UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Karl Veskus

Ethereum versus Fabric – A comparative analysis

Bachelor's Thesis (9 EAP)

Supervisor: Fredrik Payman Milani, PhD
Supervisor: Luciano García-Bañuelos, PhD

Tartu 2018

Ethereum versus Fabric - A Comparative Analysis

Abstract:

The aim of this bachelor's thesis is to introduce the blockchain technology and its two different platforms, Ethereum and Hyperledger Fabric, that are made to create very different blockchain based applications. Then, both of these platforms are used to create similar applications. It is followed by the comparison of these platforms while covering different aspects such as architecture and user friendliness. It also compares the development processes as well as applications built for the comparison

Keywords:

Blockchain Technology, Ethereum, Hyperledger Fabric

CERCS: P170 - Computer science, numerical analysis, systems, control

Ethereum versus Fabric – võrdlev analüüs

Lühikokkuvõte:

Käesolevas bakalaureusetöös tutvustatakse plokiahela tehnoloogiat ning selle kahte erinevat platvormi, Ethereum ja Hyperledger Fabric, mis on loodud väga erinevate rakenduste loomiseks kasutades just plokiahela tehnoloogiat. Peale selle luuakse mõlemat platvormi kasutades sarnane rakendus, millele järgneb võrdlus. Esiteks, annab võrdlus lugejale ülevaate, kuidas antud platvormid üksteisest arhitektuuri ning kasutajamugavuse poolest erinevad. Teiseks näitab, kuidas erineb nende platvormide abil rakenduste loomine ning mille poolest erinevad lõputöö raames loodud rakendused.

Võtmesõnad:

Plokiahela tehnoloogia, Ethereum, Hyperledger Fabric

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of content

1. Introduction

Shortly after the Financial Crisis of 2008, where many financial institutions went bankrupt, people who lost their money started to look for alternative ways to store their funds. Particularly, a currency that would not be controlled by any central authority, such as bank or government [1]. Then, a new financial system was introduced - bitcoin[1], an online payment system using a peer-to-peer network to allow payments to be sent directly from one party to another without the need for a trusted third party [2].

Soon after that, it was discovered, that the blockchain technology behind bitcoin is really powerful and can be used separately for other applications as well as it effectively solves the double spending problem [3]. This has led to a situation in which a lot of organizations such as IBM, Ethereum Foundation, etc. have built platforms that help to build decentralized applications on top of the blockchain. However, as every platform is designed for a specific business use case, their concepts are usually considerably different.

The main goal of the thesis is to give an overview of the blockchain technology and compare its different platforms, namely Ethereum[2] and Hyperledger Fabric[3]. The comparison is based on the conceptual design and is supported by identical implementations of one possible use case to cover additional development aspects.

Chapter 2 describes the background and terms used in the thesis. Chapter 3 gives an overview of related work done about the comparison of Ethereum and Hyperledger Fabric platforms. Chapter 4 starts by describing the use case for practical comparison and then it explains the concepts behind the chosen platforms. Then, chapter 5 describes the implementations and chapter 6 compares the platforms and gives results.

[1] https://bitcoin.org/en/
[2] https://www.ethereum.org/
[3] https://www.hyperledger.org/projects/fabric

2. Background

This chapter gives an overview of terms and technologies covered in the thesis. It explains the idea behind distributed ledger technology and blockchain and how they differ from each other. Then it shows, what makes them different from widely used centralized solutions.

2.1 Distributed ledger technology

Distributed ledger is a type of data structure that is shared and replicated over multiple locations [4]. All the parties participating in this kind of data storage form a network of so-called nodes where every participant holds the exact copy of the ledger [5].

These ledgers are categorized into two groups - unpermissioned and permissioned. Unpermissioned ledgers have no single owner and therefore let anybody participate in the network, no permission is required [4]. A good example of the unpermissioned ledger is Ethereum, which is covered in depth in the fourth and fifth chapter.

Permissioned ledgers, on the other hand, may have one or many owners and its network participants are preselected [4]. So, the whole network is limited only to the restricted set of users. This is true for Hyperledger Fabric which is covered in depth in the fourth and fifth chapter.

Participating in the network means that nodes are permitted to contribute to the ledger management [6]. But as all the data is distributed across the network, it is important to ensure that all the nodes agree upon a common truth, since the changes made by one node are propagated to all other nodes. The common truth agreed upon nodes is called consensus and it is found using various algorithms like proof of work, proof of stake, etc. [5].

The ledgers are categorized even further, public and private, depending on whether every actor can access the ledger or is it limited only to participating nodes. The whole taxonomy of the ledgers is illustrated in Figure 1.

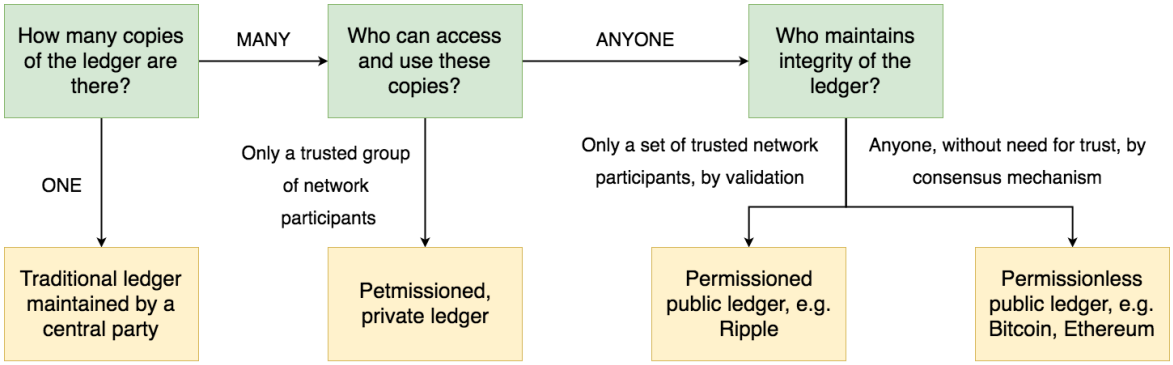

Figure 1. Distributed Ledger Taxonomy [4]

2.1.1 Centralized ledger versus distributed ledger

In the classical centralized model (Figure 2), there is a central authority who controls the ledger and multiple actors who want to gather some data from it or change it. This brings us multiple problems. First, all the data is stored and changed by the central authority and since no actor can be sure that the data has not been changed, they have to blindly trust it. Second, if something goes wrong, there is a few if any methods to check its correctness as actors do not have access to original data.

