DA51 Lab session 2

Ethereum
The Geth Client

J. Gaber, gaber@utbm.fr

Target set of this session:
After completion of this session, you will successfully know:

- how to set up a private network of multiple Geth nodes
- how o use Ethereum's Clique Proof of Authority (PoA) consensus algorithm
- how to utilize the Geth console to access the various management APIs
- how to create new Ethereum accounts
- how to invoke an Ether transfer transaction.

Set up a private Blockchain With Geth

Geth is an Ethereum client application written in the Go programming language. It allows to set up and run an Ethereum node in a network. The node can run in one of the publicly available Ethereum networks or you can set up a private one.

You should have Geth installed, otherwise you need first a package manager to download the Geth binaries from the Ethereum repository. You can install the Go implementation of Ethereum using a variety of ways. I would recommend using Brew to install Geth (whatever is your OS, for Mac, or WSL, or Linux). To use Brew to install Geth, go to brew.sh and follow install instructions. The alternative is to reach the document at the URL https://geth.ethereum.org/docs/install-and-build/installing-geth and follow the instructions.

1. once Geth has been successfully installed on your system, either using Brew or an alternative way according to your OS, you can confirm that the installation has been successful by running the Geth command `geth --help`.

2. create a new directory, call it Lab-session-2: `mkdir lab-session-2`

3. move to your new directory: `cd lab-sesion-2`

4. To set up a simple private Ethereum network consisting of two nodes, running on the local machine, create two data directories for two nodes node1 and node 2 (i.e., `mkdir node1 node 2`)

5. Each node will have an associated account. Create an account for node1 using the command `geth --datadir node1 account new`. Follow the prompts to set a password and note the public address of the key. Did the command create a keyfile? where this file is stored?

6. Repeat the same step for Node 2 by replacing node1 with node2.

7. To create your own private Ethereum network, you need first to create your own genesis block. We define the genesis block in a `genesis.json` file. With your favorite text editor,

create the file genesis.json with for example the one provided on moodle (you can find one online).

8. Among the fields which are defined within this JSON file, we have the config field (with the “chainId” argument), the alloc field, where you can allocate Ether balance to various accounts on this private Ethereum network, the “gaslimit” field which sets a limit of the gas cost per block.

It is worth noting that the account addresses in the alloc field should be replaced with those created for each node in the previous step (without the leading 0x).

9. Selecting a Consensus Algorithm and creating Signer Account Keys

While the main network uses proof-of-stake (PoS) to secure the blockchain, Geth also supports the the 'Clique' proof-of-authority (PoA) consensus algorithm. Clique is strongly recommended for private testnets because PoA is far less resource intensive than PoW.

In a Proof of Authority (PoA) consensus algorithm, the term "mining" is not typically used in the same way as it is in Proof of Work (PoW) systems like Bitcoin. Instead of miners competing to solve complex mathematical problems to create new blocks, in PoA networks, blocks are created or "minted" by a set of pre-approved authority nodes (validators).

These validators are chosen based on reputation and are given the right to create blocks and validate transactions. They take turns to create new blocks in a round-robin fashion or based on a predefined schedule, and there is no competition or computational puzzle-solving involved in block creation.

In Ethereum's Clique Proof of Authority (PoA) consensus algorithm, the extradata field in the genesis block is used to specify the initial list of validators (signers) who are allowed to mint new blocks. The extradata field has a specific structure and requires careful formatting.

Here’s a basic outline of how to structure the extradata field for Clique PoA:
extradata: '0x' + [32 bytes vanity] + [Concatenated addresses of initial signers] + [65 bytes for proposer signature]

- 32 bytes of vanity data: This is usually just 32 bytes of zeros, and it's used for any extra data you want to include, like a description or identifier for your network.
- Concatenated addresses of initial signers: This is where you list the addresses of the initial validators in the network, concatenated together without any delimiters.
- 65 bytes for proposer signature: This is usually just 65 bytes of zeros in the genesis block.

The extradata field must be exactly the right length, so be careful with the number of zeros in the vanity and proposer signature sections. After setting up the network, any changes to the validator set are usually handled through a voting mechanism by the existing validators, and not by modifying the extradata field directly.

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 Clique to work.

The keys for the signer account can be created utilizing the geth account command, and this command can be executed multiple times to generate multiple signer keys

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

It is crucial to note down the Ethereum address that this command outputs. To incorporate the signer addresses in extradata, concatenate 32 bytes of zeros, all the signer addresses, followed by an additional 65 bytes of zeros. The resulting concatenated string is then assigned as the value for the extradata key in the genesis.json file. In the example, extradata includes a single initial signer address, 0x21DBE2dc90a7366a26A53850e27e93A8fbBC9C6E.

