DLT5400 - DLT Implementation and Internals

Assignment P-3 (geth)

Balaganapathy Vanagiri Selvakumar

Centre for Distributed Ledger Technologies

Prof. Joshua Ellul

May 31, 2023

1 Q3.2 - Private Network using geth client

1.1 Introduction

Geth is a popular Go implementation of the Ethereum blockchain client. It allows users to connect to and interact with the Ethereum network. In this report, we will outline the steps to download and build Geth from the source and run a private network.

1.2 Pre-Requisites

Before downloading Geth, it is necessary to install some prerequisites. These include *Go*, *Git*, and *make*. Instructions for installing Go, Git, and make can be found on their respective websites. Make sure that you install versions that are suitable for your machine.

1.3 Geth

Once the prerequisites are installed, we can download and build Geth from source. This is done by running the following commands:

```
git clone https://github.com/ethereum/go-ethereum.git
cd go-ethereum
make geth
```

The first command clones the Geth repository from GitHub, and the second command changes the directory to the cloned repository and builds the Geth binary using make.

1.4 Accounts

This is a small step that can be used to generate an Ethereum account, that can be used in further steps.

```
geth --datadir data account new
```

You can create multiple accounts using the above command. Follow the instructions on the terminal to generate your public key and keystore file. Save the password that was used in a text file named *password.txt*.

```
(base) bala_s@Balaganapathys-MacBook-Pro privateGeth % geth --datadir node1 account new
INFO [05-15|20:07:24.1921 Maximum peer count ETH=50 LES=0 total=50
Your new account is locked with a password. Please give a password. Do not forget this
password.
Password:
Repeat password:

Your new key was generated

Public address of the key: OxeAA6d98a3658c01BcfaABA807f38d22D573Fe4dg
Path of the secret key file: node1/keystore/UTC--2023-05-15T18-07-45.230924000Z--
eaa6d98a3658c01bcfaaba807f38d22d573fe4d9

- You can share your public address with anyone. Others need it to interact with you.
- You must NEVER share the secret key with anyone! The key controls access to your funds!
- You must BACKUP your key file! Without the key, it's impossible to access account funds!
- You must REMEMBER your password! Without the password, it's impossible to decrypt the key!
```

Figure 1: New Account Output

1.5 Genesis Block

The next step is to create a Genesis block. A Genesis block is the first block of a blockchain and defines the initial state of the blockchain. We can create a Genesis block by creating a JSON file with the following content:

```
{
    "config": {
      "chainId": 12345,
      "homesteadBlock": 0,
      "eip150Block": 0,
      "eip155Block": 0,
      "eip158Block": 0,
      "byzantiumBlock": 0,
      "constantinopleBlock": 0,
      "petersburgBlock": 0,
      "istanbulBlock": 0,
      "muirGlacierBlock": 0,
      "berlinBlock": 0,
      "londonBlock": 0,
      "arrowGlacierBlock": 0,
      "grayGlacierBlock": 0,
      "clique": {
        "period": 5,
        "epoch": 30000
      }
    },
    "difficulty": "1",
    "gasLimit": "800000000",
```

```
    "extradata": "...",
    "alloc": {
      "0b70e444E1834Fd39E948c970DB94aCe309ACC29": { "balance": "500000" },
      "4C520630E1feDec204EB9278d6a5Dfd8968126fB": { "balance": "500000" }
    }
  }
```