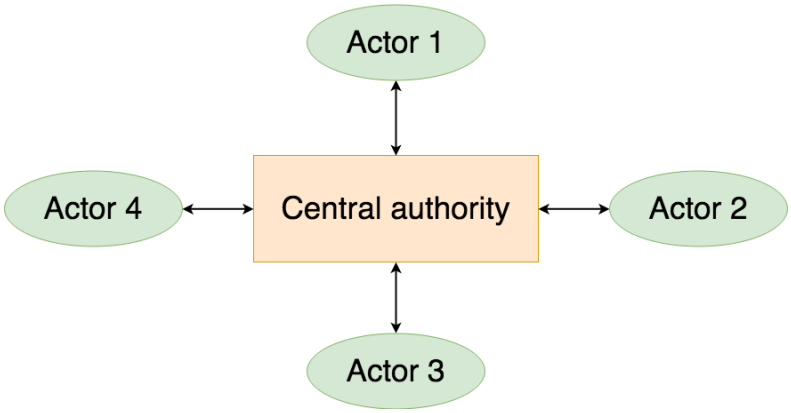

Figure 2. Centralized ledger

In contrast with a centralized model, the distributed model (Figure 3) has no central authority and all the data is stored in the distributed ledger where every node has the exact copy of it and changes to the ledger state are based on a consensus of permitted participants.

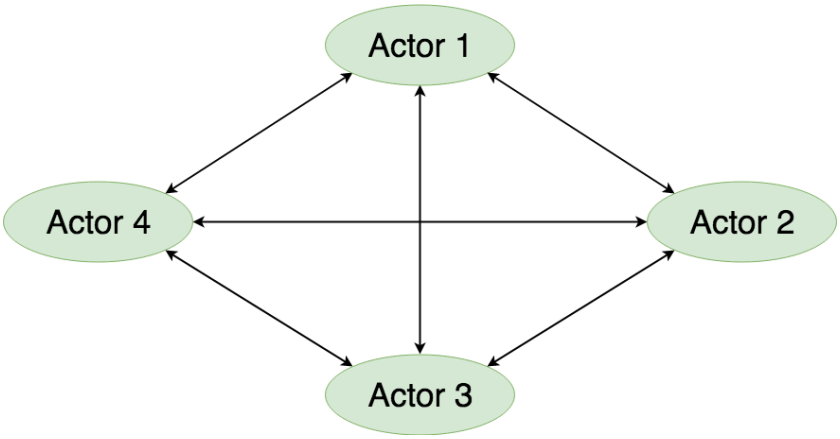

Figure 3. Distributed ledger

2.2 Blockchain

A blockchain is a type of data structure that is used in many distributed ledger technologies. It bundles all the changes made to the ledger (transactions) into packages called blocks and chains them together, using cryptographic hashing, providing an immutable record of all transactions from the genesis (first) block [7]. The structure is presented in Figure 4.

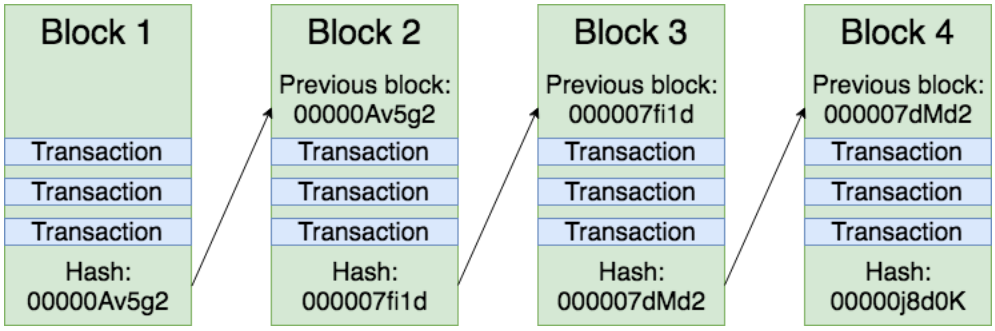

Figure 4. Blockchain structure

However, the hash of the block is calculated using all the transactions and the hash of a previous block. So, the hash of the last block represents the whole blockchain as one value, which makes comparing two chains really easy. As a result, if some transactions in the blockchain are sensitive we do not have to share the whole chain to make the comparison but only the last hash.

As described, blockchain is immutable record of all transactions. It means that changing any transaction in the chain results in a completely new chain. Let us assume that we chance one transaction in the block 3. Now its hash is changed and therefore the hash in the block 4 is chances as well as it includes the previously updated hash.

2.3 Smart contracts

A smart contract is a virtual contract that is stored and executed by the blockchain like any other transaction. It acts as a regular contract between multiple parties but differs from the latter by not being approved by any central authority, such as notary or government, but by the network itself. It may contain any rules and therefore form any business logic that gets validated by every node [8]. Basically said, it is a code, stored in the blockchain, that stipulates the conditions and waits for certain input to execute. For example, it is possible to set the code in smart contract to be executed at a certain time only.

To illustrate the idea behind the smart contracts, let us assume that parties A and B want to make a bet for the upcoming football match but they do not trust each other. For example, what happens, if the loser decides not to send the money to the winner? For that reason, we create a smart contract, that first collects a stake from both parties and then, after the match is finished, sends collected stakes to the party, that made a right bet. In case of a tie, it simply returns the stakes. Now, where the contract is made, it is up to the network to check who actually won the bet and therefore reward the winner. The result is a fast and secure contract that is confirmed by all the nodes to ensure security.

2.4 Decentralized application

Decentralized application, also known as dapp, is an application that connects users and providers directly [9]. It runs in distributed network and uses blockchain to store its data.

