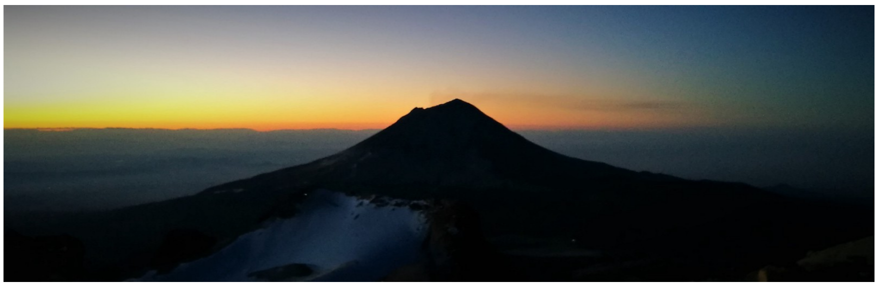
Setup your own private Proof-of-Authority Ethereum network with Geth

Salanfe Follow
Feb 11, 2018 · 14 min read



February 2018.

updates:

1. geth 1.8 was released a few days after this guide was published and fortunately does not break anything. This post is then valid and was tested for both geth 1.7.3 and geth 1.8. Awesome :)
2. I've learned *a posteriori* that the gas limit per block is dynamic. Therefore I've updated the section 2.3 to give more information about this particular case. In my private network where blocks are most of the time empty, I don't what the gas limit to decrease at all !
3. Clique requires `int(N/2+1)` sealers (where N is the number of sealers defined in the genesis file—in `extraData` field) to be online in order to run.
4. thx to Ivica Aracic for pointing out that clique PoA **DOES WORK** with a **single node**. For any reason I missed that and I apologize for the confusion. With a single node, we just need (A) create genesis file with only one sealer (only 1 address in `extraData`) (B) create an account (C) init geth (D) run geth, unlock account and mine. No bootnode is required then.
5. with geth 1.8 if you get the error "invalid host specified", try adding the option `--rpcvhosts value` to the geth command. See

`geth --help`

Goal: step by step guide to help you setup a local private ethereum network using the *Proof-of-Authority* consensus engine (also named *clique*).

In a nutshell: we will setup two nodes on the same machine, creating a peer-to-peer network on our localhost. In addition to the two nodes, a bootnode (discovery service) will also be setup.

It took me quite some time and extensive research and googling to finally have a solid ethereum development environment for testing my smart contracts and my DApps.

In this post, I've decided to share how I am setting a Proof-of-Authority network using the clique consensus engine of Geth. It's my way to thank the community by giving back and hopefully making life easier for anyone willing exploring the Ethereum universe.

OS and Software

My OS is Ubuntu 16.04 LTS (this tuto was done in a fresh virtual machine).

For the Ethereum client, I am using Geth (the Go implementation of the Ethereum protocole). I believe that Geth is easy to install with plenty of great tutorials out there, so I am not gonna cover any installation here. I am currently running Geth 1.7.3-stable:

```
$ geth version
Geth
Version: 1.7.3-stable
Git Commit: 4bb3c89d44e372e6a9ab85a8be0c9345265c763a
Architecture: amd64
Protocol Versions: [63 62]
```

and Geth 1.8.1-stable

```
$ geth version
Geth
Version: 1.8.1-stable
Git Commit: 1e67410e88d2685bc54611a7c9f75c327b553ccc
Architecture: amd64
Protocol Versions: [63 62]
```

I strongly recommend to check the Geth Command Line Interface documentation. You're gonna need it. A lot.

1. Let's get started

1.0 overview

Let's start by the end… For clarity, this is what you are supposed to get when you will have completed Chapter 1.

```
devnet$ tree -L 2
.
├── accounts.txt
├── boot.key
├── genesis.json
├── node1
│   ├── geth
│   ├── keystore
│   └── password.txt
└── node2
    ├── geth
    ├── keystore
    └── password.txt
```

1.1 create a workspace

```
$ mkdir devnet
$ cd devnet
devnet$ mkdir node1 node2
```

1.2 create your accounts

The accounts (also called wallet) hold a private-public key pair that are required for interacting with any blockchain. Any mining node (strictly speaking our nodes will not be mining but voting) needs to be able to sign transactions (using their private key) and to identify itself on the network (the address is derived from the public key). Therefore we need at least two accounts, one per node.