The above file, *genesis.json* states the default values and configurations of the private chain that is being created. We create a Proof of Authority(PoA) chain with a clique consensus mechanism. We also initialize the user balance with some amount of ethers pre-loaded. Now we can initialize our private blockchain network with the above configurations using the following command

```
geth init --datadir data genesis.json
```

```
(base) bala_s@Balaganapathys-MacBook-Pro privateGeth % geth init --datadir data genesis.json
INFO [05-15|20:31:43.260] Maximum peer count                       ETH=50 LES=0 total=50
INFO [05-15|20:31:43.264] Set global gas cap                       cap=50,000,000
INFO [05-15|20:31:43.264] Initializing the KZG library             backend=gokzg
INFO [05-15|20:31:43.285] Using pebble as the backing database
INFO [05-15|20:31:43.285] Allocated cache and file handles         database=/Volumes/
Essentials/Education/UM/SEM_2/DLT_5400/geth/privateGeth/data/geth/chaindata cache=16.00MiB
handles=16
INFO [05-15|20:31:43.336] Opened ancient database                  database=/Volumes/
Essentials/Education/UM/SEM_2/DLT_5400/geth/privateGeth/data/geth/chaindata/ancient/chain
readonly=false
INFO [05-15|20:31:43.341] Successfully wrote genesis state         database=chaindata
hash=5d2429..39b94a
INFO [05-15|20:31:43.342] Using pebble as the backing database
INFO [05-15|20:31:43.342] Allocated cache and file handles         database=/Volumes/
Essentials/Education/UM/SEM_2/DLT_5400/geth/privateGeth/data/geth/lightchaindata
cache=16.00MiB handles=16
INFO [05-15|20:31:43.405] Opened ancient database                  database=/Volumes/
Essentials/Education/UM/SEM_2/DLT_5400/geth/privateGeth/data/geth/lightchaindata/ancient/
chain readonly=false
INFO [05-15|20:31:43.409] Successfully wrote genesis state         database=lightchaindata
hash=5d2429..39b94a
```

Figure 2: Genesis Block Created

1.6 Running your nodes

1.6.1 Bootstrap Node

A bootstrap node can be considered as an entry point for all the other nodes to engage in peer-to-peer discovery and join the network. Any node can be opted for the bootstrap node, as we are creating a private network for learning purposes we use a utility tool named *bootnode*. This tool helps us to create a bootstrap node easily and can be used out of the box. In order to create a bootnode we require a key that can be generated by the below command.

```
bootnode -genkey boot.key
```

Now we create a bootnode using the key that was created

```
bootnode -nodekey boot.key -addr :30305
```

You can use any port number to your preference, but it is advised to use the same.

```
(base) bala_s@Balaganapathys-MacBook-Pro privateNew % bootnode -nodekey
boot.key -addr : 30307
enode://
54dd91dc1f332db51cf2f8d3eca5cbf5e5e5083a9d6925cd147fe8149a0bc21340321686
80e2fd51be7bde608158610a8565ba504b5fd3b5e93b66adb8484900@127.0.0.1:0?
discport=3030
Note: you're using cmd/bootnode, a developer tool.
We recommend using a regular node as bootstrap node for production
deployments.
INFO [05-15 |17:14:33.508] New local node record
seq=1, 684, 163,673,507 id=e0f2a5b49649d17b ip=<nil> udp-0 tcp=0
```

Figure 3: Bootnode Creation Output

A bootstrap node can be a private node or a cloud instance in a production network. Now we can proceed with running our node.

1.6.2 Member Node

In a separate terminal start the node by using the below command.

```
./geth --datadir node1 --port 30308 --bootnodes "..."  --networkid 12345
    --unlock 0x4C520630E1feDec204EB9278d6a5Dfd8968126fB --password
    node1/password.txt --authrpc.port 8552
```

Replace *"..."* in the *–bootnodes* parameter with the enode output that was generated by the bootnode tool as shown in Figure 3. We also pass our public key and the path to our *password.txt* file for the respective account that was created in the subsection Accounts. You can create several nodes by changing the values of the field *–datadir*, *–port*, *unlock*, *password*, *authrpc.port* respectively. The node must be initialized the same way we initialized the above node *node1* using the genesis file.

