Monetd Documentation

Release 0

Mosaic Networks

Sep 13, 2019

Contents

In this document we explain our implementation of the MONET Hub; in particular the mechanism that dictates who can participate in the consensus system, and how to make participants accountable for their actions. Before deliberating on an implementation, it is important to have a clear picture of the desired outcome. So we will start by reiterating the role of the Hub in MONET, and outline its principal requirements. We then visit the spectrum of potential implementations before explaining our choice of a permissioned Byzantine Fault Tolerant (**BFT**) consensus algorithm coupled to the Ethereum Virtual Machine (**EVM**). Lastly we weigh up the pros and cons of Proof of Stake (**PoS**), and explain our decision to implement Proof of Authoriry (**PoA**) for the time being.

MONET and the MONET Hub

MONET's mission is to boost the adoption of peer-to-peer architectures by enabling mobile devices to connect directly to one another in dynamic ad-hoc networks. We believe that a new generation of applications will emerge from this technology. The real force behind MONET, which makes it original and disruptive, is the concept of **Mobile Ad-Hoc Blockchains**, and the open-source software which implements it; particularly Babble, the powerful consensus algorithm which is suitable for mobile deployments due to its speed, bandwidth efficiency, and leaderlessness.

We anticipate that many MONET applications will require a common set of services to persist non-transient data, carry information across ad-hoc blockchains, and facilitate peer-discovery. So we set out to build the MONET Hub, an additional public utility that provides these services. In the spirit of open architecture, MONET doesn't rely on any central authority, so anyone is free to implement their own alternative, but the MONET Hub is there to offer a reliable, fast, and secure solution to kickstart the system.

As such, the qualitative requirements of the Hub are:

- **Speed**: It should support thousands of commands per second, with latencies under one second.

- **Finality**: Results from the hub should be definitive, without the possibility of being arbitrarily overridden in the future.

- **Availability**: It should provide a continuous service in the face of network failures or isolated disruptions.

- **Cost**: As we want to lower the barrier to entry for developers, using the Hub should be cheaper than rolling out one's own solution.

- **Security**: The hub should provide a trusted source of data and computation, with measures guarding against information loss, data manipulation, or censorship.

- **Governance**: The set of entities controlling this utility should be transparent, with a mechanism to add or remove participants, and keep them accountable for their actions.

- **Flexibility**: It should be possible and relatively easy to update the software, recover from failures, and adapt to changes.

Spectrum of possible Implementations

From a simple web-service hosted on a privately-owned server, to a public global blockchain like Ethereum, there are many potential ways to implement this service. However, given our requirements, a simple server scores pretty low in all categories (except perhaps speed and flexibility), and global public blockchains are too slow, too hard to update, and usually provide only probabilistic finality, which is not acceptable.

Somewhere in the middle lies a category of distributed systems consisting of relatively small clusters of servers maintaining identical copies of an application via sophisticated communication routines and consensus algorithms. Within this category, there are instances where the entire cluster is controlled by a single entity, and others where each replica is controlled by a different entity.

Modern blockchain projects, including cryptocurrencies like Facebook's Libra and the Cosmos Atom, adopt the second variant, where nodes are controlled by different entities. A naive implementation would render them vulnerable to malicious actors trying to subvert the system; hence they require strong consensus algorithms, commonly referred to as Byzantine Fault Tolerant (BFT), and a reputation system to incentivize good behavior and punish malicious actors.

Given the requirements stated in the previous section, we believe that the MONET Hub falls in the same category, and requires a permissioned BFT system.

Ethereum with Babble Consensus

We have developed the Monet Toolchain, a complete set of software tools for setting up and using the MONET Hub. This includes `monetd`, the software daemon that powers nodes on the MONET Hub.

To build `monetd`, we used our own BFT consensus algorithm, Babble, because it is fast, leaderless, and offers finality. For the application state and smart-contract platform, we use the Ethereum Virtual Mahcine (EVM) via EVM-Lite, which is a stripped down version of Go-Ethereum.

The EVM is a security-oriented virtual machine specifically designed to run untrusted code on a network of computers. Every transaction applied to the EVM modifies the State which is persisted in a Merkle Patricia tree. This data structure allows to simply check if a given transaction was actually applied to the VM and can reduce the entire State to a single hash (merkle root) rather analogous to a fingerprint.

The EVM is meant to be used in conjunction with a system that broadcasts transactions across network participants and ensures that everyone executes the same transactions in the same order. Ethereum uses a Blockchain and a Proof of Work consensus algorithm. EVM-Lite makes it easy to use any consensus system, including Babble.

The remaining question is how to govern the validator-set, and what to use as a reputation system to punish or incentivise participants to behave correctly.

PoS and PoA

A BFT consensus algorithm ensures that a distributed system remains available and consistent in adversarial conditions, with some nodes exhibiting arbitrary failures or malicious behavior, as long as a majority of participants are functioning correctly (actually). Any trust in the system therefore depends on the ability to legitimise this assumption. What is needed is a mechanism to ensure, with a high degree of confidence, that at least two thirds of participants in the consensus system are functioning correctly at all times. The problem is two-fold: who gets to be a participant, and how are participants incentivised to behave correctly? Not surprisingly, the most convincing answers revolve around money or reputational risk.

In a Proof of Stake (PoS) arrangement, participants are required to lock a significant portion of their assets (usually the blockchain's built-in token), and respect an extended un-bonding period when they want to leave. At any given time, the validator set is defined by the top N stakers, where N is the desired size of the validator-set. If they are caught undermining the network, this deposit is destroyed. Hence, participants are deterred from cheating. Additionally, participants are usually programmatically compensated for actively participating in securing the network. Hence they are incentivised to act correctly. A nice feature of PoS is that, being a very capitalistic model, it is relatively open; anyone can participate without asking for permission, as long as they put up a stake.

In Proof of Authority (PoA), the stake is tied to reputational risk. It relies on the natural aversion of most humans to tarnish their own reputation. The list of allowed validators is governed by a whitelist. The whitelist is amended through a voting process among existing whitelisted entities. This scheme is less anonymous or open than PoS but has deep roots. The trust of a PoA system rests on the initial group of participants because any amendment to the list has to gather consensus from them; so the trust (or distrust) is carried over as the validator-set evolves. In a system like Babble, the most serious offence consists in signing two different blocks at the same height. Evidence of this can be packaged into an irrefutable proof, and used to punish the guilty participants.

Proof of Stake opens exciting opportunities for a variety of stakeholders, and these economic incentives are excellent for the industry as they drive innovation. That being said, we are of the opinion that it is too early to ascertain the resilience of PoS in the face of decisive attacks, as current production deployments are very recent, and the theoretical arguments alone are not sufficiently convincing (although they sound quite reasonable). We are keeping an eye on PoS systems, hoping that they withstand the test of time. In the meantime, we have opted to implement PoA, to roll out a reliable version of the MONET Hub, with an eye on extending to PoS in a coordinated software update later down the road.

Conclusion

The MONET Hub is a pivotal utility that facilitates the creation of mobile ad-hoc blockchains, and the emergence of a new breed of decentralised applications. To maximise the performance, security, and flexibility of this system, we have opted to build the Monet Toolchain, a smart-contract platform based on the Ethereum Virtual Machine and a state-of-the-art BFT consensus algorithm, Babble. To govern the validator-set involved in the consensus algorithm, we have chosen to implement a Proof of Authority system, with the idea of extending to Proof of Stake when more evidence of its efficacy becomes available.

Overview

This document describes the tools for operating a Monet Toolchain node, and a couple of important concepts regarding the account model. In other documents, we provide guidance on using these tools to perform common tasks, as well as a complete reference of commands and API functions.

6.1 Tools

6.1.1 Monetd

monetd is the server process that connects to other nodes, participates in the consensus algorithm, and maintains its own copy of the application state. Additionaly, the **giverny** program facilitates the creation of local Monet Toolchain networks for testing purposes. We don't expect most people to use **giverny** as it is mostly a development tool.

monetd and **giverny** are written in Go, and reside in the same github repository because they share significant source code. Please follow the *installation instructions* to get started.

6.1.2 Monetcli

monetcli is the client-side program that interacts with a running Monet Toolchain node, and enables users to make transfers, query accounts, deploy and call smart-contracts, or participate in the PoA governance mechanism. `monetcli` is a Node.js project. It can be installed easily with `npm install -g monetcli`.

6.2 Accounts

6.2.1 What is an account?

The Monet Toolchain, and thus MONET, uses the same account model as Ethereum. Accounts represent identities of external agents and are associated with a balance (and storage for Contract accounts). They rely on public key cryptography to sign transactions so that the EVM can securely validate the identity of a transaction sender.

Using the same account model as Ethereum doesn't mean that existing Ethereum accounts automatically have the same balance in MONET (or vice versa). In Ethereum, balances are denoted in Ether, the cryptocurrency maintained by the public Ethereum network. On the other hand, every MONET network (even a single node network) maintains a completely separate ledger and may use any name for the corresponding coin. The official MONET token is Tenom.

What follows is mostly inspired from the Ethereum Docs:

Accounts are objects in the EVM State. They come in two types: Externally owned accounts, and Contract accounts. Externally owned accounts have a balance, and Contract accounts have a balance and storage. The EVM State is the state of all accounts which is updated with every transaction. The underlying consensus engine ensures that every participant in a Monet Toolchain network processes the same transactions in the same order, thereby arriving at the same State. The use of Contract accounts with the EVM makes it possible to deploy and use *SmartContracts* which we will explore in another document.

6.2.2 What is an account file?

This is best explained in the Ethereum Docs:

> Every account is defined by a pair of keys, a private key, and public key. Accounts are indexed by their address which is derived from the public key by taking the last 20 bytes. Every private key/address pair is encoded in a keyfile. Keyfiles are JSON text files which you can open and view in any text editor. The critical component of the keyfile, your account's private key, is always encrypted, and it is encrypted with the password you enter when you create the account.

6.3 Transactions

A transaction is a signed data package that contains instructions for the EVM. It can contain instructions to move coins from one account to another, create a new Contract account, or call an existing Contract account. Transactions are encoded using the custom Ethereum scheme, RLP, and contain the following fields:

- The recipient of the message.

