonding_period": 60,
    "required_council_node_stake": "1250000000000000000",
    "jailing_config": {
        "jail_duration": 86400,
        "block_signing_window": 100,
        "missed_block_threshold": 50
    },
    "slashing_config": {
        "liveness_slash_percent": "0.1",
        "byzantine_slash_percent": "0.2",
        "slash_wait_period": 10800
    },
    "rewards_config": {
        "monetary_expansion_cap": "6250000000000000000",
        "distribution_period": 86400,
        "monetary_expansion_r0": 450,
        "monetary_expansion_tau": 14500000000000000,
        "monetary_expansion_decay": 999860
    },
    "initial_fee_policy": {
        "base_fee": "1.1",
        "per_byte_fee": "1.25"
    },
    "council_nodes": {
        "0x3ae55c16800dc4bd0e3397a9d7806fb1f11639de": [
            "test",
            "security@example.com",
            {
                "type": "tendermint/PubKeyEd25519",
                "value": "{PUB_KEY}"
            }
        ]
    },
    "genesis_time": "{GENESIS_TIME}"
}
```

- Next, we generate the Genesis configuration based on the above configuration file.

```sh
$ ./target/debug/dev-utils genesis generate --genesis_dev_config_path ./dev-uti
```

Step 2. Start Transaction Enclaves

We also need the transaction enclave for validating transactions; it can run on the hardware platform or a virtual machine. Follow the instructions in Crypto.com Chain Transaction Enclaves to build and run the Chain Transaction Enclaves.

Step 3. Start Chain ABCI

To start the Chain ABCI, we will need two pieces of data:

- **App Hash**: Prepared in the Generate Genesis step
- **Full Chain ID**: Copy the `chain_id` found in `~/.tendermint/config/genesis.json` (e.g. test-chain-y3m1e6-AB)

We can start the ABCI by running:

```sh
chain-abci -g <APP_HASHx> -c <FULL_CHAIN_ID> --enclave_server tcp://127.0.0.1:25
```

If you need backtrace or logging, you may set the environment variables before it:

```sh
$ RUST_BACKTRACE=1 RUST_LOG=info \
chain-abci \
-g <APP_HASH> \
-c <FULL_CHAIN_ID> \
--enclave_server tcp://127.0.0.1:25933
```

Final Step: Start Tendermint Node

Once we have the **transaction enclave** and **chain-abci** running, we can start our tendermint node simply by :

```sh
tendermint node
```

Or alternatively, we can start a basic lite-client node by running

```sh
tendermint lite
```

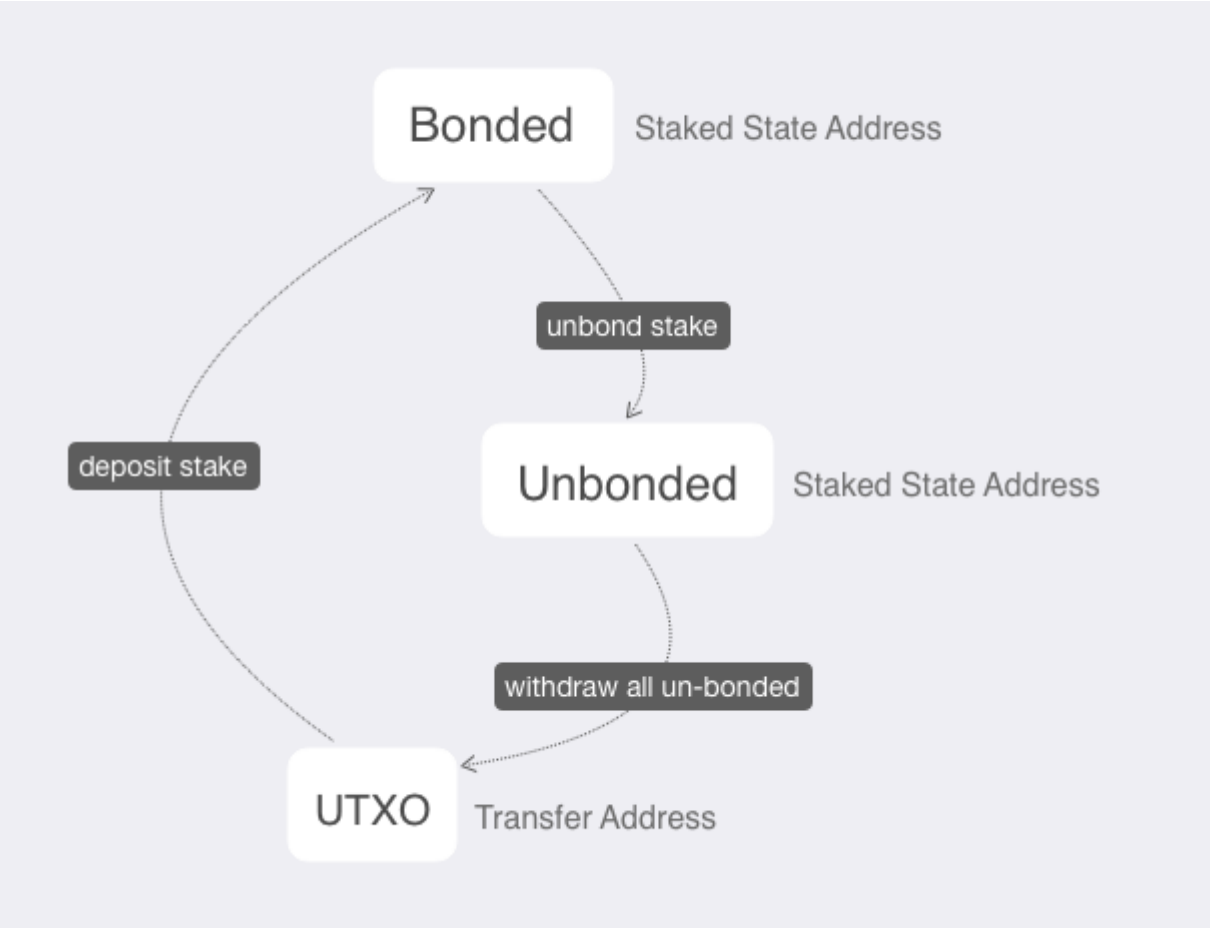
← Thaler Testnet: Running Nodes (Linux only)

Consensus →

Send Your First Transaction

> **Caution**
>
> This page is a continuation of running a local chain node of Getting Started for development.
>
> For anyone interested more on wallet management, getting testing token from the faucet and sending transaction, please refer to ClientCLI.

Before sending the transaction, please notice that the genesis fund is **bonded** (or staked) at the beginning. You first have to **withdraw** it to UTXO:



Restore the Default wallet

From the getting started section, we have already kicked off the Crypto.com Chain with the simple make commands and docker. However, there are multiple tendermint configs should be explained.

- Restore the HD wallet and name it as `Default` :

```sh
$ ./target/debug/client-cli wallet restore --name Default

Enter passphrase:       // Enter your passphrase here
Confirm passphrase:     // Confirm your passphrase
Enter mnemonic:         // Copy the mnemonic words here
Confirm mnemonic:       // Confirm the mnemonic

Authentication token: b74ce4590ebc9d3c2a3adace926304384ae9451f43560c9702402be5381
```

Then you will get the `Authentication token` , keep the token safe and it will be needed for all authorised commands.

Create Transfer & Staking Address

- First, you should init the two `Staking` type address with the `Default` wallet you just restored:

```sh
$ ./target/debug/client-cli address new --name Default --type Staking
Enter authentication token:       // Input the Authentication token
New address: 0x45c1851c2f0dc6138935857b9e23b173185fea15
```

- Run another time and you will get the address that stores the unbonded funds.

```sh
$ ./target/debug/client-cli address new --name Default --type Staking
Enter authentication token:       // Input the Authentication token
New address: 0x2dfde2178daa679508828242119dcf2114038ea8
```

- Then, you should create a `Transfer` type address to receive funds using `Default` wallet:

```sh
$ ./target/debug/client-cli address new --name Default --type Transfer
Enter authentication token:       // Input the Authentication token
New address: dcro1ayhu0665wprxf86letqlv8x4ssppeu6awf7m60qlwds9268vltwsk6ehwa
```

To be clarified, the genesis fund is stored in a pre-created Hierarchical Deterministic(HD) Wallet mnemonic here. So we should restore it before making transactions.

- Therefore, you can check the bond status of the wallet by the following command:

```sh
$ /target/debug/client-cli state --address 0x2dfde2178daa679508828242119dcf211403
```

```
+-----------------+-----------------------------+
| Nonce           |                           0 |
+-----------------+-----------------------------+
| Bonded          |                  0.00000000 |
+-----------------+-----------------------------+
| Unbonded        |        5000000000.00000000 |
+-----------------+-----------------------------+
| Unbonded From   | 2019-11-20 08:56:48 +00:00 |
+-----------------+-----------------------------+
| Jailed Until    |                  Not jailed |
+-----------------+-----------------------------+
| Punishment Type |               Not punished |
+-----------------+-----------------------------+
| Slash Amount    |               Not punished |
+-----------------+-----------------------------+
```

Withdrawal the bonded funds

Bonded address: Previously generated address in your wallet to receive genesis funds

Transfer address: Wallet Transfer address we just generated

```sh
$ ./target/debug/client-cli transaction new --name Default --type Withdraw
Enter authentication token:       // Input the Authentication token
Enter staking address: 0x2dfde2178daa679508828242119dcf2114038ea8
Enter transfer address: dcro1ayhu0665wprxf86letqlv8x4ssppeu6awf7m60qlwds9268vltws
Enter view keys (comma separated) (leave blank if you don't want any additional v
// Leave blank because this tx is in same wallet
```

- Then, you can `sync` and check `balance` of your wallet:

```sh
$ ./target/debug/client-cli sync --name Default
Enter authentication token:       // Input the Authentication token
Synchronizing: 1951 / 1951 [==================================] 100.00 % 930.09/s
Synchronization complete!
```

- You can now check your `balance`. Noted that the `Avaiable` only includes the transferable balance and the bonded/unbonded amount are not included:

```sh
$ ./target/debug/client-cli balance --name Default
Enter authentication token:       // Input the Authentication token
+-----------+---------------------+
| Total     | 5000000000.00000000 |
+-----------+---------------------+
| Pending   | 0.00000000          |
+-----------+---------------------+
| Available | 5000000000.00000000 |
+-----------+---------------------+
```

Congratulations! You successfully withdraw all the unbonded genesis fund and now can transfer to others.

Transfer CRO to another address

- First you can creat another wallet with the name `Bob`, or whatever name you like. The wallet type could be `hd` (Hierarchical Deterministic) or `basic`:

```sh
./target/debug/client-cli wallet new --name bob --type hd
Enter passphrase:
Confirm passphrase:
Please store following mnemonic safely to restore your wallet later:
Mnemonic: cabin typical scheme rather hood sunny salon mansion hazard update vide
Authentication token: 650aca93fdb6e6eeb988026d92e796c28f0306390a49d6bfd75160ea07e
```

- Get view-key of `Default` wallet and `Bob` wallet:

```sh
$ ./target/debug/client-cli view-key --name Default
Enter authentication token:       // Input the Authentication token of Default
View Key: 02b4dabfc862b9cb9f86b8d49520023aa0cccb2ad89446577dd0fee7bc946a79a1

$ ./target/debug/client-cli view-key --name Bob
Enter authentication token:       // Input the Authentication token of Bob
View Key: 03ef78b2751d43c3309b6ac68641e56528a23dc5678a201e43a7ed852511a1c276
```

> **Tip**
>
> The following 3 types of transactions: `TransferTx` , `DepositStakeTx` and `WithdrawUnbondedTx` , have some of their payloads obfuscated. Sender could associate one or more `view-keys` onto the transactions. The view-key associated wallet could easily sync and view the transaction.
>
> For more information, you could refer to Transaction Accounting Model

- Create Transfer address, which is a Transfer address, of `Bob` wallet:

```sh
$ ./target/debug/client-cli address new --name Bob --type Transfer
Enter authentication token:       // Input the Authentication token of Bob
New address: dcro135w20p56vdduzv5e4v4g2a9ucu6vw9k25aeyd7jfxuej66l4af9s7ycz35
```

- Then, you can transfer your tokens to Bob by:

```sh
$./target/debug/client-cli transaction new --name Default --type Transfer

Enter authentication token:       // Input the Authentication token of Default
Enter output address: dcro135w20p56vdduzv5e4v4g2a9ucu6vw9k25aeyd7jfxuej66l4af9s7y
Enter amount (in CRO): 12345678   // CRO token amount you will transfer to Bob
Enter timelock (seconds from UNIX epoch) (leave blank if output is not time locke
More outputs? [yN]
Enter view keys (comma separated) (leave blank if you don't want any additional v
02b4dabfc862b9cb9f86b8d49520023aa0cccb2ad89446577dd0fee7bc946a79a1,03ef78b2751d43
```

> **Tip**
>
> Remember to include Bob's `view-key` here.

- Lastly, you can `sync` and check `balance` of Bob's wallet:

```sh
$ ./target/debug/client-cli sync --name Bob
Enter authentication token:       // Input the Authentication token
Synchronizing: 5121 / 5121 [==================================] 100.00 % 1606.16/s
Synchronization complete!
```

Check the `balance` :

```sh
$ ./target/debug/client-cli balance --name Bob
Enter authentication token:       // Input the Authentication token of Bob
+-----------+--------------------+
| Total     | 12345678.00000000 |
+-----------+--------------------+
| Pending   | 0.00000000        |
+-----------+--------------------+
| Available | 12345678.00000000 |
+-----------+--------------------+
```

Congratulations! You've successfully transferred `12345678 CRO` to Bob.

If you are interested in contributing or joining our testnet, you can continue reading the following sections about `Running a Full Node to join Thaler Testnet` and `Local Full Node Development` .

Export & Import Tx

As mentioned before, sender should add the receiver's view-key to the transaction. Because sender can't push data directly to the receiver. However, it is also possible to send / receive a payment by directly exchanging the (raw) transaction payload data. The sender (who creates the transaction) would export it, the receiver would import it and check the transaction data locally and check the transaction ID against the distributed ledger. Following explains the flow:

1. **Sender**: Get your transaction id from the history, you may need to sync before running the following command:

   ```sh
   $ ./target/debug/client-cli history --limit ? --offset ? --name <sender_wallet>
   Enter authentication token: ## Insert your authentication token ##
   +----------------+--------+--------+-----+--------------+------------+
   | Transaction ID | In/Out | Amount | Fee | Block Height | Block Time |
   +----------------+--------+--------+-----+--------------+------------+
   |<transaction_id>|........|........|.....|..............|............|
   +----------------+--------+--------+-----+--------------+------------+
   ```

2. **Sender**: Export the target transaction payload from the sender's wallet:

```sh
$ ./target/debug/client-cli transaction  export --id <transaction_id>  --name <
Enter authentication token: ## Insert your authentication token ##

## transaction_payload_example ##
eyJ0eCI6eyJ0eXBlIjoiVHJhbnNmZXJUcmFuc2FjdGlvbiIsImlucHV0cyI6W3siaWQiOiI3ZDk3NzV
```

3. **Receiver**: The transaction can be imported into receiver's wallet by

```sh
$ ./target/debug/client-cli transaction import --tx <transaction_payload> --nam

Enter authentication token: ## Insert your authentication token ##

import amount: <transaction_amount>
```

4. Finally, receiver can verify this transaction by checking the transaction history:

```sh
$ ./target/debug/client-cli history --limit ? --offset ? --name <receiver_walle
```

← Getting Started

Thaler Testnet: Running Nodes (Linux only) →

Genesis

The `genesis.json` file defines the initial state of the Crypto.com chain. On top of the standard tendermint genesis format, we customize our own genesis file and facilitate the special features of the Crypto.com chain. Sample genesis file can be found here

Fields in genesis

Specifically, the genesis file includes the following fields:

- `"genesis_time"` : The time of beginning of the blockchain
- `"chain_id"` : A unique identifier for the blockchain. The *"network id"* is the last two hex characters of this field. See this for further details.
- `"app_hash"` : The expected application hash upon genesis (returning from the chain-abci).
- `"app_state"` : The application state.
  - `"council_nodes"` : The initial validator set.
    - `"keypackage"` : The public information that other nodes can use for key agreement, see this for further details.
  - `"distribution` : The initial distribution and bonding state of the tokens.
  - `"network_params"` : The network parameters
    - `"initial_fee_policy"` : The transaction fee policy.
    - `"jailing_config"` : The configuration of punishment (jailing)
      - `"block_signing_window"` : The size of the sliding window for calculating validators's liveness.
      - `"missed_block_threshold"` : The threshold of total missed blocks.
    - `"max_validators"` : The maximum number of validator.
    - `"required_council_node_stake"` : The minimum staking amount required to be a validator.
    - `"rewards_config"` : Configuration of validator's reward.
      - `"monetary_expansion_cap"` : The total amount of tokens reserved for validator's reward, in basic unit.
      - `"monetary_expansion_r0"` : The upper bound for the reward rate per annum.

        - `"monetary_expansion_tau"` : The initial value of tau in the reward function.
        - `"monetary_expansion_decay"` : The decay rate of tau.
        - `"reward_period_seconds"` : The period of reward being distributed, in seconds.
    - `"slashing_config"` : Configuration of validator's punishment.
        - `"byzantine_slash_percent"` : The maximum percentage of stake reduction for byzantine validators.
        - `"liveness_slash_percent"` : The maximum percentage of stake reduction for validators with low availability.

- `"consensus_params"` : The consensus critical parameters that determine the validity of blocks.
    - `"block"` : The block formation structure.
        - `"max_bytes"` : The maximum block size, in bytes.
        - `"max_gas"` : The default value is "-1", i.e., no rules about gas are enforced.
        - `"time_iota_ms"` : The minimum time increment between consecutive blocks, in milliseconds.
    - `"evidence"` : The configuration of evidence of malfeasance by validators.
        - `"max_age_num_blocks"` : *This field is to be deprecated.*
        - `"max_age_duration"` : The maximum age of evidence, in nanoseconds. Any evidence older than this will be rejected. It is also the "unbonding period", the time duration of unbonding.
    - `"validator"`
        - `"pub_key_types"` : The supported validator public key types.

- `"validators"` : The initial state of the validator set.
    - `"address"` : The address of the validator (hash of the validator's public key). Note that the validator is identified by its address.
    - `"name"` : The name of the validator, which should correspond to the name in *"council_nodes"*.
    - `"power"` : The voting power of the validator, which should match with the bonded amount of its related staking address in *"distribution"* in full units.
    - `"pub_key"`
        - `"type"` : The validator public key types, which should be one of the supported types in *"pub_key_types"*.

- `"value"` : The base64 encoding of validator's public key, which should correspond to the *"value"* of the validator in *"council_nodes"*.

Genesis fingerprint

Light client allows users to access and interact with the blockchain without having to run a full node. To ensure the light client is served by a full node with the correct blockchain data, we can compute the **genesis fingerprint** locally beforehand and check it against the full node that we are connecting to.

Specifically, this **genesis fingerprint** is computed by the `compute_genesis_fingerprint` function: where we take `"genesis_time"` , `"app_hash"` , `"consensus_params"` ; and `"chain_id"` from the genesis file as inputs and hash it by the *blake3* hash function.

We can use the development tool *dev-utils* to generate this genesis fingerprint from the `genesis.json` file in the default tendermint directory `~/.tendermint/config` , for example:

```sh
$ ./dev-utils genesis fingerprint
85828B9048F3A38C0446CFD8EFF1A33CE7F299E6605001238B63684E9633EE4E
```

> **WARNING**
>
> this doc may be outdated and is subject to revision

The initial state of the network is started from the existing ERC20 contract on Ethereum. On the Ethereum mainnet, it is 0xA0b73E1Ff0B80914AB6fe0444E65848C4C34450b.

The current genesis procedure is the following:

1. The initial set of validator (council node) identities is established: each council node should have an associated validator public key (from the Tendermint Ed25519 keypair, used for signing blocks) and initial staking address (same as Ethereum), ...
2. The initial network parameters (e.g. minimum stake amounts) are chosen
3. At a determined time, the list of ERC20 holders is snapshotted / computed from the Ethereum network.
4. The list should contain three dedicated addresses for:

- Secondary distribution and launch incentive (0x20a0bee429d6907e556205ef9d48ab6fe6a55531 and 0x35f517cab9a37bc31091c2f155d965af84e0bc85 on the Ethereum mainnet)
- Long term incentive (0x71507ee19cbc0c87ff2b5e05d161efe2aac4ee07 on the Ethereum mainnet)

The balances of these dedicated addresses are put to the initial "Rewards Pool" (where transaction fees are paid to and network operation rewards are paid from).

1. For every ERC20 holder address in the initial distribution, besides the above three addresses, the following rules are used for computing the initial genesis state:

- If the address is owned / controlled by a smart contract, its balance goes to the initial "Rewards Pool"

> **WARNING**
>
> DEX, multisig etc. contracts should be avoided during the Step 3

- If the address is an externally owned account and corresponds to the initial staking address of one of council nodes, its balance starts as "bonded" in the corresponding staked state (see accounting mode for more details).
- Otherwise (i.e. the address is an externally owned account, but not of any validators), the address balance starts as "unbonded" in the corresponding staked state (see accounting mode for more details).

From this initial genesis state, the initial "application hash" (APP HASH) is computed and set in the corresponding genesis.json file of Tendermint and compiled statically into the enclave binaries (that need to be signed by the developer production keys).

Tendermint extra information

As described in consensus, Tendermint executes with the application-specific code via ABCI.

The genesis information (network parameters, initial validators etc.) is set in the `app_data` field in genesis.json – this information is then passed to the ABCI application in the InitChainRequest.

← Consensus

Chain ID and Network ID →

Notes on Performance

The current discourse in cryptocurrencies focuses on "maximum TPS" as the key defining performance metric and one often finds various outlandish claims about it. For more details why these numbers don't matter, we recommend this article from the Nervos Network.

> **NOTE**
>
> There are more issues with regards to "maximum TPS metrics" that are not mentioned in the article

We'll have to eat humble pie and say that some of the existing Crypto.com Chain materials fall into similar fallacies (despite being written with the word "aim" and intended to be seen in the potential Layer-2 context).

In the future, once the system reaches certain maturity, we'd like to establish metrics (e.g. the 95th percentile of transaction end-to-end latency) that are more meaningful for user experience. We would try to benchmark against these metrics, ideally in diverse settings (rather than "the best case where nothing ever crashes") and using estimated real workloads.

← Notes on Production Deployment

Threat Model →

List of network parameters

This section aims to collect and provide brief a description of all the mentioned network parameters:

Staking-related parameters

Key	Description
`unbonding_period`	The time duration of unbonding
`max_validators`	The maximum number of validator
`required_council_node_stake`	The minimum staking amount required to be a validator

Reward parameters

Key	Description
`monetary_expansion_cap`	The total amount of tokens reserved for validator's reward in the basic unit
`reward_period_seconds`	The period of reward being distributed (unit: seconds)
`monetary_expansion_r0`	The upper bound for the reward rate per annum
`monetary_expansion_tau`	Initial value of tau in the reward function
`monetary_expansion_decay`	The decay rate of tau.

Slashing parameters

Key	Description
`MAX_EVIDENCE_AGE`	Maximum age of evidence
`block_signing_window`	Window to calculate validators's liveness
`jail_duration`	The time period for jailing
`missed_block_threshold`	Threshold of total missed blocks
`byzantine_slash_percent`	Maximum percentage of stake reduction for byzantine validators

Key	Description
`liveness_slash_percent`	Maximum percentage of stake reduction for validators with low availability

Transaction fee parameters

Key	Description
`BASE_AMOUNT`	Basic transaction fee
`PER_BYTE`	Transaction fee per byte

TODO: TX that can change them?

← Join Crypto.com chain as a validator

Notes on Production Deployment →

Reward & Punishment

This document contains the specification of the rewards and punishment mechanisms which establish the foundation of the token ecosystem of CRO.

Validator Rewards

To incentivise validators to run the network, rewards are accumulated and distributed to the validators. There are three sources for the rewards:

1. Monetary expansion with a fixed total supply
2. Transaction Fees
3. Slashing of byzantine and non-live nodes (if any)

The `RewardsPoolState` data structure stores all the information about the remaining funds and distribution states:

```
pub struct RewardsPoolState {
    pub period_bonus: Coin, // rewards accumulated from fees and slashing during
    pub last_block_height: BlockHeight, // when was the pool last updated
    pub last_distribution_time: Timespec, // when was the pool last distributed
    pub minted: Coin,  // record the number of new coins ever minted, can't excee
    pub tau: u64,  // a decaying parameter in monetary expansion process
}
```

Overview: The reward function

Motivation: To incentivise the validators, the amount of the reward should dynamically react to the actual network conditions such as the *total staking* and the *length of time* since the genesis block.

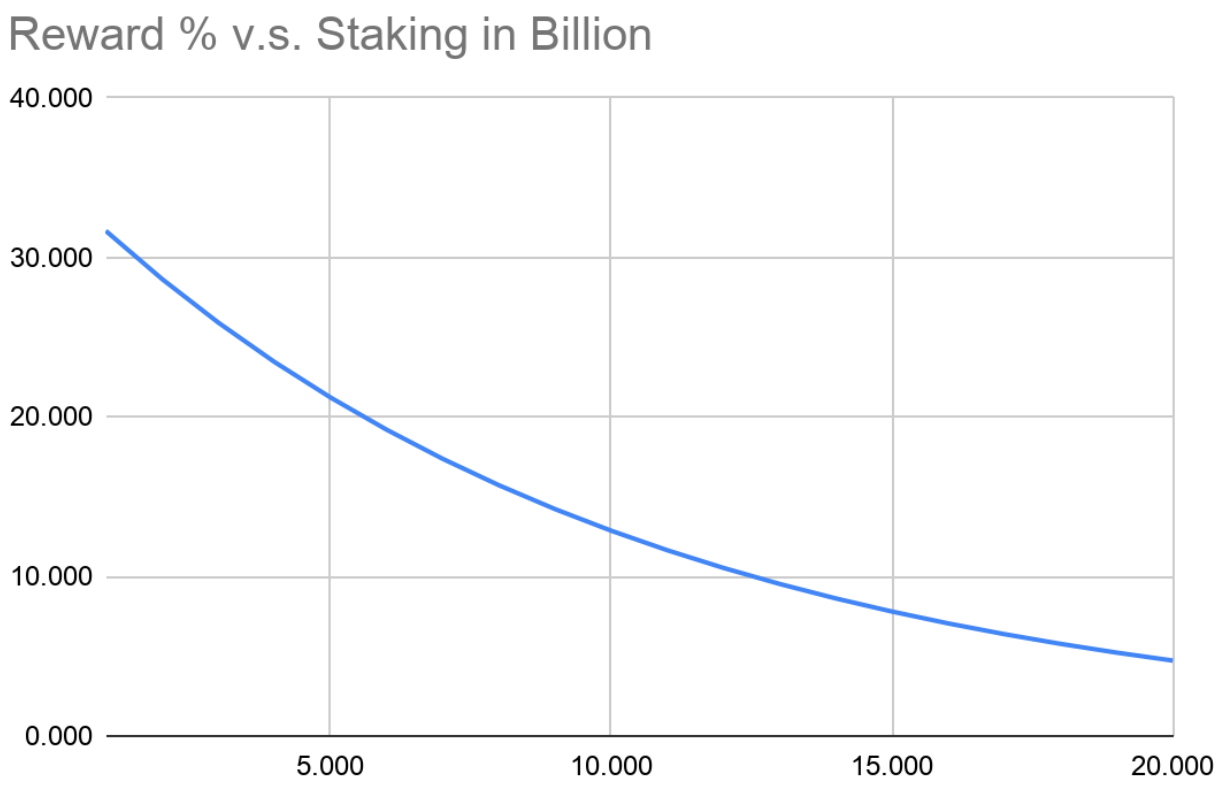
The reward being sent to the reward pool depends on two major factors:

- **S**: The total amount of staking, and
- **R**: The reward rate per annum.

Specifically, this reward rate `R` can be expressed by the following function:

```
R = (R0 / 1000) * exp(-S / tau)
```

To visualize this, if we set `tau=10 Billion`, `R0=350`, we have the following graph of the function:



Note that the above reward rate is per annum, and the number of tokens being released to the reward pool (**N**) for that epoch is calculated by

```
N = S  *  ( (1 + R) ^ (1 / f) - 1 )
```

where `S` is the total amount of tokens staked by validators to participate in the consensus process; `R0` is the upper bound for the reward rate; `tau` is a time-dependent variable that controls the exponential rate; `f` is the frequency of reward being distributed per year.

> **Example: If the reward rate is 28% with total staking of 500 million and the reward is being distributed every day, the number of tokens being released to the reward pool at the end of the day will be**
>
> `500,000,000 * ( (1+0.28)^(1/365) - 1) = 338,278`

More details about the actual calculation and its configuration can be found in the next section.

Rewards: Network parameters for monetary expansion

Monetary expansion is designed to release tokens from the reserve account to the reward pool, while keeping a fixed maximum total supply.

Precisely, the reward rate is controlled by the following parameters:

- `monetary_expansion_cap` : The total amount of tokens reserved for validator's reward in the basic unit
- `reward_period_seconds` : The period of reward being distributed
- `monetary_expansion_r0` : The upper bound for the reward rate per annum
- `monetary_expansion_tau` : Initial value of tau in the reward function
- `monetary_expansion_decay` : The decay rate of tau.

You can find the configuration under the `rewards_params` section of the genesis file `genesis.json` in `./tendermint/config` ,

▶ Example: Reward configuration in the genesis

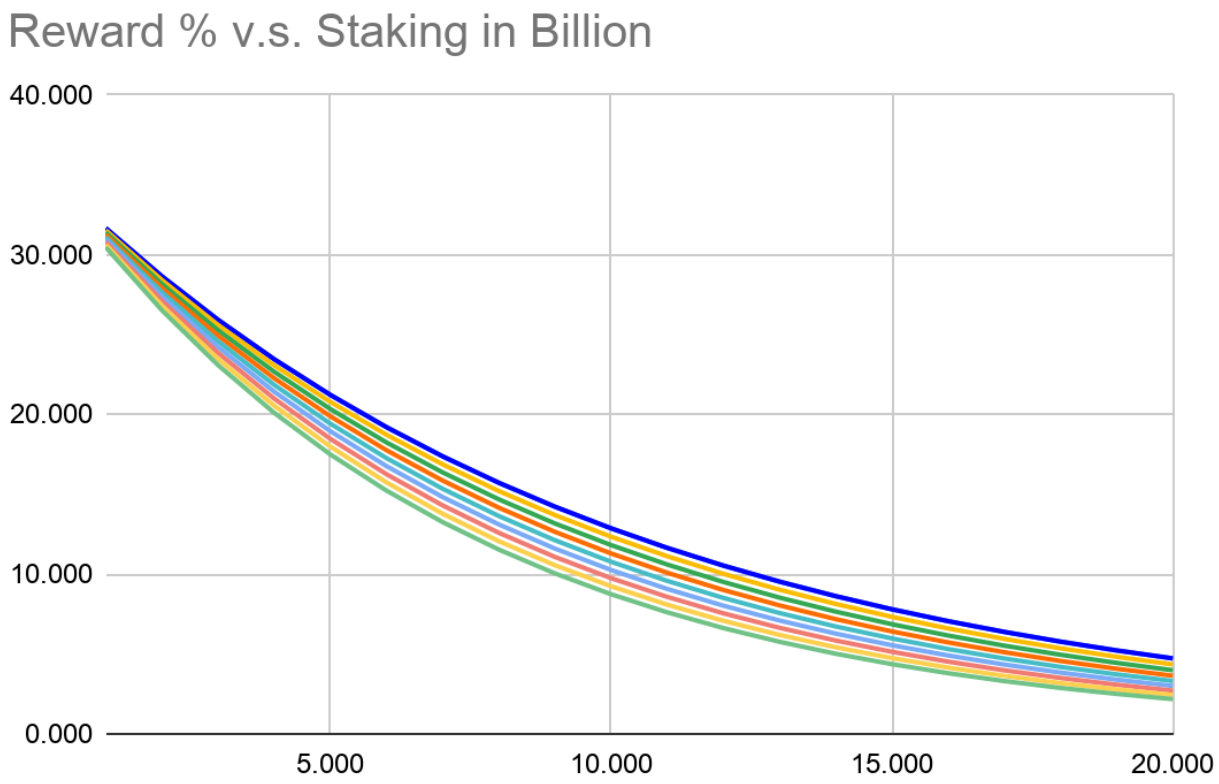
▶ Reward parameters explained:

At the end of each reward epoch, the number of tokens being released at each period is defined as:

```
R0 = rewards_config["monetary_expansion_r0"]
tau = rewards_config["monetary_expansion_tau"]
P = rewards_config["reward_period_seconds"]

# total bonded amount of the active validators
at the end of the reward epoch
S = total_staking

# seconds of a year
Y = 365 * 24 * 60 * 60

R = (R0 / 1000) * exp(-S / tau)
```

```
    N = floor(S * (pow(1 + R, P / Y) - 1))
    N = N - N % 10000

    N: released coins
    S: total stakings at current block when
       reward distribution happens
    R0: upper bound for the reward rate p.a.
```

Using the example of `tau=10 Billion` and `R0=350`, the following graph shows how the reward rate deforming when `tau` is dropping by 5% every year:



The rewards validator received goes to the bonded balance of their staking account, and results in validator voting power change accordingly.

Reward distribution

Rewards are distributed periodically (e.g. daily), which is determined by the network parameter `reward_period_seconds`. Rewards are accumulated during each period, block proposers and the total voting power of all validators are recorded.

At the end of each reward period, validators will receive a portion of the "reward pool" as a reward for participating in the consensus process. Specifically, the reward of a validator is proportional to its

contribution to the consensus process; it is calculated as follows:

```
rewards of a validator = total rewards in the pool * [Validatior's contribution]
```

where

- *"Validator's contribution"* is the total sum of the voting power participating in the consensus process by the validator throughout the reward period; and
- *"Sum of the voting power"* represents the total sum of the voting power involved in the consensus process from all of the active validators throughout the reward period.

The remainder of division will become rewards of the next period.

Rewards: Documentation and its interactions with ABCI

The detailed documentation of the reward mechanism can be found in here.

Validator Punishments

This part describes functionality that aims to dis-incentivize network-observable actions, such as faulty validations, of participants with values at stake by penalizing/slashing and jailing them. The penalties may include losing some amount of their stake (surrendered to the rewards pool), losing their ability to perform the network functionality for a period of time, collect rewards etc.

Punishments: Network Parameters for slashing

Below are all the network parameters used to configure the behavior of validator punishments. Details of all these parameters and their effect on behavior of validator punishments is discussed later in this document.

1. `UNBONDING_PERIOD` : Unbonding period will be used as jailing period (time for which an account is jailed after it gets punished) and also as slashing period (time to wait before slashing funds from an account). This should be greater than or equal to `MAX_EVIDENCE_AGE` in tendermint.
2. `BLOCK_SIGNING_WINDOW` : Number of blocks for which the liveness is calculated for uptime tracking.
3. `MISSED_BLOCK_THRESHOLD` : Maximum number of blocks with faulty/missed validations allowed for an account in last `BLOCK_SIGNING_WINDOW` blocks before it gets jailed.

4. `LIVENESS_SLASH_PERCENT` : Percentage of funds (bonded + unbonded) slashed when a validator is non-live.
5. `BYZANTINE_SLASH_PERCENT` : Percentage of funds (bonded + unbonded) slashed when validator makes a byzantine fault.

> **Important:**
>
> During slashing, funds are slashed from both, bonded and unbonded, amounts.

Overview

Punishments for a validator are triggered when they either make a *byzantine fault* or become *non-live*:

- Liveness Faults (Low availability)

  A validator is said to be **non-live** when they fail to sign at least `MISSED_BLOCK_THRESHOLD` blocks in last `BLOCK_SIGNING_WINDOW` blocks successfully. `BLOCK_SIGNING_WINDOW` and `MISSED_BLOCK_THRESHOLD` are network parameters and can be configured during genesis (currently, changing these network parameters at runtime is not supported). Tendermint passes signing information to ABCI application as `last_commit_info` in `BeginBlock` request.

> **Example:**
>
> For example, if `BLOCK_SIGNING_WINDOW` is `100` blocks and `MISSED_BLOCK_THRESHOLD` is `50` blocks, a validator will be marked as **non-live** if they fail to successfully sign at least `50` blocks in last `100` blocks.

- Byzantine Faults (Double signing)

  A validator is said to make a byzantine fault when they sign conflicting messages/blocks at the same height and round. Tendermint has mechanisms to publish evidence of validators that signed conflicting votes (it passes this information to ABCI application in `BeginBlock` request), so they can be punished by the application.

> **Implementation note:**
>
> Tendermint passes `Evidence` of a byzantine validator in `BeginBlock` request. Before jailing any account because of byzantine fault, that evidence should be verified. Also, it

> should be checked that evidence provided by tendermint is not older than `MAX_EVIDENCE_AGE` in tendermint.

Inactivity Slashing

It is important that the validators maintain excellent availability and network connectivity to perform their tasks. A penalty should be imposed on validators' misbehavior to reinforce this.

When a validator fails to successfully sign `MISSED_BLOCK_THRESHOLD` blocks in last `BLOCK_SIGNING_WINDOW` blocks, it is immediately punished by deducting funds from their bonded and unbonded amount and removing them from active validator set. The funds to be deducted are calculated based on `LIVENESS_SLASH_PERCENT` .

> **Note:**
>
> The validator is not **jailed** in this scenario. They can immediately send a `NodeJoinTx` to join back as a validator if they are qualified (have enough bonded amount and not jailed).

Jailing

A validator is jailed when they make a byzantine fault, e.g., they sign messages at same height and round.

When a validator gets jailed, they cannot perform any operations relating to their account, for example, `withdraw_stake` , `deposit_stake` , `unbond_stake` , etc., until they are un-jailed. Also, a validator cannot be un-jailed before `account.jailed_until` which is set to `block_time + UNBONDING_PERIOD` while jailing. `UNBONDING_PERIOD` is a network parameter which can be configured during genesis.

> **Important:**
>
> `block_time` used in calculating `account.jailed_until` should be the time of the block at which the fault is detected (i.e., `current_block_height`).

> **Important:**
>
> When a validator is jailed because of a byzantine fault, their validator public key is added to a list of permanently banned validators and cannot re-join the network as a validator with

> same public key.

Un-jailing

When a jailed validator wishes to resume normal operations (after `account.jailed_until` has passed), they can create `UnjailTx` which marks them as un-jailed. After successful un-jailing, validators can submit a `UnbondTx` and `WithdrawTx` to withdraw their funds.

Byzantine Slashing

Validators are responsible for signing or proposing block at each consensus round. A penalty should be imposed on validators' misbehavior to reinforce this. When there is byzantine fault detected, they are immediately slashed other than jailed. During the jailing time, it won't be slashed again for other byzantine faults. The funds to be deducted are calculated based on `BYZANTINE_SLASH_PERCENT`.

> **Important:**
>
> A validator should not be slashed more than once for byzantine faults within `UNBONDING_PERIOD`. If a validator commits multiple byzantine faults within that time period, it should only be slashed once (for simplicity, we'll only slash the validator for the first evidence that we get from tendermint and ignore other evidences until `UNBONDING_PERIOD`).

> **Implementation note:**
>
> To enforce that we only slash an account only once within `UNBONDING_PERIOD`, we can just check if an account is not jailed when we receive evidence of misbehavior from tendermint.

Punishments: Documentation and its interactions with ABCI

The detailed documentation of the slashing mechanism can be found in here

Appendix

Reward related network parameters configuration

The following tables show overall effects on different configuration of the reward related network parameters:

Key	`monetary_expansion_cap`	`reward_period_seconds`	`monetary_ex`
Higher	More reserved validator reward	Less frequent reward distribution	Tau decays slo
Lower	Less reserved validator reward	More frequent reward distribution	Tau decays fas
Constraints	Less than the maximum token supply	Value has to be positive	Positive value 1000000
Sample configuration	2e18 (20% of the total supply)	86400 (reward distributed daily)	999860 (Tau (yearly)

For initial values:

Key	`monetary_expansion_r0`	`monetary_expansion_tau`
Higher	Higher ceiling for reward rate	Steeper exponential curve
Lower	Lower ceiling for reward rate	Flatter exponential curve
Constraints	Value has to be positive	Value has to be positive
Sample configuration	350 (35% reward rate p.a.)	10 Billion

← Staking and Council Node

Join Crypto.com chain as a validator →

Serialization

After several iterations of binary formats, Crypto.com Chain settled on using the SCALE (Simple Concatenated Aggregate Little-Endian) codec.

It is formally described in Section B.1 of Polkadot Host Protocol Specification.

Test Vectors

Key pairs and Addresses

Transaction-related keys use the secp256k1 ECC, as in Bitcoin, Ethereum etc. The following test vectors were generated with the seed `ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff`.

View key pair

A secp256k1 key pair used for authentication when requesting transaction data from enclaves.

Secret: `b4518a5a1594816446f9a50390fe2d5a896b91590b794bcd7ff4bee75d2ab601`

Public key:

```
045935d68aa6ed95edbfa897b8ee6b050d428594c7542c110b24ffc568f885b22b5b9bb40d731161eb
```

NOTE: as this key pair is used in an interactive protocol, wallet client software that supports an external device (e.g. hardware wallets) may work in a "mixed" mode where view keypairs are independently generated and stored locally (unlike transfer secrets which do not leave the external device).

Staking key pair and Address

A secp256k1 key pair used for signing network operations-related transactions.

Secret: `e6733383185784496214adba3968672eae49f0165bcf60ae0c5fd3a570d946e9`

Public key:

```
04deb799363e8e779ac4f2fe6c873b2dbd6870d1cb899ef103c6d3c4af755b81aa1947e293b109bb41
```

The address is derived from the public key in the same way as externally owned account addresses on Ethereum (rightmost 20 bytes of the Keccak-256 hash of the public key...). Its textual representation is hexadecimal:

`0x410afb0ccf51aefd111bceafb6c298acd4decaf6`

Transfer key pair and Address

A secp256k1 key pair used for signing payment-related transactions. As payment transactions are signed using a BIP-340 -compatible scheme where public keys which only encode X coordinates are used (Y coordinates are chosen implicitly as quadratic residues).

Secret: `865396765de5856157d044f7fbfc93165a9a0226be83b84e376a5d284534f07b`

Public key:

```
04bea01313886bcb3f9b80846fa43df72aa9c59f48478a142ed86afe59e7b595517731c8d77a53ddce
```

X-only public key: `bea01313886bcb3f9b80846fa43df72aa9c59f48478a142ed86afe59e7b59551`

The address is defined as a root of a Merkle tree with X-only public keys being leafs, hashed using Blake3 (32 bytes). We can construct a Merkle tree with one leaf being the above X-only public key. Its textual representation is using Bech32:

- (public) testnet: `tcro1lrmqarjnxqqfsqlw9egc6w0llgnhnveuhtj6l4ryzgqleshm5eeqzre704`
- mainnet: `cro1lrmqarjnxqqfsqlw9egc6w0llgnhnveuhtj6l4ryzgqleshm5eeq0y7gwx`

Other examples

Secret: `9b7e526baa8cfcb757dd3462b0df07d62a4453e04ada1ec9325f4b6e61a577ca`

Public key:

```
04d61feba9a9aa5ca43981eeb3c23893baa5010821569f7f8b71dc987ad4114aefb4ba735d5de5c940
```

X-only public key: `d61feba9a9aa5ca43981eeb3c23893baa5010821569f7f8b71dc987ad4114aef`

- (public) testnet: `tcro1v3x0j5jeftkq0cpsahkghfadazthqulj6rh3z5kvuyzmnc8yrpqqcqrrx7`
- mainnet: `cro1v3x0j5jeftkq0cpsahkghfadazthqulj6rh3z5kvuyzmnc8yrpqq48y48d`

--

Secret: `45f3aef34147d00100469dcf8771d52436a8e312a2f8e8214995a76e36e458c7`

Public key:

```
04be31878078caa7919e18b2682cc40a49910ed9131f18a63499f850cf7552369c17bd2df5e0ea195c
```

X-only public key: `be31878078caa7919e18b2682cc40a49910ed9131f18a63499f850cf7552369c`

- (public) testnet: `tcro1h2ukeq63nmcw0l5y7jrkjasfcsmwds87976zqe5levzs068nejjsvjclvl`
- mainnet: `cro1h2ukeq63nmcw0l5y7jrkjasfcsmwds87976zqe5levzs068nejjsp4lfdv`

TODO: examples with combined public keys and multi-leaf trees

Transactions

Attributes

Each transaction payload contains additional transaction attributes. They contain:

- version number (u64): for the 0.5 version, it's "1"
- network id (u8): for the mainnet, it's 0x2A

The attribute payload for "account-based" transactions without transaction outputs (deposit stake, unbond stake, node join), the attribute payload is: `002a0100000000000000`

For transactions with outputs (transfer, withdraw unbonded stake), there is an additional information containing access information -- for the public view key above:

```
045935d68aa6ed95edbfa897b8ee6b050d428594c7542c110b24ffc568f885b22b5b9bb40d731161eb
```

with access to all transaction data, the attribute payload is (the public is serialized in the compressed form):

```
002a04025935d68aa6ed95edbfa897b8ee6b050d428594c7542c110b24ffc568f885b22b0001000000
```

Witness

Each transaction contains a segregated witness payload. For network operations, the witness payload is a recoverable ECDSA (the same as in Ethereum), with the transaction identifier (Blake3 hash of the transaction payload) being the message digest. With the secret above (`e6733383185784496214adba3968672eae49f0165bcf60ae0c5fd3a570d946e9`), here are a few examples:

- txid: `d0a35414d432dc757433e01a1523a53d34ef94dfaee09fd62564f42c783a5ea1`
- witness:

```
0001786bda8b0a8a9bdf8c0e7379f678000cb9865aaa4995213f929cdb21b8f0c0035499e14c6ba
```

- txid: `b079b56268e7e8ecac6c80e546af343fc89ec3e02ae00071b584da048789de05`
- witness:

```
0001eb9e2e9d71f12674ab87682d132eec76018c6e1c3dbd4d992b4122abc0b1cddb758096bf60b
```

- txid: `b857a50a3bc5ad7e66bd62155c608d654ce0a4052ba30d95de5415dc3d6fb6b2`
- witness:

```
000189e2f25993e6db91037ca93befef075c07975add286062433c64c83e6835c21942bad4a5f06
```

For transfer operations, the witness payload is a Merkle proof and BIP340 Schnorr signature (again with the transaction identifier being the message digest). Here are examples signed with the secret `865396765de5856157d044f7fbfc93165a9a0226be83b84e376a5d284534f07b` and a proof for the corresponding x-only public key `bea01313886bcb3f9b80846fa43df72aa9c59f48478a142ed86afe59e7b59551`

- txid: `fac96a0081af98df1a70b7db03825c41109f0b2687fbdbd957938dfcc3a2dbc8`
- witness:

```
040061327a299e52a79bda96ecdf97efbb329f546199e82e9d9813d099a2da57164219acdc13a03
```

- txid: `510f78adeae403e56b40f5c3bea1ff70f3dec2da6911e1e9b1f0ea03203019bc`
- witness:

```
0400db315f21d8dfa747b772ba6af05c3efbbf28eaf7c3cd6fcab91dfb9f566f5a3f2779b84a714
```

TODO: examples with longer Merkle proofs

Plain payloads

Certain transactions have some of their data obfuscated -- the resulting ciphertext-containing transactions (that is broadcasted on the network) is created in an interactive protocol with code running in local or remote enclaves. In this section, we show the plain payloads that can be created anywhere and are then be passed to an enclave before they are broadcasted on the network.

Withdraw unbonded stake

The witness payload is public, but the transaction outputs will be obfuscated. Example for plain payload -- withdrawing 1000 base unit coins from `0x410afb0ccf51aefd111bceafb6c298acd4decaf6` to `cro1lrmqarjnxqqfsqlw9egc6w0llgnhnveuhtj6l4ryzgqleshm5eeq0y7gwx` unbonded from time `0` (with the transaction attributed shown above).

- plain transaction payload:

```
020000000000000000000400f8f60e8e5330009803ee2e518d39fffa2779b33cbae5afd4641201fcc
```

(txid: `d0a35414d432dc757433e01a1523a53d34ef94dfaee09fd62564f42c783a5ea1`)

Transfer

The payload will be fully obfuscated. Example for plain payload:

- spending output 0 of transaction `d0a35414d432dc757433e01a1523a53d34ef94dfaee09fd62564f42c783a5ea1`: `d0a35414d432dc757433e01a1523a53d34ef94dfaee09fd62564f42c783a5ea10000` creating two outputs:
- 100 base unit coins to `cro1h2ukeq63nmcw0l5y7jrkjasfcsmwds87976zqe5levzs068nejjsp4lfdv`: `00644cf952594aec07e030edec8ba7ade8977073f2d0ef1152cce105b9e0e418408403000000000000000`

- 900 base unit coins to `cro1v3x0j5jeftkq0cpsahkghfadazthqulj6rh3z5kvuyzmnc8yrpqq48y48d`: `00bab96c83519ef0e7fe84f487697609c436e6c0fe2fb420669fcb0507e8f3cca564000000000000000000`
- witness and attribute payloads shown above
- full plain transaction payload:

```
0004d0a35414d432dc757433e01a1523a53d34ef94dfaee09fd62564f42c783a5ea100000800644
```

(txid: `fac96a0081af98df1a70b7db03825c41109f0b2687fbdbd957938dfcc3a2dbc8`)

Deposit

Here, the witness payload will be obfuscated, the transaction payload is public. Example for plain payload:

- spending output 0 of transaction `d0a35414d432dc757433e01a1523a53d34ef94dfaee09fd62564f42c783a5ea1`: `d0a35414d432dc757433e01a1523a53d34ef94dfaee09fd62564f42c783a5ea10000`
- witness and attribute payloads shown above
- depositing to the address `0x410afb0ccf51aefd111bceafb6c298acd4decaf6`
- plain transaction payload:

```
04d0a35414d432dc757433e01a1523a53d34ef94dfaee09fd62564f42c783a5ea1000000410afb0
```

(txid: `510f78adeae403e56b40f5c3bea1ff70f3dec2da6911e1e9b1f0ea03203019bc`)

Network operations

Unbond

At nonce 0, unbond 1000 base unit coins with the transaction attributes and witness above for address `0x410afb0ccf51aefd111bceafb6c298acd4decaf6`.

transaction payload:

```
010000410afb0ccf51aefd111bceafb6c298acd4decaf60000000000000000e80300000000000002a
```

(txid: `b857a50a3bc5ad7e66bd62155c608d654ce0a4052ba30d95de5415dc3d6fb6b2`)

Node join request

At nonce 1, with Tendermint Ed25519 consensus pubkey `d75a980182b10ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a`, "example" as a human-readable moniker, "security@example.com" as a security contact, TDBE keypackage (see MLS draft spec) with an example P-256 key and X.509 credential with the test environment IAS payload request a council node joining with address `0x410afb0ccf51aefd111bceafb6c298acd4decaf6` (attribute and witness payloads above).

transaction payload:

```
0102010000000000000000410afb0ccf51aefd111bceafb6c298acd4decaf6002a0100000000000000
```

(txid: `5aa9ef8a47b6b0be7da82d521affb153a7e07726cffecdbaece5d43ddb8dfbe2`)

← Transaction

Signature Schemes →

Signature Schemes

There are two address types which require corresponding signature schemes:

1. "RedeemAddress": Ethereum account address (for ERC20 redeem / backwards-compatibility); see `init/address.rs` in `chain-core`.
2. "Tree": threshold multisig addresses; records a root of a "Merklized Abstract Syntax Tree" where branches are "OR" operations and leafs are Blake3 hashes of aggregated X-only public keys:

- Merklized Abstract Syntax Tree
- MuSig: A New Multisignature Standard

← Serialization

Client Flow →

Transaction Privacy

Motivation

Payment data need confidentiality for many reasons, including compliance with different privacy regulations, fungibility properties etc. While confidentiality can be achieved through different means, Crypto.com Chain leverages symmetric encryption in *Trusted Execution Environments* (**TEE**) for the following benefits:

- *Flexibility* in terms of what computation can be done and how the data schema can evolve. As requirements for Crypto.com Chain change, it's important that the existing code and data remain forwards-compatible. For example, one may extend the transaction format to support new signature schemes.
- *Auditability with potentially fine-grained access control mechanisms*: in the initial implementation, it's a separation of the permission to spend and the permission to view transaction data, but it could be more flexible and fine-grained (e.g. permission to view certain parts of transaction data).
- *Performance due to a low overhead*: unlike, for example, fully homomorphic encryption in software, the overhead of executing computation in TEE should be minimal and the main cryptographic primitive is symmetric encryption which can be accelerated by dedicated CPU instructions, such as AES-NI.

Payloads

Depending on the transaction type (see transaction types and its processing), some of its parts (transaction data, witness or both) need to be obfuscated. In that case, the broadcasted transaction binary payload includes:

- Transaction ID (if the transaction data is obfuscated)
- List of transaction inputs (if relevant) and the number of outputs (if relevant)
- Symmetric encryption-related metadata (key generation, nonce / initialization vector)
- Obfuscated payload

The encryption inside enclaves (see enclave architecture) is done using "authenticated encryption with associated data" (AEAD) scheme – the initial planned algorithm is AEAD_AES_128_GCM_SIV.

Periodic key generation

The active set of validators is involved in periodic generation of new keys that are then used by all full node enclaves on the network – the key distribution is done over mutually attested secure channels (see enclave architecture).

← Enclave Architecture

Staking and Council Node →

Notes on Production Deployment

- See Tendermint notes on running in production and notes on setting up a validator
- Validators shouldn't be exposed directly to the internet
- RPC shouldn't be exposed directly to the internet (as it currently doesn't support rate-limiting, authentication...)
- Validator block signing should be via tmkms

Setting up Tendermint KMS for signing blocks (only for validators / council nodes)

Currently (tmkms v0.7), the system is still a bit Cosmos-centric, so the setup is slightly quirky.

Configuration

As per the example, create `~/.tmkms/tmkms.toml` (or any path) with something like:

```
[[chain]]
id = "<CHAIN_ID>"
key_format = { type = "hex" }

[[validator]]
addr = "unix:///<TMKMS_SOCKET_PATH>"
chain_id = "<CHAIN_ID>"

[[providers.<USED SIGNING PROVIDER>]]
chain_ids = ["<CHAIN_ID>"]
```

In `~/.tendermint/config/config.toml` (or wherever located), set the socket address to the same one as in `tmkms.toml` :

```
priv_validator_laddr = "unix:///<TMKMS_SOCKET_PATH>"
```

Obtaining the consensus / validator public key

Step 0. generate / initialize the keypair or seed

Step 0 depends on the signing provider -- e.g. for Ledger devices, one may need to enable *developer mode* in Ledger Live settings and install the Tendermint validator app.

Step 1. obtain the public key in the correct encoding

Depending on the signing provider, there may be a command to print out the public key. One other option is to run `tmkms start -c .tmkms/tmkms.toml -v` where `-v` is for verbose logging; the log may then include a message `... added consensus key <KEY_HEX_PAYLOAD>`.

The public key hex payload is Amino-encoded -- for the use in Tendermint configurations, there are two steps that need to be done:

1. strip off the Amino prefix; in the case of Ed25519 public keys, it's 5 bytes: 0x16, 0x24, 0xDE, 0x64, 0x20.
2. convert the stripped-off `KEY_HEX_PAYLOAD` to base64.

Step 2. place / use the public key where needed

For example, generate the `genesis.json` with it if the corresponding node is one of the initial validators / council nodes.

Step 3. start up everything

As before, but along with `tendermint node`, `tmkms start -c .tmkms/tmkms.toml` should also be launched.

← List of network parameters

Notes on Performance →

Staking and Council Node

Crypto.com Chain is based on Tendermint Core's consensus engine, it relies on a set of validators (Council Node) to participate in the proof of stake (PoS) consensus protocol, and they are responsible for committing new blocks in the blockchain.

Staked state / Account

StakedState is a data structure that holds state about staking:

```
pub struct StakedState {
    pub nonce: Nonce,    /// "from" operations counter
    pub bonded: Coin,    /// bonded amount used to determine voting power
    pub unbonded: Coin,    /// amount unbonded for future withdrawal
    pub unbonded_from: Timespec,    /// time when unbonded amount can be withdraw
    pub address: StakedStateAddress,    /// the address used (to check transactio
    pub validator: Option<Validator>,    /// validator metadata
    pub last_slash: Option<SlashRecord>,    /// record the last slash only for qu
}
```

TODO: should it have a minimum bonded amount?

Council Node

Council Node is a data structure that holds state about a node responsible for transaction validation:

```
pub struct CouncilNode {
    pub name: ValidatorName,    /// validator name / moniker (just for reference
    pub security_contact: ValidatorSecurityContact,    /// optional security@...
    pub consensus_pubkey: TendermintValidatorPubKey,    /// Tendermint consensus
    /// X.509 credential payload for MLS (https://tools.ietf.org/html/draft-ietf-
    /// (expected that attestation payload will be a part of the cert extension,
    pub confidential_init: ConfidentialInit,
}
```

EFFECTIVE_MIN_STAKE

Effective minimum stake (`EFFECTIVE_MIN_STAKE`) is defined as follows at any time:

1. if the number of validators has not reached `MAX_VALIDATORS` , it is `COUNCIL_NODE_MIN_STAKE` (the network parameter)
2. if the number of validators has reached `MAX_VALIDATORS` , it is equal to the lowest bonded amount of the staked states of the current validators plus 1.0 Coin.

Joining the network

Anyone who wishes to become a council node can submit a NodeJoinTx; this transaction is considered to be valid as long as:

1. The associated staking account has `bonded` amount >= `EFFECTIVE_MIN_STAKE` and is not punished;
2. There is no other validator with the same `staking_address` or the `consensus_pubkey`

Leaving the network

Anyone who wishes to leave the network, provided their associated staked state does not have any punishments, can submit UnbondStakeTx with the amount that will reduce the `bonded` amount to be lower than `COUNCIL_NODE_MIN_STAKE` .

Voting power and proposer selection

At the beginning of each round, a council node will be chosen deterministically to be the block proposer.

The chance of being a proposer is directly proportional to their *voting power* at that time, which, in general, is equal to the `bonded` amount (rounded to the whole unit) in the associated staking address of the council node.

The top `MAX_VALIDATORS` with the most `bonded` amount would put to the *active* validator set in `END_BLOCK` . If a validator's `bonded` amount is below the top `MAX_VALIDATORS` , It will be considered as *non-active* and would not be able to take part in the consensus. For example,

- If the number of *active* validators < `MAX_VALIDATORS` :

	`bonded` amount < `COUNCIL_NODE_MIN_STAKE`	`bonded` amount => `COUNCIL_NODE_MIN_STAKE`

	`bonded` amount < `COUNCIL_NODE_MIN_STAKE`	`bonded` amount => `COUNCIL_NODE_MIN_STAKE`
Validator's voting power	Set to 0	Set to the `bonded` amount

- On the other hand, If the number of *active* validators = `MAX_VALIDATORS` :

	`bonded` amount is below the top `MAX_VALIDATORS`	Otherwise
Validator's voting power	Set to 0	Set to the `bonded` amount

Removed validators / council nodes

A Validator is removed when its voting power is set to 0. This can happen in the following scenarios:

1. The Validator is being punished for infraction(s).
2. Its operator submitted UnbondStakeTx with sufficient stake to leave the network.
3. When the `bonded` amount <= `EFFECTIVE_MIN_STAKE`

When this happens, the following metadata is cleaned up as follows:

- its associated reward tracking-related data is cleared:
  - (a) immediately (*if the validator is being punished*)
  - (b) in the block that triggers next reward distribution (*otherwise*)
- its liveness tracking information is cleared in the block when this occurs
- its slashing related information (mapping Tendermint address / pubkey => staked state address) is scheduled to be cleared in a block after the current block time + `MAX_EVIDENCE_AGE` (`SLASH_MAP_DELETE`)

Note this inequality must be checked in network parameters: UNBOND_PERIOD >= JAIL_DURATION >= MAX_EVIDENCE_AGE

If the validator wishes to re-join the validator set, they can (unjail if necessary and) submit a NodeJoinTx (see the Joining the Network section).

If the NodeJoinTx transaction is valid AND this happens before `SLASH_MAP_DELETE` AND the consensus pubkey (or associated Tendermint address) from NodeJoinTx transaction is the same, the

previous slashing information is preserved (i.e. the "delete schedule" is cancelled).

If the NodeJoinTx transaction is valid AND this happens before next reward distribution AND the consensus pubkey from NodeJoinTx transaction is different, the associated staked state is then rewarded for interactions with both consensus pubkeys (as reward tracking is per staked state).

Validators / Council Nodes

Each `BeginBlock` contains two fields that will determine penalties:

- LastCommitInfo: this contains information which validators signed the last block
- ByzantineValidators: evidence of validators that acted maliciously

Their processing is the following:

```
FIXME: this looks out of date?

