IOV Weave Documentation

Release 0.14.0

Ethan Frey

May 25, 2020

Contents

IOV Weave is a framework to quickly build your custom ABCI application to power a blockchain based on the best-of-class BFT Proof-of-stake Tendermint consensus engine. It provides much commonly used functionality that can quickly be imported in your custom chain, as well as a simple framework for adding the custom functionality unique to your project.

Some of the highlights of Weave include a Merkle-tree backed data store, a highly configurable extension system that also applies to the core logic such as fees and signature validation. Weave also brings powerful customizations initialised from the genesis file. In addition there is a simple ORM which sits on top of a key-value store that also has proveable secondary indexes. There is a flexible permissioning system to use contracts as first-class actors, "No empty blocks" for quick synchronizing on quiet chains, and the ability to introduce "product fees" for transactions that need to charge more than the basic anti-spam fees. We have also added support for "migrations" that can switch on modules, or enable logic updates, via on-chain feature switch transactions.

CHAPTER 1

Existing Modules

Module	Description
Cash	Wallets that support fungible tokens and fee deduction functionality
Sigs	Validate ed25519 signatures
Multisig	Supports first-class multiple signature contracts, and allow modification of membership
AtomicSwap	Supports HTLC for cross-chain atomic swaps, according to the IOV Atomic Swap Spec
Escrow	The arbiter can safely hold tokens, or use with timeouts to release on vesting schedule
Governance	Hold on-chain elections for text proposals, or directly modify application parameters
PaymentChannels	Unidirectional payment channels, combine micro-payments with one on-chain settlement
Distribution	Allows the safe distribution of income among multiple participants using configurations. This can be used to distribute fee income.
Batch	Used for combining multiple transactions into one atomic operation. A powerful example is in creating single-chain swaps.
Validators_	Used in a PoA context to update the validator set using either multisig or the on-chain elections module
NFT	A generic Non Fungible Token module
NFT/Username	Example nft used by bnsd. Maps usernames to multiple chain addresses, including reverse lookups
MessageFee	Validator-subjective minimum fee module, designed as an anti-spam measure.
Utils	A range of utility functions such as KeyTagger which is designed to enable subscriptions to database.

In Progress

Light client proofs, custom token issuance and support for IBC (Inter Blockchain Communication) are currently being designed.

CHAPTER 2

Basic Blockchain Terminology

2.1 Blockchain

A “blockchain” in the simplest sense is a chain of blocks. By chain, we each block is cryptographically linked to the proceeding block, and through recursion we can securely query the entire history from any block back to the genesis. A block is a set of transactions, along with this link, and some optional metadata that varies depending on the blockchain.

2.1.1 Immutable Event Log

If you are coming from working on typical databases, you can think of the blockchain as an immutable transaction log . If you have worked with Event Sourcing you can consider a block as a set of events that can always be replayed to create a materialized view . Maybe you have a more theoretical background and recognize that a blockchain is a fault tolerant form of state machine replication .

In any case, the point is that given a node knows a block is valid (more on that in consensus), it can cryptographically prove it has the valid history of that block, and then replay that sequence of blocks to reproduce the current state. Many nodes performing this simultaneously create a Byzantine Fault Tolerant state machine. Since (most) computer programs can be mapped to state machines, we end up with an unstoppable world computer .

This means we can trust that the blockchain and state represent the proper functioning of whatever program we run on it. This allows for extremely high levels of trust in a program, levels that were previously reserved for highly controlled, centralized systems, such as banks or governments. The first generation of blockchain, Bitcoin, proved it was possible to run a system with many unknown and mutually untrusting parties, yet produce a system that is harder to hack than any bank (bitcoin hacks involve grabbing someone’s wallet, not manipulating the blockchain). This was a true marvel of vision and engineering and laid the stage for all future development, and many other projects tried to fork bitcoin to create a custom blockchain.

2.1.2 General Purpose Computer

Ethereum pioneered the second generation of blockchain, where they realized that we didn’t have to limit ourselves to handling payments, but actually have a general purpose state machine. They wanted to allow experimentation at a rate orders of magnitude faster than forking bitcoin, and produced the EVM (Ethereum Virtual Machine) that can run sandboxed code uploaded by any user. Since then, hundred of projects have experimented with porting other types of logic to the blockchain, and have demonstrated its utility for decentralized governance, currency trading, prediction markets, even collectible trading games and much more. . .

While Ethereum demonstrated the potential of blockchain technology in many areas, we it also provided some high profile examples of how hard it is to write secure contracts . As it became more popular, it also showed a popular application can overload the capacity of the network .

2.1.3 Next Generation

Since that time, many groups are working on "next generation" solutions that take the learnings of Ethereum and attempt to build a highly scalable and secure blockchain that can run general purpose programs. One pioneering project is Tendermint, which provides a highly efficient, Byzantine Fault Tolerant blockchain engine offering guaranteed finality in the order of 1-5 seconds. It was designed from the ground up to allow many projects to easily plug their application logic into the engine. Weave is a framework that provides many common tools to help you build ABCI apps rapidly. You can just focus on writing the application logic and the interface and rely on high quality and extensible libraries to solve most of the difficult problems with building a blockchain.