In Geth jargon, a voting node is called a Sealer.

for node 1 :

```
devnet$ geth --datadir node1/ account new
Your new account is locked with a password. Please give a
password. Do not forget this password.
Passphrase: pwdnode1 (for example)
Repeat passphrase: pwdnode1
Address: {87366ef81db496edd0ea2055ca605e8686eec1e6}
```

for node 2 :

```
devnet$ geth --datadir node2/ account new
Your new account is locked with a password. Please give a
password. Do not forget this password.
Passphrase: pwdnode2 (for example)
Repeat passphrase: pwdnode2
Address: {08a58f09194e403d02a1928a7bf78646cfc260b0}
```

This creates the `keystore/` folder containing your account file. Notice that the last part of the file name in `keystore/` is the address of your account (also printed in the terminal just above).

I suggest to copy these two addresses from the terminal screen and to save them in a text file. That will ease some copy-pasting job later on. However remember that you can read those addesses from the `UTC-datetime-address` file in `keystore/`.

```
devnet$ echo '87366ef81db496edd0ea2055ca605e8686eec1e6' >>
accounts.txt
devnet$ echo '08a58f09194e403d02a1928a7bf78646cfc260b0' >>
accounts.txt
```

For each node, I propose to save your password in a file. That will ease some process for later on (such as unlocking your account)

```
devnet$ echo 'pwdnode1' > node1/password.txt
devnet$ echo 'pwdnode2' > node2/password.txt
```

1.3 create your Genesis file

A genesis file is the file used to initialize the blockchain. The very first block, called the genesis block, is crafted based on the parameters in the `genesis.json` file.

Geth comes with a bunch of exectuables such as `puppeth` or `bootnode` . You can find the complete list on the Geth github. Puppeth removes the pain of creating a genesis file from scratch (and does much more). Start puppeth :

```
devnet$ puppeth
```

and happily answer the questions (every value can be updated by hand later on, so don't spent too much time engineering it for your first trials).

```
Please specify a network name to administer (no spaces,
please)
> devnet
What would you like to do? (default = stats)
 1. Show network stats
 2. Configure new genesis
 3. Track new remote server
 4. Deploy network components
> 2

Which consensus engine to use? (default = clique)
 1. Ethash - proof-of-work
 2. Clique - proof-of-authority
> 2

How many seconds should blocks take? (default = 15)
> 5 // for example

Which accounts are allowed to seal? (mandatory at least one)
> 0x87366ef81db496edd0ea2055ca605e8686eec1e6 //copy paste
from account.txt :)
> 0x08a58f09194e403d02a1928a7bf78646cfc260b0


Which accounts should be pre-funded? (advisable at least
one)
> 0x87366ef81db496edd0ea2055ca605e8686eec1e6 // free ethers
!
> 0x08a58f09194e403d02a1928a7bf78646cfc260b0

Specify your chain/network ID if you want an explicit one
(default = random)
> 1515 // for example. Do not use anything from 1 to 10

Anything fun to embed into the genesis block? (max 32 bytes)
>
```

```
What would you like to do? (default = stats)
 1. Show network stats
 2. Manage existing genesis
 3. Track new remote server
 4. Deploy network components
> 2

1. Modify existing fork rules
 2. Export genesis configuration
> 2

Which file to save the genesis into? (default = devnet.json)
> genesis.json
INFO [01-23|15:16:17] Exported existing genesis block

What would you like to do? (default = stats)
 1. Show network stats
 2. Manage existing genesis
 3. Track new remote server
 4. Deploy network components
> ^C // ctrl+C to quit puppeth
```

Side note : from Clique PoA EIP#225

> *PoA doesn't have mining rewards*

So I would highly suggest that you allocate some ethers (defined in the unit of wei) to a bunch of addresses in the genesis file, otherwise you'll hand up without any ether and thus will not be able to pay for your transactions. You could have a `gasPrice` of zero but that sometimes leads to undesired behavior from the nodes that could go under the radar (like not broadcasting pending transaction depending on the config of the other nodes on the network). I encourage you nevertheless to play with every parameter :)

1.4 Initialize your nodes

Now that we have the `genesis.json` file, let's forge the genesis block ! Each node **MUST** be initialize with the **SAME** genesis file.

```
devnet$ geth --datadir node1/ init genesis.json
devnet$ geth --datadir node2/ init genesis.json
```

tada ! done.

