# aergo-guide Documentation

AERGO team and contributors

Jan 03, 2019

# Platform


**1 Introduction** **3**
- 1.1 Design goals . . . 3
- 1.2 Current features . . . 3
- 1.3 Roadmap . . . 4

**2 Dapp Development** **5**
- 2.1 Development cycle . . . 5
- 2.2 Design . . . 5
- 2.3 Clients . . . 5
- 2.4 Smart contracts . . . 6
- 2.5 Infrastructure . . . 6

**3 Running a Node** **7**
- 3.1 Quickstart . . . 7
- 3.2 Configuration . . . 7
- 3.3 Monitoring . . . 9
- 3.4 Troubleshooting . . . 10

**4 Using the Testnet** **15**
- 4.1 Connecting to Well-known Nodes . . . 15
- 4.2 Syncing . . . 16
- 4.3 Creating Accounts . . . 16
- 4.4 Funding Accounts . . . 16
- 4.5 Monitoring . . . 16

**5 Smart Contracts** **17**
- 5.1 Lua . . . 17
- 5.2 Aergo SQL . . . 36

**6 Technical Specifications** **37**
- 6.1 Transaction Types . . . 37
- 6.2 Transaction Fees . . . 38
- 6.3 Addresses . . . 38
- 6.4 Token Units . . . 38
- 6.5 Name System . . . 38
- 6.6 Consensus Algorithm . . . 39

**7 Contributing** **41**
7.1 Ways to Contribute . . . 41
7.2 Building from Source . . . 41

**8 Join the Community** **43**

**9 SDKs** **45**
9.1 Heraj . . . 45
9.2 Herajs . . . 45
9.3 Herapy . . . 46
9.4 Other Languages . . . 46

**10 Tools** **47**
10.1 Aergocli . . . 47
10.2 Aergoluac . . . 52
10.3 Ship . . . 53
10.4 Brick . . . 53
10.5 Hub Enterprise . . . 53

**11 API** **55**
11.1 Using gRPC . . . 55
11.2 Protocol Documentation . . . 55

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 the Aergo **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
```

If you want to override other command line options as well, you need to pass `--config` as well:

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

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

## 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

[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 = "0.0.0.0"
netprotocolport = 7846
# TLS and certificate is not applied in alpha release
nptls = false
npcert = ""
# Set file path of key file
npkey = ""
npaddpeers = [
]
npmaxpeers = "100"
nppeerpool = "100"

[blockchain]
# blockchain configurations
maxblocksize = 1048576

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

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

(continues on next page)

(continued from previous page)

```
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

## 3.3 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
                }
        },
```

(continues on next page)

(continued from previous page)


```
        "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",
                "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 Troubleshooting

### 3.4.1 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.

### 3.4.2 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",
```

(continues on next page)

(continued from previous page)

```
                "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
```

### 3.4.3 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": "",
                "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.

### 3.4.4 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 the Testnet

The current public Aergo testnet launched on December 28, 2018. These pages explain how you can connect to and use the testnet.

## 4.1 Connecting to Well-known Nodes

### 4.1.1 Clients

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.aergo.io blockchain
```

### 4.1.2 Server

Without any specific settings, the server connect to Polaris for the testnet, registers itself, obtains addresses of other nodes, and automatically attempts to connect those nodes. If the server is in the NAT environment, or has multiple NICs, additional settings are required for the external node to access the server. Set up outside address for other nodes in the external network connection, and set 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
```

## 4.2 Syncing

To access the Testnet, you must first have the same genesis-block as the public Testnet. For this, you create a genesis-block.