These applications are divided into three main categories. First, financial applications, that are purely designed to manage digital money and can be used to pay for other users or managing loans. Second, semi-financial applications, that mixes money with the outside information. A good example to illustrate would be the insurance application that according to the verified accidents pays the money. Third, other application such as online voting and governance applications [8].

3. Related work

Blockchain technology has gained a lot of attention over past several years. It is really popular and widely researched area in both, academic and business area. A lot of research papers have been released and comparisons between multiple blockchain platforms have been done. Even between Ethereum and Hyperledger Fabric which are compared in the given thesis. The comparisons between these platforms have been done multiple times, but most of them have been more theoretical and therefore do not cover development aspects that are critical for developers.

One example, that compares Ethereum and Hyperledger Fabric, is directed to the decision makers new to the blockchain [5]. It gives a brief analysis of key differences of named platforms to get an overview of the blockchain technology and possible use cases. The general comparisons between the platforms have been done more [10]. In contrast to the previous report, this one explains concepts more in depth, but still does not cover any other development aspects but supported programming languages.

Another one complements these as it provides the performance analysis [11]. It shows clearly that Fabric constantly surpasses Ethereum in case of execution time, latency and throughput. This fact is also supported by another performance analysis [12] that in addition to the performance compares smart contract systems and many consensus algorithms.

As they all provide mostly theoretical analysis of the biggest differences, this thesis complements them with practical differences while covering the implementations of sample use case.

4. Use case

This chapter gives an overview of the use case selected for the comparison. After this, we explain why Ethereum and Hyperledger Fabric were selected and give a theoretical overview of these platforms.

4.1 Pet Shop

The use case itself comes from Truffle framework tutorials and it is called a Pete's Pet Shop [10]. Pete is the pet shop owner who needs a blockchain based web application where he can give away his pets. On the webpage, the clients should be able to see a picture and description of every single pet that Pete has. And, if they have chosen the pet they want to get, they should able do it using the "Adopt" button below the image. Then, the adoption should be recorded on the blockchain, saving the client information, to see later who made these adoptions.

For that reason, we need to build a blockchain application to store all the data about the pets and a smart contract to read and write data from the blockchain (business logic). Then, to interact with the blockchain application, we need to build a front-end application that lists pets as in Figure 5.

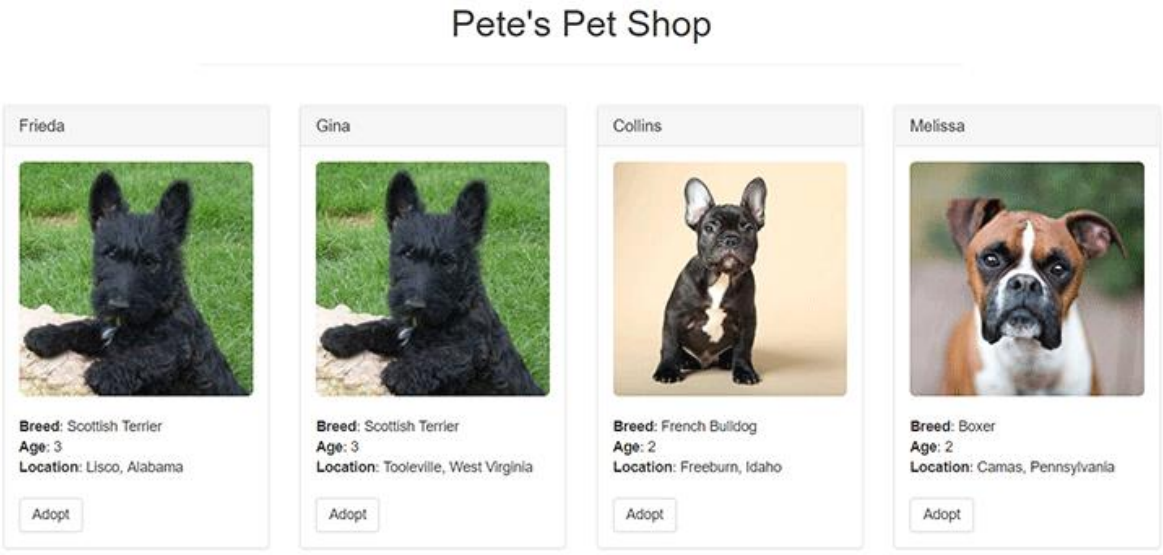

Figure 5. Pete's Pet Shop web client

4.2 Selected distributed ledger technologies

4.2.1 Ethereum

Ethereum was created as a response to Bitcoin which is only focused on transferring monetary value between parties and has a limited programming language. Ethereum is not just a digital currency, but a largest and most well-established decentralized software platform that enables developers to build decentralized applications on top of the blockchain using built-in Turing complete programming language Solidity[4] [13]. Ethereum has, similarly to Bitcoin, its own crypto-currency and it is called Ether.

As Ethereum uses its own decentralized public blockchain [14], all the transactions, data and smart contracts for applications using Ethereum are public and can be accessed using any Ethereum blockchain explorers such as Etherchain[5] and Etherscan[6].

Accounts

In the Ethereum blockchain, there are two types of accounts: externally owned Account (EOA) and contract accounts. Externally owned accounts are controlled by private keys and they are being created automatically. Contract accounts, on the other hand, are controlled by their contract code [15].

Architecture

Ethereum is a peer-to-peer network where every node runs under Ethereum Virtual Machine (EVM) to execute the smart contracts code and transactions. If the user wants to make a change in the blockchains state, he can do that by publicly announcing transaction. Then, it is up to the network to verify the transaction using consensus mechanism described below [16].

For anti-denial of service model, Ethereum transactions contain two extra values - STARTGAS and GASPRICE, where gas stands for the fundamental unit of computation. STARTGAS is the maximum number of computational steps for the transaction and GASPRICE is the fee that sender pays per computational step [8]. So, the total transaction fee can be cal-

[4] https://solidity.readthedocs.io/en/v0.4.23/
[5] https://www.etherchain.org/
[6] https://etherscan.io/

culated as STARTGAS * GASPRICE. Then, this fee is subtracted from the sender's account. If the fee exceeds the sender's balance, then the transaction is reverted. However, the gas is not returned, it will be sent to the miner.

