Tezos Clients / Wallets

Nomadic Labs

6/12/2018

First we need a node

To run one node on `localhost` we issue the following command :

```
$ alias teztool='docker run -it -v $PWD:/mnt/pwd \
  -e MODE=dind -e DIND_PWD=$PWD \
  -v /var/run/docker.sock:/var/run/docker.sock \
  registry.gitlab.com/nomadic-labs/teztool:latest'

$ teztool babylonnet sandbox --time-between-blocks 10 \
    start 18732
```

This will initialize a node

- listening for RPC on port 18732 (rpc port)
- The node will initialize and run the *babylon* protocol
- Will start a baker and create a block every 10 seconds

Tezos Client

- We install a snap binary
- snap is a container based sw distribution platform

```
$ wget wget https://gitlab.com/abate/tezos-snapcraft/-/raw\
  /master/snaps/tezos_5.1.0_multi.snap?inline=false

$ sudo snap install tezos_5.1.0_multi.snap --dangerous
```

And now finally we can talk with our node or with any Tezos node out there

```
$ export PATH=/snap/bin/:$PATH
$ tezos.client man
```

Tezos Client (cont)

We can keep track of the level that our local node has reached :

```
$ tezos.client -A localhost -P 18732 bootstrapped
```

Notice we connect to the node on `localhost:18732`

Or check the balance :

```
$ tezos.client get balance for bob
```

(We'll play with this in a moment)

Baking

Reminder

- Operations are included in Blocks
- Blocks are created by a baker
- We bake using the bootstrap accounts

First client commands

To familiarize with the command line interface of the node we start exploring the help page and the tezos manual page.

```
$ tezos.client --help
[...]
$ tezos.client man
```

The help page will provide a short list of the global options while the manual a more comprehensive list of all available (sub-)commands.

Configure the client

All command line options can be added to the client configuration file. This is particularly useful to parametrized the client without the need to create long lines of options. The Man page :

```
Commands for editing and viewing the client's config file:
  config show
    Show the config file.
  config reset
    Reset the config file to the factory defaults.
  config update
    Update the config based on the current cli values.
  config init [-o --output <path>]
    Create a config file based on the current CLI values.
    -o --output <path>: path at which to create the file
```

Tezos Client : Global options

We have already seen a few options used to wrap the tezos client and work in our sandbox.

To create a new configuration file based on the current options we can use the command `tezos.client config init`.

```
$ tezos.client -A localhost -P 18732 config init
```

- `-A` : the ip address of the node host (accepts ipv4 and ipv6 addresses)
- `-P` : the RPC port of the node

Configure the client : Configuration file

The resulting file `config` will contain :

```
$ tezos.client config show
{ "base_dir": "~/.tezos-client",
  "node_addr": "127.0.0.1",
  "node_port": 18732, "tls": false,
  "web_port": 8080, "confirmations": 0 }
```

Subsequently the file can be modified either directly or using the command `tezos.client config update` .

Interacting with the node

```
$ tezos.client list understood protocols
ProtoALphaAL
ProtoGenesis
```

- Each time the client runs, it queries the node for the protocol associated to the current chain.
- We can force the client to access all command line options for a specific protocol using `-p <protoHash>` global parameter

Wallets

Creating a wallet using the *tezos cli* is as simple as issuing the command `tezos.client gen keys bob`. Looking at the man page we have different options :

```
Commands for managing the wallet of cryptographic keys:
gen keys <new> [-f --force]
    [-s --sig <ed25519|secp256k1|p256>]
    [--encrypted]
  Generate a pair of keys.
  <new>: new secret_key alias
  -f --force: overwrite existing secret_key
  -s --sig <ed25519|secp256k1|p256>: custom signature algorithm
  --encrypted: Encrypt the key on-disk
```

Wallets : Signing schema

A wallet can also use keys associated to different schema. These can be the default `encrypted` schema or stored on a ledger or remotely. We can choose a schema to store the private keys.