2.2 Consensus

Consensus is the algorithm by which a set of computers come to agreement on which possible state is correct, and thus guarantee one consistent, global view of the state of the system.

2.2.1 Eventual finality

All PoW systems use eventual finality, where the resource cost of creating a block is extremely high. After many blocks are gossiped, the longest chain of blocks has the most work invested in it, and thus is the true chain. The "true" head of the chain can switch, in a process called "chain reorganization". But the probability of such a reorganization decreases exponentially the more blocks are built on top of it. Thus, in Bitcoin, the "6 block rule" means that if there are 6 blocks build on top of the block with your transaction, you can be extremely confident that no chain reorganization will ever generate a new true chain that does not include that block. Note this is not a guarantee that it cannot happen, just that the cost of doing so becomes so prohibitively high that is very unlikely to ever happen.

Many early PoS systems, such as BitShares, used voting instead of work to mine blocks, but still used the "longest chain wins" consensus algorithm. However, this has the critical nothing at stake problem, since the cost of "mining" blocks on 2, 3, or even 100 alternate chains is quite low.

Another issue here is that any state may have to be reverted, and the data store must maintain an "undo history" to undo several blocks and apply others. And clients must wait several blocks (minutes to hours) before they can take off-chain actions based on the transaction (eg. give you goods for a blockchain payment).

2.2.2 Immediate finality

An alternative approach used to guarantee constency comes out of academic research into Byzantine Fault Tolerance from the 80s and 90s, which "culminated" in PBFT . Tendermint uses an algorithm very similar to PBFT with optimizations learned from blockchain developments to create an extremely secure consensus algorithm. All nodes vote in multiple rounds, and only produce blocks when they are guaranteed that the block is the "correct" globally consensus. Even in the case of omnipotent network manipulation, this algorithm will never produce to blocks at the same height (a fork) if less than one third of the nodes are actively collaborating to break the system. This is possibly the strongest guarantee of any production blockchain consensus algorithm.

The benefit of this approach is that any block that has over two thirds of the signature is provably correct by light clients The state is never rolled back and clients can take actions based on that state. This opens the possibility of blockchain payments to be settled in the order of a second or two, similar latency with using a credit card in a store. It also allows reasonably responsive applications to be built on a blockchain.

2.3 Authentication

One interesting attribute of blockchains is that there are no trusted nodes, and all transactions are publicly visible and can be copied. This naturally provides problem for traditional means of authentication like passwords and cookies. If you use your password to authorize one transaction, someone can copy it to run any other. Or a node in the middle can even change your transaction before writing to a block.

Thus, all authentication on blockchains is based on public key cryptography, in particularly cryptographic signatures based on elliptic curves. A client can locally generate a public-private key pair, and share the public key with the world as his/her identity (like a fingerprint). The client can then take any message (text or binary) and generate a unique signature with the private key. The signature can only be validated by the corresponding message and public key and cannot be forged. Any changes to the message will invalidate the signature and no information is leaked to allow a malicious actor to impersonate that client with a different message.

2.3.1 Main Algorithms

- RSA - the gold standard from 1977-2014, still secure and the most widely supported. not used for blockchains as signatures are 1-4KB
- secp256k1 - elliptic curve used in bitcoin and ethereum, signatures at 65-67 bytes
- ed25519 - popularized with libsodium and most standardized elliptic curve, signatures at 64 bytes
- bn256 - maybe the next curve… used by zcash for pairing cryptography and dfinity for BLS threshold signatures. in other words, they can do crazy magic math on this particular curve.

If you want to go deeper than what you can find on wikipedia and google, I highly recommend buying a copy of `Serious Cryptography` by Jean-Philippe Aumasson.

2.4 State Machine

Inside each block is a sequence of transactions to be applied to a state machine (ran by a program). There is also the state (database) representing the materialized view of all transactions included in all blocks up to this point, as executed by the state machine.

2.4.1 Upgrading the state machine

Of course, during the lifetime of the blockchain, we will want to update the software and expand functionality. However, the new software must also be able to re-run all transactions since genesis (the birth of the chain) and produce the same state as the active network that keeps updating software over time. That means all historical blocks must be interpreted the same by new and old clients.

Bitcoin classifies various approaches to upgrading as soft forks or hard forks. Ethereum has a table defining the block height at which various functionality changes and add checks for the currently activated behavior based on block height at various places. This allows one server to handle multiple historical behaviors, but it can also add lots of dead code over time…

2.4.2 UTXO vs Account Model

There are two main models used to store the current state. The main model for bitcoin and similar chains is called UTXO, or *Unspent transaction output*. Every transaction has an input and an output, and the system must just check if the inputs have been used yet. If they have not, they are marked spent and the outputs created. If any have been spent, then the transaction fails.

This provides interesting ways to obfuscate identity (but not secure against sophisticated network analysis like ZCash), and allows easy parallelization of the transaction processing. However, it is quite hard to map non-payment systems (like voting or breeding crypto-kitties) to such a system. It is used mainly for focused payment networks.