Consensus mechanism

Currently, Ethereum uses mining based consensus algorithm called the Proof of Work. It is designed so, that creating a new block is computationally hard and therefore prevents Sybil attacks. To make it hard, all blocks must contain a nonce, a meaningless value, that is used to calculate the hash of the block. So, what miners do, is they repeatedly change the nonce to match the proof of work conditions. To motivate nodes to participate in the mining process, the miner that produces a new valid block gets rewarded with a certain number of coins. After the new block is made, the whole process of guessing the nonce starts again [8].

Longest chain rule

To illustrate the purpose of mining better, let us use the sample from Ethereum White Paper [8], where attacker's plan is as follows:

1. Send 100 Ether to a merchant to buy some product that is instantly delivered
2. Wait for the delivery
3. Make a new transaction sending the same 100 Ether back to himself
4. Convince the network to use the second transaction instead of the first one

Let us suppose that the first transaction is done and added to the block number 100 and 5 more blocks are created after that, so the last block would be number 105. At this moment, the merchant sends out the product and, as it is delivered instantly, the attacker receives it immediately. Now, the attacker takes the block number 100 and chances the previously made transaction so that the money would have been sent to himself instead. If the attacker would broadcast it to the network, then it would be ignored since the coins are spent already. To prevent it, the attacker has to fork the chain and start mining from block 100. As the transaction is changed, the proof of work has to be redone. Plus, since the original chain has the 105 blocks already, the attacker has to redo the proof of works for blocks from 101 to 105 as well. This is because all the following blocks have to refer to the previous blocks

which hashes have changed now. According to the longest chain rule, the honest miners continue working on the original chain since it is the longest and attacker alone will work on his new chain. To make the attacker's new chain longest, the attacker needs more computational power than the network together to catch up [8].

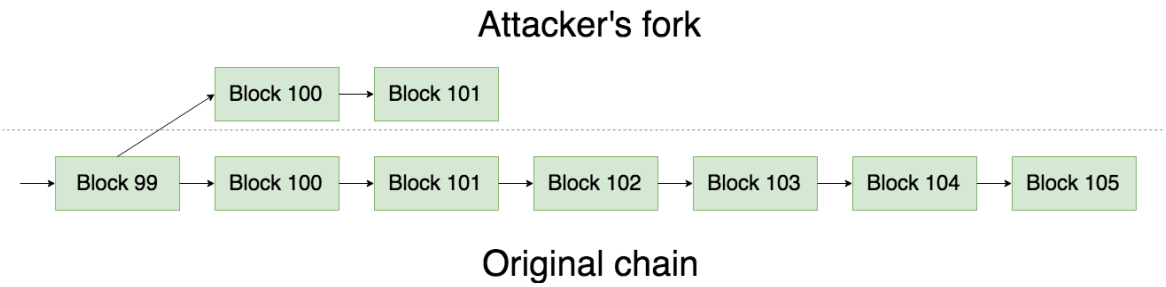

Figure 6. Forking the chain

4.2.2 Hyperledger Fabric

Hyperledger Fabric, or just Fabric, is an implementation of an enterprise-grade distributed ledger platform that is private and permissioned. In contrast to public permissionless blockchains that allow anyone to join the network, Fabric requires its participants to be enrolled through trusted Membership Service Provider (MSP). It allows for modular architecture, which means that the consensus mechanisms and MSPs can be swapped in and out [17]. That results in a universal architecture, that can be applied to most industries.

Channels

In contrast to public permissionless blockchains where all the transactions are stored in one public ledger, Fabric allows multiple private ledgers to be created using channels. Channel is a completely separate and independent "subnetwork" that defines its members and a ledger of transactions. However, its data is visible and accessible only to its participants. That makes it an important option in networks where participants are competitors and therefore cannot share all the data [18].

They can be effectively used for businesses that want to share confidential data with trusted nodes without other nodes noticing it. Let us assume that we have a farmer John who usually sells apples for 10 coins per kilogram to Mia, Tony and other shop owners. However, John agrees to give Tony a special price of 6 coins per kilogram. He does not want the special price to be available to everyone in the node, but still have every node be able to view the

details about the apples he is selling. Using channels, John and Tony can privately agree on the terms of the special deal.

Chaincode

The smart contract, in Fabric, is referred to as a chaincode. It is installed and instantiated on the channel's peers who provides a way for the end users to invoke it through a client-side application. However, while the instantiation, the endorsement policy may be defined. It is used to refer to a set of endorsers that are necessary or sufficient for a valid transaction endorsement [19].

Three main components of Fabric

Peer – the node that receives the ordered blocks from the ordering service and thereby maintains the ledger. There is also a special type of peer, an endorsing peer, also known as endorser, whose task is to endorse transactions before they are committed [20].

Ordering service or **Orderers** – orderers form the ordering service. It first collects all the endorsed transactions from the client, then it orders and bundles them into the blocks and sends these blocks to every peer to commit it into the ledger. It is also responsible for the endorsement policy verification. For example, if the policy requires 3 or more endorsements but the orderer receives only 2, then these transactions are stored only locally and will not be broadcasted to the peers [20].

Certificate Authority – a tool that is used to issue signed certificates for the users to participate in the network. Inside these certificates, it is possible to attach additional attributes such as account number, roles, etc. and it is propagated to the system so chaincode has an access to it.

Transaction flow

Now, we explain the transaction flow (Figure 7) and steps done before the transaction is committed to the ledger. In Fabric network, the transaction starts with client application sending transaction proposal to one or many endorsing peers. Then, as endorsers have a full copy of ledger, they can simulate the transaction and capture the set of Read and Written data, called RW Sets (step 1). These RW contain the data that were read from and written to the current state while simulating the transaction. Then, these RW sets are signed by

endorsers and returned to the client application. Then, the application collects the endorsements (step 2) and submits the endorsed transaction and the RW sets to the orderer (step 3). Now, the ordered has to confirm the policies and check if all the RW sets are equivalent. If there are no issues, all the data is sent to every peer (step 4). Then these RW sets are applied to the ledger, which results in every node having the same state (step 5). Lastly, the peers notify the client of the success or failure of the transaction [20].

