Setting up a private blockchain network — Documentation

Project Title: Setting up a private blockchain network
Date: 29th of December 2023
Author(s): Soorya Kuppusamy (s0585852) & Jérémie Djaoud-Deshayes (s0590692)

Table of Contents

A list of all sections and subsections:

- 1. Project Overview
- 2. Glossary
- 3. Purpose of the project and the documentation
- 4. Scope of the documentation
- 5. System requirements
- 6. Technical specifications
- 7. Blockchain architecture
- 8. Steps to configure a multiple-node private blockchain network
- 9. Interaction between nodes
- 10. Relation between private and public blockchains in Ethereum; potential attack scenarios
- 11. Is a private chain really isolated from public chains?
- 12. Conclusion
- 13. Sources, references and useful links

---

1. Project Overview

This documentation will guide you through how to run your own private blockchain ("peer-to-peer") network.
A private network is composed of multiple Ethereum **nodes** that can only connect to each other. Each node can have multiple **accounts** running inside of them.
The nodes must also know about each other and be able to exchange information, share an initial state and a common consensus algorithm.

---

2. Glossary

- **blockchain** - an information system that stores data, secures this data from falsification and loss, transfers and processes data inside the system while preserving data reliability; the protection of data is achieved by: (1) writing the data into a chain of cryptographically linked blocks, (2) decentralized storage of copies of the chains of blocks on nodes of a peer-to-peer network, and (3) synchronization of chains of blocks on all full nodes using a consensus algorithm; preserving data reliability when performing operations with data within the network is ensured by storing the algorithms of data transfer and processing (contracts) inside the blockchain; the chain of blocks itself is also referred to as the blockchain.
- **peer to peer network** - a computer network, consisting of equally privileged nodes (without a central server).
- **private network** - refers to a permissioned blockchain that operates within a closed ecosystem, where access and participation are restricted to a specific group of known entities. Unlike public blockchains, where anyone can join and participate, a private blockchain is designed for a defined set of participants who have been granted specific permissions to engage in the network.
- **EVM** (Ethereum Virtual Machine) - a decentralized, runtime environment that executes smart contracts on the Ethereum blockchain. It serves as the runtime execution model for Ethereum, enabling the processing of smart contract code by all participants in the network. The EVM is responsible for ensuring consistency in the execution of smart contracts across all nodes in the Ethereum network, providing a secure and decentralized platform for the deployment and execution of **Dapps**
- **block**- a collection of transactions grouped by a validating node into a special data structure after their format and signatures are verified; a block contains a hash pointer as a link to the previous block, which is one of the measures that ensure the cryptographic security of the blockchain; a block is added to the blockchain after the consensus with other validating nodes on the network is achieved.
- **genesis block** - the initial block or the first block in the chain. It serves as the foundation from which the entire blockchain is built. The genesis block contains essential information, including the initial configuration, parameters, and often a predefined set of transactions that establish the initial state of the blockchain.
- **hash** - a uniquely reproducible cryptographic representation of a file or any other set of digital data; it ensures the invariability of data – any modification of data changes its hash.
- **block validation**- verification of the correctness of a block's structure, its creation time, its compatibility with the previous block, the transaction signatures, and the correspondence of transactions to the data in the blockchain.
- **validating node** - a node on the network that has the right to create and validate blocks.
- **bootnode** - a specialized node that plays a crucial role in facilitating network discovery. It acts as a bootstrap or entry point for other nodes to join the network. The bootnode helps nodes find and connect to each other by providing them with information about the network's topology. It maintains a list of active nodes and their IP addresses, enabling new nodes to quickly discover and establish connections with the existing network.
- **consensus** - an agreement between validating nodes on the procedure of adding new blocks to the blockchain or an algorithm of such agreement.
- **transaction** - a single operation of data transfer on a blockchain network, or a record of such transaction in the blockchain.
- **token** - a unit that represents a share of rights, fixed as a set of identifiable numerical records in the register that includes a mechanism for exchanging shares of rights between these records.
- **identification** - a cryptographic procedure for recognition of a user in the system.
- **unique identification** - a procedure that links a user with a unique person; it requires legal and organizational efforts to implement biometric or other procedures for association of usernames with real persons.
- **private key** - a string of characters kept in secret by its owner and used for accessing this person's Virtual Account on the network and signing transactions.
- **public key** - a string of characters which can be used to check the authenticity of a signature made with a private key; public keys can be uniquely derived from private keys, but not vice versa.
- **digital signature** - an attribute of a digital document or message, obtained as a result of cryptographic data processing; a digital signature is used to check the integrity (absence of modifications) and authenticity of a document (verify sender's identity).
- **smart contract** - a program that performs operations with data stored in the blockchain; all contracts are stored in the blockchain.
- **Dapp** (short for **decentralized application**) - a software application that runs on a decentralized network using blockchain technology. It operates without a central authority, utilizing smart contracts to automate and enforce rules.
- **transaction fee (gas)** - payment to a validating node for execution of a transaction.
- **double spend** - an attack on a blockchain network with the purpose of using the same tokens for two different transactions; this attack is implemented by forming and maintaining a fork (alternate version) of the blockchain; such an attack can be executed only in the event the attacker controls 50% or more of the network's validating power.
- **encryption** - transformation of digital data in such a way that only the party that possesses the appropriate decryption key is able to read it.

- **private blockchain** - a blockchain network where all nodes and access rights to data are under the centralized control of a single organization (government, corporation, or private individual).
- **public blockchain** - a blockchain network which is not controlled by any organization, all decisions are made by reaching a consensus among the network's participants, and data is available to everyone for read access.
- **delegated proof of stake (DPoS)** - a blockchain network consensus algorithm, where validating nodes are assigned by delegates (usually, token owners), who vote using their shares of rights.
- **proof of authority (PoA)** - a blockchain network consensus algorithm, where solving arbitrarily difficult mathematical problems isn't at the core, but uses a set of "authorities" nodes that are explicitly allowed to create new blocks and secure the chain. It is mostly used for private chains setup, as it is not particularly useful at larger scales, and loses the point of pure "decentralization", as the chain has to be signed off by the majority of authorities, therefore keeping the block issuers accountable.
- **IPC (Inter-Process Communication)** - a mechanism that allows different processes on the same machine to communicate with each other.
- **RPC (Remote Procedure Call)** - a protocol that allows a program or process to execute code (procedures) on another address space, typically on a different machine or node in a network. It enables communication and coordination between different processes or systems by allowing one program to initiate the execution of code in another, as if it were a local procedure call.

---

3. Purpose of the project and the documentation

- It can be useful *"as a backend for core developers"*
- It can also be useful *"for Dapp developers testing multi-block and multi-user scenarios".*
- Finally, running transactions / smart contracts on Ethereum (or any public blockchain network) costs fees (which we will refer to as gas going forward); for testing purposes when developing smart contracts or testing networking, these fees can add up quite quickly, so it makes sense to test your smart contracts on a private network before deploying them.

---

4. Scope of the documentation

1. Install the Ethereum client on each machine (Go-Ethereum - will be referred to as `geth` going forward)
2. Create the nodes folders in each location (remote server and local machines)
3. Create account(s) in each node folders
4. Create the Genesis block in `JSON` format
5. Initialize each node with the Genesis block created above
6. Start the bootnode
7. Start each node in the network
8. Verify that the whole network is up, running and interconnected:
    1. Check whether all nodes are synced to the current network state
    2. Try to execute transactions between multiple node's accounts
    3. Check that the balance of both interacting nodes have been successfully updated

---

5. System Requirements

Hardware Requirements

From the official go-ethereum documentation:

> *"The hardware requirements for running a Geth node depend upon the node configuration and can change over time as upgrades to the network are implemented. Ethereum nodes can be run on low power, resource-constrained devices such as Raspberry Pi's. Prebuilt, dedicated staking machines are available from several companies - these might be good choices for users who want plug-and-play hardware specifically designed for Ethereum. However, many users will choose to run nodes on laptop or desktop computers."*

- Processor: *"It is preferable to use a **quad-core** (or dual-core hyperthreaded) CPU."*
- Memory: *"It is recommended to use at least **16GB RAM**."*
- Disk space:
    - *To run a full node running on the official Ethereum mainnet, it is required to use a >2TB SSD, as at the time of writing, "Geth itself requires >650GB of disk space for a snap-synced full node, [while] a full archive node that keeps all state back to genesis requires more than 12TB of space."*
    - However, for the scale of our project, considering the low difficulty required to seal new blocks and the low amount of transactions expected to happen on our network, we ran `geth` on a machine with **30GB of SSD storage** for experimenting.
- Bandwidth: *"It is recommended to have at least **25Mbps** download speed to run a node. Running a node also requires a lot of data to be uploaded and downloaded so it is better to use an ISP that does not have a capped data allowance."*

---

6. Technical Specifications

- We required a remote server running on HTW's network, to which we can connect via SSH to. Here are it's specifications:

    - CPU: 4 cores
    - RAM: 8GB
    - Storage: 30GB of SSD
    - Operating system: Ubuntu 20.04.6 LTS
    - We disabled the firewall rules on this server to avoid any potential issues when accessing it from our local machines.

