merkle-bridge

Release 0.4.0

Feb 06, 2020

Contents

CHAPTER 1

What is the Aergo Merkle bridge ?

The Aergo Merkle bridge is an efficient and descentralized way of connecting blockchains.

Release blog article: https://medium.com/aergo/the-aergo-merkle-bridge-explained-d95f7dcec510.

In order to transfer an asset from one blockchain to another blockchain, it should be locked on it's origin chain and minted on the destination chain. At all times the minted assets should be pegged to the locked assets.

The Aergo Merkle Bridge enables decentralized custody and efficient minting of assets.

At regular intervals, a proposer publishes the block state root of each chain on the other connected chain's oracle contract. The state root is recorded only if it has been signed by 2/3 of validators. Validators only sign the general block state root, and the proposer creates a Merkle proof, proving that the bridge contract storage state is included in the general block state. Users can then independently mint assets on the destination bridge contract by verifying a merkle proof of their locked assets with the anchored storage root.

The proposers do not need to watch and validate user transfers: the benefit of the merkle bridge design comes from the fact that validators simply make sure that the state roots they sign are correct. Since onchain signature verification is only done once per root anchor, it is possible to use a large number of validators for best safety and sensorship resistance.

1.1 Getting started

1.1.1 Download

```
$ git clone git@github.com:aergoio/merkle-bridge.git
```

1.1.2 Install

Install dependencies

```
$ cd merkle-bridge
$ virtualenv -p python3 venv
$ source venv/bin/activate
$ pip install -r requirements.txt
```

Optional dev dependencies (lint, testing. . .)

```
$ pip install -r dev-dependencies.txt
```

Now you can start using the bridge tools to:

- create a configuration file with the cli
- deploy a new bridge
- start a proposer
- start a validator
- update bridge settings
- transfer assets through the bridge with the cli

1.2 Using the Aergo CLI

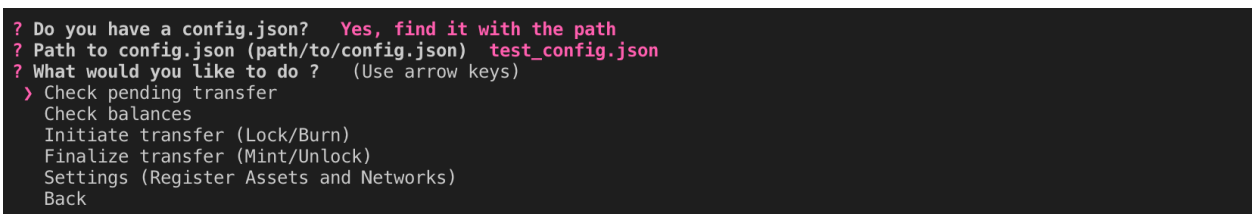
1.2.1 CLI for the proposer/validator

Start the cli:

```
$ python3 -m aergo_cli.main
```

The first step is to create a config file or load an existing one

Then the main menu appears with cli functionalty:



These are the settings available from the cli

```
? What would you like to do ?   Settings (Register Assets and Networks)
? What would you like to do ?   (Use arrow keys)
 ❯ Register new asset
   Register new network
   Register new bridge
   Register new set of validators
   Update anchoring periode
   Update finality
   Back
```

Creating a config file from scratch

```
▶ python3 -m aergo_cli.main
 _____  ___           _____      _____
___   |/  /_____________  /_________  /
__  /|_/ /_  _ \_  ___/_  //_/  _ \_  /
_  /  / / /  __/  /    _  ,<  /  __/  /
/_/  /_/  \___//_/     /_/|_| \___//_/

 _______       _____________                      _________________
___  __ )_____________(_)_____  /_____ _____      __  ____/__  /__(_)
__  __  |_  ___/_  /_  __  /__  __ `/  _ \     _ /    __  /__  /
_  /_/ /_  /    _  / / /_/ / _  /_/ //  __/     / /___  _  / _  /
/_____/ /_/     /_/  \__,_/  _\__, / \___/      \____/  /_/  /_/
                             /____/

Welcome to the Merkle Bridge Interactive CLI.
This is a tool to transfer assets across the Aergo Merke bridge and manage wallet settings (config.json)

? Do you have a config.json?   No, create one from scratch
Let's register 2 networks, validators(optional) and a private key for interacting with each network.
? Network name  net1
? Network IP  ...
? Network name  net2
? Network IP  ...
? Would you like to register a bridge ?  Yes
Bridge between net1 and net2
? Bridge contract address on net1
? Anchoring periode of net2 on net1  10
? Finality of net2  15
? Oracle address on net1
? Bridge contract address on net2
? Anchoring periode of net1 on net2  20
? Finality of net1  25
? Oracle address on net2
? Would you like to register validators ? (not needed for bridge users)  Yes
WARNING : Validators must be registered in the correct order
? Aergo Address  AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ
? Validator ip  ...
? Add next validator ?  Yes
? Aergo Address  ...
? Validator ip  ...
? Add next validator ?  No
Register a private key for net1
? Give your key a short descriptive name  proposer
? Encrypted exported key string  47sDAWjMFTP7r2JP2BJ29PJRfY13yUTtVvoLjAf8knhH4GryQrpMJoTqscDjed1YPHVZXY4sN
? Aergo address matching private key  AmPxVdu993eosN3UjnPDdN3wb7TNbHeiHDvn2dvZUcH8KXDK3RLU
? Path to save new config file  config.json
```

Registering a new bridge

```
(venv)
Python_workspace/aergoio/merkle-bridge  master ✗
▶ python3 -m aergo_cli.main
______  ___          ______      ______
___   |/  /_____________  /_________  /
__  /|_/ /_  _ \_  ___/_  //_/  _ \_  /
_  /  / / /  __/  /   _  ,<  /  __/  /
/_/  /_/  \___//_/    /_/|_| \___//_/

_________        ______________         ________________
___  __ )________(_)______  /______ _____      __  ____/__  /__(_)
__  __  |_  ___/_  /_  __  /__  __ `/  _ \    _  /    __  /__  /
_  /_/ /_  /   _  / / /_/ / _  /_/ //  __/    / /___  _  / _  /
/_____/ /_/    /_/  \__,_/  _\__, / \___/     \____/  /_/  /_/
                            /____/

Welcome to the Merkle Bridge Interactive CLI.
This is a tool to transfer assets across the Aergo Merke bridge and manage wallet settings (config.json)

? Do you have a config.json?  Yes, find it with the path
? Path to config.json (path/to/config.json)  dummy.json
? What would you like to do ?  Settings (Register Assets and Networks)
? What would you like to do ?  Register new bridge
? Departure network  mainnet
? Destination network  sidechain2
Bridge between mainnet and sidechain2
? Bridge contract address on mainnet  AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw
? Anchoring periode of sidechain2 on mainnet  6
? Finality of sidechain2  4
? Bridge contract address on sidechain2  AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw
? Anchoring periode of mainnet on sidechain2  7
? Finality of mainnet  5
? What would you like to do ?  (Use arrow keys)
 ❯ Register new asset
   Register new network
   Register new bridge
   Register new encrypted private key
   Update validators set
   Update anchoring periode
   Update finality
   Back
```