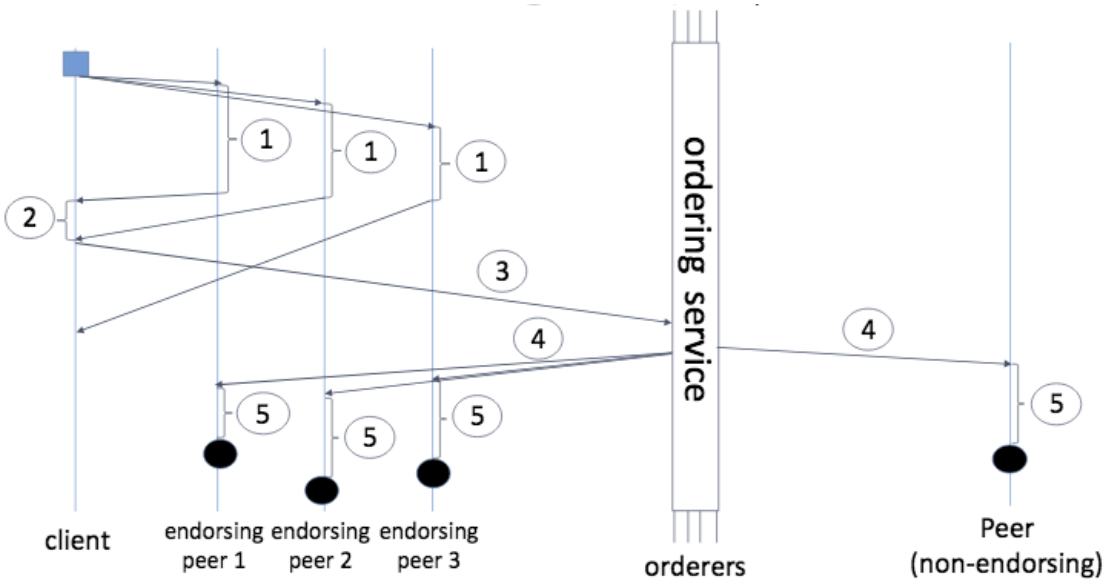

Figure 7. Fabric transaction flow. Blue square defines the moment of chaincode invocation and black circle commit to the ledger.

5. Implementation

This chapter is about the pet shop use case implementations done using Ethereum and Hyperledger Fabric. It explains how the use cases were implemented, what tools where used and what decisions were made.

5.1 Ethereum application

Ethereum application is following the same truffle framework tutorial as the use case itself. The source code for this specific implementation is publicly available on GitHub[7].

The application is running using the Truffle framework[8] that makes developing Ethereum applications easier providing smart contract compilation, deployment, binary management and automated testing which are effectively used while developing given application.

It is possible to develop applications right on the Ethereum blockchain, but as every transaction has its fee, it would be cheaper to test it locally before releasing it to the main Ethereum network. For that reason, we use the Ganache[9] as it provides us a virtual Ethereum blockchain with fake accounts for testing.

Since we are following the official Truffle tutorial, they have provided unfinished code for this project. It includes initial project structure and partial code for the user interface. It can be downloaded using the truffle *unbox pet-shop* command.

5.1.1 Writing the smart contracts

To be able to make changes to the ledger we need to write a smart contract. In Ethereum they can be written using many languages like Solidity, LLL, Vyper, etc. We chose Solidity in following reasons: it is used in the provided tutorial, it is the most popular and well documented.

To manage pet adoptions, we create the smart contract called Adoption (Figure 8), located at *contracts/Adoptions.sol*. The first line of the code defines the minimum version of Solidity and is used by the compiler. As the use case requirements include saving the transaction senders we define the variable adopters, an array of addresses where keys represent the pet ID's and values corresponding client addresses. The address is a special type, in Solidity,

[7] https://github.com/karlveskus/ethereum-pet-shop
[8] http://truffleframework.com/
[9] http://truffleframework.com/ganache/

that refers to Ethereum address and therefore stores a 20-byte value. As the variable is noted as public, it has automatic getter methods.

As we now have an array of addresses, we can store the transaction senders there. We create a function adopt, that takes pet ID as a parameter and returns it if adoption was successful. In meantime, it gets the sender's address using msg.sender and saves it to the adopters, using the index provided as a petId. Then, we create the getAdopters function that simply returns the adopters, an array of addresses, as the client needs to know, what pets are already adopted.

```
pragma solidity ^0.4.17;

contract Adoption {
    address[16] public adopters;

    // Adopting a pet
    function adopt(uint petId) public returns (uint) {
        require(petId >= 0 && petId <= 15);

        adopters[petId] = msg.sender;

        return petId;
    }

    // Retrieving the adopters
    function getAdopters() public view returns (address[16]) {
        return adopters;
    }
}
```

Figure 8. contracts/Adoption.sol – Smart contract for adoptions

5.1.2 Compiling and migrating smart contracts

Next, as Solidity is a compiled language, we have to compile our Solidity code to bytecode for Ethereum virtual machine to execute it [21]. To do that we use the *truffle compile* command provided by Truffle. Then, the compiled contracts have to be migrated to the blockchain. For that, we create a new migration script, 2_deploy_contracts.js (displayed in Figure 9), where we import and deploy previously compiled smart contract.

```
var Adoption = artifacts.require("Adoption");

module.exports = function(deployer) {
    deployer.deploy(Adoption);
};
```

Figure 9. migrations/2_deploy_contracts.js – Migration for smart contract

As mentioned, we use Ganache to run smart contracts on virtual Ethereum blockchain. After starting it, the local blockchain running on port 7545 will be generated. To migrate the contracts to the blockchain we run the *truffle migrate* command.

5.1.3 Front-end

As the use case requirements include building a web application where users can adopt pets by clicking on the button, we need to build a separate front-end application. For that, we use Bootstrap component library[10] to easily add styles for the user interface and web3 JavaScript library[11] to interact with the Ethereum blockchain. The front-end code is located at *src* folder and it contains CSS styles, fonts, pet images, JavaScript code for interactions and data about the pets.

