aergo-guide Documentation

AERGO team and contributors

Jun 04, 2019

Platform

Aergo is an open blockchain platform that allows businesses to build innovative applications and services by sharing data on a trustless and distributed IT ecosystem. Aergo combines the best of public and private blockchains to be scalable, enterprise-ready, and easy to develop with.

This website contains guides for developers to get started developing with Aergo and detailed references.

CHAPTER 1

Introduction

Aergo is developing a practical blockchain platform, frameworks, libraries, and tools.

1.1 Design goals

There are four main ideologies regarding this project.

- Developer-friendly
- Guaranteed performance
- Scalable architecture
- Open, extensible, inter-connectable network

1.2 Current features

- BFT-dPOS with voting
- Name system
- Aergo Lua smart contract
- Ship (development framework, package management)
- Simple client API
- Client SDKs
- Sub projects
  - Litetree
  - Sparse Merkle Tree
- Hub Enterprise (management and monitoring)

- Merkle bridge verification
- Testnet deployment

1.3 Roadmap

These features are under active development or planned.

- Aergo SQL
- Parallelism (inter-contract)
- Simple branching (2WP or simple Plasma)
- Orchestration with Aergo Horde
- Service with Aergo Hub
- Advanced performance features

CHAPTER 2

Dapp Development

This article gives an introduction into the high level approach to developing dapps on the AERGO blockchain.

2.1 Development cycle

These are components of today's blockchain application development cycle that Aergo offers new solutions for.

1. Design (UX patterns, economic models)
2. Clients (interacting with the blockchain)
3. Smart contracts (new programming patterns, debugging and testing)
4. Infrastructure (deployment cost, security, privacy, monitoring)

2.2 Design

The possibilities of designing applications for Aergo are endless: social networks, distributed organizations, games, monetary systems, asset management, trading, databases, public deliberation, crowdsourcing, ...

Aergo tries to not limit designers' and developers' creativity to create novel applications. However, the platform aims to offer reasonable guidelines, standard protocols, and well-tested models to make designing dapps easier than ever.

2.3 Clients

Using the Hera SDK family, client applications can easily interact with the blockchain.

- Read more

2.4 Smart contracts

Smart contracts are the main backend component of distributed applications. As other blockchain platforms had a serious limitation in terms of cost and scalability, it used to be not feasible to develop large scale, complex systems. With Aergo and the use of embedded sidechains and proof systems, smart contracts can really shine.

- Read more

Aergo also offers a development toolchain that includes package management, Aergo Ship.

2.5 Infrastructure

Aergo is currently available as a public testnet and separate private installations. In the future, a network of blockchains can be orchestrated using Aergo Horde. Aergo Hub will offer an easy configuration and monitoring interface. With these features, developers will not have to worry about the deployment of their applications.

CHAPTER 3

Running a Node

This section includes general instructions for running Aergo nodes.

If you want to connect to an existing Aergo network (such as the public **mainnet and testnet**), please also refer to this section for more details.

3.1 Quickstart

The easiest way to run a local Aergo server is using Docker.

```
docker run -p 7845:7845 aergo/node
```

You can pass arguments to the server like this:

```
docker run -p 7845:7845 aergo/node aergosvr --testmode
```

To supply your own config file, use:

```
docker run -p 7845:7845 -v $(pwd)/config.toml:/aergo/config.toml aergo/node aergosvr -
↪-config /aergo/config.toml
```

Building from source: You can also build the binaries yourself from source. See here for details.

Syncing with the public testnet: Please refer to the testnet guide for more instructions.

3.2 Configuration

This page explains the possible ways to configure an Aergo node.

There are three common modes of operation for aergo server:

- BP Node - Block provider node, which mines and propagates blocks

- Full Node - Node to get and validate all blocks
- Single Node - BP Node without peer, which uses to something like test.

We describe only single node configuration in this. More will updated for the BP Node and full node.

3.2.1 Single Aergo Server (default)

We will show default configuration file next. You can check the each fields' meaning in here

```
# aergo TOML Configuration File (https://github.com/toml-lang/toml)
# base configurations
datadir = ".aergo/data"
enableprofile = false
profileport = 6060
enablerest = false
enabletestmode = false
authdir = ".aergo/auth"

[rpc]
netserviceaddr = "127.0.0.1"
netserviceport = 7845
nstls = false
nscert = ""
nskey = ""
nsallowcors = false

[rest]
restport = "8080"

[p2p]
# Set address and port to which the inbound peers connect, and don't set loopback␣
↪address or private network unless used in local network
netprotocoladdr = ""
netprotocolport = 7846
npbindaddr = ""
npbindport = -1
# Set file path of key file
npkey = ""
npaddpeers = [
]
nphiddenpeers = [
]
npmaxpeers = 100
nppeerpool = 100
npexposeself = true
npusepolaris = true
npaddpolarises = [
]

[blockchain]
# blockchain configurations
maxblocksize = 1048576

[mempool]
showmetrics = false
dumpfilepath = ".aergo/mempool.dump"
```

(continues on next page)

(continued from previous page)

```
[consensus]
enablebp = true
enabledpos = false
blockinterval = 1
dposbps = 23
bpids = [
]
```

3.2.2 Testmode

To enable testmode, either pass the command line option `--testmode` to aergosvr or set `enabletestmode = true` in the configuration.

In testmode, all new accounts are assigned a high number of Aergo tokens by default, basically circumventing balance checks. This means you can send any transaction without first pre-funding hard-coded accounts using a genesis block.

Testmode **MUST NOT** be used in production.

3.2.3 Reference

3.2.4 Logging options

It is possible to customize the log output format of all Aergo CLI tools using a file called `arglog.toml` placed in the current working directory.

This file is specified here.

```
level = "info"  # default log level
formatter = "json"  # format: console, console_no_color, json
caller = true  # enabling source file and line printer
timefieldformat = "RFC3339"

[chain]
level = "info"  # optional, log level for 'chain' module

[dpos]
level = "info"

[p2p]
level = "info"

[consensus]
level = "info"

[mempool]
level = "info"

[contract]
level = "info"

[syncer]
level = "info"
```

(continues on next page)

(continued from previous page)

```
[bp]
level = "info"
```

3.3 Monitoring

Aergo server has 5 major subcomponents: Account, Chain, MemPool, RPC, and P2P.

We can get status of component using aergocli node command (see here for detail).

```
$ aergocli node
{
        "AccountsSvc": {
                "status": "started",
                "acc_processed_msg": 3,
                "msg_queue_len": 0,
                "msg_latency": "69.851µs",
                "error": "",
                "actor": null
        },
        "ChainSvc": {
                "status": "started",
                "acc_processed_msg": 238,
                "msg_queue_len": 0,
                "msg_latency": "68.849µs",
                "error": "",
                "actor": {
                        "orphan": 0
                }
        },
        "MemPoolSvc": {
                "status": "started",
                "acc_processed_msg": 237,
                "msg_queue_len": 0,
                "msg_latency": "59.462µs",
                "error": "",
                "actor": {
                        "cache_len": 0,
                        "dead": 0,
                        "orphan": 0
                }
        },
        "RPCSvc": {
                "status": "started",
                "acc_processed_msg": 120,
                "msg_queue_len": 0,
                "msg_latency": "71.785µs",
                "error": "",
                "actor": null
        },
        "p2pSvc": {
                "status": "started",
                "acc_processed_msg": 120,
                "msg_queue_len": 0,
                "msg_latency": "54.783µs",
```

(continues on next page)

(continued from previous page)

```
                "error": "",
                "actor": null
        }
}
```

As it shown above, each component has 6 values, which are

- status *
- acc_processed_msg
  - number of message processed by the component
  - message is basic unit for communicate with other components
- msg_queue_len
  - number of pending messages for the component
- msg_latency
  - average latency for processing a message
- error *
- actor
  - component specific statistics are shown
  - for example, ChainSvc

    “actor”: { “orphan”: 0 }

    orphan represents number of orphan block the component has

3.4 Configuring a Network

This article explains the steps needed to configure a network of multiple block producers. You can follow this guide to setup a private Aergo blockchain network.

This guide specifically requires no prior setup, so it should be easy to follow along with new bare machines running Ubuntu. It has been tested with AWS EC2 instances. We will setup three block producers, but you can adjust the procedure to any number of BPs.

3.4.1 Per-machine setup

Install Docker

You will need Docker for this setup, so let’s install that now.

```
sudo apt-get update && sudo apt-get install docker.io
sudo usermod -aG docker ubuntu
```

NTP

Timing is critical in a blockchain, so it’s better to configure the machines’ time setup manually:

```
sudo bash
apt-get install chrony
vi /etc/chrony/chrony.conf
```

```
server time1.google.com iburst
server time2.google.com iburst
server time3.google.com iburst
server time4.google.com iburst
```

```
systemctl restart chronyd
chronyc makestep
```

3.4.2 Generate BP accounts and keys

You can use any machine (e.g. your own local machine) for this.

Start by installing aergocli if you haven't already.

Generate accounts

```
aergocli account new --password yourpassword --path genesis
(displays generated address)

aergocli account new --password yourpassword --path bp01
(displays generated address)

aergocli account new --password yourpassword --path bp02
(displays generated address)

aergocli account new --password yourpassword --path bp03
(displays generated address)
```

Export accounts

```
aergocli account export --address [insert address from genesis]  --password␣
↪yourpassword --path genesis
(displays exported private key)

aergocli account export --address [insert address from bp01]  --password yourpassword␣
↪--path bp01
(displays exported private key)

aergocli account export --address [insert address from bp02]  --password yourpassword␣
↪--path bp02
(displays exported private key)

aergocli account export --address [insert address from bp03]  --password yourpassword␣
↪--path bp03
(displays exported private key)
```

Generate peer keys

```
aergocli keygen bp01
Wrote files bp01.{key,pub,id}.
```

(continues on next page)

(continued from previous page)

```
aergocli keygen bp02
Wrote files bp02.{key,pub,id}.

aergocli keygen bp03
Wrote files bp03.{key,pub,id}.
```

3.4.3 Write configuration files

genesis.json (same for all machines)

```
{
    "chain_id":{
        "magic": "[insert an identifier string for your network]",
        "public": false,
        "mainnet": false,
        "consensus": "dpos"
    },
    "timestamp": 1548918000000000000,
    "balance": {
        "[insert address from genesis]": "470000000000000000000000000",
        "[insert address from bp01]": "100000000000000000000000000",
        "[insert address from bp02]": "100000000000000000000000000",
        "[insert address from bp03]": "100000000000000000000000000"
    },
    "bps": [
        "[insert text from bp01.id]",
        "[insert text from bp02.id]",
        "[insert text from bp03.id]"
    ]
}
```

config.toml (one per machine)

```
# aergo TOML Configration File (https://github.com/toml-lang/toml)
# base configurations
datadir = "./data"
enableprofile = true
profileport = 6060
enablerest = true
personal = false

[rpc]
netserviceaddr = "0.0.0.0"
netserviceport = 7845
nstls = false
nscert = ""
nskey = ""
nsallowcors = false

[p2p]
netprotocoladdr = "{LOCAL_IP}"  # Insert IP address from this machine
netprotocolport = 7846
npbindaddr = "0.0.0.0"
npbindport = 7846
nptls = false
```

(continues on next page)

(continued from previous page)

```
npcert = ""
npkey = "bp{01,02,03}.key"  # Name of key file of node
npaddpeers = [
    "/ip4/[IP ADDRESS FROM BP 01]/tcp/7846/p2p/[PEER ID FROM BP 01]",
    "/ip4/[IP ADDRESS FROM BP 02]/tcp/7846/p2p/[PEER ID FROM BP 02]",
    "/ip4/[IP ADDRESS FROM BP 03]/tcp/7846/p2p/[PEER ID FROM BP 03]"
]

npexposeself = false
npusepolaris = false
nphiddenpeers= [
    "[PEER ID FROM BP 01]",
    "[PEER ID FROM BP 02]",
    "[PEER ID FROM BP 03]"
]

[blockchain]
usefastsyncer = true
blockchainplaceholder = false
coinbaseaccount = "[ADDRESS FROM THIS PEER]"

[mempool]
showmetrics = true
dumpfilepath = "./data/mempool.dump"

[consensus]
enablebp = true
```

3.4.4 Running

We are going to use the Docker image aergo/node to run the server. Please refer to the Docker documentation for learn about the available run options.

Check the directory contents

After following the above steps, you should now have these files in one directory for each machine. In this example we use the path /blockchain, but you can use any directory as long as you substitute its local path in the Docker run commands below.

```
/blockchain/
    bp{01,02,03}.key
    genesis.json
    config.toml
```

Create genesis block

```
docker run --rm \
    -v /blockchain:/aergo \
    aergo/node \
    aergosvr init --genesis /aergo/genesis.json --home /aergo --config /aergo/config.
↪toml
```

Start the node

```
docker run -d --log-driver json-file --log-opt max-size=1000m --log-opt max-file=7 \
    -v /blockchain:/aergo \
    -p 7845:7845 -p 7846:7846 -p 6060:6060 \
    --restart="always" --name aergo-node \
    aergo/node \
    aergosvr --home /aergo --config /aergo/config.toml
```

3.4.5 Further reading

You may now want to setup further full nodes as well as Polaris for automatic node discovery.

3.5 Advanced Topics

3.5.1 Genesis block configuration

The genesis block is the first block created on a new chain according to a predefined specification. If you want to bootstrap your own Aergo blockchain, you will need to configure and create the genesis block. All nodes producing blocks and syncing with the network need to use the same genesis block.

You can find an example genesis configuration json file here. It is the one used for the public testnet.