Once the definition of the genesis block of your private Ethereum blockchain is created, the next step is to initialize our blockchain using this genesis block and create a database where all the data for the blockchain will be stored.

10. This database will include information about the various accounts in the network, among other details. To do so, use the command:
    ```
    geth init --datadir node1 genesis.json
    ```

the datadir parameter specify the subdirectory within your current working directory in which this database will be created. As feedback when you run this Geth command, you get the notification that the genesis block has now been successfully initialized and that a database has also been created out of it.

11. Explore the contents of this directory (via file expolorer tool or via command line). Within this node1 directory, there are two subdirectories. One is called Geth and the other is called keystore. This database represents all the data which is needed for an Ethereum node. And within the Geth directory, you can see not only the full chain data in the chaindata directory (in case you wish to run this as a light node, then the data will be stored in the lightchaindata directory). The keystore directory could be empty if you have not configured any accounts to map onto this Ethereum node. Once you create an account for this node, then the keystore files for that account will be stored over here.

So far, you created your first private blockchain network and you initialized it using a genesis. block. you also defined two nodes, which includes a database to store the chain data, as well as account information.

Once the nodes have been configured and initialized, establishing a peer-to-peer network is the ensuing step. This necessitates a bootstrap node, a standard node assigned as the entry point for other nodes to connect to the network. The choice of which node to designate as the bootstrap node is flexible, as any node can ulfill this role.

In this lab, the developer tool bootnode is employed to efficiently configure a dedicated bootnode with ease. Initially, a key is required for the bootnode, which can be created using the command below, saving a key to boot.key: `bootnode -genkey boot.key`

12. Subsequently, this key can be utilized to generate a bootnode with the following command: `bootnode -nodekey boot.key -addr :30305`. The port selection for -addr is discretionary; however, it is advisable to avoid using 30303 as it is commonly used by

public Ethereum networks. Check the bootnode command output as logs in the terminal, to confirm that it is running (to be used next).

13. You can now initiate the two nodes. Launch separate terminals for each node while keeping the bootnode operational in the initial terminal. Execute the command below in each terminal, substituting node1 with node2 where necessary, and assigning unique IDs to --port and authrpc.port for each node. Additionally, the account address and password file for node 1 need to be specified:

```
geth --datadir node1 --port 30307 --bootnodes
enode://3249c006405694e334bc3fe6ff785f0a26cd805b774a8ad76f4721991a946793
7441115b915ef720282fc826cfca87a31b2e865efcaafe1539d3d0ff9042ef5b@127.0.0.
1:0?discport=30305  --networkid 1234567890 --unlock
21DBE2dc90a7366a26A53850e27e93A8fbBC9C6E --password node1/password.txt
--authrpc.port 8551
```

Executing this will initiate the node, utilizing the bootnode as an entry point.

14. Replicate the same command, incorporating the relevant information for node 2. The logs in each terminal should signify a successful operation. In the first terminal, which is presently active, the logs will display the ongoing discovery process.

15. You can now connect a Javascript console to any of the nodes to inquire about the network properties using the following command: `geth attach node1/geth.ipc`

16. After initiating the Javascript console, verify that the node has established a connection with the other node (node 2) by running: `net.peerCount`

17. You can also retrieve the details of this connected peer to confirm that it is indeed Node 2 by using: `admin.peers`

18. The account linked to Node 1 should have been credited with some ether at the genesis of the chain. This can be easily verified using eth.getBalance():
    `eth.getBalance(eth.accounts[0])`

19. Subsequently, this account can be unlocked, and some ether can be transferred to Node 2 with the commands below:
    ```
    // To send some Wei
    eth.sendTransaction({
      to: 'put here the address',
      from: eth.accounts[0],
      value: 25000
    });
    ```

20. // To verify the success of the transaction by checking Node 2's account balance
    `eth.getBalance("put here the address ');`

21. Do the same to connect a console to Node 2.

22. Run the command to check the balance in your accounts?

23. How match the total of ether has been credited to node2’s account?

24. Is this balance represented in the units of Wei or Ether?

25. How much 1 Ether is equal to Wei?

Any operation involving the transfer of ether will invoke a transaction. And for that you can call the send transaction method available within the eth module.
The argument to this is a JSON data structure which includes all the details of the ether transfer. This includes a 'From' address, a 'To' address, as well as the value of ether to be transferred.
The units used is Wei. To transfer 1 ether, you can either specify one followed by 18 zeroes, or to make things easier, you can make use of the toWei method available in the web3 API.

26. Add a new Node node3 to the Network (enumerate all the steps) and trigger transactions with the other nodes.

You reach the end of the lab.