The JavaScript code for the front-end is located at the *src/js/app.js*. It has a global object App with multiple functions and its init function invocation at the bottom. As init function is the first one that gets invoked, it loads all the data about pets from the pets.json and then uses provided DOM element, with id #petTemplate, as a template to insert all the data to the web page.

To make the front-end application working as required, we implement initWeb3, initContract, markAdopted and handleAdopt, as in Figure 10, by taking the following steps:

1. We create a initWeb3 function (Figure 10) that checks if web3 instance is already present. If yes, then we use its provider to create a new web3 object. Otherwise, we create the new web3 object using the local provider as a fall back to Ganache. The reason behind it is explained in depth in chapter 5.1.4.

```
26    initWeb3: function() {
27      // Is there an injected web3 instance?
28      if (typeof web3 !== 'undefined') {
29        App.web3Provider = web3.currentProvider;
30      } else {
31        // If no injected web3 instance is detected, fall back to Ganache
32        App.web3Provider = new Web3.providers.HttpProvider('http://localhost:7545');
33      }
34      web3 = new Web3(App.web3Provider);
35
36      return App.initContract();
37    },
```

Figure 10. initWeb3 function at the *src/js/app.js*

[10] https://getbootstrap.com/
[11] https://github.com/ethereum/web3.js/

2. We create an initContract function (Figure 11) that uses a truffle-contract library to instantiate the smart contract so we know where to find it later. For that, we first get the information about our contract. Then, we pass it to the TruffleContract function to interact with it and set its provider to App.web3Provider set in the previous step. After that, we call the markAdopted function to update our user interface described in the next step.

```
initContract: function() {
  $.getJSON('Adoption.json', function(data) {
    // Get the necessary contract artifact file and instantiate it with truffle-contract.
    var AdoptionArtifact = data;
    App.contracts.Adoption = TruffleContract(AdoptionArtifact);

    // Set the provider for our contract.
    App.contracts.Adoption.setProvider(App.web3Provider);

    // Use our contract to retieve and mark the adopted pets.
    return App.markAdopted();
  });

  return App.bindEvents();
},
```

Figure 11. initContract function at the *src/js/app.js*

3. We create a markAdopted function (Figure 12) to disable all the buttons where the pet is already adopted. For that, we access the Adoption contract and use a call function on its getAdopters function to read the data without making a new transaction. Then, we loop over all received adopters and disable every button where adopter address is already set. However, if the address is not set, it is equal to an address '0x0000000000000000000000000000000000000000'.

```
markAdopted: function(adopters, account) {
  var adoptionInstance;

  App.contracts.Adoption.deployed().then(function(instance) {
    adoptionInstance = instance;

    return adoptionInstance.getAdopters.call();
  }).then(function(adopters) {
    for (i = 0; i < adopters.length; i++) {
      if (adopters[i] !== '0x0000000000000000000000000000000000000000') {
        $('.panel-pet').eq(i).find('button').text('Success').attr('disabled', true);
      }
    }
  }).catch(function(err) {
    console.log(err.message);
  });

}
```

Figure 12. handleAdopt function at the *src/js/app.js*

4. We create a handleAdopt function (Figure 13) to handle adoption requests (for example "Adopt" button click). For that, we use web3.wth.getAccount function to get all the accounts and save the first on as our account. Then, we access the Adoption contract and use its adopt method to adopt a pet with an ID parsed from the click event. This time, the transaction is sent instead. In case of a successful result, we update the user interface using markAdopted function described in step 3.

```
handleAdopt: function() {
  event.preventDefault();

  var petId = parseInt($(event.target).data('id'));

  var adoptionInstance;

  web3.eth.getAccounts(function(error, accounts) {
    if (error) {
      console.log(error);
    }

    var account = accounts[0];

    App.contracts.Adoption.deployed().then(function(instance) {
      adoptionInstance = instance;

      return adoptionInstance.adopt(petId, {from: account});
    }).then(function(result) {
      return App.markAdopted();
    }).catch(function(err) {
      console.log(err.message);
    });
  });
},
```

Figure 13. markAdopted function at the *src/js/app.js*

5.1.4 Interacting with the application using a MetaMask

At this point, we can interact with a blockchain using created front-end application. We use the lite-server library[12] and start the server by using *npm run dev* command. After that, a new browser tab with a web application opens.

Next step is to connect our browser to Ethereum network. Unfortunately, at the moment of writing, regular browsers such as Chrome, Firefox, etc. does not work in such a way that it could be possible to interact with distributed applications (dapps) directly without running a full node on browser's machine. For the solution to this problem, there are multiple applications such as Mist browser[13], MetaMask[14] extension, etc. that provide a way to interact

[12] https://www.npmjs.com/package/lite-server
[13] https://github.com/ethereum/mist
[14] https://metamask.io/

with a network without running a full node. As the Chrome has the MetaMask extension and it is really easy to use, the application was tested using this.

5.2 Hyperledger Fabric application

The Hyperledger Fabric network code is based on the basic-network project from the official Hyperledger Fabric fabric-samples GitHub repository[15]. The client application, on the other hand, is based on the tuna-app application[16] used in the "Blockchain for Business - An Introduction to Hyperledger Technologies" course provided by the edX.org[17]. The source code for the Fabric implementation of the Pet shop in publicly available at GitHub[18].

We begin the implementation by providing a network with all the pre-defined participants in the configuration. Then, we create a chaincode to interact with the ledger. After this, we create a client application that acts as an end-user and therefore shows the pets and provides a way to adopt them.

To run every peer independently and download Fabric binaries we use Docker[19]. To get the latter, we run the bootstrap.sh script that pulls and tags docker images required for the project.

5.2.1 Network

To make the network suitable for our requirements, we need the network configuration file at the *network/docker-compose.yml* to contain at least certificate authority, orderer and peer. There we define container name, docker image and port for each of these services. As a result, running the *network/start.sh* script generates the docker containers, using the latter configuration, for each of the services. Addition to that, it also creates a channel and connects the peer to it.

5.2.2 Chaincode

To interact with a ledger, we have to create a chaincode. At the moment, Fabric supports chaincode written using Node.js and Go, but as the Go language is more popular and easier to use, we choose to write our chaincode in Go.