    However, it stays available only from the HTW gateway, so we need to be connacted via the HTW's VPN to be able to SSH into it.

Its IP address is the following: `141.45.212.243` and it's server "name" is `private-blockchain`.

- The technical infrastructure will be as follows
    - We set up **one node per machine** (one on the remote server (the **signer node** - `node1`), one on Soorya's local machine (running Windows 10 - `node2`) and one on Jérémie's local machine (running Linux Mint 21.3 - Ubuntu-based) - `node3`)

- Each of these nodes will contain a single **account**, for simplicity. Therefore, we will have the following configuration:
  - `node1` is the one on the remote server. This will be our **signer node**.
  - `node2` is the one on Soorya's local machine
  - `node3` is the one on Jérémie's local machine

---

7. Blockchain Architecture

This part will provide an overview of our chosen blockchain architecture:

- **Blockchain Protocol: Ethereum**

> *Our private blockchain relies on the Ethereum protocol. It serves as the backbone for creating a decentralized peer-to-peer environment for managing digital assets and executing smart contracts.*

The Ethereum blockchain is a **distributed ledger that records transactions across a network of computers.** This ledger, maintained by participants known as nodes, ensures transparency and immutability.

One of its strength lies in smart contracts, self-executing code that runs on the **Ethereum Virtual Machine (EVM)**. These contracts enable automated and trustless agreements, revolutionizing various industries, including but not limited to decentralized finance and NFTs.

Users submit transactions to the Ethereum network. Nodes validate and process these transactions, grouping them into blocks, which are sequentially added to the blockchain.

Ethereum's core is to maintain a decentralized approach, distributing authority across a network of nodes. Trust is established through cryptographic principles (including the consensus algorithm), ensuring the integrity of transactions and smart contract execution.

- **Consensus Mechanism: Proof of Authority - *PoA***

In a PoA system, a **predefined set of nodes**, known as **authorities**, are granted the power to create new blocks and validate transactions.
These authorities are typically identified entities with a level of trust within the network.
Unlike other consensus algorithms like Proof-of-Work (PoW) or Proof-of-Stake (PoS), **PoA doesn't rely on complex mathematical puzzles or the amount of cryptocurrency held**, but rather on the **identity and authority of the participants.**

Here's a simplified breakdown of how PoA works:

- Selection of Authorities: A limited number of trusted entities are chosen as authorities. These entities are responsible for creating new blocks and validating transactions.
- Transaction Validation: When a participant initiates a transaction, the authorities validate it based on predefined rules. If the transaction meets the criteria, it is added to a block.
- Block Creation: Authorities take turns proposing blocks of transactions. The consensus is reached when a majority of authorities agree on the validity of the transactions in a proposed block.
- Block Confirmation: Once a block is agreed upon, it is added to the blockchain. The consensus mechanism ensures that all nodes in the network have a consistent view of the transaction history.
- Rewards and Penalties: In some PoA systems, authorities may be rewarded for their role in block creation, or there may be penalties for malicious behavior. This encourages good behavior and discourages attacks on the network.

> PoA is known for its efficiency and lower energy consumption compared to PoW, making it a suitable choice for private or consortium blockchains where a level of trust among participants is established.

- **Node Types in a PoA network**:
  In blockchain technology, there are multiple types of nodes that serve various purposes.
  As defined previously, *"**nodes** communicate with each other through a **peer-to-peer** network, allowing them to exchange information while maintaining **consensus** on the state of the **blockchain**"*. We won't go through every single different possible type of node, as it would deviate from the initial subject, but will go through the most important and relevant in regard to our project, that is following a Proof-of-Authority consensus mechanism:

  1. **Authority Nodes** (Sealers/Signers):
     - **Sealing Blocks:** Authority nodes are responsible for creating and sealing new blocks in the chain. Unlike public PoW (Proof-of-Work) networks where miners compete to solve complex mathematical challenges, PoA networks have a fixed set of trusted nodes that take turns creating new blocks. Each authority node gets a chance to create a block in a deterministic order.
     - **Transaction Validation:** Authority nodes validate and include transactions in the blocks they create. They have the authority to decide which transactions are valid and should be added to the blockchain.
     - **Node Permissions:** Authority nodes are granted special permissions to participate in the consensus algorithm. In a real-life scenario, they are usually selected based on their trustworthiness and are expected to maintain the integrity of the network.
  2. **Regular Nodes** (Non-Sealers/Non-Signers):
     - **Transaction Propagation:** Regular nodes are responsible for propagating transactions across the network. While they don't participate in the block creation process, they play a crucial role in broadcasting transactions to the authority nodes.
     - **Block Verification:** Regular nodes verify the validity of the blocks created by authority nodes. They ensure that the transactions within the blocks adhere to the network's rules and consensus algorithm.
     - **Decentralization:** Regular nodes contribute to the decentralization of the network by participating in the validation process, even though they do not have the authority to create new blocks.

---

8. Steps to configure a multiple-node private blockchain network:

> *Note: the following instructions are for an Unix-based system.*

> *Refer to the official geth documentation for other operating systems.*

1. Install the `geth` client on each machine (see detailed instructions here):

- Add the `ethereum` repository:
  `sudo add-apt-repository -y ppa:ethereum/ethereum`
- Then, update your local package index and install the stable version of `go-ethereum`:

```
sudo apt-get update
sudo apt-get install ethereum
```

2. Create the node folders on both remote and local machines:

- Create a separate folder to isolate the installation; for example, on our local server:
  `mkdir private-blockchain`
- Get into that directory:
  `cd private-blockchain`
- From there, create a data directory per node:
  `mkdir node1`

> *Here, we created the `node1` folder in the `private-blockchain` directory on the remote server, while the `node2` folder is on Soorya's local machine and the `node3` folder is on Jérémie's local machine.*

3. Create accounts in each of the nodes' folders:

The next step is to create one (or multiple) accounts in each node data directory (referred to as *'datadirs'* going forward, as this is how they're referred to by `geth`.):

> *For simplicity, we'll just create 1 account per node, but we could create much more.*

- To create an account for all the three nodes, run the following commands from the `nodeX` directory:
  - On the remote server: `geth --datadir "./data" account new`

```
local@private-blockchain:~/blockchain$ cd node1
local@private-blockchain:~/blockchain/node1$ geth --datadir "./data" account new
INFO [01-28|19:31:31.119] Maximum peer count                       ETH=50 LES=0 total=50
INFO [01-28|19:31:31.121] Smartcard socket not found, disabling    err="stat /run/pcscd/pcscd.comm: no such file or directory"
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:   0x1669A246063AA439CD5A9C071C2F5d6b4F1e0cb0
Path of the secret key file: data/keystore/UTC--2024-01-28T19-31-35.574384543Z--1669a246063aa439cd5a9c071c2f5d6b4f1e0cb0
```

  - On Soorya's machine: `geth --datadir "./data" account new`

```
PS D:\FinalProjectPROTD\final-demo-ethereum\demo> cd node2
PS D:\FinalProjectPROTD\final-demo-ethereum\demo\node2> geth --datadir "./data" account new
INFO [01-28|20:34:27.617] 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:   0x739ae6402E4b12Eda3dA0439CcEac110F90CAcdc
Path of the secret key file: data\keystore\UTC--2024-01-28T19-34-32.016951700Z--739ae6402e4b12eda3da0439cceac110f90cacdc

- You can share your public address with anyone. Others need it to interact with you.
```

  - On Jérémie's machine: `geth --datadir "./data" account new`

```
jerem@T14-Gen2i:~/Desktop/node3$ geth --datadir "./data" account new
INFO [01-28|20:26:39.149] Maximum peer count                       ETH=50 total=50
INFO [01-28|20:26:39.151] Smartcard socket not found, disabling    err="stat /run/pcscd/pcscd.comm: no such file or
 directory"
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:   0xad4502669E98ed44EcdAA8EE1a630036cD6E93d0
Path of the secret key file: data/keystore/UTC--2024-01-28T19-26-43.336225192Z--ad4502669e98ed44ecdaa8ee1a630036cd6
e93d0

- You can share your public address with anyone. Others need it to interact with you.
```

These commands will request for a password (and a confirmation of it).
Write it down in a text file in the `nodeX` directory (in this example, we'll use `pwd.txt`).

