

University of Basel
Faculty of Business and Economics

Master's Thesis:

Building a Private Ethereum Blockchain in a Box

Submission Date: January 6th, 2020

Supervisor: Prof. Dr. Fabian Schär

Florian Bitterli

2015-057-037

florian.bitterli@unibas.ch

Major: Monetary Economics and Financial Markets

Abstract

In this paper, we describe the process of building a private Ethereum blockchain including applications such as a faucet smart contract, a token factory, a blockexplorer and a decentralized exchange, including a web interface to interact with them. It furthermore provides theoretical background regarding the implemented proof of authority consensus protocol, ERC20 tokens and the decentralized exchange Uniswap. An actual prototype was created and it includes three Raspberry Pi computers that are connected by a local network and run a private Ethereum blockchain with the Clique proof of authority consensus protocol.

Contents

1	Introduction	2
2	Ethereum	3
2.1	Private Blockchain	4
2.2	Consensus Protocol	5
2.3	ERC20 Tokens	7
2.4	Uniswap	8
3	Components of the Briefcase	11
3.1	Hardware	11
3.2	Software	13
4	Installation Guide	14
4.1	Set Up Raspberry Pi	15
4.2	RTC Module	16
4.3	Initialize Geth	18
4.4	Network	20
4.5	Startup Scripts	21
4.6	Uniswap	25
4.7	Website	27
5	Conclusion	35
	References	i
	Appendix A User Manual	v
	Appendix B Faucet Webpage	vii

Basel, January 6, 2020

Ich bezeuge hiermit, dass meine Angaben über die bei der Abfassung meiner Arbeit benutzten Hilfsmittel sowie über die mir zuteil gewordene Hilfe in jeder Hinsicht der Wahrheit entsprechen und vollständig sind. Ich habe das Merkblatt zu Plagiat und Betrug vom 22. Februar 2011 gelesen und bin mir der Konsequenzen eines solchen Handelns bewusst.

Florian Bitterli

For any questions or comments on this paper, please contact:

florian.bitterli@unibas.ch

I would like to thank the following people for their valuable inputs and feedback:

Fabian Schär, Tobias Bitterli, Jan Dietrich, Patrice Bitterli and Célia Racine.

1 Introduction

Since the proposition of Bitcoin (Nakamoto, 2008), blockchain has been an increasingly present topic and a vast amount of different cryptocurrencies emerged. The Ethereum platform is of significant importance, because of its Turing complete programming language that led to its widely adopted smart contract possibilities. In addition to the main Ethereum network, there exist various testing networks which are used to develop and test new smart contracts and decentralized applications. For some use cases, there may be reasons not to use a public testing network and instead creating an own private network. For example, this may be because of privacy concerns. While there exists software to quickly create a local development blockchain, these solutions usually run locally on a single computer and the idea of a decentralized network, working together to find consensus, may not be directly apparent. However, such a tangible private blockchain can be very useful for some applications, especially if it involves educational purposes. For example, it provides a great foundation for showcasing the technology to someone, who not yet acquainted with blockchain technology or its smart contract possibilities. This is what we address with this paper, by providing comprehensive instructions on how to set up a private Ethereum blockchain using Linux based minicomputers. Additionally, as a part of the project, such a private Ethereum blockchain was actually set up and embedded into a briefcase. It consists of three Raspberry Pi 4 computers and offers an automatically starting, plug and play blockchain including a web user interface to facilitate usage.

The rest of the paper is structured as follows: Section 2 offers a short motivation, why Ethereum was chosen and dives into different Ethereum related aspects that are relevant for creating a private Ethereum blockchain. Section 3 concerns the *private Ethereum blockchain in a briefcase* project that was built during the process of writing this paper. It gives an overview of all the hardware and software components that were required for the building process. Finally, Section 4 provides a comprehensive installation guide, including all the steps required for setting up such a private Ethereum network on Linux based minicomputers. The individual subsections cover all the necessary parts, from setting up the computers to initializing the Ethereum software and finally making everything accessible for user interaction.

2 Ethereum

In this section, we provide a brief overview of Ethereum. We will especially cover areas, which are relevant for the creation of a private Ethereum network, for instance what a private Ethereum blockchain is and possible alternative consensus mechanisms to use in such a setting.

Ethereum was first described in 2013 in a whitepaper by Buterin (2013). Today, measured by market capitalization, it is the second most important cryptocurrency, trailing only Bitcoin (Coinmarketcap, 2019). Ethereum is especially important because of its Turing complete programming language and the smart contract possibilities (Buterin, 2013). This makes Ethereum a very popular platform for building decentralized applications and tokens using smart contracts. The importance of Ethereum is shown by the fact, that about 90% of tokens that are listed on exchanges are using this platform (Roth et al., 2019).

A smart contract is a self-executing contract consisting of lines of code. It is saved on the blockchain and is secured by it. This allows trusted transactions and agreements to be made between anonymous parties without the need for an external enforcement mechanism (Frankenfield, 2019). Smart contracts for the Ethereum platform are usually written in a high level programming language like solidity or vyper. The contract is then compiled into bytecode and an ABI (*Application Binary Interface*) and the bytecode gets deployed to the blockchain (Cassidy, 2019). This is the code that actually gets executed by the *Ethereum Virtual Machine* (EVM), whenever a function from the smart contract is called. The EVM is the bytecode execution environment of Ethereum and can be interpreted as a distributed global computer. Every node runs it as a part of the block verification protocol. Therefore, these computing tasks are not split up between nodes rather than being executed simultaneously by every node in the network (Araoz, 2016; Ethereum-Homestead, 2019). To prevent excessive usage of these computational resources, every operation has a price denoted as *gas* and paid in Ether. The total cost of the execution of a transaction or smart contract is the product of two parts: *gasUsed* and *gasPrice*. Simply put, the amount of gas used per computational step is fixed within the protocol and depends on the actual operation. The *gasPrice* is variable and can be set by the user for every transaction. It specifies the amount someone is willing to pay per unit of gas. Usually, the higher the *gasPrice*, the higher the probability to be included in a block by a miner (Ethereum-Homestead, 2019).

2.1 Private Blockchain

When thinking about Ethereum, naturally the Ethereum mainnet comes to mind – a public, permissionless blockchain. However, this is only one Ethereum network of many, although obviously the most important one. That said, there exist various different testnets and additionally, everyone is free to create their own (private) network. In fact, every Ethereum network with its nodes not connected to the main network is considered private, although private does not imply any protection or security but merely that it is reserved or isolated (Ethereum-Foundation, 2019c).

When creating an own network based on the Ethereum protocol, one is free to adjust the code and parameters. While it is theoretically possible to change about everything, a very simple and usually desirable thing to do, is creating a unique genesis block. The genesis block is the first block of the blockchain and has to be identical on all nodes. This is why the genesis block of the mainnet is usually embedded into the client (Ethereum-Foundation, 2019c). The genesis block is created based on a JSON file that contains initial values for the network, such as `chainId`,¹ `gaslimit`, the applied consensus protocol, pre-funded accounts and more. Codeblock 1 shows an excerpt of the genesis block used for this project. The first part (`config`) specifies the `chainId`, which is used for protection against replay attacks.² Further, it defines the releases to Ethereum which apply (e.g. if the chain starts before or after the homestead block), as well as the consensus protocol used (Kakarla, 2018). Here, this is Clique (see Section 2.2). Notable other entries are the `extraData` field, which is used in the Clique protocol to include allowed sealer’s addresses, and the `alloc` field that allows pre-funding of accounts.

```
{
  "config": {
    "chainId": 4596,
    "homesteadBlock": 0,
    [...]
    "petersburgBlock": 0,
    "clique": {
      "period": 15,
      "epoch": 30000
    }
  }
}
```

¹The chainId of the Ethereum mainnet is 1.

²An example of a replay attack would be, if someone receives a signed transaction message on a testnet or forked network and broadcasts it on the mainnet. To prevent this, the chainID is part of the transaction signature since EIP155 (Buterin, 2016).

```

    }
  },
  "nonce": "0x0",
  "timestamp": "0x5dae52aa",
  "extraData": "0x000[...]2064ae077aed399cec90225e1af48f6e915fab75
    87c20716f4378bbc6bda182a208da6ed27dcf9d3bc1ce2fc7949a12e0
    aaaadbfa47b6d2338c8578e3E3E1BF9833058f71413A0Fb8cA6141854C488
    6b000[...] ",
  "gasLimit": "0x47b760",
  "difficulty": "0x1",
  "mixHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x000000000000000000000000000000000000000000000000",
  "alloc": {
    "2064ae077aed399cec90225e1af48f6e915fab75": {
      "balance": "0x2000000000000000000000000000000000000000000000000000000000000000",
      [...]
    },
    [...]
  },
  "number": "0x0",
  "gasUsed": "0x0",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  [...]
}

```

Codeblock 1: Excerpt: Genesis block implementing the Clique proof of authority consensus protocol.