The account model creates one account per public key address and stores the information there. Sending money becomes reducing the balance on one account and incrementing on another. And many other more complex logic become easy to express, using logic that many developers are used to from interacting with databases or key-value stores.

The downside is that the account allows an observer to easy view all activity by one key. Sure you have pseudoanonymity, but if you make one payment to me, I now can see your entire investment and voting history with little effort. Another downside is that it become harder to parallelize transaction processing, as sending from one account and receiving payments will conflict with each other. In practice, no production chains use optimistic concurrency on account based systems.

2.4.3 Merkle Proofs

Weave uses an account model much like Ethereum, and leaves anonymity to other developments like mixnets and zkSNARKs.

Under the hood, we use a key-value store, where different modules write their data to different key-spaces. This is not a normal key-value store (like redis or leveldb), but rather merkle trees. Merkle trees are like binary trees, but hash the children at each level. This allows us to provide a proof as a chain of hashes the same height as the tree. This proof can guarantee that a given key-value pair is in the tree with a given root hash. This root hash is then added to a block header after running the transactions, and validated by consensus. If a client follows the headers, they can securely verify if a node if providing them the correct data for eg. their account balance.

In practice, the block header can maintain multiple hashes, each one the *merkle root* of another tree. Thus, a client can use a header to prove, state, presence of a transaction, or current validator set.

If you are new to blockchains (or Tendermint), this is a crash course in just enough theory to follow the rest of the setup. Read all

2.5 Immutable Event Log

If you are coming from working on typical databases, you can think of the blockchain as an immutable transaction log . If you have worked with Event Sourcing you can consider a block as a set of events that can always be replayed to create a materialized view . Maybe you have a more theoretical background and recognize that a blockchain is a fault tolerant form of state machine replication . Read more

2.6 General Purpose Computer

Ethereum pioneered the second generation of blockchain, where they realized that we didn't have to limit ourselves to handling payments, but actually have a general purpose state machine. Read more

2.7 Next Generation

Since that time, many groups are working on "next generation" solutions that take the learnings of Ethereum and attempt to build a highly scalable and secure blockchain that can run general purpose programs. Read more

2.8 Eventual finality

All Proof-of-Work systems use eventual finality, where the resource cost of creating a block is extremely high. After many blocks are gossiped, the longest chain of blocks has the most work invested in it, and thus is the true chain. Read more

2.9 Immediate finality

An alternative approach used to guarantee constency comes out of academic research into Byzantine Fault Tolerance from the 80s and 90s, which "culminated" in PBFT . Read more

2.10 Authentication

One interesting attribute of blockchains is that there are no trusted nodes, and all transactions are publically visible and can be copied. Read more

2.11 Upgrading the state machine

Of course, during the lifetime of the blockchain, we will want to update the software and expand functionality. However, the new software must also be able to re-run all transactions since genesis. Read more

2.12 UTXO vs Account Model

There are two main models used to store the current state. The main model for bitcoin and similar chains is called UTXO, or Unspent transaction output. The account model creates one account per public key address and stores the information there. Read more

2.13 Merkle Proofs

Merkle trees are like binary trees, but hash the children at each level. This allows us to provide a proof as a chain of hashes. Read more

CHAPTER 3

Running an Existing Application

3.1 Prepare Requirements

Before you can run this code, you need to have a number of programs set up on your machine. In particular, you will need a bash shell (or similar), and development tooling for both go and node.

WARNING

This is only tested under Linux and OSX. If you want to run under Windows, the only supported *development* environment is using WSL (Windows Subsytem for Linux) under Windows 10. Follow these directions to setup Ubuntu in WSL, then try the rest in your Ubuntu shell

3.1.1 Install Go

You will need to have the Go tooling installed, version 1.11.4+ (or 1.12). If you do not already have it, please download and follow the instructions from the official Go language homepage. Make sure to read down to Test Your Installation. (Note this is not included in Ubuntu apt tooling until 19.04)

We assume a standard setup in the Makefiles, especially to build tendermint nicely. With `go mod` much of the go configuration is unnecessary, but make sure to have the default "install" directory in your `PATH`, so you can run the binaries after compilation.

```
# this line should be in .bashrc or similar
export PATH="$PATH:$HOME/go/bin"
# this must report 1.11.4+
go version
# this will properly place the code in $HOME/go/src/github.com/iov-one/weave
go get github.com/iov-one/weave
```

Go related tools

You must also make sure to have a few other developer tools installed. If you are a developer in any language, they are probably there. Just double check. If not, a simple `sudo apt get` should provide them.

- git
- make
- curl
- jq

3.2 Installation

To run our system, we need two components:

- `mycoind`, our custom ABCI application
- `tendermint`, a powerful blockchain consensus engine

If you have never used tendermint before, you should read the ABCI Overview and ideally through to the bottom of the page. The end result is that we have three programs communicating:

```
+---------+                     +------------+                      +----------+
| mycoind |  <- (local) ABCI -> | Tendermint |   <- websocket ->    | client   |
+---------+                     +------------+                      +----------+
```

`mycoind` and `tendermint` run on the same computer and communicate via a binary protocol over localhost or a unix socket. Together they form a "blockchain". In a real setup, you would have dozens (or hundreds) of computers running this backend communicating over a self-adjusting p2p gossip network to replicate the state. For application development (and demos) one copy will work, but has none of the fault tolerance of a real blockchain.