> *For demonstration purposes, we will use the same password - `1234` - for all three accounts, but of course, in a real-life scenario, you should use a secure password / passphrase.*

- Write down the public addresses of the generated accounts (starting with `0x...`); you will need them later.
  For this example, we'll store all the relevant info in `info.txt`, located in the root folder.
  Here are our public addresses:

- `node1` public address: **0x1669A246063AA439CD5A9C071C2F5d6b4F1e0cb0**
- `node2` public address: **0x739ae6402E4b12Eda3dA0439CcEac110F90CAcdc**
- `node3` public address: **0xad4502669E98ed44EcdAA8EE1a630036cD6E93d0**

> *These are in hexadecimal format (i.e. contain only numbers and letters from {a-A} to {f-F}).*

4. Create the Genesis block in `JSON` format and configure it with appropriate parameters:

> *For this example, we chose the **Proof-of-Authority** consensus algorithm, as it's the more suitable for a small and private peer-to-peer network ("Clique is strongly recommended for private testnets because PoA is far less resource-intensive than PoW"), but we won't describe why in detail here.*
> *Refer to the official geth documentation to learn more about consensus algorithms.*

- Create a `genesis.json` file in the root directory, with the following structure (that we'll describe later:)

```
{
    "config": {
      "chainId": 123456,
      "homesteadBlock": 0,
      "eip150Block": 0,
      "eip155Block": 0,
      "eip158Block": 0,
      "byzantiumBlock": 0,
      "constantinopleBlock": 0,
      "petersburgBlock": 0,
      "istanbulBlock": 0,
      "berlinBlock": 0,
      "clique": {
        "period": 0,
        "epoch": 30000
      }
    },
    "difficulty": "0",
    "gasLimit": "15000000",
    "extradata": "0x0000000000000000000000000000000000000000000000000000000000000000001669A246063AA439CD5A9C071C2F5d6b4F1e0cb0000000000
    "alloc": {
      "1669A246063AA439CD5A9C071C2F5d6b4F1e0cb0": { "balance": "9000000000000000000" },
      "739ae6402E4b12Eda3dA0439CcEac110F90CAcdc": { "balance": "9000000000000000000" },
      "ad4502669E98ed44EcdAA8EE1a630036cD6E93d0": { "balance": "9000000000000000000" }
    }
}
```

The `config` section ensures that *"all known protocol changes are available and configures the 'Clique' engine to be used for consensus."*

> *Note that the initial signer set must be configured through the `extradata` field. This field is required for the Clique engine to work appropriately; we will describe it later down the road.*

- `chainId` is the ID of the network. The Ethereum's mainnet `chainId` is 1.

  > *"There are also many other networks that Geth can connect to by providing alternative Chain IDs, some are testnets and others are alternative networks built from forks of the Geth source code. Providing a network ID that is not already being used by an existing network or testnet means the nodes using that network ID can only connect to each other, creating a private network. A list of current network IDs is available at Chainlist.org."*

Source: official Geth documentation.

Reason: To prevent **replay attacks** between the Ethereum and Ethereum Classic networks.

> *Chain ID was introduced in EIP-155 or Ethereum Improvement Proposal 155 standard to prevent replay attacks between the Ethereum chain (ETH) and the Ethereum Classic chain (ETC) which both have a network ID of 1. Chain ID is required when signing transactions, meaning transactions signed on the ETH network end up with a different hash than those signed on ETC. Before EIP- 155, signed transactions on each network would look the same, and could be replayed.*

Here, we set it to `123456` for demonstration, after checking that it is not used by any other existing chain.

- The lines after that (the ones ending with `Block: 0`) correspond to various Ethereum hard forks, and they indicate at which block number each specific upgrade should take effect.
  In the context of a private blockchain network, these lines are not strictly necessary, especially since we are not planning to include these specific Ethereum network upgrades in our private chain. However, as we follow the official Geth documentation, we still included them to follow good practices for potential future changes and scalability of the project.

- `clique` subsection:

> As mentioned, we're using the Proof-of-Authority consensus mechanism (using the 'Clique' engine).
> In this system, new blocks can only be created by authorized 'signers' or sealer nodes.
>
> The initial set of authorized signers is configured under `extradata` flag. Signers can be authorized and de-authorized using a voting mechanism, thus allowing the set of signers to change while the blockchain operates.

> Sealers take turns proposing new blocks in a 'round-robin' fashion or through another predetermined algorithm. This rotation ensures fairness and prevents any single sealer from having undue influence for an extended period. The selected sealer creates a block containing transactions, signs it with their private key, and broadcasts it to the network.
>
> The protocol defines a voting mechanism to dynamically add new signers and remove existing ones. In Geth, this can be done via the `clique.propose(address, authorized)` method.
>
> To authorize a new signer, existing ones can propose it by executing `clique.propose("0x...",true)` from the node's console. When more than half the signers proposed it, the authorization comes into effect immediately and the newly authorized signer account can start signing blocks.
>
> Similarly, existing signers can propose deauthorizing existing ones by executing `clique.propose("0x...", false)`, revoking an authorized signer.
> Again, if more than half, specifically 51%, of the signers deauthorize a signer, it is immediately removed from the list, and blocks minted by it are rejected from that point onward.

- `period`: this configuration option *"sets the target block time of the chain"*.
  In order to avoid empty block creation in a Proof-of-Authority mechanism, setting it to `0` means that **new blocks will only be created whenever there is a new transaction**, committing new sealing ('mining' equivalent) work whenever a transaction happens.
- `epoch`: this is used to *"define the number of blocks after which the validator set is recalculated"*.

  > In this example, it means that after every 30000 blocks, a new set of validators will be chosen.
- `difficulty`: this field is assigned to a value of `2` when the block is sealed by the in-order signer, referred to as `DIFF_INTURN`. Alternatively, it is set to a value of `1` when the block is sealed by an out-of-order signer, referred to as `DIFF_NOTURN`.
  The need for an out-of-order signer arises in scenarios where the regular in-order signer is unavailable or encounters issues. For example, if the in-order signer is down for any reason, a block would be missed after the fixed period of `BLOCK_PERIOD`. In such cases, the remaining out-of-order signers will introduce a random delay of (`SIGNER_COUNT` * 500) ms, and the first out-of-order signer to surpass this delay will then seal the next block. This mechanism ensures continuity in block generation even when the in-order signer is temporarily unavailable.

Source: StackExchange and EIP-225

- `gasLimit` is the initial block gas limit.

  > *This impacts how much EVM computation can happen within a single block. Mirroring the main Ethereum network is generally a good choice.*

However, here we set it to **15,000,000** (15 million) to approximately mirror half of the current Ethereum mainnet's gas limit.

- `extradata`: this is one of the most **crucial fields** for our Genesis block.
  It allows for the Clique consensus protocol to identify the initial set of authorized signer nodes and their order.

It is a hexadecimal data field that will contain information about the signer nodes (**responsible for validating and creating new blocks**).
Here, we will **encode the signer node (node1) account address**.
To do so: 1) We need to **concatenate `0x` followed by 64 `zero` characters.** 2) Then **include all signer account addresses** (*excluding the leading* `0x`).
For this example, we'll only include `node1`'s account address, as it is our **signer node**. 3) Then, **concatenate an additional 128 `zero` characters.
The result will look like this:

```
"extradata": "0x0000000000000000000000000000000000000000000000000000000000000001669A246063AA439CD5A9C071C2F5d6b4F1e0cb0000000000000
```

- `alloc`: is the initial allocation of `wei` per account.

  > *This determines how much ether (an ether is equal to 10^{18} wei) should be allocated to the addresses listed in the genesis block. Additional ether can be created through mining as the chain progresses.*

Here, we assigned a `balance` of **9.000.000.000.000.000.000** wei (9,000 billion, corresponding to **9 ether**) to our three accounts addresses **(without the leading `0x`'s!** so we have plenty of room to play with later on.