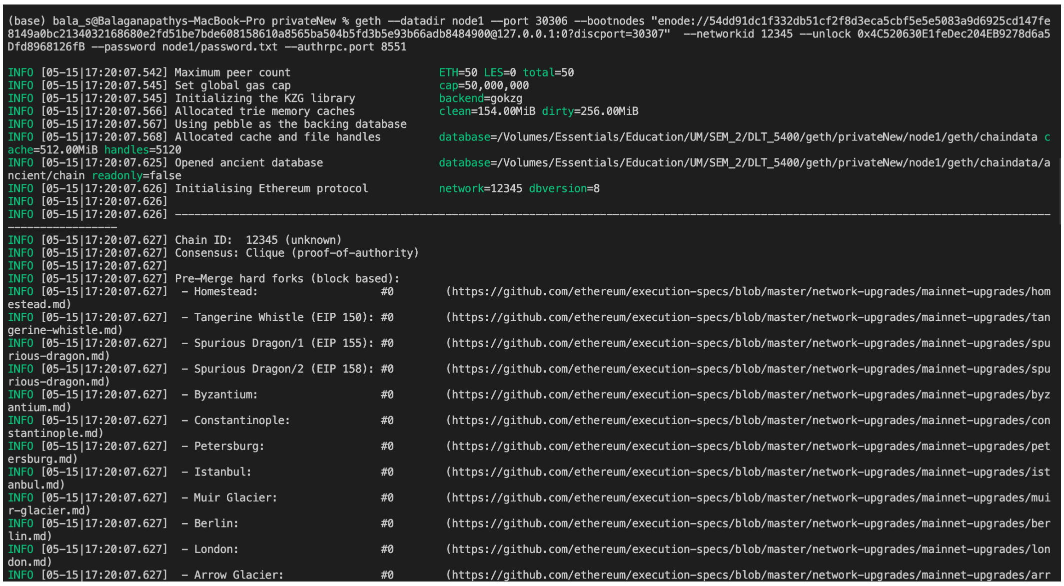

Figure 4: Running node1

In the above Figure 4, we can see that the blockchain has started with chain id 12345, as mentioned in the *genesis.json* file and the consensus is clique(PoA). We can further see some details about the peers in the below Figure 5 after spinning up another node by performing the same above-listed steps.

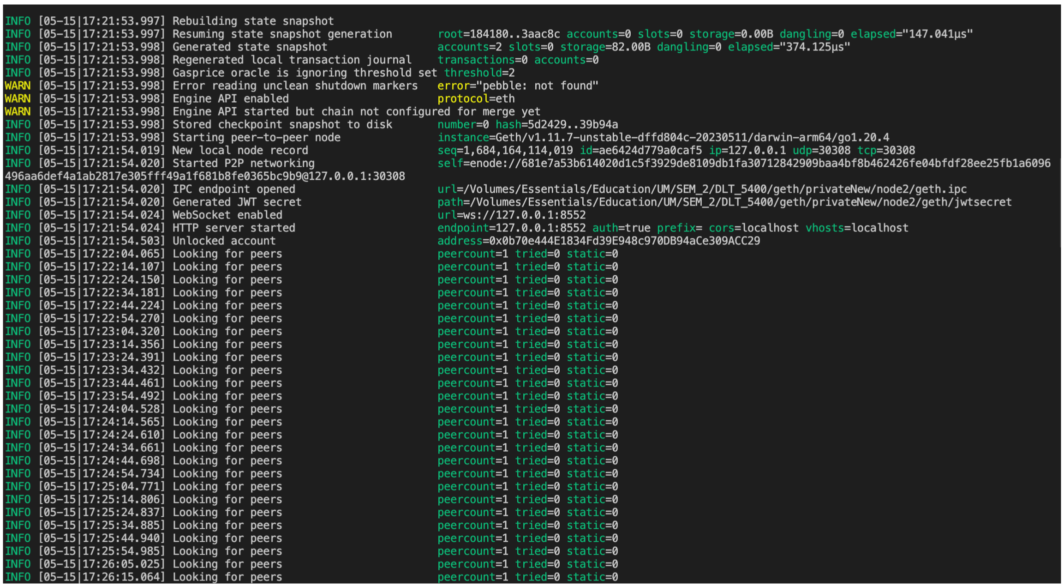

Figure 5: Node connected with a peer

1.6.3 Console for our node

We can now attach a javascript console to either one of the nodes to query network properties and perform actions.

```
geth attach node1/geth.ipc
```