for each evidence in ByzantineValidators:
    if evidence.Timestamp >= BeginBlock.Timestamp - MAX_EVIDENCE_AGE:
        account = (... get corresponding account ...)
        if account.slashing_period.end is set and it's before BeginBlock.Timestam
            account.slashing_period.slashed_ratio = max(account.slashing_period.s
        else:
            if account.slashing_period.end is set and it's after BeginBlock.Times
                END/COMMIT_BLOCK_STATE_UPDATE(deduct(slashing_period, account))

            account.slashing_period = Some(SlashingPeriod(start = Some(BeginBlock
                end = Some(BeginBlock.Timestamp + SLASHING_PERIOD_DURATION, slash
            account.jailed_until = Some(BeginBlock.Timestamp + JAIL_DURATION)


for each signer in LastCommitInfo:
    council_node = (... lookup by signer / tendermint validator key ...)
    update(council_node, BLOCK_SIGNING_WINDOW)

for each council_node:
    missed_blocks = get_missed_signed_blocks(council_node)
    if missed_blocks / BLOCK_SIGNING_WINDOW >= MISSED_BLOCK_THRESHOLD:
        account = (... lookup corresponding account ...)
        if account.slashing_period.end is set and it's before BeginBlock.Timestam
            account.slashing_period.slashed_ratio = max(account.slashing_period.s
        else:
            if account.slashing_period.end is set and it's after BeginBlock.Times
```

```
                END/COMMIT_BLOCK_STATE_UPDATE(deduct(slashing_period, account))

            account.slashing_period = Some(SlashingPeriod(start = Some(BeginBlock
                end = Some(BeginBlock.Timestamp + SLASHING_PERIOD_DURATION, slash
            account.jailed_until = Some(BeginBlock.Timestamp + JAIL_DURATION)
```

"Global state" / APP_HASH

Tendermint expects a single compact value, `APP_HASH`, after each BlockCommit that represents the state of the application. In the early Chain prototype, this was constructed as a root of a Merkle tree from IDs of valid transactions.

In this staking scenario, some form of "chimeric ledger" needs to be employed, as staking-related functionality is represented with accounts. In Ethereum, Merkle Patricia Trees are used: (The alternative in Tendermint: depends on the order of transactions though)

They could possibly be used to represent an UTXO set: https://medium.com/codechain/codechains-merkleized-utxo-set-c76c9376fd4f

The overall "global state" then consists of the following items:

- UTXO set
- Account
- RewardsPool
- CouncilNode

So each component could possibly be represented as MPT and these MPTs would then be combined together to form a single `APP_HASH`.

END/COMMIT_BLOCK_STATE_UPDATE

FIXME: scheduling slashing cleanup

Besides committing all the relevant changes and computing the resulting `APP_HASH` in `BlockCommit`; for all changes in *Accounts*, the implementation needs to signal `ValidatorUpdate` in `EndBlock`.

For example, when the number of *active validators* is less then `MAX_VALIDATORS`:

- When the changes are relevant to the `bonded` amount of the council node's staking address and the validator is not jailed:

    - If the `bonded` amount changes and < `COUNCIL_NODE_MIN_STAKE`, then the validator's voting power should be set to 0;
    - If the `bonded` amount changes and >= `COUNCIL_NODE_MIN_STAKE`, then the validator's voting power should be set to that amount (rounded to the whole unit).

- When the changes are relevant to the jailing condition of the council node's staking address:
    - If the `jailed_until` changes to `Some(...)` (i.e. the node is being *jailed)*, then the validator's power should be set to 0;
    - If the `jailed_until` changes to `None` (i.e. the node was *un-jailed*) and `bonded` amount >= `COUNCIL_NODE_MIN_STAKE`, then the validator's power should be set to the `bonded` amount (rounded to the whole unit).

      It can be summarized in the following table:

	`bonded` < `COUNCIL_NODE_MIN_STAKE`	**`bonded` >= `COUNCIL_NODE_MIN_STAKE` but *Jailed***	**`bonded` `COUNCIL_` and *NOT J***
Validator's voting power	Set to 0	Set to 0	Set to the

On the other hand, If the number of the current *active* validators is equal to `MAX_VALIDATORS`, the validator's voting power will also be depended on whether its `bonded` amount is at the top `MAX_VALIDATORS`, please refer to the previous section

InitChain

The initial prototype's configuration will need to contain the following elements:

- Network parameters
- ERC20 snapshot state / holder mapping `initial_holders`: `Vec<(address: RedeemAddress, is_contract: bool, amount: Coin)>` (or `Map<RedeemAddress, (bool, Coin)>`)
- Network long-term incentive address: `nlt_address`
- Secondary distribution and launch incentive addresses: `dist_address1`, `dist_address2`
- Initial validators: `Vec<CouncilNode>`
- Bootstrap nodes / initially bonded: `Vec<RedeemAddress>`

The validation of the configuration is the following:

1. validate parameters format (e.g. CUSTOMER_ACQUIRER_SHARE + MERCHANT_ACQUIRER_SHARE + COUNCIL_NODE_SHARE = 1.0)
2. check that sum of amounts in `initial_holders` == MAX_COIN (network constant)
3. check there are no duplicate addresses (if Vec) in `initial_holders`
4. for each `council_node` in `Vec<CouncilNode>` : check `staking_address` is in `initial_holders` and the amount >= `COUNCIL_NODE_MIN_STAKE`
5. for each `address` in initially bonded `Vec<RedeemAddress>` : check `address` is `initial_holders` and the amount >= `COMMUNITY_NODE_MIN_STAKE`
6. size(Validators in InitChainRequest) == size(`Vec<CouncilNode>`) and for every CouncilNode, its `consensus_pubkey` appears in Validators and power in Validators corresponds to staking address' amount

If valid, the genesis state is then initialized as follows:

1. initialize RewardsPool's amount with amounts corresponding to: `nlt_address` + `dist_address1` + `dist_address2`
2. for every council node, create a `CouncilNode` structure
3. for every council node's staking address and every address in initially bonded: create `Account` where all of the corresponding amount is set as `bonded`
4. for every address in `initial_holders` except for `nlt_address` , `dist_address1` , `dist_address2` , council nodes' staking addresses, initially bonded addresses: if `is_contract` then add the amount to RewardsPool else create `Account` where the amount is all in `unbonded` and `unbonded_from` is set to genesis block's time (in InitChain)

← Transaction Privacy

Reward & Punishment →

Technical glossary

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

A

Authentication token

The access token for wallet related commands, It can be shown by cli-command.

B

Bonded (staking state)

The non-transferable amount of token staked to the corresponding address, it can be unlocked by the `unbond` transaction.

Byzantine Faults (Double signing)

A validator is said to make a byzantine fault when they sign conflicting messages/blocks at the same height and round.

C

Chain ID

A unique identifier for the blockchain. Different prefixes of the Chain ID are used to distinguish between *devnet*, *testnet* and *mainnet*. For example, the Chain ID of our testnet is `testnet-thaler-crypto-com-chain-42`

Chain-abci

The **A**pplication **B**lock**C**hain **I**nterface connects Tendermint (for consensus operations) to the actual applications.

Client-cli

The command line interface for the wallet client. It supports wallet management, funds transfer and basic staking operations.

Client-rpc

The JSONRPC interface of the wallet client. It can be used to be integrated with different services and also power the Sample Wallet. It provides nearly the same set of operations as Client-cli does.

D

Deposit (Transaction type: `deposit`)

The transaction that transfers funds from a transfer address to a staking address for staking purposes.

H

HD wallet

The BIP-32 compatible Hierarchical Deterministic (HD) wallet is a system of deriving keys from a single starting point known as a seed. The seed can be presented in the form of a human-readable mnemonic that can be used for easily backup and restore the wallet.

J

Jailing

A validator is jailed when they make a byzantine fault. When a validator gets jailed, they cannot perform any staking related operations relating to their account.

L

Liveness Faults

A validator is said to be non-live when they fail to sign at least `missed_block_threshold` blocks in last `block_signing_window` blocks successfully.

M

Mnemonic (of a wallet)

The BIP-39 compatible 24-word mnemonic phrase of the wallet. User can restore their wallet and associated addresses by this human-readable mnemonic.

Multisig addresses

The threshold multi-signature addresses that require multiple keys to authorize a transaction.

N

Network ID

The last two hex characters of the Chain ID. Using our testnet Chain ID `testnet-thaler-crypto-com-chain-42` as an example, the network ID would be `42` (in hex).

Node-join (Transaction Type: `node-join`)

The transaction for joining the network as a validator.

Nonce (of a staking address)

The nonce is the number of transactions that have the witness of the staking address. It also serves as an important safety precaution for preventing replay attacks.

P

Proposer (of a block)

Validator can become a proposer and commit new blocks in the blockchain according to a deterministic proposer selection procedure. The probability of a validator being selected as the proposer for a round is proportional to their voting power.

S

SGX

The Intel® Software Guard Extensions (SGX) is a set of instructions that increases the security of application code and data. It ensures the integrity and confidentiality of the data by isolating them from the OS or other enclaves.

Slashing

The penalty imposed on validators' misbehaviour such as byzantine faults or liveness faults, which resulted in losing some amount of their staked tokens.

Staking address

The address for staking related operations, it follows the format of the 20 bytes Ethereum account address.

State (of a staking address)

The general state of a staking address that includes *nonce*, *bounded/unbonded* amount, *validator metadata*, and *slashing related information* (if any).

T

TDBE

Transaction **d**ata **b**ootstrapping **e**nclave responsible for fetching current UTXO set transaction data; and handling periodic key generation operations.

Tendermint

The underneath byzantine fault tolerant protocol for performing distributed consensus.

Tendermint KMS(tmkms)

The key management system for tendermint validators.

TQE

Transaction **q**uery **e**nclave serves the encryption and decryption requests from wallets / clients. It allows semi-trusted client querying of sealed tx payloads.

Transfer (Transaction type: `transfer`)

The transaction that transfers funds between transfer addresses.

Transfer address

The address for payments/value transfers. Different prefixes of the transfer address are used to distinguish between devnet, testnet and mainnet.

U

Unbond (Transaction type: `unbond`)

The transaction to unbond funds in the staking address. Note that funds will only be available for withdrawal after the unbonding period has passed.

Unbonding period

The time duration of unbonding, which is the waiting period of the stake state's unbonded amount can be withdrawn.

Unjail (Transaction type: `unjail`)

The transaction to unjail a validator.

V

Validator

The participant in the proof of stake (PoS) consensus protocol. They are responsible for transaction validation and committing new blocks in the blockchain.

Validator keys

The key pair for signing messages from the validator. The full key pair is located under the tendermint `priv_validator_key.json` folder after the initialization. In production deployment, it is strongly advised not to keep the private key reside on the machine (see the deployment notes and recommendations)

View key

The key pair to enforce access to the confidential transaction data.

Voting power

The voting power is determined by the bonded amount in validator's staking address. The probability of a validator being selected as the proposer for a round is proportional to their voting power.

W

Withdraw (Transaction type : `withdraw`)

The transaction for withdrawing funds from a staking address to a transfer address.

← Threat Model

Threat Model

Threat modeling is a systematic approach to find potential threats by decomposing and enumerating system components. There are many different methodologies and/or frameworks when conducting threat modeling, such as STRIDE, DREAD, Attack Tree, etc. In our case, our Thread Model is based on STRIDE and Attack Tree.

STRIDE provides a set of security threats in five categories (the last one is ignored):

1. Spoofing - impersonating the identity of another
2. Tampering - data is changed by an attacker
3. Repudiation - an attacker refuses to confirm an action was conducted
4. Information Disclosure - exposing sensitive information
5. Denial of Service - degrade the availability or performance of the system
6. Elevation of Privilege (not applicable in our case)

For each category, we enumerate all the potential threats by breaking down a high-level goal into more specific sub-goals, in a way similar to attack tree enumeration. And in each sub-goal, we set the risk level by combining the `Severity` and `Exploitability` of the item.

Severity

- `5` : Severe impact on the whole system
- `4` : High impact on the whole system
- `3` : Moderate impact on the whole system or Severe impact on individual user/node
- `2` : High impact on individual user/node
- `1` : Moderate impact on individual user/node

Exploitability

- `5` : Existing exploit code available
- `4` : Relatively easy to exploit
- `3` : Attack is practical but not easy, a successful attack may require some special conditions
- `2` : Theoretically possible, but difficult in practice
- `1` : Very difficult to exploit
- `0.1` : Almost impossible

Assets

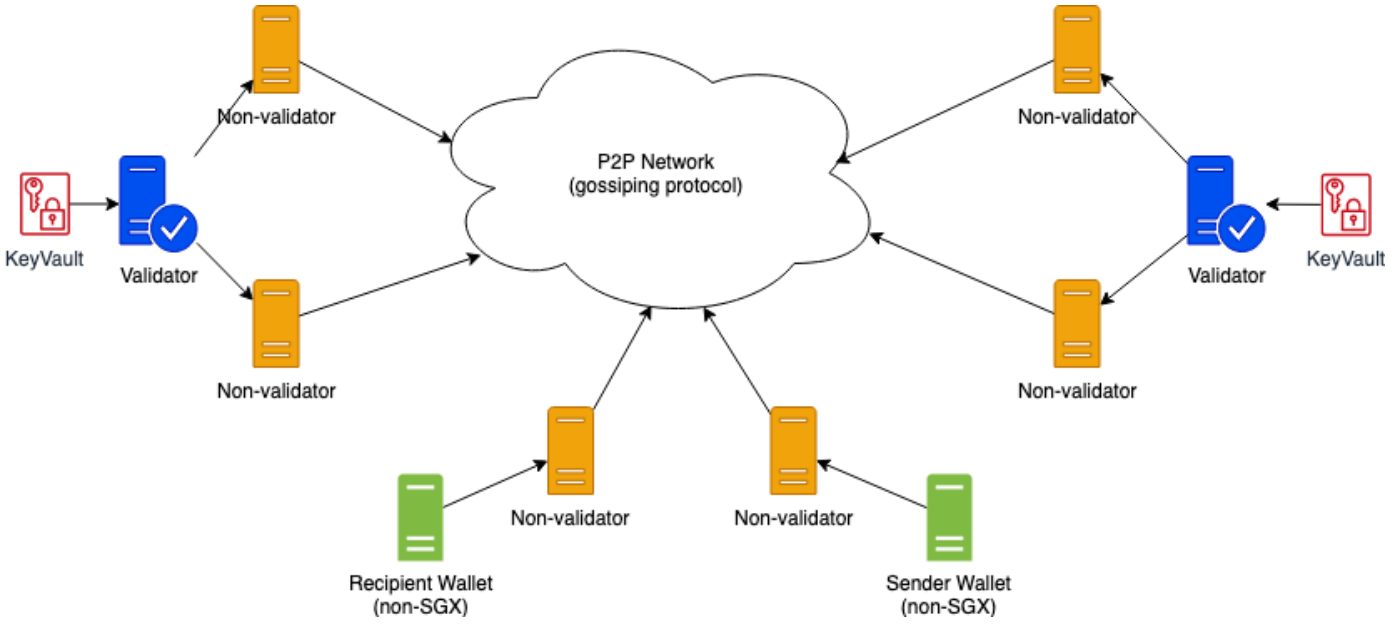
1. **The integrity of the account balance**: the most important piece of information in the blockchain.
2. **Validator secret keys**: one of the most powerful key, lost 1/3 of these keys will render the whole system to an unstable state.
3. **User secret keys**: key owner implies fund owner
4. **Transaction encryption keys**: transaction privacy of the system relies on the secrecy of this key

Scope

The whole Crypto.com Chain is a complex system and involves many different components. And therefore, the scope of this threat model is limited only to the major components of the system. To be more specific, the threat modeling of Tendermint and Intel SGX is **NOT** in the scope of this threat modeling.

We also assume standard security measures like OS level hardening, software patching, anti-virus, network firewall, physical security, etc, are properly implemented, executed and monitored. These mitigation strategies are not mentioned here.

System Diagram



Result

HERE

← Notes on Performance

Technical glossary →

Thaler Testnet: Running Nodes (Linux only)

The Crypto.com Chain Testnet has been named as **"Thaler"**.

This is an early tutorial for the developers and brave and patient super-early adopters.

Common Setup

Step 0. Install Intel SGX SDK 2.9 and other pre-requisites

- Make sure your CPU supports SGX, and it is enabled in BIOS. This GitHub repository has more information about supported hardware and cloud vendors.
- You can download the Linux SGX SDK installers from the Intel Open Source website. More details can be found in this installation guide.
- Note that some motherboards may have only "software controlled" option where an extra step is needed for enabling it.
- You may also need to install libzmq (e.g. libzmq3-dev package in Ubuntu 18.04).

> **NOTE**
>
> There is an Ubuntu-based Docker image `cryptocom/chain:latest` on Dockerhub that has Intel PSW and other dependencies pre-installed (you still need to have the SGX driver installed on the host and expose it to the container by running docker with the `--device /dev/isgx` flag).

Step 1. Get Tendermint and Chain v0.5.2 released binaries

Download the latest version of Tendermint 0.33.*. Chain v0.5.2 can be downloaded from GitHub.

> **CAUTION**

> Crypto.com Chain v0.5 is not backwards compatible with v0.3 nor v0.4 released earlier. So, if you were running a node with the old version of Crypto.com Chain, you will have to delete all the associated data.
>
> Also, please note the **released binary changes**.

Step 2. Configure Tendermint

- After placing all binaries on the path. You can initialize Tendermint with `tendermint init`. In `.tendermint/config/`, change the content of `genesis.json` to:

```json
{
  "app_hash": "F62DDB49D7EB8ED0883C735A0FB7DE7F2A3FA322FCD2AA832F452A62B38607D5",
  "app_state": {
    "council_nodes": {
      "0x6dbd5b8fe0dad494465aa7574defba711c184102": [
        "eastus_validator_1",
        "security@crypto.com",
        {
          "type": "tendermint/PubKeyEd25519",
          "value": "/SvfTeO4Du4oR/VYTjm7IgObc14zzddEAyFb4nU8E3Q="
        },
        {
          "cert": "ABCD"
        }
      ],
      "0x6fc1e3124a7ed07f3710378b68f7046c7300179d": [
        "canadacentral_validator_1",
        "security@crypto.com",
        {
          "type": "tendermint/PubKeyEd25519",
          "value": "QMegiWt9+5K1b1ZVd7zOJZxhTnbAtWzvGhViiElAlaw="
        },
        {
          "cert": "ABCD"
        }
      ],
      "0xb8c6886da09e12db8aebfc8108c67ce2ba086ac6": [
        "uksouth_validator_1",
        "security@crypto.com",
        {
          "type": "tendermint/PubKeyEd25519",
          "value": "tDLheZJwsA8oYEwarR6/X+zAmNKMLHTVkh/fvcLqcwA="
```

```
        },
        {
          "cert": "ABCD"
        }
      ]
    },
    "distribution": {
      "0x4ae85b35597fcb61c6c47b1fe0bdd7eed8421cdd": [
        "Bonded",
        "6000000000000000"
      ],
      "0x4b75f275dde0a8c8e70fb84243adc97a3afb78f2": [
        "UnbondedFromGenesis",
        "7946000000000000000"
      ],
      "0x4fd8162521f2e628adced7c1baa39384a08b4a3d": [
        "Bonded",
        "6000000000000000"
      ],
      "0x6c2be7846219eab3086a66f873558b73d8f4a0d4": [
        "Bonded",
        "6000000000000000"
      ],
      "0x6dbd5b8fe0dad494465aa7574defba711c184102": [
        "Bonded",
        "6000000000000000"
      ],
      "0x6fc1e3124a7ed07f3710378b68f7046c7300179d": [
        "Bonded",
        "6000000000000000"
      ],
      "0x9baa6de71cbc6274275eece4b1be15f545897f37": [
        "Bonded",
        "6000000000000000"
      ],
      "0xa9528abb92709370600d2cef41f1677374278337": [
        "Bonded",
        "6000000000000000"
      ],
      "0xb328a39002ede64c33bb60f1dc43f5df9eb47043": [
        "Bonded",
        "6000000000000000"
      ],
      "0xb8c6886da09e12db8aebfc8108c67ce2ba086ac6": [
        "Bonded",
```

```
        "6000000000000000"
      ]
    },
    "network_params": {
      "initial_fee_policy": {
        "coefficient": 1250,
        "constant": 1100
      },
      "jailing_config": {
        "block_signing_window": 720,
        "missed_block_threshold": 360
      },
      "max_validators": 50,
      "required_council_node_stake": "5000000000000000",
      "rewards_config": {
        "monetary_expansion_cap": "2000000000000000000",
        "monetary_expansion_decay": 999860,
        "monetary_expansion_r0": 350,
        "monetary_expansion_tau": 999999999999999999,
        "reward_period_seconds": 86400
      },
      "slashing_config": {
        "byzantine_slash_percent": "0.200",
        "liveness_slash_percent": "0.100"
      },
      "unbonding_period": 5400
    }
  },
  "chain_id": "testnet-thaler-crypto-com-chain-42",
  "consensus_params": {
    "block": {
      "max_bytes": "22020096",
      "max_gas": "-1",
      "time_iota_ms": "1000"
    },
    "evidence": {
      "max_age_duration": "5400000000000",
      "max_age_num_blocks": "200"
    },
    "validator": {
      "pub_key_types": ["ed25519"]
    }
  },
  "genesis_time": "2020-05-01T12:09:01.568951Z",
  "validators": [
```