2.2 Consensus Protocol

Ethereum, in its current state, uses a proof of work (PoW) consensus mechanism referred to as *Ethash* (Ethereum-Foundation, 2019a). However, there are plans to eventually transition to a proof of stake (PoS) based mechanism called *Casper*. The Ethereum Foundation expects advantages such as a reduced risk of centralization and energy efficiency from the switch to PoS (Ethereum-Foundation, 2019b).

Ethereum has various public testnets that already use other consensus algorithms than proof of work. These networks are not always compatible with the same clients. Some popular ones are (EthHub, 2019):

- **Ropsten** is a testnet using the same *Ethash* PoW protocol like the Ethereum mainnet.
- **Kovan** uses the Aura proof of authority (PoA) protocol and was implemented by the team of the *Parity* client.
- **Rinkeby** was implemented by the *Geth* client team and uses the Clique PoA protocol.
- **Görli** is a fairly new community-based project that uses the Clique PoA protocol as well. It focuses on compatibility with various clients.

As described in Section 2.1, it is also possible to create new private networks. In many cases, a proof of authority consensus protocol might be a good solution in this case, as it conserves computing power compared to PoW and the participants of a small, private test network probably don't have a trust issue. For this project, we decided to use the *Geth* Clique PoA protocol to conserve computing power of the Raspberry Pi computers.

The Clique PoA protocol was introduced in *Ethereum improvement proposal EIP225*³ (Szilágyi, 2017). As with any PoA consensus protocol, the distinctive mining mechanism is, that only approved signers can create (*seal*) new blocks. Although the list of approved signers is initiated in the genesis block, it is dynamic. More precisely, existing signers can vote on adding new signers or removing existing ones (Badretdinov, 2018).

The creation of new blocks is computationally light, because there is no threshold for the block's hash value. Besides, the overall creation of a block is similar to the *Ethash* PoW mechanism. To prevent excessively mined blocks, Clique allows only one block per period of time. The length of a period can be set in the genesis block and is set to 15 seconds in the example in Figure 1 (Badretdinov, 2018).

In order to minimize chain reorganizations and to ensure that the blocks are created by different sealers, two main rules are implemented. First, blocks have a different difficulty, depending on whether they have been sealed in order or not. The definition, who's turn it is, is given by the following formula:

$$blocknumber \mod signercount$$

³<https://eips.ethereum.org/EIPS/eip-225>

Let's assume there are three signers. In this case, block 1 should be signed by the signer with index 1, block 2 by the signer with index 2 and block 3 by the signer with index 0. If a block is signed by the “in turn” signer, it has a difficulty of 2, while an “out of turn” block has a difficulty of 1. Hence, if there is a small fork, the chain with more “in turn” blocks prevails. The second rule ensures that at least 51% of sealers need to participate in the sealing process. It states that every signer can only sign once in every $\text{floor}\left(\frac{\text{signercount}}{2}\right) + 1$ consecutive blocks. (Badretdinov, 2018)

It is worth noting, how the consensus protocol is implemented with very little changes to the block structure. Basically, it just re-uses existing, not substantial fields – no additional fields are needed. The main changes concern the **extraData**, **beneficiary** and **nonce** fields of the block header. The **extraData** field is used for the signers signature. For that, it is extended from 32 bytes to 65 bytes to fit the signature hash value. Since there is no mining reward and the sealer's information is already included in the **extraData** field, the **beneficiary** field becomes unused. This is why it gets a new purpose: it is used to propose modifications to the list of signers. To implement a proposal, the corresponding address is added to the field, otherwise it should be filled with zeros. If there is a voting process regarding a sealer (e.g. there is an address in the **beneficiary** field), the **nonce** field serves to indicate whether to authenticate (nonce value 0xffffffffffff) or drop (nonce value 0x0000000000000000) the suggested address. Such voting processes work “epoch”-wise, whereas the epoch length is specified in the genesis block (Sheffield, 2018; Szilágyi, 2017).

2.3 ERC20 Tokens

A widely used field of application for smart contracts are so-called tokens. These are digital units of value, that can represent assets, utilities or promises for the delivery of goods (Roth et al., 2019). Such tokens can represent almost anything, with a few examples being currencies, stocks and property (Buterin, 2013). While there is no specific requirement how such a token has to be implemented, there was demand for some sort of *token standard* to improve the compatibility and re-usability of tokens. This was first accomplished with EIP20⁴ that proposed the ERC20 token. Although, there has been further development in the area and in

⁴<https://eips.ethereum.org/EIPS/eip-20>

the meantime, there exist other standards, the ERC20 token is still by far the most used token standard with more than 200'000 ERC20 compatible token contracts on the Ethereum blockchain (Etherscan, 2019).

The ERC20 token standard defines six functions a compliant token has to implement: `balanceOf`, `totalSupply`, `transfer`, `transferFrom`, `approve` and `allowance`. Additionally, there are optional fields for the parameters `name`, `symbol` and `decimals`. (McKie, 2017).

Main advantages of the ERC20 token standard are it's broad user and developer base, the support of all major wallet applications and the already existing open-source implementations. These are the reasons, why this token standard is used in this project for the token factory application described in Section 4.7. It has to be noted that the ERC20 token standard has some drawbacks as well. One of them being that it is possible for tokens to get stuck, if the wrong transfer function is used in a transfer to another smart contract. This already led to many lost ERC20 tokens (Roth et al., 2019).

2.4 Uniswap

Uniswap is a decentralized exchange. Unlike many other exchanges, Uniswap does not use an order book. Instead, it holds separate reserves for each trading pair, which is always given by some ERC20 compliant token and Ether. Trades are always directly executed against these reserves with an exchange rate set in a way to keep them in relative equilibrium. To incentivize a sufficient supply of liquidity, providers receive a fee for each trade (Uniswap, 2019).

The Uniswap protocol consists of two types of smart contracts: One *factory* and multiple *exchange* contracts, as every trade pair (every listed token) has its own exchange contract with its own reserves consisting of Ether and the respective ERC20 token. The factory contract serves on one hand as a registry for all exchange contracts and can be used on the other hand by anyone to deploy new exchange contracts for ERC20 tokens that are not listed yet (Uniswap, 2019).

Each exchange requires its own pool of liquidity. Let's assume someone deployed a new exchange for a token that wasn't listed before. This exchange needs reserves before it is able to process any trades. The first liquidity provider provides any amount of Ether and tokens in a ratio she thinks corresponds to the correct

exchange rate. This sets the initial exchange rate for this token. If the initial exchange rate is inaccurate, arbitrage will correct it at the cost of the initial provider.⁵ Every subsequent liquidity provider has to deposit Ether and tokens according to the current exchange rate of the pool. To keep track of the composition of the liquidity pool, liquidity ERC20 tokens are minted for every deposit and automatically credited to the liquidity provider. The amount of created tokens reflects the relative size of the deposit in respect to the pool size. If, for example, a new liquidity provider wishes to deposit Δx Ether and there are already x Ether in the liquidity pool, the amount of Ether in the pool increases by the factor $\alpha = \frac{\Delta x}{x}$. She has to ensure, that the amount of tokens in the pool (y) increases by the same factor such that the exchange rate is maintained. Hence, she has to deposit $\Delta y = \alpha * y$ tokens. This increases the total pool by the factor α and makes her eligible for minting $\alpha * l$ liquidity tokens, where l represents the initial amount of liquidity tokens. This is summarized in set of equations 1 (Uniswap, 2019; Zhang et al., 2018).

$$\begin{aligned}x' &= (1 + \alpha)x \\y' &= (1 + \alpha)y \\l' &= (1 + \alpha)l\end{aligned}\tag{1}$$

where $\alpha = \frac{\Delta x}{x}$

These liquidity tokens can be burned at any time in order to withdraw the corresponding part of the pool. In this case, she burns Δl liquidity tokens and thus decreases the supply of tokens by the factor $\alpha = \frac{\Delta l}{l}$. Therefore, she has the claim on $\alpha * (x + y)$ of the reserves (Zhang et al., 2018).

$$\begin{aligned}x'' &= (1 - \alpha)x' \\y'' &= (1 - \alpha)y' \\l'' &= (1 - \alpha)l'\end{aligned}\tag{2}$$

where $\alpha = \frac{\Delta l}{l}$

⁵As we will show later, the exchange rate is determined by the $x * y = k$ model. If the exchange rate is not the real exchange rate, it will be possible for anyone to buy either Ether or the token via the exchange below its true value.

As every token pair has its own liquidity pool, liquidity tokens are unique for each exchange. However, because they are represented by standard ERC20 tokens, they are transferable, which makes it possible for anyone to transfer ownership of their share of the pool without having to withdraw it (Uniswap, 2019).