5. Initialize each node with the Genesis block:

Now that we have created our genesis block (`genesis.json`), we need to store a perfectly similar copy of this genesis file on each of the nodes' root folder.
Then, we will initiate our three nodes using `geth init` as follows:

- On the remote server: `geth init --datadir "./data" genesis.json`
- On Soorya's local machine: `geth init --datadir "./data" genesis.json`
- On Jérémie's local machine: `geth init --datadir "./data" genesis.json`

Here's an example of the output of this command (for `node3`):

```
jerem@T14-Gen2i:~/Desktop/node3$ geth init --datadir "./data" genesis.json
INFO [01-28|23:41:25.766] Maximum peer count                       ETH=50 total=50
INFO [01-28|23:41:25.767] Smartcard socket not found, disabling    err="stat /run/pcscd/pcscd.comm: no such file or directory"
INFO [01-28|23:41:25.770] Set global gas cap                       cap=50,000,000
INFO [01-28|23:41:25.770] Initializing the KZG library             backend=gokzg
INFO [01-28|23:41:25.810] Using pebble as the backing database
INFO [01-28|23:41:25.810] Allocated cache and file handles         database=/home/jerem/Desktop/node3/data/geth/chaindata cache=16.00MiB handles=16
INFO [01-28|23:41:25.817] Opened ancient database                  database=/home/jerem/Desktop/node3/data/geth/chaindata/ancient/chain readonly=false
INFO [01-28|23:41:25.817] State scheme set to already existing     scheme=hash
INFO [01-28|23:41:25.817] Successfully wrote genesis state         database=chaindata hash=13b4cf..7ba608
INFO [01-28|23:41:25.817] Using pebble as the backing database
INFO [01-28|23:41:25.817] Allocated cache and file handles         database=/home/jerem/Desktop/node3/data/geth/lightchaindata cache=16.00MiB handles=16
INFO [01-28|23:41:25.825] Opened ancient database                  database=/home/jerem/Desktop/node3/data/geth/lightchaindata/ancient/chain readonly=false
INFO [01-28|23:41:25.825] State scheme set to already existing     scheme=hash
INFO [01-28|23:41:25.825] Successfully wrote genesis state         database=lightchaindata hash=13b4cf..7ba608
```

The last line of the output should be similar to this:

```
INFO [01-28|20:46:26.413] Successfully wrote genesis state         database=lightchaindata hash=13b4cf..7ba608
```

where the exact same hash values (`hash=13b4cf..7ba608`) should match for each node, meaning that the initialization has been successful with the exact same genesis block for all nodes, as it's like a “cryptographic confirmation” that the Genesis block is exactly the same for each node initiated.

6. Start the bootnode:

1. To configure a **bootnode**, we require a **key**. We will use the remote server to host this bootnode.
   This key can be created using the following command, that will save the keyfile to `boot.key` (or whatever name you want to give it):
   `bootnode -genkey boot.key`
2. Then, this key can be used to 'generate' a bootnode as follows:
   `bootnode -nodekey boot.key -addr "127.0.0.1:30301" -verbosity 7`

- *The `nodekey` flag should point to the previously created keyfile.*
- *The choice of port passed to the `addr` flag is arbitrary, but public Ethereum networks use 30303, so it's best to avoid it for good practices.*
- *The `verbosity` flag determines how much information about the node's activity and processes is displayed in the console or logs. We set it to `7` to have a comprehensive and detailed log output.*

The `bootnode` command, if executed successfully, will output an `enode`, which, in our scenario, is the following:

```
enode://93c69d7147503d4d38d0dc427cbd60e4b67d29e1fe6b6fed081e1944feff94413da2f14750817eefa9cae3a347db8261b670352fb340c15007a5ae9bc897
```

The output should look like this:

```
$ bootnode -nodekey boot.key -addr "127.0.0.1:30301" -verbosity 7
enode://93c69d7147503d4d38d0dc427cbd60e4b67d29e1fe6b6fed081e1944feff94413da2f14750817eefa9cae3a347db8261b670352fb340c15007a5ae9bc89734c1@127.0.0.1
:0?discport=30301
Note: you're using cmd/bootnode, a developer tool.
We recommend using a regular node as bootstrap node for production deployments.
INFO [01-28|22:53:50.022] New local node record                    seq=1,706,482,430,021 id=b325bb95ce13d5f0 ip=<nil> udp=0 tcp=0
```

DO NOT CLOSE THIS TERMINAL WINDOW! We'll need to use it later.
We add this `enode` value to our `info.txt` file, as we will also use it later on.

The root folder structure of our remote server should now look like this:

- `boot.key` : bootnode's keyfile
- `genesis.json` : Genesis block file
- `node1` directory:
  - `pwd.txt` : password file
  - `data` directory:
    - `geth` directory: contains all the chain's data.(we won't describe the whole structure of this directory, as it's not relevant)
    - `geth.ipc` : **IPC endpoint** used by the Geth client
    - `keystore` directory:
      - `UTC--2023-12-13T19-32-44.598426613Z--91fd9b466df5a24e4435d0820b91d785f7b6963f` : private key file of `node1`

7. Start each node in the network:

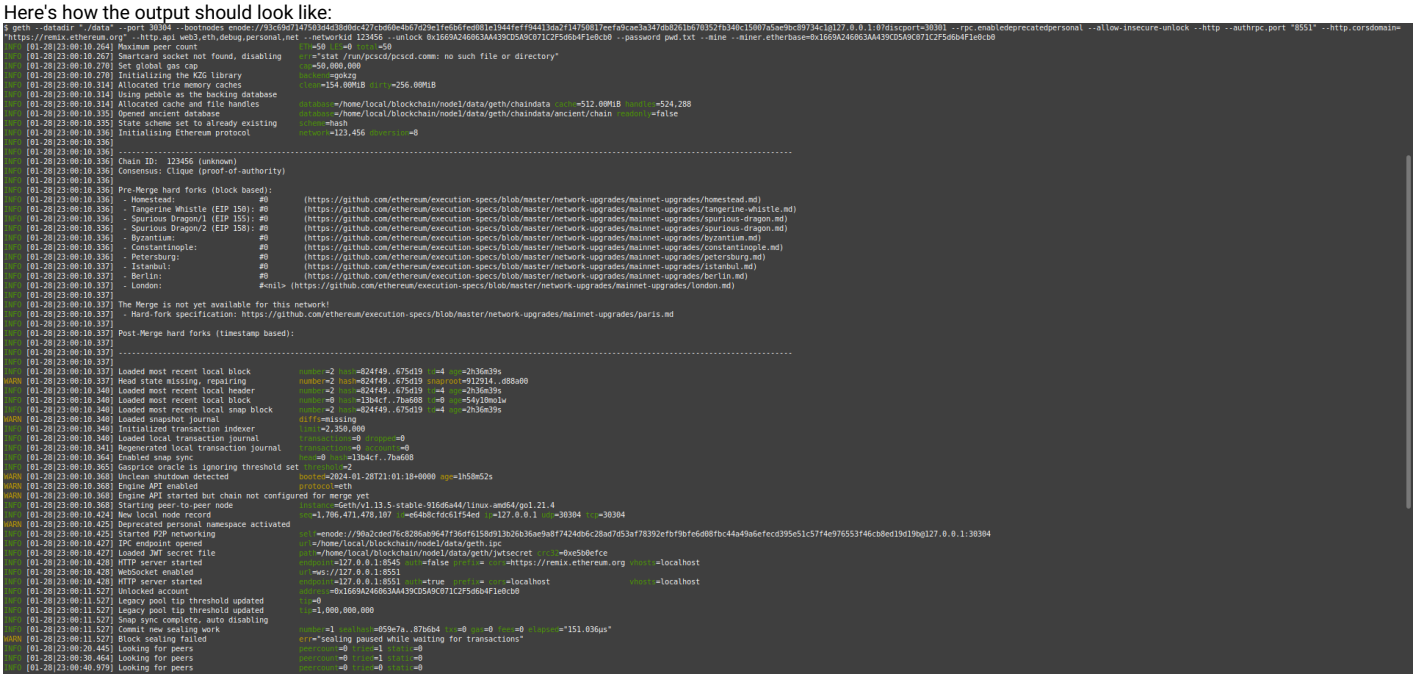
- Open separate terminals for each node in the root directory **while leaving the bootnode running in the original terminal!**
  - In the `node1` terminal window (from the `node1` directory), run the following command:

```
geth --datadir {NODE1_DATADIR}
--port {NODE1_PORT}
--bootnodes {BOOTNODE_ENODE}
--networkid {NETWORK_ID}
--allow-insecure-unlock
--unlock {NODE1_PUBLIC_ADDRESS}
--password {NODE1_PASSWORD_FILE}
--authrpc.port "{NODE1_AUTHRPC_PORT}"
--rpc.enabledeprecatedpersonal
--http
--http.corsdomain="https://remix.ethereum.org"
--http.api web3,eth,debug,personal,net
--mine
--miner.etherbase={SIGNER_NODE_PUBLIC_ADDRESS}
```