You can connect to tendermint rpc via various client libraries. We recommend IOV Core which has very good support for weave-based apps, as well as different blockchains (such as Ethereum and Lisk).

3.2.1 Install backend programs

You should have a proper go development environment, as explained in the last section. Now, check out the most recent version of iov-one/weave and build `mycoind` then get the version 0.31.5 for `tendermint` from here. You can also build `tendermint` from source following the instructions there but make sure to use the tag **v0.31.5** as other versions might not be compatible.

Note we use `go mod` for dependency management. This is enabled by default in go 1.12+. If you are running go 1.11.4+, you must run the following in the terminal (or add to `~/.bashrc`): `export GO111MODULE=on`

Those were the most recent versions as of the time of the writing, your code should be a similar version. If you have an old version of the code, you may have to delete it to force go to rebuild:

```
rm `which tendermint`
rm `which mycoind`
```

3.2.2 Initialize the Blockchain

Before we start the blockchain, we need to set up the initial state. This is defined in a genesis block. Both `tendermint` and `mycoind` have a directory to store configuration and internal database state. By default those are `~/.tendermint` and `~/.mycoind`. However, to make things simpler, we will ask them both to put everything in the same directory.

First, we create a default genesis file, the private key for the validator to sign blocks, and a default config file.

```
# make sure you really don't care what was in this directory and...
rm -rf ~/.mycoind
tendermint init --home ~/.mycoind
```

You can take a look in this directory if you are curious. The most important piece for us is `~/.mycoind/config/genesis.json`. You may also notice `~/.mycoind/config/config.toml` with lots of options to set for power users.

We want to add a bunch of tokens to the account we just made before launching the blockchain. And we'd also like to enable the indexer, so we can search for our transactions by id (default state is off). But rather than have you fiddle with the config files by hand, you can just run this to do the setup:

```
mycoind init CASH bech32:tiov1qrw95py2x7fzjw25euuqlj6dq6t0jahe7rh8wp
```

Make sure you enter the same hex address, this account gets the tokens. You can take another look at `~/.mycoind/config/genesis.json` after running this command. The important change was to "app_state". You can also create this by hand later to give many people starting balances, but let's keep it simple for now and get something working. Feel free to wipe out the directory later and reinitialize another blockchain with custom configuration to experiment.

You may ask where this address comes from. It is a demo account derived from our test mnemonic: `dad kiss slogan offer outer bomb usual dream awkward jeans enlist mansion` using the hd derivation path: `m/44'/234'/0'`. This is the path used by our wallet, so you can enter your mnemonic in our web-wallet and see this account. Note that you can define the addresses both in *hex:* and *bech32:* formats (if prefix is ommitted, hex is assumed)

3.2.3 Start the Blockchain

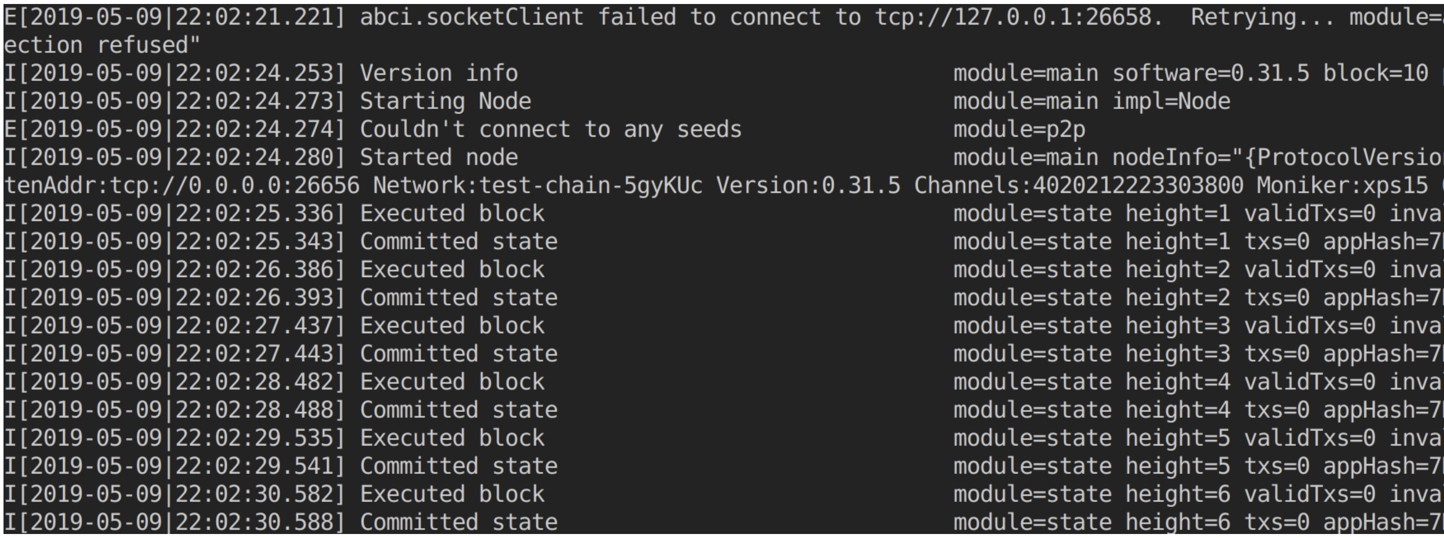
We have a private key and setup all the configuration. The only thing left is to start this blockchain running.

```
tendermint node --home ~/.mycoind > ~/.mycoind/tendermint.log &
mycoind start
```