For every exchange, the exchange rate is given by the so-called $x * y = k$ model, which was proposed for usage in decentralized exchanges by Buterin (2018). In the model, x and y represent the amounts of Ether and token in the pool and k is a constant. Thus, if someone trades Ether to get some token, x increases and y decreases in a way that leaves its product, k , constant. This is represented in set of equations (3) (Uniswap, 2019; Zhang et al., 2018).

$$\begin{aligned}
 k &= x * y \\
 &= (x + \Delta x) * (y - \Delta y) \\
 &= (1 + \alpha)x * \frac{1}{(1 + \alpha)}y
 \end{aligned} \tag{3}$$

where $\alpha = \frac{\Delta x}{x}$

Therefore, after each trade, the composition of the liquidity pool will be different and thus, the subsequent trade will have a different exchange rate. If two subsequent trades are in the same direction, the second one will have a “worse” exchange rate than the first one. If they are in opposite directions, the second one will have a “better” exchange rate than it would have had with the exchange rate from the beginning (Uniswap, 2019; Zhang et al., 2018).

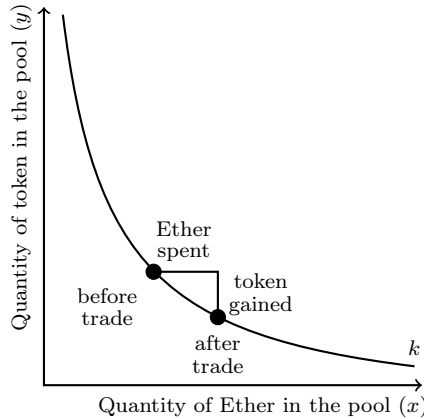


Figure 1: The Uniswap pricing mechanism. Own illustration based on Buterin (2018).

This relation is also shown in Figure 1, where the slope of the curve represents the marginal exchange rate. The more Ether are in the pool relatively to the token, the more expensive the token gets and vice versa.

Because the liquidity providers actually receive a fee for each trade, the above equations (3) have to be slightly adjusted. The fee is represented by a fixed percentage (σ) of the initial amount sent to the smart contract and is deducted before the trade takes place. If for example someone sends Ether (Δx) to receive tokens (Δy), the fee is deducted from the amount of Ether, hence only $(1 - \sigma)\Delta x$ will be exchanged. Based on this, the exchange rate is determined using the $x * y = k$ model specified in equation 3. After the trade, the fee ($\sigma * \Delta x$) is added to the liquidity pool. Thus, k will in fact increase slightly after each trade. This represents the profit a liquidity provider is able to earn once she burns her liquidity tokens to withdraw her share of the pool (Uniswap, 2019).

3 Components of the Briefcase

In this section, we will briefly describe the *private Ethereum blockchain in a briefcase* and its components, while Section 4 will then provide an extensive overview of all steps required to reproduce the project. Figure 2 shows schematically the briefcase’s contents. The main elements are three Raspberry Pi 4 computers which are connected by a network switch. Each computer runs an instance of the *Geth* (Go Ethereum) software client (see Section 3.2), hence each computer runs a *full node*. The last hardware component is a wifi router which is connected to the network switch as well. It creates a wifi network for users to conveniently access the private blockchain.

In what follows, we will provide an in depth overview of all individual hardware and software components that are used in the project.

3.1 Hardware

The heart of the system are the Raspberry Pi 4 minicomputers. These are ARM based computers that are hardly larger than the area of a credit card.⁶ They were set up with a variant of the Arch Linux operating system called Manjaro

⁶More information about Raspberry Pi can be found on raspberrypi.org

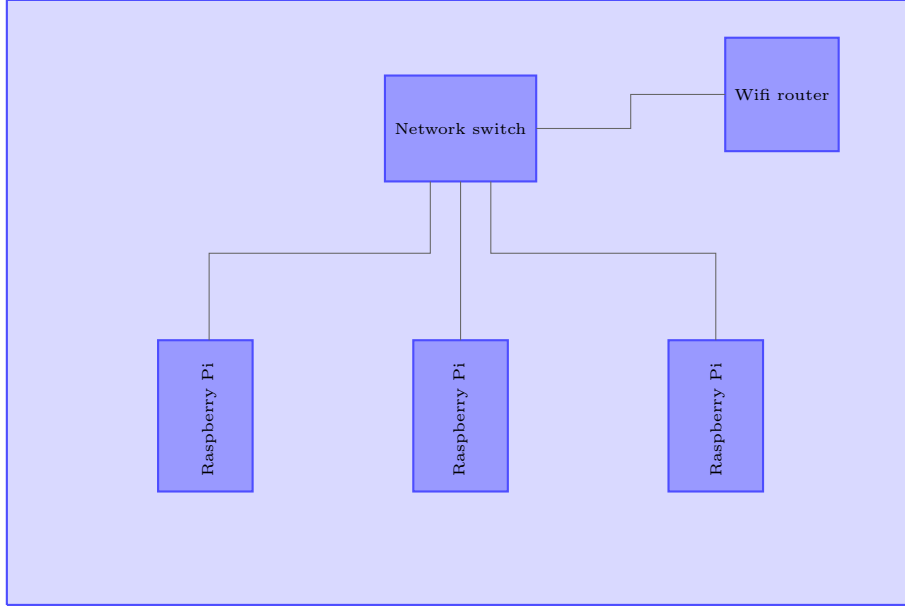


Figure 2: Hardware components of the briefcase.

(see Section 3.2). Each computer runs a full signing node and they are the only devices in the private network that can create blocks. More precisely, they have the private keys for the only authorized accounts for signing blocks in the used Clique PoA consensus protocol (see Section 2.2). One of the computers has two additional tasks. First, it runs a second node that can't seal blocks, but acts as a HTTP JSON-RPC (*remote procedure call*) server to facilitate the connection of users via their client or browser. Additionally, it also hosts the internal website with its functions like the blockexplorer, token factory and Uniswap.

Each Raspberry Pi computer has two additional components connected to its GPIO (*general-purpose input/output*) Pins. First, two pins are used to power a heat sink with a fan which should keep the processor cool and protect it from overheating when being used for a long period of time. Second, a RTC (*real time clock*) module is connected. In contrast to standard desktop computers, Raspberry Pi computers do not include an internal RTC module, which is required to keep the time while not being connected to a power source. Instead, they usually rely on the NTP (*network time protocol*) to request the current time from a server during the boot process. However, this method is not feasible in our set up, because the computers form a local network which is not connected to the Internet. Thus, they cannot connect to any NTP server. The RTC module offers a solution, because it is powered by an own battery which allows it to keep the correct time, even if the Raspberry is shut down. Instead of consulting a NTP server during the boot process, the Raspberry is

programmed to request the current time directly from the RTC module and set the software clock accordingly. Keeping the time synchronized across the computers is especially important, because it is a prerequisite for Ethereum nodes to connect to other nodes and for their acceptance of blocks.

The three Raspberry Pi nodes form a private network by being connected via a network switch. Additionally, a wifi-router is part of the network that allows users to connect to the network and hence, the blockchain.

3.2 Software

There is a variety of software involved in creating and running such a private Ethereum blockchain. The following table lists and describes all the software involved in the set up process described in Section 4.

	<p><i>Manjaro</i> is an open-source Linux distribution based on Arch Linux. It is one of the first 64 bit operating systems available for the Raspberry Pi, which is why it was selected instead of Raspbian (the default operating system provided by the Raspberry Foundation). The 64 bit capability is required for executing the mining function in <i>Geth</i>. More information and download: https://manjaro.org.</p>
	<p><i>Go</i> (also known as <i>Golang</i>) is an open-source programming language created at and supported by Google. It is a prerequisite for the <i>Geth</i> Ethereum client. More information on: https://golang.org.</p>
	<p><i>Geth</i> is short for <i>Go Ethereum</i> and is the <i>Go</i> client of the Ethereum protocol. There are other Ethereum implementations such as <i>Parity</i>, <i>Aleth</i> and more. More information on: https://geth.ethereum.org.</p>
	<p><i>Atom</i> is a text editor that can be configured with various packages for syntax highlighting in different programming languages. We will mainly use it for the coding of the website and for the faucet and token smart contracts. More information and download: https://atom.io.</p>

	<i>Remix</i> is a browser based IDE for Solidity code. It is a simple solution to write, compile and deploy smart contracts and was used for the deployment of the faucet and Uniswap smart contracts. It is accessible on: https://remix.ethereum.org .
	<i>Node.js</i> is a JavaScript runtime to build network applications. We will use it to built our webserver for the internal website. More information and download: https://nodejs.org .
	<i>npm</i> is the default package manager for <i>Node.js</i> . It is included in the <i>Node.js</i> installation.
	<i>Express</i> is a <i>Node.js</i> web application framework. We will use it to create the <i>Node.js</i> webserver. It can be installed using the npm package manager and the following command: <code>\$ npm install express --save</code> . More information on: https://expressjs.com .
	<i>Balena Etcher</i> is a tool for “flashing” operating system images to an SD card. It will be used to write the <i>Manjaro</i> OS onto a micro-SD card to insert it in the Raspberry Pi. More information and download: https://www.balena.io/etcher .
	<i>MetaMask</i> is a browser extension that provides a bridge between the browser and Ethereum. It stores a user’s private keys and offers the possibility to interact with decentralized applications via the browser. We will use it for the interaction between a user and the private blockchain. More information and download: https://metamask.io .