By executing the above command, we now attach a console to our node *node1*

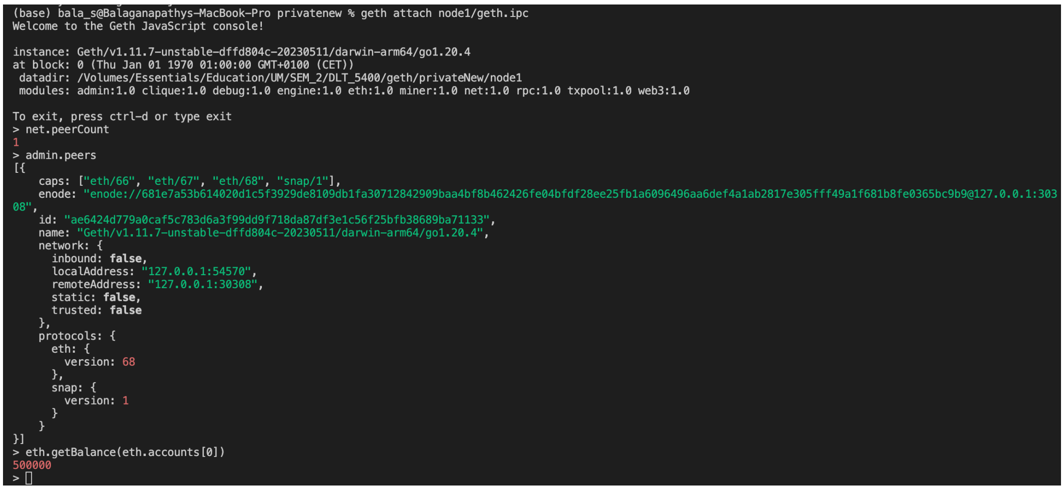

Figure 6: JavaScript console attached to a node

In the above Figure 6, you can see the JS console that can be used to do basic network queries, like fetching the number of peers attached, getting the account balance for a particular address, and fetching peer details.

2 Q3.3 - Demo of geth networks with PoA, PoS & PoW consensus

2.1 PoW Demo

2.1.1 Pre-requisites

As Ethereum has moved to proof of stake consensus, it is not possible for us to run a PoW ethereum network. In order to do that we will be using an older version of geth. We create an older version of geth binary by performing the same tasks listed in sub-section 1.3, but with a different version of the geth repository.

```
git clone https://github.com/ethereum/go-ethereum.git
```

The above command clones the github repository of geth version 1.9, which will help us build geth binary which will support ethash or PoW(Mining).

2.1.2 Genesis Block

We need to configure our blockchain using our *genesis.json* file as mentioned in sub - section 1.5. Here we will be using a different genesis file in order to run a network with PoW consensus.

```
{
    "alloc": {},
    "config": {
        "chainId": 2019,
        "homesteadBlock": 0,
        "DAOForkBlock": 0,
        "DAOForkSupport": true,
        "eip150Block": 0,
        "eip155Block": 0,
        "eip158Block": 0
    },
    "difficulty": "0x400",
    "gasLimit": "0x989680",
    "nonce": "0x0000000000000042",
    "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "coinbase": "0x0000000000000000000000000000000000000000",
    "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "extraData": "",
    "timestamp": "0x00"
}
```

2.1.3 Start PoW Network

We can start our network by performing the below command.

```
./geth --datadir data --networkid 2019 console
```

Make sure that the genesis block has been created and the directory files and chain id is correct.

Figure 7: PoW Based private network started

Here we can see that the blockchain has started and we have peer discovery and a javascript console for us to interact with the blockchain. In order for us to start mining we need to set an account as etherbase to receive the miner rewards. You can use the Accounts that was created in sub-section 1.4 or generate new accounts using below commands.

```
personal.newAccount('yourpassword')
miner.setEtherbase("<AddressGenerated>")
```

The above commands will create a new account and set the account as the etherbase.