Side note : how does your node know about the genesis parameters when joining the Ethereum Mainnet or the Ropsten testnet, or the

Rinkeby testnet ? They are already defined in the source code in `params/config.go` .

1.5 Create a bootnode

A bootnode only purpose is to helping nodes discovering each others (remember, the Ethereum blockchain is a peer-to-peer network). Nodes could have dynamic IP, being turned off, and on again. The bootnode is usually ran on a static IP and thus acts like a pub where nodes know they will find their mates.

Initialize the bootnode :

```
devnet$ bootnode -genkey boot.key
```

This creates a value called the `enode` uniquely identifying your bootnode (more on this soon) and we store this enode in the `boot.key` file.

1.6 Midway celebration

Congrats ! Chapter 1 is done :) try

```
devnet$ tree -L 2
```

and compare the output with the section 1.0. Hopefully you should get the same tree.

At this point the setup is done and we are ready to make this blockchain live.

2. Make it live

2.1 Start the bootnode service

```
devnet$ bootnode -nodekey boot.key -verbosity 9 -addr :30310
INFO [02-07|22:44:09] UDP listener up
self=enode://3ec4fef2d726c2c01f16f0a0030f15dd5a81e274067af2b
2157cafbf76aa79fa9c0be52c6664e80cc5b08162ede53279bd70ee10d02
4fe86613b0b09e1106c40@[::]:30310
```

I like to have some verbosity for my bootnode as it is nice to see when the nodes are playing ping-pong on the network (meaning it's working!).

Feel free to use any port you like but please avoid the mainstream ones (like 80 for HTTP). `30303` is used for the public ethereum networks.

2.2 Starting your nodes

Big time ! Finally (but usually here the troubles arrive too).

Everything in one huge command ! I am gonna cover some options but please do your homework and refer to the doc.

starting node 1

```
devnet$ geth --datadir node1/ --syncmode 'full' --port 30311
--rpc --rpcaddr 'localhost' --rpcport 8501 --rpcapi
'personal,db,eth,net,web3,txpool,miner' --bootnodes
'enode://3ec4fef2d726c2c01f16f0a0030f15dd5a81e274067af2b2157
cafbf76aa79fa9c0be52c6664e80cc5b08162ede53279bd70ee10d024fe8
6613b0b09e1106c40@127.0.0.1:30310' --networkid 1515 --
gasprice '1' -unlock
'0x87366ef81db496edd0ea2055ca605e8686eec1e6' --password
node1/password.txt --mine
```