- A signature identifying the sender and proving their intention to send the transaction.

- The number of coins to transfer from the sender to the recipient.

- An optional data field, which can contain the message sent to a contract.

- A STARTGAS value, representing the maximum number of computational steps the transaction execution is allowed to take.

- a GASPRICE value, representing the fee the sender is willing to pay for gas. One unit of gas corresponds to the execution of one atomic instruction, i.e., a computational step.

Installing monetd

7.1 Versioning

`monetd` versions follow semantic versioning. As we are still in the 0.x range, different versions might contain undocumented and/or breaking changes. At this stage, the prefered way of installing `monetd` is building from source, or using our public Docker images.

7.2 Docker

Docker images of `monetd` are available from the `mosaicnetworks` organisation. Use the `latest` tag for the latest released version. The advantage of using Docker containers is that they come packaged with all the necessary binary files, including solc, and contain an isolated running environment where `monetd` is sure to run.

Example: Mount a configuration directory, and run a node from inside a `monetd` container.

```
docker run --rm -v ~/.monet:/.monet mosaicnetworks/monetd run
```

7.3 Downloads

Binary packages of `monetd` will be available from github.

7.4 Building From Source

7.4.1 Dependencies

The key components of the Monet Toolchain, which powers the MONET Hub, are written in Golang. Hence, the first step is to install **Go version 1.9 or above**, which is both the programming language and a CLI tool for managing Go

code. Go is very opinionated and requires defining a workspace where all Go code resides. The simplest test of a Go installation is:

```
$ go version
```

`monetd` uses Glide to manage dependencies.

```
$ curl https://glide.sh/get | sh
```

Solidity Compiler

The Monet Toolchain uses Proof of Authority (PoA) to manage the validator set. This is implemented using a smart-contract written in Solidity, with the corresponding EVM bytecode set in the genesis file. For every newly defined network, the smart-contract needs to be recompiled because it embeds the initial whitelist. Hence, the Solidity compiler (solc) is a requirement to define a new network and produce the appropriate genesis file.

Please refer to the solidity compiler installation instructions.

Attention: The Node.js version of the compiler is not supported. **Do not install via** `npm install solc`.

Other requirements

Bash scripts used in this project assume the use of GNU versions of coreutils. Please ensure you have GNU versions of these programs installed:-

example for macOS:

```
# --with-default-names makes the `sed` and `awk` commands default to gnu sed and gnu
↪awk respectively.
brew install gnu-sed gawk --with-default-names
```

7.4.2 Installation

Clone the repository in the appropriate GOPATH subdirectory:

```
$ mkdir -p $GOPATH/src/github.com/mosaicnetworks/
$ cd $GOPATH/src/github.com/mosaicnetworks
[...]/mosaicnetworks$ git clone https://github.com/mosaicnetworks/monetd.git
```

Run the following command to download all dependencies and put them in the **vendor** folder.

```
[...]/monetd$ make vendor
```

Then build and install:

```
[...]/monetd$ make install
```

Getting Started

In this document we explain how to run a single node and how to use `monetcli` to interact with it. In another section, we will explain how to join an existing network. For details about any command, please refer to the *specification*.

8.1 Creating A Single Node Network

In short, run the following three commands to start a standalone node:

```
$ monetd keys new node0
$ monetd config build node0
$ monetd run
```

The `keys new` command will prompt us for a password, and generate a new encrypted keyfile in the default keystore `~/.monet/keystore`. We identified our key with the `node0` moniker.

The `config build` command takes our key, and generates a minimal network configuration with a single validator node, and a prefunded account. Again, the configuration is written to `~/.monet`.[1]

Finally, the `run` command starts a monetd node, which will default to using the configuration files in `~/.monet`.[1]

8.2 Using monetcli

Let's use `monetcli` to query the newly created node. First of all, install `monetcli` with `npm install -g monetcli`.

While `monetd` is still running, open another terminal and start `monetcli` in interactive mode:

---

[1] This location is for Linux instances. Mac and Windows uses a different path. The path for your instance can be ascertain with this command: `monetd config location`

```
$monetcli i


 __  __                    _         ____ _      ___
|  \/  |  ___  _ __    ___ | |_      / ___| | |     |_ _|
| |\/| | / _ \ | '_ \  / _ \| __|   | |     | |      | |
| |  | || (_) || | | ||  __/| |_    | |___  | |___   | |
|_|  |_| \___/ |_| |_| \___| \__|    \____| |_____| |___|


Mode:        Interactive
Data Dir:    /home/user/.monet
Config File: /home/user/.monet/monetcli.toml
Keystore:    /home/user/.monet/keystore


 Commands:

  help [command...]                Provides help for a given command.
  exit                             Exits application.
  accounts create [options]        Creates an encrypted keypair locally
  accounts get [options] [address] Fetches account details from a connected node
  accounts list [options]          List all accounts in the local keystore␣
→directory
  accounts update [options] [address] Update passphrase for a local account
  accounts import [options]        Import an encrypted keyfile to the keystore
  config set [options]             Set values of the configuration inside the␣
→data directory
  config view [options]            Output current configuration file
  poa check [options] [address]    Check whether an address is on the whitelist
  poa info [options]               Display Proof of Authority information
  poa nominate [options] [address] Nominate an address to proceed to election
  poa nomineelist [options]        List nominees for a connected node
  poa vote [options] [address]     Vote for an nominee currently in election
  poa whitelist [options]          List whitelist entries for a connected node
  transfer [options]               Initiate a transfer of token(s) to an address
  info [options]                   Display information about node
  version [options]                Display current version of cli
  debug                            Toggle debug mode
  clear                            Clear output on screen
```

Type info to check the status of the node:

```
monetcli$ info
.------------------------------------.
|        Key          |    Value    |
|---------------------|-------------|
| consensus_events    | 0           |
| consensus_transactions | 0        |
| events_per_second   | 0.00        |
| id                  | 1022922485  |
| last_block_index    | -1          |
| last_consensus_round | nil        |
| moniker             | node0       |
| num_peers           | 1           |
| round_events        | 0           |
| rounds_per_second   | 0.00        |
| state               | Babbling    |
| sync_rate           | 1.00        |
| transaction_pool    | 0           |
```

(continues on next page)

```
| type                  | babble     |
| undetermined_events   | 0          |
'------------------------------------'
```

Type `accounts list` to get a list of accounts in the keystore, and the balance associated with them.

```
monetcli$ accounts list
.-------------------------------------------------------------------------------
↪--.
| Moniker |                 Address                |          Balance           |␣
↪Nonce |
|---------|----------------------------------------|----------------------------|-----
↪--|
| node0   | 0xa10aae5609643848fF1Bceb76172652261dB1d6c | 12345678900000000000000 |    ␣
↪0 |
'-------------------------------------------------------------------------------
↪--'
```

So we have a prefunded account. The same account is used as a validator in Babble, and as a Tenom-holding account in the ledger. This is the same account, node0, that we created in the previous steps, with the encrypted private key residing in `~/.monet/keystore`.

Now, let's create a new key using `monetcli`, and transfer some tokens to it.

```
monetcli$ accounts create
? Moniker:  node1
? Output Path:  /home/user/.monet/keystore
? Passphrase:  [hidden]
? Re-enter passphrase:  [hidden]
{"version":3,"id":"89970faf-8754-468e-903c-c9d3248a08cc","address":
↪"960c13654c477ac1d2d7f8fc7ae84d93a2225257","crypto":{"ciphertext":
↪"7aac819c1bed442d778
97b690e5c2f14416589c7bdd6bdd2b5df5d03584ce0ec","cipherparams":{"iv":
↪"3d15a67d76293c3b7123f2bde76ba120"},"cipher":"aes-128-ctr","kdf":"scrypt","kdfparams
↪"
:{"dklen":32,"salt":"730dd67f175a77c9833a230e334719292cbb735607795b1b84484e3d04783510
↪","n":8192,"r":8,"p":1},"mac":"7535c31c277a698207d278cd1f1df90747463
e390b822cfef7d2faf8f1fa1809"}}
```

Like `monetd keys new` this command created a new key and wrote the encrypted keyfile in ~/.monet/keystore. Let's double check that the key was created:

```
monetcli$ accounts list
.-------------------------------------------------------------------------------
↪--.
| Moniker |                 Address                |          Balance           |␣
↪Nonce |
|---------|----------------------------------------|----------------------------|-----
↪--|
| node0   | 0xa10aae5609643848fF1Bceb76172652261dB1d6c | 12345678900000000000000 |    ␣
↪0 |
| node1   | 0x960c13654c477ac1d2d7f8fc7ae84d93a2225257 | 0                          |    ␣
↪0 |
'-------------------------------------------------------------------------------
↪--'
```

Now, let's transfer 100 tokens to it.

```
monetcli$ transfer
? From:  node0 (1,234,567,890,000,000,000,000)
? Enter password:  [hidden]
? To 0x960c13654c477ac1d2d7f8fc7ae84d93a2225257
? Value:  100
? Gas:  1000000
? Gas Price:  0
{
  "from": "0xa10aae5609643848fF1Bceb76172652261dB1d6c",
  "to": "0x960c13654c477ac1d2d7f8fc7ae84d93a2225257",
  "value": 100,
  "gas": 1000000,
  "gasPrice": 0
}
? Submit transaction Yes
Transaction submitted successfully.
```

Finally, we can check the account balances again to verify the outcome of the transfer:

```
monetcli$ accounts list
.------------------------------------------------------------------------------
↪--.
| Moniker |                 Address                 |         Balance         |␣
↪Nonce |
|---------|-----------------------------------------|-------------------------|-----
↪--|
| node0   | 0xa10aae5609643848fF1Bceb76172652261dB1d6c | 1234567889999999999900 |   ␣
↪1 |
| node1   | 0x960c13654c477ac1d2d7f8fc7ae84d93a2225257 | 100                     |   ␣
↪0 |
'------------------------------------------------------------------------------
↪--'
```

Joining a Network

This section describes how to join an existing network that is already running, such as the one created in *Getting Started*.