Updating bridge settings

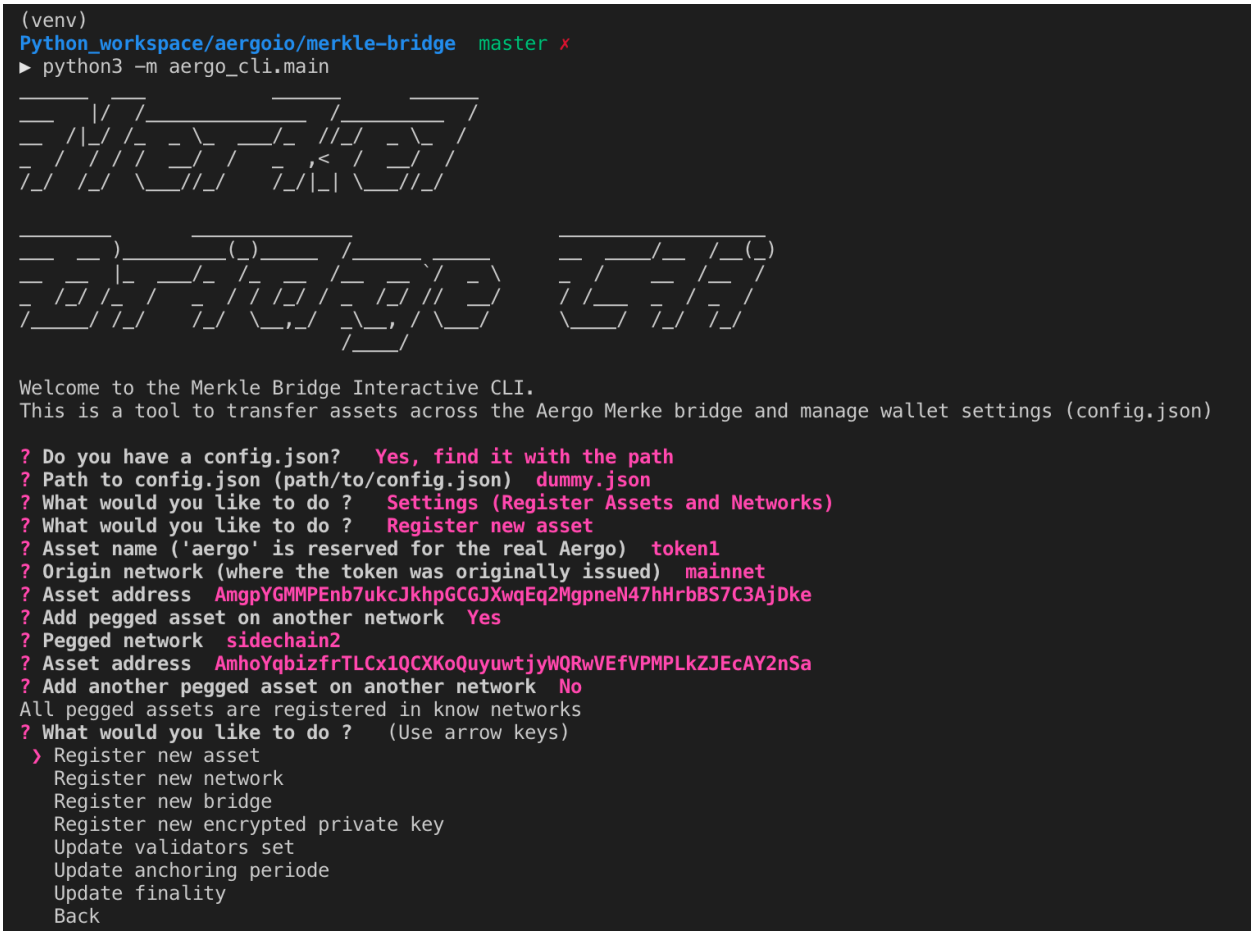
```
? What would you like to do ?  Update anchoring periode
? Departure network  mainnet
? Destination network  sidechain2
? New anchoring periode (nb of blocks) of mainnet onto sidechain2  7
? What would you like to do ?  (Use arrow keys)
 ❯ Register new asset
   Register new network
   Register new bridge
   Register new encrypted private key
   Update validators set
   Update anchoring periode
   Update finality
   Back
```

If the new anchoring periode reached validator consensus, it can then be automatically updated in the bridge contract by the proposer.

```
proposer: mainnet: "Anchoring periode update requested: 7"
proposer: mainnet: " tAnchorUpdate success"
```

1.2.2 CLI for asset transfers

Registering a new asset in config file



Transfering a registered asset

```
(venv)
Python_workspace/aergoio/merkle-bridge  master ✗
▶ python3 -m aergo_cli.main
```

```
Welcome to the Merkle Bridge Interactive CLI.
This is a tool to transfer assets across the Aergo Merke bridge and manage wallet settings (config.json)

? Do you have a config.json?  Yes, find it with the path
? Path to config.json (path/to/config.json)  test_config.json
? What would you like to do ?  Initiate transfer (Lock/Burn)
? Departure network  mainnet
? Destination network  sidechain2
? Name of asset to transfer  token1
? Receiver of assets on other side of bridge  AmNMFbiVsqy6vg4njsTjgy7bKPFHFYhLV4rzQyrENUS9AM1e3tw5
? Amount of assets to transfer  1.2
? Choose account to sign transaction :   default
Lock transfer summary:
Departure chain: mainnet (AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw)
Destination chain: sidechain2 (AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw)
Asset name: token1 (AmgpYGMMPEnb7ukcJkhpGCGJXwqEq2MgpneN47hHrbBS7C3AjDke)
Receiver at destination: AmNMFbiVsqy6vg4njsTjgy7bKPFHFYhLV4rzQyrENUS9AM1e3tw5
Amount: 1200000000000000000

? Confirm you want to execute tranfer tx  Yes, execute transfer

mainnet -> sidechain2
Decrypt exported private key 'default'
Password:
💰 token1 balance on origin before transfer: 499999790.8
🔒 Lock success:  9ir2BnaJaAs73BZEzydXD3rtwX4uDCbrxTrZe1exhEgT
💰 remaining token1 balance on origin after transfer: 499999789.6
Transaction Hash : 9ir2BnaJaAs73BZEzydXD3rtwX4uDCbrxTrZe1exhEgT
Block Height : 2441
```

```
? What would you like to do ?  Finalize transfer (Mint/Unlock)
? Choose a pending transfer  ['mainnet', 'sidechain2', 'token1', 'AmNMFbiVsqy6vg4njsTjgy7bKPFHFYhLV4rzQyrENUS9AM1e3tw5', 2441]
? Choose account to sign transaction :  default
Mint transfer summary:
Departure chain: mainnet (AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw)
Destination chain: sidechain2 (AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw)
Asset name: token1 (AmgpYGMMPEnb7ukcJkhpGCGJXwqEq2MgpneN47hHrbBS7C3AjDke)
Receiver at destination: AmNMFbiVsqy6vg4njsTjgy7bKPFHFYhLV4rzQyrENUS9AM1e3tw5
Block height of lock/burn/freeze: 2441

? Confirm you want to execute tranfer tx  Yes, execute transfer

mainnet -> sidechain2
Decrypt exported private key 'default'
Password:
💰 token1 balance on destination before transfer : 0.0
⚙ Built lock proof
⛏ Mint success:  HQkYL9EHjoKE53NyCqezgkaD3sZFQwTm6A4zaAndFtt5
💰 token1 balance on destination after transfer : 2.4
? What would you like to do ?   (Use arrow keys)
 ❯ Check pending transfer
   Check balances
   Initiate transfer (Lock/Burn)
   Finalize transfer (Mint/Unlock)
   Settings (Register Assets and Networks)
   Back
```