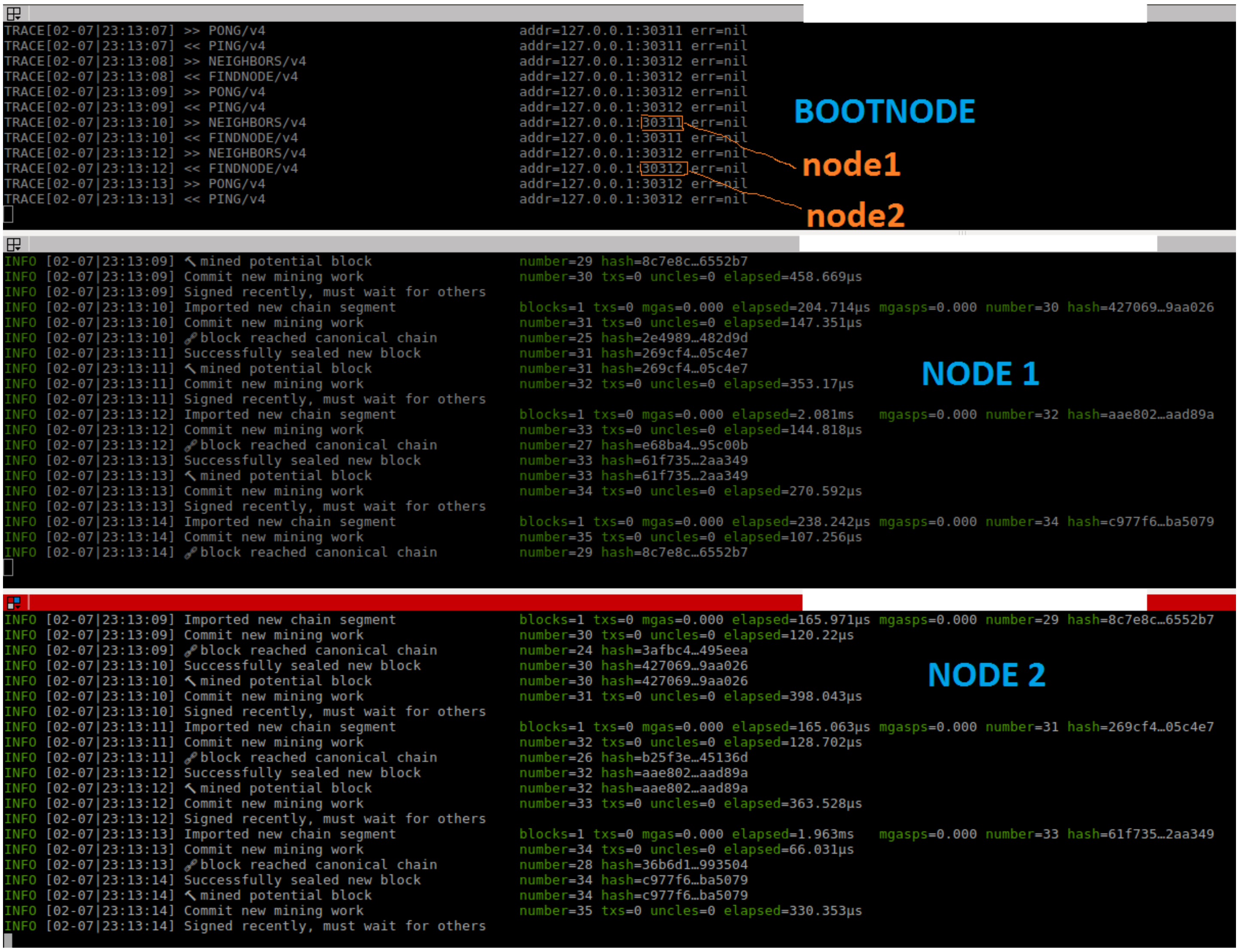
- `--syncmode 'full'` helps preventing the error Discarded Bad Propagated Block.
- `--port 30311` is the enode port for node1 and has to be different from the bootnode port (that is `30310` if you followed my command) because we are on a localhost. On a real network (one node per machine), use the same port.
- `--rpcapi` allows the listed modules to be used over RPC calls (see section 3.3 for an example). See the Geth Management APIs for more info. Be mindful about hacks as everyone can call your RPC methods if no firewall is protecting your node.
- `--bootnodes` tells your node at what address to find your bootnode. Replace `[::]` with the bootnode IP. **No domain name are allowed** ! Only IPs. Check enode URL format.

- `--networkId` as defined in the `genesis.json` file. Please use the same id !
- `--gasprice '1'` I don't like to pay on my own network :) be careful with gasprice. If your transactions are not being broadcasted to the network but only the node receiving the transactions is processing them, this means you sent a transaction with a gasprice that is not accepted (too low) by the other nodes on the network. No error will be return. If you have two nodes, only one will be processing the transactions. This is sneaky and reduces your network throughput by a factor 2.
- `--unlock` `--password` `--mine` tell the node to unlock this account, with the password in that file and to start mining (i.e. voting/sealing for Proof-of-Authority)
- `--targetgaslimit value` see the **update** in section 2.3.

same for node 2 (update parameters specific to the node)

```
devnet$ geth --datadir node2/ --syncmode 'full' --port 30312
--rpc --rpcaddr 'localhost' --rpcport 8502 --rpcapi
'personal,db,eth,net,web3,txpool,miner' --bootnodes
'enode://3ec4fef2d726c2c01f16f0a0030f15dd5a81e274067af2b2157
cafbf76aa79fa9c0be52c6664e80cc5b08162ede53279bd70ee10d024fe8
6613b0b09e1106c40@127.0.0.1:30310' --networkid 1515 --
gasprice '0' --unlock
'0x08a58f09194e403d02a1928a7bf78646cfc260b0' --password
node2/password.txt --mine
```