- Scheme ‘encrypted’: Built-in signer using encrypted keys.
- Scheme ‘http’ / ‘https’: Built-in tezos.signer using remote signer through hardcoded http / https requests.
- Scheme ‘ledger’: Built-in signer using Ledger Nano S.
- Scheme ‘tcp’: Built-in tezos.signer using remote signer through hardcoded tcp socket.
- Scheme ‘unencrypted’: Built-in signer using raw unencrypted keys.
- Scheme ‘unix’: Built-in tezos.signer using remote signer through hardcoded unix socket.

Generate a new pair of keys for our friend bob

```
$ tezos.client gen keys bob --encrypted -s ed25519
```

Tezos client has support for three ECC signature schemes:

- *Ed25519* : default for user keys,
- *secp256k1* (the one used in Bitcoin), and
- *P-256* (also called secp256r1) used with Hardware Security Modules (HSMs) mostly.

```
$ tezos.client show address bob
Hash: tz1M4zWSnYfsVyTLqL3hsHifuwwLWo2J196z
Public Key: edpkuYeUB47rzkkcm5tS8Ctrvp7ERooGZUtbH5eRVJkDQ913R8Dx6x
```

Generate a new pair of keys for our friend bob (cont)

```
$ tezos.client show address bob
Hash: tz1M4zWSnYfsVyTLqL3hsHifuwwLWo2J196z
Public Key: edpkuYeUB47rzkkcm5tS8Ctrvp7ERooGZUtbH5eRVJkDQ913R8Dx6x
```

The name `bob` is just a local shortcut to access our account information. When bob wants to give his account to other he must provide his public key (from which others can compute the hash).

A `tz1...` account is called an *implicit* account

Managing the wallet of cryptographic keys

The keys are stored in three files in the client data dir

`~/.tezos-client/` in our example

- `public_key_hashs`,
- `public_keys` and
- `secret_keys`

The format of these files is json. The secret keys are stored on disk encrypted with a password except when using a hardware wallet, or using the schema `unencrypted`.

Sandbox Test accounts

The sandbox contains a few keys.

- Five *bootstrap accounts* are added in the sandbox environment
- One *activator key* that is used to activate the alpha protocol
- These keys are used for testing and have a lot of tokens to play with.
- Do not exist on mainnet/babylonnet

Sandbox Test accounts (cont)

The client on our machine does not know these accounts we need to add them.

```
$ tezos.client list known addresses

tezos.client import secret key bootstrap1 \
  unencrypted:edsk3gUfUPyBSfrS9CCgmCiQsTC...
tezos.client import secret key bootstrap2 \
  unencrypted:edsk39qAm1fiMjgmPkw1EgQYkMz...
tezos.client import secret key bootstrap3 \
  unencrypted:edsk4ArLQgBTLWG5FJmnGnT689V...
tezos.client import secret key bootstrap4 \
  unencrypted:edsk2uqQB9AY4FvioK2YMdfmyMr...
tezos.client import secret key bootstrap5 \
  unencrypted:edsk4QLrcijEffxV31gGdN2HU7U...
```

Sandbox Test accounts (cont)

```
$ tezos.client list known addresses

bob: tz1M4zWSnYfsVyTLqL3hsHifuwwLWo2J196z (encrypted sk known)
activator: tz1TGu6TN5GSez2... (unencrypted sk known)
bootstrap5: tz1ddb9NMYHZi5... (unencrypted sk known)
bootstrap4: tz1b7tUupMgCNw... (unencrypted sk known)
bootstrap3: tz1faswCTDciRz... (unencrypted sk known)
bootstrap2: tz1gjaF81ZRRvd... (unencrypted sk known)
bootstrap1: tz1KqTpEZ7Yob7... (unencrypted sk known)
```

Transactions

One of the basic uses of the `tezos.client` is to make add transactions to the blochchain and to check account balances.