Check pending transfers

It is possible to check withdrawable balances of pending transfer between chains.

```
Welcome to the Merkle Bridge Interactive CLI.
This is a tool to transfer assets across the Aergo Merke bridge and manage wallet settings (config.json)

? Do you have a config.json?   Yes, find it with the path
? Path to config.json (path/to/config.json)  test_config.json
? What would you like to do ?   Initiate transfer (Lock/Burn)
? Departure network  mainnet
? Destination network  sidechain2
? Name of asset to transfer  token1
? Receiver of assets on other side of bridge  AmNMFbiVsqy6vg4njsTjgy7bKPFHFYhLV4rzQyrENUS9AM1e3tw5
? Amount of assets to transfer  2.3
? Choose account to sign transaction :   default
Lock transfer summary:
Departure chain: mainnet (AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw)
Destination chain: sidechain2 (AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw)
Asset name: token1 (AmgpYGMMPEnb7ukcJkhpGCGJXwqEq2MgpneN47hHrbBS7C3AjDke)
Receiver at destination: AmNMFbiVsqy6vg4njsTjgy7bKPFHFYhLV4rzQyrENUS9AM1e3tw5
Amount: 2300000000000000000

? Confirm you want to execute tranfer tx  Yes, execute transfer

mainnet -> sidechain2
Decrypt exported private key 'default'
Password:
💰 token1 balance on origin before transfer: 499999789.6
🔒 Lock success:  H91rbqFAuAQEFKmFL6XF9Ppx6rX3LUANufrFv4trQUS4
💰 remaining token1 balance on origin after transfer: 499999787.3
Transaction Hash : H91rbqFAuAQEFKmFL6XF9Ppx6rX3LUANufrFv4trQUS4
Block Height : 2660

? What would you like to do ?   Check pending transfer
? Choose a pending transfer  ['mainnet', 'sidechain2', 'token1', 'AmNMFbiVsqy6vg4njsTjgy7bKPFHFYhLV4rzQyrENUS9AM1e3tw5', 2660]
Withdrawable: 2.3  Pending: 0.0
? What would you like to do ?   (Use arrow keys)
 ❯ Check pending transfer
   Check balances
   Initiate transfer (Lock/Burn)
   Finalize transfer (Mint/Unlock)
   Settings (Register Assets and Networks)
   Back
```

If a transfer was made with the cli, the transfer parameters are recorded but it is also possible to check the withdrawable balance of a custom transfer between any chain. 'Withdrawable' is the balance that can be immediatly withdrawn on the other side of the bridge. 'Pending' is the balance that was deposited in the bridge contract but the anchor has not happened on the other side of the bridge so it is not yet withdrawable.

Pending transfers are recorded as an array of [departure chain, destination chain, asset name, receiver, block height of lock/burn]. All pending transfer are store in cli/pending_transfers.json and deleted once finalized.

1.3 Proposer

A proposer connects to all validators and requests them to sign a new anchor with the GetAnchorSignature rpc request. To prevent downtime, anybody can become a proposer and request signatures to validators. It is the validator's responsibility to only sign correct anchors. The bridge contracts will not update the state root if the anchoring time is not reached (t_anchor).

1.3.1 Starting a Proposer

```
$ python3 -m aergo_bridge_operator.proposer_client --help

    usage: proposer_client.py [-h] -c CONFIG_FILE_PATH --net1 NET1 --net2 NET2
                              [--privkey_name PRIVKEY_NAME]
                              [--privkey_pwd PRIVKEY_PWD] [--anchoring_on]
```

(continues on next page)

(continued from previous page)

```
                          [--auto_update] [--oracle_update]

    Start a proposer between 2 Aergo networks.

    optional arguments:
    -h, --help            show this help message and exit
    -c CONFIG_FILE_PATH, --config_file_path CONFIG_FILE_PATH
                            Path to config.json
    --net1 NET1           Name of Aergo network in config file
    --net2 NET2           Name of Aergo network in config file
    --privkey_name PRIVKEY_NAME
                            Name of account in config file to sign anchors
    --privkey_pwd PRIVKEY_PWD
                            Password to decrypt privkey_name
    --anchoring_on        Enable anchoring (can be diseabled when wanting to
                            only update settings)
    --auto_update         Update bridge contract when settings change in config
                            file
    --oracle_update       Update bridge contract when validators or oracle addr
                            change in config file

$ python3 -m aergo_bridge_operator.proposer_client -c './test_config.json' --net1
↪'mainnet' --net2 'sidechain2' --privkey_name "proposer" --anchoring_on

    proposer: MainThread: "mainnet Validators: [
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ',
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ',
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ']"
    proposer: MainThread: "sidechain2 (t_final=4) -> mainnet  : t_anchor=7"
    proposer: MainThread: "Set Sender Account"
    proposer: MainThread: "mainnet Proposer Address:␣
↪AmPxVdu993eosN3UjnPDdN3wb7TNbHeiHDvn2dvZUcH8KXDK3RLU"
    proposer: MainThread: "sidechain2 Validators: [
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ',
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ',
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ']"
    proposer: MainThread: "mainnet (t_final=5) -> sidechain2  : t_anchor=7"
    proposer: MainThread: "Set Sender Account"
    proposer: MainThread: "sidechain2 Proposer Address:␣
↪AmPxVdu993eosN3UjnPDdN3wb7TNbHeiHDvn2dvZUcH8KXDK3RLU"
    proposer: sidechain2: "Current mainnet -> sidechain2  anchor: height: 3585, root:␣
↪0x0x4abe990463eeaf2ebb98971c5358bf0a1e8e33cbc8a75c05222cb324cd503705, nonce: 245"
    proposer: mainnet: "Current sidechain2 -> mainnet  anchor: height: 3585, root:␣
↪0x0x5b5b2ebddf46829d05ba0efbc756c53dbd6603413c9557e3d720e8d5c37ccf94, nonce: 315"
    proposer: sidechain2: " Gathering validator signatures for: root:␣
↪0x36b7ed1f97ff9fb4af052d3c36a80a00961f0e0be569d8012a08678dc8d27a98, height: 3604'"
    proposer: mainnet: " Gathering validator signatures for: root:␣
↪0x3bd469d09fdc0e195063b811c59e88c4d72af53f69d85b783927c76aac34d4cc, height: 3605'"
    proposer: mainnet: " Anchor success,  wait until next anchor time: 7s..."
    proposer: sidechain2: " Anchor success,  wait until next anchor time: 7s..."
```

```
from aergo_bridge_operator.proposer_client import BridgeProposerClient

proposer = BridgeProposerClient(
    "./test_config.json", 'mainnet', 'sidechain2', privkey_name='proposer,
    anchoring_on=True
```

(continues on next page)

(continued from previous page)

```
)
proposer.run()
```

1.3.2 Updating bridge settings

Bridge settings are updated when the config file changes and the proposer is started with –auto_update The proposer will then try to gather signatures from validators to make the update on chain.

```
? What would you like to do ?   Update anchoring periode
? Departure network  mainnet
? Destination network  sidechain2
? New anchoring periode (nb of blocks) of mainnet onto sidechain2  7
? What would you like to do ?   (Use arrow keys)
 ❯ Register new asset
   Register new network
   Register new bridge
   Register new encrypted private key
   Update validators set
   Update anchoring periode
   Update finality
   Back
```

If the new anchoring periode reached validator consensus, it can then be automatically updated in the bridge contract by the proposer.

```
proposer: mainnet: "Anchoring periode update requested: 7"
proposer: mainnet: " tAnchorUpdate success"
```

1.4 Validator

A validator will sign any state root from any proposer via the GetAnchorSignature rpc request as long as it is valid. Therefore a validator must run a full node. Assets on the sidechain are secure as long as 2/3 of the validators validate both chains and are honnest. Since signature verification only happens when anchoring (and not when transfering assets), the number of validators can be very high as the signature verification cost is necessary only once per anchor.