Key	Type	Explanation
chain_id	obj	
– magic	string	A name identifier used to distinguish different chains. Example: testnet.aergo.io
– public	bool	If this chain is for a public (true) or private network. This may affect certain features.
– mainnet	bool	If this is the main network (true) or a sidechain for another network.
– consensus	string	This blockchain's consensus type (dpos)
– coinbasefee	string	The default transaction fee amount
timestamp	number	The genesis block's creation timestamp in nanoseconds
balance	obj	A mapping of addresses and allocated balances
bps	list	A list of fallback block producer peer ids.

3.5.2 Manually creating peer identification

Every peer (i.e. instance of aergosvr) is identified by a peer id, derived from its public key. If unspecified, aergosvr creates these automatically, but you may want to do it manually to control the generation of keys and make backups. You can use the CLI to do that:

```
aergocli keygen your-keyname

# Using Docker:
docker run --rm -v $(pwd):/tools/ aergo/tools aergocli keygen your-keyname
```

You then need to specify the path to the generated your-keyname.key file in the Aergo configuration:

```
...
[p2p]
npkey = "/aergo/your-keyname.key"
...
```

3.6 Troubleshooting

3.6.1 Storage errors

DB initilaize failure

Aergo uses Badger DB as its internal storage. If the DB creation fails for Badger at initial boot, boot process will fail.

In this case, you can see the following error when checking the server's log.

```
failed to initialize chaindb
failed to initialize statedb
```

You need to make sure that you have write access to the path you want to create the data directory on the config file If the database server was rebooted while it was running, it is possible that the DB datafile has been crashed. In this case, a reinstallation is required.

check previously running processes

The DB initialize problem is most likely to occur when a new aergosvr is run without shutting down the existing aergosvr completely. In this case, you can check with the following log.

```
panic: Fail to Create New DB: Cannot acquire directory lock on "config data directory
↪".
Another process is using this Badger database.: resource temporarily unavailable`
```

In this case, terminate the existing aergosvr completely and restart it.

Miscellaneous DB errors

If an error occurs in DB, the following log is output.

```
Database Error:  description...
```

In this case, it's a good idea to first check to see if there is a problem with the filesystem .If there is no problem with the filesystem, the DB datafile can be seen as corrupt. In this case, reinstallation is required.

Disk performance problem

The BP node generates a block every 1 second. When a block is created, it is stored in the DB and Disk I / O occurs at this time.

If the disk I / O is too slow, the block created by the BP will not be added to the chain. This is because the block generated by the node is propagated to another node but the block of the corresponding height is already generated in the next BP. Therefore, the block arriving late comes to be discarded.

Check the disk I / O execution time through dd. This allows you to check for disk errors.

3.6.2 Network trouble

peer connection failure

If there is a problem with the network configuration, the peer connections may not be normal. you can check peer connections as follows.

1. You need to make sure that the entire peer set in the config is connected. If there is an unconnected peer, check the network configuration with the peer.

   You can check the connection status of peers with the following command:

```
aergocli -p $ServerPort getpeers
[
        {
                "Address": "127.0.0.1",
                "PeerID": "16Uiu2HAkxUPj8zUnSkBzP8f4mw9WvJ4s5DeMVarzsqBuyrDy6dmD",
                "Port": "11002",
                "State": "RUNNING"
        }
]
```

network performance problem

aergo uses DPOS as a consensus algorithm. Blocks generated by one node must be transmitted to other nodes at a high speed. If the block transmission is too slow due to the network problem, the block created by the BP will not be included in the chain.

You can track the block creation time and the time passed to other nodes.

```
Oct 30 15:50:57.009 |INFO| block produced␣
↪BP=16Uiu2HAmF5fpuk9ZWQbzHUuu9C1jwtqKnZdtkQYYDapA7AyoibWP caller=/Users/everjs/go/
↪src/github.com/aergoio/aergo/consensus/impl/dpos/blockfactory.go:213 confirms=2␣
↪id=6mdjTYFEKXJ4UQ3xdKEu5RybnVWh3Hio6VG8a5SAXPbp lpb=1356 module=dpos no=1358␣
↪sroot=9hbmSFxzGj58YQuM2642zyo8jDAuu5qTWNG8JCMb78iZO`
```

```
Oct 30 15:50:57.012 |INFO| added block successfully. best=1358
```

```
Oct 30 15:50:57.013 |DEBUG| add block chainservice blockNo=1358 caller=/Users/ev
erjs/go/src/github.com/aergoio/aergo/chain/chainservice.go:252␣
↪hash=6mdjTYFEKXJ4UQ3xdKEu5RybnVWh3Hio6VG8a5SAXPbp module=chain
```

3.6.3 Server Process Hang

The aergo server was developed using the proto actor framework. Each module in the server receives a request using a message, performs an action on it, and returns a response.

At this time, if a specific module does not operate normally and hangs, it can be checked with the following command.

```
aergocli -p 10001 node
{
        "AccountsSvc": {
                "status": "started",
                "acc_processed_msg": 202,
                "msg_queue_len": 0,
                "msg_latency": "55.623µs",
                "error": "",
```

(continues on next page)

(continued from previous page)

```
                "actor": null
        },

...

}
```

If a particular module hangs, the value of **msg_queue_len** field is high.

The details can be analyzed through the log by increasing the log level of the module.

3.6.4 Tracking status of Transaction

A transaction is added to a mempool and then executed when it is included in a block. Transactions are not included in block if validation fails due to nonce, balance, etc.. All transactions in the block will have a receipt. If the VM fails to execute contract, the error is recorded in the receipt. Therefore, the state of transaction may be as follows.

- State of transaction
  - pending in mempool
  - included in block
    - sucess
    - fail if contract

check status of transaction

Use the following command to track the status of the transaction.

```
aergocli -p $PORT gettx $TXHASH
```

If the transaction stays in the pending state in the mempool, the following status is output.

```
aergocli -p $PORT gettx $TXHASH
Pending:  {
 "Hash": "BTY3zrsBkVHTqFbpqKtwbXwgfpGwRoiAmKRrdFKfZRna",
 "Body": {
  "Nonce": 101,
  "Account": "AmM5wCL3XhDngGAyYk6pbd2AYKSomVAKRqLBhnP8QsBMRRoA62fv",
  "Recipient": "AmNeY5fDqvGAfpYYLxhy7AhSsFzCSL96iYBApR2aHvRzMY5D34Ky",
  "Amount": 2560000,
  "Payload": "",
  "Limit": 0,
  "Price": 0,
  "Type": 0,
  "Sign":
↪"381yXZRAPSGAFbGYndN5xU3s9RXHddFbCxyutNtUYvyT4DbDt7jWyia4oM6CLGtXYeQqtQBvqTaNU2gwpdrCm5FjwFP6fZCQ
↪"
 }
}
```

check account

First, check the nonce and balance of the corresponding account. The status of the current account can be seen by the command below.

```
aergocli -p $PORT  getstate --address $ADDRESS
```

Ensure that the nonce and balance of transaction are consistent with the account status.

check status of mempool

There are too many transactions in mempool, which can take a long time to be included in the block. To see the current status of mempool, try the following command.

```
aergocli -p $PORT  node
```

In the **cache_len** field, you can see the total number of tx's in mempool.

check receipt

If the transaction is contained in a block, you can use the receipt to check the execution result. Receipt can be seen using transaction hash

```
aergocli -p $PORT receipt get $TXHASH
{
 "contractAddress": "AmNeY5fDqvGAfpYYLxhy7AhSsFzCSL96iYBApR2aHvRzMY5D34Ky",
 "status": "CREATED",
 "ret": ""
}
```

CHAPTER 4

Using Public Networks

These pages explain how you can connect to and sync with public Aergo networks.

4.1 Connecting to Well-known Nodes

4.1.1 Testnet

Use *testnet.aergo.io* and the default port 7845 to connect to a public GRPC server for the testnet.

```
docker run --rm aergo/tools aergocli -H testnet-api.aergo.io blockchain
```

4.1.2 Mainnet

Use *mainnet-api.aergo.io* and the default port 7845 to connect to a public GRPC server for the mainnet.

```
docker run --rm aergo/tools aergocli -H mainnet-api.aergo.io blockchain
```

4.2 Syncing

This setup assumes some basic knowledge about using Docker. Using Docker is the recommended way to run an Aergo node as it requires minimal configuration and enables you to use Docker features such as logging and restart policies.

Note: When you run the Docker image named *aergo/node* for the first time, Docker downloads the latest image automatically. To update the image to the latest version, run `docker pull aergo/node`. You can also specify a specific version by replacing `aergo/node` with, for example, `aergo/node:1.0`. Refer to Docker Hub to see a list of available tags.

4.2.1 Running

This is the main command to run a node syncing with the main Aergo network:

```
docker run -v $(pwd)/data:/aergo/data -p 7845:7845 -p 7846:7846 aergo/node
```

The `-p` argument maps ports from the aergo server inside the container to your host machine. 7846 is for the peer to peer protocol. 7845 is for the RPC API for connecting clients to the server; you can remove this port binding to disallow access to the API.

When running this command for the first time, it will create the default genesis block (block number 0). Afterwards it automatically starts synchronizing.

Note: If your machine is behind a NAT (such as a router), you need to setup manual forwarding of the port 7846 to allow other peers to sync with your node.

Testnet

To sync with the testnet instead, use the testnet flag:

```
docker run -v $(pwd)/data:/aergo/data -p 7845:7845 -p 7846:7846 aergo/node aergosvr --
↪home /aergo --testnet
```

4.2.2 Configuration

The above commands use default configuration files suitable to connect to officially supported networks. If you want to override configuration parameters, you can supply a custom config file using a Docker volume, for example:

```
docker run -v $(pwd)/data:/aergo/data -v $(pwd)/config.toml:/aergo/config.toml -p␣
↪7846:7846 aergo/node aergosvr --home /aergo --config /aergo/config.toml
```

Please refer to Configuration for a detailed explanation of the available settings.

Note: To customize the log format, place an arglog.toml file in the container's working directory: `-v $(pwd)/arglog.toml:/aergo/arglog.toml`.

Server key file

Every node is identified by a key and peer id. In the default configuration, a key is generated every time you launch the docker container. This may not be what you want. You can persist the generated key by adding a volume `-v $(pwd)/auth:/aergo/auth`. You can also supply the key manually by placing *aergo-peer.key* and *aergo-peer.id* files in that volume.

Refer to Configuration for details about server keys.

Tip: Instead of binding several volumes, you can use one combined volume, e.g. `-v $(pwd)/aergo:/aergo`.

Node discovery

Without any specific settings, the server connects to Polaris, registers itself, obtains addresses of other nodes, and automatically attempts to connect to those nodes. If the server is in a NAT environment or has multiple NICs, additional settings are required for external nodes to access the server. You have to setup the external address in the external network connection and set the internal address for binding address.

```
netprotocoladdr = "211.12.34.56" # external address to which other peer can connect
netprotocolport = 7846
npbindaddr = "192.168.0.2" # no config element or empty string means using same␣
↪address as external
npbindport = 17846 # negative number means it is same as external port, in this case␣
↪7846
```

4.3 Creating Accounts

Accounts are identified by addresses that belong to private keys. To own an account, all you need is access to its private key. Always make sure that nobody except you gains access to your keys!

There are several methods to create accounts.

1. If you are **running your own node** on your machine, you can create a local account:

   ```
   aergocli account new

   # Or using Docker:
   docker run --rm --net=host aergo/tools aergocli account new
   ```

2. To create an account **without a node** (aka offline):

   ```
   aergocli account new --password your_password --path ./wallet

   # Or using Docker:
   docker run --rm -v $(pwd):/wallet aergo/tools aergocli account new --password␣
   ↪your_password --path /wallet
   ```

   The output shows the address of your new account. It is saved in account/data.db in the given path.

   You can export this new account:

   ```
   aergocli account export --address your_address --password your_password --path ./
   ↪wallet

   # Or using Docker:
   docker run --rm -v $(pwd):/wallet aergo/tools aergocli account export --address␣
   ↪your_address --password your_password --path /wallet
   ```

   The output shows the encrypted private key belonging to your address which can be imported in other wallets.

4.4 Funding Accounts

Note: The method described here only applies to the testnet.

To create transactions on the public testnet, you need an account with a positive balance. After creating an account, this is how can fund your account for testing.

1. Go to faucet.aergoscan.io, enter your account address, and click Request Tokens.
2. Depending on demand, you need to wait for a moment. If the queue is full, try again a bit later.
3. Check your account on Aergoscan. You should have received some AERGO tokens for your testing purposes.

These AERGO tokens are only valid on the testnet and cannot be sold or bought. They are only useful for developing and testing applications. Please only request as many tokens as you actually need.

If you have any trouble creating an account or receiving testnet tokens, please ask for help on our Discord channel.

4.5 Monitoring

Aergoscan is an explorer for the officially supported networks.

- Testnet Aergoscan
- Mainnet Aergoscan

You can monitor blocks as they get produced and check transactions and accounts.

4.6 Snapshots

To quickly bootstrap a new full node, you can sart from a snapshot.

1. Download the latest snapshot from snapshot.aergo.io.
2. Extract the archive, e.g. `tar zxvf 672571_20190423.tar.gz`.
3. Start the node, pointing it to the extracted data directory: `docker run -v $(pwd)/data:/aergo/data -p 7845:7845 -p 7846:7846 aergo/node`

CHAPTER 5

Smart Contracts

Aergo provides its own smart contract platform for implementing various business logic on the blockchain.

Currently, you can create smart contracts using the Lua programming language.

5.1 Lua

To create smart contracts for Aergo, you can use the Lua scripting language.

Lua is a powerful, efficient, lightweight, embeddable scripting language. It has a simple procedural syntax with a powerful data description structure. Lua supports a variety of programming methods: procedural programming, object-oriented programming, functional programming.

We use LuaJIT 2.1.0 as the VM. LuaJIT is a Just-In-Time Compiler (JIT) for the Lua programming language.