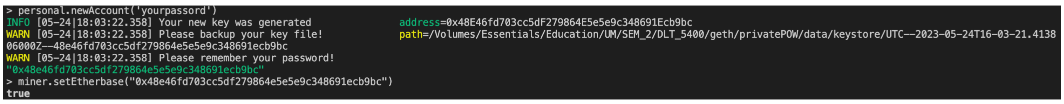

Figure 8: Etherbase account

In the above Figure 8, we can see that the etherbase has been set. Now we can start our miner and run our network's consensus.

```
miner.start()
```

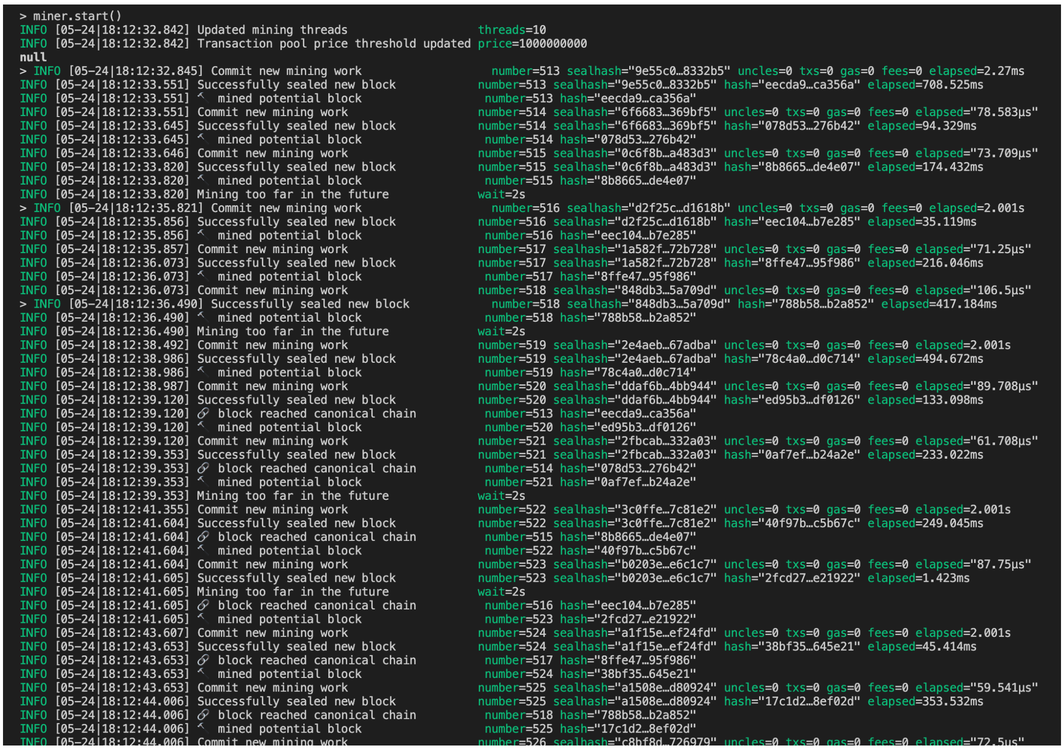

Figure 9: PoW Miner running

Here, in Figure 9, we can see that the mining process is active and new blocks are being mined and added to our private blockchain. The following commands can be used to stop the miner and kill the blockchain instance.

```
miner.stop()
exit
```

2.2 PoS Demo

2.2.1 Pre-requisites

Running a Proof of Stake private network does not make any sense as we need our own beacon chain, validator setup and several nodes to reach consensus. So for the demonstration purposes, we run a node in the ethereum mainnet using *prysm* as consensus client and *geth* as our execution client.

2.2.2 Client Setup

We can use the geth binaries that was built on sub-section 1.3. Now to set up a consensus client we need to setup *prysm* and run the same. The below steps can be followed to install and setup prysm.

```
mkdir prysm && cd prysm
```

```
curl https://raw.githubusercontent.com/prysmaticlabs/prysm/master/prysm.sh --
output prysm.sh && chmod +x prysm.sh
```

After installing the prysm binaries, we need to generate a JWT secret. Now, you generate this using an online tool like this, or generate through geth or prysm itself.

```
./prysm.sh beacon-chain generate-auth-secret
```

This will generate and store your *jwt.hex* file in a directory and output a path on the console.