```
docker run -v $(pwd)/data:/aergo/data aergo/node aergosvr init /aergo/testnet-genesis.
↪json
```

When you run a server after creating genesis-block, the server automatically starts synchronizing after accessing the network.

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

## 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, you can create a local account:

   ```
   docker run --rm --net=host aergo/node aergocli account new
   ```

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

   ```
   docker run --rm aergo/tools aergocli keygen --json --password yourPassword
   ```

   Now you can see your new private key (shown in encrypted form) and the corresponding address. Using the encrypted private key and password, you can import this account to other nodes or wallets.

## 4.4 Funding Accounts

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

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 2.0.5 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 ABI of contract

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

```
./aergocli receipt get FPqA3kNQHoVXqKJv8JNpUSsh8F8id87yvRr5UzQFoCcH
{
 "contractAddress": "AmfzX3SHXVTBU9NSEWXaLxxjN11KsUpm1Gb3YjF7kmsHrgmL41WU",
 "status": "CREATED",
 "ret": "{''}"
}
```

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.

### 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

#### Restrictions

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.

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

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.

#### Libraries

We provide libraries for smart contract as follows:

- Blockchain API
  - system module
  - contract module
  - state module

  - db module
  - abi module
- Utils
  - json 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()`.

### SQL

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

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)
  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
```

(continues on next page)

(continued from previous page)

```
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
- receipt get
- 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 current contract transaction caller address

#### 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.

### getAmount()

This function returns a transaction containing balance

### setItem(key, value)

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

- restriction
  - key type string only
  - 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.

### 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", bugnum.number(
↪"999999999999999"))
```

### 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)
```

### 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.

### 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

```
contract.pcall(inc, 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.

### 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

#### 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.

- Default: https://www.sqlite.org/lang_corefunc.html
- Date and time: https://www.sqlite.org/lang_datefunc.html
- Set: 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)")
```

(continues on next page)

(continued from previous page)

```
    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.

### 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

### neq(x) (same as -x)

Negate bignum x and return as bignum

### sqrt(x)

Returns the square root of a positive number as bignum

#### 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

#### 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

```
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 SQL

Aergo SQL is a new language to write smart contracts, inspired by SQL. This makes it easier to think in terms of database operations while developing your contract and has a number of performance improvements.

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

CHAPTER 6

# Technical Specifications

## 6.1 Transaction Types

There are two kinds of transactions.

### 6.1.1 Normal type

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

### 6.1.2 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 | Account | Recipient | Amount | Payload | Type | Sign |
|---|---|---|---|---|---|---|
| staking | Sender | `aergo.system` | amount to stake | `s` | Governance | Signature of sender |
| unstaking | " | `aergo.system` | amount to unstake | `u` | " | " |
| voting | " | `aergo.system` | 0 | `v<peer ids bytes, no separator>` | " | " |
| create name | " | `aergo.name` | 0 | `c<name string>` | " | " |
| update name | " | `aergo.name` | 0 | `u<name string>,<new owner address>` | " | " |

## 6.2 Transaction Fees

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

Currently, the public testnet has a fixed fee of 1 gaer. This fee may change in the future.

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

## 6.3 Addresses

### 6.3.1 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`.

### 6.3.2 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.

## 6.4 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": "1000000000000000000"
}
```

## 6.5 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.

### 6.5.1 Using the name system

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

To register a new name:

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

To retrieve the owner of a registered name:

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

### 6.5.2 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.

## 6.6 Consensus Algorithm

The public Aergo network uses **Delegated Proof of Stake (DPoS)**. This article explains some of the main concepts and their implementation in Aergo.

**Block Producer (BP):** In Aergo, only a limited number of nodes generate blocks. They are called Block Producers. BPs are elected via users' voting, where the voting power is weighted by each voter's staked tokens. Based on this voting, BPs are re-elected round by round so that they can be changed over time. *The number of BPs and the number of stand-by BPs (candidates) have not been decided yet.*

**Staking:** DPoS uses a weighted voting system, where the voting power weighted by staked tokens. Hence, a voter must stake one's tokens for voting. *The detailed policy about staking/untaking has not been decided yet. This includes the minimum amount of staking and the mandatory days of staking.*

**Voting:** Any user with staked tokens can vote for BPs by using a voting transaction. But the voting results are not affected immediately. The current BPs are elected based on the voting result gathered at the block number: (<current block number> / <the total number of BPs> - 1) * <the total number of BPs>. In other words, the voting results gathered in the past (approximatedly 1 round before) are used for stability (recent blocks may be rollbacked via a reorganization).