You can learn the Lua programming language through the following documents:

- Lua 5.1 Reference Manaul
- Programming in Lua (The second edition was aimed at Lua 5.1)

Table of contents

5.1.1 Hello World

This is the most basic lua smart contract to store and retrieve states in aergo. You can save a name on the blockchain with the contract call function. And you can print 'hello . . . ' with the query function.

```
-- Define global variables.
state.var {
  Name = state.value(),
  My_map = state.map()

}
```

(continues on next page)

(continued from previous page)

```
-- Initialize a name in this contract.
function constructor()
  -- a constructor is called at a deployment, only one times
  -- set initial name
  Name:set("world")
  My_map["key"] = "value"
end

-- Update a name.
-- @call
-- @param name          string: new name.
function set_name(name)
  Name:set(name)
end

-- Say hello.
-- @query
-- @return              string: 'hello ' + name
function hello()
  return "hello " .. Name:get()
end

-- register functions to expose
abi.register(set_name, hello)
```

Tutorial

This is explained based on using cli. Variables used in this example are

- Account to deploy and execute a contract: `AmPbWrQbtQrCaJqLWdMtfk2KiN83m2HFpBbQQSTxqqchVv58o82i`

Check Account and Balance

First, you need an account with enough balances to deploy and execute smart contracts. (If you don't) Import or Unlock Account to aergo server.

Compile Contract

Copy above code and save it to a file (e.g. helloword.lua). And Compile using the `aergoluac` compiler

```
./aergoluac --payload helloworld.lua
37mGLDoCPNDQw7HbCG5WPfcM3E3cLhqhgE2V2UJKwQp9QZ5nJhT14nkCdGFcmN91fewB2ZuMZ5NWJUPyD4G4G2beaTeE1cigLzyNc
```

Deploy Contract

With the payload generated above, Deploy contract

```
./aergocli contract deploy AmPbWrQbtQrCaJqLWdMtfk2KiN83m2HFpBbQQSTxqqchVv58o82i --
↪payload 37mGLDoCPNDQw7HbCG5WPfcM3E3cLhqhgE2V2UJK
wQp9QZ5nJhT14nkCdGFcmN91fewB2ZuMZ5NWJUPyD4G4G2beaTeE1cigLzyNdGuuU4Y7cY2A6MUMq5weoAGGJdyf6PUfzgQ7k1cw
1 : FPqA3kNQHoVXqKJv8JNpUSsh8F8id87yvRr5UzQFoCcH TX_OK
```

Get receipt of contract

Look up the actual contract address with the transaction ID above.

```
./aergocli receipt get FPqA3kNQHoVXqKJv8JNpUSsh8F8id87yvRr5UzQFoCcH
{
 "BlokNo": 317,
 "BlockHash": "48zceVwBZt5dpzuEFMtJB9icPXUu7YG1Xkxvw5N92yFW",
 "contractAddress": "AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU",
 "status": "CREATED",
 "ret": {},
 "txHash": "AWeaoCTpohuQpBMTFaW3qFpZqWwuTehXA8ZkAX59UjMV",
 "txIndex": 0,
 "from": "AmPbWrQbtQrCaJqLWdMtfk2KiN83m2HFpBbQQSTxqqchVv58o82i",
 "to": "",
 "usedFee": 10,
 "events": []
}
```

If the status is not 'CREATED', it may not be included in the block yet, or there may be an error. Wait a while until the transaction is included in the block. Or check the server's error log.

Get ABI of contract

Look up ABI of contract with the contract address above.

```
./aergocli contract abi AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU
{
 "version": "0.2",
 "language": "lua",
 "functions": [
  {
   "name": "hello"
  },
  {
   "name": "set_name",
   "arguments": [
    {
     "name": "name"
    }
   ]
  },
  {
   "name": "constructor"
  }
 ],
 "state_variables": [
  {
   "name": "Name",
```

(continues on next page)

(continued from previous page)

```
    "type": "value"
   },
   {
    "name": "My_map",
    "type": "map"
   }
  ]
}
```

Query Initial State

You can query the generated contract in the following way.

```
./aergocli contract query AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU hello
value:"\"hello world\""
```

You can see that the name 'world' assigned by the constructor is output.

Call Contract

You can change the name recorded in the block chain as follows:

```
./aergocli contract call AmPbWrQbtQrCaJqLWdMtfk2KiN83m2HFpBbQQSTxqqchVv58o82i␣
↪AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU
 set_name '["aergo"]'
1 : 8mcuEFNxNCF6h4Q3FJk3mGN356R1AmWgGptAgJHNfaKs TX_OK
```

Query Changed State

If you look at the results again, it has changed.

```
./aergocli contract query AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU hello
value:"\"hello aergo\""
```

Query contract variable with merkle proof

Value

```
./aergocli contract statequery AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU␣
↪Name --compressed
```

Map

```
./aergocli contract statequery AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU␣
↪My_map key --compressed
```

Array

```
./aergocli contract statequery AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU␣
↪array_name array_index --compressed
```

By default, the returned state is the one at the latest block, but you may specify any past block's state root.

```
./aergocli contract statequery AmhbdCEg4TUFm6Hpdoz8d81eSdzRncsekBLN3mYgLCbAVdPnu1MZ␣
↪var_name --root "9NBSjkcNTdE5ciBxfb52RmsVW7vgX5voRsv6KcosiNjE"
```

5.1.2 Programming Guide

Lua provides useful functions and libraries. So you can easily create smart contracts using these functions.

Please refer to the Lua Reference Manual for detailed syntax, explanation, basic built-in functions and libraries.

Restrictions

Because Lua smart contract is performed in the aergo, OS related functions including Input/Output are not provided for stability and security.

Here is a list of the base functions that are not available.

> print, dofile, loadfile, module, and require

You can replace `print` with `system.print`.

And you can use `import` instead of `require`. `import` is not a Lua syntax.

Use SHIP to build and deploy smart contracts using multiple files.

Here is a list of the default libraries that are not available.

> coroutine, io, os, debug, and jit

The `string`, `math`, `bit`, and `table` packages are available. However, you can not use the `random`, `randomseed` functions in the `math` package.

There are no restrictions on literals, expressions, and statements.

During the early period, the execution of a contract has some limitations.

- The maximum number of instruction can be executed per contract is 5000000
- The maximum size of the memory can be used per contract is 10 MB
- The maximum size of the state DB that can be modified per contract is 200 KB

If the maximum is exceeded, the execution of a contract will fail.

Libraries

We provide libraries for smart contract as follows:

- Blockchain API
  - system module
  - contract module
  - state module
  - db module
  - abi module
- Utils

- json module
- crypto module
- bignum module

You can find detailed descriptions for libraries on this page

Smart Contract

Layout

```
import "./path/to/library"

state.var {
  Value = state.value(),
  Map = state.map(),
  Array = state.array(10)
}

function constructor(init_value)
  Value:set(init_value)
end

function contract1(name, id)
  Map["name"] = name
  Map["ID"] = id
end

function contract2()
  local sum = 0
  for i, v in state.array_pairs(Array) do
    if v ~= nil then
      sum = sum + v
    end
  end
  return sum
end

abi.register(contract1, contract2) -- , contract3, ...
```

import

This replaces the `require` function.

It allows you to divide and develop one smart contract into multiple modules(files).

This is not a Lua feature. You should use SHIP to build and deploy smart contracts using multiple files.

state variable

The `state.var` function defines global state variables.

Three types of state variables can be defined.

value

This type store any Lua values.

You can define a state value with the syntax `var_name = state.value()`. It has `get` and `set` methods for reading and writing data.

```
Value:set("data")
local data = Value:get()
```

map

The type map implements associative arrays.

It can be indexed only with `string`, but the value of a map element can be of any type.

You can define a state map with the syntax `var_name = state.map()`. The index operator is used for reading and inserting elements.

```
state.var {
  Map_var = state.map()
}

function contract_func()
  Map_var["name"] = "kslee"
  Map_var["age"] = 38
  -- ...
  local age = Map_var["age"]
end
```

array

The type array is a fixed-length ordinary array.

It can be indexed only with `integer`, but the value of an array element can be of any type. The index starts at 1.

You can define a state array with the syntax `var_name = state.array(size)`. The index operator is used for reading and inserting elements.

```
state.var {
  Arr_var = state.array(3)
}

function contract()
  Arr_var[1] = 1
  Arr_var[2] = 2
  Arr_var[3] = 3

  local sum1 = 0
  for i, v in state.array_pairs(Array) do
    if v ~= nil then
      sum1 = sum1 + v
    end
  end
```

(continues on next page)

(continued from previous page)

```
  local sum2 = 0
  for i = 1, #Arr_var do
    if Arr_var[i] ~= nil then
      sum2 = sum2 + Arr_var[i]
    end
  end

  if sum1 == sum2 then
  -- ...
  end

end
```

Note: The state variables are just syntax sugar that replace `system.getItem(), system.setItem()` functions.

The fields of state variables that are directly modified cannot update the state db.

See `InvalidUpdateAge()` and `ValidUpdateAge()` functions in the example.

```
state.var{
   Person = state.value()
}

function constructor()
  Person:set({ name = "kslee", age = 38, address = "blahblah..." })
end

function InvalidUpdateAge(age)
  Person:get().age = age
end

function ValidUpdateAge(age)
  local p = Person:get()
  p.age = age
  Person:set(p)
end

function GetPerson()
  return Person:get()
end

abi.register(InvalidUpdateAge, ValidUpdateAge, GetPerson)
```

constructor

The `constructor` is executed only once during deployment. It can has arguments. It does not need to register into the `abi.register()` function because it is handled automatically.

functions

Write business logic and help functions.

export contract function(s)

You should add global functions that must be called from contract call/query commands to the `abi.register()`.

special functions

default

`default` is a special function. It is called when the function name can not be found or when the transaction has no a call information. It does not need to export through `abi.register()`. `default` is the name of this function. `default` is used internally by the VM. You should not use `default` for any other purpose.

You can define a default function as follows:

```
...
function default()
  ...
end
...
```

You can call this default function. There is no call information for the contract function.

```
./aergocli contract call <sender> <contract>
```

payable

The `payable` is a property of a function. Only payable function can receives Aergo(s) sent from a sender. We can make a payable function using `abi.payable()`. `payable` functions are automatically exported. Therefore, you do not have to register using the `abi.register` function. `constructor` and `default` are not payable functions by default. They can be payable functions using `abi.payable()`.

You can call the ReceiveAergo with aergo, But you can not call the NotReceiveAergo:

```
...
function ReceiveAergo()
  ...
end

function NotReceiveAergo()
  ...
end

abi.register(NotReceiveAergo)
abi.payable(RecieveAergo)
```

```
./aergocli contract call --amount=10 <sender> <contract> ReceiveAergo     # success
./aergocli contract call --amount=10 <sender> <contract> NotReceiveAergo  # fail
```

view

The `view` is a property of a function. Functions can be declared view in which case they promise not to modify the state(send aergo, emit event, set state, etc...). We can make a view function using `abi.register_view()`.

`register_view` functions are automatically exported. Therefore, you do not have to register using the `abi.register`.

you can not call the sendAergo:

```
...
function sendAergo()
  contract.send(addr, "1 aergo")
end

abi.register_view(sendAergo)
```

```
./aergocli contract call <sender> <contract> sendAergo  # fail
```

SQL

Aergo smart contract has `db` library that supports SQL features.

> Note: The db package is only available on private networks(SQL TestNet).

The below code is a example of creating table and insert a row using `db.exec()`

```
-- creates a customer table
function createTable()
  db.exec([[create table if not exists customer(
        id text,
        passwd text,
        name text,
        birth text,
        mobile text
    )]])
end

-- insert a row to the customer table
function insert(id, passwd, name, birth, mobile)
  db.exec("insert into customer values ('" .. id .. "', '"
      .. passwd .. "', '"
      .. name .. "', '"
      .. birth .. "', '"
      .. mobile .. "')")
end
```

The `db.query()` function returns a result set. You can fetch rows from the result set.

```
function query(id)
  local rt = {}
  local rs = db.query("select * from customer where id like '%'" .. id .. "'%'")
  while rs:next() do
    local col1, col2, col3, col4, col5 = rs:get()
    local item = {
        id = col1,
        passwd = col2,
        name = col3,
        birth = col4,
        mobile = col5
    }
    table.insert(rt, item)
```

(continues on next page)

(continued from previous page)

```
  end
  return rt
end
```

You can also use prepared statements. The following examples is rewrite `insert` and `query` contract functions using prepared statements.

```
function insert(id , passwd, name, birth, mobile)
  stmt = db.prepare("insert into customer values (?, ?, ?, ?, ?)")
  stmt.exec(id, passwd, name, birth, mobile)
end

function query(id)
  local rt = {}
  local stmt = db.query("select * from customer where id like '%' || ? || '%'")
  local rs = stmt:query(id)
  while rs:next() do
    local col1, col2, col3, col4, col5 = rs:get()
    local item = {
        id = col1,
        passwd = col2,
        name = col3,
        birth = col4,
        mobile = col5
    }
    table.insert(rt, item)
  end
  return rt
end
```

Restrictions

Litetree is used as the SQL processing engine for the aergo smart contract. Litetree is implemented based on SQLite.

Detailed SQL usage can be found at https://sqlite.org/lang.html and https://sqlite.org/lang_corefunc.html

However, we do not provide full SQL functionality. There are some limitations due to stability and security.

Data types

Allow only SQL datatypes corresponding to Lua strings and numbers(int, float).

- text
- integer
- real
- null
- date, datetime

SQL statements

You can execute the following SQL statements. However, DDL and DML can not be run on smart contract queries.

- DDL
  - TABLE: ALTER, CREATE, DROP

  - VIEW: CREATE, DROP
  - INDEX: CREATE, DROP
- DML
  - INSERT, UPDATE, DELETE, REPLACE
- Query
  - SELECT

functions

Here is a list of functions that are not available:

- `load_XXX` functions
- `random` function
- `sqlite_XXX` functions
- data and time related functions can be used, except `now` *timestring* and `localtime` *modifier*.