[15] https://github.com/hyperledger/fabric-samples
[16] https://github.com/hyperledger/education/tree/master/LFS171x/fabric-material/tuna-app
[17] https://courses.edx.org/courses/course-v1:LinuxFoundationX+LFS171x+3T2017/course/
[18] https://github.com/karlveskus/hyperledger-fabric-pet-shop
[19] https://www.docker.com/

We start out by creating the pet-shop chaincode at *chaincode/pet-shop/pet-shop.go* based on the *fabric-samples/chaincode/fabcar/go/fabcar.go* provided in the fabric-samples repository. There, we first import all the required libraries and define the SmartContract and Pet types as in Figure 14. To store all the data about the pet, we define it as a structs with fields for name, picture location, breed, location, age and its owner.

```go
package main

import (
    "bytes"
    "crypto/x509"
    "encoding/json"
    "encoding/pem"
    "fmt"
    "strconv"

    "github.com/hyperledger/fabric/core/chaincode/shim"
    sc "github.com/hyperledger/fabric/protos/peer"
)

type SmartContract struct {
}

type Pet struct {
    Name string `json:"name"`
    Picture string `json:"picture"`
    Breed string `json:"breed"`
    Location string `json:"location"`
    Age int `json:"age"`
    Owner string `json:"owner"`
}
```

Figure 14. Import libraries and define types for SmartContract and Pet at the pet-shop chaincode at the *chaincode/pet-shop/pet-shop.go*

Then, we define init and invoke methods (Figure 15) for the chaincode. Init is the one, that gets called when the chaincode gets instantiated by the network and invoke method gets called as a result of client application request to run the chaincode. It has to be called with a function and arguments specified. Then, depending on a function name passed in, we call a new function.

```go
func (s *SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response {
    return shim.Success(nil)
}

/*…
func (s *SmartContract) Invoke(APIstub shim.ChaincodeStubInterface) sc.Response {

    // Retrieve the requested Smart Contract function and arguments
    function, args := APIstub.GetFunctionAndParameters()

    // Route to the appropriate handler function to interact with the ledger
    if function == "initLedger" {
        return s.initLedger(APIstub)
    } else if function == "queryAllPets" {
        return s.queryAllPets(APIstub)
    } else if function == "adoptPet" {
        return s.adoptPet(APIstub, args)
    }

    return shim.Error("Invalid Smart Contract function name.")
}
```

Figure 15. Init and Invoke methods defined at the pet-shop chaincode at the *chaincode/pet-shop/pet-shop.go*

After the Invoke method is defined we add methods that are described inside it. We start with the initLedger (Figure 16) as it gets invoked right after the network and channel are set up to insert the initial data. It first defines the array of pets and then puts them into the state, one by one, using the PutState function. As a result, the state contains all the pets which keys represent the pet ID's and values corresponding pets saved as byte arrays.

```go
func (s *SmartContract) initLedger(APIstub shim.ChaincodeStubInterface) sc.Response {
    pets := []Pet{
        Pet{Name: "Frieda", Picture: "images/scottish-terrier.jpeg", Age: 3, Breed: "Scottish Terrier", Location: "Lisco, Alabama"},
        Pet{Name: "Gina", Picture: "images/scottish-terrier.jpeg", Age: 3, Breed: "Scottish Terrier", Location: "Tooleville, West Virgini
        Pet{Name: "Collins", Picture: "images/french-bulldog.jpeg", Age: 2, Breed: "French Bulldog", Location: "Freeburn, Idaho"},
        Pet{Name: "Melissa", Picture: "images/boxer.jpeg", Age: 2, Breed: "Boxer", Location: "Camas, Pennsylvania"},
        Pet{Name: "Jeanine", Picture: "images/french-bulldog.jpeg", Age: 2, Breed: "French Bulldog", Location: "Gerber, South Dakota"},
        Pet{Name: "Elvia", Picture: "images/french-bulldog.jpeg", Age: 3, Breed: "French Bulldog", Location: "Innsbrook, Illinois"},
        Pet{Name: "Latisha", Picture: "images/golden-retriever.jpeg", Age: 3, Breed: "Golden Retriever", Location: "Soudan, Louisiana"},
        Pet{Name: "Coleman", Picture: "images/golden-retriever.jpeg", Age: 3, Breed: "Golden Retriever", Location: "Jacksonwald, Palau"},
        Pet{Name: "Nichole", Picture: "images/french-bulldog.jpeg", Age: 2, Breed: "French Bulldog", Location: "Honolulu, Hawaii"},
        Pet{Name: "Fran", Picture: "images/boxer.jpeg", Age: 3, Breed: "Boxer", Location: "Matheny, Utah"},
        Pet{Name: "Leonor", Picture: "images/boxer.jpeg", Age: 2, Breed: "Boxer", Location: "Tyhee, Indiana"},
        Pet{Name: "Dean", Picture: "images/scottish-terrier.jpeg", Age: 3, Breed: "Scottish Terrier", Location: "Windsor, Montana"},
        Pet{Name: "Stevenson", Picture: "images/french-bulldog.jpeg", Age: 3, Breed: "French Bulldog", Location: "Kingstowne, Nevada"},
        Pet{Name: "Kristina", Picture: "images/golden-retriever.jpeg", Age: 4, Breed: "Golden Retriever", Location: "Sultana, Massachuset
        Pet{Name: "Ethel", Picture: "images/golden-retriever.jpeg", Age: 2, Breed: "Golden Retriever", Location: "Broadlands, Oregon"},
        Pet{Name: "Terry", Picture: "images/golden-retriever.jpeg", Age: 2, Breed: "Golden Retriever", Location: "Dawn, Wisconsin"},
    }

    i := 0
    for i < len(pets) {
        petAsBytes, _ := json.Marshal(pets[i])
        APIstub.PutState(strconv.Itoa(i+1), petAsBytes)
        i = i + 1
    }

    return shim.Success(nil)
}
```