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

## 7.2 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.

### 7.2.1 Linux

1. Install dependencies

```
apk update
apk add git glide 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
```

### 7.2.2 macOS

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

```
brew install go
brew install glide
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. Run server

```
./bin/aergosvr
```

### 7.2.3 Windows

Building on Windows is currently not supported.

### 7.2.4 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 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

# SDKs

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


## 9.1 Heraj

Please refer to the Github Wiki pages for Heraj.

## 9.2 Herajs

Please refer to the Herajs documentation.

## 9.3 Herapy

## 9.4 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 10

# Tools

## 10.1 Aergocli

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

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
committx    Send transaction
contract    contract command
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
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
```

(continues on next page)

(continued from previous page)

```
    --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.
```

### 10.1.1 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 AmQFgm1gCvoRw2RfBXnipRmeCLEc6tTQ1kBMmLEzHjp91xYnXK78 --to␣
↪AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ --amount 1000
75yMzPTS2oFJYvfj6QDxPsuYXVnwgQvKh1xaYgfuqGhJ TX_OK
```

#### Look up transaction

```
$ aergocli gettx 75yMzPTS2oFJYvfj6QDxPsuYXVnwgQvKh1xaYgfuqGhJ

Confirm:  {
"TxIdx": {
"BlockHash": "GGT9wahqcKKGKUncMuhRLLL3JaCs2MEBx7V8UdrK9JNi",
```

(continues on next page)

(continued from previous page)

```
"Idx": 0
},
"Tx": {
"Hash": "75yMzPTS2oFJYvfj6QDxPsuYXVnwgQvKh1xaYgfuqGhJ",
"Body": {
"Nonce": 1,
"Account": "AmQFgm1gCvoRw2RfBXnipRmeCLEc6tTQ1kBMmLEzHjp91xYnXK78",
"Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
"Amount": 1000,
"Payload": "",
"Limit": 0,
"Price": 0,
"Type": 0,
"Sign":
↪"AN1rKvt8ZQ3Dg7UU86NqmTPmwgmQTkp7WqMDGmqvYBXyYoDiLTx6WyCmhqTcYUMXBW5NYvCeDviYPWyMniEsnYsYz2AdXFCno
↪"
}
}
}
```

### Check block

```
$ aergocli getblock --hash GGT9wahqcKKGKUncMuhRLLL3JaCs2MEBx7V8UdrK9JNi

{
"Hash": "GGT9wahqcKKGKUncMuhRLLL3JaCs2MEBx7V8UdrK9JNi",
"Header": {
"PrevBlockHash": "49E5xQxnuDnhSa2qZNb59qDRd48d8vsWFQyxuc33DijV",
"BlockNo": 7454,
"Timestamp": 1540968429100585000,
"BlockRootHash": "",
"TxRootHash": "75yMzPTS2oFJYvfj6QDxPsuYXVnwgQvKh1xaYgfuqGhJ",
"Confirms": 0,
"PubKey": "",
"Sign": ""
},
"Body": {
"Txs": [
{
    "Hash": "75yMzPTS2oFJYvfj6QDxPsuYXVnwgQvKh1xaYgfuqGhJ",
    "Body": {
    "Nonce": 1,
    "Account": "AmQFgm1gCvoRw2RfBXnipRmeCLEc6tTQ1kBMmLEzHjp91xYnXK78",
    "Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
    "Amount": 1000,
    "Payload": "",
    "Limit": 0,
    "Price": 0,
    "Type": 0,
    "Sign":
↪"AN1rKvt8ZQ3Dg7UU86NqmTPmwgmQTkp7WqMDGmqvYBXyYoDiLTx6WyCmhqTcYUMXBW5NYvCeDviYPWyMniEsnYsYz2AdXFCno
↪"
    }
}
]
```

(continues on next page)

(continued from previous page)