```
    {
      "address": "FA7B721B5704DF98EF3ECD3796DDEF6AA2A80257",
      "name": "eastus_validator_1",
      "power": "60000000",
      "pub_key": {
        "type": "tendermint/PubKeyEd25519",
        "value": "/SvfTeO4Du4oR/VYTjm7IgObc14zzddEAyFb4nU8E3Q="
      }
    },
    {
      "address": "7570B2D23A4C7B638BEFE02EB4FC7927BFDED6B7",
      "name": "canadacentral_validator_1",
      "power": "60000000",
      "pub_key": {
        "type": "tendermint/PubKeyEd25519",
        "value": "QMegiWt9+5K1b1ZVd7zOJZxhTnbAtWzvGhViiElAlaw="
      }
    },
    {
      "address": "D527DAECDE0501CF2E785A8DC0D9F4A64760F0BB",
      "name": "uksouth_validator_1",
      "power": "60000000",
      "pub_key": {
        "type": "tendermint/PubKeyEd25519",
        "value": "tDLheZJwsA8oYEwarR6/X+zAmNKMLHTVkh/fvcLqcwA="
      }
    }
  ]
}
```


- For network configuration, in `.tendermint/config/config.toml`, you can put the following as `seeds` and `create_empty_blocks_interval`:

```
seeds = "f3806de90c43f5474c6de2b5edefb81b9011f51f@52.186.66.214:26656,29fab3b66
```

```
create_empty_blocks_interval = "60s"
```

> **NOTE**
>
> This page only shows the minimal setup.

> Depending on what you wish to test on the testnet, e.g. monitoring, you can refer to the Tendermint documentation for more details.

Step 3. Run everything

Before we move forward:

- Make sure `aesmd` service is running by

  ```
  ## check whether aesm service is on
  $ ps ax | grep aesm
  ## check whether driver is on
  $ ls /dev/isgx
  ```

  You may potentially start it up manually with `LD_LIBRARY_PATH=/opt/intel/libsgx-enclave-common/aesm /opt/intel/libsgx-enclave-common/aesm/aesm_service`. See also page 11 of the installation guide.

- The full node (exposed for wallets / clients) and the council node (validator) have slightly different steps described in the following sections.

A. Running a full node (to serve data to wallets / clients)

> **CAUTION**
>
> This page only shows the minimal setup.
>
> You may want to disable unsafe operations in Tendermint configuration, restrict the incoming connections RPC connections (e.g. over NGINX or equivalent), want to execute the processes using supervisor or equivalent etc.

Step 3-a-1. Obtain and set the service provider credentials for development

On the Intel's developer portal, you can obtain credentials for the non-production Intel Attestation Service and choose *"unlinkable quotes"*.

Once you obtained the credentials in the portal, set the following environment variables:

- `SPID` : Set it to the "Service Provider ID" value from the portal;
- `IAS_API_KEY` : Set it to the primary or secondary API key from the portal.

Step 3-a-2. Run everything

- Start the tx-query enclave app (in `tx-query-HW-debug/`), e.g.:

```
RUST_LOG=info ./tx-query-app 0.0.0.0:3322 ipc://$HOME/enclave.socket
```

- Start chain-abci, e.g.:

```
RUST_LOG=info ./chain-abci --chain_id testnet-thaler-crypto-com-chain-42 --gene
```

- Finally, start Tendermint:

```
tendermint node
```

B. Running a council node (validator)

> **CAUTION**
>
> This page only shows the minimal setup.
>
> You may want to run full nodes (see above) as sentries (see **Tendermint** and **local notes on production deployment**), restrict your validator connections to only connect to your full nodes, test secure storage of validator keys etc.

Step 3-b-0. (Optional) restoring a wallet

If you have participated in the v0.3.1 testnet and have backed up your seed phrase, you can restore it with the client-cli, for example:

```sh
$ client-cli wallet restore --name <WALLET_NAME>
```

You can then create a staking address with:

```
$ client-cli address new --name <WALLET_NAME> --type Staking
```

If the created address matches one of the ones listed in the initial *genesis.json* distribution, you can skip to Step 3-b-3.

Step 3-b-1. Create a staking address

This can be done, for example, with the client-cli command-line tool. Set the required environment variables:

1. `CRYPTO_CHAIN_ID=testnet-thaler-crypto-com-chain-42` ;
2. `CRYPTO_CLIENT_TENDERMINT=<YOUR_FULL_NODE>` .

> **Note for environment variables setting:**
>
> If you would like to connect to a local full node, you can put `<YOUR_FULL_NODE>=ws://localhost:26657/websocket` , or alternatively, you may use the external full node by setting `<YOUR_FULL_NODE>=ws://13.90.34.32:26657/websocket` .

And run the followings to create a new HD-wallet and staking address:

```sh
$ client-cli wallet new --name <WALLET_NAME> --type hd
$ client-cli address new --name <WALLET_NAME> --type Staking
```

You should obtain a hexadecimal-encoded address, e.g.
`0xa861a0869c02ab8b74c7cb4f450bcbeb1e472b9a`

Step 3-b-2. Obtain the minimal required stake

Unless you have obtained the CRO testnet token before, simply send a message on Gitter, stating who you are and your staking address (@devashishdxt or @lezzokafka would typically reply within a day).

Step 3-b-3. Create a validator key pair

- In a development mode, the full key pair is located in the `.tendetmint/config/priv_validator_key.json` ;

- If the file does not exist, you can initialize the tendermint root directory by running `tendermint init`. The public key should be base64-encoded, e.g. `R9/ktG1UifLZ6nMHNA/UZUaDiLAPWt+m9I4aujcAz44=`.

> **NOTE**
>
> If you plan to test a production setting with the Tendermint Key Management System (KMS) tool, please see production deployment notes on how it can be converted at the current (0.7) version.

Step 3-b-4. Run everything

- Start chain-abci, e.g.:

```
RUST_LOG=info ./chain-abci --chain_id testnet-thaler-crypto-com-chain-42 --gene
```

- Finally, start Tendermint:

```
tendermint node
```

Step 3-b-5. Send a council node join request transaction

As in Step 3-b-1, this can be done, for example, with `client-cli` with the required environment variables.

```sh
$ client-cli transaction new --name <WALLET_NAME> --type node-join
```

You will be required to insert the following:

- the staking address that holds your bonded funds;
- a moniker for your validator node; and
- a base64 encoded tendermint validator public key.

Thaler testnet block explorer and CRO faucet

- You are welcome to utilize the Crypto.com Chain Explorer to search and get more information on blocks and transactions on **Thaler** testnet network.
- To interact with the blockchain, simply use the CRO faucet to obtain test CRO tokens for performing transactions on the **Thaler** testnet.
    - Note that you will need to create a wallet, a transfer address and obtain the viewkey before using the faucet.

← Send Your First Transaction

Local Full Node Development →

Jellyfish Merkle Trie

> The ascii arts is copied from libra code comments.

Concept

```
type H256 = [u8; 32];
fn hash(value: Option<Value>) -> H256;
fn hash(left: H256, right: H256) -> H256;
```

We build the conceptual model first, then apply the optimizations on top of it to make it practical.

- It's a complete binary tree with 256 depth, the leaf node contains `Option<Value>`, `None` means value not exists.
- It's a binary trie which implements the map from `H256` to `Option<Value>`.
- It's a merkle tree, so all the internal nodes contains hashes of it's direct children.
  - Hash of leaf nodes: `hash(value)`.
  - `hash(None)` is a constant, it's called a placeholder hash.
  - Hash of internal nodes: `hash(left, right)`.

Every modification in leaf nodes should cause the root hash and all the internal nodes along the path to change.

> `hash(left, right) != hash(right, left)`
> This is important to fulfill above property.

All the nodes are stored in underlying key-value storage(usually on disk), and all the nodes committed into underlying storage are considered immutable, so we can query the old state, unless we decide to prune the states that is too old.

Merkle Proof

- Membership merkle proof for a key is the list of sibling hashes along the path from leaf node to the root node, combined with the key itself.
- Non-membership proof is just an membership proof lead to a `None` value.
- Range proof, TODO.

Optimization

- Flatten a 4 layer complete binary tree into array of 16 leaf nodes, this reduces the total numbers of nodes.

  The internal node only stores leaf nodes of this sub-tree, these leaf nodes might denote either a leaf node or an internal node in outer tree.

  The hash of the internal node is the root hash of this subtree, which is computed like this:

```
InternalNode[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F]

  4 ->              +------ root hash ------+
                    |                       |
  3 ->        +---- # ----+           +---- # ----+
              |           |           |           |
  2 ->        #           #           #           #
            /   \       /   \       /   \       /   \
  1 ->     #     #     #     #     #     #     #     #
          / \   / \   / \   / \   / \   / \   / \   / \
  0 ->   0   1 2   3 4   5 6   7 8   9 A   B C   D E   F
  ^
height
Note: # denotes the hash of its two direct children
```

  We also need treat it as a sub-tree when generating merkle proof.

  > **Nibbles**
  >
  > When traversing the internal nodes, we need to iterate the 32 bytes key in units of 4-bit nibbles like this:
  >
  > ```
  > 0x0123456789abcdef -> [0x0, 0x1, 0x2, 0x3, ...]
  > ```

- Replace a subtree which has no non-empty leaf node with a `None`.

  Replace a subtree which has single non-empty leaf node with the leaf node itself.

  It further shortens the average depth of path greatly in sparse case.

> The merkle patrica tree in ethereum further utilize extension node to shorten the path, we don't implement that to keep the logic simpler.

This rule applies to both the outer tree and the sub-tree inside the internal node.

For example (notice the path of `C` is not shortened because it references an internal node):

```
InternalNode[
  0 => Child(leaf),
  3 => Child(leaf),
  8 => Child(leaf),
  C => Child(internal),
  _ => None,
]

  4 ->              +------ root hash ------+
                    |                       |
  3 ->        +---- # ----+           +---- # ----+
              |           |           |           |
  2 ->        #           @           8           #
            /   \                               /   \
  1 ->     0     3                             #     @
                                              / \
  0 ->                                       C   @
  ^
height
Note: @ denotes placeholder hash.
```

- Identify nodes in underlying storage with `(version, nibble path)` rather than hash, the `version` is a monotonically increasing number.

  This helps to keep data compact in underlying storage, because the nodes are inserted in key order.

  > We use the block height as the version number, we'll use them interchangeably in this doc.

- We can compress the duplicates of placeholder hashes in the merkle proof.

  > Not implemented in this library yet, can be implemented outside of the library.
  > Effectiveness need further benchmarks.

Algorithm

Types

```
type Value = Vec<u8>;

struct NodeKey {
  version: Version,
  nibbles: Vec<Nibble>,
}
struct Leaf {
  key: H256,
  value: Value,
}
struct Internal {
  children: HashMap<Nibble, Child>,
}
struct Child {
  version: Version,
  hash: H256,
  is_leaf: bool,
}
enum Node {
    Null,  // Special branch only for empty tree, ignored for simplicity.
    Internal(Internal),
    Leaf(Leaf),
}
```

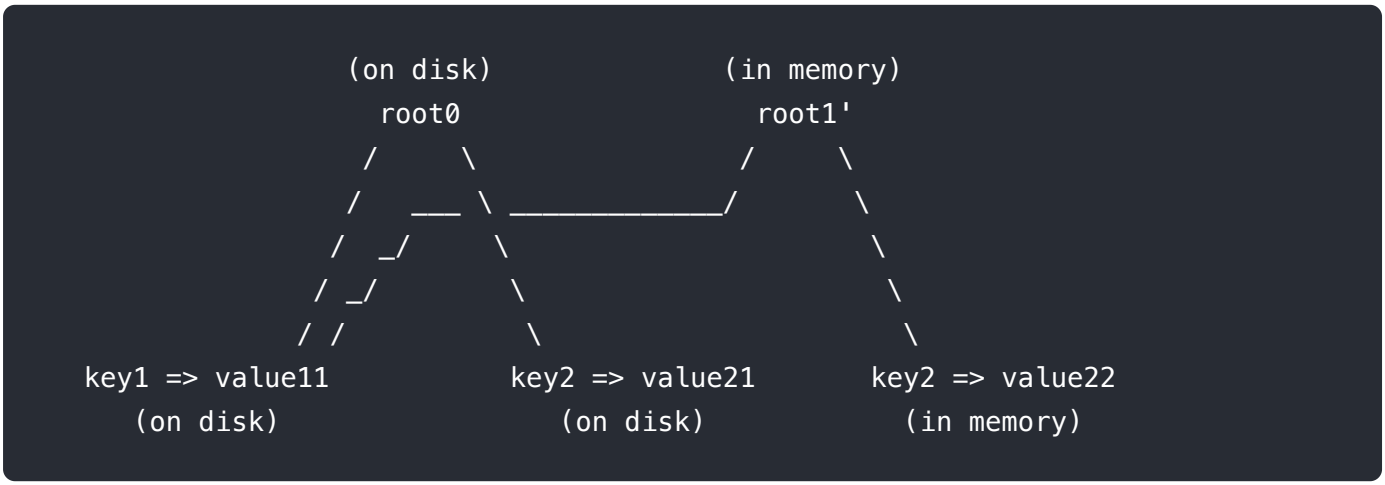
`get`

```
// proof generation is ignored.
fn get(key: H256, version: Version) -> Option<Value> {
  let mut next_key = NodeKey::new(version, vec![]);
  for i in 0..64 {
    match get_node(next_key) {
      Internal(node) => {
        let nibble = key.nibbles(i);  // get the i-eth nibble.
        if let Some(child) = node.children.get(nibble) {
          next_key = NodeKey::new(child.version, next_key.nibbles + nibble);
        } else {
          // not exists
          return None;
```

```
        }
      }
      Leaf(node) => {
        // nibble prefix is the same, but key maybe different.
        return if node.key == key {
          Some(node.value)
        } else {
          None
        };
      }
    }
  }
  None
}
```

`set`

Since the nodes on disk are immutable, so when we need to update a value, we create new nodes. For example, when we update `key2` 's value:



And to optimize batch operations, when a node is already in memory, it's replaced. For example, if we update `key1` 's value now, `root1'` is deleted and replaced by `root1''` :

```
  key1 => value11              key2 => value21  key1 => value12              key2 =
    (on disk)                    (on disk)       (in memory)                  (in
```

In the end, all the in memory nodes need to be persisted into the underlying storage.

We need a cache api to manage these in memory nodes:

```
fn cache_get(node_key) -> Node {
  if let Some(node) = cache.get(node_key) {
    node.clone()
  } else {
    // load from underlying storage
    get_node(node_key)
  }
}
fn cache_delete(node_key) {
  if cache.remove(node_key).is_none() {
    assert!(stale_nodes.insert(node_key), "Node gets stale twice unexpectedly.");
  }
}
fn cache_put(node_key, node) {
  assert!(cache.insert(node_key, node).is_none(),
    "Node gets inserted twice unexpectedly");
}
```

The actual `put` implementation:

```
fn put(key, value, version) {
  let root_key = NodeKey::new(version, vec![]);
  put_at(root_key, key, value, version, )
}

fn put_at(node_key, key, value, version, i_visited) -> (NodeKey, Node) {
  cache_delete(node_key);

  match get_node(node_key) {
    Internal(node) => {
      let visited = key.nibbles()[0 .. i_visited];
      let nibble = key.nibbles(i_visited);
      let new = if let Some(child) = node.children.get(nibble) {
        put_at(NodeKey::new(child.version, visited + nibble),
               key, value, version, i_visited + 1).1
```

```
      } else {
        let new = Node::new_leaf(key, value);
        cache_put(NodeKey::new(version, visited + nibble), new);
        new
      };
      node.children.insert(nibble, new);
      node_key.version = version;
      cache_put(node_key, node);
      (node_key, node)
    },
    Leaf(existing_leaf) => {
      cache_delete(node_key);

      let visited = key.nibbles()[0..i_visited];
      let remaining = key.nibbles()[i_visited..];
      let existing = existing_leaf.key.nibbles()[i_visited..];
      let (prefix, remaining, existing) = skip_common_prefix(remaining, existing)
      if remaining.is_empty() {
        assert!(existing.is_empty());
        // same key, update the leaf
        let new = Node::new_leaf(key, value);
        cache_put(NodeKey::new(version, node_key.nibbles), new);
        new
      } else {
        // create internal node.
        cache_put(NodeKey::new(version, visited + prefix + existing[0]), existing
        let new = Node::new_leaf(key, value);
        cache_put(NodeKey::new(version, visited + prefix + remaining[0]), new);

        let node = InternalNode::two_children(
          existing[0], existing_leaf.hash(),
          remaining[0], new.hash(),
        );
        cache_put(NodeKey::new(version, visited + prefix), node);
        while let Some(nibble) = prefix.pop() {
          node = InternalNode::one_child(nibble, node);
          node_key = NodeKey::new(version, visited + prefix);
          cache_put(node_key, node);
        }
        (node_key, node)
      }
    }
  }
}
```

Encoding

The encoding is simply concatenate the fields together.

- `H256` It is encoded as fixed length byte array, which is just bytes themselves, no length prefixed.
- `Vec<u8>`

  Variable length byte array is encoded as the bytes prefixed with a `u64` length.
- Numbers are encoded as little endian by default, unless indicated otherwise.

`Node`

- `tag: u8`

  `Null: 0, Internal: 1, Leaf: 2`
- Content of different branches, empty for `Null` branch.

`Internal`

- `existence_bitmap: u16` . The bit is set if the child exists.
- `leaf_bitmap: u16` . The bit is set if the child exists and is leaf node.
- `for i in existence_bitmap:`
  - `version: u64` , ULEB128

    > The high bit is set if more bytes follow

    ```
    for _ in 0..8 {
      let low_bits = num & 0x7f;
      num >>= 7;
      let more = num > 0;
      push(low_bits | more << 7);
      if !more {
        break;
      }
    }
    ```

- `hash: H256`

`Leaf`

- `key: H256`
- `hash: H256`
- `value: Vec<u8>`
  - `size: u32`, ULEB128, max `2^31`
  - bytes

`NodeKey`

The key used to identify node in key-value storage.

- `version: u64`, big endian.

  > Big endian is used here to keep the numeric order.

- `num_nibbles: u8`
- `nibble_bytes: bytes`

  One byte for two nibbles.

Implementation

Data structures in Punishment module

`PunishmentConfig`

`PunishmentConfig` holds the values for all the network parameters related to punishment. `PunishmentConfig` can be initialized from `genesis.json`.

```
/// Configuration for computing and executing validator punishments
pub struct PunishmentConfig {
    /// Number of blocks for which the liveness is calculated for uptime tracking
    block_signing_window: u16,
    /// Maximum number of blocks with faulty/missed validations allowed for an ac
    /// blocks before it gets jailed.
    missed_block_threshold: u16,
    /// Percentage of funds (bonded + unbonded) slashed when a validator is non-l
    liveness_slash_percent: Milli,
    /// Percentage of funds (bonded + unbonded) slashed when validator makes a by
    byzantine_slash_percent: Milli,
}
```

> **Note:**
>
> `UNBONDING_PERIOD` is a *global* network parameter and is not stored in `PunishmentConfig`.

`LivenessTracker`

`LivenessTracker` is used to track liveness of all the current active validators. At the beginning of each block, liveness of each active validator is updated using `update_liveness` function. Besides, `LivenessTracker` also exposes function to `add` and `remove` a validator from liveness tracking. Finally, `is_live` function can be used to check if a given validator is live or not.