At this point your bootnode should stream connections coming from node1 (port 30311) and node2 (port 30312) as shown in the upper terminal window. Node1 (middle terminal) and node2 (lower terminal) should be happily mining and signing blocks. Here I have a period of 1 second (defined in the genesis file) therefore the fast block creation.



```
TRACE[02-07|23:13:07] >> PONG/v4                      addr=127.0.0.1:30311 err=nil
TRACE[02-07|23:13:07] << PING/v4                      addr=127.0.0.1:30311 err=nil
TRACE[02-07|23:13:08] >> NEIGHBORS/v4                 addr=127.0.0.1:30312 err=nil
TRACE[02-07|23:13:08] << FINDNODE/v4                  addr=127.0.0.1:30312 err=nil
TRACE[02-07|23:13:09] >> PONG/v4                      addr=127.0.0.1:30312 err=nil
TRACE[02-07|23:13:09] << PING/v4                      addr=127.0.0.1:30312 err=nil
TRACE[02-07|23:13:10] >> NEIGHBORS/v4                 addr=127.0.0.1:30311 err=nil
TRACE[02-07|23:13:10] << FINDNODE/v4                  addr=127.0.0.1:30311 err=nil
TRACE[02-07|23:13:12] >> NEIGHBORS/v4                 addr=127.0.0.1:30312 err=nil
TRACE[02-07|23:13:12] << FINDNODE/v4                  addr=127.0.0.1:30312 err=nil
TRACE[02-07|23:13:13] >> PONG/v4                      addr=127.0.0.1:30312 err=nil
TRACE[02-07|23:13:13] << PING/v4                      addr=127.0.0.1:30312 err=nil
```

```
INFO [02-07|23:13:09] ⛏ mined potential block          number=29 hash=8c7e8c…6552b7
INFO [02-07|23:13:09] Commit new mining work            number=30 txs=0 uncles=0 elapsed=458.669µs
INFO [02-07|23:13:09] Signed recently, must wait for others
INFO [02-07|23:13:10] Imported new chain segment        blocks=1 txs=0 mgas=0.000 elapsed=204.714µs mgasps=0.000 number=30 hash=427069…9aa026
INFO [02-07|23:13:10] Commit new mining work            number=31 txs=0 uncles=0 elapsed=147.351µs
INFO [02-07|23:13:10] 🔗 block reached canonical chain   number=25 hash=2e4989…482d9d
INFO [02-07|23:13:11] Successfully sealed new block     number=31 hash=269cf4…05c4e7
INFO [02-07|23:13:11] ⛏ mined potential block          number=31 hash=269cf4…05c4e7
INFO [02-07|23:13:11] Commit new mining work            number=32 txs=0 uncles=0 elapsed=353.17µs
INFO [02-07|23:13:11] Signed recently, must wait for others
INFO [02-07|23:13:12] Imported new chain segment        blocks=1 txs=0 mgas=0.000 elapsed=2.081ms   mgasps=0.000 number=32 hash=aae802…aad89a
INFO [02-07|23:13:12] Commit new mining work            number=33 txs=0 uncles=0 elapsed=144.818µs
INFO [02-07|23:13:12] 🔗 block reached canonical chain   number=27 hash=e68ba4…95c00b
INFO [02-07|23:13:13] Successfully sealed new block     number=33 hash=61f735…2aa349
INFO [02-07|23:13:13] ⛏ mined potential block          number=33 hash=61f735…2aa349
INFO [02-07|23:13:13] Commit new mining work            number=34 txs=0 uncles=0 elapsed=270.592µs
INFO [02-07|23:13:13] Signed recently, must wait for others
INFO [02-07|23:13:14] Imported new chain segment        blocks=1 txs=0 mgas=0.000 elapsed=238.242µs mgasps=0.000 number=34 hash=c977f6…ba5079
INFO [02-07|23:13:14] Commit new mining work            number=35 txs=0 uncles=0 elapsed=107.256µs
INFO [02-07|23:13:14] 🔗 block reached canonical chain   number=29 hash=8c7e8c…6552b7
```

```
INFO [02-07|23:13:09] Imported new chain segment        blocks=1 txs=0 mgas=0.000 elapsed=165.971µs mgasps=0.000 number=29 hash=8c7e8c…6552b7
INFO [02-07|23:13:09] Commit new mining work            number=30 txs=0 uncles=0 elapsed=120.22µs
INFO [02-07|23:13:09] 🔗 block reached canonical chain   number=24 hash=3afbc4…495eea
INFO [02-07|23:13:10] Successfully sealed new block     number=30 hash=427069…9aa026
INFO [02-07|23:13:10] ⛏ mined potential block          number=30 hash=427069…9aa026
INFO [02-07|23:13:10] Commit new mining work            number=31 txs=0 uncles=0 elapsed=398.043µs
INFO [02-07|23:13:10] Signed recently, must wait for others
INFO [02-07|23:13:11] Imported new chain segment        blocks=1 txs=0 mgas=0.000 elapsed=165.063µs mgasps=0.000 number=31 hash=269cf4…05c4e7
INFO [02-07|23:13:11] Commit new mining work            number=32 txs=0 uncles=0 elapsed=128.702µs
INFO [02-07|23:13:11] 🔗 block reached canonical chain   number=26 hash=b25f3e…45136d
INFO [02-07|23:13:12] Successfully sealed new block     number=32 hash=aae802…aad89a
INFO [02-07|23:13:12] ⛏ mined potential block          number=32 hash=aae802…aad89a
INFO [02-07|23:13:12] Commit new mining work            number=33 txs=0 uncles=0 elapsed=363.528µs
INFO [02-07|23:13:12] Signed recently, must wait for others
INFO [02-07|23:13:13] Imported new chain segment        blocks=1 txs=0 mgas=0.000 elapsed=1.963ms   mgasps=0.000 number=33 hash=61f735…2aa349
INFO [02-07|23:13:13] Commit new mining work            number=34 txs=0 uncles=0 elapsed=66.031µs
INFO [02-07|23:13:13] 🔗 block reached canonical chain   number=28 hash=36b6d1…993504
INFO [02-07|23:13:14] Successfully sealed new block     number=34 hash=c977f6…ba5079
INFO [02-07|23:13:14] ⛏ mined potential block          number=34 hash=c977f6…ba5079
INFO [02-07|23:13:14] Commit new mining work            number=35 txs=0 uncles=0 elapsed=330.353µs
INFO [02-07|23:13:14] Signed recently, must wait for others
```