1.4.1 Starting a Validator

```
$ python3 -m aergo_bridge_operator.validator_server --help

    usage: validator_server.py [-h] -c CONFIG_FILE_PATH --net1 NET1 --net2 NET2 -i
                               VALIDATOR_INDEX [--privkey_name PRIVKEY_NAME]
                               [--anchoring_on] [--auto_update] [--oracle_update]
                               [--local_test]

    Start a validator between 2 Aergo networks.

    optional arguments:
    -h, --help            show this help message and exit
    -c CONFIG_FILE_PATH, --config_file_path CONFIG_FILE_PATH
                            Path to config.json
    --net1 NET1           Name of Aergo network in config file
    --net2 NET2           Name of Aergo network in config file
    -i VALIDATOR_INDEX, --validator_index VALIDATOR_INDEX
```

(continues on next page)

(continued from previous page)

```
                            Index of the validator in the ordered list of
                            validators
    --privkey_name PRIVKEY_NAME
                            Name of account in config file to sign anchors
    --anchoring_on          Enable anchoring (can be diseabled when wanting to
                            only update settings)
    --auto_update           Update bridge contract when settings change in config
                            file
    --oracle_update         Update bridge contract when validators or oracle addr
                            change in config file
    --local_test            Start all validators locally for convenient testing

$ python3 -m aergo_bridge_operator.validator_server -c './test_config.json' --net1
↪'mainnet' --net2 'sidechain2' --validator_index 1 --privkey_name "validator" --
↪anchoring_on

    "Bridge validators : ['AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ',
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ',
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ']"
    "mainnet <- sidechain2 (t_final=4) : t_anchor=6"
    "mainnet (t_final=5) -> sidechain2 : t_anchor=7"
    "WARNING: This validator will vote for settings update in config.json"
    Decrypt exported private key 'validator'
    Password:
    "Validator Address: AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ"
    server 1 started
    {"val_index": 1, "signed": true, "type": " anchor", "value": {"root":
↪"0xff7c55cba10790c3476cfe141b7579338fdc5ef623788ba634c958b8974c9109", "height":␣
↪3965}, "destination": "sidechain2", "nonce": 281}
    {"val_index": 1, "signed": true, "type": " anchor", "value": {"root":
↪"0x86a270e930624ffd614e211019c0d613320bedad4f3b464759a24b41120061df", "height":␣
↪3971}, "destination": "mainnet", "nonce": 358}
```

```
from aergo_bridge_operator.validator_server import ValidatorServer

validator = ValidatorServer(
    "./test_config.json", 'mainnet', 'sidechain2', privkey_name='validator',
    validator_index=2, anchoring_on=True
)
validator.run()
```

1.4.2 Updating bridge settings

The information (validator set, anchoring periods, finality of blockchains) contained in the config file will be used by the validator to vote on changes if –auto_update is enabled. Be careful that the information in config file is correct as any proposer can request a signature of that information. If the proposer gathers 2/3 signatures for the same information them the bridge settings can be updated.

```
? What would you like to do ?   Update anchoring periode
? Departure network  mainnet
? Destination network  sidechain2
? New anchoring periode (nb of blocks) of mainnet onto sidechain2  7
? What would you like to do ?   (Use arrow keys)
 ❯ Register new asset
   Register new network
   Register new bridge
   Register new encrypted private key
   Update validators set
   Update anchoring periode
   Update finality
   Back
```

1.5 Deploying a new bridge

1.5.1 Process

1- Each Validator generates a private key and address to sign bridge messages (anchors, settings update...) and shares the address and validator ip with the bridge Proposer.

2- Proposer creates a config.json file draft. (See *Create a new config file* below).

3- Proposer deploys the eth-merkle-bridge.lua contract on bridged networks (See *Deploy the bridge contracts* below).

4- Proposer deploys the oracle.lua on both networks and transfers bridge controle to oracles (See *Transfer control of the bridge to the multisig oracle* below).

5- Proposer removes his private key registered in config.json, and shares config.json with Validators.

6- Each Validator adds his private key to his config.json.

7- The Validators start validating (see validator docs) with the correct validator index (see position of validator in config.json).

8- Proposer starts operating the bridge (see proposer docs).

1.5.2 Create a new config file

A config file can be created with the cli tool or manually.

```
▶ python3 -m aergo_cli.main
______  ___          ______       ______
___   |/  /_____________  /_________  /
__  /|_/ /_  _ \_  ___/_  //_/  _ \_  /
_  /  / / /  __/  /   _  ,<  /  __/  /
/_/  /_/  \___//_/    /_/|_| \___//_/

________       ______________             __________________
___  __ )_________(_)_____  /_______ _____    __  ____/__  /__(_)
__  __  |_  ___/_  /_  __  /__  __ `/  _ \   _  /    __  /__  /
_  /_/ /_  /   _  / / /_/ / _  /_/ //  __/   / /___  _  / _  /
/_____/ /_/    /_/  \__,_/  _\__, / \___/    \____/  /_/  /_/
                            /____/

Welcome to the Merkle Bridge Interactive CLI.
This is a tool to transfer assets across the Aergo Merke bridge and manage wallet settings (config.json)

? Do you have a config.json?  No, create one from scratch
Let's register 2 networks, validators(optional) and a private key for interacting with each network.
? Network name  net1
? Network IP  ...
? Network name  net2
? Network IP  ...
? Would you like to register a bridge ?  Yes
Bridge between net1 and net2
? Bridge contract address on net1
? Anchoring periode of net2 on net1  10
? Finality of net2  15
? Oracle address on net1
? Bridge contract address on net2
? Anchoring periode of net1 on net2  20
? Finality of net1  25
? Oracle address on net2
? Would you like to register validators ? (not needed for bridge users)  Yes
WARNING : Validators must be registered in the correct order
? Aergo Address  AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ
? Validator ip  ...
? Add next validator ?  Yes
? Aergo Address  ...
? Validator ip  ...
? Add next validator ?  No
Register a private key for net1
? Give your key a short descriptive name  proposer
? Encrypted exported key string  47sDAWjMFTP7r2JP2BJ29PJRfY13yUTtVvoLjAf8knhH4GryQrpMJoTqscDjed1YPHVZXY4sN
? Aergo address matching private key  AmPxVdu993eosN3UjnPDdN3wb7TNbHeiHDvn2dvZUcH8KXDK3RLU
? Path to save new config file  config.json
```

1.5.3 Deploy the bridge contracts

The sender of the deployment tx will be the bridge owner. Ownership is then transfered to the multisig oracle.

```
$ python3 -m aergo_bridge_operator.bridge_deployer --help

    usage: bridge_deployer.py [-h] -c CONFIG_FILE_PATH --net1 NET1 --net2 NET2
                            [--privkey_name PRIVKEY_NAME] [--local_test]

    Deploy bridge contracts between 2 Aergo networks.

    optional arguments:
    -h, --help            show this help message and exit
    -c CONFIG_FILE_PATH, --config_file_path CONFIG_FILE_PATH
                            Path to config.json
    --net1 NET1           Name of Aergo network in config file
    --net2 NET2           Name of Aergo network in config file
    --privkey_name PRIVKEY_NAME
                            Name of account in config file to sign anchors
```

(continues on next page)

(continued from previous page)

```
    --local_test          Start all validators locally for convenient testing


$ python3 -m aergo_bridge_operator.bridge_deployer -c './test_config.json' --net1
↪'mainnet' --net2 'sidechain2' --privkey_name "proposer"

    DEPLOY BRIDGE
    Decrypt exported private key 'proposer'
    Password:
    ------ DEPLOY BRIDGE BETWEEN mainnet & sidechain2 -----------
    ------ Connect AERGO -----------
    ------ Set Sender Account -----------
    > Sender Address: AmPxVdu993eosN3UjnPDdN3wb7TNbHeiHDvn2dvZUcH8KXDK3RLU
    ------ Deploy SC -----------
        > result[G412HSrJUKbEL3P5QLuXn7mt5DxkGvdwMgmdujQz1w3W] : TX_OK
        > result[HVtjZC4r3fB3PYztJjLsmnfdnA29i17aJHLVwebYoX2v] : TX_OK
    ------ Check deployment of SC -----------
    > Bridge Address mainnet: AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw
    > Bridge Address sidechain2: AmgQqVWX3JADRBEVkVCM4CyWdoeXuumeYGGJJxEeoAukRC26hxmw
    ------ Store bridge addresses in config.json  -----------
    ------ Disconnect AERGO -----------
```