```
/// Liveness tracker for validators
pub struct LivenessTracker {
    /// Number of blocks to use for calculating validator liveness (jailing param
```

```
    block_signing_window: u16,
    /// Number of blocks a validator can miss signing from last `block_signing_wi
    missed_block_threshold: u16,
    /// Holds data to measure liveness of a validator
    ///
    /// # Note
    ///
    /// - Size of this `BitVec` should be equal to `block_signing_window`.
    /// - Stores `true` at `index = height % block_signing_window`, if validator
    ///   otherwise.
    liveness: BTreeMap<TendermintValidatorAddress, BitVec>,
}

impl LivenessTracker {
    /// Adds a validator to liveness tracking
    ///
    /// # Note
    ///
    /// - Returns `Err` when a validator with `tendermint_validator_address` alre
    pub fn add_validator(&mut self, tendermint_validator_address: TendermintValid
        todo!()
    }

    /// Removes validator from liveness tracking
    ///
    /// # Note
    ///
    /// - Returns `Err` when validator with given address does not exist
    pub fn remove_validator(
        &mut self,
        tendermint_validator_address: &TendermintValidatorAddress,
    ) -> Result<()> {
        todo!()
    }

    /// Updates liveness of a validator with new block data
    ///
    /// # Note
    ///
    /// - Returns `Err` when validator with given address does not exist
    pub fn update_liveness(
        &mut self,
        tendermint_validator_address: &TendermintValidatorAddress,
        block_height: BlockHeight,
        signed: bool,
```

```
    ) -> Result<()> {
        todo!()
    }

    /// Checks if a validator is live or not
    ///
    /// # Note
    ///
    /// - Returns `Err` when validator with given address does not exist
    pub fn is_live(&self, tendermint_validator_address: &TendermintValidatorAddre
        todo!()
    }
}
```

Working of Punishment module (interactions with ABCI)

`InitChain`

- `PunishmentConfig` is initialized with values in `genesis.json`.
- `LivenessTracker` is initialized with all the council nodes in `genesis.json`.

`BeginBlock`

- Update `LivenessTracker` with all the signers/non-signers of last block (data can be obtained from `request.last_commit_info`). Note that the `request.last_commit_info` is not present for `block_height = 1`, so, `LivenessTracker` starts getting updated from `block_height = 2`. Also, `last_commit_info` may contain data for non-validators (validators which were recently removed from validator set), it is safe to ignore this data with a `warn` log.
- Obtain all the evidences of detected byzantine faults from `request.byzantine_validators`. For each evidence:
  - Check if the evidence is valid, i.e., `evidence.height + UNBONDING_PERIOD >= current_block_height`.
  - Get address of faulty validator from `evidence.validator.address` (if present). Ignore the evidence if faulty validator's information is not present in the evidence. Raw address bytes can be converted to `TendermintValidatorAddress` using `TendermintValidatorAddress::try_from(validator.address.as_slice())`.
  - If the validator is not already jailed (ignore if the validator is already jailed):
    - Jail and slash the validator (set `jailed_until = current_block_time + UNBONDIND_PERIOD` and slash by `BYZANTINE_SLASH_PERCENT`). Note that, both, slashing

and jailing should happen as one command, i.e., validator account's `nonce` will only increase by one.
    - Remove the validator from current validator set, i.e., set their voting power to zero. Validator will get removed from `LivenessTracker` in `EndBlock`.
    - Add validator address to a list of permanently banned validators.
    - Generate slashing and jailing events for validator.

> **Note:**
>
> An additional validation for `NodeJoinTx` will be to verify if the validator address is not present in the list of permanently banned validators.

`EndBlock`

- Update `LivenessTracker`, i.e., add all the validators who joined in current block and remove all the validator who, left/got jailed in current block.
- Set `response.validator_updates` for all the validators whose voting power was changed. Tendermint will remove all the validators whose voting power is set to zero, so, all the jailed validators should have zero voting power.

Transaction

Transaction Identifier

Each transaction has an identifier (typically shortened as TX ID). It is defined as

blake3_hash(SCALE-encoded transaction binary data)

See serialization for more details about the transaction binary format.

Witness

See signature-schemes for more details

Textual Address Representation

Crypto.com Chain supports threshold / multi-signature addresses that are represented as a single hash (see signature-schemes) which is different from Ethereum.

To represent the underlying byte array in a textual form, Bech32 is used. The convention for the human-readable prefix is the following:

- `cro` : mainnet payment
- `tcro` : testnet payment
- `dcro` : local devnet/regtest payment
- staking addresses (see accounting) are textually represented in hexadecimal encoding to match the initial Ethereum ones

Transaction Fees

The purpose of transaction fees in the initial prototype is an anti-spam measure, i.e. to prevent broadcasting valid transactions indefinitely. The general scheme is:

- If the transaction type allows indefinite valid transactions in an immediate time span (e.g. "transfer"), a fee (calculated as below) must be paid -- i.e. each transaction should be validated that it included this sufficient fee. Note that the fee amount must be exactly equal to the

computed one (this is to prevent errors where a much larger fee could be accidentally paid if the fee amount could be set arbitrarily large).
- If the transaction type allows a limited number of valid transactions, there is no fee. One example is "unjail" where only one valid transaction can be sent for a given state after the unjailing period.

More details can be found in the transactions document.

(In the future, if necessary, a dynamic Dutch auction-style fee market may be introduced for congestion control -- this may, however, be outside of the consensus state machine, i.e. "off-chain", in order to incentivize full nodes or other network layers.)

Fee Calculation

The initial prototype uses a linear fee system. The minimal transaction fee is defined according to the formula:

```
<BASE_AMOUNT> + <PER_BYTE> * size
```

`BASE_AMOUNT` and `PER_BYTE` are special network parameters in a fraction of CRO. `size` is the serialized transaction data's size in bytes.

To verify a basic transaction one would need to check:

```
sum(inputs amounts) or account.unbonded/bonded == sum(outputs amounts) + fee
```

The transaction fee goes to the rewards pool to reward the validations.

Transaction Types

Basic Types (plain version):

> **NOTE**
>
> All these types should also contain metadata, such as network ID. Furthermore, some of these transactions will be obfuscated to provide privacy protections to the users.

Tx type	Inputs	Outputs	Fees involved?	Obfuscated?
`TransferTx`	UTXOs	UTXOs	Yes	Yes
`DepositStakeTx`	UTXOs	Deposit the bonded amount to the specified account	Yes	Yes
`WithdrawUnbondedTx`	Nonce, account	UTXOs	Yes	Yes
`UnbondStakeTx`	Nonce, amount, account	Moves funds from `bonded` to `unbonded` under the same account with timelock	Yes	No

Please also refer to this flowchart that shows different types of transaction and how they interact with each other.

Advanced Types:

Besides the above-mentioned basic transactions, there are some advanced types of transactions related to the council node and service node state metadata management, for example:

- `UnjailTx` : This transaction can be broadcasted to un-jail a node. It takes *nonce*, *account* and has to be signed by the account's corresponding key.
- `NodeJoinTx` : Anyone who wishes to become a council node can broadcast this transaction. It takes *council node data*, *staking address* and has to be signed by the node's staking key. For further details on the process of joining the Crypto.com chain as a validator, please refer to this documentation.

> **Note**
>
> There will be no transaction fee for advanced types Tx in the initial prototype.

Cross-currency transactions and settlements

A proof of concept on the cross-currency transfers and settlement on CRO can be found in this repository. It demonstrates how to configure Interledger nodes for performing CRO-ETH cross-currency transactions between the Ethereum network (testnet or mainnet) and the CRO devnet.

← Transaction Accounting Model

Serialization →

Reward

Abstract

Rewarding module enable crypto chain to reward protocol participators, the design should meet following requirements:

- The total supply of CRO is fixed.
- The distribution should be "fair", i.e. all honest participators should get rewards, non-participators should not, and the amount of rewards get distributed to a validator should be proportional to it's contribution (sum of the voting power of each vote).
- The newly emitted coins for rewarding is adjusted according to total staking and current block time.

The basic procedure is:

- Record participators for each block, sum the voting powers for each participator at the vote time.
- Distribute the accumulated reward pool according to the recorded sum powers at the end of reward period.

For example, in one reward period with N blocks, validator `A` only participate in the first block, and all the other validators participate in all blocks, assuming the validator set is static during this period, then we have:

```
p: voting power of validator A
p': sum of voting power of all other validators

block1: participators: p', p
block2: participators: p'
block3: participators: p'
...
blockN: participators: p'

# The rewards of A is:
reward_a = reward_pool * p / (p + p' * N)
```

The result of reward distribution should be written into events of block results.

Concepts

Reward participator

All the validators who follow the protocol honestly should participate in the reward distribution.

In tendermint's terms, following the protocol honestly means being a signed voter in `last_commit_info` of `BeginBlockRequest`.

It can be described precisely as:

```
begin_block_req
  .last_commit_info
  .votes
  .filter(|vote| vote.signed_last_block)
```

Reward period

Rewards are distributed to the validators periodically, the length of the period is configured in network parameter `reward_period_seconds` in genesis config.

Reward pool

During the reward period, the following sources of funds are accumulated into reward pool:

- Monetary expansion
- Transaction Fees
- Slashings

The reward pool gets accumulated throughout the whole reward period; it gets distributed once at the end of each reward period.

Due to fixed point calculations, there may be a few remainders from arithmetic operations after distribution. These remainder amounts should be left in the rewards pool for the next period.

If there is zero participant for the period, the rewards pool should be left to the next period, rather than get burned.

Total staking

The sum of bonded coins of all active validators at some block.

Monetary expansion

New coins are minted into the reward pool at the end of reward period.

The amount is calculated based on the total staking of current state and current block time.

The total minted coins for rewarding shouldn't exceed the amount configured in network parameter `monetary_expansion_cap`.

The calculation is as follows:

```py
R0 = network_parameter["monetary_expansion_r0"]  # upper bound for the reward rat
P = network_parameter["reward_period_seconds"]
tau = current tau  # refer to section "Decay of parameter tau"
S = total staking  # refer to section "Total staking"
Y = 365 * 24 * 60 * 60  # seconds of a year

CAP = network_parameter["monetary_expansion_cap"]
minted = total minted coins so far  # recorded in state

R = (R0 / 1000) * exp(-S / tau)
N = floor(S * (pow(1 + R, P / Y) - 1))
N = N - N % 10000  # less precision requirement
result = min(N, CAP - minted)  # the amount of newly minted coins
```

Decay of parameter tau

The parameter `tau` in the monetary expansion calculation should be initialized to network parameter `monetary_expansion_tau` and gets decayed for each reward distribution.

The rate of decaying is configured in network parameter `monetary_expansion_decay`.

The calculation is like this:

```py
rate = network_parameter["monetary_expansion_decay"]

# init
tau = network_parameter["monetary_expansion_tau"]

# decay
tau = tau * rate / 1000
```

To prevent overflow of integer multiplication, it can be transformed into:

```py
tau = (tau / 1000) * rate + (tau % 1000) * rate
```

Fixed point arithmetic

We should use continued fraction method to compute the `exp` and `pow` with fixed point arithmetics.

First we transform the power function into exponential and natural logarithm functions:

```
pow(x, y) = exp(y * log(x))
```

The `exp` and `log` are computed with continued fractions representation, using a form with better convergence:

```
exp2(x, y) = exp(x / y)
log2(x, y) = log(1 + x / y)
```

$$e^{\frac{x}{y}} = 1 + \cfrac{2x}{2y - x + \cfrac{x^2}{6y + \cfrac{x^2}{10y + \cfrac{x^2}{14y + \ddots}}}}$$

$$\ln\left(1+\frac{x}{y}\right) = \cfrac{x}{y+\cfrac{1x}{2+\cfrac{1x}{3y+\cfrac{2x}{2+\cfrac{2x}{5y+\cfrac{3x}{2+\ddots}}}}}}$$

$$= \cfrac{2x}{2y+x-\cfrac{(1x)^2}{3(2y+x)-\cfrac{(2x)^2}{5(2y+x)-\cfrac{(3x)^2}{7(2y+x)-\ddots}}}}$$

With above substitutes, we can transform the formula like this:

```
R = (R0 / 1000) * exp(-S/tau)
  = R0 * exp2(-S, tau) / 1000
N = S * (pow(1 + R, P / Y) - 1)
  = S * (exp(P * log(1 + R) / Y) - 1)
  = S * (exp2(P * log(1 + R), Y) - 1)
  = S * (exp2(P * log(1 + R0 * exp2(-S, tau) / 1000), Y) - 1)
  = S * (exp2(P * log2(R0 * exp2(-S, tau), 1000), Y) - 1)
```

Break it down into simpler computation steps:

```
# To keep the intermediate numbers smaller
S' = S / 10000000_00000000
tau' = tau / 10000000_00000000

n0 = exp2(-S', tau')
n1 = log2(R0 * n0, 1000)
n2 = exp2(P * n1, Y)
n3 = floor(S * (n2 - 1))
N  = n3 - n3 % 10000
```

> Fixed point number format: `I65F63`.

> `exp2` runs 25 iterations.
> `log2` runs 10 iterations.

TODO, how to compute the continued fractions form of `exp2` and `log2`.

Implementation

Network parameters

```
pub struct RewardsParameters {
    /// Reward period in seconds
    pub reward_period_seconds: u64,

    /// Maximum minted coins for rewards
    pub monetary_expansion_cap: Coin,
    /// Monetary expansion formula parameter R0
    pub monetary_expansion_r0: Milli,
    /// Monetary expansion formula parameter tau
    pub monetary_expansion_tau: u64,

    /// Decay rate of the parameter tau
    pub monetary_expansion_decay: u64,
}
```

Validation of `RewardsParameters`:

```
impl RewardsParameters {
    pub fn validate(&self) -> Result<(), &'static str> {
        if self.monetary_expansion_r0 > Milli::integral(1) {
            return Err("R0 can't > 1");
        }
        if self.monetary_expansion_tau == 0 {
            return Err("tau can't == 0");
        }
        if self.reward_period_seconds > 365 * 86400 {
            return Err("reward period can't exceed 365 days");
        }
        if self.monetary_expansion_decay > 1_000_000 {
            return Err("decay can't > 1_000_000");
        }
        Ok(())
```

```
    }
}
```

State

```
/// Participate in the computation of app_hash, empty block should not modify
pub struct RewardState {
    /// Fees and slashings accumulated through current reward period
    pub period_bonus: Coin,
    /// last block height that updated it
    pub last_block_height: BlockHeight,
    /// last reward distribution time, default to the genesis time
    pub last_distribution_time: Timespec,
    /// Record how many coins have been minted so far, shouldn't exceed the monet
    pub minted: Coin,
    /// Current tau value of monetary expansion calculation
    pub current_tau: u64,
}

pub struct ValidatorState {
    ...

    /// Records the participators for reward distribution, the value is the sum o
    reward_participators: HashMap<StakingAddress, u64>,
}

pub struct ChainNodeApp {
    pub reward_state_updated: bool
}
```

Use `u64` for the sum of voting powers is enough, because total staking powers is smaller than 10 billion, so if maximum staking powers participates in all the blocks, the number of blocks we can support before `u64` overflows is:

```
>>> 9223372036854775807 / 10000000000
922337203
```

It should be more than enough for one reward period, but still the implementation should avoid panic or wrap-around when overflows happen in extreme case.

Predefined symbols

In the following sections, we use following predefined symbols:

- `reward_state` for the current `RewardState`
- `validator_state` for the current `ValidatorState`
- `node_state` for the current `ChainNodeApp`
- `network_parameter` for the current `NetworkParameter`

ABCI Events

Begin block

The main actions all happens in begin block event, it's the time we get reports of participators through `last_commit_info`, also the time the node state is consistent.

1. Collect the validator addresses of reward participators from `last_commit_info`.
2. Record them, and accumulate their voting powers.
3. If a reward period has passed, distribute the reward pool according to the participator statistics.

`begin_block`

```
fn begin_block(req: RequestBeginBlock) {
    let addrs = req.last_commit_info.votes
        .filter(|vote| vote.signed_last_block)
        .map(|vote| get_validator_address(vote));

    for addr in addrs {
        // Panic or not is decided by the implementator
        let staking_addr = find_staking_address(addr).unwrap();
        // Panic or not is decided by the implementator
        let power = get_voting_power(staking_addr).unwrap();
        let amount = u64::from(power);
        if validator_state.reward_participators.contains(staking_addr) {
            // Use saturating_add for better robustness
            validator_state.reward_participators[staking_addr] += amount;
        } else {
            validator_state.reward_participators[staking_addr] = amount;
        }
    }

    // Use saturating_add for better robustness
    if reward_state.last_distribution_time + network_parameter.reward_period_seco
```

```
        > node_state.block_time {
        // Don't distribute rewards if reward period not reached yet
        return None;
    }
    reward_state.last_distribution_time = node_state.block_time;

    // Mark the dirty flag as soon as reward period reached.
    node_state.rewards_pool_updated = true;

    let minted = min(
      // Use saturating_sub to prevent underflow
      network_parameter.monetary_expansion_cap - reward_state.minted,
      // Refer to the following sections for the implementation
      monetary_expansion(
          // Refer to the following sections for the implementation
          total_staking(),
          reward_state.current_tau,
          network_parameter.monetary_expansion_r0,
          network_parameter.reward_period_seconds,
      )
    );
    reward_state.minted += minted;

    // Decay the tau parameter, use the overflow avoid method mentioned above
    reward_state.current_tau = mul_micro(
        reward_state.current_tau,
        network_parameter.monetary_expansion_decay,
    );

    // Total reward pool for this period
    let reward_pool = reward_state.period_bonus + minted;

    // Use saturating_add for better robustness
    let sum_power = validator_state.reward_participators.values().sum();

    // Compute the rewards distribution,
    // compute `reward_pool * power / sum_power` for each participator
    let distribution = vec![];
    if sum_power == 0 {
        // If there is zero participator, the newly minted coins should be left t
        reward_state.period_bonus += minted;
    } else {
        // Remaining coins after distribution
        let remains = reward_pool;
```

```
        for (addr, power) in validator_state.reward_participators {
            // Use some type bigger than u64 (e.g. u128) to prevent overflow of m
            let amount = reward_pool * power / sum_power;
            remains -= amount;
            distribution.push((addr, amount));
        }

        // Remains of distribution is left to the next period
        reward_state.period_bonus = remains;
    }

    // Add the rewards to the accounts, implementation is in following sections
    add_rewards(distribution);

    // Clear rewards statistics
    validator_state.reward_participators.clear();

    // The Result (distribution, minted) should be written into block result even
}
```

`monetary_expansion`

TODO

`total_staking`

```
fn total_staking() {
    let sum = 0;
    for staking_addr in validator_state.validators {
        // Panic or not is decided by implementator
        let account = get_account(staking_addr).unwrap();
        sum += account.bonded;
    }
    sum
}
```

`add_rewards`

```
// accounts should be stored in merkle trie, the details are ignored here.
fn add_rewards(dist) {
    for staking_addr, amount in dist {
        // Panic or not is decided by implementator
        let account = get_account(staking_addr).unwrap();
```

```
        account.bonded += amount;
    }
}
```

Commit

Update `last_block_height` if `RewardsPoolState` changed:

```
if node_state.reward_state_updated {
    reward_state.last_block_height = current_block_height;
    node_state.reward_state_updated = false;
}
```

Hooks

When validator becomes inactive at `end_block`

When validator get jailed or unbonded enough to become inactive during a reward period, it's reward participation record should be removed, so it won't receive any rewards.

```
validator_state.reward_participators.remove(staking_addr)
```

Slashing executed

When slashing executed, the `slashed_amount` needs to be added to `period_bonus` of `RewardState`.

```
reward_state.period_bonus += slashed_amount;
node_state.rewards_pool_updated = true;
```

Fee collected

When transaction fees are collected, the amount needs to be added to `period_bonus` of `RewardState`.