A list of other functions and descriptions is available via the links below.

- basic : https://www.sqlite.org/lang_corefunc.html
- data and time : https://www.sqlite.org/lang_datefunc.html
- aggregation : https://www.sqlite.org/lang_aggfunc.html

contraints

You can use the following contraints.

- NOT NULL
- DEFAULT
- UNIQUE
- PRIMARY KEY (FOREIGN KEY)
- CHECK

Tools

aergoluac

`aergoluac` is a compiler for Lua smart contracts.

- Reference

aergocli

`aergocli` is a command line tool that interfaces with the GRPC exposed by `aergosvr`.

It provides smart contract-related commands as follows:

- contract deploy/call/query/abi/statequery
- receipt get
- event list/stream

- Reference

brick

Toy for Contract Developers. You can use it to test smart contracts.

https://github.com/aergoio/aergo/tree/master/cmd/brick

Style conventions

It is good adopt a consistent coding style for readability. We recommend the Lua style guide.

5.1.3 Reference

Overview

aergo smart contract uses Lua, a lightweight scripting language, as a smart contract language. The following is an example of a simple coin stack smart contract written in Lua that stores key-value values in a block-chain state store and reads the values.

```
-- Storing key-values in the state store
function set(key, value)
    system.setItem(key, value);
end

-- Returns the value corresponding to the key in the state store
function get(key)
    return system.getItem(key);
end

-- Output the hash value of the current block. The output is printed on the server␣
↪log.
function printBlock()
    system.print(system.getBlockhash());
end

-- Functions to be called for contract declare as abi
abi.register(set, get, printBlock)
```

To read and write data on a block chain within a contract, you must use a system package. In the above example, smart contract provides the function to access the key-value repository through the setItem and getItem functions of the system package and store the data permanently. In addition, a simple debug message can be output to the log file of the node via the print function.

system package

This packages provides blockchain information and store/get state

getSender()

This function returns caller address of current contract

getBlockheight()

This function returns the block number that contains the current contract transaction.

getTxhash()

This function returns the id of the current contract transaction.

getTimestamp()

This function returns the creation start time of the block that contains current contract transaction.

getContractID()

This function returns the current contract address.

setItem(key, value)

This function sets the value corresponding to key to the storage belonging to current contract

- restriction
  - key type string only (number type is implicitly converted to string)
  - value type available : number, string, table

getItem(key)

This function returns the value corresponding to key in storage belonging to current contract

- If there is no value corresponding to key, it returns nil.

getAmount()

This function returns number of AER sent with contract call. Return type is string.

getCreator()

This function returns creator address of current contract

getOrigin()

This function returns sender address of current transaction

getPrevBlockHash()

This function returns the hash of the previous block

print(args...)

This function print args with json format at console log in node running current contract

contract package

This packages provides contract operation

send(address, amount)

This function transfers the coins in this contract by address and amount(in AER units). Amount form can be string, number, bignum.

```
contract.send("Amh4S9pZgoJpxdCoMGg6SXEpAstTaTQNfQdZFsE26NpkqPwmaWod", 1)
contract.send("Amh4S9pZgoJpxdCoMGg6SXEpAstTaTQNfQdZFsE26NpkqPwmaWod", "1 aergo 10 gaer
↪")
contract.send("Amh4S9pZgoJpxdCoMGg6SXEpAstTaTQNfQdZFsE26NpkqPwmaWod", bignum.number(
↪"999999999999999"))
```

deploy(code, args...)

The deploy function creates a contract account using code and args, and returns the corresponding address and the return of the constructor function

- If you use an existing address instead of code, deploy it with the code of the address.
- In addition, can call the value function to send a coin. Value function can get string, number, bignum argument like send function.

```
    src = [[
        function hello(say)
            return "Hello " .. say .. system.getItem("made")
        end
        function constructor(check)
            system.setItem("made", check)
        end
        abi.register(hello)
        abi.payable(constructor)
    ]]
    addr = contract.deploy.value("1 aergo")(src, system.getContractID())
```

call(address, function_name, args...)

The call function returns the result of the function of the contract being executed in the state of the corresponding address.

- In addition, can call the value function to send a coin. Value function can get string, number, bignum argument like send function.

```
contract.call("Amh4S9pZgoJpxdCoMGg6SXEpAstTaTQNfQdZFsE26NpkqPwmaWod", "inc", 1)
contract.call.value(10)("Amh4S9pZgoJpxdCoMGg6SXEpAstTaTQNfQdZFsE26NpkqPwmaWod", "inc",
↪ 2)
```

Restrictions

Before the fee system for smart contract is applied to mainnet, there are the following restrictions

- limit the call depth of call or delegatecall in one transaction to 5

delegatecall(address, function_name, args...)

The delegatecall function returns the result of the function of the calling process, executed in the state in the current address.

```
contract.delegatecall("Amh4S9pZgoJpxdCoMGg6SXEpAstTaTQNfQdZFsE26NpkqPwmaWod", "inc",␣
↪1)
```

Restrictions

Before the fee system for smart contract is applied to mainnet, there are the following restrictions

- limit the call depth of call or delegatecall in one transaction to 5

pcall(fn, args...)

It is an error handling function that works just like pcall in lua. The difference is that when the error occurs, the modified state,table or balance of the function executed rollback

```
success = contract.pcall(contract.send,
↪"Amh4S9pZgoJpxdCoMGg6SXEpAstTaTQNfQdZFsE26NpkqPwmaWod", "1 AERGO")
if success == false then
    return 0
end
return 1
```

balance(address)

This function return balance of the address(argument) in AER. return type is string. If address is nil then return balance of current address.

```
contract.balance() --get balance of current contract address
contract.balance("Amh4S9pZgoJpxdCoMGg6SXEpAstTaTQNfQdZFsE26NpkqPwmaWod")
```

event(eventName, args...)

This function causes eventName and args to remain in the contract result receipt. The user can search for event and receive notification the event with rpc when the receipt is added to the blockchain.

```
contract.event("send", 1, "toaddress")
```

Restrictions

Before the fee system for smart contract is applied to mainnet, there are the following restrictions

- limit length of event name to 64
- limit size of event argument to 4k
- limit the number of events in one transaction to 50

Search event and receive notification with aergocli

you can search for events with event name "send" in the contract address(AmhbdCEg4TUFm6Hpdoz8d81eSdzRncsekBLN3mYgLCbAVdPnu1MZ)

```
./aergocli event list --address AmhbdCEg4TUFm6Hpdoz8d81eSdzRncsekBLN3mYgLCbAVdPnu1MZ -
↪-event send
```

you can search for events with event argument 0 is 1 and argument 1 is "toaddress" in the contract address(AmhbdCEg4TUFm6Hpdoz8d81eSdzRncsekBLN3mYgLCbAVdPnu1MZ)

```
./aergocli event list --address AmhbdCEg4TUFm6Hpdoz8d81eSdzRncsekBLN3mYgLCbAVdPnu1MZ -
↪-argfilter '{"0":1, "1":"toaddress"}'
```

you can get notified for events with event name "send" for contract(AmhbdCEg4TUFm6Hpdoz8d81eSdzRncsekBLN3mYgLCbAVdPnu

```
./aergocli event stream --address␣
↪AmhbdCEg4TUFm6Hpdoz8d81eSdzRncsekBLN3mYgLCbAVdPnu1MZ --event send
```

stake(amount), unstake(amount), vote([bps,...])

you can do governance(stake, unstake, vote) in contract with contract address

```
contract.stake("1 aergo")
contract.vote(["<bp1 address>", "<bp2 address>"....)
contract.unstake("1 aergo")
```

Built-in Functions

Lua provides the language itself as a useful function and basic package. It provides useful functions such as string management functions, so you can easily create smart contracts using these functions. Please refer to the Lua Reference Manual for detailed syntax, explanation, basic built-in functions and packages.

Since Lua Smart Contract is performed in Blockchain, OS related functions including input / output are not provided for stability and security.

Here is a list of the main functions that are not available.

```
print, loadstring, dofile, loadfile and module
```

Here is a list of the default packages that are not available.

```
coroutine, io, os and debug
```

The string, math, and table packages are available. However, you can not use the random, randomseed functions in the math package.

DB package

If the smart contract is handling simple types of data, it would not be difficult to implement using only the basic APIs (getItem (), setItem ()). However, complex data structures, data association, scope queries, filtering, sorting, and other features require the complexity and size of the data logic so developers can not focus on critical business logic. To solve this problem, Aergo supports SQL. This section details the types and usage of SQL APIs available in smart contracts

> Note: The db package is only available on private networks(SQL TestNet).

exec(sql)

This function perform DDL or DML statements

query(sql)

This function perform SELECT statements and return result set object

prepare(sql)

This function create prepared statement and return statement object

```
-- create customer table
function createTable()
  db.exec([[create table if not exists customer(
        id text,
        passwd text,
        name text,
        birth text,
        mobile text
    )]])
end

function insert(id, passwd, name, birth, mobile)
  db.exec("insert into customer values ('" .. id .. "', '"
      .. passwd .. "', '"
      .. name .. "', '"
      .. birth .. "', '"
      .. mobile .. "')")
end
```

functions of result set object

Object functions must be called with the: operator

next

This function prepare the next result row. Returns false if there is a row, false if it is not.

get

This function return result row

```
function query(id)
  local rt = {}
  local rs = db.query("select * from customer where id like '%'" .. id .. "'%'")
  while rs:next() do
    local col1, col2, col3, col4, col5 = rs:get()
    local item = {
        id = col1,
        passwd = col2,
        name = col3,
        birth = col4,
        mobile = col5
    }
    table.insert(rt, item)
  end
  return rt
end
```

functions of prepared statement object

equivalent to the prepareStatement object in JDBC. You can use the parameters in SELECT or DML to view, add, modify, or delete information.

query(bind1 , bind2, ...)

This function execute SELECT statement by specifying argument value corresponding to bind parameter and return result set object

exec(bind1, bind2, ...)

This function execute DML statement by specifying argument value corresponding to bind parameter

SQL Restrictions

Smart Contract's SQL processing engine is built on SQLite. Therefore, detailed SQL usage grammar can be found at https://sqlite.org/lang.html and https://sqlite.org/lang_corefunc.html. However, because of the stability and security of the Aergo, not all SQL is allowed.

types

Allow only SQL datatypes corresponding to Lua strings and numbers (int, float).

- text
- integer
- real
- null
- date, datetime

SQL statement

The allowed SQL statements are listed below. However, DDL and DML are only allowed in smart contract transactions.

- DDL
  - TABLE: ALTER, CREATE, DROP
  - VIEW: CREATE, DROP
  - INDEX: CREATE, DROP
- DML
  - INSERT, UPDATE, DELETE, REPLACE
- Query
  - SELECT

Function

The following is an unavailable list

- Data and time related functions can be used except 'now' timestring and 'localtime' modifier
- load_XXX function
- random function
- sqlite_XXX function

For a list of other functions and descriptions, please refer to the links below.

- Core: https://www.sqlite.org/lang_corefunc.html
- Date and time: https://www.sqlite.org/lang_datefunc.html
- Aggregation: https://www.sqlite.org/lang_aggfunc.html

Constraint

The following constraints can be used.

- NOT NULL
- DEFAULT
- UNIQUE
- PRIMARY KEY(FOREIGN KEY)

- CHECK

```
function init_database()
    db.exec("drop table if exists customer")
    db.exec("create table if not exists customer (cid integer PRIMARY KEY ASC␣
↪AUTOINCREMENT , passwd text , cname text, birthdate date, rgdate date)")
    db.exec("insert into customer (cid, passwd, cname, birthdate, rgdate) values (100␣
↪,'passwd1','', date('1988-01-03'),date('2018-05-30'))")
    db.exec("insert into customer (passwd, cname, birthdate, rgdate) values ('passwd2
↪','', date('1978-11-03'),date('2018-05-30'))")
    db.exec("insert into customer (passwd, cname, birthdate, rgdate) values ('passwd3
↪','', date('1938-04-23'),date('2018-05-30'))")

    db.exec("drop table if exists product")
    db.exec("create table if not exists product (pid integer PRIMARY KEY ASC␣
↪AUTOINCREMENT, pname text, price real, rgdate date)")
    db.exec("insert into product (pid, pname, price, rgdate) values (1000 ,'',1000,␣
↪date('2018-05-30'))")
    db.exec("insert into product (pname, price, rgdate) values ('',10000, date('2018-
↪05-30'))")
    db.exec("insert into product (pname, price, rgdate) values ('',3500, date('2018-
↪05-30'))")

    db.exec("drop table if exists order_hist")
    db.exec("create table order_hist (oid integer PRIMARY KEY ASC AUTOINCREMENT, cid␣
↪integer, pid integer, p_cnt integer, total_price real, rgtime datetime, FOREIGN␣
↪KEY(cid) REFERENCES customer(cid), FOREIGN KEY(pid) REFERENCES product(pid) )")
    db.exec("insert into order_hist(oid, cid, pid,  p_cnt, total_price,rgtime)␣
↪values(10000,100,1000,3,3000, datetime('2018-05-30 16:00:00'))")
    db.exec("insert into order_hist(cid, pid, p_cnt, total_price, rgtime) values(100,
↪1000,3,3000, datetime('2018-05-30 17:00:00'))")
    db.exec("insert into order_hist(cid, pid, p_cnt, total_price, rgtime) values(100,
↪1000,3,3000, datetime('2018-05-30 18:00:00'))")
end
```

json package

Json package is provided for user convenience in input and output. This package allows automatic conversion between Json format strings and Lua Table structures.

encode(arg)

This function returns a JSON-formatted string with the given lua value.

decode(string)

This function converts a string in JSON format to the corresponding Lua structure and returns it

crypto package

sha256(arg)

This function compute the SHA-256 hash of the argument.

ecverify(message, signature, address)

This function verify the address associated with the public key from elliptic curve signature.

```
function validate_sig(data, signature, address)
    msg = crypto.sha2565(data)
    return crypto.ecverify(msg, signature, address)
end
```

bignum package

Since the lua number type has a limit on the range that can be represented by an integer (less than 2 ^ 53), the bignum module is used to provide an exact operation for larger numbers.