This connects over tcp://localhost:26658 by default, to use unix sockets (arguably more secure), try the following:

```
tendermint node --home ~/.mycoind --proxy_app=unix://$HOME/abci.socket > ~/.mycoind/
↪tendermint.log &
mycoind start -bind=unix://$HOME/abci.socket
```

Open a new window and type in `tail -f ~/.mycoind/tendermint.log` and you will be able to see the output. That means the blockchain is working away and producing new blocks, one a second.



```
E[2019-05-09|22:02:21.221] abci.socketClient failed to connect to tcp://127.0.0.1:26658.  Retrying... module=
ection refused"
I[2019-05-09|22:02:24.253] Version info                                 module=main software=0.31.5 block=10
I[2019-05-09|22:02:24.273] Starting Node                                module=main impl=Node
E[2019-05-09|22:02:24.274] Couldn't connect to any seeds                module=p2p
I[2019-05-09|22:02:24.280] Started node                                 module=main nodeInfo="{ProtocolVersio
tenAddr:tcp://0.0.0.0:26656 Network:test-chain-5gyKUc Version:0.31.5 Channels:4020212223303800 Moniker:xps15
I[2019-05-09|22:02:25.336] Executed block                               module=state height=1 validTxs=0 inva
I[2019-05-09|22:02:25.343] Committed state                              module=state height=1 txs=0 appHash=7
I[2019-05-09|22:02:26.386] Executed block                               module=state height=2 validTxs=0 inva
I[2019-05-09|22:02:26.393] Committed state                              module=state height=2 txs=0 appHash=7
I[2019-05-09|22:02:27.437] Executed block                               module=state height=3 validTxs=0 inva
I[2019-05-09|22:02:27.443] Committed state                              module=state height=3 txs=0 appHash=7
I[2019-05-09|22:02:28.482] Executed block                               module=state height=4 validTxs=0 inva
I[2019-05-09|22:02:28.488] Committed state                              module=state height=4 txs=0 appHash=7
I[2019-05-09|22:02:29.535] Executed block                               module=state height=5 validTxs=0 inva
I[2019-05-09|22:02:29.541] Committed state                              module=state height=5 txs=0 appHash=7
I[2019-05-09|22:02:30.582] Executed block                               module=state height=6 validTxs=0 inva
I[2019-05-09|22:02:30.588] Committed state                              module=state height=6 txs=0 appHash=7
```

Note: if you did anything funky during setup and managed to get yourself a rogue tendermint node running in the background, you might encounter errors like *panic: Error initializing DB: resource temporarily unavailable*. A quick `killall tendermint` should get you back on track.

3.3 Using IOV-Core Client

While the blockchain code is in the Go language, we have developed a TypeScript (javascript-compatible) client side sdk in order to access the functionality of the blockchain. Iov-Core works for many blockchains, not just weave (mycoind and bnsd), so take a look, it is useful for more than this demo

3.3.1 Installing Tooling

You will need node 8+ to run the example client. Unless you know what you are doing, stick to even numbered versions (6, 8, 10, . . .), the odd numbers are unstable and get deprecated every few weeks it seems. For ease of updating later, I advise you to install nvm and then add the most recent stable version

```
# this install most recent v8 version, use lts/dubnium for v10 track
nvm install lts/carbon

# test it out
node --version
node
> let {x, y} = {x: 10, y:10}
```

Node related tools

Yarn is a faster alternative to npm for installing modules, so we use that as default.

```
npm install -g yarn
```

3.3.2 Using Iov-Core

Please refer to the offical iov-core documentation Note that you can use the `BnsConnection` to connect to a `mycoind` blockchain, as long as you restrict it to just sending tokens and querying balances and nonces (it is a subset of `bnsd`). You may also find iov-cli a useful debug tool. It is an enhanced version of the standard node REPL (interactive coding shell), but with support for top-level `await` and type-checks on all function calls (you can code in typescript).

The iov-core library supports the concept of user profiles and identities. An identity is a BIP39 derived key. Please refer to those docs and tutorials for a deeper dive, it is out of the scope of the weave documentation.

A good way to get familiar with setting up and running an application is to follow the steps in the mycoin sample application. You can run this on your local machine. If you don't have a modern Go development environment already set up, please follow these instructions.

To connect a node to the BNS testnet on a cloud server, the steps to set up an instance on Digital Ocean are explored in this blog post.

Once you can run the blockchain, you will probably want to connect with it. You can view a sample wallet app for the BNS testnet at https://wallet.hugnet.iov.one Those that are comfortable with Javascript, should check out our IOV Core Library which allows easy access to the blockchain from a browser or node environment.