In our case, that should be:

```
geth --datadir "./data"
--port 30304
--bootnodes enode://93c69d7147503d4d38d0dc427cbd60e4b67d29e1fe6b6fed081e1944feff94413da2f14750817eefa9cae3a347db8261b670352fb340c150
--networkid 123456
--allow-insecure-unlock
--unlock 0x1669A246063AA439CD5A9C071C2F5d6b4F1e0cb0
--password pwd.txt
--authrpc.port "8551"
--rpc.enabledeprecatedpersonal
--http
--http.corsdomain="https://remix.ethereum.org"
--http.api web3,eth,debug,personal,net
--mine
--miner.etherbase=0x1669A246063AA439CD5A9C071C2F5d6b4F1e0cb0
```

Here's how the output should look like:



After executing this step, when checking again the bootnode's terminal window, we should see logs and debugs message:

```
DEBUG[01-28|23:01:38.656] Revalidated node                         b=15 id=e64b8cfdc61f54ed checks=1
TRACE[01-28|23:01:39.071] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:39.072] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:39.572] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:39.572] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:40.072] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:40.073] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:40.573] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:40.573] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:41.074] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:41.074] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:41.574] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:41.574] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:42.075] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:42.075] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:42.575] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:42.575] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:43.076] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:43.076] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:43.391] << PING/v4                               id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:43.391] >> PONG/v4                               id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:43.391] << ENRREQUEST/v4                         id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:43.391] >> ENRRESPONSE/v4                        id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:43.577] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:43.577] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:44.077] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:44.077] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:44.577] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:44.578] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:45.078] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:45.078] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:45.579] << FINDNODE/v4                           id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:45.580] >> NEIGHBORS/v4                          id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:45.724] >> PING/v4                               id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:45.724] << PONG/v4                               id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:45.724] >> ENRREQUEST/v4                         id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
TRACE[01-28|23:01:45.725] << ENRRESPONSE/v4                        id=e64b8cfdc61f54ed addr=127.0.0.1:30304 err=nil
```

- In the `node2` terminal window (from the `node2` directory), run the following command:

```
geth --datadir {NODE2_DATADIR}
--port {NODE2_PORT}
--bootnodes {NODE1_ENODE}
--networkid {NETWORK_ID}
--allow-insecure-unlock
--unlock {NODE2_PUBLIC_ADDRESS}
--password {NODE2_PASSWORD_FILE}
--authrpc.port "{NODE2_AUTHRPC_PORT}"
```

In our case, that should be:

```
geth --datadir "./data"
--port 30309
--bootnodes enode://90a2cded76c8286ab9647f36df6158d913b26b36ae9a8f7424db6c28ad7d53af78392efbf9bfe6d08fbc44a49a6efecd395e51c57f4e9765
--networkid 123456
--allow-insecure-unlock
--unlock 0x739ae6402E4b12Eda3dA0439CcEac110F90CAcdc
--password pwd.txt
--authrpc.port "8567"
```

- Finally, in the `node3` terminal window (from the `node3` directory), run the following command:

```
geth --datadir {NODE3_DATADIR}
--port {NODE3_PORT}
--bootnodes {NODE1_ENODE}
--networkid {NETWORK_ID}
--allow-insecure-unlock
--unlock {NODE3_PUBLIC_ADDRESS}
--password {NODE3_PASSWORD_FILE}
--authrpc.port "{NODE3_AUTHRPC_PORT}"
```

In our case, that should be:

```
geth --datadir "./data"
--port 30307
--bootnodes enode://90a2cded76c8286ab9647f36df6158d913b26b36ae9a8f7424db6c28ad7d53af78392efbf9bfe6d08fbc44a49a6efecd395e51c57f4e9765
--networkid 123456
--allow-insecure-unlock
--unlock 0xad4502669E98ed44EcdAA8EE1a630036cD6E93d0
--password pwd.txt
--authrpc.port "8549"
```

A few things to take into consideration while running these commands:

- The order of the flags passed does not matter, however we keep them structured in the same order for good practice.
- The port used for both `node1`, `node2` and `node3` should be different.
- `node1` (our signer node) should connect to the bootnode's enode, while `node2` and `node3` should connect to `node1`'s enode.
- The `allow-insecure-unlock` flag allows the node to unlock an account without encryption.
- The `rpc.enabledeprecatedpersonal` flag enables the deprecated personal RPC API. The personal API provides functionalities related to managing accounts, signing transactions, and interacting with the Ethereum wallet.
- The `authrpc.port` used by each of the three nodes should be different. We arbitrarily used, respectively, `8551`, `8567` and `8549` in our scenario. This `authrpc.port` flag (**RPC: Remote Procedure Call**) serves as the interface allowing other nodes or applications to interact with that node.
- The `http` flag enables the HTTP-RPC server, allowing remote clients to interact with the Geth node over HTTP.
- The `http.corsdomain="https://remix.ethereum.org"` flag specifies the **CORS** (Cross-Origin Resource Sharing) domain for the HTTP-RPC server. It restricts which domains are allowed to access the Geth node's API using web browsers.
- The `http.api web3,eth,debug,personal,net` specifies the API modules to enable for the HTTP-RPC server. In this case, it enables the `web3`, `eth`, `debug`, `personal`, and `net` modules.
- The `mine` flag instructs the Geth node to start mining new blocks.
- The `miner.etherbase` flag specifies the Ethereum address to which mining rewards will be sent (coinbase address).

Now, we can check the information about `node1` by opening the JavaScript console:

```
cd data
geth attach geth.ipc
```

From the console, we should run `admin.nodeInfo` which should return the following:

```
> admin.nodeInfo
{
  enode: "enode://90a2cded76c8286ab9647f36df6158d913b26b36ae9a8f7424db6c28ad7d53af78392efbf9bfe6d08fbc44a49a6efecd395e51c57f4e976553f46cb8ed19d19b@127.0.0.1:30304",
  enr: "enr:-KO4QAJiMChha93PM65H5N8782pnpvsopl-CcV4JFCfIR5vgQAArZkYPRS9WvErUJ7z1oaBhJZld773NEW2OVvbBWM6GAY1RoFNag2V0aMfGhGFHYVSAgmlkgnY0gmlwhH8AAAGJc2VjcDI1NmsxoQOQos3t
dsgoarlkfzbfYVjZE7JrNq6aj3Qk22worX1Tr4RzbmFwwIN0Y3CCdmCDdWRwgnZg",
  id: "e64b8cfdc61f54ed978fefd931a823ff6796420392e8f1a2c3d7c2499807cf03",
  ip: "127.0.0.1",
  listenAddr: "0.0.0.0:30304",
  name: "Geth/v1.13.5-stable-916d6a44/linux-amd64/go1.21.4",
  ports: {
    discovery: 30304,
    listener: 30304
  },
  protocols: {
    eth: {
      config: {
        berlinBlock: 0,
        byzantiumBlock: 0,
        chainId: 123456,
        clique: {...},
        constantinopleBlock: 0,
        eip150Block: 0,
        eip155Block: 0,
        eip158Block: 0,
        homesteadBlock: 0,
        istanbulBlock: 0,
        petersburgBlock: 0
      },
      difficulty: 2,
      genesis: "0x13b4cf8e7766f7cbbaf629fe9d9614ae098cab8f65e07ce47efefece5f7ba608",
      head: "0xa03fc043a1373b5102bfd4d1c3cdfd7804e0de7ac914af7d54b89b9496506fda",
      network: 123456
    },
    snap: {}
  }
}
```