- Notice
  - == Operations on bignum and other types always return false.
  - Bignum does not allow a decimal point.
  - Bignum value range : -(2^256 - 1) ~ (2^256 -1)

number(x)

This function make bignum object with argument x(string or number)

isneg(x)

Check bignum x if negative than return true else false

iszero(x)

Check bignum x if zero than return true else false

tonumber(x)

Convert bignum x to lua number

tostring(x) (bignum.tostring(x) same as tostring(x))

Convert bignum x to lua string

neg(x) (same as -x)

Negate bignum x and return as bignum

sqrt(x)

Returns the square root of a positive number as bignum. not permitted negative value to x

compare(x, y)

Compare two big numbers. Return value is 0 if equal, -1 if x is less than y and +1 if x is greater than y.

add(x, y) (same as x + y)

Add two big numbers and return bignum

sub(x, y) (same as x - y)

Subtract two big numbers and return bignum

mul(x, y) (same as x * y)

Multiply two big numbers and return bignum

mod(x, y) (same as x % y)

Returns the bignum remainder after bignum x is divided by bignum y

div(x, y) (same as x / y)

Divide two big numbers and return bignum

pow(x, y) (same as x ^ y)

Power of two big numbers and return bignum. not permitted negative value to y

divmod(x, y)

Returns a pair of big numbers consisting of their quotient and remainder

powmod(x, y, m)

Return the bignum remainder after pow(bignum x,bignum y) is divided by bignum m. not permitted negative value to y

```
function factorial(n,f)
 for i=2,n do f=f*i end
 return f
end
for i=1,30 do
  local b=factorial(i,bignum.number(1))
  return bignum.mod(b, 10)
end
```

5.1.4 Examples

Reusable code

https://github.com/aergoio/athena-370

Using libraries

https://github.com/aergoio/athena-371

Aergo Contract Example

https://github.com/aergoio/aergo-contract-ex

5.2 Aergo SCL

Aergo Smart Contract Language (Aergo SCL, we pronounce it "askel") is a language dedicated to smart contact for Aergo, designed to make it easier for developers to develop smart contracts. Although the specification is less than traditional languages, it aims to be faster and more powerful. Aergo SCL has the following characteristics.

- Strongly typed language
- Object-oriented language (except for inheritance and abstraction)
- Support SQL extension for database access
- Support extension for blockchain access
- Compiled with WebAssembly

Aergo SCL is currently not supported on the testnet but under active development.

CHAPTER 6

Technical Specifications

6.1 Addresses

6.1.1 Client-side

Account and contract addresses are Base58-check encoded strings that look like this:

```
AmQA7dHJFiA4mXXxTV5sAniLpxMrankdW4Cow3ykj1UM5G14QKL5
```

- Technical explanation of Base58-check encoding

The prefix used for encoding Aergo addresses in base58-check is `0x42`.

6.1.2 Internal

Internally (in the server and over the wire, i.e. in GRPC requests) addresses are byte arrays with a length of 33. They represent the compressed public key of an account.

- Technical explanation of public keys

In the case of smart contracts, the address is generated from a hash of the creator's account and the creating transaction's nonce, prefixed with the byte 0x0C to arrive at a compatible length of 33 bytes. As a developer, you don't have to worry about this: smart contract addresses can be used just the same as account addresses.

6.2 Token Units

1 aergo = 1 * 10^18 aer = 1 * 10^9 gaer

The Aergo CLI and client libraries have support for these units, i.e. you can specify transaction amounts as `1 aergo` instead of `1000000000000000000`.

Note that amounts in the base unit aer exceed the range of 64-bit integers. You need some implemention of Big Integer to deal with these numbers. Aergo SDKs come bundled with a recommended way to do that. In most cases, you can just use strings instead of numbers. For example, when creating a JSON transaction, set

```
{
    "amount": "10000000000000000000"
}
```

6.3 Name System

Aergo contains a name service to translate addresses into easy to remember short names.

One name maps to exactly one address. Names are currently fixed to a length of 12 characters.

Names include an `owner` and a `destination`. The owner is used to determine who can change the name, the destination is the actual adress the name should resolve to. In case of normal accounts, these two values will be identical, but they differ in case of smart contracts.

6.3.1 Creating and updating names

The easiest way to create and update names is the aergocli.

Registering and updating names currently requires spending 1 aergo.

To register a new name:

```
aergocli name new --from my_unlocked_account_address --name my12charname
```

To change the destination of the name:

```
aergocli name update --from my_unlocked_account_address --to account_address_or_
↪contract_address --name my12charname
```

Warning: The update command affects both the destination and the owner of a name. If the new destination is a regular account, both owner and destination are set to that address. If the new destination is a smart contract address, the owner is set to the address of that contract's creator.

6.3.2 Lookups

To retrieve the owner and destination of a registered name:

```
aergocli name owner --name my12charname
```

The `destination` field contains the resolved address.

This method is also available over RPC and in the various SDKs.

6.3.3 Reverse lookups

Aergo server currently supports no way to do reverse lookups, i.e. finding all names associated with an address. On public networks, you can use block explorers like Aergoscan to find that information.

6.3.4 Technical details

Names are stored in the state of the special account `aergo.name`. They are created and updated using special governance transactions. Refer to transaction types for the technical specification of these actions. Governance transactions currently don't require any fee, but the name system requires an amount of 1 aergo to be included in the transaction.

Transactions that create or update names are effective after they are included in a block. That means that you can only refer to new or updated names in the following block.

6.4 Consensus Algorithm

The objective of all blockchain protocols is to replicate a blockchain and its associated state across participating nodes. To achieve such an agreement, each blockchain protocol deploys a consensus algorithm.

The public Aergo network uses **Delegated Proof of Stake (DPoS)** for blockchain consensus.

BP Election In DPoS, blocks are generated only by a limited number of nodes called Block Producers (BPs). BPs are elected via voting, where the voting power is weighted by staked tokens.

BPs are re-elected round by round (blocks per round = 100). In each round, time is split into slots and each slot is assigned to one of the elected BPs. Only the permitted BP can produce a block in a time slot.

Staking & Voting Staking means locking up one's tokens for a minimum period of time. Any user wanting to vote must stake their tokens since the voting power is weighted by the number of staked tokens, as remarked above.

All of these requests are performed via a transaction. Therefore, all proccesses are transparently recorded in the blockchain and can be verified by anyone.

After the voting transaction is included in the block, the results are caculated immediately. But there is a slight delay until it is applied. The current BPs are elected based on the voting result gathered at the block number: (<current block number> / 100 - 1) * <the total number of BPs>.

In other words, the voting results gathered in the past (approximately 1 round before) are used for stability (recent blocks may be roll-backed via a reorganization).

Votes are locked for a certain period of time to prevent users from spamming votes. On the public Aergo network, this is currently approx. 1 day (after 60 * 60 * 24 block number). That means, after casting a vote, a user can only change their vote after 1 day has passed.

Last Irreversible Block (LIB) In some blockchain protocols, a blockchain may branch into two or more, which is called a fork. Later, only one of them is chosen as the main branch via a set of rules defined by the protocol. Such reorganizations limit each block's finality and, in turn, transaction's finality. For example, a transaction, included in a block at one time, might be rejected later.

DPoS also allows blockchain forks. However, a block becomes a last irreversible block (LIB) when it is (double time) confirmed by a majority (2/3+) of BPs. Once a block is determined as a LIB, it cannot be rolled back, i.e. it achieves finality.

6.5 Transactions

6.5.1 Tx

Field		Type	Description
Hash		bytes	Hash of body
Body	Nonce	uint64	Increasing number used only once per sender account
	Account	bytes	Decoded sender account address
	Recipient	bytes	Decoded receiver account address
	Amount	bytes	Amount of transfer
	Payload	bytes	Smart contract data
	Limit	uint64	Reserved
	Price	bytes	Reserved
	Type	int	0 is normal type, 1 is governance type
	Sign	bytes	ECDSA signature with secp256k1

6.5.2 Payload

A payload can be any kind of binary data, but is most often used with JSON strings for smart contract calls.

6.5.3 Transaction types

There are two kinds of transactions.

Normal type

Normal transactions are used to transfer tokens and calling smart contracts.

Governance type

Governance transactions are used for calling system contracts, such as staking and voting. Transactions of this type have a special payload format and recipient.

The following table shows the specification for each field of the transaction body.

Action	Recipient	Amount	Payload
staking	`aergo.system`	amount to stake	`{"Name":"v1stake"}`
unstaking	`aergo.system`	amount to unstake	`{"Name":"v1unstake"}`
voting	`aergo.system`	0	`{"Name":"v1voteBP","Args":[<peer IDs>]}`
create name	`aergo.name`	1 aergo	`{"Name":"v1createName","Args":[<a name string>]}`
update name	`aergo.name`	1 aergo	`{"Name":"v1updateName","Args":[<a name string>, <new owner address>]}`

The aergo.system transactions, including staking, unstaking and voting, can be sent about once per day per account. For staking and unstaking, there is a limit to the amount of requests. It must be over 10000 aergo based on the amount

of staked. Therefore, the first staking request should exceed 10000 aergo, and in the case of the unstaking request, more than 10000 must be left or withdrawn altogether.

6.5.4 Transaction receipts

See also:

See API → Receipt for a detailed explanation of all the receipt data.

Every transaction generates a receipt upon succesful execution. The `status` can be one of three values:

SUCCESS Simple value transfer transactions and succesful contract executions. For contract calls, the result is available in `result`.

ERROR Failed contract execution. The error message can be found in `result`.

CREATED Succesful contract deployment transaction. The created address can be found in `contractAddress`.

6.6 Smart Contracts

Contracts are special kinds of accounts that have code, an ABI, and state.

This is a technical description of the protocol and execution. It is recommended to interact with contracts using higher level API supported by aergocli and the various SDKs.

6.6.1 Deployment

Contracts are deployed by sending a transaction with the contract code as payload to the null receiver (empty receiver address). Upon execution, the contract state is created at an address calculated from the sender and the nonce of the deployment transaction. You can find out the created contract address in the transaction receipt.

See here for details about address generation.

6.6.2 Calling Contracts

To call contract methods, send a transaction to the contract's address specifying the function name and arguments as JSON payload.

6.6.3 Events

Contracts can log events during execution. This is the preferred way to notify the outside world of important state changes.

Logged events can be found as part of the transaction receipt and also easily searched for using the GetEvents API.

6.7 Transaction Fees

The Aergo protocol includes transaction fees that need to be paid according to the configuration of the network.

Currently, the public mainnet has a minimum fee of 0.002 aergo.

And additional fees is required to use a contract.

- 5 gaer per byte of the tx's payload
- 5 gaer per byte requested to update the state DB

Because Aergo does not have gas prime and gas limit, during the early period, execution of the contract has some limitations.

- To send a coin transfer TX, the sender must have a balance of 1.026 aergo or more.
- And the sender who to execute a contract must have a balance of 2.05 aergo or more.
- The maximum size of the state DB that can be modified per contract is 200 KB.

This fee system may change in the future.

Sidechains can configure a custom fee. For private chains, free transactions are also possible.

6.8 Blocks

6.8.1 Block header

Field	Type	Description
ChainID	[]byte	identifier of chain
PrevBlockHash	[]byte	hash of previous block
BlockNo	uint64	height of block
Timestamp	int64	block timestamp
BlocksRootHash	[]byte	root hash of State SMT(sparse merkle tree)
TxsRootHash	[]byte	Merkle root of transactions
ReceiptsRootHash	[]byte	Merkle root of transaction receipts
Confirms	uint64	indicates how many block is confirmed by block in DPOS consensus
PubKey	[]byte	public key of the block producer (BP)
CoinbaseAccount	[]byte	the account address to which block reward is given
Sign	[]byte	BP's Signature for block header

ChainID

It is used to prevent the wrong block transmission from other chains.

Field	Type	Description
Version	int32	version number of chain
PublicNet	bool	true if public net
MainNet	bool	true if main net
CoinbaseFee	string	Fee consumed per tx. Fixed value
Magic	string	Magic string, arbitrarily chosen by each blockchain
Consensus	string	dpos or sbp

Version

Version is used to identify when the block format changes or when the function of the chain changes.

PublicNet

Differentiate between public and private networks.

MainNet

Differentiate between main net and other test net or other use net.

CoinbaseFee

CoinbaseFee is used to set the peak per Tx.

Magic

Magic can be considered as a name of a blockchain. The two blockchains with different magic strings reject each other's blocks (they are seperate, indenpendent blockchains).

Consensus

Specify the concensus method name used in the chain.

6.8.2 Block body

Field	Type	Description
Txs	[]Tx	Transactions

6.9 P2P

6.9.1 P2P wire protocol

This page describe the p2p wire protocol for protocol version 0.3.

Note: Some contents are work in progress and will be implemented before the launch of the mainnet.

Messages

Handshake message

A handshake message is used once at starting handshake. It contains two 4-byte number

~4 : Magic

~8 : Version

Normal message

Header (48bytes) + Payload (variable size)

Message header

~4 : code number of subprotocol . Big endian number

~8 : payload size. Big endian number.

~16 : creation time of this message. unix timestamp with precision of nanosecond . Big endian number

~32 : message id. binary form of uuid.

~48 : original request id. only meaningful if message is response message. binary form of uuid.

Message payload

Payload is serialized form of protobuf struct. The size and struct type is differ by subprotocol.

List of Subprotocols

Code is hexadecimal number. Refer to Subprotocols for detailed information of each subprotocol.