```
reward_state.period_bonus += fees;
node_state.rewards_pool_updated = true;
```

Staking states and transitions

```
struct StakedState {
    address: StakedStateAddress,
    nonce: u64,
    bonded: Coin,
    unbonded: Coin,
    unbonded_from: Timespec,
    validator: Option<Validator>,
}

struct Validator {
    council_node: CouncilNode,
    jailed_until: Option<Timespec>,
    inactive_time: Option<Timespec>,
    used_validator_keys: Vec<(TendermintValidatorPubKey, Timespec)>,
}
```

States

Clean staking

`validator.is_none()`

Validator

`validator.is_some()`

There are several variants of it:

Active

`validator.inactive_time.is_none()`

> ***NOTE:*** Active validator doesn't necessarily mean the final validator take effect in tendermint, please refer to Choose final validators

Inactive

`validator.inactive_time.is_some()`

Jailed

`validator.jailed_until.is_some()`

> ***NOTE:*** Jailed implies inactive, but not vice versa

State transitions

From "clean staking" or "inactive(unjailed) validator" to active validator

Node join

The only way to transit to active validator is by executing `NodeJoinTx`, the preconditions are:

- `bonded >= min_required_staking`
- The validator pubkey/address is not already used by others, it's ok to re-use the old keys used by itself if it's a re-join from an inactive validator.
- Not jailed if transiting from inactive validator

From "active validator" to "inactive validator"

There are several cases for this:

Bonded coins become lower than required

When `bonded < min_required_staking`, this transition happens.

The reasons for dropping of bonded coins maybe:

- Execute `UnbondTx` at `deliver_tx` event
- Slashed for non-live or byzantine faults at `begin_block` event

> ***NOTE:*** The transition happens immediately in `deliver_tx` or `begin_block` events, won't reverse automatically when bonded coins become enough again even in the same block, so the activeness is always well-defined during the whole process.

Jailed for byzantine faults

Jailed always implies inactive.

This happens in `begin_block` event.

From "jailed validator" to "inactive(unjailed) validator"

Unjail

The only way to leave jailed validator state is by executing `UnjailTx`, the preconditions are:

- Already jailed
- `block_time >= jailed_until`

From "inactive validator" to "clean staking"

Clean up

The clean up procedure will remove the validator record if:

- Not jailed
- `block_time >= inactive_time + cleanup_period`

> *NOTE:* `cleanup_period`
> Currently `cleanup_period = unbonding_period`, but logically, `cleanup_period` only needs such constraints:
>
> - `> max_evidence_age`, so we can handle delayed byzantine evidences (inactive validator can still be slashed for later detected byzantine faults)
> - `> 2 blocks`, so we don't panic when seeing signing vote of inactivated validators

Appendix

Choose final validators

The final validator set that take effect in tendermint is chosen at `end_block` event by:

- Sort all the active validators by `voting_power desc, staking_address`
- Take the first `max_validators` ones

The abci protocol of `end_block` event expect validator set updates in response, so we need to diff the new set against the current set to get the updates.

For example, assuming `max_validators = 3`, if you are the fourth active validator, so you are not chosen yet, but in the future if any validator in the top 3 quit, you will be chosen automatically at the next `end_block` event:

```yaml
max_validators = 3

genesis:
  validators (map of validator address to voting power):
  - addr1 -> 10
  - addr2 -> 9

block1:
  deliver_tx
  - join_node(addr3, 8)
  - join_node(addr4, 7)
  active validators:
  - addr1 -> 10
  - addr2 -> 9
  - addr3 -> 8
  - addr4 -> 7
  end_block validator updates:
  - addr3 -> 8

block2:
  deliver_tx:
  - unbond_all(addr1)
  active validators:
  - addr2 -> 9
  - addr3 -> 8
  - addr4 -> 7
  end_block validator updates:
  - addr1 -> 0
  - addr4 -> 7
```

Implications of jailing

Transactions

Only `UnjailTx` is allowed to be executed on a jailed staking if the `jailed_until` time is passed.

Disallowed transactions are:

- `DepositTx`

- `WithdrawTx`
- `UnbondTx`
- `NodeJoinTx`

Reward distribution

It won't distribute rewards to jailed validators, inactive(unjailed) validators will get the rewards as normal.

When a validator is jailed, it's reward participation tracking records are removed immediately.

Process byzantine faults

Jailed validators won't be slashed again for byzantine faults detected in jailing period.

Nonce

The nonce is the number of transactions that have the witness of the staking address, which includes:

- `WithdrawTx`
- `UnbondTx`
- `UnjailTx`
- `NodeJoinTx`

Liveness tracking

- All active validators's liveness trackers are maintained no matter if it's chosen into the final validator set.

  If it doesn't appear in the votes reported by tendermint, it's recorded as a `true` which means live by default.

- Inactive validator's liveness trackers are also maintained, and recorded as a `true` for each block, this serves two purposes:

  - After a validator inactivated, the signing vote might still arrive for the next two blocks, we don't want to issue a false warning in this case.
  - Validator might quit and re-join very fast (by `UnbondTx` / `DepositTx` / `NodeJoinTx`), in this case it's liveness tracking record is preserved.

It means the liveness tracker is only removed when validator record get cleaned up.

Inactive validator re-join with different validator key

When an inactive validator re-join, it can provide different validator key, but it still needs to be hold responsible for byzantine fault committed before for as long as `max_evidence_age`. So we need to keep the old validator keys for sometime.

Whenever validator change consensus key, the old key and current block time are pushed into `used_validator_keys`, before that, the used keys older than `max_evidence_age` are removed.

There is a maximum bound (`max_used_validator_keys`) on the size of `used_validator_keys` to prevent attack. After the maximum bound reached, re-join with new validator key is not allowed.

Non exists and empty staking

Empty staking is defined as:

```
StakedState {
    address,
    nonce: 0,
    bonded: 0,
    unbonded: 0,
    unbonded_from: 0,
    validator: None,
}
```

For all the logic processing, the result of success execution should be the same for both non exists staking and empty staking.

The error message for failed execution maybe different, for example `WithdrawTx` might report `StakingNotExists` on non exists staking, but `CoinError` on empty staking.

So the implementor should be free to choose either semantics.

Transaction processing

> *Caution:* Some details are subject to changes.

The `check_tx` and `deliver_tx` events share the same transaction processing code, the only difference is they operate on different storage buffers (refer to buffered-storage).

In `commit` event, the `deliver_tx` buffer is flushed, while the `check_tx` buffer is dropped.

Transaction types

```
Enclave
  Transfer
  Deposit
  Withdraw
Public
  Unbond
  Unjail
  NodeJoin
```

The enclave transactions are obfuscated in enclave, so they can only be verified completely in the validation enclave, while the public transactions don't need enclave to verify.

The basic process procedure is like this:

```
let context = ...
match tx {
    Enclave(tx) => {
        let action = validate_enclave_tx(storage, context, tx)?;
        execute_enclave_tx(storage, tx, action)?;
    },
    Public(tx) => {
        process_public_tx(storage, context, tx)?;
    }
}
```

Context

```
struct ChainInfo {
    /// minimal fee computed based on tx size expected to be paid
    pub min_fee_computed: Fee,
    /// network hexamedical ID
    pub chain_hex_id: u8,
    /// block time of current processing block
    pub block_time: Timespec,
    /// height of current processing block
    pub block_height: BlockHeight,
    /// related network parameters, used by validation of withdraw tx
    pub unbonding_period: u32,
}
```

Enclave transactions

`validate_enclave_tx`

- Check inputs unspent and load sealed log for them, and pass to validation enclave
- Load staking state for withdraw and deposit transactions, and pass to validation enclave
- Validation enclave decrypt the transaction, unseal the inputs and do the following validations:
  1. Check tx attributes (`chain_hex_id` , `app_version`)
  2. Check inputs/outputs basic
     - Not empty
     - No duplicates
  3. Check input timelock
  4. Verify inputs witnesses for transfer/deposit
  5. Verify staking address witness for withdraw
  6. Check staking state not jailed for withdraw/deposit
  7. Check staking state for withdraw:
     - `nonce` match
     - `context.block_time >= unbonded_from`

- `unbonded != 0`
- `outputs.iter().all(|x| x.valid_from == account.unbonded_from)`

8. Check fee and input/output sum amounts:

   - Transfer

     `sum_input - sum_output == min_fee`

     Return `sum_input - sum_output` as paid fee

   - Deposit

     `sum_input > min_fee`

     Return `sum_input - min_fee` as deposit amount

   - Withdraw

     `account.unbonded - sum_output == min_fee`

     Return `account.unbonded - sum_output` as paid fee

`execute_enclave_tx`

The `TxEnclaveAction` is returned as the result of success validation of enclave transaction:

```
enum TxEnclaveAction {
    Transfer {
        fee: Fee,
        spend_utxo: Vec<TxoPointer>,
        create_utxo: TxoIndex,
        sealed_log: SealedLog,
    },
    Deposit {
        fee: Fee,
        spend_utxo: Vec<TxoPointer>,
        deposit: (StakedStateAddress, Coin),
    },
    Withdraw {
        fee: Fee,
        // withdraw supposed to withdraw all unbonded,
        // the amount here only to do sanity check
        withdraw: (StakedStateAddress, Coin),
```

```
        create_utxo: TxoIndex,
        sealed_log: SealedLog,
    },
}
```

It's executed as follows:

- `fee`

  Written into block result events, and get accumulated into reward pool afterward.

- `spend_utxo` / `create_utxo`

  Modify UTxO storage

- `sealed_log`

  Write into storage

- `deposit` / `withdraw`

  Modify staking state and related states

Public transactions

Common validations:

- Check tx attributes (`chain_hex_id` , `app_version`)
- Verify staking witness
- Check the witness match the staking address
- Check `nonce` matches

Unbond

The actual paid fee is `context.min_fee_computed` .

Actions:

- `staking.bonded -= tx.value`
- `staking.unbonded += tx.value - context.min_fee_computed`
- `staking.unbonded_from = block_time + unbonding_period`

Validations:

- Not jailed
- Coin computations no error, which implies:
  - `tx.value >= context.min_fee_computed`
  - `staking.bonded >= tx.value`

Unjail

The actual paid fee is zero.

Action:

- Transit staking state from "jailed validator" to "inactive(unjailed) validator" (refer to staking state).

Validations:

- Already jailed
- `block_time >= jailed_until`

Node Join

The actual paid fee is zero.

Action

- Transit staking state from "clean staking" or "inactive(unjailed) validator" to active validator (refer to staking state)

Validations:

- Not jailed
- Not active
- `staking.bonded >= minimal_required_staking`
- Validator address not used
- Validator address not banned
- Used validator address list not full when re-join with new validator key.

Appendix

Tx Storage

KV Store

- `COL_TX_META` , `txid -> BitVec` UTxO spent status.
- `COL_ENCLAVE_TX` , `txid -> sealed tx payload`
  - abci query sealed payload for tx inputs, pass them to tx-validation-enclave for it to validate the tx.
  - For tx-query to query sealed tx payload.
- `COL_WITNESS` , `txid -> TxWitness` Only for abci-query
- `COL_BODIES` , `txid -> Tx` Only for abci-query

Merkle Trie

Staking states are stored in merkle trie.

Wallets

There are three wallet options available for you to manage your funds and interact with Crypto.com chain. They are

- ClientCLI
- Sample Wallet
- ClientRPC

ClientCLI

ClientCLI is a command line interface version of the wallet client. It supports most of the operations including wallet management, funds transfer and basic staking operations.

Sample Chain Wallet

Sample Chain Wallet is a sample wallet client with graphical user interface. It supports wallet management and funds transfer.

ClientRPC

ClientRPC is the JSONRPC interface of the wallet client. It can be used to be integrated with different services and also power the Sample Wallet. It provides nearly the same set of operations as ClientCLI.

ClientCLI →

ClientCLI

ClientCLI is a command line interface for the wallet client. It supports wallet management, funds transfer and basic staking operations.

Build and configurations

Build Prerequisites

- Crypto.com Chain: https://github.com/crypto-com/chain

Build instructions

ClientCLI is bundled with the Crypto.com chain code. After you have compile the binaries, it is available under `./bin/client-cli`.

How to use

```sh
$ ./bin/client-cli [command] [argument]
```

There is also a help command available at

```sh
$ ./bin/client-cli --help
```

Wallet Storage

By default, your wallets are stored in the folder `./storage` located at the same path of where `./client-cli` is executed.

Make sure you have backed up your wallet storage after creating the wallet or else your funds may be inaccessible in case of accident forever.

Configure Wallet Storage Location

To customize the wallet storage location when running ClientCLI, you can update the environment variable `CRYPTO_CLIENT_STORAGE` with the path:

```sh
$ CRYPTO_CLIENT_STORAGE=/my-wallet-storage ./bin/client-cli ...
```

Chain ID

Crypto.com Chain has different Chain ID to distinguish between *devnet*, *testnet* and *mainnet*. Accordingly, you should set up your ClientCLI and use the correct configuration for the node you are connecting to.

Configure Chain ID

To customize the Chain ID while running ClientCLI, you can update the environment variable `CRYPTO_CHAIN_ID` with the full chain ID. For example, we can change it to the testnet with Chain ID `testnet-thaler-crypto-com-chain-42` by:

```sh
$ CRYPTO_CHAIN_ID=testnet-thaler-crypto-com-chain-42 ./bin/client-cli ...
```

Configure Tendermint URL

Similarly, we can also change the Tendermint URL when running ClientCLI, update environment variable `CRYPTO_CLIENT_TENDERMINT` with the Tendermint URL.

Options

A list of supported environment variables of ClientCLI is listed below:

Option	Description	Type	Default Value
CRYPTO_CLIENT_DEBUG	How detail should the debug message be on error	true/false	false
CRYPTO_CHAIN_ID	Full Chain ID	String	---
CRYPTO_CLIENT_STORAGE	Wallet storage directory	Storage directory	.storage

Option	Description	Type	Default Value
CRYPTO_CLIENT_TENDERMINT	Websocket endpoint for tendermint	String	ws://localhost:26657/websocl

Wallet operations

First of all, you will need a wallet to store and spend your CRO.

`wallet new` - Create a new wallet

Currently, `client-cli` supports two types of wallets: The *basic wallet* and the *HD (Hierarchical Deterministic) wallet*.

> **Important note:**
>
> For safety reasons, it is **strongly suggested** that users should create their wallet in the form of an HD wallet to prevent loss or damage of the device.

- **Basic wallet** [`--type basic`]

You can create a new basic wallet with the name "Default" as in the following example:

▶ Example: Create a basic wallet

- **HD wallet** [`--type hd`]

  The HD wallet comes with a "seed phrase", which is serialized into a human-readable 24-word mnemonic. User can restore their wallet and associated addresses with the seed phrase.

  ▶ Example: Create a HD wallet

> **WARNING**

> It is important that you keep the passphrase (and mnemonic for HD wallet) secure, as there is **no way** to recover it. You would not be able to access the funds in the wallet if you forget the passphrase.

`wallet restore` - Restore an HD wallet

You can restore an HD wallet with the mnemonic.

▶ Example: Restore an HD wallet

`wallet list` - List your wallets

Multiple wallets can be created when needed. You can list all wallets saved under the storage path.

▶ Example: List all of your wallets

`wallet delete` - Delete a wallet

You can deleted a wallet in your storage path.

> **WARNING**
>
> Make sure you have backed up the wallet mnemonic before removing any of your wallets, as there will be no way to recover your wallet without the mnemonic.

▶ Example: Remove a wallet

`wallet auth-token` - Show the authentication token

All authorised commands required the authentication token of the wallet, it can be shown by using the `auth-token` subcommand:

▶ Example: Show authentication token of your wallet

Address operations

Once we have a wallet, we are now ready to create new addresses for receiving/staking funds. The most common address types in Crypto.com Chain are

1. **Transfer address**: For normal token transfer;
2. **Staking address**: For staking related operations.

`address new` - Create a new address

Addresses can be created by using the `address new` command:

- Create a *Transfer* address [`--type Transfer`]

  ▶ Example: Create a Transfer type address

- Create a *Staking* address [`--type Staking`]

  ▶ Example: Create a Staking type address

`address list` - List your addresses

You can list all of your addresses with specified type under a wallet.

▶ Example: List your addresses

`sync` - Sync your wallet

It is important to keep your wallet sync with the blockchain before doing any wallet operation. This can be easily done by the `sync` command.

▶ Example: Sync your wallet

`view-key` - Obtain the View Key

In Crypto.com Chain, transactions are encrypted, and it can only be viewed by the owner of the view key. This functionality provides us with an extra layer of privacy and accountability.

It is important that in order for the receiver to spend the funds, they would need to be able to view the transaction details and obtain the corresponding UTXO data. Therefore, the receiver's view key is one of the essential components when launching a transaction.

▶ Example: Obtain the View Key

Transaction operations

Transactions can be created using the `transaction new` command. In this section, we break it down into Transfer and Staking operations for different transaction types.

Transfer operations

Transfer operation involves the transfer of tokens between two *transfer* addresses.

Transfer prerequisites

As mentioned earlier, to send funds to another address, we will have to obtain the receiver's **transfer address** and their **view key** in the first place. More than one view keys can be inserted into a transaction.

Send Funds [`--type Transfer`]

▶ Example: Send funds from a transfer address to another.

Receive funds

On the other hand, similarly, to receive funds, you will need to present your **address** and **view key** to the sender.

Staking operations

Staking operations involve the interaction between *transfer* address and *staking* address. It allows you to lock/unlocking funds for staking purposes.

- **Bond you funds** [`--type Deposit`]

  To bond funds for staking, you can deposit funds (an unspent transaction) to a *staking* address by the `Deposit` operation.

  ▶ Example: Bond funds from a transfer address to a staking address

  > **TIP**
  > - Note that a `<TRANSASTION_ID>` is required as an input of this transaction. It can be found by checking the history of your wallet. This deposit transaction is valid only if the transaction input is unspent.

- **Unlock your bonded funds** [`--type Unbond`] On the other hand, we can create a `Unbond` transaction to unbond staked funds

  ▶ Example: Unbond funds from a staking address

  > **TIP**
  > - The unbonded amount will go to the `Unbonded` balance in your staking address. It will be locked until the `unbonding_period` has passed. Details about the deposited funds in the staking address can be found by checking its staking state.

- **Withdraw your unbonded funds** [`--type Withdraw`]

  Once the `unbonding_period` has passed, we can create a `Withdraw` transaction to withdraw the staked funds (view-key required)

  ▶ Example: Withdraw funds from a staking address

Please also refer to this diagram for interaction between *staking address* and *transfer address*

Balance & transaction history

`balance` - Check your transferable balance

You can check your *transferable* balance with the `balance` command.

▶ Example: Check your transferable balance

> **Note**
>
> - Please note that `balance` will only show your *transferable* balance, for *staking* related balance, please check it with the `state` command.
> - Once a transaction has been sent, the remaining amount (*Total* amount subtracted by the transaction amount and fees) will go to "*Pending*" status. The balance will be settled once the transaction has been confirmed. Therefore, It is suggested that you should `sync` your wallet before checking the balance to obtain the latest balance.

`history` - Check your transaction history

Besides checking the transferable balance of your wallet, you can view a more detailed transaction history.

▶ Example: Check your transaction history

It provides you with details such as the **Transaction ID**, **Direction**(In/out), **Amount** and **Fee** of the transaction, the **Block Height** of where the transaction happened and its corresponding **Block Time**.

`state` - Check the staking state

Crypto.com Chain uses a mixed UTXO+Accounts model, besides checking *transferable* balance of a *transfer* type address, we can check the state of a *staking* type address.

▶ Example: Check the state of a staking address

Advance operations and transactions

`node-join` - Joining the network as a validator

Anyone who wishes to become a validator can submit a `NodeJoinTx` by

```sh
$ ./bin/client-cli transaction new --name Default --type node-join
```

See here for the actual requirement of becoming a validator.

`unjail` - Unjailing a validator

Validator could be punished and jailed due to network misbehaviour. After the jailing period has passed, one can broadcast a `UnjailTx` to unjail the validator and resume its normal operations by

```sh
$ ./bin/client-cli transaction new --name Default --type unjail
```

`export` / `import` - Export & Import Transactions

As mentioned before, sender should add the receiver's view-key to the transaction. Because sender can't push data directly to the receiver. However, it is also possible to send / receive a payment by directly exchanging the (raw) transaction payload data. The sender (who creates the transaction) would export it, the receiver would import it and check the transaction data locally and check the transaction ID against the distributed ledger. Following explains the flow:

1. **Sender**: Get your transaction id from the history, you may need to sync before running the following command:

   ▶ Example: Obtain the transaction id

2. **Sender**: Export the target transaction payload from the sender's wallet:

▶ Example: Export the raw transaction data

3. **Receiver**: The transaction can be imported into receiver's wallet by

▶ Example: Import the raw transaction data

4. Finally, receiver can verify this transaction by checking the transaction history

`multisig` - Multi-signature operations

Crypto.com Chain implemented the threshold Multisig for multi-signature related features. Specifically, a threshold multi-signature addresses requires multiple keys to authorize a transaction.

Create a public key for multi-signature address

To begin, we would need to create a new public key for the multi-signature address by the `new-address-public-key` command:

▶ Example: Create a public key for multi-signature address

Generating a multi-signature address

Now we are ready to create a *M-of-N* multi-signature address by the `new-address` command, where

- *M* is the minimum signatures required to spend the funds from the multi-signature address;
- *N* is the total number of keys involved;
- *N* has to be greater or equal to *M*.

Some of the actual use cases of multi-signature are covered in these application examples.

▶ Example: Create a M-of-N multi-signature address

List your public keys

We can list the public keys for multi-signature address by the `list-address-public-keys` command:

▶ Example: List public keys for multi-signature address

List your multi-signature addresses:

Finally, multi-signature addresses will be kept in your wallet after it has been generated. We can list them by using the address-list command as mentioned earlier.

▶ Example: List multi-signature addresses

← Wallets

Sample Chain Wallet →

ClientRPC

ClientRPC is the JSONRPC interface of the wallet client. It can be used to be integrated with different services and also power the Sample Wallet. It provides nearly the same set of operations as ClientCLI.

Build

Build Prerequisites

- Crypto.com Chain: https://github.com/crypto-com/chain

Build instructions

ClientCLI is bundled with the Crypto.com chain code. After you have compile the binaries, it is available under `./bin/client-rpc`.

How to start?