2.3 Update your genesis file

I am sure you'll want to modify some values in your genesis file. Go ahead ! However in order for those changes to become effective, we have to initialize a new blockchain. Here is the genesis file I am currently using :

```
{
    "config": {
        "chainId": 1515,
        "homesteadBlock": 1,
        "eip150Block": 2,
        "eip150Hash":
"0x0000000000000000000000000000000000000000000000000000000000000000",
        "eip155Block": 3,
        "eip158Block": 3,
        "byzantiumBlock": 4,
        "clique": {
            "period": 1,
            "epoch": 30000
        }
    },
    "nonce": "0x0",
    "timestamp": "0x5a722c92",
    "extraData":
"0x000000000000000000000000000000000000000000000000000000000
000000008a58f09194e403d02a1928a7bf78646cfc260b087366ef81db49
6edd0ea2055ca605e8686eec1e6000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000",
    "gasLimit": "0x59A5380",
    "difficulty": "0x1",
    "mixHash":
"0x000000000000000000000000000000000000000000000000000000000
0000000",
    "coinbase":
"0x0000000000000000000000000000000000000000",
    "alloc": {
        "08a58f09194e403d02a1928a7bf78646cfc260b0": {
            "balance":
"0x200000000000000000000000000000000000000000000000000000000
000000"
        },
        "87366ef81db496edd0ea2055ca605e8686eec1e6": {
            "balance":
"0x200000000000000000000000000000000000000000000000000000000
000000"
        },
        "F464A67CA59606f0fFE159092FF2F474d69FD675": {
            "balance":
"0x200000000000000000000000000000000000000000000000000000000
000000"
        }
    },
    "number": "0x0",
    "gasUsed": "0x0",
    "parentHash":
"0x000000000000000000000000000000000000000000000000000000000
0000000"
}
```

I've cleaned the empty addresses that puppeth includes when creating the file (at section 1.3). I've also added a third address that gets funded when the genesis block is created. Then I have changed the `period` from 15 second to 1 to get those blocks mined faster (be careful as one empty block weights 1024 bytes—here my `chaindata/` folder gains 1024 bytes per second (and more if the blocks are not empty). Finally I've increased the `gasLimit` to allow for more transaction (trully speaking, computation) per block.

update: The `gasLimit` defined in the genesis file only applies to the genesis block ! The `gasLimit` of new blocks is **DYNAMIC** meaning its value is changing over time depending on how much gas was used in the parent (previous) block. The computation of the new `gasLimit` is done in the function `CalcGasLimit` (github source). If you want a constant gas Limit use the option `--targetgaslimit intValue` when running geth. I would recommend to set it equal to the `gasLimit` in the genesis file (the command option is an integer whereas the genesis value is hexadecimal) so that you get a constant gas limit that does not change over time anymore. Given the genesis file above with `"gasLimit":"0x59A5380"` , I am running my node with `--targetgaslimit 94000000` for a constant gas limit across all blocks.