Name	Code	Remark
StatusRequest	0001	My node status, which includes chainID, address information, last blocks or others. Used in handshake
PingRequest	0002	Ping including last block hash and number
PingResponse	0003	Response ping including last block hash and number
GoAway	0004	Disconnect notice with reason
AddressesRequest	0005	Query request to get list of connected peers
AddressesResponse	0006	Response for AddressesRequest.
GetBlocksRequest	0010	
GetBlocksResponse	0011	
GetBlockHeadersRequest	0012	
GetBlockHeadersResponse	0013	
NewBlockNotice	0016	
GetAncestorRequest	0017	
GetAncestorResponse	0018	
GetHashesRequest	0019	
GetHashesResponse	001A	
GetHashByNoRequest	001B	
GetHashByNoResponse	001C	
GetBlockHeadersResponse	001D	
GetTXsRequest	0020	
GetTxsResponse	0021	
NewTxNotice	0022	
BlockProducedNotice	0030	

List of Response Status

Some subprotocols for responsing other message have ResultStatus property.

Name	Code	Remark
OK	0	OK is returned on success.
CANCELED	1	when operation was canceled
UNKNOWN	2	
INVALID_ARGUMENT	3	INVALID_ARGUMENT is missing or wrong value of argument
DEAD-LINE_EXCEEDED	4	timeout
NOT_FOUND	5	Resource is not found
ALREADY_EXISTS	6	
PERMISSION_DENIED	7	
RE-SOURCE_EXHAUSTED	8	
FAILED_PRECONDITION	9	
ABORTED	10	
OUT_OF_RANGE	11	
UNIMPLEMENTED	12	indicates operation is not implemented or not supported/enabled in this service.
INTERNAL	13	
UNAVAILABLE	14	Unavailable indicates the service is currently unavailable.
DATA_LOSS	15	
UNAUTHENTICATED	16	indicates the request does not have valid authentication credentials for the operation.

Payload of Subprotocols

StatusRequest

1. sender: information of sender (address, port, peerID or etc)
2. bestBlockHash: current best block of sender
3. bestHeight: current best block height of sender
4. chainID: ChainID which sender is storing

6.9.2 Peer Connect

NOTE: Some document is not translated yet.

Node Discovery

When the Aergo server starts running, you need a way to connect to the network. To do this, you need to know and connect to other Aergo server nodes already connected to the chain. Aergo does this using several methods.

Query polaris

Polaris keep list of aergo server nodes, such like DNS. Aergo server connect and query to official Aergo Polaris automitically, if the chain is Official AergoMainNet or AergoTestNet.

You can build and run custom Polaris for your private chain or custom public chain. You should configure to use custom Polaris by modifying config file.

Designate Peer

You can add a list of designated known peers to connect to at boot time in configuration file using the option 'npaddpeers'.

Dynamic peer discovery

Aergo server requests peer list from other connected peers as well as Polaris if it cannot find enough peers.

Choosing remote peers

(This feature is currently work in progress)

The list of peers to connect to is managed in Kademlia-like manner. The number of peers that are closer to the peer is larger and the number of peers that are connected to the peer is smaller. The peer's distance is based on the difference based on the hash value, not the physical distance.

Peer connect process

The network communication of Aergo server operates on libp2p basis, and libp2p is responsible for encryption and node distinction at transmission level. After a tcp session is created, both peers start the handshake operation.

Peer Handshake

In the Handshake phase, peers exchanges each other's version, chain ID and state to determine whether to connect. If the other node version is not supported by the current server or the operating chain ID is different, the connection is stopped. Once the handshake is successful, the difference in block height is compared and synchronization started.

Keep Alive

Aergo server maintains a connection-based communication. Both peers disconnect when the internally defined retention score increases beyond a certain level. This score decreases to a low level when a query is requested, and increases when a bad block or TX notification is sent.

Peer blacklist

(This feature is currently work in progress)

When an internally defined ban score exceeds a certain level, the Aergo server blocks the peer's connection. This score increases due to the connection being disconnected due to exceeding the connection maintenance score, etc., and the

blocking period is also changed by calculating the score or the number of blocking times. You can also permanently block by specifying a block address in the configuration file.

6.10 Block Management

This article describes implementation details of management of blocks in the Aergo server. It is aimed at blockchain server developers.

6.10.1 Block generation

The block factory is responsible for generating blocks and part of the consensus engine.

The consensus engine manages when blocks are created on which nodes. AERGO uses DPoS by default. A new block producer is selected every second.

The process for creating blocks in the block factory is as follows.

1. Select transactions from from mempool
2. Perform a transaction one by one. If the execution is successful, the transaction is included in the block.
3. If the block size and time limit are exceeded, the transaction is no longer included.
4. Create blocks that contain executed transactions and add signatures to blocks.

The block factory sends the generated blocks to the chain service for propagation and storage.

6.10.2 Notify block

Blocks passed to the chain service are propagated to the connected peer nodes and stored in the database.

The chain service of a node that received the block verifies the block, executes the transactions, and adds the block to its own database.

6.10.3 Save block

The chain service stores the generated blocks in the DB for permanent storage. It also stores additional meta information to store connection information about the chain.

AERGO uses the Badger database as its permanent storage. Badger stores data in key/value form and supports the concept of transactions similar to a typical database.

Information stored in the DB is as follows.

The DB is managed in two separate categories, ChainDB and StateDB. This article describes the ChainDB used to store chain information.

Block info

This information is used to search for blocks in a chain using a hash. Typically, when a block is requested as a hash from another node, the Hash is searched for and responded to the block with a key.

Key	Value
Hash of Block	serialized data of Block

Chain info

Chain info is the information that indicates when a block is connected to the main branch.

If the main branch is not connected, that is, only block info is saved and the chain info is not stored.

You can search the blocks of every height of the main branch using the chain info. You can know the height of the top block of the chain.

Chain info is also used to determine whether the block belongs to the main branch or the sub branch.

Key	Value
Number of Block	block hash
Latest block Key	latest block number of main branch

Transaction Info

tx info is meta information of the transaction that was performed in the main branch. If you search the DB with transaction hash, you can see which block the transaction is stored in and where the transaction is stored in the block.

The contents of the transaction are not saved separately. Therefore, in order to know the contents of the actual transaction, you must search Block info using Block hash and index found in tx info.

Key	Value
Transaction hash	(Block Hash, index of transaction in block)

6.11 Chain Management

This article describes implementation details of chain management in the Aergo server. It is aimed at blockchain server developers.

The chain service is responsible for managing the chain. It performs the following tasks in large part:

1. Validation
2. Insert block to chain
3. Reorganization
4. Syncronization

6.11.1 Block validation

The blocks received in the network may not be valid, so a number of checks are made.

Pre-execution execution

The validator module ensures that the block was generated from a valid BP and the transaction contained in the block was not forged.

- Consensus validation

  Validate the block generated by the valid BP through block creation time and signature.

- Transaction merkle validation

  Validate that the transactions were not forged. The verifier module generates a merkle tree with the transactions and checks if it is the same as the transaction merkle root stored in the block header.

Post-execution validation

These checks ensure that the results of the transaction contained in the block are the same as the results of the BP node that generated the block.

- State root validation

  Checks if the changed state root node hash is the same as the blocksRootHash stored in the block head.

- Receipt merkle validation

  Generates a merkle tree with the receipts generated as a result of the transactions and checks if they are identical to the receipts stored in the block header.

6.11.2 Insert block into chain

There are two cases in which the chain service adds blocks:

1. Block generated by block factory

   When it is a BP node's turn, it generates new blocks in the block factory.

   The block factory performs all transactions and forwards the block to the chain service. The execution results are included with the block.

   The execution results include the results of the transaction, receipt, and state changed entry indicating the change to the account.

   The chain service stores information related to blocks in the ChainDB and information about accounts in the StateDB.

2. Block received from another peer node

   For blocks received from other peer nodes, there are three main cases:

   - Orphan block

     This is the case if the parent block has not yet been stored in the DB. Orphan blocks are stored in the orphan block cache on memory. Then, when the parent block is received, it is removed from the organ block cache and reprocessed.

   - Side branch block

     In case the parent block is stored but is not part of the main branch. In this case, the block is not performed and only the block info{hash, block} is stored.

   - Main branch block

     In case the head of the main branch is the next block. In this case, the blocks are stored after performing the transactions.

The process for storing blocks in the main branch is as follows:

1. Validation before execution
2. Execute transactions
3. Apply changed account entries state merkle tree

4. Validate after execution
5. Save state merkle tree to StateDB
6. Save chain meta data to ChainDB

6.11.3 Reorganization

The chain service selects and maintains the longest chain as the main branch. The side branches are not executed and only the block info is stored in the DB. If the side branch received from another peer is longer than the main branch held by the node, the side branch is changed to the main branch. This is called the reorgnize process.

The reorgnization is performed as follows.

1. Find common ancestor

   Syncer finds the last common ancestor block of the main branch and side branch.

2. Rollback master branch

   State is reset to the point at which the common ancestor block was executed

3. Rollforward side branch

   Syncer runs from the next number of the common ancestor block to the head block of the side branch. At this time, only StateDB is changed and Chain info and Tx info are not changed.

4. Swap chain meta

   Syncer do not change the chain info during rollback and rollforward to atomically change the chain. Change the chain meta information after the previous process has been successfully completed. At this time, chain info and transaction info are deleted for the rollbacked block, and new chain info and transaction info are added for the rollforwarded block.

   Transactions belonging to rollbacked blocks but not included in rollforwarded blocks are returned to mempool. This is to prevent transaction loss.

6.11.4 Syncronization

When you add a new node or restart a node that was temporarily stopped, you need to get the latest chain information from existing nodes. This is called the synchonization process.

The situation that causes sync is as follows:

- When the peer goes through a handshake process to connect, the height of the chain of the remote peer is higher than the current node
- If the height of the block notified in the peer is higher than the head of the current main branch

The syncer specifies the node that sent the block that caused the sync to the target node and synchronizes with the chain of that node.

To synchronize a long chain, a large amount of block information must be received from the peer node. This is likely to cause a performance degrade at the peer node. Therefore, it gets information from as many peers as possible to distribute the load.

Synchronize step

1. Find common ancestor: Syncer finds the last common ancestor of the current node chain and the target node chain.

2. Get hashes: It gets the hashes of the block after the common ancestor from the target node.
3. Get blocks: N blocks are requested from all valid peers connected to the current node.
4. Insert blocks to chain: The received block is added to the chain using the chain service.

2, 3, and 4 are performed in parallel. Most of the time is spent in the insert part of the chain.

CHAPTER 7

Contributing

As AERGO is an open-source platform, individual contributions are vital.

There are many ways that you can help, no matter if you are a novice or advanced programmer or user.

7.1 Ways to Contribute

These are just some examples for welcome contributions.

- Improve this documentation
- File a bug report
- Submit a pull request
- Discuss protocol changes
- Help answering questions

CHAPTER 8

Join the Community

An open-source platform lives from its community. You can reach out to the Aergo team and other members of the developer community via these channels:

- Discord (for technical discussions)
- Github
- Twitter
- Telegram
- Telegram Announcements

CHAPTER 9

Building from Source

This page explains how you can build aergo from source.

If you are only interested in running the aergo tools, it is easier to use pre-built binaries or Docker images instead, as explained in Quickstart.

9.1 Linux

1. Install dependencies

```
apk update
apk add git build-base go libgcc
```

2. Install aergo

```
go get -d github.com/aergoio/aergo
cd ${GOPATH}/src/github.com/aergoio/aergo
git submodule init && git submodule update
make
```

3. Set environment

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${GOPATH}/src/github.com/aergoio/aergo/
↪libtool/lib
```

9.2 macOS

1. If you haven't already, install homebrew.
2. Install dependencies

```
brew install go
brew install cmake  # optional
brew install protobuf  # optional
```

3. Install aergo

```
go get -d github.com/aergoio/aergo
cd `go env GOPATH`/src/github.com/aergoio/aergo
git submodule init
git submodule update
make
```

4. Set environment

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:${GOPATH}/src/github.com/aergoio/aergo/
↪libtool/lib
```

5. Run server

```
./bin/aergosvr
```

9.3 Windows

Building on Windows is currently not supported.

9.4 Generating protobuf files

If you changed the protobuf files, you need to re-compile the type definitions.

1. Install protoc-gen-go

```
GIT_TAG="v1.1.0" # set version of protoc-gen-go to 1.1.0, on which aergo v0.8.x␣
↪depends.
go get -u github.com/golang/protobuf/protoc-gen-go
git -C "$(go env GOPATH)"/src/github.com/golang/protobuf checkout $GIT_TAG
go install github.com/golang/protobuf/protoc-gen-go
```

2. Generate type definitions from protobuf files

```
make protoc
```

CHAPTER 10

SDKs

Aergo offers language-specific SDKs to make it easy to develop Dapps. There are currently SDKs for Java, Javascript, and Python programming environments.

10.1 Heraj

Please refer to the Github Wiki pages for Heraj.

10.2 Herajs

Please refer to the Herajs documentation.

10.3 Herapy

Please refer to the Herapy documentation.

10.4 Other Languages

If you are using another language that is currently not supported by a decicated SDK, you can still interact with AERGO using the API directly.

Please refer to the API reference.

CHAPTER 11

Tools

11.1 Aergocli

aergocli is a command line tool that interfaces with the GRPC exposed by **aergosvr**.

11.1.1 Installation

Using the binary

You can find the latest binary release on Github. Just download and extract the archive.

Using Docker

You can also use the Docker image aergo/tools. Example: `docker run --rm aergo/tools aergocli version`

11.1.2 Usage

In order to use all features of aergocli you will need to have the end point (IP address and port number) to a aergosvr instance.