Here's a summary of the steps required to join an existing network built with the Monet Toolchain:

```
$ monetd keys new node1
$ monetd config pull [address]:[port] --key node1
$ monetcli poa nominate -h [address] -p [port] --from [node1 address] --pwd [password␣
↪file for node1 key] --moniker node1 [node1 address]

# wait to be accepted in the whitelist, which can be checked with
$ monetd poa whitelist
# or
$ monetd poa nomineelist

$ monetd run
```

Where [address] and [port] correspond to the endpoint of an existing peer in the network.

This scenario is designed to be run on a machine other than the one that is running the existing node.

9.1 Create An Account

We need to generate a new key-pair for our account:

```
$ monetd keys new node1
Passphrase:
Repeat passphrase:
Address: 0x5a735fC1235ce1E60eb5f9B9BCacb643a9Da27F4
```

9.2 Pull the Configuration From an Existing Node

We now pull the `monetd` configuration files from an existing peer. The syntax for this command is:

```
$ monetd config pull [peer] [--key] [--address]
```

The peer parameter is the address/IP of an existing node on the network. The network's configuration is requested from this peer. If the address does not specify a port, the default API port (8080) is assumed.

We also need to specify the IP address of our own node. For a live network that would clearly be a public IP address, but for an exploratory testnet, we would recommend using an internal IP address. On Linux `ifconfig` will give you IP address information. This can be set by using the –address flag. If not specified `monetd` will pick the first non-loopback address.

The `--key` parameter specifies the keyfile to use by moniker.

Thus we need to run the following command, but replace `192.168.1.5:8080` with the endpoint of the existing peer.

```
$ monetd config pull 192.168.1.5:8080 --key node1
```

9.3 Apply to Join the Network

If we tried to run `monetd` at this stage, it would not be allowed to join the other node because it isn't whitelisted yet. So we need to apply to the whitelist first.

We do so with the `monetcli poa nominate` command. The syntax is:

```
$ monetcli poa nominate -h <existing node> --from <moniker> --moniker <nominee
→moniker> --pwd <passphrase file> <nominee address>
```

But we can also do it interactively. **On the existing instance (node0), run the following interactive ``monetcli`` session**:

```
 monetcli i
  __  __                                  _       ____    _        ___
 |  \/  |    ___    _ __      ___    | |_     / ___|  | |      |_   _|
 | |\/| |   / _ \  | '_ \    / _ \  | __|   | |     | |        | |
 | |  | |  | (_) | | | | |  |  __/  | |_    | |___   | |___     | |
 |_|  |_|   \___/  |_| |_|   \___|   \__|    \____|  |_____|   |___|

Mode:        Interactive
Data Dir:    /home/user/.monet
Config File: /home/user/.monet/monetcli.toml
Keystore:    /home/user/.monet/keystore

 Commands:
  [...]


 monetcli$ poa nominate
 ? From:  node0
 ? Passphrase:  [hidden]
 ? Nominee:  0x960c13654c477ac1d2d7f8fc7ae84d93a2225257
 ? Moniker:  node1
```

```
You (0xa10aae5609643848ff1bceb76172652261db1d6c) nominated 'node1'␣
→(0x960c13654c477ac1d2d7f8fc7ae84d93a2225257)

monetcli$ poa nomineelist
.------------------------------------------------------------------------------.
| Moniker |                   Address                    | Up Votes | Down Votes |
|---------|----------------------------------------------|----------|------------|
| Node1   | 0x960c13654c477ac1d2d7f8fc7ae84d93a2225257 |        0 |          0 |
'------------------------------------------------------------------------------'
```

Now that, we have applied to the whitelist (via node0), we need all the entities in the current whitelist to vote for us.
At the moment, only node0 is in the whitelist, so let's cast a vote.

```
monetcli$ poa whitelist
.----------------------------------------------------.
| Moniker |                 Address                  |
|---------|------------------------------------------|
| Node0   | 0xa10aae5609643848ff1bceb76172652261db1d6c |
'----------------------------------------------------'

monetcli$ poa vote
? From:  node0
? Passphrase:  [hidden]
? Nominee:  0x960c13654c477ac1d2d7f8fc7ae84d93a2225257
? Verdict:  Yes
You (0xa10aae5609643848ff1bceb76172652261db1d6c) voted 'Yes' for
→'0x960c13654c477ac1d2d7f8fc7ae84d93a2225257'.
Election completed with the nominee being 'Accepted'.

monet$ poa whitelist
.----------------------------------------------------.
| Moniker |                 Address                  |
|---------|------------------------------------------|
| Node0   | 0xa10aae5609643848ff1bceb76172652261db1d6c |
| Node1   | 0x960c13654c477ac1d2d7f8fc7ae84d93a2225257 |
'----------------------------------------------------'
```

Finaly node1 made it into the whitelist.

9.4 Starting the Node

To start node1, run the simple `monetd run` command. You should be able see the JoinRequest going through
consensus, and being accepted by the PoA contract.

```
$ monetd run
```

Transaction Fees

Every operation that modifies the state (transfer, smart-contract creation, smart-contract call, etc.) carries a cost. Within the EVM, this cost is denominated in gas. For example, a simple transfer costs 21000 gas. When users create and submit transactions, they can set the maximum amount of gas they want to spend, and how much `om` (10^-18 Tenom) they are willing to pay per unit of gas consumed. Therefore, if their transaction is applied, it will cost them a transaction fee of `gas-price * gas-consumed`, which is capped by `gas-price * gas-max`.

Transaction fees serve a dual purpose: to incentivise validators, and to prevent denial of service attacks.

10.1 Distribution Among Validators

Every transaction applied to the EVM is associated with a coinbase address (possibly empty), which receives the transaction fee. In monetd, we have implemented a system that fairly and securily distributes fees among validators.

Upon committing a Babble block, we fetch the corresponding validator-set from Babble. Then we use the block hash to obtain a pseudo-random number which we use to select a peer from the validator-set. This peer will receive all the transaction fees from that block. This system is fair and secure because the selection process is evenly distributed and it is impossible for malicious validators to game it by manipulating the block hash.

10.2 Minimum Gas Price

Validators running a monetd node can set a minimum gas price, via the `eth.min-gas-price` configuration flag, to refuse broadcasting transactions with lower gas-prices. To send a transaction via a node, the transaction creator must set the gas price to a value greater or equal to that node's minimum gas price. Note that this filtering is done at the service layer, so it will not prevent other nodes from including cheaper transactions.

Monetd Documentation, Release 0

26 **Chapter 10. Transaction Fees**

Monetd API

`monetd` exposes an HTTP API at the address specified by the `--api-listen` flag. This document contains the API specification with some basic examples using curl. For API clients (javascript libraries, CLI, and GUI), please refer to Monet CLI

11.1 Get Account

Retrieve information about any account.

```
GET /account/{address}
returns: JsonAccount
```

```
type JsonAccount struct {
  Address string   `json:"address"`
  Balance *big.Int `json:"balance"`
  Nonce   uint64   `json:"nonce"`
  Code    string   `json:"bytecode"`
}
```

Example:

```
host:~$ curl http://localhost:8080/account/0xa10aae5609643848fF1Bceb76172652261dB1d6c
→-s | jq
{
    "address": "0xa10aae5609643848fF1Bceb76172652261dB1d6c",
    "balance": 1234567890000000000000,
    "nonce": 0,
    "bytecode": ""
}
```

11.2 Call

Call a smart-contract READONLY function. These calls will NOT modify the EVM state, and the data does NOT need to be signed.

```
POST /call
data: JSON SendTxArgs
returns: JSON JsonCallRes
```

```
type SendTxArgs struct {
    From     common.Address  `json:"from"`
    To       *common.Address `json:"to"`
    Gas      uint64          `json:"gas"`
    GasPrice *big.Int        `json:"gasPrice"`
    Value    *big.Int        `json:"value"`
    Data     string          `json:"data"`
    Nonce    *uint64         `json:"nonce"`
}

type JsonCallRes struct {
    Data string `json:"data"`
}
```

Example:

```
curl http://localhost:8080/call \
-d '{"constant":true,"to":"0xabbaabbaabbaabbaabbaabbaabbaabbaabbaabba","value":0,"data
→":"0x8f82b8c4","gas":1000000,"gasPrice":0,"chainId":1}' \
-H "Content-Type: application/json" \
-X POST -s | jq
{
  "data": "0x0000000000000000000000000000000000000000000000000000000000000001"
}
```

11.3 Submit Transaction

Send a SIGNED, NON-READONLY transaction. The client is left to compose a transaction, sign it and RLP encode it. The resulting bytes, represented as a Hex string, are passed to this method to be forwarded to the EVM. This is a SYNCHRONOUS operation; it waits for the transaction to go through consensus and returns the transaction receipt.

```
POST /rawtx
data: STRING Hex representation of the raw transaction bytes
returns: JSON JsonReceipt
```

```
type JsonTxRes struct {
    TxHash string `json:"txHash"`
}
```

```
type JsonReceipt struct {
    Root            common.Hash     `json:"root"`
    TransactionHash common.Hash     `json:"transactionHash"`
    From            common.Address  `json:"from"`
    To              *common.Address `json:"to"`
```

```
    GasUsed            uint64          `json:"gasUsed"`
    CumulativeGasUsed uint64          `json:"cumulativeGasUsed"`
    ContractAddress   common.Address  `json:"contractAddress"`
    Logs              []*ethTypes.Log `json:"logs"`
    LogsBloom         ethTypes.Bloom  `json:"logsBloom"`
    Status            uint64          `json:"status"`
}
```

Example:

```
host:~$ curl -X POST http://localhost:8080/rawtx -d
→'0xf8600180830f424094a10aae5609643848ff1bceb76172652261db1d6c648026a03c14b99e14420e34¢15885ff3afc1(
→' -s | json_pp
{
    "txHash":"0x96764078446cfbaec6265f173fb5a2411b7c272052640bca622252494a74dbb4"
}
```

11.4 Get Receipt

Get a transaction receipt. When a transaction is applied to the EVM, a receipt is saved to record if/how the transaction affected the state. This contains such information as the address of a newly created contract, how much gas was use, and the EVM Logs produced by the execution of the transaction.

```
GET /tx/{tx_hash}
returns: JSON JsonReceipt
```