The field `extraData` contains the address that are allowed to seal (that's why puppeth is nice to have).

I have investigate the impact of changing the `period` and the `gasLimit` on the number of transaction per second (transaction rate) that the blockchain can process. But that's gonna be another article; link here.

When you are happy with your genesis file. Kill your nodes if they are running (ctrl C in the terminal). Then delete the folder `geth/` in `node1/` and `geht/` in `node2/` . Delete only `geth/` folders!

Then initialize your nodes. From section 1.4 :

```
devnet$ geth --datadir node1/ init genesis.json
devnet$ geth --datadir node2/ init genesis.json
```

and start your nodes again with the commands in section 2.2

3. Interact with your nodes

Great your network is now live :) but how to connect to it and starting exploring ?

3.1 Open a Geth Javascript Console

The simplest and probably more straight forward way to play with a node is probably to attach a Geth javascript console to one of the nodes.

3.1.1 Through IPC

IPC (**I**nter-**P**rocess **C**ommunication) works only locally : you should be on the same machine as your node. Open an extra terminal and attach to your node. To connect to node1 :

```
$ cd devnet
devnet$ geth attach node1/geth.ipc
Welcome to the Geth JavaScript console!

instance: Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9
coinbase: 0x87366ef81db496edd0ea2055ca605e8686eec1e6
at block: 901 (Sat, 10 Feb 2018 21:15:30 CET)
 datadir: /home/salanfe/privateNetworks/devnet/node1
 modules: admin:1.0 clique:1.0 debug:1.0 eth:1.0 miner:1.0
net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

>
```

The file `geth.ipc` is created only when the node is running. So do not expect to find it if your node1 is off.

RPC gives access without restriction to all modules listed in the terminal : `admin: 1.0 clique:1.0 debug:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0`

3.1.2 Through RPC

RPC (**R**emote **P**rocedure **C**all) works over the internet as HTTP requests. Therefore be careful when you open RPC to the outside world as everyone will have access to your node. For this reason RPC is disabled by default and when enabled it does not give access to all modules. In this guide we allowed RPC on our Geth node with the command `--rpc` and gave access to the modules `personal,db,eth,net,web3,txpool,miner` (from section 2.2). To connect to node1 using RPC :

```
$ cd devnet
devnet$ geth attach 'http://localhost:8501'
Welcome to the Geth JavaScript console!

instance: Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9
coinbase: 0x87366ef81db496edd0ea2055ca605e8686eec1e6
at block: 945 (Sat, 10 Feb 2018 21:16:14 CET)
 modules: eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0
txpool:1.0 web3:1.0

>
```

3.1.3 Using the Geth Javascript Console

Here are some examples of methods

```
> net.version
"1515"
> eth.blockNumber
1910
> eth.coinbase
"0x87366ef81db496edd0ea2055ca605e8686eec1e6"
> eth.sendTransaction({'from':eth.coinbase,
'to':'0x08a58f09194e403d02a1928a7bf78646cfc260b0',
'value':web3.toWei(3, 'ether')})
"0x299a99baa1b39bdee5f02e3c660e19e744f81c2e886b5fc24aa83f92f
e100d3f"
>
eth.getTransactionReceipt("0x299a99baa1b39bdee5f02e3c660e19e
744f81c2e886b5fc24aa83f92fe100d3f")
{
  blockHash:
"0x212fb593980bd42fcaf3f6d1e6db2dd86d3764df8cac2d90408f481ae
7830de8",
  blockNumber: 2079,
  contractAddress: null,
  cumulativeGasUsed: 21000,
  from: "0x87366ef81db496edd0ea2055ca605e8686eec1e6",
  gasUsed: 21000,
  logs: [],
  logsBloom:
"0x000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000",
  status: "0x1",
  to: "0x08a58f09194e403d02a1928a7bf78646cfc260b0",
  transactionHash:
"0x299a99baa1b39bdee5f02e3c660e19e744f81c2e886b5fc24aa83f92f
e100d3f",
  transactionIndex: 0
}
> exit // to quit the Geth javascript console
```

for the full list of methods, see Management APIs and JSON RPC API.