For a list of all commands known to `aergocli`, simply run it with no arguments.

```
Usage:
aergocli [command]

Available Commands:
account     account command
blockchain  Print current blockchain status
bp          show bp voting stat
chaininfo   Print current blockchain information
committx    Send transaction
contract    contract command
```

(continues on next page)

(continued from previous page)

```
event       Get event
getblock    Get block information
getpeers    Get Peer list
getstate    Get account state
gettx       Get transaction information
help        Help about any command
keygen      Generate private key
listblocks  Get block headers list
metric      Show metric informations
name        Name command
node        Show internal metric
receipt     receipt command
sendtx      Send transaction
signtx      Sign transaction
verifytx    Verify transaction
version     Print the version number of Aergocli

Flags:
    --config string   config file (default is config.toml) (default "config.toml")
-h, --help            help for aergocli
    --home string     aergo home path (default ".")
-H, --host string     Host address to aergo server (default "localhost")
-p, --port int32      Port number to aergo server (default 7845)

Use "aergocli [command] --help" for more information about a command.
```

11.1.3 Examples

Create account

```
$ aergocli account new --password yourpasswordhere
AmP96xd6WYRe1jrWixC3VyFXRCzwZxAJtU3Uc54vn182xG9Yn9Cs
```

or

```
$ aergocli account new
Enter Password:
Repeat Password:
AmP96xd6WYRe1jrWixC3VyFXRCzwZxAJtU3Uc54vn182xG9Yn9Cs
```

Now we can use the account in aergosvr. Please remember the password carefully because there is no way to retrieve the password again.

Send transaction

Before request send transaction you must unlock your account first

```
$ aergocli account unlock --address␣
↪AmQFgm1gCvoRw2RfBXnipRmeCLEc6tTQ1kBMmLEzHjp91xYnXK78
Enter Password:
AmQFgm1gCvoRw2RfBXnipRmeCLEc6tTQ1kBMmLEzHjp91xYnXK78
```

or

```
$ aergocli account unlock --address␣
↪AmQFgm1gCvoRw2RfBXnipRmeCLEc6tTQ1kBMmLEzHjp91xYnXK78 --password yourpasswordhere
AmQFgm1gCvoRw2RfBXnipRmeCLEc6tTQ1kBMmLEzHjp91xYnXK78
```

and then

```
$ ./aergocli sendtx --from AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q --to␣
↪AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ --amount 1aergo
{
 "hash": "AkjqFcayutenhonnZPU4X5QB1fNBTZv3o2fNzMLNQR3q"
}
```

Look up transaction

Use gettx command. If transaction is in block, return transaction with index that represent where it's included.

When transaction is in mempool, gettx result is like below

```
$ aergocli gettx 9cAphBMD2zJCD13QfCn7rmxh5iDfrj6M9Wmo54TPNPCg
{
 "Hash": "9cAphBMD2zJCD13QfCn7rmxh5iDfrj6M9Wmo54TPNPCg",
 "Body": {
  "Nonce": 2,
  "Account": "AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q",
  "Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
  "Amount": "1230000000000000000",
  "Payload": "",
  "Limit": 0,
  "Price": "0",
  "Type": 0,
  "Sign":
↪"AN1rKvt8EZHKEE2wNPXAhGcDA4pMo7yRRjTWCcpyrW9QCMv6nMhvhqriWujHdaDJgJ6ft6VLDActEFtUFA2pRnRJFVFSWxPSR
↪"
 }
}
```

When transaction is in block, gettx result is like below

```
$ aergocli gettx 9cAphBMD2zJCD13QfCn7rmxh5iDfrj6M9Wmo54TPNPCg
{
 "TxIdx": {
  "BlockHash": "ECVG696Jc7FvUL86sDSxT28akh4Hf1RXFRzmGqtAv9zU",
  "Idx": 1
 },
 "Tx": {
  "Hash": "9cAphBMD2zJCD13QfCn7rmxh5iDfrj6M9Wmo54TPNPCg",
  "Body": {
   "Nonce": 2,
   "Account": "AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q",
   "Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
   "Amount": "1230000000000000000",
   "Payload": "",
   "Limit": 0,
   "Price": "0",
   "Type": 0,
   "Sign":
↪"AN1rKvt8EZHKEE2wNPXAhGcDA4pMo7yRRjTWCcpyrW9QCMv6nMhvhqriWujHdaDJgJ6ft6VLDActEFtUFA2pRnRJFVFSWxPSR
↪"
```

(continues on next page)

(continued from previous page)

```
  }
 }
}
```

Check block

```
$ aergocli getblock --hash GGT9wahqcKKGKUncMuhRLLL3JaCs2MEBx7V8UdrK9JNi
{
 "Hash": "FwBq14HiBPMPoGV6jYxW4AaGHsgoD9UJjmYyWwnQR6xU",
 "Header": {
   "ChainID": "1111117eaxEoT4pDHzFTCyKv93acfDHbsytUPK3oqU6",
   "PrevBlockHash": "GocCiGXUVV4ygsbEi1VaJKkBwyvRsV4W4Qj72J5ZciZK",
   "BlockNo": 17655,
   "Timestamp": 1548314097754698000,
   "BlockRootHash": "5etqxP9HTtzgN8a3tN5Ev8ka4aG3aQM5GYNikwmsV41q",
   "TxRootHash": "AkjqFcayutenhonnZPU4X5QB1fNBTZv3o2fNzMLNQR3q",
   "ReceiptsRootHash": "7ZpYoMA2feXoXAit5J2FFuX16cwz9hp8twGwu7rUHCRZ",
   "Confirms": 2,
   "PubKey": "GZsJqUTtFVJTJ5SbwuFac4NxZFWgTRjpioPD76UL1DHZLhxmWg",
   "Sign":
↪"AN1rKvtSPyBqB34TGmEQ6FQ8Lz61dvQ4zVAHodAnPGTnXD7gihmSkgrWcyeEoXNeS6zLTtgjukcTiwSKTHmMATGe8PXgwcJJK
↪",
   "CoinbaseAccount": ""
  },
  "Body": {
   "Txs": [
    {
     "Hash": "AkjqFcayutenhonnZPU4X5QB1fNBTZv3o2fNzMLNQR3q",
     "Body": {
      "Nonce": 3,
      "Account": "AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q",
      "Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
      "Amount": "1000000000000000000",
      "Payload": "",
      "Limit": 0,
      "Price": "0",
      "Type": 0,
      "Sign":
↪"381yXZAWNW8ZiST7PCLmywUsNVQvzgwhiyu9pxVDSvUhwyLHmoL7BQGXQhCNp7QZQycvbwT2nQJ7etscArfRbHu98Qi5MSmY
↪"
     }
    }
   ]
  }
 }
```

Sign transaction

After unlock the account

```
$ aergocli signtx --jsontx "{ \
\"Nonce\": 2, \
\"Account\": \"AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q\", \
```

(continues on next page)

(continued from previous page)

```
\"Recipient\": \"AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ\", \
\"Amount\": \"1.23aergo\", \
\"Payload\": \"\", \
\"Limit\": 0, \
\"Price\": \"0\", \
\"Type\": 0 }"
{
 "Hash": "9cAphBMD2zJCD13QfCn7rmxh5iDfrj6M9Wmo54TPNPCg",
 "Body": {
  "Nonce": 2,
  "Account": "AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q",
  "Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
  "Amount": "1230000000000000000",
  "Payload": "",
  "Limit": 0,
  "Price": "0",
  "Type": 0,
  "Sign":
↪"AN1rKvt8EZHKEE2wNPXAhGcDA4pMo7yRRjTWCcpyrW9QCMv6nMhvhqriWujHdaDJgJ6ft6VLDActEFtUFA2pRnRJFVFSWxPSR
↪"
 }
}
```

Commit Transaction

Send given transactions to **aergosvr**

```
$ aergocli -p 27845 committx --jsontx "{
\"Hash\": \"9cAphBMD2zJCD13QfCn7rmxh5iDfrj6M9Wmo54TPNPCg\",
\"Body\": {
  \"Nonce\": 2,
  \"Account\": \"AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q\",
  \"Recipient\": \"AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ\",
  \"Amount\": \"1230000000000000000\",
  \"Payload\": \"\",
  \"Limit\": 0,
  \"Price\": \"0\",
  \"Type\": 0,
  \"Sign\": \
↪"AN1rKvt8EZHKEE2wNPXAhGcDA4pMo7yRRjTWCcpyrW9QCMv6nMhvhqriWujHdaDJgJ6ft6VLDActEFtUFA2pRnRJFVFSWxPSR\
↪"
}
}"
{
 "results": [
  {
   "hash": "9cAphBMD2zJCD13QfCn7rmxh5iDfrj6M9Wmo54TPNPCg"
  }
 ]
}
```

Get Account state

Check account's state (nonce, balance)

```
$ aergocli getstate --address "AmNvFyqKFGVWvQ3MTi3eMFiNB9zvL9cK43B9c9bzcA732YZjZgfn"
```

Get state with a compressed merkle proof.

```
$ aergocli getstate --address "AmNvFyqKFGVWvQ3MTi3eMFiNB9zvL9cK43B9c9bzcA732YZjZgfn" -
↪-proof --compressed
```

By default, the returned state is the one at the latest block, but you may specify any past block's state root.

```
$ aergocli getstate --address "AmNvFyqKFGVWvQ3MTi3eMFiNB9zvL9cK43B9c9bzcA732YZjZgfn" -
↪-root "9NBSjkcNTdE5ciBxfb52RmsVW7vgX5voRsv6KcosiNjE"
```

Show connected peers

Use getpeers to show list of peers connected to a aergosvr.

It shows remote peers by default, but with `--self` options, local aergosvr itself is shown in list. You can find it by checking `Self` property.


```
[
 {
  "Address": {
   "Address": "13.124.83.51",
   "Port": "7846",
   "PeerId": "16Uiu2HAmQn3nFBGhJM7TnZRguLhgUx1HnpNL2easdt2JrxdbFjtb"
  },
  "BestBlock": {
   "BlockHash": "BXAFbTbwEuywukzBqRKdCsuUpinnNQsVJCAXNVJAR6F4",
   "BlockNo": 1251651
  },
  "LastCheck": "2019-02-27T11:26:38.481451+09:00",
  "State": "RUNNING",
  "Hidden": false,
  "Self": true
 },
 {
  "Address": {
   "Address": "13.211.156.203",
   "Port": "7846",
   "PeerId": "16Uiu2HAkvbHmK1Ke1hqAHmahwTGE4ndkdMdXJeXFE3kgBs17k2oQ"
  },
  "BestBlock": {
   "BlockHash": "AF68dtMMHd1h5LjxSyYY9AomMve6Qk2kBRSoQuuuSkhM",
   "BlockNo": 1251650
  },
  "LastCheck": "2019-02-27T11:26:38.349938+09:00",
  "State": "RUNNING",
  "Hidden": false,
  "Self": false
 },
 {
  "Address": {
   "Address": "203.109.12.23",
   "Port": "7846",
   "PeerId": "16Uiu2HAmAnQ5jjk7huhepfFtDFFCreuJ21nHYBApVpg8G7EBdwme"
  },
```


(continues on next page)

(continued from previous page)

```
  "BestBlock": {
   "BlockHash": "BXAFbTbwEuywukzBqRKdCsuUpinnNQsVJCAXNVJAR6F4",
   "BlockNo": 1251651
  },
  "LastCheck": "2019-02-27T11:26:38.364262+09:00",
  "State": "RUNNING",
  "Hidden": false,
  "Self": false
 }
]
```

11.1.4 Example without aergosvr

There are some feature working on **aergocli** itself without **aergosvr**.

Create, Export, Import account

With `--path` option, aergocli creates an account in the given path and not in **aergosvr**.

```
$ aergocli account new --password yourpasswordhere --path path/to/save/account
AmNFcocofUvmyLtXA6WgpANbjiF7RScGvQ4memNyNzS4ARJox3yq
```

Private key of account is store in the given path.

Of course this account can be exported and imported to aergosvr or another path.

```
$ aergocli account export --address␣
↪AmNFcocofUvmyLtXA6WgpANbjiF7RScGvQ4memNyNzS4ARJox3yq --password yourpasswordhere --
↪path path/to/save/account
47rsdfckuUCcjY3SmzCtmthhQm336Cpz9341xQHq6sr5Wm3md9FaTZDj6Gkqtff3WBPoqtzVV
```

```
$ aergocli account import --if␣
↪47rsdfckuUCcjY3SmzCtmthhQm336Cpz9341xQHq6sr5Wm3md9FaTZDj6Gkqtff3WBPoqtzVV --
↪password yourpasswordhere --path other/path/to/save/account
AmNFcocofUvmyLtXA6WgpANbjiF7RScGvQ4memNyNzS4ARJox3yq
```

Sign transaction

With `--path` option, aergocli can sign the transaction using private key of account in given path.

Unlike using aergosrv, parameter `--address` and `--password` are needed instead of unlock.

```
$ aergocli -p 17845 signtx --address␣
↪AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q --jsontx "{ \
\"Nonce\": 2, \
\"Account\": \"AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q\", \
\"Recipient\": \"AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ\", \
\"Amount\": \"1.23aergo\", \
\"Payload\": \"\", \
\"Limit\": 0, \
\"Price\": \"0\", \
\"Type\": 0 }" --path path/to/save/account --password yourpasswordhere
```

(continues on next page)

(continued from previous page)


```
{
 "Hash": "9cAphBMD2zJCD13QfCn7rmxh5iDfrj6M9Wmo54TPNPCg",
 "Body": {
  "Nonce": 2,
  "Account": "AmPLWBzx4tAYt91JM3jKWFs3aYWHSvKpYYzdUQuQMNa7jAw5t65q",
  "Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
  "Amount": "1230000000000000000",
  "Payload": "",
  "Limit": 0,
  "Price": "0",
  "Type": 0,
  "Sign":
↪"AN1rKvt8EZHKEE2wNPXAhGcDA4pMo7yRRjTWCcpyrW9QCMv6nMhvhqriWujHdaDJgJ6ft6VLDActEFtUFA2pRnRJFVFSWxPSR
↪"
 }
}
```


11.2 Aergoluac

aergoluac is a command line tool that compile lua script to binary code for contract.