CHAPTER 4

Configuring your Blockchain

4.1 Configuring Tendermint

Tendermint docs provide a brief introduction to the tendermint cli. By default all files are writen to the `~/.tendermint` directory, unless you override that with a different "HOME" directory by providing `TMHOME=xyz` or `tendermint --home=xyz`.

When you call `tendermint init`, it generates a `config` and `data` directory under the "HOME" dir. `data` will contain all blockchain state as well as the application state. `config` will contain configuration files. There are three main files to look at:

- `genesis.json` must be shared by all validators on a chain and is used to initialize the first block. We discuss this more in *Application Config*
- `config.toml` is used to configure your local server, and can be configured much in the way the config for apache or postgres, to tune to your local system.
- `priv_validator.json` is used by any validating node to sign the blocks, and must be kept secret. We discuss this more in the next section.

4.1.1 Overriding Options

In general, any option you see in the configuration file can also be provided via command-line or environmental variable. It is a simple conversion:

Config:

```
[rpc]
laddr = "tcp://0.0.0.0:8080"
```

Environment: `export TM_RPC_LADDR=tcp://0.0.0.0:8080` or `export TMRPC_LADDR=tcp://0.0.0.0:8080` (optional _ after TM)

Command line: `tendermint --rpc.laddr=tcp://0.0.0.0:8080 ...`

4.1.2 Important Options

There are many options to tune tendermint, but a few are quite useful when configuring and deploying dev environements or testnets. I will cover them here, but please take a longer look at all available options. I use the command line format for these options, as it seems the most readable, but most of these should be writen to the `config.toml` file or stored in environmental options in the service ini (if using 12-factor style).

Dev: - `--p2p.upnp --proxy_app noop`: Don't try to determine external address

(*noop* for local testing)

- `--log_level=p2p:info,evidence:debug,consensus:info,*:error`: Set the log levels for different subsystems (debug, info, error)
- `--tx_index.index_all_tags=true` to enable indexing for search and subscriptions. Should be on for public services, off for validators to conserve resources.
- `--prof_laddr=tcp://127.0.0.1:7777` to open up a profiling server at this port for debugging

Testnet: - `--moniker=billy-bob` chooses a name to display on the node list

to help understand the p2p network connections

- `--mempool.recheck=false` and `--mempool.recheck_empty=false` limit rechecking all leftover tx in mempool, which can help throughput at the expense of possibly invalid tx making it into blocks
- `--rpc.laddr=tcp://0.0.0.0:46657` to change the interface or port we expose the rpc server (what we expose to the world)
- `--p2p.laddr=tcp://0.0.0.0:46656` to change the interface or port we expose the p2p server (what we use to connect to other nodes)
- `--p2p.seeds=tcp://12.34.56.78:46656,tcp://33.44.55.66:46656` to set the seed nodes we connect to on startup to discover the rest of the p2p network
- `p2p.pex=true` turns on peer exchange, to allow us to dynamically update the network
- `--consensus.create_empty_blocks=false` to only create a block when there are tx (otherwise blockchain grows fast even with no activity)
- `--consensus.create_empty_blocks_interval=300` to create a block every 300s even if no tx
- `--consensus.timeout_commit=5000` to set block interval to 5s (5000ms) + time it takes to achieve consensus (which is generally quite small with < 20 or so well-connected validators)

Production: - `p2p.persistent_peers=tcp://77.77.77.77:46656` contains peers we

always remain connected to, regardless of peer exchange

- `p2p.private_peer_ids=...` contains peers we do not gossip. this is essential if we have a non-validating node acting as a buffer for a validating node
- `--priv_validator_laddr=???` to use a socket to connect to an hsm instead of using the priv_validator.json file

There are quite a few more options, but this is a good place to get started, and you can dig in deeper once you see how these numbers affect blockchains in practice.

4.2 Configuring the Application

The application is fed `genesis.json` the first time it starts up via the `InitChain` ABCI message. There are three fields that the application cares about:

- `chain_id` must be consistent on all nodes and distinct from all other blockchains. This is used in the tx signatures to provide replay protection from one chain and another
- `validators` are the initial set and should be stored if the app wishes to dynamically adjust the validator set
- `app_state` contains a map of data, to set up the initial blockchain state, such as initial balances and any accounts with special permissions.

4.2.1 App State

If the backend ABCI app is weave-based, such as `mycoind` or `bns`, the app_state contains one key for each extension that it wishes to initialize. Each element is an array of an extension-specific format, which is fed into `Initialized.FromGenesis` from the given extension.

Sample to set the balances of a few accounts:

```
"app_state": {
  "cash": [
    {
      "address": "849f0f5d8796f30fa95e8057f0ca596049112977",
      "coins": [ "88888888 BNS" ]
    },
    {
      "address": "9729455c431911c8da3f5745a251a6a399ccd6ed",
      "coins": [ "7777777.666666 IOV" ]
    }
  ]
}
```

This format is application-specific and extremely important to set the initial conditions of a blockchain, as the data is one of the largest distinguishing factors of a chain and a fork.