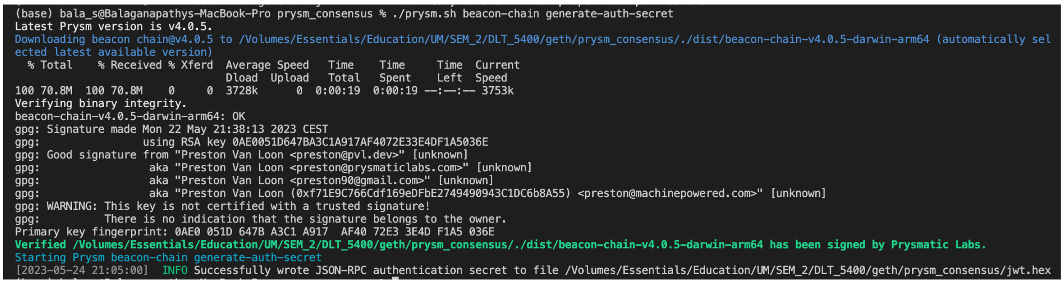

Figure 10: JWT secret generation

Now, we can proceed with running our execution client with certain flags that can help the execution client connect with the consensus client.

```
geth --http --http.api eth,net,engine,admin --authrpc.jwtsecret /path/to/jwt.hex
```

Replace "/path/to/jwt.hex" with the path generated on the previous step as shown in Figure 10. Now lets proceed with our consensus client, and run our beacon chain.

```
./prysm.sh beacon-chain --execution-endpoint=http://localhost:8551 --jwt-
secret=path/to/jwt.hex
```

In the below Figures 11 & 12, we can see that the mainnet has started with PoS consensus, and our beacon node is connected with geth. We have an execution client and a consensus client running together. We have a successfully managed to run an ethereum node with PoS consensus mechanism

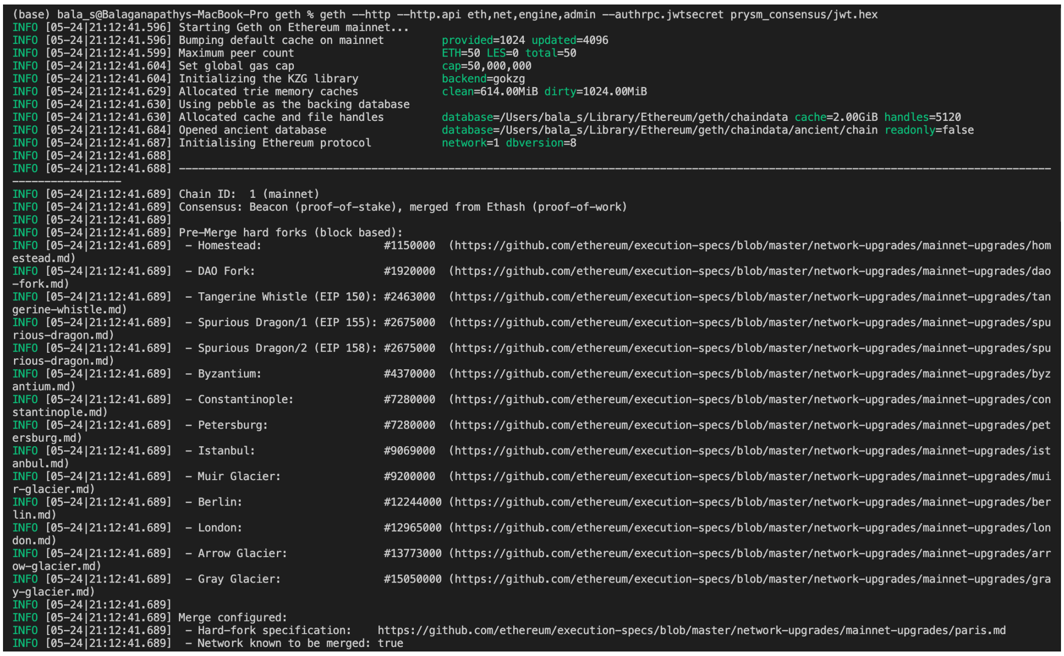

Figure 11: PoS Mainnet - Execution client(geth)