4 Installation Guide

In this section, we will provide a comprehensive overview of all steps involved in creating a private Ethereum blockchain in a briefcase. It covers all the steps from installing the operating system for the Raspberry Pi computer and the initialization of Geth to the hosting of the internal website.

4.1 Set Up Raspberry Pi

The Raspberry Pi is distributed without a pre-installed operating system. It is therefore necessary to download your own copy and install it on a micro-SD card. *Manjaro* for Raspberry Pi 4, which is used in this project, can be downloaded from <https://manjaro.org/download>. Using Balena Etcher, the image can be transferred to a (micro-)SD card.

With the micro-SD card inserted, the Raspberry Pi can be connected to a power source, a monitor and peripherals. It will start the boot process, prompt a few questions, such as username, password, timezone etc. and finally set up the system. Once the boot process is finished, it is possible to log in with the specified username. If it is desired to do the subsequent steps from another computer with *ssh*, connect the Raspberry Pi to the Internet and find out its *IP address* using the `ip a` command. After, it is possible to log in from any other computer in the local network through *ssh*. It is furthermore advisable, to already create static IP addresses, as described in Section 4.4, because otherwise it would be necessary to look up the Raspberry's IP address every time.

```
$ ip a    # In the Raspberry Pi terminal
$ ssh username@ip_address    # In the terminal of a remote computer
e.g. $ ssh pi@192.168.0.100
```

Because *Manjaro* is based on Arch Linux, it is equipped with the *pacman* package manager. It is advisable to update all packages to get started. To do this, use the following command:

```
$ sudo pacman -Syu
```

To run an Ethereum node, we will then install the *Geth* client, which is built on *Go*. Hence, we will install *Go* and *Geth* next. Once the installation is completed, it can be verified with the `version` command.

```
$ sudo pacman -S go go-ethereum
$ go version
$ geth version
```

4.2 RTC Module

As mentioned in Section 3.1, the Raspberry Pi does not feature a hardware clock. In order to keep time up to date and synced across the nodes, a real time clock module is added to the GPIO board of the computer. We used a module of the DS3231 type that can be put directly on the first five pins on the inside of the GPIO board (i.e. pins 1,3,5,7 and 9).

Before we initialize the RTC module, we will disable the automatic time setting via NTP. This command might need to be run as `sudo`.

```
$ timedatectl set-ntp false
$ timedatectl # Check that NTP service is inactive
```

Once the RTC module is in place, we first follow the steps of Brittain (2019) to enable the pins. To do that, we first install the necessary software and the *Raspi-Config* tool which we will then use to activate the pins.

```
$ sudo pacman -S git python2 i2c-tools base-devel python2-pip
python2-distribute
$ sudo pip2 install RPi.GPIO
$ sudo pacman -S xorg-xrandr libnewt
$ git clone https://aur.archlinux.org/raspi-config.git
$ cd raspi-config
$ makepkg -i
$ sudo raspi-config
```

The last command should start the *Raspi-Config* tool. Here, we select *interfacing options* and then enable *I2C*.

For the changes to take effect, a reboot is required. If everything worked as intended, the Raspberry Pi should now be able to detect the module. The following command should show an overview and in one of the spots a numerical value should be displayed – that’s the RTC module.

```
$ sudo i2cdetect -y 1
```

Now that our hardware clock gets detected, we have to enable it. For that, the following commands are required for Arch Linux (Levavasseur, 2019). They need to be run as root to work.

```
$ sudo bash
$ echo "dtoverlay=i2c-rtc,ds3231" » /boot/config.txt
$ cat <<EOF > /etc/udev/rules.d/55-rtc-i2c.rules
#/lib/udev/rules.d/50-udev-default.rules:SUBSYSTEM=="rtc",
ATTR{hctosys}=="1", SYMLINK+="rtc"
#/lib/udev/rules.d/50-udev-default.rules:SUBSYSTEM=="rtc",
KERNEL=="rtc0", SYMLINK+="rtc", OPTIONS+="link_priority=-100"
#I2C RTC, when added and not the source of the sys clock (kernel),
is used
ACTION=="add", SUBSYSTEMS=="i2c", SUBSYSTEM=="rtc",
KERNEL=="rtc0", ATTR{hctosys}=="0", \
RUN+="/sbin/hwclock '--rtc=\$root/\$name' --hctosys", \
RUN+="/sbin/logger --tag systemd-udevd 'System clock set from i2c
hardware clock \$name (\$attr{name})'"
EOF
```

A reboot is now required. After the reboot, we can get information about our newly configured hardware clock. However, the time will not yet be correct. The `hwclock` command should give us the current time of the RTC if everything is set up correctly. With the `timedatectl` command, we can set the correct time and check it again using `hwclock`.

```
$ sudo hwclock --verbose
$ timedatectl set-time "YYYY-MM-DD HH:MM:SS"
$ sudo hwclock
```

Right now, after a reboot, the Raspberry will again display the wrong time. However, with the `hwclock` command, the correct time should be displayed. This is because we have only set up the hardware clock so far and did not yet specify a command to set the software clock accordingly during the boot process. This isn't an issue for now, because we will add such a command to the startup script in Section 4.5.

4.3 Initialize Geth

To initialize *Geth*, we need to specify a data directory and a *genesis file*. Using the `mkdir` command, we can create a folder which we will use to store the blockchain data. To navigate inside the file structure, the `cd` command can be used. Additionally, the `ls` command might be useful, as it lists all files in the current directory.

```
$ mkdir datadir
$ cd datadir    # Move to specified directory
$ cd ../        # Go back to previous directory
$ ls            # List files in current directory
```

To get started and to be able to create a genesis file in case of the PoA consensus protocol, we need to create at least one account first, because we have to specify at least one authorized sealer address. This can be done with the following command and may be repeated as often as desired.

```
$ geth --datadir ./datadir account new
```

Next up is the genesis file. This can effortlessly be configured using the *puppeth*-tool, which is included in the *Geth* installation.⁷ Once open, simply type in any network name. Then, follow the prompts and provide the required information to automatically generate your genesis file. After successful creation, it is possible to export the genesis block as a JSON file.

```
$ /usr/bin/puppeth
"network_name"
2.Configure new genesis
... provide the required information ...
2.Manage existing genesis
2.Export genesis configurations
```

Once the JSON file is created, the same file must be used to initialize all nodes. It is also possible (but not necessary) to load the previously created accounts on every

⁷*Puppeth* can not only be used for creating a genesis block. In fact, it is a very handy tool to set up a complete private Ethereum blockchain, including applications like a faucet, ethstats and more. It works by using ssh to deploy docker containers to remote servers. Unfortunately, it was not possible to use *puppeth* for this project, because its docker containers are not compatible with the Raspberry's ARM processors.

computer by transferring the files inside the *keystore* directory. In this project, initially 10 accounts were created and loaded onto every computer, so that they can be accessed on every node. The file transfer can be done either with a USB stick or with the `scp` command over *ssh*. In order to transfer a folder, the recursive `-r` option can be used.

```
$ scp path/to/file username@ipaddress:path/to/destination
e.g. $ scp genesis.json pi@192.168.0.100:datadir
e.g. $ scp -r datadir/keystore pi@192.168.0.100:datadir
```

With the genesis file and the data directory specified, *Geth* can now initialize the node.

```
$ geth --datadir ./datadir init path/to/genesis.json
e.g. $ geth --datadir ./datadir init datadir/genesis.json
```

The node is now ready to be started. To start a local session with an interactive JavaScript console interface, use:

```
$ geth --datadir ./datadir console
```

A few examples of useful commands can be found on page 25. Note, that the start process of the console shows the *enode* value, which will be required later in order to manually connect new nodes. For that, we create a file inside the `datadir/geth` data directory called *static-nodes.json*, which contains the enode values of all nodes. Once *Geth* is running, it will always automatically try to connect to the nodes provided in this file (Go-Ethereum, 2019b). Note, that this already requires static IP addresses, as these are part of the *enode* value. If no static IP addresses have been set yet, it is best to create the *static-nodes.json* file later.

```
[
  "enode://pubkey@ip:port",
  "enode://f4642fa65af50cfdea8fa7414a5def7bb7991478b768e296f5e4a54
    e8b995de102e0ceae2e826f293c481b5325f89be6d207b003382e18a8ecba
    66fbaf6416c0@192.168.0.100:30303"
]
```

Codeblock 2: Example *static-nodes.json* file.

4.4 Network

To connect everything and form a private network, the computers and the wifi router are connected to the network switch. Usually, the router requires some sort of set up procedure to specify its mode of operation as well as the network SSID (the network name) and potentially a password. The router used in this project offered a web-interface to guide through the required steps.