Example:

```
host:~$ curl http://localhost:8080/tx/
→0x96764078446cfbaec6265f173fb5a2411b7c272052640bca622252494a74dbb4 -s | jq
{
  "root": "0x348c230578e27e20a10924e925f7cddb28279561b52cab7b31750c6d4716ac21",
  "transactionHash":
→"0x96764078446cfbaec6265f173fb5a2411b7c272052640bca622252494a74dbb4",
  "from": "0xa10aae5609643848ff1bceb76172652261db1d6c",
  "to": "0xa10aae5609643848ff1bceb76172652261db1d6c",
  "gasUsed": 21000,
  "cumulativeGasUsed": 21000,
  "contractAddress": "0x0000000000000000000000000000000000000000",
  "logs": [],
  "logsBloom":
→"0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
→",
  "status": 0
}
```

11.5 Info

Get information about a Babble instance.

```
GET /info
returns: JSON map
```

Example:

```
host:-$ curl http://localhost:8080/info | jq
{
    "rounds_per_second" : "0.00",
    "type" : "babble",
    "consensus_transactions" : "10",
    "num_peers" : "4",
    "consensus_events" : "10",
    "sync_rate" : "1.00",
    "transaction_pool" : "0",
    "state" : "Babbling",
    "events_per_second" : "0.00",
    "undetermined_events" : "22",
    "id" : "1785923847",
    "last_consensus_round" : "1",
    "last_block_index" : "0",
    "round_events" : "0"
}
```

11.6 POA

Get details of the PoA smart-contract.

```
GET /poa
returns: JsonContract
```

```
type JsonContract struct {
    Address common.Address `json:"address"`
    ABI     string         `json:"abi"`
}
```

Example (trunctated output):

```
host:-$ curl http://localhost:8080/poa | jq
{
    "address": "0xabbaabbaabbaabbaabbaabbaabbaabbaabbaabba",
    "abi": "[\n\t{\n\t\t\"constant\": true,\n\t\t\"inputs\"...]"
}
```

11.7 Genesis.json

This endpoint returns the content of the genesis.json file in JSON format. This allows new nodes to pull the genesis file from an existing peer.

```
GET /genesis
returns: JSON Genesis
```

```
type Genesis struct {
    Alloc AccountMap
    Poa   PoaMap
}
```

(continues on next page)

```go
type AccountMap map[string]struct {
    Code        string
    Storage     map[string]string
    Balance     string
    Authorising bool
}

type PoaMap struct {
    Address string
    Balance string
    Abi     string
    Code    string
}
```

Example (truncated output):

```
host:-$ curl://http://locahost:8080/genesis | jq
{
  "Alloc": {
    "a10aae5609643848ff1bceb76172652261db1d6c": {
      "Code": "",
      "Storage": null,
      "Balance": "12345678900000000000000",
      "Authorising": false
    }
  },
  "Poa": {
    "Address": "0xaBBAABbaaBbAABbaABbAABbAABbaAbbaaBbaaBBa",
    "Balance": "",
    "Abi": "[\n\t{\n\t\t\"constant\": ...]",
    "Code": "608060405260043610601095..."
  }
}
```

11.8 Block

Get a Babble Block by index.

```
GET /block/{index}
returns: JSON Block
```

```go
type Block struct {
    Body       BlockBody
    Signatures map[string]string
}

type BlockBody struct {
    Index                    int
    RoundReceived            int
    StateHash                []byte
    FrameHash                []byte
    PeersHash                []byte
    Transactions             [][]byte
```

```
    InternalTransactions        []InternalTransaction
    InternalTransactionReceipts []InternalTransactionReceipt
}
```

Example:

```
host:-$ curl http://locahost:8080/block/0 | jq
{
  "Body": {
    "Index": 0,
    "RoundReceived": 1,
    "StateHash": "VY6jFi7P5bIajdWvwZU2jU0q3KXDcp1sFx7Ye6kl1/k=",
    "FrameHash": "Nek4dF0ybGZQ1XEuJQrjmPtNrfPLAtGU4sTQSSB2iKM=",
    "PeersHash": "Gv+YqIq56l6LZWdhAsx0XEB4gjZluMaziv7hCXT5b9k=",
    "Transactions": [
      "+GSAgIMPQkCUq7qruqu6q7qruqu6q7qruqu6q7qAhOHHOSoloGCfTsLEOcMMXDX1W/
→78zpaZTXXK8BSR1Q8cCqicSrExoDv/0YGlpaGMJO8B6ZAJ/WAiEOKG00uzF8piaCvW3GHH"
    ],
    "InternalTransactions": [],
    "InternalTransactionReceipts": []
  },
  "Signatures": {

→"0X04F91D4429AE73229141F960B70CD2E83BF39D6EBF1B951C4E65BA9F0EE7FA2365C859CC9BF856709F78F0B9DD6BFBA4
→":
→"2gtf6rkdc0q29n1isef0x2fib64qlf075uybtva6558r8onv31|2gnym6xat1ok68nqtsymcpg4x9ihj1ouwab8inode5m8eb8
→"
  }
}
```

11.9 Current Peers

Get Babble's current peer-set.

```
Get /peers
returns: []Peer
```

```
type Peer struct {
    NetAddr   string
    PubKeyHex string
    Moniker   string
}
```

Example:

```
$host:-$ curl http://localhost:8080/peers | jq
[
  {
    "NetAddr": "192.168.1.3:1337",
    "PubKeyHex":
→"0X04F91D4429AE73229141F960B70CD2E83BF39D6EBF1B951C4E65BA9F0EE7FA2365C859CC9BF856709F78F0B9DD6BFBA4
→",
    "Moniker": "node0"
```

```
    }
]
```

11.10 Genesis Peers

Get Babble's initial validator-set.

```
GET /genesispeers
returns: []Peer
```

POA Smart Contract

This document describes the requirements for a smart-contract to implement POA in a Monet Toolchain hub. The default contract supplied with `monetd` already meets these requirements.

12.1 Solidity

12.1.1 Version

The first line of the contract is a pragma specifying the solidity version required. Currently this is set to greater than or equal to `0.4.22`.

```
pragma solidity >=0.4.22;
```

12.1.2 Constructor

The contract is embedded in the genesis block. This means that there is no conventional constructor. It is possible to add a hook to payable function calls to set an initial state if it has not already been initialised.

12.1.3 Modifier

`checkAuthorisedModifier` is used to restrict access to payable functions. The internals of that function could be ameneded to your new scheme.

12.1.4 CheckAuthorised

Babble calls the following function to verify whether a peer making a join request is authorised. Any replacement smart-contract will need to implement this function.

```
function checkAuthorised(address _address) public view returns (bool)
```

12.1.5 Payable calls

Functions that the client tools expect to be present.

```
function submitNominee (address _nomineeAddress, bytes32 _moniker) public payable␣
→checkAuthorisedModifier(msg.sender)
function castNomineeVote(address _nomineeAddress, bool _accepted) public payable␣
→checkAuthorisedModifier(msg.sender) returns (bool decided, bool voteresult)
```

12.1.6 Decision Function

This function decides when a vote is complete. Currently it requires all people on the whitelist to approve. It is anticipated that some form of majority voting would be implemented to prevent paralysis if a peer drops out.

```
function checkForNomineeVoteDecision(address _nomineeAddress) private returns (bool␣
→decided, bool voteresult)
```

12.1.7 Information Calls

The following information calls are available:

```
function getNomineeElection(address _address) public view returns (address nominee,␣
→address proposer, uint yesVotes, uint noVotes)
function getNomineeCount() public view returns (uint count)
function getNomineeAddressFromIdx(uint idx) public view returns (address␣
→NomineeAddress)
function getNomineeElectionFromIdx(uint idx) public view returns (address nominee,␣
→address proposer, uint yesVotes, uint noVotes)
function getCurrentNomineeVotes(address _address) public view returns (uint yes, uint␣
→no)
function getWhiteListCount() public view returns (uint count)
function getWhiteListAddressFromIdx(uint idx) public view returns (address␣
→WhiteListAddress)
function getYesVoteCount(address _nomineeAddress)  public view returns (uint count)
function getYesVoterFromIdx(address _nomineeAddress, uint _idx)  public view returns␣
→(address voter)
function getNoVoteCount(address _nomineeAddress)  public view returns (uint count)
function getNoVoterFromIdx(address _nomineeAddress, uint _idx) public view returns␣
→(address voter)
function getMoniker(address _address) public view returns (bytes32 moniker)
```

12.1.8 Events

The following events are emitted by the smart contract. It is envisaged that the same events would be emitted by any replacement contract.

```
/// @notice Event emitted when the vote was reached a decision
/// @param _nominee The address of the nominee
/// @param _yesVotes The total number of yes votes cast for the nominee to date
/// @param _noVotes The total number of no votes cast for the nominee to date
/// @param _accepted The decision, true for added to the whitelist, false for rejected
    event NomineeDecision(
        address indexed _nominee,
        uint  _yesVotes,
        uint _noVotes,
        bool indexed _accepted
    );
```

```
/// @notice Event emitted when a nominee vote is cast
/// @param _nominee The address of the nominee
/// @param _voter The address of the person who cast the vote
/// @param _yesVotes The total number of yes votes cast for the nominee to date
/// @param _noVotes The total number of no votes cast for the nominee to date
/// @param _accepted The vote, true for accept, false for rejected
    event NomineeVoteCast(
        address indexed _nominee,
        address indexed _voter,
        uint  _yesVotes,
        uint _noVotes,
        bool indexed _accepted
    );
```

```
/// @notice Event emitted when a nominee is proposed
/// @param _nominee The address of the nominee
/// @param _proposer The address of the person who proposed the nominee
    event NomineeProposed(
        address indexed _nominee,
        address indexed _proposer
    );
```

```
/// @notice Event emitted to announce a moniker
/// @param _address The address of the user
/// @param _moniker The moniker of the user
    event MonikerAnnounce(
        address indexed _address,
        bytes32 indexed _moniker
    );
```

12.2 Generated Genesis Whitelist Section

The template contract has a block of code delimited by the commments `//GENERATED GENESIS BEGIN` and `//GENERATED GENESIS END`. In the `monetd config build` command that block of code is replaced with generated code. That code is customised to include the initial authorised peers list in the genesis block.