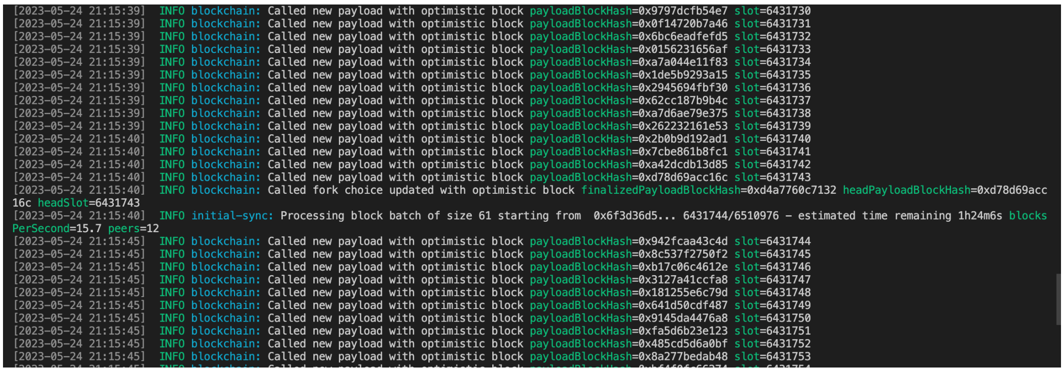

Figure 12: PoS Mainnet - Consensus client(prysm)

2.3 PoA Demo

PLEASE REFER TO SECTION 1.

As a PoA(Clique) based private network was demonstrated in section 1, we will not repeat the same here. All processes are same as the mentioned section.

We have successfully managed to run an ethereum node in Proof of Work(PoW), Proof of Stake(PoS) and Proof of Authority(PoA) consensus mechanisms. This gives us an idea and clarity on different consensus mechanisms, their setup and implementation.

3 Q3.4 - Blacklist Addresses

3.1 Introduction

We are going to use the PoW geth version(v1.9.25) that was used in the Section 2.1. We would need to make some changes in the implementation of geth in order to implement the blacklist to avoid transactions from.

3.2 Changes in Geth

The file named *transaction_signing.go* in the path */go-ethereum/core/types/* needs to be modified and added with our blacklist implementation.

3.2.1 New Error

We create a new *Error* object after the import block in the file. This error object can be used to flag the error that the sender is blacklisted and invalid.

```

var (
    ErrInvalidChainId      = errors.New("invalid chain id for signer")
    ErrBlacklistedAddress = errors.New("Adress Blacklisted, Malicious Sender")
)
```

Figure 13: Error for Blacklisted addresses

3.3 Blacklist Functionality

Now we create a new array named *blacklist* and add a new function *addressBlacklist* that will return the boolean value *True* if the address is blacklisted and *False* if absent.

```
var blacklist = []common.Address{
    common.HexToAddress("f5a2f640992744ca1312bfb13e7c566ab663c1e2"),
    common.HexToAddress("592bea02739197d861571174a133fe2fd78d440c"),
}

func addressBlacklist(address common.Address) bool {
    for _, blacklistedAddress := range blacklist {
        if blacklistedAddress == address {
            return true
        }
    }
    return false
}
```

Figure 14: Blacklist Function

3.3.1 Implementation

In order for our node to avoid transactions from the addresses present in *blacklist* array, we need to make sure that we check the sender address before we accept and execute a transaction. We modify the code in function *func Sender(signer Signer, tx *Transaction) (common.Address, error)* on the same file. We a *if* statement and we use the *addressBlacklist* function to check if our sender address is present in the blacklist. We return the address and error object that was created in the above sub-section 3.2.1. If not blacklisted we proceed with the transaction.

```
func Sender(signer Signer, tx *Transaction) (common.Address, error) {
    if sc := tx.from.Load(); sc != nil {
        sigCache := sc.(sigCache)
        // If the signer used to derive from in a previous
        // call is not the same as used current, invalidate
        // the cache.
        if sigCache.signer.Equal(signer) {
            if addressBlacklist(sigCache.from) {
                return sigCache.from, ErrBlacklistedAddress
            }
            return sigCache.from, nil
        }
    }

    addr, err := signer.Sender(tx)
    if err != nil {
        return common.Address{}, err
    }
    tx.from.Store(sigCache{signer: signer, from: addr})
    return addr, nil
}
```

Figure 15: Checking if blacklisted

3.3.2 Build

Now we can build our modified version of geth and start our modified geth node by using the steps used in 2.1. Start the geth node and attach a JS console to the geth node in a new terminal.