First lets check how many tokens are associated to the account `bootstrap1`

```
 tezos.client get balance for bootstrap1
4000000 tz
```

Transactions (cont)

Now we move 1 tz from `bootstrap1` to the account of our friend `bob`

```
$ tezos.client transfer 1 from bootstrap1 to bob \
   --fee 0.05 --burn-cap 0.257

Node is bootstrapped, ready for injecting operations.
Estimated gas: 10100 units (will add 100 for safety)
Estimated storage: 257 bytes added (will add 20 for safety)
Operation successfully injected in the node.
Operation hash: oonCrfYc4eJmAgaj8uQH5hz28p5C4soFnyiz2xo5tG6KyAGvLfv
Waiting for the operation to be included...
[...]
```

We have to wait for the operation to be added to a new block (i.e. for the baker to create this new block)

Transactions options

- `transfer 1 from bootstrap1 to bob` : this is easy
- `--fee 0.05` : in this transaction we decide to pay a small fee to the baker that will add out transaction to a block
- `--burn-cap 0.257` : this is a fixed fee to be payed for each transaction as anti-spam measure. It is an acknowledgment that `0.257 tz` will be burned.

Since this command can be used also to interact with smart contracts, there are many other options that will be explained later.

Lets analyze the output of this transaction

The first line tells the hash of the block in which our operation was included.

```
Operation found in block:
  BLQAf2eLjmmcTZWhcqmsmQvKp5KRsA3BPLcm2bV4kHT9sUjx8LY
  (pass: 3, offset: 0)
```

The following is the transaction *receipt* : We paid a fee to the baker (in this case this `tz1b7t...` correspond to `bootstrap4`

```
From: tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx
Fee to the baker: tz 0.05
Balance updates:
  tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx ........... - tz 0.05
  fees(tz1b7tUupMgCNw2cCLpKTkSD1NZzB5TkP2sv,84) ... + tz 0.05
```

Lets analyze the output of this transaction (cont)

```
Transaction:
  Amount: tz 1
  From: tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx
  To: tz1iT4BQ9xQxuNgZYAbzWNxS5miMWgycpJsH
  This transaction was successfully applied
  Balance updates:
    tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx ... - tz 1
    tz1iT4BQ9xQxuNgZYAbzWNxS5miMWgycpJsH ... + tz 1
    tz1KqTpEZ7Yob7QbPE4Hy4Wo8fHG8LhKxZSx ... - tz 0.257
```

These fields details the transaction we just executed. Notice that bob paid the burn fee, but since this is a burn fee, it is not credited to anybody.

Receipts

- A receipt recapitulates the effects of the operation on the blockchain.
- Associated to a manager operation, such as a transaction, we have three important parameters: *counter*, *gas* and *storage limit*.
- The counters belongs to each account, they increase at each operation signed by that account
    - each operation is unique
    - each operation is applied once
    - operations are applied in order
    - if we emit operation n and n+1, and n gets lost then n+1 cannot be applied.

Receipts (cont)

We can check the receipts of all the operations included in a block.

```
$ tezos.client rpc get \
  /chains/main/blocks/head/metadata
```

Interacting with the node via RPC

`tezos.client get timestamp` is a shortcut for

```
 tezos.client rpc get \
  /chains/main/blocks/head/header/shell
```

```
{ "level": 6124, "proto": 1,
  "predecessor": "BKsGPx5f94g4jBRUVXdcn1mRtZkYo6ufdfCi8gTFJYx3g896iI
  "timestamp": "2019-01-11T11:37:54Z", "validation_pass": 4,
  "operations_hash": "LLoa7bxRTKaQN2bLYoitYB6bU2DvLnBAqrVjZcvJ364cT
  "fitness": [ "00", "00000000000017ec" ],
  "context": "CoVXsGY6sJdNjNh5UbaB9U7y28mq3cFszKiTJN9UCkM44z95KVsm"
```

RPC lists

- The list of all available RPCs for a specific protocol is available as `tezos.client rpc list`.
- The detailed explanation of each RPC call can be found at Tezos documentation website
- RPCs respect the REST specification.
- The verbs used are `GET`, `POST`, `PUT` and `DELETE`. Some verbs can be called with an additional payload.

Signer

The `tezos.signer` used by the baker to sign blocks on its behalf.

- You can see the signer just as a remote `tezos.client` w.r.t. keys creation (same code base and same commands).
- It listens on http / https / tcp / unix and accepts request to sign a block. By default the signer will sign anything and it needs to be properly configured to be used.
- To run the signer :

```
$ tezos.signer launch socket signer -a your-ip
```