A sample generated block is included below.

```
//GENERATED GENESIS BEGIN

    address constant initWhitelist0 = 0xDc3062F7E88C456c2aD6EeaAc2D6Da4034F6CD7C;
    bytes32 constant initWhitelistMoniker0 = "node0";
```

```
    address constant initWhitelist1 = 0xdB77c5DBb8c39a82F131252853000E8691a772E1;
    bytes32 constant initWhitelistMoniker1 = "node1";
    address constant initWhitelist2 = 0xe9fa241921dF673E932B173C1a41bc532Db4C330;
    bytes32 constant initWhitelistMoniker2 = "node2";


    function processGenesisWhitelist() private
    {
     addToWhitelist(initWhitelist0, initWhitelistMoniker0);
     addToWhitelist(initWhitelist1, initWhitelistMoniker1);
     addToWhitelist(initWhitelist2, initWhitelistMoniker2);
    }


    function isGenesisWhitelisted(address _address) pure private returns (bool)
    {
        return (  ( initWhitelist0 == _address ) || ( initWhitelist1 == _address ) ||␣
→( initWhitelist2 == _address ) );
    }

 //GENERATED GENESIS END
```

The following functions must be defined in the generated block as they are referenced in the non-generated code.

```
function processGenesisWhitelist() private
function isGenesisWhitelisted(address _address) pure private returns (bool)
```

Giverny Examples

For reference, the options for `giverny network new`:

```
[..monetd] $ giverny help network new

giverny network build

Usage:
  giverny network new [network_name] [flags]

Flags:
      --generate-pass        generate pass phrases
  -h, --help                 help for new
      --initial-ip string    initial IP address of range
      --initial-peers int    number of initial peers
      --names string         filename of a file containing a list of node monikers
      --no-build             disables the automatic build of a new network
      --no-save-pass         don't save pass phrase entered on command line
  -n, --nodes int            number of nodes in this configuration (default -1)
      --pass string          filename of a file containing a passphrase

Global Flags:
  -g, --giverny-data-dir string   Top-level giverny directory for configuration and␣
→data (default "/home/jon/.giverny")
  -m, --monet-data-dir string     Top-level monetd directory for configuration and␣
→data (default "/home/jon/.monet")
  -v, --verbose                   verbose messages
```

13.1 Development Test Networks

To make commands repeatable, and to reflect code changes, the following commands can be prefixed to all the commands below:

```
make installgiv; rm -rf ~/.giverny/networks/test9;
```

The command above rebuilds the `giverny` app and removes the network `test9``allow the ``new` commands to be run repeatedly. If you do not remove the previous network `test9` before running `giverny network new` then the command aborts. The `make installgiv` is only required if you are making code changes.

Adding `-v` or `--verbose` to each of these commands gives addition information and progress messages in the command output.

13.1.1 New

8 node network, 4 initial peers, named from prebaked list of names, generated passphrases.

```
giverny network new test9 --generate-pass --names sampledata/names.txt --nodes 8 --
→initial-peers 4 -v
```

3 node network with named nodes, 2 initial peers. Passphrased prompted for on the command line and used for all key files.

```
make installgiv; rm -rf ~/.giverny/networks/test9; giverny network new test9 --save-
→pass --names sampledata/withnodes.txt --nodes 3 --initial-peers 2 -v
```

The withnodes.txt file is interesting as it shows the expanded syntax:

```
Jon,192.168.1.18,1T,true
Martin,192.168.1.3,1G,true
Kevin,192.168.1.16,1M,false
```

13.1.2 Export Network

The export command writes the configuration of one or more nodes to a zip file.

To export the configuration of all nodes in a network, type this:

```
$ giverny network export test9
```

Take a look in `~/.giverny/exports`.[1] There should be numerous files named `test9_[node].zip`. These can be applied to `monetd` directly on the same instance by:

```
$ giverny network import test9 Danu --from-exports
```

Alternatively you can use a secondary channel such as slack or e-mail to send that zip file and then load it — without changing the name of the file:

```
$ giverny network import test9 Danu --dir ~/Downloads
```

Or you can use giverny server and pull it directly. Assuming that you have run `giverny server start` on the instance you ran the exports you can:

```
$ giverny network import test9 Danu --server 192.168.1.4
```

---

[1] This location is for Linux instances. Mac and Windows uses a different path. The path for your instance can be ascertain with this command: `giverny network location`

Monetd Configuration

All the configuration required to run a node is stored under a directory with a very specific structure. By default, `monetd` will look for this directory in `$HOME/.monet`[1] (on Linux), but it is possible to override this with the `--datadir` flag.

The directory must respect the following stucture:

```
host:~/.monet$ tree
├── babble
│       ├── peers.genesis.json
│       ├── peers.json
│       └── priv_key
├── eth
│       ├── genesis.json
│       └── poa
│               ├── compile.toml
│               ├── contract0.abi
│               └── contract0.sol
├── keystore
│       ├── node0.json
├── monetd.toml
```

You would not normally need to access these configuration files directly. The `monetd config` tool provides a CLI interfaces to set up a network. The command `monetd config location --expanded` provides further details of the filepaths used for your instance.

14.1 Eth

The `eth/genesis.json` file defines prefunded accounts in the state, as well as the POA smart-contract. This file is useful to predefine a set of accounts that own all the initial tokens at the inception of the network. In addition, the `poa` section contains information about the POA smart-contract.

---

[1] This location is for Linux instances. Mac and Windows uses a different path. The path for your instance can be ascertain with this command: `monetd config location`

Example `genesis.json` defining one prefunded account (the ABI and bytecode of the smart-contract have been truncated):

```
{
    "alloc": {
            "a10aae5609643848ff1bceb76172652261db1d6c": {
                    "balance": "1234567890000000000000",
                    "moniker": "node0"
            }
    },
    "poa": {
            "address": "0xaBBAABbaaBbAABbaABbAABbAABbaAbbaaBbaaBBa",
            "abi": "[\n\t{\n\t\t\"constant\": true, ... ]",
            "code": "6080604052600436106101095760003560e01c8063..."
            }
}
```

14.2 Babble

- **babble/genesis.peers.json**: defines Babble's initial peer-set.

- **babble/peers.json**: defines Babble's current peer-set

- **babble/priv_key**: contains the validator's private key for Babble.

14.3 Run Options

Options pertaining to the operation of the node are read from the [datadir]/monetd.toml file, or overwritten by the following flags. It is envisaged that you would not need to use these flags in a production environment.

```
Flags:
    --api-listen string        IP:PORT of HTTP API service (default ":8080")
    --babble.bootstrap         bootstrap Babble from database
    --babble.cache-size int    number of items in LRU caches (default 50000)
    --babble.heartbeat duration   heartbeat timer milliseconds (time between gossips)␣
→(default 200ms)
    --babble.listen string     IP:PORT of Babble node (default "192.168.1.3:1337")
    --babble.max-pool int      max number of pool connections (default 2)
    --babble.sync-limit int    max number of Events per sync (default 1000)
    --babble.timeout duration  TCP timeout milliseconds (default 1s)
    --eth.cache int            megabytes of memory allocated to internal caching␣
→(min 16MB / database forced) (default 128)
    --eth.min-gas-price string   minimum gasprice of transactions submitted through␣
→this node (ex 1K, 1M, 1G, etc.) (default "0")
 -h, --help                    help for run

Global Flags:
 -d, --datadir string   top-level directory for configuration and data (default "/
→home/martin/.monet")
 -v, --verbose          verbose output
```

Example of a monet.toml file:

```
datadir = "/home/user/.monet"
verbose = "false"
api-listen = ":8080"

[babble]
  listen = "192.168.1.3:1337"
  heartbeat = "500ms"
  timeout = "1s"
  cache-size = 50000
  sync-limit = 1000
  max-pool = 2
  bootstrap = false

[eth]
  cache = 128
```

Monetd Reference

`monetd` provides the core commands needed to configure and run a node. It has context sensitive help accessed either by running `monetd help` or by adding a `-h` parameter to the relevant command.

```
[..monetd] $ monetd help
MONET-Daemon

Monetd provides the core commands needed to configure and run a Monet
node. The minimal quickstart configuration is:

        $ monetd config clear
        $ monetd keys new node0
        $ monetd config build node0
        $ monetd run

See the documentation at https://monetd.readthedocs.io/ for further information.

Usage:
  monetd [command]

Available Commands:
  config      manage monetd configuration
  help        Help about any command
  keys        monet key manager
  run         run a MONET node
  version     show version info

Flags:
  -d, --datadir string   top-level directory for configuration and data (default "/
→home/jon/.monet")
  -h, --help             help for monetd
  -v, --verbose          verbose output

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

There are 5 subcommands. `help` is described above. The other 4 commands are described in separate sections below:

- **help** — show help for the command and subcommands
- **version** — shows the current version of monetd and subsystems
- **keys** — creates and manages keys
- **config** — creates and manages configurations
- **run** — runs the monet daemon, i.e. starts a node

15.1 Global Parameters

Global Parameters are available for all subcommands.

- **-d, --datadir string** — overrides the default location of the configuration files
- **-h, --help** — help command as discussed above
- **-v, --verbose** — turns on verbose messages. Defaults to false.

15.2 Version

The `version` subcommand outputs the version number for `monetd`, `EVM-Lite`, `Babble` and `Geth`.

If you compile your own tools, the suffices are the GIT branch and the GIT commit hash.

```
[..monetd] $ monetd version
Monet Version: 0.2.1-develop-397c075f
     EVM-Lite Version: 0.2.1-develop
     Babble Version: 0.5.1-develop
     Geth Version: 1.8.27
```

15.3 Keys

The `keys` subcommand is used to manage Monet Toolchain keys. There are 4 subcommands, each described in a seperate section below:

- **inspect** — inspect a keyfile
- **list** — list keyfiles
- **new** — create a new keyfile
- **update** — change the passphrase on a keyfile

The `keys` subcommand writes and reads keys from the `keystore` sub-folder in the `monetd` configuration folder. You can see the location for your instance with this command:

```
$ monetd config location -x
```

The help for the keys command is:

```
[..monetd] $ monetd keys help

Manage keys in the [datadir]/keystore folder.
```

(continues on next page)

```
Note that other Monet tools, like monetcli and monet-wallet, use the same
default [datadir]/keystore.