3.2 Using Mist

The Mist browser provides a graphical user interface for deploying and interacting with smart contracts and managing accounts. To connect Mist to your local private network over IPC, simply do :

```
devnet$ mist --rpc node1/geth.ipc
```

and over RPC (make sure RPC is enabled)

```
$ mist --rpc 'http://localhost:8501'
```

The procedure is exactly the same if you want to use the Ethereum wallet instead of mist. Just replace mist by `ethereumwallet` in the commands above.

3.3 Making RPC calls with your favorite programming language

In section 3.1, we saw how to interact with the Geth API by hand. Now let's use our PC for what it is best at : automation.

The reference and by far for sending JSON-RPC requests to your node is the web3.js javascript library. I believe the internet of full of great tutorial and example on how to use the web3.js library. Therefore I am not gonna covert any of it here.

The JSON-RPC APIs are currently also being implemented in java with the web3.j library and in python with the web3.py library. Those libraries offer high-level methods for working with the ethereum blockchain just like web3.js.

However, it's also possible to send raw JSON-RPC requests directly to your node. I think it is worth trying as it's providing a valuable understanding on how those high-level libraries work under the hood.

Here is a simple example of sending a raw JSON-RPC request to my node using python 3 :

```
$ python
Python 3.6.4 |Anaconda custom (64-bit)| (default, Jan 16
2018, 18:10:19)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import requests
>>> import json
>>> session = requests.Session()
>>> method = 'eth_getTransactionCount'
>>> params =
["0x627306090abaB3A6e1400e9345bC60c78a8BEf57","latest"]
>>> PAYLOAD = {"jsonrpc":"2.0",
...            "method":method,
...            "params":params,
...             "id":67}
>>> PAYLOAD = json.dumps(PAYLOAD)
>>> headers = {'Content-type': 'application/json'}
>>> response = session.post('http://127.0.0.1:8501',
data=PAYLOAD, headers=headers)
>>> response.content
b'{"jsonrpc":"2.0","id":67,"result":"0x0"}\n'
>>> json.loads(response.content)['result']
'0x0'
```

The method `geth_transactionCount` is documented here.

The "account" nonce is a transaction counter and has nothing to do with the nonce for Proof-of-Work. A account nonce of zero means that the address `0x627306090abaB3A6e1400e9345bC60c78a8BEf57` never did any transaction on the network : `0x0` is the hexadecimal representation of zero.

3.4 Deploy and test your smart contracts with Truffle on your private network

Development frameworks like Truffle (or Embark, Populus) are great tools for developing and testing smart contracts.

When you initialize a workspace with

```
$ truffle init
```

Truffle creates a series of files and folders to help you get started. I usually edit the `truffle.js` file as such

```
module.exports = {
    // See
<http://truffleframework.com/docs/advanced/configuration>
    // to customize your Truffle configuration!
    networks: {
        devnet: {
            host: '127.0.0.1',
            port: 8501,
            network_id: '*'
        },
        ganache: {
            host: '127.0.0.1',
            port: 7545,
            network_id: '*'
        }
    }
};
```

then use the command

```
$ truffle deploy --network devnet
```

to deploy your smart contracts defined in `migrations/X_deploy.js` . Or for running your tests in `test/`

```
$ truffle test --network devnet
```

Usually the Ethereum Blockchain simulator Ganache is more than enough for running your tests. However I like to use my private blockchain for ultimate testing on a real node and not only on a simulator. With Ganache I believe that the layer of abstraction is too big, what is the beauty of it but also a danger as it requires no understanding what so ever of the complexity of a real node (transaction pool, gasPrice, gasLimit, broadcasting transactions between nodes, mining or voting, computation time, consensus engine, etc.).

What's next ?

That's pretty much it for this guide. If you understand everything here I believe you're already on very good tracks and you have a solid foundation on which you can continue your journey with confidence.

You can start developing Dapps (Decentralized Applications) by grabbing a web3 library or by making your own custom JSON-RPC wrapper.

In this post, I explore how to use python for deploying and transacting with a smart contract using only raw HTTP requests.

Ethereum: create raw JSON-RPC requests with Python for deploying and transacting with a...

February 2018.

medium.com

Final Words

Congratulation if you made it until the end. I hope this guide is comprehensive and helped you on your journey. I welcome any feedback to improve this guide !

And a BIG Thank you to the community for all the documentation, tutorials, Q&A websites and guides out there.

Happy Hacking !