```
}
}
```

### Sign transaction

After unlock the account

```
$ aergocli signtx --jsontx \
                        "{\"account\":\
↪"AmNBZ8WQKP8DbuP9Q9W9vGFhiT8vQNcuSZ2SbBbVvbJWGV3Wh1mn\", \
                        \"nonce\": 2 , \
                        \"price\": 1 , \
                        \"limit\": 100 , \
                        \"recipient\":\
↪"AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ\", \
                        \"type\": 0, \
                        \"amount\": 25000 }"
{
"Hash": "HB44gJvHhVoEfgiGq3VZmV9VUXfBXhHjcEvroBMkJGnY",
"Body": {
"Nonce": 2,
"Account": "AmNBZ8WQKP8DbuP9Q9W9vGFhiT8vQNcuSZ2SbBbVvbJWGV3Wh1mn",
"Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
"Amount": 25000,
"Payload": "",
"Limit": 100,
"Price": 1,
"Type": 0,
"Sign":
↪"381yXYxTtq2tRPRQPF7tHH6Cq3y8PvcsFWztPwCRmmYfqnK83Z3a6Yj9fyy8Rpvrrw76Y52SNAP6Th3BYQjX1Bcmf6NQrDHQ
↪"
}
}
```

### Commit Transaction

Send given transactions to **aergosvr**

```
$ aergocli committx --jsontx "{ \
\"Hash\": \"HB44gJvHhVoEfgiGq3VZmV9VUXfBXhHjcEvroBMkJGnY\", \
\"Body\": { \
\"Nonce\": 2, \
\"Account\": \"AmNBZ8WQKP8DbuP9Q9W9vGFhiT8vQNcuSZ2SbBbVvbJWGV3Wh1mn\", \
\"Recipient\": \"AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ\", \
\"Amount\": 25000, \
\"Payload\": \"\", \
\"Limit\": 100, \
\"Price\": 1, \
\"Type\": 0, \
\"Sign\": \
↪"381yXYxTtq2tRPRQPF7tHH6Cq3y8PvcsFWztPwCRmmYfqnK83Z3a6Yj9fyy8Rpvrrw76Y52SNAP6Th3BYQjX1Bcmf6NQrDHQ\
↪" \
} \
```

(continues on next page)

(continued from previous page)

```
}"
1 : HB44gJvHhVoEfgiGq3VZmV9VUXfBXhHjcEvroBMkJGnY TX_OK
```

#### 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"
```

### 10.1.2 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 signtx --address AmNFcocofUvmyLtXA6WgpANbjiF7RScGvQ4memNyNzS4ARJox3yq --
↪jsontx \
                    "{\"account\":\
↪"AmNBZ8WQKP8DbuP9Q9W9vGFhiT8vQNcuSZ2SbBbVvbJWGV3Wh1mn\", \
                    \"nonce\": 2 , \
                    \"price\": 1 , \
                    \"limit\": 100 , \
                    \"recipient\":\
↪"AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ\", \
                    \"type\": 0, \
                    \"amount\": 25000 }" --path path/to/save/account --password␣
↪yourpasswordhere
{
    "Hash": "HB44gJvHhVoEfgiGq3VZmV9VUXfBXhHjcEvroBMkJGnY",
    "Body": {
        "Nonce": 2,
        "Account": "AmNBZ8WQKP8DbuP9Q9W9vGFhiT8vQNcuSZ2SbBbVvbJWGV3Wh1mn",
        "Recipient": "AmLnVfGwq49etaa7dnzfGJTbaZWV7aVmrxFes4KmWukXwtooVZPJ",
        "Amount": 25000,
        "Payload": "",
        "Limit": 100,
        "Price": 1,
        "Type": 0,
        "Sign":
↪"381yXYxTtq2tRPRQPF7tHH6Cq3y8PvcsFWztPwCRmmYfqnK83Z3a6Yj9fyy8Rpvrrw76Y52SNAP6Th3BYQjX1Bcmf6NQrDHQ
↪"
    }
}
```

## 10.2 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
```

### 10.2.1 Examples

#### out to file

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

#### out to console

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

## 10.3 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.

## 10.4 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

## 10.5 Hub Enterprise

**Hub Enterprise** is a framework that can manage and monitor **Aergo blockchain** and the nodes that make up this blockchain for enterprise customer.