1.5.4 Transfer control of the bridge to the multisig oracle

The oracle_deployer script will deploy the oracle contract (with validators previously registered in config.json), and transfer ownership to the newly deployed contract.

```
$ python3 -m aergo_bridge_operator.oracle_deployer --help

    DEPLOY ORACLE
    usage: oracle_deployer.py [-h] -c CONFIG_FILE_PATH --net1 NET1 --net2 NET2
                              [--privkey_name PRIVKEY_NAME] [--local_test]

    Deploy oracle contracts to controle the bridge between 2 Aergo networks.

    optional arguments:
    -h, --help            show this help message and exit
    -c CONFIG_FILE_PATH, --config_file_path CONFIG_FILE_PATH
                            Path to config.json
    --net1 NET1           Name of Aergo network in config file
    --net2 NET2           Name of Aergo network in config file
    --privkey_name PRIVKEY_NAME
                            Name of account in config file to sign anchors
    --local_test          Start all validators locally for convenient testing

$ python3 -m aergo_bridge_operator.oracle_deployer -c './test_config.json' --net1
↪'mainnet' --net2 'sidechain2' --privkey_name "proposer"

    DEPLOY ORACLE
    Decrypt exported private key 'proposer'
    Password:
    ------ DEPLOY ORACLE BETWEEN mainnet & sidechain2 -----------
    ------ Connect AERGO -----------
    ------ Set Sender Account -----------
    > Sender Address: AmPxVdu993eosN3UjnPDdN3wb7TNbHeiHDvn2dvZUcH8KXDK3RLU
```

(continues on next page)

(continued from previous page)

```
    ------ Deploy SC -----------
    validators :  ['AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ',
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ',
↪'AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ']
        > result[GG6THEXUbmj6E2SDxVri67iF1ExLeUoS5WRgCi5vH5zF] : TX_OK
        > result[42dqiZrkb5tn6tDiNGL2ePdAd6akoVs3LVYytuD7iNR9] : TX_OK
    ------ Check deployment of SC -----------
    > Oracle Address mainnet: AmhXrQ7KdNA4naBi2sTwHj13aBzVBohRhxy262nXsPbV2YbULXUR
    > Oracle Address sidechain2: AmhXrQ7KdNA4naBi2sTwHj13aBzVBohRhxy262nXsPbV2YbULXUR
    ------ Store bridge addresses in config.json  -----------
    ------ Transfer bridge control to oracles -----------
    ------ Disconnect AERGO -----------
```

1.6 Configuration file

The config.json file is used by bridge operators, the wallet and the cli to store information about node connections, validator connections, bridge parameters, assets and private keys.

It can be created and updated manually of with the help of the cli.

```
{
    "networks": { // list of registered networks
        "mainnet": {  // name of blockchain network
            "bridges": {  // bridge contracts to other networks
                "sidechain2": {  // name of the network being connected
                    "addr": "AmgEZebmD4BcV4dhKq6h2HcJS2E8vvy5CEYPyrTvuohjQMiJqMC4",  /
↪/ bridge contract (on mainnet) address to sidechain
                    "oracle": "AmgQdbUqDuoX5krsmvSEHc9X3apBuXyJTQ4mimfWzejEsYScTo3f",␣
↪// oracle controling bridge contract 'addr'
                    "t_anchor": 25,  // anchoring periode of sidechain to mainnet
                    "t_final": 5  // minimum finality time of sidechain
                }
            },
            "ip": "localhost:7845",  // ip of a mainnet node for herapy
            "tokens": {  // tokens issued on this network
                "token1": {  // name of token issued on mainnet
                    "addr": "AmghHtk2gpcpMa6bj1v59qCBfNmKZTi8qDGeuMNg5meJuXGTa2Y1",  /
↪/ address of token issued on mainnet
                    "pegs": {  // other networks where this token exists (pegged)
                        "sidechain2":
↪"AmgssNKd5xXoCguDUnF9Bzhh78W5arwnMtTgDvPZxaAViGDCWa3m"   // token contract of the␣
↪asset pegged on another chain
                    }
                }
            }
        },
        "sidechain2": {  // name of blockchain network
            "bridges": {  // bridge contracts to other networks
                "mainnet": {  // name of the network being connected
                    "addr": "Amho9dBsJZdbqC1nG4Vztgy7HWfkc6mxiRKjxMUrjPx6kgszdrsa",  /
↪/ bridge contract (on sidechain) address to mainnet
                    "oracle": "AmgQdbUqDuoX5krsmvSEHc9X3apBuXyJTQ4mimfWzejEsYScTo3f",␣
↪// oracle controling bridge contract 'addr'
                    "id": "3e688cb882552b4f7d9032e0ae55d9",  // bridge id used to␣
↪prevent bridge update replay
```

(continues on next page)

(continued from previous page)

```
                    "t_anchor": 10,  // anchoring periode of mainnet to sidechain2
                    "t_final": 10  // minimum finality time of mainnet
                }
            },
            "ip": "localhost:8845",  // ip of a sidechain2 node for herapy
            "tokens": {}  // tokens issued on this network
            }
        }
    },
    "validators": [  // array of validators that can update the bridge contracts␣
↪(order is important)
        {
            "addr": "AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ",  //␣
↪address of the validator's signing private key
            "ip": "localhost:9841"  // ip address of the validator server signing␣
↪anchors
        },
        {
            "addr": "AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ",
            "ip": "localhost:9842"
        },
        {
            "addr": "AmNLjcxUDmxeGZL7F8bqyaGt3zqog5HAoJmFBEZAx1RvfTKLSBsQ",
            "ip": "localhost:9843"
        }
    ],
    "wallet": {  // named accounts
        "broadcaster": {  // name of account
            "addr": "AmMQNBFXuSqiN97rv2m1NoxWu7D2UKAojKCpWqrLmZh676GgfRGG",  //␣
↪address matching the private key
            "keystore": "keystore/
↪AmMQNBFXuSqiN97rv2m1NoxWu7D2UKAojKCpWqrLmZh676GgfRGG__2020-01-20T04:13:16__keystore.
↪json"  // path to keystore file
        },
        "default": {
            "addr": "AmNPWDJMjU4g98Scm4AikW8JwQMGwWMztM7Qy8ggxNTkhgZMJHFp",
            "keystore": "keystore/
↪AmNPWDJMjU4g98Scm4AikW8JwQMGwWMztM7Qy8ggxNTkhgZMJHFp__2020-01-20T04:13:06__keystore.
↪json"
        },
    }
}
```

1.7 Python wallet

The Python wallet can be used as a python command line tool for making simple transfers, bridge transfers, quering balances … The merkle-bridge/aergo_wallet repository can also be used as a module for other applications as the tools are separate and don't need config.json to be used.

1.7.1 Create / Register a new account

```
from aergo_wallet.wallet import AergoWallet

# create a wallet
wallet = AergoWallet("./config.json")

# create a new account (new private key),
# you will be requested to create a password
# (DO NOT LOSE IT, it is the only way to decrypt your private key)
wallet.create_account("default")

# if you already have an exported private key (created by aergocli for example)
wallet.register_account('default', "exported_private_key", addr="Address_of_private_
↪key")
```

1.7.2 Balance query

```
from aergo_wallet.wallet import AergoWallet

# create a wallet
wallet = AergoWallet("./config.json")

# get Aer balance of the default account on 'mainnet'
balance, _ = wallet.get_balance('aergo', 'mainnet')

# get Aer balance of Aer minted on 'sidechain2'
balance, _ = wallet.get_balance('aergo', 'sidechain2',
                                asset_origin_chain='mainnet')
```

1.7.3 Deploy a test token

```
from aergo_wallet.wallet import AergoWallet

# load the compiled bytecode
with open("./contracts/token_bytecode.txt", "r") as f:
    bytecode = f.read()[:-1]

# create a wallet
wallet = AergoWallet("./config.json")

total_supply = 500*10**18
token_name = "my_token"
# deploy the token and store the address in config.json
wallet.deploy_token(bytecode, token_name, total_supply, "mainnet")
```