We can also get information about peers from the console by running `admin.peers`:

```
$ geth attach geth.ipc
WARN [01-28|23:10:56.312] Enabling deprecated personal namespace
Welcome to the Geth JavaScript console!

instance: Geth/v1.13.5-stable-916d6a44/linux-amd64/go1.21.4
coinbase: 0x1669a246063aa439cd5a9c071c2f5d6b4f1e0cb0
at block: 0 (Thu Jan 01 1970 00:00:00 GMT+0000 (UTC))
 datadir: /home/local/blockchain/node1/data
 modules: admin:1.0 clique:1.0 debug:1.0 engine:1.0 eth:1.0 miner:1.0 net:1.0 personal:1.0 rpc:1.0 txpool:1.0 web3:1.0

To exit, press ctrl-d or type exit
> admin.peers
[{
    caps: ["eth/67", "eth/68", "snap/1"],
    enode: "enode://6f3fa226602970416c47aaed5c119a5e5060bc156bb906f683f8bc7371372624f43d6f7d3a5e4f362cdd03bd6c4d85fa7298355f2c1de497ad94fd0bfe11ba54@141.45.242.57:30307",
    id: "c5e689039bdde01dc0e0af0e6e0b1ca4dd70ebff255968e229d779291872e9c4",
    name: "Geth/v1.13.11-stable-8f7eb9cc/linux-amd64/go1.21.6",
    network: {
      inbound: false,
      localAddress: "141.45.212.243:38134",
      remoteAddress: "141.45.242.57:30307",
      static: false,
      trusted: false
    },
    protocols: {
      eth: {
        version: 68
      },
      snap: {
        version: 1
      }
    }
}]
```

Once the network is all successfully up and running, we should see in any of the nodes console that `peercount=2` :

```
INFO [01-29|12:52:24.078] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:52:34.097] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:52:44.118] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:52:54.137] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:53:04.159] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:53:14.179] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:53:24.197] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:53:34.218] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:53:44.237] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:53:54.259] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:54:04.280] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:54:14.300] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:54:24.321] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:54:34.340] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:54:44.362] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:54:54.383] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:55:04.403] Looking for peers                        peercount=2 tried=0 static=0
INFO [01-29|12:55:14.423] Looking for peers                        peercount=2 tried=0 static=0
```

If a peer is not connecting automatically, we can add it manually by using the `admin.addPeer(enode://)` method.

In our scenario, to add `node3` as a peer if it's not connecting automatically, we can use the following command to manually add `node3` as a peer from its `enode` :

```
admin.addPeer(enode://6f3fa226602970416c47aaed5c119a5e5060bc156bb906f683f8bc7371372624f43d6f7d3a5e4f362cdd03bd6c4d85fa7298355f2c1de4
```

A possible error is **"Block sealing failed"**:

```
INFO [01-28|23:29:04.931] Looking for peers                        peercount=0 tried=1 static=0
INFO [01-28|23:29:08.618] Block synchronisation started
INFO [01-28|23:29:08.619] Mining aborted due to sync
WARN [01-28|23:29:08.746] Synchronisation failed, dropping peer    peer=c5e689039bdde01dc0e0af0e6e0b1ca4dd70ebff255968e229d779291872e9c4 err="peer is stalling"
INFO [01-28|23:29:08.746] Commit new sealing work                  number=1 sealhash=a64dd3..529a74 txs=0 gas=0 fees=0 elapsed="165.679µs"
WARN [01-28|23:29:08.746] Block sealing failed                     err="sealing paused while waiting for transactions"
```

On a Clique (Proof-of-Authority) network, this error might occur when the signer nodes are not up. Remember that in a PoA network, **at least 51% of the signer nodes must be up and running**, and they should also be connected to the network.

We can find the list of the signers currently in the network by running `clique.getSigners()`, which should return the list of the addresses of the signer nodes:

```
> clique.getSigners()
["0x1669a246063aa439cd5a9c071c2f5d6b4f1e0cb0"]
```

We have to ensure that the signer nodes are indeed mining, which is by using the following flags when starting the nodes:

```
--mine
--miner.etherbase={SIGNER_NODE_PUBLIC_ADDRESS}
```

If both of these arguments haven't been provided when starting the node, then they must be restarted with these two additional arguments.

Lastly, we need to verify a couple of things on the network is up and running:

- Check whether all nodes are synced to the current state:
  To do that, we need to execute the following commands from the JavaScript console:
  `eth.syncing` : should return `false` and `eth.blockNumber` should return the number of the most recent sealed block (in our current state: `2`). If both of

these conditions are met, it most likely means that nodes have completed syncing.
![ethSyncing ethBlockNumber](https://i.imgur.com/urRrjDW.png)

---

9. Interaction between nodes

To check if our nodes interoperate, we can try to execute transactions from one node's account to another by sending ether between `node2` and `node3` for example.

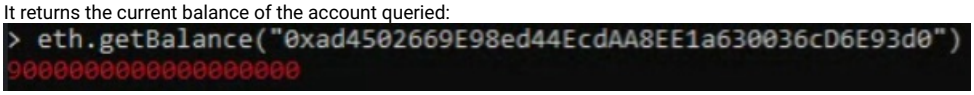
1. Let's first check the balance of `node2` and `node3`; to do so, let's first open the console in each node's folder, by running the following command: `geth attach geth.ipc`
2. To check the balance of the account where you're executing the command from, you can run any of the following commands, referring to itself by `eth.accounts[0]` or `eth.coinbase`:

```
eth.getBalance(eth.accounts[0])
eth.getBalance(eth.coinbase)
```

To check the balance of `node3`, we can also refer to its public address (when running from an external account, such as from the `node2`'s console):

```
eth.getBalance("0xad4502669E98ed44EcdAA8EE1a630036cD6E93d0")
```

It returns the current balance of the account queried:



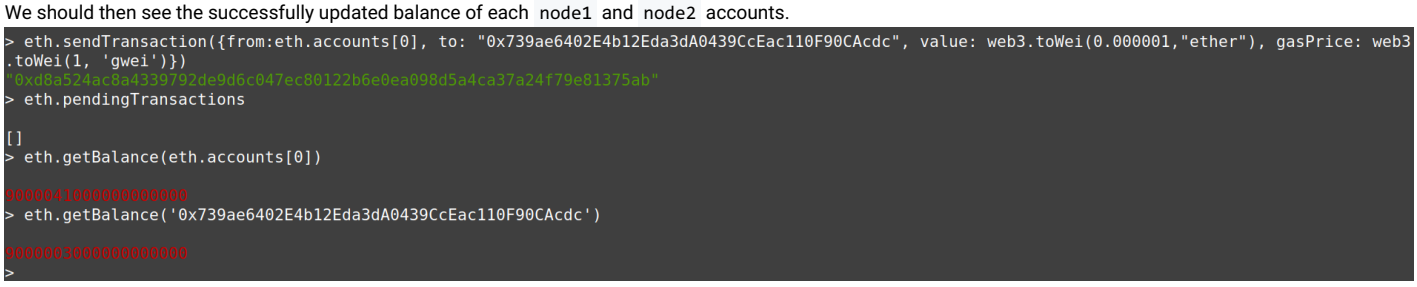
3. Then, to actually initiate a transaction between two accounts, we need to execute the following command (from the `node1` JavaScript console):

```
// send some wei
eth.sendTransaction({
  from: eth.accounts[0],
  to: '{0x_NODE2_ADDRESS}',
   // refers to the account the command it is executed from - node2 account in that case
  value: {{VALUE}}, // value of wei / ether to be sent
  gasPrice: web3.toWei(1, 'gwei')// gas price
});

// check if the transaction was successful by querying node1 and node2's account balances:
eth.getBalance('{0x_NODE1_ADDRESS}');
eth.getBalance('{0x_NODE2_ADDRESS}');
```

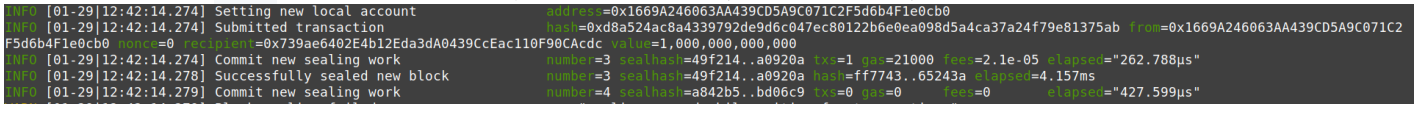
In our scenario:

```
eth.sendTransaction({
  from: eth.accounts[0], // node1 only account
  to: '0x739ae6402E4b12Eda3dA0439CcEac110F90CAcdc', // node2 public address
  value: web3.toWei(0.000001,"ether"), // 0.000001 ether
  gasPrice: web3.toWei(1, 'gwei')
});

eth.getBalance('0x1669A246063AA439CD5A9C071C2F5d6b4F1e0cb0');
eth.getBalance('0x739ae6402E4b12Eda3dA0439CcEac110F90CAcdc');
```

We should then see the successfully updated balance of each `node1` and `node2` accounts.