+-----------------------------------------------------------------------+
| Please take all the necessary precautions to secure these files and remember |
| the passwords, as it will be impossible to recover the keys without them.    |
+-----------------------------------------------------------------------+

Keys are associated with monikers and encrypted in password-protected files in
[datadir]/keystore/[moniker].json. Keyfiles contain JSON encoded objects, which
Ethereum users will recognise as the de-facto Ethereum keyfile format. Indeed,
Monet and the underlying consensus algorithm, Babble, use the same type of keys
as Ethereum. A key can be used to run a validator node, or to control an account
with a token balance.

Usage:
  monetd keys [command]

Available Commands:
  inspect     inspect a keyfile
  list        list keyfiles
  new         create a new keyfile
  update      change the passphrase on a keyfile

Flags:
  -h, --help              help for keys
      --json              output JSON instead of human-readable format
      --passfile string   file containing the passphrase

Global Flags:
  -d, --datadir string   top-level directory for configuration and data (default "/
→home/jon/.monet")
  -v, --verbose          verbose output

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

15.3.1 Parameters

All of the keys subcommands support the --passfile flag. This allows you to pass the path to a plain text file containing the passphrase for your key. This removes the interactive prompt to enter the passphrase that is the default mechanism.

15.3.2 Monikers

Keys generated by monetd have a moniker associated with them. The moniker is used to manage the keys as it is far more user friendly that an Ethereum address or public key.

15.3.3 New

The new subcommand generates a new key pair and associates it with the specified moniker. You will be prompted for a passphrase which is used to encrypt the keyfile. It writes the encrypted keyfile to the monetd keystore area

by default. The moniker must be unique within your keystore. If you attempt to create a duplicate, the command will abort with an error.

```
[..monetd] $ monetd help keys new

Generate a new cryptographic key-pair identified by [moniker].

If the --passfile flag is not specified, the user will be prompted to enter the
passphrase manually.

Usage:
  monetd keys new [moniker] [flags]

Flags:
  -h, --help   help for new

Global Flags:
  -d, --datadir string    top-level directory for configuration and data (default "/
→home/jon/.monet")
      --json              output JSON instead of human-readable format
      --passfile string   file containing the passphrase
  -v, --verbose           verbose output
```

15.3.4 Inspect

```
[..monetd] $ monetd help keys display

Manage keys in the [datadir]/keystore folder.

Note that other Monet tools, like monetcli and monet-wallet, use the same
default [datadir]/keystore.

+--------------------------------------------------------------------------+
| Please take all the necessary precautions to secure these files and remember |
| the passwords, as it will be impossible to recover the keys without them.   |
+--------------------------------------------------------------------------+

Keys are associated with monikers and encrypted in password-protected files in
[datadir]/keystore/[moniker].json. Keyfiles contain JSON encoded objects, which
Ethereum users will recognise as the de-facto Ethereum keyfile format. Indeed,
Monet and the underlying consensus algorithm, Babble, use the same type of keys
as Ethereum. A key can be used to run a validator node, or to control an account
with a token balance.

Usage:
  monetd keys [command]

Available Commands:
  inspect     inspect a keyfile
  list        list keyfiles
  new         create a new keyfile
  update      change the passphrase on a keyfile

Flags:
  -h, --help               help for keys
      --json               output JSON instead of human-readable format
```

```
      --passfile string   file containing the passphrase


Global Flags:
  -d, --datadir string   top-level directory for configuration and data (default "/
→home/jon/.monet")
  -v, --verbose          verbose output


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

A sample session showing the command usage with and without the `--private` parameter.

```
$ monetd keys inspect node0 --private
Passphrase:
Address:        0x02f6f3D24E447218d396C14F3B47f9Ea369DADf9
Public key:    ␣
→0481d3528eec6138f8428932e4fe99571a4f77bd79ae13219540b0a929014cb490a4e5ced2f9e651b531522c2567b6dc5de
Private key:    bc553aaa7e55c5d0f58f6897ba9bffdb88233c420da622d363f2fe4bd6d78df1
```

```
$ monetd keys inspect node0
Passphrase:
Address:        0x02f6f3D24E447218d396C14F3B47f9Ea369DADf9
Public key:    ␣
→0481d3528eec6138f8428932e4fe99571a4f77bd79ae13219540b0a929014cb490a4e5ced2f9e651b531522c2567b6dc5de
```

15.3.5 Update

The `update` subcommand allows you to change the passphrase for an encrypted key file. You are prompted for the old passphrase, then you need to enter, and confirm, the new passphrase.

You can suppress the prompts by specifying the `--passfile` parameter to supply the current passphrase and `--new-passphrase` to supply the new passphrase.

```
[..monetd] $ monetd help keys update
change the passphrase on a keyfile


Usage:
  monetd keys update [moniker] [flags]


Flags:
  -h, --help                 help for update
      --new-passfile string  the file containing the new passphrase


Global Flags:
  -d, --datadir string   top-level directory for configuration and data (default "/
→home/jon/.monet")
      --json             output JSON instead of human-readable format
      --passfile string  file containing the passphrase
  -v, --verbose          verbose output
```

An example session updating the passphrase for a key:

```
$ monetd keys update node0
Passphrase:
Please provide a new passphrase
```

```
Passphrase:
Repeat passphrase:
```

15.3.6 List

The `list` subcommand outputs a list of monikers corresponding to the keyfiles in the keystore. These are the valid monikers that can be specified to other `monetd` commands.

```
[..monetd] $ monetd help keys list
list keyfiles

Usage:
  monetd keys list [flags]

Flags:
  -h, --help   help for list

Global Flags:
  -d, --datadir string    top-level directory for configuration and data (default "/
→home/jon/.monet")
      --json              output JSON instead of human-readable format
      --passfile string   file containing the passphrase
  -v, --verbose           verbose output
```

An example session:

```
$ monetd keys list
node0
node1
node2
```

15.4 Config

The `config` subcommand initialises the configuration for a `monetd` node. The folder can be overridden by the `--datadir` parameter. The configuration commands create all the files necessary for a node to join an existing network or to create a new one.

There are 5 subcommands each described in a separate section below:

- **clear** — backup and clear configuration folder
- **contract** — display poa contract
- **location** — show the location of the configuration files
- **build** — create the configuration for a single-node network
- **pull** — pull the configuration files from a node

The two most common scenarios are:

- **config build - config build creates the configuration for a single-node** network, based on one of the keys in [datadir]/keystore. This is a quick and easy way to get started with `monetd`. See *Getting Started*.
- **config pull - config pull is used to join an existing network. It fetches the** configuration from one of the existing nodes. See *Joining a Network*.

For more complex scenarios, please refer to *Giverny Reference*, which is a specialised Monet Toolchain configuration tool.

15.4.1 Clear

The `clear` subcommand safely clears any previous `monetd` configurations. It renames the previous configuration with a `.~n~` suffix, where n is the lowest integer where the resultant filename does not already exist.

The configurations are renamed and not deleted to avoid the potential for inadvertent deletion of keys.

```
$ monetd config clear
Renaming /home/user/.monet to /home/user/.monet.~1~
```

15.4.2 Contract

The `contract` subcommand generates the Solidity source for a POA smart contract with the supplied node as the sole entry on the initial whitelist. This command is not used in the standard workflow, but is provided as a convenient mechanism to retrieve the solidity source.

```
[..monetd] $ monetd help config contract

monetd config contract

Outputs the standard monetd contract, configured with [moniker] as the initial
whitelist.

Usage:
  monetd config contract [moniker] [flags]

Flags:
  -h, --help   help for contract

Global Flags:
  -d, --datadir string   top-level directory for configuration and data (default "/
→home/jon/.monet")
  -v, --verbose          verbose output
```

A sample session is as follows. The contract is written to stdout, so you will probably wish to redirect it to a file or a pager.

```
$ monetd config contract node0 | more
pragma solidity >=0.4.22;

/// @title Proof of Authority Whitelist Contract

...
```

15.4.3 Location

The `location` subcommand displays the path to the configuration folder. With the `--expanded` parameter, a list of directories and configuration files are output.

```
[..monetd] $ monetd help config location

The location subcommand shows the location of the monetd configuration files. It
respects any --datadir parameter.

If you specify --expanded then a list of configuration folders and directories
is output.

Usage:
  monetd config location [flags]

Flags:
  -x, --expanded   show expanded information
  -h, --help       help for location

Global Flags:
  -d, --datadir string   top-level directory for configuration and data (default "/
→home/jon/.monet")
  -v, --verbose          verbose output
```

```
$ monetd config location
/home/user/.monet
```

```
$ monetd config location --expanded
Config root   : /home/user/.monet
Babble Dir    : /home/user/.monet/babble
EVM-Lite Dir  : /home/user/.monet/eth
Keystore Dir  : /home/user/.monet/keystore
Config File   : /home/user/.monet/monet.toml
Wallet Config : /home/user/.monet/wallet.toml
Peers         : /home/user/.monet/babble/peers.json
Genesis Peers : /home/user/.monet/babble/peers.genesis.json
Genesis File  : /home/user/.monet/eth/genesis.json
```

15.4.4 Build

The `build` subcommand initialises the bare-bones configuration to start `monetd`. It uses one of the accounts from the keystore to define a network consisting of a unique node, which is automatically added to the PoA whitelist. Additionally, all the accounts in [datadir]/keystore are credited with a large amount of tokens in the genesis file. This command is mostly used for testing.

If the `--address` flag is omitted, the first non-loopback address for this instance is used.

```
[..monetd] $ monetd help config build

The build subcommand initialises the bare-bones configuration to get started
with monetd. It uses one of the accounts from the keystore to define a network
consisting of a unique node, which is automatically added to the PoA whitelist.
Additionally, all the accounts in [datadir]/keystore are credited with a large
amount of tokens in the genesis file. This command is mostly used for testing.