Hub Enterprise provides a Web UI that makes it easier to manage block chains for users.

Hub Enterprise offers three key features:

- Managing Blockchain & Blockchain node
- Monitoring Blockchain & Blockchain node
- Deploying and Managing Smart Contract

### 10.5.1 Managing Blockchain & Blockchain node

**Create blockchain**

The following user input arguments are required to create a blockchain:

- Name : Blockchain name
- bNodes : Number of nodes to be used for this blockchain

As many blockchain nodes as you enter are created, which run in one of a Docker container provided by the node provider. Blockchain nodes in one blockchain are joined by peer to each other, and are fully connected. Peer for each blockchain node can be set and checked in the config file.

After you create the block chain, you can use the returned IDs of blockchain nodes to check the information of each node. The Web address for viewing information on the blockchain node is as follows:

```
[Docker container IP]/bnode/[node ID]
```

Now you can use the blockchain in Hub Enterprise. You can manage and monitor the blockchain.

### 10.5.2 Monitoring Blockchain & Blockchain node

**Monitoring Blockchain**

You can monitor the created blockchain and monitor the following resources:

- Best Block for each Block Provider
- Transactions performed per minute
- CPU
- RAM
- Aergo log

### 10.5.3 Deploying and Managing Smart Contract

CHAPTER 11

# API

The aergo server exposes an RPC API over gRPC.

## 11.1 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.

## 11.2 Protocol Documentation

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

- *rpc.proto*
  - *AergoRPCService*
  - *AccountAndRoot*
  - *BlockHeaderList*
  - *BlockMetadata*
  - *BlockMetadataList*
  - *BlockchainStatus*
  - *CommitResult*
  - *CommitResultList*
  - *Empty*

  - *ImportFormat*
  - *Input*
  - *ListParams*
  - *Name*
  - *NameInfo*
  - *NodeReq*
  - *Output*
  - *Peer*
  - *PeerList*
  - *Personal*
  - *SingleBytes*
  - *Staking*
  - *VerifyResult*
  - *Vote*
  - *VoteList*
  - *CommitStatus*
  - *VerifyStatus*
- *blockchain.proto*
  - *ABI*
  - *Block*
  - *BlockBody*
  - *BlockHeader*
  - *ContractVarProof*
  - *FnArgument*
  - *Function*
  - *Query*
  - *Receipt*
  - *State*
  - *StateProof*
  - *StateQuery*
  - *StateQueryProof*
  - *StateVar*
  - *Tx*
  - *TxBody*
  - *TxIdx*
  - *TxInBlock*

  - *TxList*
  - *TxType*
- *metric.proto*
  - *Metrics*
  - *MetricsRequest*
  - *PeerMetric*
  - *MetricType*
- *Scalar Value Types*


### 11.2.1 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.

#### AccountAndRoot

#### BlockHeaderList

#### BlockMetadata

#### BlockMetadataList

#### BlockchainStatus

BlockchainStatus is current status of blockchain

**CommitResult**

**CommitResultList**

**Empty**

**ImportFormat**

**Input**

**ListParams**

**Name**

**NameInfo**

**NodeReq**

**Output**

**Peer**

**PeerList**

**Personal**

**SingleBytes**

**Staking**

**VerifyResult**

**Vote**

**VoteList**

**CommitStatus**

**VerifyStatus**

### 11.2.2 blockchain.proto

**ABI**

**Block**

**BlockBody**

**BlockHeader**

**ContractVarProof**

**FnArgument**

**Function**

**Query**

**Receipt**

#### TxInBlock

#### TxList

#### TxType

### 11.2.3 metric.proto

#### Metrics

#### MetricsRequest

#### PeerMetric

#### MetricType

### 11.2.4 Scalar Value Types