1.7.4 Register an already deployed token

This can be done with aergo_cli

```
(venv)
Python_workspace/aergoio/merkle-bridge  master ✗
▶ python3 -m aergo_cli.main
 _____  ___           _____      _____
___  /|/  /_____________  /_____ ___  /
__  /|_/ /_  _ \_  ___/_  //_/  _ \_  /
_  /  / / /  __/  /    _  ,<  /  __/  /
/_/  /_/  \___//_/     /_/|_| \___//_/

 _______       _____________                    _________________
___  __ )_____________(_)_____  /_____ _____     ___  ____/__  /__(_)
__  __  |_  ___/_  /_  __  __ \_  __ `/  _ \    _  /    __  /__  /
_  /_/ /_  /   _  / / /_/ /_  / /_/ //  __/    / /___  _  / _  /
/_____/ /_/    /_/  \__,_/  /_\__, / \___/     \____/  /_/  /_/
                             /____/

Welcome to the Merkle Bridge Interactive CLI.
This is a tool to transfer assets across the Aergo Merke bridge and manage wallet settings (config.json)

? Do you have a config.json?   Yes, find it with the path
? Path to config.json (path/to/config.json)  dummy.json
? What would you like to do ?   Settings (Register Assets and Networks)
? What would you like to do ?   Register new asset
? Asset name ('aergo' is reserved for the real Aergo)  token1
? Origin network (where the token was originally issued)  mainnet
? Asset address  AmgpYGMMPEnb7ukcJkhpGCGJXwqEq2MgpneN47hHrbBS7C3AjDke
? Add pegged asset on another network  Yes
? Pegged network  sidechain2
? Asset address  AmhoYqbizfrTLCx1QCXKoQuyuwtjyWQRwVEfVPMPLkZJEcAY2nSa
? Add another pegged asset on another network  No
All pegged assets are registered in know networks
? What would you like to do ?   (Use arrow keys)
 ❯ Register new asset
   Register new network
   Register new bridge
   Register new encrypted private key
   Update validators set
   Update anchoring periode
   Update finality
   Back
```

or by editing config.json directly.

```
{
    "tokens": {
        "my_token": {
            "addr": "AmgY8WARSNfjgCnFhFJBv145wkHJRTC7YR5MeJGAMvKzVD9kKeFz",
            "pegs": {
                "sidechain2": "AmheFWQf5decPrKZE1dnjh1EFwDq7qqAmobPbrUt4XeNK9QNCyxK"
            }
        }
    }
}
```

or with the wallet:

```
from aergo_wallet.wallet import AergoWallet

# create a wallet
wallet = AergoWallet("./config.json")

" Register a 'mainnet' token and it's pegged self on 'sidechain2'
wallet.register_asset("my_token", "mainnet", "Address on mainnet",
                      pegged_chain_name="sidechain2",
                      addr_on_pegged_chain="Address on sidechain2")
```

1.7.5 Simple Transfers

```
from aergo_wallet.wallet import AergoWallet

# create a wallet
wallet = AergoWallet("./config.json")

# simple asset transfer on 'mainnet'
wallet.transfer(2*10**18, to_address, asset_name="my_token", network_name="mainnet")

# simple asset transfer of 'mainnet' assets pegged on 'sidechain'
wallet.transfer(2*10**18, to_address, asset_name="my_token", network_name="sidechain",
                asset_origin_chain="mainnet")
```

1.7.6 Bridge Transfers

The bridge_transfer method calls transfer_to_sidechain or transfer_from_sidechain depending whether the token was minted or not.

```
from aergo_wallet.wallet import AergoWallet

# create a wallet
wallet = AergoWallet("./config.json")

amount = 1*10**18
asset = 'token1'
# transfer aergo from 'mainnet' to 'sidechain2'
wallet.bridge_transfer('mainnet',
                       'sidechain2',
                       asset,
                       amount)
```

The transfer_to_sidechain method performs the following:

- lock assets in the bridge contract
- wait for the next anchor on sidechain
- create a merkle proof of lock in the anchored state
- mint the asset on the sidechain with the merkle proof

The transfer_from_sidechain method performs the following:

- brun assets in the bridge contract
- wait for the next anchor on mainnet
- create a merkle proof of burn in the anchored state
- unlock the asset on the mainnet with the merkle proof

```
from aergo_wallet.wallet import AergoWallet

# create a wallet
wallet = AergoWallet("./config.json")

amount = 1*10**18
asset = 'token1'
```

(continues on next page)

(continued from previous page)

```
# transfer aergo from 'mainnet' to 'sidechain2'
wallet.transfer_to_sidechain('mainnet',
                             'sidechain2',
                             asset,
                             amount)

# transfer minted aergo from sidechain2 mainnet
wallet.transfer_from_sidechain('sidechain2',
                               'mainnet',
                               asset,
                               amount)
```

It is also possible to perform the lock/burn and mint/unlock operations individually.

```
from aergo_wallet.wallet import AergoWallet

# create a wallet
wallet = AergoWallet("./config.json")

amount = 1*10**18
asset = 'token1'
# lock asset in the bridge contract to 'sidechain2'
lock_height, tx_hash = wallet.initiate_transfer_lock('mainnet', 'sidechain2',
                                                     asset, amount)
# lock more assets in the bridge contract to 'sidechain2'
lock_height, tx_hash = wallet.initiate_transfer_lock('mainnet', 'sidechain2',
                                                     asset, amount)

# get the amount of assets locked but not yet minted on 'sidechain2'
pending_mint = wallet.get_mintable_balance(
    'mainnet', 'sidechain2', asset, pending=True
)

# mint the total balance of two previous locked amounts
pegged_address, tx_hash = wallet.finalize_transfer_mint(
    'mainnet', 'sidechain2', asset, lock_height=lock_height
)


# Similarly,
# wallet.initiate_transfer_burn()
# wallet.get_unlockable_balance()
# wallet.finalize_transfer_unlock()
# can be used to burn and unlock minted assets from a sidechain.
```

1.7.7 Wallet utils

If you wish to use the wallet as a module for other applications, the following tools are available:

- wallet_utils.py
- transfer_to_sidechain.py
- transfer_from_sidechain.py
- token_deployer.py

You will need to connect your own herapy instances to nodes and load your private key in herapy.

1.8 aergo_bridge_operator

class aergo_bridge_operator.validator_server.**ValidatorService**(*config_file_path: str*, *aergo1: str*, *aergo2: str*, *privkey_name: str = None*, *privkey_pwd: str = None*, *validator_index: int = 0*, *anchoring_on: bool = False*, *auto_update: bool = False*, *oracle_update: bool = False*)

Validates anchors for the bridge proposer

GetAnchorSignature(*anchor*, *context*)
Verifies the anchors are valid and signes them aergo1 and aergo2 must be trusted.

GetOracleSignature(*oracle_msg*, *context*)
Get signature to update bridge oracle

GetTAnchorSignature(*tempo_msg*, *context*)
Get a vote(signature) from the validator to update the t_anchor setting in the Aergo bridge contract

GetTFinalSignature(*tempo_msg*, *context*)
Get a vote(signature) from the validator to update the t_final setting in the Aergo bridge contract

GetValidatorsSignature(*val_msg*, *context*)
Get signature to update validators of anchors

get_oracle(*hera: aergo.herapy.aergo.Aergo*, *aergo_from: str*, *aergo_to: str*, *oracle_to: str*, *id_to: str*, *bridge_to: str*, *oracle_msg*)
Get a vote(signature) from the validator to update the oracle controlling the bridge contract

is_valid_anchor(*anchor*, *aergo_from: aergo.herapy.aergo.Aergo*, *aergo_to: aergo.herapy.aergo.Aergo*, *oracle_to: str*) → Optional[str]
An anchor is valid if : 1- it's height is finalized 2- it's root for that height is correct. 3- it's nonce is correct 4- it's height is higher than previous anchored height + t_anchor

class aergo_bridge_operator.proposer_client.**BridgeProposerClient**(*config_file_path: str*, *aergo_mainnet: str*, *aergo_sidechain: str*, *privkey_name: str = None*, *privkey_pwd: str = None*, *anchoring_on: bool = False*, *auto_update: bool = False*, *oracle_update: bool = False*, *bridge_anchoring: bool = True*)

The BridgeProposerClient starts proposers on both sides of the bridge

class aergo_bridge_operator.proposer_client.**ProposerClient**(*config_file_path: str*, *aergo_from: str*, *aergo_to: str*, *is_from_mainnet: bool*, *privkey_name: str = None*, *privkey_pwd: str = None*, *anchoring_on: bool = False*, *auto_update: bool = False*, *oracle_update: bool = False*, *bridge_anchoring: bool = True*)

The proposer client periodically (every t_anchor) broadcasts the finalized block trie state root (after lib) on the other side of the bridge after validation by the Validator servers. It first checks the last merged height and waits until now > lib + t_anchor is reached, then merges the current finalised block (lib). Start again after waiting t_anchor.