If the --address flag is omitted, the first non-loopback address for this
instance is used.

Usage:
```

```
  monetd config build [moniker] [flags]

Flags:
      --address string    IP/hostname of this node (default "172.17.0.1")
  -h, --help              help for build
      --passfile string   file containing the passphrase

Global Flags:
  -d, --datadir string    top-level directory for configuration and data (default "/
→home/jon/.monet")
  -v, --verbose           verbose output
```

15.4.5 Pull

The `pull` subcommand is used to join an existing network. It takes the address of a running peer, and downloads the following set of files into the configuration directory [datadir]:

- babble/peers.json : The current validator-set

- babble/peers.genesis.json : The initial validator-set

- eth/genesis.json : The genesis file

It also builds all the other configuration files required to run a monetd node. If the peer specified does not include a port, the default gossip port (1337) is used.

```
[..monetd] $ monetd help config pull

The pull subcommand is used to join an existing Monet network. It takes the
address (host:port) of a running node, and downloads the following set of files
into the configuration directory [datadir]:

- babble/peers.json         : The current validator-set
- babble/peers.genesis.json : The initial validator-set
- eth/genesis.json          : The genesis file

Additionally, this command configures the validator-key and the network address
of the new node. The --key and --passfile options refer to the validator-key,
while --address sets the network address of monetd.

Usage:
  monetd config pull [host:port] [flags]

Examples:
  monetd config pull "192.168.5.1:8080"

Flags:
      --address string    IP/hostname of this node (default "172.17.0.1")
  -h, --help              help for pull
      --key string        moniker of the key to use for this node (default "Jon")
      --passfile string   file containing the passphrase

Global Flags:
  -d, --datadir string    top-level directory for configuration and data (default "/
→home/jon/.monet")
  -v, --verbose           verbose output
```

15.5 Run

The `run` subcommands starts the monetd node running. Whilst there are legacy parameters `--babble.*` and `--eth.*`, we strongly recommend that they are not used. The equivalent changes can be made in the configuration files.

```
[..monetd] $ monetd help run

Run a MONET node.

Start a daemon which acts as a full node on a MONET network. All data and
configuration are stored under a directory [datadir] controlled by the
--datadir flag ($HOME/.monet by default on UNIX systems).

[datadir] must contain a set of files defining the network that this node is
attempting to join or create. Please refer to monetd config for tools to manage
this configuration.

Further options pertaining to the operation of the node are read from the
[datadir]/monetd.toml file, or overwritten by the following flags. The following
command displays the expected output:

monetd config location

Usage:
  monetd run [flags]

Flags:
      --api-listen string         IP:PORT of Monet HTTP API service (default ":8080
↪")
      --babble.bootstrap          Bootstrap Babble from database
      --babble.cache-size int     Number of items in LRU caches (default 50000)
      --babble.heartbeat duration Heartbeat time milliseconds (time between␣
↪gossips) (default 200ms)
      --babble.listen string      IP:PORT of Babble node (default "172.17.0.1:1337")
      --babble.max-pool int       Max number of pool connections (default 2)
      --babble.sync-limit int     Max number of Events per sync (default 1000)
      --babble.timeout duration   TCP timeout milliseconds (default 1s)
      --eth.cache int             Megabytes of memory allocated to internal caching␣
↪(min 16MB / database forced) (default 128)
  -h, --help                      help for run

Global Flags:
  -d, --datadir string   top-level directory for configuration and data (default "/
↪home/jon/.monet")
  -v, --verbose          verbose output
```

Giverny Reference

`giverny` is the advanced configuration tool for the Monet Toolchain.

The current subcommands are:

- **help** — help
- **version** — outputs version information
- **keys** — key management tools
- **server** — configuration server management
- **network** — configure and build networks
- **transactions** — generate test transactions sets

16.1 Global Flag

The `--verbose` flag, or `-v` for short, turns on extended messages for each `giverny` command.

16.2 Help

`giverny` has context sensitive help accessed either by running `giverny help` or by adding a `-h` parameter to the relevant command.

16.3 Version

The `version` subcommand outputs the version number for `monetd`, `EVM-Lite`, `Babble` and `Geth`.

If you compile your own tools, the suffices are the GIT branch and the GIT commit hash.

```
[..monetd] $ giverny version
Monet Version: 0.2.1-develop-397c075f
     EVM-Lite Version: 0.2.1-develop
     Babble Version: 0.5.1-develop
     Geth Version: 1.8.27
```

16.4 Keys

The `keys` subcommand offers tools to manage keys.

16.4.1 Keys Flags

In addition to the `--verbose` flag, the `keys` subcommand defines addtional flags as follows:

```
Global Flags:
  -g, --giverny-data-dir string   Top-level giverny directory for configuration and
→data (default "/home/jon/.giverny")
  -m, --monet-data-dir string     Top-level monetd directory for configuration and
→data (default "/home/jon/.monet")
  -v, --verbose                   verbose messages

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

16.4.2 Import

The `import` subcommand is used to import a pre-existing private key into the `monetd` keystore, creating the asso-
ciated `toml` file, assigning a moniker and setting a passphrase.

```
[..monetd] $ giverny help keys import

Import keys to [moniker] from private key file [keyfile].

Usage:
  giverny keys import [moniker] [keyfile] [flags]

Flags:
  -h, --help   help for import

Global Flags:
  -g, --giverny-data-dir string   Top-level giverny directory for configuration and
→data (default "/home/jon/.giverny")
      --json                      output JSON instead of human-readable format
  -m, --monet-data-dir string     Top-level monetd directory for configuration and
→data (default "/home/jon/.monet")
      --passfile string           the file that contains the passphrase for the
→keyfile
  -v, --verbose                   verbose messages
```

16.5 Server

The `server` subcommand is used for adminstering a REST server used to co-ordinate configurations between multiple nodes prior to the initial node of a network.

The server listens on port 8088. It writes logs to `~/.giverny/server/server.pid`.[1]

For usage examples, see the recipes for setting up networks.

16.5.1 Start

To start the server in the foreground:

```
$ giverny server start
```

To start the server in the background:

```
$ giverny server start --background
```

16.5.2 Stop

To stop a server running in the background:

```
$ giverny server stop
```

16.5.3 Status

Reports on the status of the server. It both checks for the PID file in `~/.giverny/server/server.pid`[1] and checks the the server is responding on localhost:8088.

```
$ giverny server status
```

16.6 Network

The `network` command is used to build complex monet networks. The `new` command generates the nodes and keys for a network, and automatically calls the `build` command which generates and builds `genesis.json` and `peers.json` files. You can adjust the network by editting the `network.toml` file. The `location` command outputs the relevant paths. The `push` command is used to push a giverny network node configuration to a docker or actual node so it can be used by `monetd`. `start`, `stop` and `status` are used to manage the docker instance.

The *network name* and *node names* must contain only standard letters (i.e. no accented versions), digits (0–9) or underscores (_).

---

[1] This location is for Linux instances. Mac and Windows uses a different path. The path for your instance can be ascertain with this command: `giverny network location`

16.6.1 Location

The `giverny network location` subcommand takes a single optional parameter `network_name`. If the network is specified it outputs the location of key files and folders for that network. If not, only the root giverny configuration folder is output.

Example without a network name:

```
$ giverny network location
/home/user/.giverny
```

Example with a network specified:

```
$ giverny network location node7
Network             : node7
Giverny Config Dir   : /home/user/.giverny
Giverny Networks Dir : /home/user/.giverny/networks/node7
Giverny KeyStore Dir : /home/user/.giverny/networks/node7/keystore
Peers JSON           : /home/user/.giverny/networks/node7/peers.json
Genesis JSON         : /home/user/.giverny/networks/node7/genesis.json
Monetd TOML          : /home/user/.giverny/networks/node7/monetd.toml
Network TOML         : /home/user/.giverny/networks/node7/network.toml
```

16.6.2 New

The `new` subcommand creates a new test network configuration. It also

Syntax

```
[..monetd] $ giverny help network new

giverny network build

Usage:
  giverny network new [network_name] [flags]

Flags:
      --generate-pass       generate pass phrases
  -h, --help                help for new
      --initial-ip string   initial IP address of range
      --initial-peers int   number of initial peers
      --names string        filename of a file containing a list of node monikers
      --no-build            disables the automatic build of a new network
      --no-save-pass        don't save pass phrase entered on command line
  -n, --nodes int           number of nodes in this configuration (default -1)
      --pass string         filename of a file containing a passphrase

Global Flags:
  -g, --giverny-data-dir string   Top-level giverny directory for configuration and␣
→data (default "/home/jon/.giverny")
  -m, --monet-data-dir string     Top-level monetd directory for configuration and␣
→data (default "/home/jon/.monet")
  -v, --verbose                   verbose messages
```

Nodes

The number of nodes in this network is specified by the `--nodes [int]` parameter. The `--initial-peers [int]` parameter specifies the number of initial peers. If not set it assumes that all nodes are in the initial peer set.

IP Addresses

An initial IP address is supplied using the `--initial-ip` parameter. It is assumed the IP address range will be assigned by simply incrementing the last octet of the IP address for each node. N.B. the first node will be assigned the actual IP supplied by the `initial-ip` parameter.

Node Names

The default node names are a standard prefix of *node* with a unique integer suffix. You can override the default and supply a list of node names, which are used in the order supplied, via the `--names` parameter.

Node names must contain only standard Latin alphabet characters (ie *a–z* or *A–Z* with no accents), underscores (_), or digits (*0–9*).

Pass Phrases

There are numerous pass phrase flags for the `new` subcommand.

- `--pass [passfile]` — uses the given pass phrase file for all nodes
- `--generate-pass` — generates a unique passphrase for each key pair and writes it to a file nodename.txt in the network configuration keystore directory
- `--save-pass` — saves pass phrases in the network configuration keystore directory

The typical use case scenarios for these flags would be:

- None specified — you are prompted to enter the passphrase for each node which is not saved
- `--pass` only — the specified pass phrase is used, but not saved in the config folder
- `--pass` and `--save-pass` — the specified pass phrase is used **and** saved in the config folder
- `--generate-pass` only — pass phrases are generated and saved
- `--save-pass` only — you are prompted to enter the passphrase for each node, which is saved in the config folder

Build

By default `giverny network new` will run `giverny network build` automatically. This can be disabled by specifying the `-no-build` flag.