For the discovery and manual connection to the other nodes, static IP addresses are required. They also facilitate the usage of *ssh*. There are various options to assign a static IP address. Usually the user-interface of the router can be used to assign a permanent IP address to any device on the network. This is how we will proceed. If it is preferred to set the IP address directly on the Raspberry Pi, Haley (2016) offers a comprehensive step by step guide on how to accomplish that with Arch Linux.

Usually, when accessing the router's settings (most likely through the same web interface used to set it up), there is a section titled DHCP (*Dynamic Host Configuration Protocol*). This section should provide a list of connected devices, including their MAC and IP addresses. Additionally, it should provide an option to enter a MAC address and assign a static IP address to it. This should be done with all three Raspberry Pi computers. To find out the MAC address of a specific Raspberry Pi, the `ip a` command can be used.

Another network-related thing to be done is to set a hostname for the Raspberry computer that will later host the internal website. As everything will only be accessible in a local network, the most simple way is using *mDNS* (*multicast Domain Name System*). This will make the computer accessible through a `hostname.local` address. After installing the necessary packages, we can activate the service and set a hostname according to the following steps (Frields, 2018).

```
$ sudo pacman -S nss-mdns avahi
$ sudo systemctl enable --now avahi-daemon.service
$ hostnamectl set-hostname briefcase # Set the desired hostname
$ sudo systemctl restart avahi-daemon.service # Restart the service
$ ping briefcase.local # Use on a remote computer to test the set up
```

4.5 Startup Scripts

This section provides an overview of the startup script that automatically starts *Geth* and the one that starts the web server. Basically, there are two things involved. First, a *bash* script and second, a *service* that starts the former every time the Raspberry boots. We will first take a look at the *bash* script to start *Geth*. It is located in the `/usr/bin/` directory. This script is present in this form on all computers, although it needs to be slightly adjusted to reflect the different signing accounts and static IP addresses of each computer.

```
#!/bin/bash
# Ether startup

sleep 3
sudo hwclock -s
geth --datadir /home/pi/datadir --miner.gasprice 0 --txpool.
    pricelimit 0 --networkid 4596 --nodiscover --nat extip:192.168.
    0.100 --port 30303 --syncmode "full" --miner.ethersbase 0x123...
    --unlock 0x123... --password /home/pi/datadir/pw.txt --mine --
    miner.threads 1
```

Codeblock 3: *Geth* startup script "scriptEther" on one Raspberry Pi.

The script contains only three commands. To start with, there's a sleep command to ensure, that the process to communicate with the hardware clock is already running. Next, using the `hwclock -s` command, the system time is set to the time of the hardware clock (which was set to the actual time in Section 4.2). The last and most important command starts *Geth* and hence, the node. There are various flags involved which will now be described briefly according to Go-Ethereum (2019a).

- **--datadir:** Specifies the data directory where the blockchain data is stored in.
- **--miner.gasprice:** Sets the minimal gas price for a transaction to be mined by this node. Here, it is set to 0. This facilitates the faucet function, because it ensures that a new user can send a withdraw request to the smart contract without any initial funds.

- **--txpool.pricelimit:** Similar to the previous flag, this sets the gas price for the acceptance of a transaction in the transaction pool. This is required to accept the incoming transactions with a gas price of 0, which are relayed by the RPC-node.
- **--networkid:** Allows to specify a unique network identification number. This is set to a non-default value, because nodes are required to have the same protocol version and network ID to connect to each other. It is therefore used to generate an isolated network (Ethereum-Foundation, 2019c).
- **--nodiscover:** This is used to disable the peer discovery mechanism. With this setting, manual peer addition is required.
- **--nat extip:** Adds the value of the static IP defined in Section 4.4, so that it can be included in the node's enode address (Lange, 2016).
- **--port:** Specifies the network listening port.
- **--syncmode:** Accepts the values "full", "fast" and "light". "Full" means, that all blocks are downloaded and all transactions validated from the beginning. "Fast" still stores all blocks, but does only validate more recent transactions. "Light" only stores the current state and requests blocks on an as needed basis (Balla, 2018).
- **--miner.etherbase:** Sets the public address for mining rewards. This is especially relevant in the Clique PoA protocol, because it has to reflect the public address of an approved sealer.
- **--unlock:** Specifies an account to be unlocked with the following password. This is needed, because with Clique, miners have to *seal* blocks using their private key (Badretdinov, 2018).
- **--password:** The password (or path to the file containing the password) to unlock the previously defined account.
- **--mine:** Used to enable mining.
- **--miner.threads:** Sets the amount of CPU threads used for the mining (or *sealing*) process.

As already mentioned, one Raspberry Pi runs two nodes simultaneously. The second node facilitates the RPC server, which is needed for the users to connect to

the network (e.g. via MetaMask). This process requires a separate node, because for security reasons, this function is not compatible with the block sealing process which requires an unlocked account. The overall startup script for the RPC server looks similar, however, *Geth* is initialized using different flags. Additionally, the `sleep` command is set to a few seconds more and the `hwlock` command is missing. This is because the time is already set on this computer by the first script.

```
#!/bin/bash
# Ether startup

sleep 5
geth --datadir /home/pi/datadirRPC --networkid 4596 --nodiscover
    --syncmode "full" --nat extip:192.168.0.101 --rpc --rpcport "85
45" --rpcaddr "[192.168.0.101]" --rpccorsdomain=* --rpcapi="
admin,db,eth,debug,miner,net,shh,txpool,personal,web3"
```

Codeblock 4: *Geth* startup script "scriptEtherRPC".

Different flags used for this process are (Go-Ethereum, 2019a):

- **--rpc:** Enables the HTTP JSON-RPC server.
- **--rpcport:** Specifies the server's listening port.
- **--rpcaddr:** Specifies the listening interface of the server.
- **--rpccorsdomain:** List of domains from which the server accepts cross-origin requests.⁸ The provided value (*) allows requests from any domain.
- **--rpcapi:** List of the APIs offered via the RPC-interface.

The last startup script is not related to *Geth*. Instead, it is required to automatically start the web server that serves the internal website which will be described in Section 4.7. The included commands navigate to the directory containing the website data and start the *Node.js* process with the `npm run start` command. The command is run with `sudo`, because we will specify the web server to listen on standard port 80 in Section 4.7.

⁸Cross Origin Resource Sharing (CORS) is used to allow applications running on one origin to access resources from a different origin (Mozilla, 2019).

```
#!/bin/bash
# Ether startup

sleep 5
cd home/pi/websiteBriefcase
sudo npm run start
```

Codeblock 5: Startup script for *Node.js*.

All scripts described so far in this section should start automatically during the boot process. This requires, that they are executable. Once they are executable, we can create a service which starts them. Setting the permission to execute a script is accomplished with the `chmod` command.

```
$ sudo chmod 755 /usr/bin/scriptEther
```

The last thing to do is to create and enable a service that executes these scripts automatically during the boot process. The files for these services should be located in the `/etc/systemd/system` directory. We will call them `ether.service`, `etherPRC.service` and `website.service`, respectively. All of them are built in a similar way and Codeblock 6 depicts the service to start `scriptEther`.

```
[Unit]
Description=Script

[Service]
ExecStart=/usr/bin/scriptEther

[Install]
WantedBy=multi-user.target
```

Codeblock 6: `ether.service` file used to start *scriptEther* during the boot process.

Now, we simply have to enable the service to execute it during the boot process.

```
$ sudo systemctl enable ether.service
```

Alternatively, to manually start the service, the command can be used with `start` instead. Additionally, using `status`, it is possible to check, whether the task runs as expected.

```
$ sudo systemctl start ether.service # Start service manually
$ sudo systemctl status ether.service # Check the status of the service
```

Once everything is set up and running, it is possible to attach to the node, to issue commands and check that everything works as intended. The `attach` command can be used to launch a *JavaScript* console that exposes the full *web3* API. A few commands that might be useful for usage in the console are shown below.

```
$ sudo geth --datadir ./datadir attach
$ admin.peers # Lists the node's connected peers
$ eth.blockNumber # Returns the number of the latest block
$ miner.ethbase # Returns the coinbase account
$ miner.start # Starts the mining process
$ miner.stop # Stops the mining process
$ txpool # Shows the transaction pool
$ eth.getBalance("publicAddress") # Shows the balance of an account
```

4.6 Uniswap

As already mentioned earlier, the decentralized exchange Uniswap will be accessible on the private network. For that, three components are necessary. These are:

- The initial exchange contract which will be deployed for each tradeable token pair.
- A factory contract that can be used to deploy new exchanges and which keeps track of all deployed exchanges.
- A user interface to facilitate usage of the application.