Note on config_data:

- config_data is used to store current validators and their ip when the proposer starts. (change validators after the proposer has started)
- After starting, when users change the config.json, the proposer will attempt to gather signatures to reflect the changes.
- t_anchor value is always taken from the bridge contract
- validators are taken from the config_data because ip information is not stored on chain
- when a validator set update succeeds, self.config_data is updated
- if another proposer updates to a new set of validators and the proposer doesnt know about it, proposer must be restarted with the new current validator set to create new connections to them.

buildBridgeAnchorArgs(*root: bytes*) → Tuple[str, List[str]]
Build arguments to derive bridge storage root from the anchored state root with a merkle proof

extract_signatures(*approvals: List[Any]*) → Tuple[List[str], List[int]]
Convert signatures to hex string and keep 2/3 of them.

get_anchor_signatures(*root: str*, *merge_height: int*, *nonce: int*) → Tuple[List[str], List[int]]
Query all validators and gather 2/3 of their signatures.

get_new_oracle_signatures(*oracle*)
Request approvals of validators for the new oracle.

get_new_validators_signatures(*validators*)
Request approvals of validators for the new validator set.

get_signature_worker(*rpc_service: str*, *request*, *h: bytes*, *index: int*) → Optional[Any]
Get a validator's (index) signature and verify it

get_tempo_signatures(*tempo*, *rpc_service*, *tempo_id*)
Request approvals of validators for the new t_anchor or t_final.

monitor_settings()
Check if a modification of bridge settings is requested by seeing if the config file has been changed and try to update the bridge contract (gather 2/3 validators signatures).

monitor_settings_and_sleep(*sleeping_time*)
While sleeping, periodicaly check changes to the config file and update settings if necessary. If another proposer updated settings it doesnt matter, validators will just not give signatures.

new_state_anchor(*root: str, next_anchor_height: int, validator_indexes: List[int], sigs: List[str]*) → None
Anchor a new root on chain

new_state_and_bridge_anchor(*stateRoot: str, next_anchor_height: int, validator_indexes: List[int], sigs: List[str], bridge_contract_proto: str, merkle_proof: List[str]*) → None
Anchor a new state root and update bridge anchor on chain

run() → None
Gathers signatures from validators, verifies them, and if 2/3 majority is acquired, set the new anchored root in bridge_to.

set_oracle(*new_oracle*, *validator_indexes*, *sigs*)
Update oracle on chain

set_tempo(*t_anchor*, *validator_indexes*, *sigs*, *contract_function*) → bool
Update t_anchor or t_final on chain

set_validators(*new_validators*, *validator_indexes*, *sigs*)
Update validators on chain

update_oracle(*oracle*)
Try to update the oracle periode registered in the bridge contract.

update_t_anchor(*t_anchor*)
Try to update the anchoring periode registered in the bridge contract.

update_t_final(*t_final*)
Try to update the anchoring periode registered in the bridge contract.

update_validator_connections()
Update connections to validators after a successful update of bridge validators with the validators in the config file.

update_validators(*new_validators*)
Try to update the validator set with the one in the config file.

wait_next_anchor(*merged_height: int*) → int
Wait until t_anchor has passed after merged height. Return the next finalized block after t_anchor to be the next anchor

exception aergo_bridge_operator.proposer_client.**ValidatorMajorityError**

1.9 aergo_wallet

class aergo_wallet.wallet.**AergoWallet**(*config_file_path: str*, *config_data: Dict[KT, VT] = None*)
A wallet loads it's private key from config.json and implements the functionality to transfer tokens to sidechains

config_data(**json_path, value: Union[str, int, List[T], Dict[KT, VT]] = None*)
Get the value in nested dictionary at the end of json path if value is None, or set value at the end of the path.

deploy_token(*payload_str: str*, *asset_name: str*, *total_supply: int*, *network_name: str*, *receiver: str = None*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*) → str
Deploy a new standard token, store the address in config_data

finalize_transfer_mint(*from_chain: str*, *to_chain: str*, *asset_name: str*, *receiver: str = None*, *lock_height: int = 0*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*) → Tuple[str, str]
Finalize a transfer of assets to a sidechain by minting then after the lock is final and a new anchor was made. NOTE anybody can mint so sender is not necessary. The amount to mint is the difference between total deposit and already minted amount. Bridge tempo is taken from config_data

finalize_transfer_unlock(*from_chain: str*, *to_chain: str*, *asset_name: str*, *receiver: str = None*, *burn_height: int = 0*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*) → str
Finalize a transfer of assets from a sidechain by unlocking then after the burn is final and a new anchor was made. NOTE anybody can unlock so sender is not necessary. The amount to unlock is the difference between total burn and already unlocked amount. Bridge tempo is taken from config_data

get_aergo(*network_name: str*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*, *skip_state: bool = False*) → aergo.herapy.aergo.Aergo
Return aergo provider with account loaded from keystore

get_asset_address(*asset_name: str*, *network_name: str*, *asset_origin_chain: str = None*) → str
Get the address of a time in config_data given it's name

get_balance(*asset_name: str*, *network_name: str*, *asset_origin_chain: str = None*, *account_name: str = 'default'*, *account_addr: str = None*) → Tuple[int, str]
Get account name balance of asset_name on network_name, and specify asset_origin_chain for a pegged asset query,

get_bridge_tempo(*from_chain: str*, *to_chain: str*, *aergo: aergo.herapy.aergo.Aergo = None*, *bridge_address: str = None*, *sync: bool = False*) → Tuple[int, int]
Return the anchoring periode of from_chain onto to_chain and minimum finality time of from_chain. This information is queried from bridge_to.

get_mintable_balance(*from_chain: str*, *to_chain: str*, *asset_name: str*, *account_name: str = 'default'*, *account_addr: str = None*) → Tuple[int, int]
Get the balance that has been locked on one side of the bridge and not yet minted on the other side

Calculates the difference between the total amount deposited and total amount withdrawn. Set pending to true to include deposits than have not yet been anchored

get_unlockable_balance(*from_chain: str*, *to_chain: str*, *asset_name: str*, *account_name: str = 'default'*, *account_addr: str = None*) → Tuple[int, int]

Get the balance that has been burnt on one side of the bridge and not yet unlocked on the other side Calculates the difference between the total amount deposited and total amount withdrawn. Set pending to true to include deposits than have not yet been anchored

initiate_transfer_burn(*from_chain: str*, *to_chain: str*, *asset_name: str*, *amount: int*, *receiver: str = None*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*) → Tuple[int, str]

Initiate a transfer from a sidechain by burning the assets.

initiate_transfer_lock(*from_chain: str*, *to_chain: str*, *asset_name: str*, *amount: int*, *receiver: str = None*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*) → Tuple[int, str]

Initiate a transfer to a sidechain by locking the asset.

register_account(*account_name: str*, *keystore_path: str*, *password: str = None*, *addr: str = None*) → str

Register and exported account to config.json

register_asset(*asset_name: str*, *origin_chain_name: str*, *addr_on_origin_chain: str*, *pegged_chain_name: str = None*, *addr_on_pegged_chain: str = None*) → None

Register an existing asset to config.json

transfer(*value: int*, *to: str*, *asset_name: str*, *network_name: str*, *asset_origin_chain: str = None*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*) → str

Transfer aer or tokens on network_name and specify asset_origin_chain for transfers of pegged assets.

transfer_from_sidechain(*from_chain: str*, *to_chain: str*, *asset_name: str*, *amount: int*, *receiver: str = None*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*) → None

Transfer assets from from_chain to to_chain The asset being transfered back to the to_chain native chain should be a minted asset on the sidechain.