3.4 Working

We will now try transacting with valid and invalid sender and see the results of our implementation.

3.4.1 Valid Sender

```
> personal.unlockAccount("65a16a338ce89f519e9afdfffd93744758f14dec")
Unlock account 65a16a338ce89f519e9afdfffd93744758f14dec
Passphrase:
true
> web3.eth.sendTransaction({
    from: "65a16a338ce89f519e9afdfffd93744758f14dec",
    to: "c138c3cf7a50ebf1e39287dcfdc8c92db3c97071",
    value: web3.toWei(1, "ether")
})
"0xcc489ce34141cdd01cd80f5e44b044d24dd10a0c6eee1392440da01d681f777f"
```

Figure 16: Transaction from valid sender

3.4.2 Invalid Sender

```
> web3.eth.sendTransaction({
    from: "f5a2f640992744ca1312bfb13e7c566ab663c1e2",
    to: "0x3d9c5cf9f17b4a6fd911e950921d4133c0c54ddd",
    value: web3.toWei(1, "ether")
})
Error: invalid sender
        at web3.js:6347:37(47)
        at web3.js:5081:62(37)
        at <eval>:1:25(16)
```

Figure 17: Transaction from invalid sender

4 Q3.5 - Proof of Stake implementation

4.1 Introduction

Historically, geth was sufficient to run an entire ethereum node including the consensus mechanism i.e. Proof of Work and the execution engine together. But after the merge, you need geth, the execution client, and a consensus client to run the proof of stake consensus algorithm.

4.1.1 Execution client

An execution client is a node that runs the software that is tasked with processing and broadcasting transactions and managing Ethereum's state. They run the computations for each transaction using the Ethereum Virtual Machine to ensure that the rules of the protocol are followed. These nodes running the execution clients together form the execution layer. There are four execution clients that are being used currently such as *geth*, *Besu*, *Nethermind*, and *Erigon*.

4.1.2 Consensus client

These are nodes with the consensus client such as, *Lighthouse*, *Nimbus*, *Prysm*, *Teku*, and *Lodestar*. These clients can be used to facilitate the Proof of Stake consensus for your network and must be connected with your execution client.

4.2 Connecting geth with prysm

4.2.1 Theory

In this section, we are going to discuss the theory behind implementing PoS with geth. As we have seen before, geth is an execution client. Geth handles state management, and other EVM and protocol tasks as mentioned in sub-section 4.1.1. In order to achieve consensus our execution client needs to be connected with a consensus client. Prysm is ethereum PoS consensus client written in Go. You can run a beacon node and can also stake 32ETH to run a validator node and gain rewards.

In order to connect your execution client(geth) and your consensus client(prysm), we need to generate a *JWT Secret* (Refer Section 2.2.2 & Figure 10). JSON Web Tokens(JWT) are an open, industry standard RFC 7519 method for representing claims securely between two parties. After successfully generating your jwt secret, now you can start your execution by mentioning a port number to engage in RPC connections. You can use the generated jwt secret to authenticate the RPC connection between your clients.

All consensus clients expose a Beacon API that can be used to query information and status of the client and blocks. You can use a public checkpoints to sync your chain faster and sync your geth node.

4.2.2 Practical Implementation

For this part, we are going to see how to connect an execution client(geth) with a consensus client(prysm) and successfully run an etehreum node. PLEASE REFER TO SECTION 2.2. HERE WE HAVE DEMONSTRATED ON RUNNING AN EXECUTION CLIENT AND CONNECT IT WITH A CONSENSUS CLIENT.