The components will be deployed in this order. For that, we will use steps similar to Bakaoh (2019). First and foremost, the relevant contracts can be downloaded from the Uniswap git repository.⁹ For the deployment of the contracts, we will use Remix and MetaMask. On MetaMask, we can connect to the relevant network, which should be the private network we set up earlier in this section. Then, the

⁹github.com/Uniswap/contracts-vyper

Uniswap exchange contract can be compiled using the Remix vyper compiler. Afterwards, the compiled contract needs to be deployed to the injected environment from MetaMask. After successful deployment of the initial exchange contract, the same procedure can be applied to the Uniswap factory contract. The lone difference is, that the address of the already deployed exchange contract needs to be added before being able to deploy the factory contract. Now, Uniswap is already deployed, the only missing piece is a user interface to simplify usage of the contracts. The official frontend can be cloned from the git repository as well. It can then be modified to work with the newly deployed contracts on our private network.

```
$ git clone https://github.com/Uniswap/uniswap-frontend
$ cd uniswap-frontend
$ yarn
```

A few values have to be specified. First, the `.env.local.example` file should be renamed to `.env.local`. In this file, our `networkID` and the URL of our RPC server need to be specified.

```
REACT_APP_NETWORK_ID="4596"
REACT_APP_NETWORK_URL="http://192.168.0.101:8545"
```

Codeblock 7: Specified values in the *env.local* file.

Additionally, the address of the factory contract on our network has to be specified within the `src/constants/index.js` file.

```
export const FACTORY_ADDRESSES = {
  1: '0xc0a47dFe034B400B47bDaD5FecDa2621de6c4d95',
  3: '0x9c83dCE8CA20E9aAF9D3efc003b2ea62aBC08351',
  4: '0xf5D915570BC477f9B8D6C0E980aA81757A3AaC36',
  42: '0xD3E51Ef092B2845f10401a0159B2B96e8B6c3D30',
  4596: '0x082c43c8C47b7B3763deB221D25737A3C9e6288A'
}
```

Codeblock 8: First part of the *src/constants/index.js* file.

Now, the frontend should be ready to be started. To do this, the `yarn start` command can be used.

```
$ yarn start    # Run the frontend at local.  
$ yarn build    # Create a build version to add to the website in Section 4.7.
```

4.7 Website

After all the steps described so far in this section, everything should be set up now. That is, there are the Raspberry Pis equipped with a RTC module each, running an automatically starting signing node, as well as one of them running an additional HTTP JSON-RPC server and a script that starts the web server. The missing part is the website which will provide the user access to the information about the network, as well as functions such as a faucet, a blockexplorer and a token factory. Additionally, the Uniswap frontend from the previous section can be linked to it as well.

For hosting the website, a light web server is built using *Node.js* and *express*. The center piece is the *Server.js* file depicted in Codeblock 9. It handles all browser requests and responds by sending the appropriate files.

```
var express = require('express');  
var web3 = require('web3')  
var bodyParser = require('body-parser')  
var app = express();  
var router = express.Router();  
var path = __dirname + '/views/';  
  
router.get('/', function(req, res){  
  res.sendFile(path + 'index.html');  
});  
  
router.get('/contract', function(req, res) {  
  console.log(req.body);  
  res.json(require('./build/contracts/CreateToken.json'));  
});  
  
[...]
```

```

router.get('/uniswap',function(req,res){
    res.sendFile(__dirname + '/uniswap/index.html');
});

app.use(bodyParser.urlencoded({extended: false}));
app.use('/',router);

app.use(express.static(__dirname + '/public'));
app.use(express.static(__dirname + '/uniswap'));

app.use('*',function(req,res){
    res.sendFile(path + '404.html');
});

app.listen(80,function(){
    console.log('Live at Port 80');
});

```

Codeblock 9: Excerpt: *Server.js* file.

At the beginning of the file, the relevant modules are loaded using the `require` command. They are saved as variables. Below, the main part consists of `router.get` commands, which are used for the routing. Whenever the server gets a specific URL, it responds with the defined action, i.e. by sending the appropriate file. The last part of the file defines middleware to be used by the application. Finally, the `app.listen` command defines the port, *Node.js* is listening. Here, this is set to port 80, which is the default port for a web server. The advantage of using this port is, that it does not have to be specified by the user in the URL. This means, it is sufficient to type the URL *briefcase.local* instead of something like *briefcase.local:3000* to access the website.

The individual webpages are *html files* located in the *views* directory. The main structure of each page is based on a template by Quackit (2019). The individual pages won't be explained in detail, because they consist of a bunch of similar code and would be beyond the scope of this paper. However, to get an idea, the Codeblock in Appendix B shows the *html* code for the faucet webpage.

A number of libraries are used for the webpages. The main functionality and design relies on the *bootstrap* and *jQuery* libraries. Additionally, the usage of the *web3* libraries is notable, as these facilitate the interaction with the blockchain through JavaScript. The basic structure of each individual webpage consists of a naviga-

tion bar on top, which provides links to all other webpages. Then, each page has a similar looking header with its respective title and a large cover picture. Below, the specific application is accessible. There are separate pages for the faucet function, the token factory, a blockexplorer which was forked from the Etherparty git repository,¹⁰ and a page with technical details concerning the blockchain specification. Additionally, the main page (*index.html*) offers descriptions and links to all applications, including the Uniswap exchange frontend which was described in Section 4.6.

All webpages that need a connection to MetaMask and the blockchain include a custom JavaScript file which enables the connection to MetaMask and also contains the functions for the faucet and token factory. Codeblock 10 shows the code.

```
// Offset for Site Navigation
$('#siteNav').affix({
  offset: {
    top: 100
  }
})

// Connect to MetaMask
window.addEventListener('load', async () => {
  if (window.ethereum) {
    window.web3 = new Web3(ethereum);
    try {
      await ethereum.enable();
    } catch (error) {
    }
  }
  else if (window.web3) {
    window.web3 = new Web3(web3.currentProvider);
  }
  else {
    console.log('Non-Ethereum browser detected. You should
      consider trying MetaMask!');
  }
});
ethereum.autoRefreshOnNetworkChange = false;
```

¹⁰<https://github.com/etherparty/explorer>

```

//Faucet Function
function withdrawfromFaucet() {
    const faucetCaller = window.ethereum.selectedAddress;
    var faucetContract = new web3.eth.Contract([contractABI],
        'contractAddress');
    faucetContract.methods.withdraw().send({from: faucetCaller,
        gasPrice: '0', gas: '300000'}, function(error, result){
    });
}

function faucetFunction(e) {
    e.preventDefault();
    withdrawfromFaucet();
}

//Tokenfactory Function
function deployContract(compiledContract) {
    var name = document.getElementById('Tokenname').value;
    var symbol = document.getElementById('Tokensymbol').value;
    var supplyWithoutDecimals = document.getElementById('Tokensupply
        ').value;
    var decimals = document.getElementById('Tokendecimal').value;
    var supply = supplyWithoutDecimals * 10 ** decimals;
    const deployAccount = window.ethereum.selectedAddress;
    localStorage['name'] = document.getElementById('Tokenname').
        value;

    var createContract = new web3.eth.Contract(compiledContract.abi)
        ;
    createContract.deploy({
        data: compiledContract.bytecode,
        arguments: [name, symbol, decimals, supply]
    })
    .send({
        from: deployAccount,
        gas: '1500000',
        gasPrice: '5000000000'
    }, function(error, transactionHash){ })
    .on('error', function(error){
        console.log(error)})
    .on('transactionHash', function(transactionHash){
        console.log(transactionHash);
        localStorage['trxhash'] = transactionHash})
    .on('receipt', function(receipt){

```



```

        localStorage['tokenAddress'] = receipt.contractAddress})
    .on('confirmation', function(confirmationNumber, receipt){ })
    .then(function(newContractInstance){
        console.log(newContractInstance.options.address)
    });
}

function getContractAndDeploy(e) {
    e.preventDefault();
    var xmlHttp = new XMLHttpRequest();
    xmlHttp.onload = function() {
        var compiledContract = JSON.parse(this.responseText);
        deployContract(compiledContract);
    }
    xmlHttp.overrideMimeType('application/json');
    xmlHttp.open('GET', '/contract');
    xmlHttp.send();
}

```

Codeblock 10: The *custom.js* file with the functions for the faucet and the token factory.

The first part is used to connect to MetaMask. It is specified to distinguish between modern and legacy *dapp* browsers as well as non-Ethereum browsers.

The faucet and token factory functionality consist of two functions each. The main function of the faucet is `withdrawfromFaucet`, which saves the user's current MetaMask address and issues a contract call on her behalf of the specified faucet smart contract shown in Codeblock 11. It is notable, that the transaction has a specified `gasPrice` of 0, because the user does not have funds yet. As seen in Section 4.5, the nodes are set up in a way, that they accept such transactions and include them in the blockchain. The second function (`faucetFunction`) is the one that gets executed, as soon as the user requests funds. It has a `e.preventDefault()` command and then calls the already described `withdrawfromFaucet` function. This nested set up is used, because the transaction transmission has to wait for the user's approval in MetaMask.

```

pragma solidity ^0.5.0;

contract faucet {
    mapping (address => bool) public alreadyRequested;

    function() external payable {
    }

    function balanceOfFaucet() public view returns(uint256){
    return address(this).balance;
    }

    function withdraw() public {
        if (alreadyRequested[msg.sender] == false){
            msg.sender.transfer(10000000000000000000);
            alreadyRequested[msg.sender] = true;
        }
    }
}

```

Codeblock 11: The faucet smart contract.