`mycoind init` will set up one account with a lot of tokens of one name. For anything more complex, you will want to set this up by hand. Note that you should make sure someone has saved the private keys for all addresses or the tokens will never be usable. Also, for cash, ticker must be 3 or 4 upper-case letters.

4.3 Setting the Validators

Since Tendermint uses a traditional BFT algorithm to reach consensus on blocks, signatures from specified validator keys replace hashes used to mine blocks in typical PoW chains. This also means that the selection of validators is an extremely important part of the blockchain security, and every validator should have strong security in place to avoid their private keys being copied or stolen.

4.3.1 Static Validators

In the simplest setup, every node can generate a private key with `tendermint init`. Note that this is stored as a clear-text file on the harddrive, so the machine should be well locked-down, and file permissions double-checked. This file not only contains the private key itself, but also information on the last block proposal signed, to avoid double-signing blocks, even in the even of a restart during one round.

Every validator can find their validator public key, which is different than the public keys / addresses that are assigned tokens, via:

```
cat ~/.mycoind/config/priv_validator.json | jq .pub_key
```

If you still have the default genesis file from *tendermint init*, this public key should match the one validator registered for this blockchain, so it can mint blocks all by itself.

```
cat ~/.mycoind/config/genesis.json | jq .validators
```

In a multi-node network, all validators would have to generate their validator key separately, then share the public keys, and forge a genesis file will all the public keys present. Over two-thirds of these nodes must be online, connected to the p2p network, and acting correctly to mint new blocks. Up to one-third faulty nodes can be tolerated without any problems, and larger numbers of nodes usually halt the network, rather than fork it of mint incorrect blocks.

The Tendermint dev team has produced a simple utility to help gather these keys.

Note that this liveness requirement means that after initializing the genesis and starting up tendermint on every node, they must set proper `--p2p.seeds` in order to connect all the nodes and get enough signatures gathered to mint the first block.

4.3.2 HSMs

If we really care about security, clearly a plaintext file on our machine is not the best solution, regardless of the firewall we put on it. For this reason, the tendermint team is working on integrating Hardware Security Modules (HSM) that will maintain the private key secret on specialized hardware, much like people use the Ledger Nano hardware wallet for cryptocurrencies.

This is under active development, but please check the following repos to see the current state:

- Signatory provides a rust api exposing many curves to sign with
- YubiHSM provides bindings to a YubiKey HSM
- KMS is a work in progress to connect these crates via sockets to a tendermint node.

TODO Update with current docs, now that cosmos mainnet is live and some people are actually using this.

4.3.3 Dynamic Validators

A static validator set in the genesis file is quite useless for a real network that is not just a testnet. Tendermint allows the ABCI application to send back messages to update the validator set at the end of every block. Weave-based applications can take advantage of this and implement any algorithm they want to select the validators, such as:

- PoA where a set of keys (held by clients) can appoint the validators. This allows them to bring up and down machines, but the authority of the chain rests in a fixed group of individuals.
- `PoS` or proof-of-stake, where any individual can bond some of their tokens to an escrow for the right to select a validator. Each validator has a voting power proportional to how much is staked. These staked tokens also receive some share of the block rewards as compensation for the work and risk.
- `DPoS` where users can either bond tokens to their own validator, or "delegate" their tokens to a validator run by someone else. Everyone gets some share of the block rewards, but the people running the validator nodes typically take a commission on the delegated rewards, as they must perform real work.

For each of these general approaches there is a wide range of tuning of incentives and punishments in order to achieve the desired level of usability and security.

The only current implementation shipping with weave is a POA implementation allowing some master key (can be a multisig or even an election) update the validator set. This can support systems from testnets to those with strong on-chain governance, but doesn't work for the PoS fluid market-based solution.

If you wish to build an extension supporting PoS, previous related work from cosmos-sdk can be found in their simple stake implementation and the more complicated DPoS implementation with incentive mechanisms.

When you ran the `mycoind` tutorial, you ran the following lines to configure the blockchain:

```
tendermint init --home ~/.mycoind
mycoind init CASH bech32:tiov1qrw95py2x7fzjw25euuqlj6dq6t0jahe7rh8wp
```

This is nice for automatic initialization for dev mode, but for deploying a real network, we need to look under the hood and figure out how to configure it manually.

4.4 Tendermint Configuration

Tendermint docs provide a brief introduction to the tendermint cli. Here we highlight some of the more important options and explain the interplay between cli flags, environmental variables, and config files, which all provide a way to customize the behavior of the tendermint daemon. Read More

4.5 Application State Configuration

The application is fed `genesis.json` the first time it starts up via the `InitChain` ABCI message. There are three fields that the application cares about: `chain_id`, `app_state`, and `validators`. To learn more about these fields Read More

4.6 Setting the Validators

Since Tendermint uses a traditional BFT algorithm to reach consensus on blocks, signatures from specified validator keys replace hashes used to mine blocks in typical PoW chains. This also means that the selection of validators is an extremely important part of the blockchain security. Read More

CHAPTER 5

Building your own Application

5.1 Guiding Design Principles

Before we get into the structure of the application, there are a few design principles for weave (but also tendermint apps in general) that we must keep in mind. If you are coming from developing web servers or microservices, some of these are counter-intuitive. (Eg. you cannot make external API calls and concurrency is limited)

5.1.1 Determinism

The big key to blockchain development is determinism. Two binaries with the same state must **ALWAYS** produce the same result when passed a given transaction. This seems obvious, but this also occurs when the transactions are replayed weeks, months, or years by a new version, attempting to replay the blockchain.