Examples

An example of the new subcommand:

```
$ giverny network new test11 --names sampledata/names.txt --nodes 7 --pass sampledata/
↪pwd.txt --initial-peers 3 --initial-ip 192.168.1.19
```

16.6.3 Build

The `giverny network build` subcommand takes a configuration created by the `new` subcommand and builds `peers.json` and `genesis.json` files.

`build` can be run repeatably safely. It is envisaged that users will edit the `network.toml` file to adjust token allocations or change addresses.

`--no-generate-keys` disables the creation of any keys not already in the keystore.

A "built" network will have a file structure like this:

```
test7
├── compile.toml
├── contract0.abi
├── contract0.sol
├── genesis.json
├── keystore
│   ├── Amelia.json
│   ├── Amelia.txt
│   ├── Becky.json
│   ├── Becky.txt
│   ├── Chloe.json
│   ├── Chloe.txt
│   ├── Danu.json
│   ├── Danu.txt
├── monetd.toml
├── network.toml
└── peers.json
```

16.6.4 Export

The `export` subcommand takes a configuration that has been generated and exports it to the exports subfolder of the giverny configuration folders as a zip file. The `network export` command has a mandatory network name parameter, and optionally one or more node names. If the node names are omitted, all of the nodes for that network are exported.

16.6.5 Import

The `import` subcommand takes a configuration previously exported by the `export` and configures `monetd` to use the new configuration. You will always need to specify a network name and a node name for the import. The source for the import can be configured thus:

- `--from-exports` — from the exports subfolder in the giverny configuration folders. This is the default output location for the `export` command.

- `--server` — from a giverny server. The giverny server will look in the exports subfolder in the giverny configuration folders on the instance it is running on. N.B. do not run the giverny server on any instance with live key pairs or sensitive configuration, as it may be exposed.

- `--dir` — specify the folder the export zip is in. Do not rename the zip file. This is used when a secondary channel is used to communicate the keys.

16.7 Transactions

The transaction commands has one subcommand: `generate`. It is used to generate transactions set for end to end testing of networks.

The following flags can be set:

```
    --count int       number of tranactions to generate (default 20)
    --faucet string   faucet account moniker (default "Faucet")
-h, --help            help for generate
    --ips string      ips.dat file path
-n, --network string  network name
    --surplus int     additional credit to allocate each account from the faucet␣
↪above the bare minimum (default 1000000)
```

Licences

The Monet Toolchain is an open source project, licensed under the MIT License (TLDR version). The software is provided as-is and we are not liable. We use many other libraries to build the Toolchain. This section presents the output of Glice for the Monet Toolchain. Glice reports on the licences used within a golang project.

The 3 tables are for the monetd, EVM-Lite and Babble repositories respectively. These tables are the output from `glice -r`, which only looks one level deep. These tables are for information only and are not legal advice.

monetd:

DEPENDENCY	REPOURL	LICENSE
github.com/AndreasBriese/bbloom	https://github.com/AndreasBriese/bbloom	Other
github.com/btcsuite/btcd	https://github.com/btcsuite/btcd https://github.com/dgraph-io/badger https://github.com/dgryski/go-farm https://github.com/docker/docker https://github.com/ethereum/go-ethereum https://github.com/fatih/color https://github.com/fsnotify/fsnotify https://github.com/golang/protobuf https://github.com/gorilla/mux https://github.com/hashicorp/hcl https://github.com/magiconair/properties https://github.com/mattn/go-colorable https://github.com/mattn/go-isatty https://github.com/mgutz/ansi https://github.com/mitchellh/mapstructure https://github.com/mosaicnetworks/babble https://github.com/mosaicnetworks/evm-lite https://github.com/pelletier/go-toml https://github.com/pkg/errors https://github.com/sirupsen/logrus https://github.com/spf13/afero https://github.com/spf13/cast https://github.com/spf13/cobra https://github.com/spf13/jwalterweatherman https://github.com/spf13/pflag https://github.com/spf13/viper https://github.com/ugorji/go https://github.com/x-cray/logrus-prefixed-formatter https://go.googlesource.com/crypto https://go.googlesource.com/net https://go.googlesource.com/net https://go.googlesource.com/sys https://go.googlesource.com/text https://go.googlesource.com/text	isc
github.com/dgraph-io/badger		Apache-2.0
github.com/dgryski/go-farm		Other
github.com/docker/docker		Apache-2.0
github.com/ethereum/go-ethereum		LGPL-3.0
github.com/fatih/color		MIT
github.com/fsnotify/fsnotify		bsd-3-clause
github.com/golang/protobuf		bsd-3-clause
github.com/gorilla/mux		bsd-3-clause
github.com/hashicorp/hcl		MPL-2.0
github.com/magiconair/properties		Other
github.com/mattn/go-colorable		MIT
github.com/mattn/go-isatty		MIT
github.com/mgutz/ansi		MIT
github.com/mitchellh/mapstructure		MIT
github.com/mosaicnetworks/babble		MIT
github.com/mosaicnetworks/evm-lite		MIT
github.com/pelletier/go-toml		MIT
github.com/pkg/errors		bsd-2-clause
github.com/sirupsen/logrus		MIT
github.com/spf13/afero		Apache-2.0 MIT
github.com/spf13/cast		Apache-2.0 MIT
github.com/spf13/cobra		bsd-3-clause
github.com/spf13/jwalterweatherman		MIT
github.com/spf13/pflag		MIT
github.com/spf13/viper		MIT
github.com/ugorji/go github.com/x-cray/logrus-prefixed-formatter		
golang.org/x/crypto/ssh/terminal		
golang.org/x/net/internal/timeseries		
golang.org/x/net/trace		
golang.org/x/sys/unix		
golang.org/x/text/transform		
golang.org/x/text/unicode/norm		
gopkg.in/yaml.v2		

EVM-Lite:

DEPENDENCY	REPOURL	LICENSE
github.com/sirupsen/logrus	https://github.com/sirupsen/logrus https://go.googlesource.com/crypto https://go.googlesource.com/sys https://github.com/ethereum/go-ethereum	MIT
golang.org/x/crypto/ssh/terminal		LGPL-3.0
golang.org/x/sys/unix		
github.com/ethereum/go-ethereum		

Babble:

DEPENDENCY	REPOURL	LICENSE
github.com/AndreasBriese/bbloom	https://github.com/AndreasBriese/ bbloom https://github.com/btcsuite/btcd	Other
github.com/btcsuite/btcd		isc
github.com/btcsuite/fastsha256	https://github.com/btcsuite/fastsha256	Other
github.com/dgraph-io/badger	https://github.com/dgraph-io/badger	Apache-2.0
github.com/dgryski/go-farm	https://github.com/dgryski/go-farm	
github.com/fsnotify/fsnotify	https://github.com/fsnotify/fsnotify	Other
github.com/golang/protobuf	https://github.com/golang/protobuf	bsd-3-clause
github.com/hashicorp/hcl	https://github.com/hashicorp/hcl https: //github.com/magiconair/properties	bsd-3-clause
github.com/magiconair/properties		
github.com/mitchellh/mapstructure	https://github.com/mitchellh/mapstructure	clause
github.com/pelletier/go-toml github.com/pkg/errors	https://github.com/pelletier/go-toml https://github.com/pkg/errors https: //github.com/sirupsen/logrus https://github. com/spf13/afero https://github.com/spf13/cast https://github.com/spf13/cobra https: //github.com/spf13/jwalterweatherman https://github.com/spf13/pflag https://github. com/spf13/viper https://github.com/ugorji/go https://go.googlesource.com/crypto https://go.googlesource.com/net https://go. googlesource.com/net https://go.googlesource. com/sys https://go.googlesource.com/text https://go.googlesource.com/text https://github.com/rifflock/lfshook	MPL-2.0
github.com/sirupsen/logrus		Other
github.com/spf13/afero		MIT
github.com/spf13/cast		MIT
github.com/spf13/cobra		bsd-2-clause
github.com/spf13/jwalterweatherman		MIT
github.com/spf13/pflag		Apache-2.0 MIT
github.com/spf13/viper		Apache-2.0 MIT
github.com/ugorji/go		bsd-3-clause
golang.org/x/crypto/ssh/terminal		MIT
golang.org/x/net/internal/timeseries		MIT
golang.org/x/net/trace		MIT
golang.org/x/sys/unix		
golang.org/x/text/transform		
golang.org/x/text/unicode/norm		
gopkg.in/yaml.v2		
github.com/rifflock/lfshook		

FAQ

18.1 General

- *What is the difference between MONET, MONET Hub, Monet Toolchain and monetd?*
- *Why Giverny?*

18.1.1 What is the difference between MONET, MONET Hub, Monet Toolchain and `monetd`?

`monetd` is the daemon that runs a node on a blockchain. `monetd` is also a repository on Github. The repository also hosts `giverny` which implements advanced configuration options.

The Monet Toolchain consists of the `monetd` repository, plus additional software, noticeably monetcli. Together these repositories provide a complete suite of tools for running a blockchain. Whilst the Monet Toolchain was initially developed for the MONET Hub, it has been designed to be easily used by other projects.

The MONET Hub is a specific blockchain running on the Monet Toolchain software.

MONET is the whole ecosystem containing the MONET Hub, and mobile adhoc blockchains that use the MONET Hub to persist state.

18.1.2 Why Giverny?

Whilst the name MONET is derived from Mosaic Networks, Giverny was famously the home of Claude Monet and inspiration for many of his most renowned works.

The Monet Toolchain

The Monet Toolchain provides software to run and interact with a distributed smart-contract platform based on EVM-Lite and Babble consensus.

It underpins the MONET Hub, which is an important part of the MONET project, but is licensed under the MIT license and available for use in other projects. You can read more about MONET in the whitepaper.

19.1 Quick Start

For the impatient, we recommend you start here:

- *Installation Documents*
- *Quick Start*