Here is what happens in `node1`'s console when successfully initiating a transaction (a new block is sealed):



We can verify that it is successful by running `eth.blockNumber` again; it should be increased:

```
> eth.blockNumber
3
```

10. Relation between private and public blockchains in Ethereum; potential attack scenarios

- **Relation between private and public blockchains in Ethereum:**

Knowing the difference between block chains plays a huge role for companies seeking the perfect type blockchain for their use-cases.
Without that knowledge, a company may end up selecting the wrong platform, and the solution may not work for their needs and wants.

This section explains the differences between a public and private blockchain in terms of data access, authority, transactions, consensus mechanisms, data handling, and immutability.

	Public blockchain	Private blockchain
Access	In a public network, **anyone can access the platform** through a P2P network. Each node in a public blockchain has equal rights to access, send, and receive transactions, information, and agreements without any restrictions.	In a private network, **only authority-verified nodes can participate**. Nodes in a private blockchain do not have equal rights to access, send, and receive transactions, information, and agreements.
Authority	A **decentralized system** has no single entity controlling the network. This absence of central control helps to reduce additional transaction costs such as transaction fees. Instead, it enhances security by eliminating central authorities like banks, large organizations, or other entities.	A **partially decentralized system**, allowing restricted public access. Only users verified by the central authority can enter the network and must obtain additional permissions to run a node, publish and execute smart contracts, or access and create data in the network.
Transaction speed	**Slow**. A large number of nodes participate in the network, causing requests to take a significant amount of time to process. Because of that, **transaction fees** are typically higher than those in private blockchains, that can be configured according to specific use-cases. For instance, the Ethereum public network can only process **approximately 30 transactions per second** (TPS).	**Fast**. Private block chains are much smaller than public ones, allowing them to execute transactions much faster. As a result, **their transaction fees are usually extremely low**.
Consensus	Nodes **don't have any restrictions** in joining the consensus protocol, which means it is **permission-less** (anyone can access and run a node). Nodes that participate can remain anonymous.	A private blockchain decides beforehand who can join the consensus and who cannot. **The access to the network and their rights are restricted to specific nodes**. User identities are shared among others participants in the network.
Data handling	**Anyone can read and write on the ledger**, but once any information is broadcasted, it cannot be altered.	**Only a single organization has the privilege to read and write on the ledger**. Moreover, a limited number of nodes possess the authority to write on the ledger, and in certain cases, they could even delete a block.
Efficiency	**Less efficient** compared to private blockchains, it can encounter scalability issues when more nodes participate, leading to a slowdown in the processing speed of transaction requests, especially when multiple requests are made by participating nodes.	**High efficiency.** On the other hand, a private blockchain usually operates with only a handful of nodes on the network, ensuring consistent efficiency regardless of the circumstances.
Immutability	The public blockchain network is **fully immutable**. What does it mean? It implies that once a block is added to the chain, there is theoretically no way to change or delete it. This ensures that no one can manipulate a specific block to gain undue advantages from others.	In contrast, a private blockchain has **partial immutability**, allowing the authority to overwrite a transaction or information at their discretion.

- **Potential attack scenarios:**
    - In a public blockchain:

1. **The potential for centralization in the network**:
   In a public blockchain network, users follow an algorithm that verifies transactions by committing software and hardware resources, solving cryptographic puzzles through brute-force. The first user to find the solution is rewarded, and each new solution, along with the verified transactions, forms the foundation for the next puzzle. Due to the high computational power required, currently, over half of the network's hashing power is concentrated in a single country — China, due to lower electricity prices. This concentration may lead to network centralization, collusion, and vulnerability to changes in policy.
2. **Theft of cryptocurrencies**:
   With the increasing value of cryptocurrencies, cybercriminals are showing a growing interest in stealing them. The code of blockchain is still new, and it may be susceptible to currently unknown security vulnerabilities.
3. **Double-spending attack**:
   A serious threat to blockchain transactions, where the attacker successfully initiates multiple transactions using a single coin, invalidating the 'honest' transaction.
4. **Time-jacking attack**:
   In this scenario, the attacker announces an inaccurate timestamp while connecting to a node for a transaction. The network time counter of the node

is manipulated, and the deceived node may accept an "alternate" blockchain.
5. **Sybil attack**:
   Hackers may leverage blockchain cryptographic algorithms and mechanisms to carry out malicious activities without leaving any traces.
6. **Selfish Mining:**
   Refers to an attack strategy where a group of colluding miners exploits vulnerabilities to manipulate the blockchain, gaining disproportionate revenue. In this attack scenario, colluding miners force honest miners to waste computational power on a stale public branch while secretly creating a fork. When revealed, 'honest' miners switch to the new blocks, ultimately benefiting the selfish miners.

- In a private blockchain:

1. **Replay attack**:
   A replay attack between the Ethereum and Ethereum Classic (*a fork of the Ethereum network that didn't undergo the mainnet's hard forks*) networks refers to a situation where a transaction initiated on one blockchain (Ethereum or Ethereum Classic) is maliciously replicated or re-executed on the other blockchain. Given the similarities in their transaction formats and cryptographic underpinnings, there is a risk that a valid transaction on one network could be unintentionally or maliciously repeated on the other.
   Chain ID was introduced in **EIP-155** (Ethereum Improvement Proposal 155) standard to prevent replay attacks between the Ethereum chain (ETH) and the Ethereum Classic chain (ETC) which both have a network ID of 1. Chain ID is now required to sign any transaction, meaning that transactions signed on the ETH network end up with a different hash than those signed on ETC.

2. **"51% attack"**:
   In a proof-of-authority network, where 51% of sealers must be operational and connected, managing uncommunicative or intermittently active nodes is a crucial security concern. While nodes may go offline for benign reasons, the network requires a robust structure that can operate smoothly in their absence and promptly reintegrate them upon return. This ensures the network's overall resilience and responsiveness, allowing it to adapt to occasional node unavailability without compromising functionality.

> *A public blockchain is ideal for businesses that desire user freedom, allowing them to read and write with complete transparency. All users operate on a level playing field in a public blockchain and can choose to remain anonymous. On the other hand, a private blockchain is more suitable for businesses seeking control over users on the network and the activities within it.*

---

11. Is a private chain really isolated from public chains?

- **One possible test: Creating a chain with a used number.**

> *"Converting a Chain ID associated with a public blockchain into a private blockchain context is not a straightforward process. Public blockchains operate on open consensus mechanisms (like Proof of Work or Proof of Stake), and their security relies on a large number of decentralized nodes. Private blockchains, on the other hand, may use different consensus models and have a more controlled membership. Transitioning from a public to a private blockchain involves fundamental changes to the blockchain's architecture and consensus mechanisms. It may require redefining access controls, modifying consensus protocols, and adjusting the overall design to suit the requirements of a private network. In summary, while the statement is generally true, the conversion process involves more than just adapting the Chain ID. It requires a thorough reconfiguration of the blockchain to align with the characteristics and goals of a private blockchain network."*

Source: ChatGPT

> *Note: for the following test, we have to **disable our Internet connection.***

We tried to create a private blockchain with a `chainId` of `1` (the same as the Ethereum mainnet). Here is the Genesis block configuration:

```
{
  "config": {
    "chainId": 1,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "petersburgBlock": 0,
    "istanbulBlock": 0,
    "berlinBlock": 0,
    "clique": {
      "period": 0,
      "epoch": 30000
    }
  },
  "difficulty": "0",
  "gasLimit": "15000000",
  "extradata": "0x0000000000000000000000000000000000000000000000000000000000000000bfEb5986622D10AB07Fd9bef81782be5Fc5da2680000000000
  "alloc": {
    "bfEb5986622D10AB07Fd9bef81782be5Fc5da268": { "balance": "9000000000000000000" },
    "700f375c64693F89A7807FC5d3138876CB52f96d": { "balance": "9000000000000000000" }
  }
}
```