- You cannot relay on walltime (just the timestamp in the header)
- No usage of floating point math
- No random numbers!
- No network calls (especially external APIs)!
- No concurrency (unless you **really** know what you are doing)
- JSON encoding in the storage is questionable, as the key order may change with newer JSON libraries.
- Etc. . . .

The summary is that everything is executed sequentially and deterministically, and thus we require extremely fast transaction processing to enable high throughput. Aim for 1-2 ms per transaction, including committing to disk at the end of the block. Thus, attention to performance is very important.

5.1.2 ABCI

To understand this design, you should first understand what an ABCI application is and how that level blockchain abstraction works. ABCI is the interface between the tendermint daemon and the state machine that processes the transactions, something akin to wsgi as the interface between apache/nginx and a django application.

There is an

in-depth reference to the ABCI protocol, but in short, an ABCI application is a state machine that responds to messages sent from one client (the tendermint consensus engine). It is run in parallel on every node, and they must all run the same set of transactions (what was included in the blocks), and then verify they have the same result (merkle root).

The main messages that you need to be concerned with are:

- Validation - CheckTx

  Before including a transaction, or gossiping it to peers, every node will call `CheckTx` to check if it is valid. This should be a best-attempt filter, we may reject transactions that are included in the block, but this should eliminate much spam

- Execution of Blocks

  After a block is written to the chain, the tendermint engine makes a number of calls to process it. These are our hooks to make any *writes* to the datastore.

  - BeginBlock

    BeginBlock provides the new header and block height. You can also use this as a hook to trigger any delayed tasks that will execute at a given height. (see `Ticker` below)

  - DeliverTx - once per transaction

    DeliverTx is passed the raw bytes, just like CheckTx, but it is expected to process the transactions and write the state changes to the key-value store. This is the most important call to trigger any state change.

  - EndBlock

    After all transactions have been processed, EndBlock is a place to communicate any configuration changes the application wishes to make on the tendermint engine. This can be changes to the validator set that signs the next block, or changes to the consensus parameters, like max block size, max numbers of transactions per block, etc.

  - Commit

    After all results are returned, a Commit message is sent to flush all data to disk. This is an atomic operation, and after a crash, the state should be that after executing block `H` entirely, or block `H+1` entirely, never somewhere in between (or else you are punished by rebuilding the blockchain state by replaying the entire chain from genesis. . .)

- Query

  A client also wishes to *read* the state. To do so, they may query arbitrary keys in the datastore, and get the current value stored there. They may also fix a recent height to query, so they can guarantee to get a consistent snapshot between multiple queries even if a block was committed in the meantime.

  A client may also request that the node returns a merkle proof for the key-value pair. This proof is a series of hashes, and produces a unique root hash after passing the key-value pair through the list. If this root hash matches the `AppHash` stored in a blockheader, we know that this value was agreed upon by consensus, and we can trust this is the true value of the chain, regardless of whether we trust the node we connect to.

  If you are interested, you can read more about using validating light clients with tendermint

5.1.3 Persistence

All data structures that go over the wire (passed on any external interface, or saved to the key value store), must be able to be serialized and de-serialized. An application may have any custom binary format it wants, and to support this flexibility, we provide a `Persistent` interface to handle marshaling similar to the `encoding/json` library.

```
type Persistent interface {
    Marshal() ([]byte, error)
    Unmarshal([]byte) error
}
```

Note that Marshal can work with a struct, while Unmarshal (almost) always requires a pointer to work properly. You may define these two functions for every persistent data structure in your code, using any codec you want. However, for simplicity and cross-language parsing on the client size, we recommend to define `.proto` files and compile them with protobuf.

gogo protobuf will autogenerate Marshal and Unmarshal functions requiring no reflection. See the Makefile for `tools` and `protoc` which show how to automate installing the protobuf compiler and compiling the files.

However, if you have another favorite codec, feel free to use that. Or mix and match. Each struct can use it's own Marshaller.

Before we get into the strucutre of the application, there are a few design principles for weave (but also tendermint apps in general) that we must keep in mind.

5.2 Determinism

The big key to blockchain development is determinism. Two binaries with the same state must **ALWAYS** produce the same result when passed a given transaction. Read More

5.3 Abstract Block Chain Interface (ABCI)

To understand this design, you should first understand what an ABCI application is and how that level blockchain abstraction works. ABCI is the interface between the tendermint daemon and the state machine that processes the transactions, something akin to wsgi as the interface between apache/nginx and a django application. Read More

5.4 Persistence

All data structures that go over the wire (passed on any external interface, or saved to the key value store), must be able to be serialized and deserialized. An application may have any custom binary format it wants, although all standard weave extensions use protobuf. Read More

CHAPTER 6

Additional Reading

We are in the process of doing a large overhaul on the docs. Until we are finished, please look at the older version of the docs for more complete (if outdated) information