The smart contract for the faucet shown in Codeblock 11 consists of four elements:

- The `alreadyRequested` mapping which saves addresses that have already requested Ether. This is used to limit the requests one individual can make.
- A fallback function to ensure, the contract accepts funds.
- The `balanceOfFaucet` function with the purpose to return the remaining funds of the faucet.
- A `withdraw` function, which enables the main functionality of the contract: requesting funds. When called, the function checks whether the message sender has already made a request before. If not, it will transfer 10 Ether to her address and set her status to `alreadyRequested = true`. Otherwise, it won't do anything.

The last part of the *custom.js* file in Codeblock 10 contains the functionality for the token factory. The `deployContract` function first creates variables with the data provided by the user on the website (*name*, *symbol*, *supply* and *decimal places*

of the token). It also saves the user's address as a constant. It then creates a new web3 contract instance with the ABI of the token smart contract which is provided by the `getContractAndDeploy` function. The contract instance gets extended by including the provided bytecode and the arguments of the user. This makes the contract ready for deployment. This transaction is prepared on behalf of the user and she simply has to confirm it on MetaMask. The function listens for the receipt and finally saves the transaction hash and the token address into local storage to make it accessible for the follow up "*deploymentinfo*" page. This function is specified similarly to the `withdrawfromFaucet` function of the faucet, hence, it does not get called directly, but by the `getContractAndDeploy` function. This is the function the user activates when she submits her arguments on the token factory webpage. The idea is similar to the `faucetFunction` from before. However, it additionally creates a `xmlHTTP` request to load the smart contract's information from the web server. The uncompiled code of the token that gets deployed is shown in Codeblock 12.

```
pragma solidity ^0.5.0;

import "@openzeppelin/contracts/token/ERC20/ERC20Detailed.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract CreateToken is ERC20, ERC20Detailed {
    constructor(string memory _name, string memory _symbol, uint8
        _decimals, uint256 _supply)
        ERC20Detailed(_name, _symbol, _decimals)
    public {
        _mint(msg.sender, _supply);
    }
}
```

Codeblock 12: The token smart contract.

The token used for the token factory is based on the OpenZeppelin secure smart contracts.¹¹ These are available as *npm module* and can then simply be imported as shown in the code. The `ERC20Detailed` smart contract accepts a constructor for the *name*, *symbol* and *decimal units* of the token. Additionally, the `_mint` command creates the specified amount of tokens and transfers them to the creator of the token.

¹¹<https://openzeppelin.com/contracts>

At any time during the development of the website, the web server can be run to test the individual webpages and functions. This can be done with the following command. As long as some other port than port 80 is used, the commands can be run normally. For the final web server, when we specify port 80, running these commands as `sudo` is required.

```
$ npm run start or node Server.js
```

Once everything is set up, the directory containing the website can be moved to the correct Raspberry Pi. Here, we already created a startup script to start the web server in Section 4.5. Because we use the standard port 80 in the final version of the *Server.js* file, we have specified in the startup script to run the command with `sudo`.

5 Conclusion

This paper covered and explained all steps necessary to build a private Ethereum blockchain. The Ethereum platform was chosen because it is the most important platform for smart contracts and decentralized applications. When creating an own, private blockchain, it is possible to change various specifications. Usually, this is accomplished by creating a unique genesis file which is the basis to generate the genesis block. Within this file, it is possible to specify the consensus protocol applied, pre-fund accounts and more. A requirement for the connection of multiple nodes is, that they have to be initialized with the same genesis block. An additional factor to note is the need of a synchronized system time across the individual computers, because blocks whose timestamp is not conform with the node's own time are not going to be accepted. This is especially relevant when using Raspberry Pi computers, because they do not include a module to keep time while they are not connected to a power source.

In connection with this paper, an actual prototype of such a private Ethereum blockchain was created. Within its genesis file, the usage of the Clique proof of authority consensus protocol was specified. The resulting product is given by three Raspberry Pi computers that are embedded into a briefcase and hence offer a portable solution. Each computer runs a signing node, while one computer runs an additional node that handles and transmits the requests from users. Additionally, a web user interface with different applications was developed. It offers functionality to request Ether from a faucet smart contract and allows the tracking of blocks and transactions via a blockexplorer. Furthermore, it offers a token factory function that takes a user's arguments and deploys an ERC20 token smart contract on her behalf. These tokens can be added to the included decentralized exchange Uniswap. This exchange has the defining characteristic, that trades are facilitated by a liquidity pool instead of relying on order books.

The overall product offers a wide range of applications with possible use cases in the development of smart contracts but also especially its usefulness for educational purposes should be noted. Its design with three distinct nodes and its intuitive user interface make it a viable tool to show the mode of operation of a blockchain.

Despite the various applications, there is still area for improvement and extension. This especially concerns the number of available applications. There is a lot of additional functionality that could be implemented. For example, a way to invest

Ether and tokens could be implemented using the compound protocol. Also, the included blockexplorer is very basic and thus limited. While it reliably lists all blocks and the included Ether transactions, it is not capable of properly displaying transactions to and between smart contracts and token transactions are not traceable as well. This however would be very useful, especially when developing and testing smart contracts. Another useful functionality to implement, especially for usage of the briefcase for educational purposes in groups, would be a database for token addresses. This would allow users to access and add addresses of existing and newly created tokens, which would greatly improve their discoverability, because the address of a token is required to display them in MetaMask and also to find their exchange on Uniswap.

References

- Araoz, M. (2016), ‘The Hitchhiker’s Guide to Smart Contracts in Ethereum’.
URL: <https://medium.com/zeppelein-blog/the-hitchhikers-guide-to-smart-contracts-in-ethereum-848f08001f05>, Accessed: December 31, 2019
- Badretdinov, T. (2018), ‘Clique: cross-client Proof-of-authority algorithm for Ethereum’.
URL: <https://medium.com/@Destiner/clique-cross-client-proof-of-authority-algorithm-for-ethereum-8b2a135201d>, Accessed: December 18, 2019
- Bakaoh (2019), ‘Deploying Uniswap on the Matic testnet’.
URL: <https://bakaoh.com/hunter/3398-matic-network-matic-bounties/>, Accessed: December 17, 2019
- Balla, J. (2018), ‘Optimal sync mode for running an ethereum node that can process transactions’.
URL: <https://ethereum.stackexchange.com/questions/38907/optimal-sync-mode-for-running-an-ethereum-node-that-can-process-transactions>, Accessed: December 23, 2019
- Brittain, T. (2019), ‘Setup i2c on Raspberry Pi Zero W using Arch Linux’.
URL: <https://ladvien.com/arch-linux-i2c-setup/>, Accessed: December 20, 2019
- Buterin, V. (2013), ‘A next-generation smart contract and decentralized application platform’.
URL: <https://github.com/ethereum/wiki/wiki/White-Paper>, Accessed: December 27, 2019
- Buterin, V. (2016), ‘EIP 155: Simple replay attack protection’.
URL: <https://eips.ethereum.org/EIPS/eip-155>, Accessed: December 29, 2019
- Buterin, V. (2018), ‘Improving front running resistance of $x*y=k$ market makers’.
URL: <https://ethresear.ch/t/improving-front-running-resistance-of-x-y-k-market-makers/1281>, Accessed: December 28, 2019

- Cassidy, J. (2019), ‘Understanding smart contract compilation and deployment’.
URL: *https://kauri.io/understanding-smart-contract-compilation-and-deployment/973c5f54c4434bb1b0160cff8c695369/a*,
Accessed: December 31, 2019
- Coinmarketcap (2019), ‘Top 100 Cryptocurrencies by Market Capitalization’.
URL: *https://coinmarketcap.com*, *Accessed: December 27, 2019*
- Ethereum-Foundation (2019a), ‘Ethereum Wiki - Ethash’.
URL: *https://github.com/ethereum/wiki/wiki/Ethash*, *Accessed: December 18, 2019*
- Ethereum-Foundation (2019b), ‘Ethereum Wiki - Proof of Stake FAQ’.
URL: *https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ*, *Accessed: December 18, 2019*
- Ethereum-Foundation (2019c), ‘Go Ethereum’.
URL: *https://geth.ethereum.org/docs/interface/private-network*, *Accessed: December 15, 2019*
- Ethereum-Homestead (2019), ‘Account Types, Gas, and Transactions’.
URL: *https://ethereum-homestead.readthedocs.io/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html*, *Accessed: December 31, 2019*
- Etherscan (2019), ‘Token Tracker’.
URL: *https://etherscan.io/tokens*, *Accessed: December 30, 2019*
- EthHub (2019), ‘Test Networks’.
URL: *https://docs.ethhub.io/using-ethereum/test-networks/*, *Accessed: December 18, 2019*
- Frankenfield, J. (2019), ‘Smart Contracts’.
URL: *https://www.investopedia.com/terms/s/smart-contracts.asp*, *Accessed: December 31, 2019*
- Frields, P. W. (2018), ‘Find your systems easily on a LAN with mDNS’.
URL: *https://fedoramagazine.org/find-systems-easily-lan-mdns/*, *Accessed: December 22, 2019*

Go-Ethereum (2019a), ‘Command-line Options’.