transfer_to_sidechain(*from_chain: str*, *to_chain: str*, *asset_name: str*, *amount: int*, *receiver: str = None*, *privkey_name: str = 'default'*, *privkey_pwd: str = None*) → None

Transfer assets from from_chain to to_chain. The asset being transfered to the to_chain sidechain should be native of from_chain

aergo_wallet.transfer_to_sidechain.**build_lock_proof**(*aergo_from: aergo.herapy.aergo.Aergo*, *aergo_to: aergo.herapy.aergo.Aergo*, *receiver: str*, *bridge_from: str*, *bridge_to: str*, *lock_height: int*, *token_origin: str*) → aergo.herapy.obj.sc_state.SCState

Check the last anchored root includes the lock and build a lock proof for that root

aergo_wallet.transfer_to_sidechain.**lock**(*aergo_from: aergo.herapy.aergo.Aergo*, *bridge_from: str*, *receiver: str*, *value: int*, *asset: str*, *gas_limit: int*, *gas_price: int*) → Tuple[int, str]

Lock can be called to lock aer or tokens. it supports delegated transfers when tx broadcaster is not the same as the token owner

aergo_wallet.transfer_to_sidechain.**mint**(*aergo_to: aergo.herapy.aergo.Aergo*, *receiver: str*, *lock_proof: aergo.herapy.obj.sc_state.SCState*, *token_origin: str*, *bridge_to: str*, *gas_limit: int*, *gas_price: int*) → Tuple[str, str]

Mint the receiver's deposit balance on aergo_to.

aergo_wallet.transfer_from_sidechain.**build_burn_proof**(*aergo_from: aergo.herapy.aergo.Aergo*, *aergo_to: aergo.herapy.aergo.Aergo*, *receiver: str*, *bridge_from: str*, *bridge_to: str*, *burn_height: int*, *token_origin: str*) → aergo.herapy.obj.sc_state.SCState

Check the last anchored root includes the burn and build a burn proof for that root

aergo_wallet.transfer_from_sidechain.**burn**(*aergo_from: aergo.herapy.aergo.Aergo*, *bridge_from: str*, *receiver: str*, *value: int*, *token_pegged: str*, *gas_limit: int*, *gas_price: int*) → Tuple[int, str]

Burn a minted token on a sidechain.

aergo_wallet.transfer_from_sidechain.**unlock**(*aergo_to: aergo.herapy.aergo.Aergo*, *receiver: str*, *burn_proof: aergo.herapy.obj.sc_state.SCState*, *token_origin: str*, *bridge_to: str*, *gas_limit: int*, *gas_price: int*) → str

Unlock the receiver's deposit balance on aergo_to.

aergo_wallet.token_deployer.**deploy_token**(*payload_str: str*, *aergo: aergo.herapy.aergo.Aergo*, *receiver: str*, *total_supply: int*, *fee_limit: int*, *fee_price: int*) → str

Deploy a token contract payload and give the total supply to the deployer

aergo_wallet.wallet_utils.**build_deposit_proof**(*aergo_from: aergo.herapy.aergo.Aergo*, *aergo_to: aergo.herapy.aergo.Aergo*, *receiver: str*, *bridge_from: str*, *bridge_to: str*, *deposit_height: int*, *token_origin: str*, *key_word: str*) → aergo.herapy.obj.sc_state.SCState

Check the last anchored root includes the lock and build a lock proof for that root

aergo_wallet.wallet_utils.**get_balance**(*account_addr: str*, *asset_addr: str*, *aergo: aergo.herapy.aergo.Aergo*) → int

Get an account or the default wallet balance of Aer or any token on a given network.

aergo_wallet.wallet_utils.**transfer**(*value: int*, *to: str*, *asset_addr: str*, *aergo: aergo.herapy.aergo.Aergo*, *sender: str*, *fee_limit: int*, *fee_price: int*) → str

Support 3 types of transfers : simple aer transfers, token transfer, and signed token transfers (token owner != tx signer)

exception aergo_wallet.exceptions.**InsufficientBalanceError**

exception aergo_wallet.exceptions.**InvalidArgumentsError**

exception aergo_wallet.exceptions.**InvalidMerkleProofError**

exception aergo_wallet.exceptions.**TxError**

exception aergo_wallet.exceptions.**UnknownContractError**

1.10 aergo_cli

class aergo_cli.main.**MerkleBridgeCli**(*root_path: str = './'*)
CLI tool for interacting with the AergoWallet.

First choose an existing config file or create one from scratch. Once a config file is chosen, the CLI provides an interface to the AergoWallet and has the following features: - edit config file settings - transfer assets between networks - check status of transfers - check balances for each asset on each network

check_balances()
Iterate every registered wallet, network and asset and query balances.

check_withdrawable_balance()
Check the status of cross chain transfers.

create_config()
Create a new configuration file from scratch.

This tool registers 2 networks, bridge contracts, a private key for each network and bridge validators

edit_settings()
Menu for editing the config file of the currently loaded wallet

finalize_transfer()
Finalize a token transfer between 2 chains.

finalize_transfer_arguments(*prompt_last_deposit=True*)
Prompt the arguments needed to finalize a transfer.

The arguments can be taken from the pending transfers or inputed manually by users.

Returns: List of transfer arguments

get_registered_assets(*from_chain*, *to_chain*)
Get the list of registered assets on each network.

get_registered_networks()
Get the list of networks registered in the wallet config.

initiate_transfer()
Initiate a new transfer of tokens between 2 networks.

load_config()
Load the configuration file from path and create a wallet object.

menu()
Menu for interacting with network.

Users can change settings, query balances, check pending transfers, execute cross chain transactions

prompt_bridge_networks()
Prompt user to choose 2 networks between registered networks.

prompt_commun_transfer_params()
Prompt the common parameters necessary for all transfers.

Returns: List of transfer parameters : from_chain, to_chain, from_assets, to_assets, asset_name, receiver

prompt_signing_key(*wallet_name*)
Prompt user to select a private key.

Note: Keys are displayed by name and should have been registered in wallet config.

prompt_transfer_networks()
Prompt user to choose 2 networks between registered bridged networks.

register_asset()
Register a new asset and it's pegs on other networks in the wallet's config.

register_bridge()
Register bridge contracts between 2 already defined networks.

register_key()
Register new key in wallet's config.

register_network()
Register a new network in the wallet's config.

register_new_validators()
Register new validators in the wallet's config.

start()
Entry point of cli : load a wallet configuration file of create a new one

store_pending_transfers()
Record pending transfers in json file so they can be finalized later.

aergo_cli.utils.**format_amount**(*num: str*)
Format a float string to an integer with 18 decimals.

Example: '2.3' -> 2300000000000000000

aergo_cli.utils.**promptYN**(*q*, *y*, *n*)
Prompt user to procede with a transfer of not.

aergo_cli.utils.**prompt_aergo_keystore**()
Prompt user to register a new aergo keystore.

Returns:

- name of the key
- address of the key
- keystore path

aergo_cli.utils.**prompt_amount**()
Prompt a number of tokens to transfer.

aergo_cli.utils.**prompt_deposit_height**()
Prompt the block number of deposit.

aergo_cli.utils.**prompt_new_asset**(*networks*)
Prompt user to input a new asset by providing the following: - asset name - origin network (where it was first issued) - address on origin network - other networks where the asset exists as a peg - address of pegs

aergo_cli.utils.**prompt_new_bridge**(*net1*, *net2*)
Prompt user to input bridge contracts and tempo.

For each contract on each bridged network, provide: - bridge contract address - anchoring periode - finality of the anchored chain

aergo_cli.utils.**prompt_new_network**()
Prompt user to input a new network's information: - Name - IP/url

aergo_cli.utils.**prompt_new_validators**()
Prompt user to input validators

Note: The list of validators must have the same order as defined in the bridge contracts

Returns: List of ordered validators

aergo_cli.utils.**prompt_number**(*message*, *formator*=<*class 'int'*>)
Prompt a number.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

I

L

M

N

P

R

S

T

U

V

W