We were able to create a private blockchain network isolated from the public blockchain; here is the evidence:

```
haindata\\ancient/chain" readonly=false
INFO [01-14|13:51:36.461] State scheme set to already existing     scheme=hash
INFO [01-14|13:51:36.473] Initialising Ethereum protocol           network=1 dbversion=<nil>
INFO [01-14|13:51:36.548]
INFO [01-14|13:51:36.549] ------------------------------------------------------------------------------------------------------------------------
----------------------------------------
INFO [01-14|13:51:36.553] Chain ID:  1 (mainnet)
INFO [01-14|13:51:36.555] Consensus: Clique (proof-of-authority)
INFO [01-14|13:51:36.557]
INFO [01-14|13:51:36.560] Pre-Merge hard forks (block based):
INFO [01-14|13:51:36.562]  - Homestead:                   #0        (https://github.com/ethereum/execution-specs/blob/master/network-upgrad
es/mainnet-upgrades/homestead.md)
INFO [01-14|13:51:36.562]  - Tangerine Whistle (EIP 150): #0        (https://github.com/ethereum/execution-specs/blob/master/network-upgrad
es/mainnet-upgrades/tangerine-whistle.md)
INFO [01-14|13:51:36.564]  - Spurious Dragon/1 (EIP 155): #0        (https://github.com/ethereum/execution-specs/blob/master/network-upgrad
es/mainnet-upgrades/spurious-dragon.md)
INFO [01-14|13:51:36.565]  - Spurious Dragon/2 (EIP 158): #0        (https://github.com/ethereum/execution-specs/blob/master/network-upgrad
es/mainnet-upgrades/spurious-dragon.md)
INFO [01-14|13:51:36.566]  - Byzantium:                   #0        (https://github.com/ethereum/execution-specs/blob/master/network-upgrad
```

```
> admin.nodeInfo
{
  enode: "enode://90ea9f777b38fb8d955e9a7d1fe562ecb9af2ea62cbbee9892b78a60efd0c0e7062056d8250f1e392cff1fb9bd8d5bb1555520865d806d2e991938959
657655d@127.0.0.1:30304",
  enr: "enr:-KO4QEfGzQvMHxrahT-9qB9908Xv_XVxrHnH4d6_W9Oz74gxKAnlfD7kbtppJ0DPS1tmNnYvbUG1OFhPevnusGiVncSGAY0IBxHDg2V0aMfGhBqYmE6AgmlkgnY0gml
whH8AAAGJc2VjcDI1NmsxoQOQ6p93ezj7jZVemn0f5WLsua8upiy77piSt4pg79DA54RzbmFwwIN0Y3CCdmCDdWRwgnZg",
  id: "7cb11635725b2745cc08c14bc074b2c5f34d9112b5a8a2e43e2189a9f0a0772f",
  ip: "127.0.0.1",
  listenAddr: "[::]:30304",
  name: "Geth/v1.13.3-stable-0d45d72d/windows-amd64/go1.21.1",
  ports: {
    discovery: 30304,
    listener: 30304
  },
  protocols: {
    eth: {
      config: {
        berlinBlock: 0,
        byzantiumBlock: 0,
        chainId: 1,
        clique: {...},
        constantinopleBlock: 0,
        eip150Block: 0,
        eip155Block: 0,
        eip158Block: 0,
        homesteadBlock: 0,
        istanbulBlock: 0,
```

```
Ln 6, Col 189 (188 selected)  Spaces: 4  UTF-8  CRLF  Plain T
```

Here are screenshots demonstrating the successful addition of a peer node to the private chain with a `chainId` of `1` and a successful transaction between nodes:

```
> admin.peers
[{
    caps: ["eth/67", "eth/68", "snap/1"],
    enode: "enode://aa3bc95b9f37c0e81d294613b361def242fe4d9ae9a779c6b1b148578a2bd1d2287decdf3123b2a58359a796c3809a6ec11deb1de2c155bbbcd9
3c7556877@127.0.0.1:30308",
    id: "bce2fc81b4e110290c84517db789a21d6efea4edaa6be74d7bdd539b2b31b2a9",
    name: "Geth/v1.13.3-stable-0d45d72d/windows-amd64/go1.21.1",
    network: {
      inbound: false,
      localAddress: "127.0.0.1:63322",
      remoteAddress: "127.0.0.1:30308",
      static: false,
      trusted: false
    },
    protocols: {
      eth: {
        version: 68
      },
      snap: {
        version: 1
      }
    }
}]
> eth.getBalance("0xbfEb5986622D10AB07Fd9bef81782be5Fc5da268")
9000000000000000000
> eth.getBalance("0x700f375c64693F89A7807FC5d3138876CB52f96d")
9000000000000000000
```

```
> eth.getBalance("0xbfEb5986622D10AB07Fd9bef81782be5Fc5da268")
900000000000000000
> eth.getBalance("0x700f375c64693F89A7807FC5d3138876CB52f96d")
900000000000000000
> eth.sendTransaction({from:"0xbfEb5986622D10AB07Fd9bef81782be5Fc5da268", to:"0x700f375c64693F89A7807FC5d3138876CB52f96d", value:web3.toWei
(0.000001,"ether"),gasPrice:web3.toWei(1, 'gwei')})
"0x53ad9bd06629058bdc811cd77dd75c6d0bcc19f895d4812b618e7e0eaeb686e9"
> eth.getBalance("0x700f375c64693F89A7807FC5d3138876CB52f96d")

900000100000000000
> eth.getBalance("0xbfEb5986622D10AB07Fd9bef81782be5Fc5da268")

899999900000000000
>
>
```

So, as a conclusion, **it is indeed possible to create an "isolated" private blockchain** with the same `chainId` as the Ethereum mainnet (`1`), **if and only if it is completely offline** (isolated).

However, if we try to do the same **while being connected to the Internet, it is not possible**, as it will connect to the actual Ethereum mainnet instead of connecting to our private chain just created.

---

12. Conclusion

- In this project, we set up a fully operational blockchain network running three different nodes on separate machines:
    - `node1` is running on the HTW's server - it was our initial **signer node**, as we defined it in the Genesis block.
      This node was also running our `bootnode` , as the "starting point" of the chain, to which `node1` is connecting to.
    - `node2` is running on Soorya's personal laptop
    - `node3` is running on Jérémie's personal laptop
- Each of these nodes ran 1 account, but we could have created much more accounts on each of them.
- We used the Proof-of-Authority consensus algorithm as after researching, we realized that it was the most suitable for setting up such an infrastructure; indeed, in this system, a pre-defined set of nodes (in our case, `node1`) are known as **authorities**, which are, at the initialization of the network, the only ones that have the "power" to create new blocks and validate transactions (which is a correlated process, as new blocks are created only when transactions are happening), and that it doesn't require heavy computational power unlike in a Proof-of-Work network.
- We were able to execute transactions among each account (sending Ethereum from one account to another), showing that the network is fully functional.
- We were also able to add / remove **signer nodes** by running `clique` "proposals" from the node's console.
- We compared the pros and cons of private blockchains VS public ones, describing their differences and use-cases, as well as the risks that one could face compared to the other.
- We also checked if a private chain was actually isolated from public chains by running some tests, and realized it was indeed completely isolated as long as it is run on a local network.
- For potential and future improvements, we could tweak our infrastructure to mirror as closely as possible the Ethereum network or according to our wants / needs.
  We could also add multiple other nodes, each of them containing multiple accounts, and create scripts so that anyone at HTW could easily run their own node connecting to this private chain running on HTW's network.
  Finally, we could develop Dapps running smart contracts to give "purpose" to this private blockchain (voting system, lotteries...) rather than just exchanging virtual coins with each other.
  We could also potentially develop an "academic record management" at HTW for an immutable system for storing, updating and checking on students marks, as a further, more complex example of a Dapp.

Sources, references & useful links

- Official Geth documentation
- Private Networks (Geth)
- Ethereum Development Documentation
- What are the Types of Nodes in Blockchain?
- Everything You Need to Know About Validator Nodes: A Deep Dive
- All Major Blockchain Consensus Algorithms Explained
- Peer-to-peer (Wikipedia)
- How to build an Ethereum private blockchain using Geth
- General Blockchain Technology Terms
- Proof-of-Authority Chains - Wikl (OpenEthereum Documentation)
- Public VS Private Blockchain: Key Differences
- Security risks with public and private blockchains (page removed since then for some reason)
- EIP-155
- ChatGPT