URL: *<https://geth.ethereum.org/docs/interface/command-line-options>*, Accessed: December 31, 2019

Go-Ethereum (2019b), ‘Connecting To The Network’.

URL: *<https://geth.ethereum.org/docs/interface/peer-to-peer>*, Accessed: December 20, 2019

Haley, W. (2016), ‘Use a Static IP in Arch Linux with dhcpd’.

URL: *<https://willhaley.com/blog/static-ip-in-arch-linux/>*, Accessed: December 22, 2019

Kakarla, S. (2018), ‘An Introduction to the Genesis Block in Ethereum’.

URL: *<https://www.skcript.com/svr/genesis-block-ethereum/>*, Accessed: December 15, 2019

Lange, F. (2016), ‘No IP address in enode addresses’.

URL: *<https://github.com/ethereum/go-ethereum/issues/2765>*, Accessed: December 22, 2019

Levavasseur, A. (2019), ‘Properly set-up i2c RTC ds1307 on ArchLinux’.

URL: *<https://gist.github.com/Alex131089/de45d552372a9296abbbbe407ae52180>*, Accessed: December 20, 2019

McKie, S. (2017), ‘The Anatomy of ERC20’.

URL: *<https://medium.com/blockchannel/the-anatomy-of-erc20-c9e5c5ff1d02>*, Accessed: December 30, 2019

Mozilla (2019), ‘Cross-Origin Resource Sharing (CORS)’.

URL: *<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>*, Accessed: December 31, 2019

Nakamoto, S. (2008), ‘Bitcoin: A Peer-to-Peer Electronic Cash System’.

URL: *<https://bitcoin.org/bitcoin.pdf>*, Accessed: November 25, 2019

Quackit (2019), ‘Business Website Templates’.

URL: *https://www.quackit.com/html/templates/business_website_templates.cfm*, Accessed: November 28, 2019

Roth, J., Schär, F. and Schöpfer, A. (2019), ‘The Tokenization of Assets: Using Blockchains for Equity Crowdfunding’.

URL: *<https://dx.doi.org/10.2139/ssrn.3443382>*, Accessed: December 27, 2019

Sheffield, N. (2018), ‘Ethereum Clique PoA vs PoW’.

URL: *<https://medium.com/coinmonks/ethereum-clique-poa-vs-pow-11be52cddde1>*, Accessed: December 18, 2019

Szilágyi, P. (2017), ‘EIP225: Clique proof-of-authority consensus protocol’.

URL: *<https://github.com/ethereum/EIPs/blob/master/EIPS/eip-225.md>*, Accessed: December 18, 2019

Uniswap (2019), ‘Uniswap Whitepaper’.

URL: *https://hackmd.io/C-DvuDSfSxuh-Gd4WKE_ig*, Accessed: December 28, 2019


Zhang, Y., Chen, X. and Park, D. (2018), ‘Formal Specification of Constant Product ($x * y = k$) Market Maker Model and Implementation’.

URL: *<https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf>*, Accessed: December 30, 2019

A User Manual

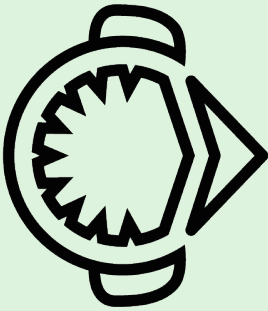
USER MANUAL

Private Ethereum Blockchain in a Box



This booklet offers a guide on how to connect, request initial funds and use this Ethereum blockchain and its pre-installed applications.

GET CREATIVE BY YOUR OWN!



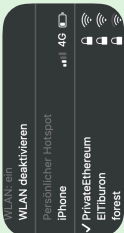
The possibilities of this briefcase are not limited to the internal applications. You can use this network like any other Ethereum test network. A few possibilities are:

- Use *remix* to create and deploy your own smart contract.
- Use the browser console to interact via *web3*-commands.
- Build your own website to interact with this blockchain.

CONTACT:
Florian Bitterli
florian.bitterli@icloud.com


CONNECT TO THE NETWORK

1




Look for the PrivateEthereum Network
Password: WWZunibas

2



Open the Networks Tab & Select „Custom RPC“

3



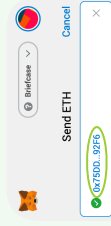
Enter the Details as Shown in the Picture

4 Visit the Website

HTTP://BRIEFCASE.LOCAL



REQUEST 10 ETHER



You can find various applications on the internal website. Once you requested funds from the faucet, you can send transactions and use the pre-installed applications like you would on another public testnet.

AVAILABLE APPLICATIONS

[illegible]

B Faucet Webpage

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale
    =1">
  <title>Faucet</title>

  <!-- CSS -->
  <link href="stylesheets/bootstrap.min.css" rel="stylesheet">
  <link href="stylesheets/custom.css" rel="stylesheet">
  <link href='stylesheets/customfonts' rel='stylesheet' type='text
    /css'>

  <!-- Favicon -->
  <link rel="shortcut icon" href="favicon.png" type="image/x-icon
    ">
</head>

<body>
  <!-- Navigation -->
  <nav id="siteNav" class="navbar navbar-default navbar-fixed-top"
    role="navigation">
    <div class="container">
      <!-- Logo and responsive toggle -->
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="
          collapse" data-target="#navbar">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="/">
          <span>
            
          </span>
          Briefcase
        </a>
      </div>
    </div>
  </nav>
</body>
</html>
```

```

    </a>
</div>
<!-- Navbar links -->
<div class="collapse navbar-collapse" id="navbar">
  <ul class="nav navbar-nav navbar-right">
    <li>
      <a href="/">Home</a>
    </li>
    <li>
      <a href="details">Chain Specifications</a>
    </li>
    <li class="dropdown active">
      <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria-expanded="false">Applications <span class="caret"></span></a>
      <ul class="dropdown-menu" aria-labelledby="about-us">
        <li><a href="faucet">Faucet</a></li>
        <li><a href="explorer">Blockexplorer</a></li>
        <li><a href="tokenfactory">Tokenfactory</a></li>
        <li><a href="uniswap">Uniswap</a></li>
      </ul>
    </li>
    <li>
      <a href="/#contact">Contact</a>
    </li>
  </ul>
</div><!-- /.navbar-collapse -->
</div><!-- /.container -->
</nav>

<!-- Header -->
<header>
  <div class="header-content">
    <div class="header-content-inner">
      <h1>Faucet</h1>
      <p>Press the Request Button below, sign the transaction and receive your test-Ether within a couple of seconds.
        <br>
        To avoid excessive usage, requests are limited to one per address. If you need more funds, request again from a different address.</p>
      <a href="#faucetform" class="btn btn-primary btn-lg">Go to

```

```

        Faucet</a>
    </div>
</div>
</header>

<!-- Content 1 -->
<section id="faucetform" class="content">
    <div class="container">
        <div class="row">
            <div class="col-sm-6">
                
            </div>
            <div class="col-sm-6">
                <h2 class="section-header">Faucet</h2>
                <p class="lead text-muted">Press the button and receive
                    10 test-Ether.<br>
                    Make sure to connect your MetaMask to the correct
                    network (Custom RPC: <i>http://192.168.0.101:8545</i>)</p>
                Requests are limited to one per address. If you need
                    more funds, use different addresses for multiple
                    requests.</p>
                <form onsubmit="return faucetFunction(event)">
                    <center>
                        <button id="faucetRequest" type="submit" class="btn
                            btn-primary btn-lg">Request 10 Ether</button>
                    </center>
                </form>
            </div>
        </div>
    </div>
</section>

<!-- Footer -->
<footer class="page-footer">
    <!-- Copyright -->
    <div class="small-print">
        <div class="container">
            <p>Copyright &copy; Florian Bitterli 2019</p>
        </div>
    </div>
</footer>

```

```
<!-- jQuery -->
<script src="javascripts/jquery-1.11.3.min.js"></script>
<!-- Bootstrap Core JavaScript -->
<script src="javascripts/bootstrap.min.js"></script>
<!-- Plugin JavaScript -->
<script src="javascripts/jquery.easing.min.js"></script>
<!-- web3 -->
<script src="javascripts/web3-trufflesuite.min.js"></script>
<!-- Custom Javascript -->
<script src="javascripts/custom.js"></script>

</body>
</html>
```

Codeblock 13: HTML code for the faucet webpage.