For a list of all commands known to aergoluac, simply run it with no arguments.

```
Usage:
aergoluac [flags] srcfile bcfile

Flags:
-a, --abi string   abi filename
-h, --help         help for aergoluac
    --payload      print the compilation result consisting of bytecode and abi
```

11.2.1 Examples

out to file

```
aergoluac test.lua test.bc -a test.abi
```

out to console

```
aergoluac test.lua --payload
```

11.3 Ship

Ship is a tool for developing lua smart contracts. Its main features are:

- Manage project structure
- Install and publish packages

- Build source fragments
- Run unit test cases
- Deployment, execution and querying of a contract.

Please refer to the Ship Github Wiki for details.

11.4 Brick

Brick is an interactive shell program to communicate with an aergo VM for testing. This also provides a batch function to test and help to develop smart contracts.

The main advantage of using this tool is that you don't need a real blockchain network. Instead, you can run operations directly in the aergo virtual machine.

See Brick reference

11.5 Polaris

Polaris is a server that provides node discovery for Aergo server.

For bootstrapping new nodes, the npaddpeers configuration option can be used to connect to designated peers.

However, since Aergo version 0.11, the new Polarais server has added the ability to automatically connect to nodes belonging to the specified block chain without manually creating the node list information.

11.5.1 Features

- Aergo nodes can query addresses of other Aergo nodes. In this case, the chain of the Aergo node and the chain of Polaris must be the same.
- Aergo nodes can register itself with Polaris. Polaris checks to see if it can connect to the Aergo node and adds it to the node list.
- One Polaris server per designated block chain

11.5.2 Building Polaris

This section describes how to build Polaris from source without using the Docker. Polaris is available as a sub-module in the aergo project currently in version 0.11.

1. Get the source from github.com/aergoio/aergo.
2. Build the polaris executable with `make polaris`.

11.5.3 Configuration

Four files are used to set Polaris behavior.

1. Polaris configuration file: Determines the overall operation of Polaris. It also specifies the path to other configuration related files.
2. Private key file: PK file to use for Polaris communication. It uses the same format as aergosvr's private key file.

3. Genesis file: Contains the chain information of nodes to be provided by Polaris. Use the same format as the genesis file used to initialize aergosvr.
4. Log configuration file: The file name is arglog.toml, and it uses the same format as the file used by aergosvr.

Configuration file creation and example

Create private key file

It can be generated by aergocli using the keygen command.

```
aergocli keygen mychain-polaris
Wrote files mychain-polaris.{key,pub,id}.
```

Genesis file creation

```
{
    "chain_id":{
        "magic": "[insert an identifier string for your network]",
        "public": false,
        "mainnet": false,
        "coinbasefee": "1000000000",
        "consensus": "dpos"
    },
    "timestamp": 1548918000000000000,
    "balance": {
    },
    "bps": [
    ]
}
```

Polaris configuration file

```
[rpc]
netserviceaddr = "127.0.0.1"               # RPC access address. The default setting␣
↪is 127.0.0.1, which allows RPC access only on the local machine and blocks RPC␣
↪connections remotely.
netserviceport = 8915

[p2p]
netprotocoladdr = "[real IP address]"      # An externally accessible IP address or␣
↪domain name
netprotocolport = 8916
npbindaddr = ""
npkey = "mychain-polaris.key"              # Location of private key file

[polaris]
allowprivate = true                        # Whether to allow the private address of␣
↪the node's access address. Used when building Polaris for private chains operated␣
↪within a test or private network.
genesisfile = "[location of genesis file]" # Genesis file location
```

Log configuration file

Refer to the arglog documentation.

11.5.4 Running Polaris

Using Docker

```
docker run -d -w /tools -v /blockchain/polaris:/tools -p 8916:8916 -p 8915:8915 --
↪restart="always" --name polaris-node aergo/polaris polaris --home /tools --config /
↪tools/polaris-conf.toml
```

Manually

Manually build and run the live polaris executable in the following format:

```
./polaris --config polaris-conf.toml
```

11.5.5 Connecting to Polaris

To connect to a Polaris server, supply its address in the aergosvr configuration file.

See Node Configuration for details.

```
    [p2p]
    ...
    npusepolaris= true
    npaddpolarises = [
        "/ip4/192.168.0.2/tcp/8915/p2p/
↪16Uiu2HAmJCmxe7CrgTbJBgzyG8rx5Z5vybXPWQHHGQ7aRJfBsoFs"
    ]
...
```

11.5.6 Colaris

Colaris is a client for Polaris RPC connection.

Building colaris

Like Polaris, build as a sub-module of aergo.

1. Get the source from github.com/aergoio/aergo.
2. Build the executable with `make colaris`.

Usage

It is the same interface as aergocli.

```
./colaris [flags] <command> [[arg1]...]
```

Flags

1. `-H <hostname>` Address to remote server when requesting. The default value is localhost (127.0.0.1)
2. `-p <portnumber>` RPC port number, default is 8915

Commands

`node`: returns the actor state of Polaris.

`current`: returns list of nodes registered in Polaris.

Example:

```
ubuntu@mypolaris:/blockchain/polaris$ ./colaris -p 8915 current
{
 "total": 1,
 "peers": [
  {
   "address": {
    "address": "52.231.31.38",
    "port": 7846,
    "peerID": "16Uiu2HAmBfFABqQ2eWwNMv1A2WJCqVykgPS2sz72jrYTHeZgyors"
   },
   "connected": 1549526282,
   "lastCheck": 1549526463
  }
 ]
}
```

CHAPTER 12

API

The Aergo server exposes an RPC API over gRPC.

12.1 Using gRPC

gRPC is a standard for RPC APIs using protobuffer messages for data exchange. The main advantages are typed messages and efficiency: data is packed tightly and sent over HTTP2.

If you want to use the gRPC API directly, please refer to the official documentation to get started. You can find the protobuf definitions in aergoio/aergo-protobuf.

Alternatively, you can use one of the SDKs which wrap the same functionality in language-specific styles.

12.2 Protocol Documentation

This reference is auto-generated from aergoio/aergo-protobuf.

- *rpc.proto*
  - *AergoRPCService*
    - *NodeState*
    - *Metric*
    - *Blockchain*
    - *GetChainInfo*
    - *ChainStat*
    - *ListBlockHeaders*
    - *ListBlockMetadata*
    - *ListBlockStream*

    - *ListBlockMetadataStream*
    - *GetBlock*
    - *GetBlockMetadata*
    - *GetBlockBody*
    - *GetTX*
    - *GetBlockTX*
    - *GetReceipt*
    - *GetABI*
    - *SendTX*
    - *SignTX*
    - *VerifyTX*
    - *CommitTX*
    - *GetState*
    - *GetStateAndProof*
    - *CreateAccount*
    - *GetAccounts*
    - *LockAccount*
    - *UnlockAccount*
    - *ImportAccount*
    - *ExportAccount*
    - *QueryContract*
    - *QueryContractState*
    - *GetPeers*
    - *GetVotes*
    - *GetAccountVotes*
    - *GetStaking*
    - *GetNameInfo*
    - *ListEventStream*
    - *ListEvents*
    - *GetServerInfo*
    - *GetConsensusInfo*
    - *ChangeMembership*
  - *AccountAddress*
  - *AccountAndRoot*
  - *AccountVoteInfo*
  - *BlockBodyPaged*

- *BlockBodyParams*
- *BlockHeaderList*
- *BlockMetadata*
- *BlockMetadataList*
- *BlockchainStatus*
- *ChainId*
- *ChainInfo*
- *ChainStats*
- *CommitResult*
- *CommitResultList*
- *ConfigItem*
- *ConfigItem.PropsEntry*
- *ConsensusInfo*
- *Empty*
- *EventList*
- *ImportFormat*
- *Input*
- *KeyParams*
- *ListParams*
- *Name*
- *NameInfo*
- *NodeReq*
- *Output*
- *PageParams*
- *Peer*
- *PeerList*
- *PeersParams*
- *Personal*
- *ServerInfo*
- *ServerInfo.ConfigEntry*
- *ServerInfo.StatusEntry*
- *SingleBytes*
- *Staking*
- *VerifyResult*
- *Vote*
- *VoteInfo*

12.2.1 rpc.proto

AergoRPCService

AergoRPCService is the main RPC service providing endpoints to interact with the node and blockchain. If not otherwise noted, methods are unary requests.

NodeState

Request Type: NodeReq Response Type: SingleBytes

Returns the current state of this node

Metric

Request Type: MetricsRequest Response Type: Metrics

Returns node metrics according to request

Blockchain

Request Type: Empty Response Type: BlockchainStatus

Returns current blockchain status (best block's height and hash)

GetChainInfo

Request Type: Empty Response Type: ChainInfo

Returns current blockchain's basic information

ChainStat

Request Type: Empty Response Type: ChainStats

Returns current chain statistics

ListBlockHeaders

Request Type: ListParams Response Type: BlockHeaderList

Returns list of Blocks without body according to request

ListBlockMetadata

Request Type: ListParams Response Type: BlockMetadataList

Returns list of block metadata (hash, header, and number of transactions) according to request

ListBlockStream

Request Type: Empty Response Type: Block

Returns a stream of new blocks as they get added to the blockchain

ListBlockMetadataStream

Request Type: Empty Response Type: BlockMetadata

Returns a stream of new block's metadata as they get added to the blockchain

GetBlock

Request Type: SingleBytes Response Type: Block

Return a single block incl. header and body, queried by hash or number

GetBlockMetadata

Request Type: SingleBytes Response Type: BlockMetadata

Return a single block's metdata (hash, header, and number of transactions), queried by hash or number

GetBlockBody

Request Type: BlockBodyParams Response Type: BlockBodyPaged

Return a single block's body, queried by hash or number and list parameters

GetTX

Request Type: SingleBytes Response Type: Tx

Return a single transaction, queried by transaction hash

GetBlockTX

Request Type: SingleBytes Response Type: TxInBlock

Return information about transaction in block, queried by transaction hash

GetReceipt

Request Type: SingleBytes Response Type: Receipt

Return transaction receipt, queried by transaction hash

GetABI

Request Type: SingleBytes Response Type: ABI

Return ABI stored at contract address

SendTX

Request Type: Tx Response Type: CommitResult

Sign and send a transaction from an unlocked account

SignTX

Request Type: Tx Response Type: Tx

Sign transaction with unlocked account

VerifyTX

Request Type: Tx Response Type: VerifyResult

Verify validity of transaction

CommitTX

Request Type: TxList Response Type: CommitResultList

Commit a signed transaction

GetState

Request Type: SingleBytes Response Type: State

Return state of account

GetStateAndProof

Request Type: AccountAndRoot Response Type: AccountProof

Return state of account, including merkle proof

CreateAccount

Request Type: Personal Response Type: Account

Create a new account in this node

GetAccounts

Request Type: Empty Response Type: AccountList

Return list of accounts in this node

LockAccount

Request Type: Personal Response Type: Account

Lock account in this node

UnlockAccount

Request Type: Personal Response Type: Account

Unlock account in this node

ImportAccount

Request Type: ImportFormat Response Type: Account

Import account to this node

ExportAccount

Request Type: Personal Response Type: SingleBytes

Export account stored in this node

QueryContract

Request Type: Query Response Type: SingleBytes

Query a contract method

QueryContractState

Request Type: StateQuery Response Type: StateQueryProof

Query contract state

GetPeers

Request Type: PeersParams Response Type: PeerList

Return list of peers of this node and their state

GetVotes

Request Type: VoteParams Response Type: VoteList

Return result of vote

GetAccountVotes

Request Type: AccountAddress Response Type: AccountVoteInfo

Return staking, voting info for account

GetStaking

Request Type: AccountAddress Response Type: Staking

Return staking information

GetNameInfo

Request Type: Name Response Type: NameInfo

Return name information

ListEventStream

Request Type: FilterInfo Response Type: Event

Returns a stream of event as they get added to the blockchain

ListEvents

Request Type: FilterInfo Response Type: EventList

Returns list of event

GetServerInfo

Request Type: KeyParams Response Type: ServerInfo

Returns configs and statuses of server

GetConsensusInfo

Request Type: Empty Response Type: ConsensusInfo

Returns status of consensus and bps

ChangeMembership

Request Type: MembershipChange Response Type: MembershipChangeReply

Add & remove member of raft cluster

AccountAddress

AccountAndRoot

AccountVoteInfo

BlockBodyPaged

BlockBodyParams

BlockHeaderList

BlockMetadata

BlockMetadataList

BlockchainStatus

BlockchainStatus is current status of blockchain

ChainId

ChainInfo

ChainInfo returns chain configuration

ChainStats

ChainStats corresponds to a chain statistics report.

CommitResult

CommitResultList

ConfigItem

ConfigItem.PropsEntry

ConsensusInfo

info and bps is json string

Empty

EventList

ImportFormat

Input

KeyParams

ListParams

Name

NameInfo

NodeReq

Output

PageParams

Peer

PeerList

PeersParams

Personal

ServerInfo

ServerInfo.ConfigEntry

ServerInfo.StatusEntry

SingleBytes

Staking

VerifyResult

Vote

VoteInfo

VoteList

VoteParams

CommitStatus

VerifyStatus

12.2.2 blockchain.proto

ABI

AccountProof

Block

BlockBody

BlockHeader

ContractVarProof

Event

FilterInfo

FnArgument

Function

Query

Receipt

State

StateQuery

StateQueryProof

StateVar

Tx

TxBody

TxIdx

TxIdx specifies a transaction's block hash and index within the block body

TxInBlock

TxList

TxType

12.2.3 metric.proto

Metrics

MetricsRequest

PeerMetric

MetricType

12.2.4 Scalar Value Types