```sh
$ ./bin/client-rpc
```

For help, there is a help command available:

```sh
$ ./bin/client-rpc --help
```

Wallet Storage

By default, your wallets are stored in the folder `./storage` located at the same path of where `./client-rpc` is executed.

Make sure you have backed up your wallet storage after creating the wallet or else your funds may be inaccessible in case of accident forever.

Configure Wallet Storage Location

To customize the wallet storage location when running ClientRPC, you can provide option `storage-dir` with the path:

```sh
$ ./bin/client-rpc --storage /my-wallet-storage ...
```

Chain ID

Crypto.com Chain has different Chain ID to distinguish between *devnet*, *testnet* and *mainnet*. Accordingly, you should set up your ClientCLI and use the correct configuration for the node you are connecting to.

Configure Chain ID

To customize the Chain ID while running ClientRPC, you can update the option `chain-id` with the chain ID. For example, we can change it to the testnet with Chain ID `testnet-thaler-crypto-com-chain-42` by:

```sh
$ ./bin/client-rpc --chain-id=testnet-thaler-crypto-com-chain-42 ...
```

Configure Tendermint URL

Similarly, we can also change the Tendermint URL when running ClientRPC, update both options `tendermint-url` and `websocket-url` with the Tendermint URL.

Options

A list of supported options of ClientRPC is listed below

Option	Description	Default Value
host	Host ClientRPC server listen to	0.0.0.0
port	Port ClientRPC server listen to	9981
chain-id	Full Chain ID	---
storage-dir	Wallet storage directory	.storage
tendermint-url	Tendermint URL	http://localhost:26657/

Option	Description	Default Value
websocket-url	Tendermint WebSocket URL	ws://localhost:26657/websocket

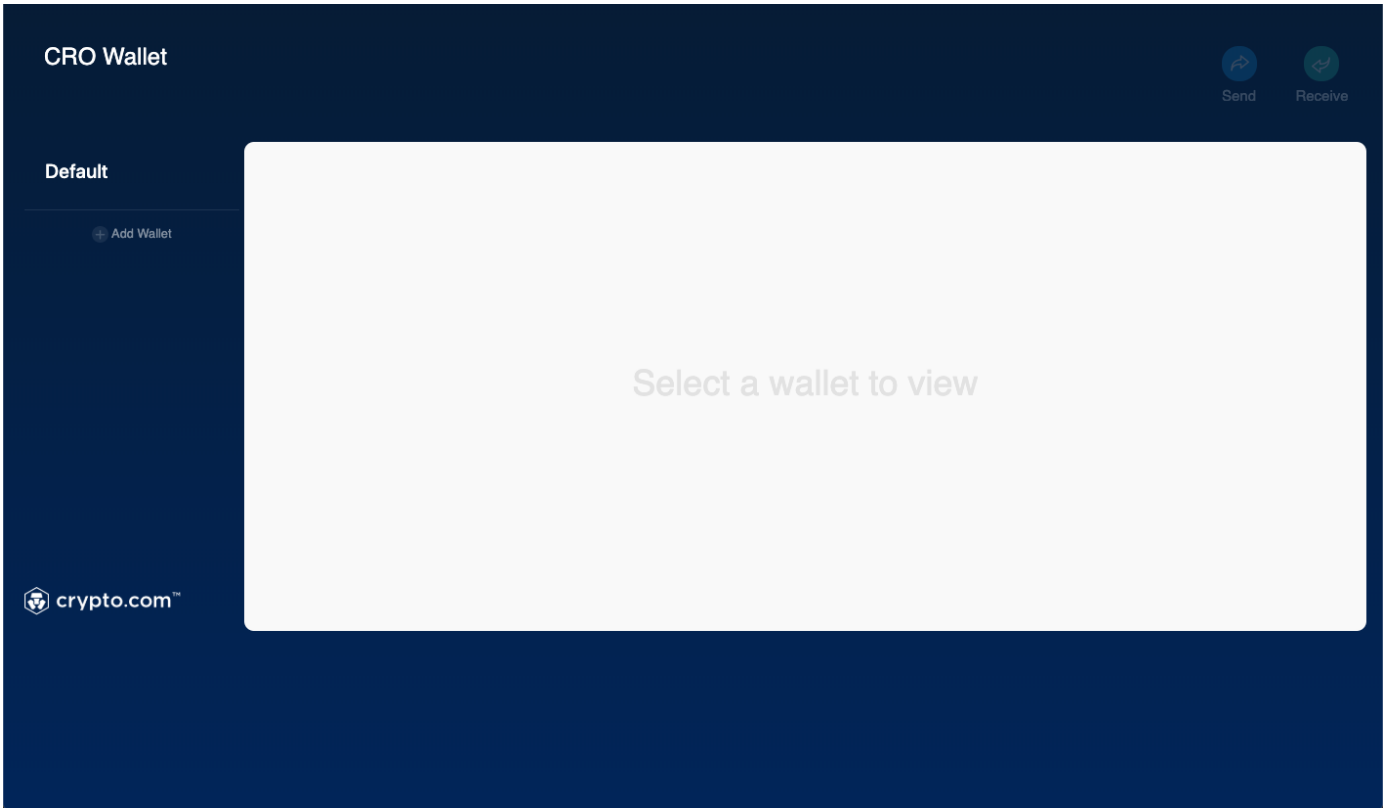
RPC Methods List

TODO

← Sample Chain Wallet

Sample Chain Wallet

Sample Chain Wallet is a sample implementation of the wallet client with a graphical user interface. It supports wallet management and funds transfer.



Build

Right now, the Sample Chain Wallet does not provide one-click installation or pre-built binary. You will have to build from source code.

Please follow the instructions to build and run the sample wallet.

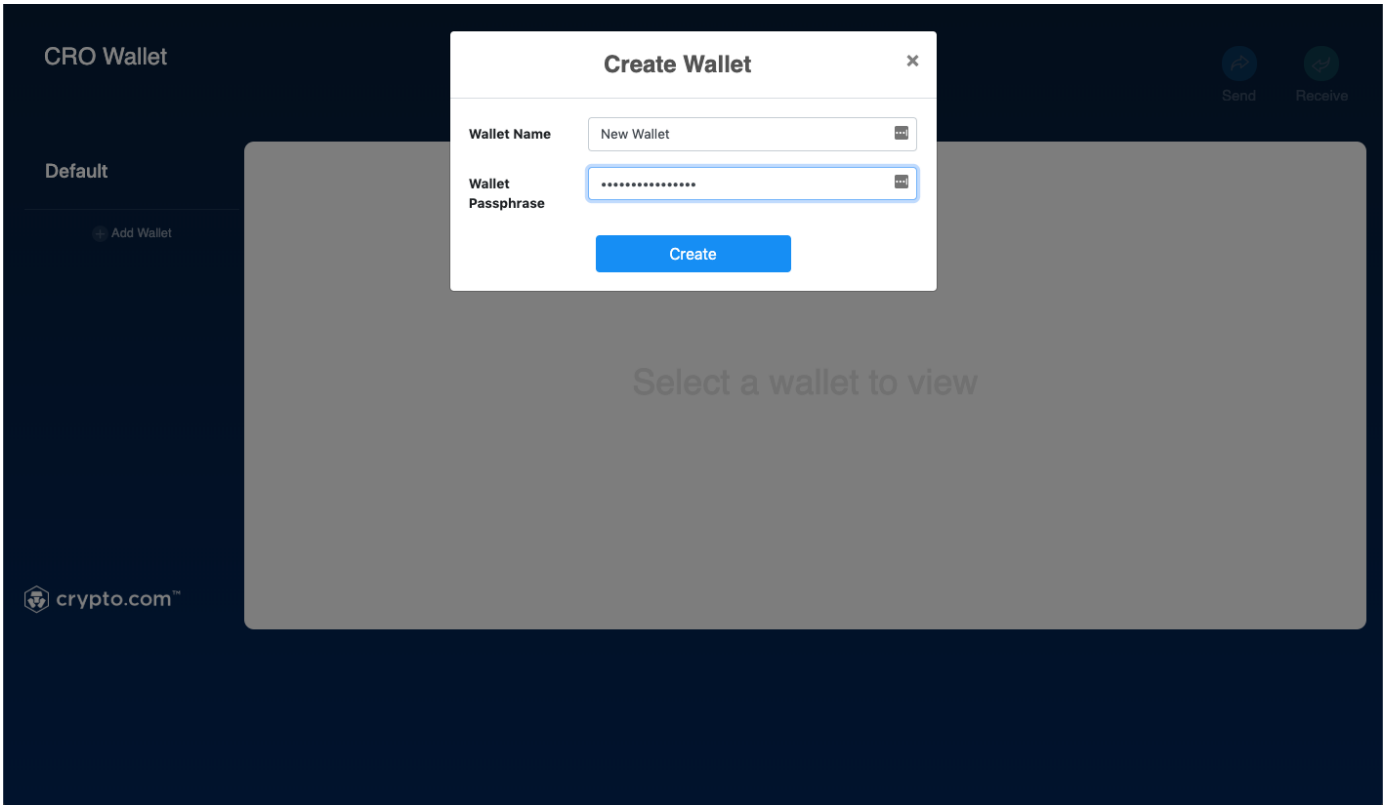
Start ClientRPC

Sample wallet is powered by ClientRPC. To run the Sample Wallet, you will have to start the ClientRPC by following the instructions.

Wallet Management

Create Wallet

1. Click "Add Wallet" at the left side navigation menu;
2. Enter your desired wallet name and passphrase in the popup;
3. Click "Create", and your wallet is created.

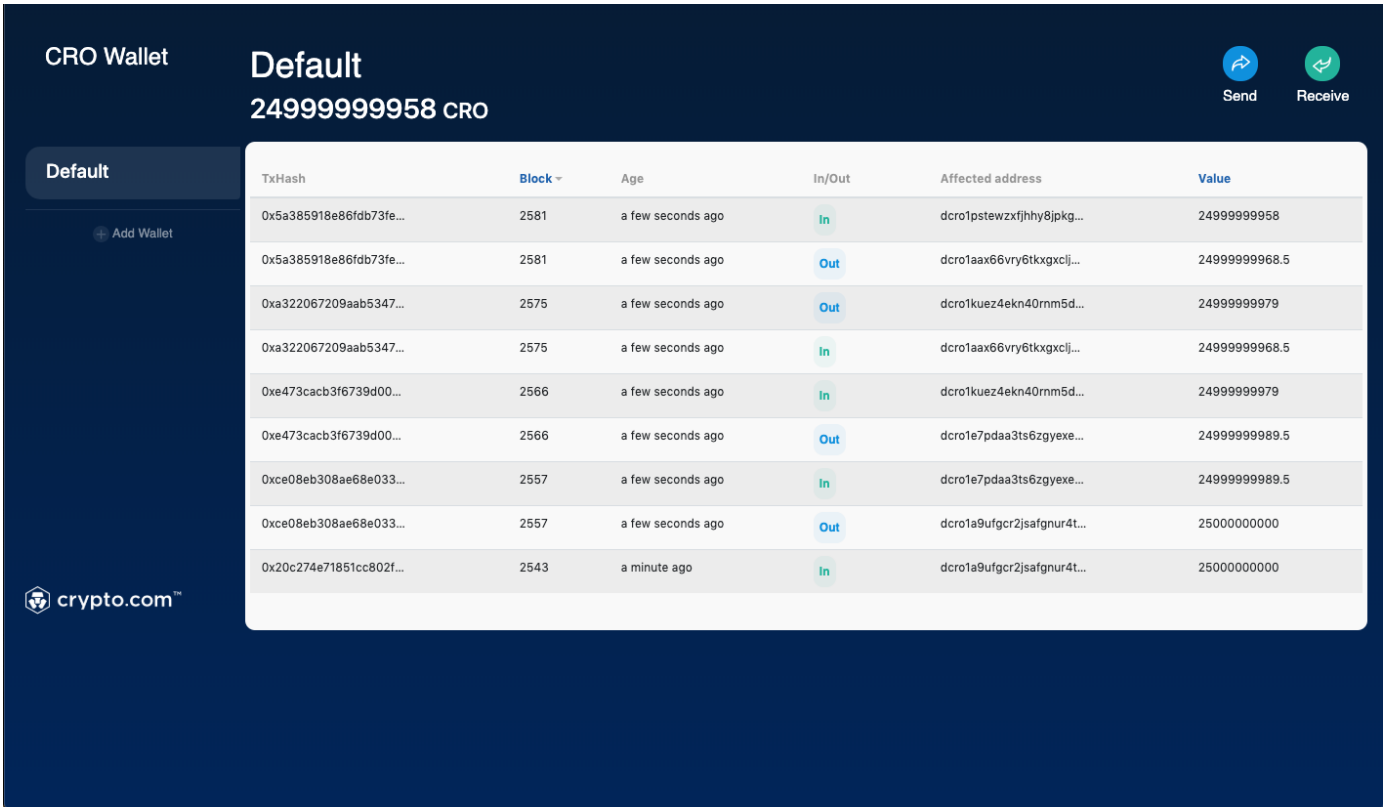


Wallet List

Wallet you owned are listed at the left side navigation menu

List Transactions

1. Select a wallet from the wallet list;
2. Unlock your wallet with the passphrase;
3. You can view the list of transactions related to this wallet.

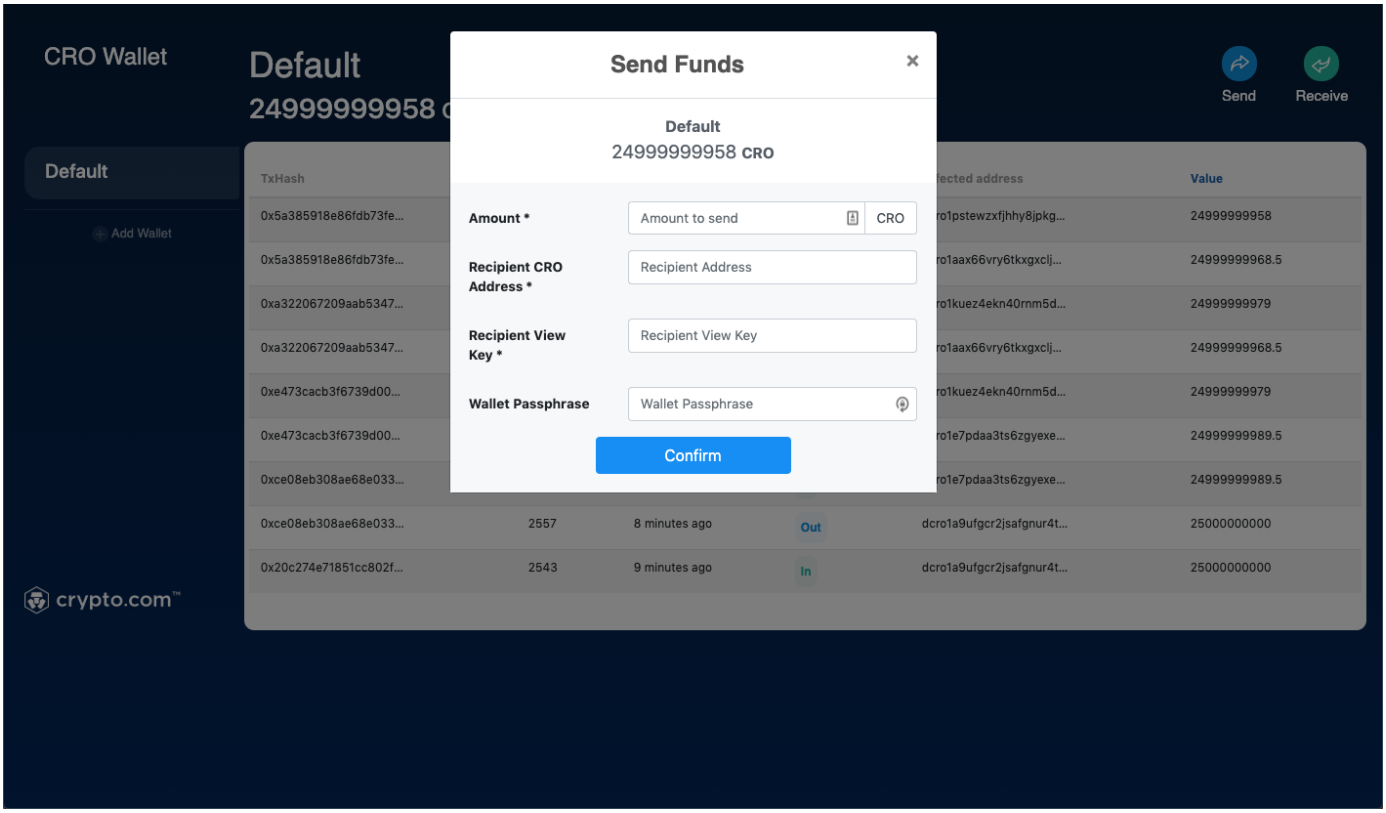


Send and Receive Funds

Send Funds

To send funds to another address, you will have to first obtain the other party's address and their view key.

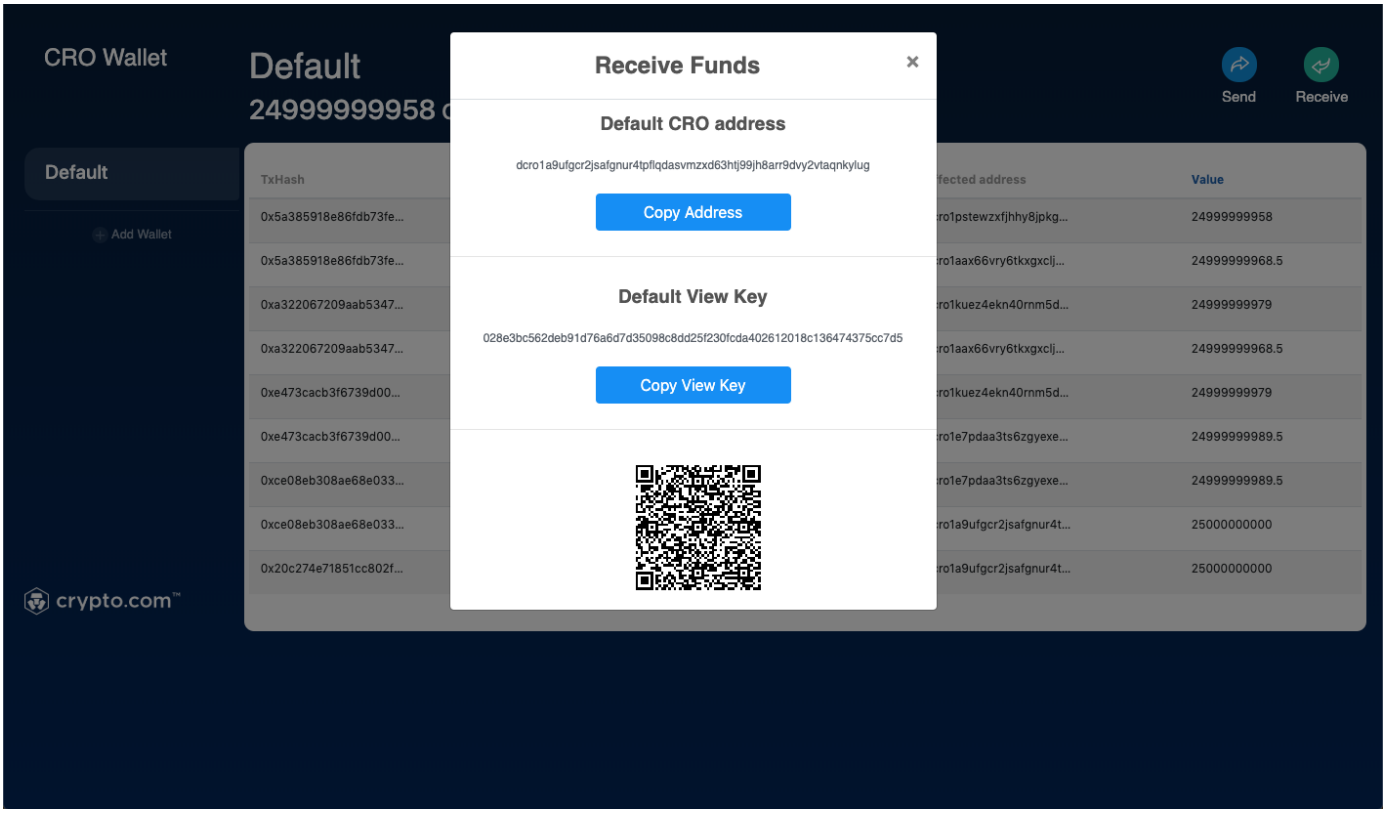
1. Unlock your wallet with the passphrase;
2. Click the *Send* button on the top right-hand corner;
3. Fill in the amount, recipient CRO address, view key and confirm with your passphrase;
4. Click *Confirm*;
5. Preview and make sure the information is correct;
6. Click *Send*.



Receive Funds

To receive funds, you will need to present your address and view key to the sender; these can be obtained by:

1. Unlock your wallet;
2. Click *Receive* at the top right-hand corner;
3. Copy your recipient address and view key.



← ClientCLI

ClientRPC →