Figure 16. initLedger method at the *chaincode/pet-shop/pet-shop.go*

Next, we create the adoptPet function (Figure 17) that takes in one parameter, a pet ID. If this condition is not met, the error will be returned. Using the pet ID, we read the pet information from the state using the GetState function. Then, we access the creator of the transaction using GetCreator function, parse its name from it and save it as a new pet owner. In the end, we save the updated pet data to the state. Again, in case of error, we return it with a corresponding description.

```
144 func (s *SmartContract) adoptPet(APIstub shim.ChaincodeStubInterface, args []string) sc.Response {
145
146     if len(args) != 1 {
147         return shim.Error("Incorrect number of arguments. Expecting 1")
148     }
149
150     petAsBytes, _ := APIstub.GetState(args[0])
151     if petAsBytes == nil {
152         return shim.Error("Could not locate pet")
153     }
154     pet := Pet{}
155
156     json.Unmarshal(petAsBytes, &pet)
157
158     // GetCreator returns marshaled serialized identity of the client
159     creator, err := APIstub.GetCreator()
160     if err != nil {
161         return shim.Error(fmt.Sprintf("Error received on GetCreator", err))
162     }
163     certStart := bytes.IndexAny(creator, "-----BEGIN CERTIFICATE-----")
164     if certStart == -1 {
165         return shim.Error("No certificate found")
166     }
167     certText := creator[certStart:]
168     block, _ := pem.Decode(certText)
169     if block == nil {
170         return shim.Error(fmt.Sprintf("Error received on pem.Decode of certificate",  certText))
171     }
172     ucert, err := x509.ParseCertificate(block.Bytes)
173     if err != nil {
174         return shim.Error(fmt.Sprintf("Error received on ParseCertificate", err))
175     }
176     pet.Owner = ucert.Subject.CommonName
177
178     petAsBytes, _ = json.Marshal(pet)
179     err = APIstub.PutState(args[0], petAsBytes)
180     if err != nil {
181         return shim.Error(fmt.Sprintf("Failed to change pet owner: %s", args[0]))
182     }
183
184     return shim.Success(nil)
185 }
```

Figure 17. adoptPet function at the *chaincode/pet-shop/pet-shop.go*

The last task is to implement the queryAllPets function (Figure 18) that simply returns all the pets. First, we define resultsIterator to loop over the pets in the state. Then we define the buffer to store the data and start looping. In every iteration, we create a string representing a JSON object with two properties, Key and Record, where Key refers to the pet ID and

Record to the corresponding pet data. Finally, we return generated list of objects as an array of bytes.

```go
func (s *SmartContract) queryAllPets(APIstub shim.ChaincodeStubInterface) sc.Response {

    startKey := "0"
    endKey := "999"

    resultsIterator, err := APIstub.GetStateByRange(startKey, endKey)
    if err != nil {
        return shim.Error(err.Error())
    }
    defer resultsIterator.Close()

    // buffer is a JSON array containing QueryResults
    var buffer bytes.Buffer
    buffer.WriteString("[")

    bArrayMemberAlreadyWritten := false
    for resultsIterator.HasNext() {
        queryResponse, err := resultsIterator.Next()
        if err != nil {
            return shim.Error(err.Error())
        }
        // Add comma before array members,suppress it for the first array member
        if bArrayMemberAlreadyWritten == true {
            buffer.WriteString(",")
        }
        buffer.WriteString("{\"Key\":")
        buffer.WriteString("\"")
        buffer.WriteString(queryResponse.Key)
        buffer.WriteString("\"")

        buffer.WriteString(", \"Record\":")
        // Record is a JSON object, so we write as-is
        buffer.WriteString(string(queryResponse.Value))
        buffer.WriteString("}")
        bArrayMemberAlreadyWritten = true
    }
    buffer.WriteString("]")

    fmt.Printf("- queryAllPets:\n%s\n", buffer.String())

    return shim.Success(buffer.Bytes())
}
```

Figure 18. queryAllPets function at the *chaincode/pet-shop/pet-shop.go*

To start the network, we use a *pet-shop/startFabric.sh* script that first executes the *network/start.sh* script, described in chapter 5.2.1, and therefore launches the network, creates a channel and joins the peer to this channel. Next, it launches the CLI container and uses it in order to install and instantiate pet-shop chaincode and invoke its initLedger method to insert the initial data about the pets to the ledger.

5.2.3 Client application

The client application is divided into two parts, server and front-end. In addition to that, there are also registerAdmin.js and registerUser.js scripts that register and enroll the client to get certificates that are necessary to interact with the network.

Server

As the Fabric requires every action in the network to be signed, we need a way to allow clients to be able to store and use their certificates in their own machine. For that reason, we use Node.js to build our server application, located at the *pet-shop/server.js*. It uses the Express.js framework[20], which is designed for building web applications. Its routes are defined at the *pet-shop/routes.js*. There, we define two endpoints, one for requesting an adoption and the second for querying all the pets. Both of them use the same controller, located at the *pet-shop/controller.js,* to connect with a Fabric network. This controller defines two methods, getAllPets and adoptPet, that both use fabric-client library[21] to make requests to the network. The library provides us a way to use locally stored certificates and send the query proposals to the peers.

Front-end

Front-end code is located in the *pet-shop/client* folder. It uses AngularJS[22] front-end framework to connect HTML and JavaScript code easily. The code at the *app.js* is structured similarly to the Ethereum front-end application described in chapter 5.1.3. There, we first define a controller that is responsible for the user interface and then an angular factory that reaches the endpoints defined in the server. We begin creating the controller by defining the init function that gets called at the moment when the HTML is loaded. It queries all the pets and then it inserts them into the web page. Next, we define the handleAdopt function, a click event listener, that uses the latter factory and thereby reaches the ledger to request for the adoption. MarkAdopted function is to disable buttons, where pets are already adopted.

[20] https://expressjs.com/
[21] https://github.com/hyperledger/fabric-sdk-node
[22] https://angularjs.org/

6. Comparison

This section compares Ethereum and Hyperledger Fabric platforms. Sections 6.1 - 6.5 explain the overall differences and then section 6.6 compares the specific implementations described in section 5.

6.1 Architecture

Ethereum and Hyperledger Fabric are designed keeping really different concerns in mind. As the Ethereum idea is to be the public blockchain for any kind of applications, it is designed to be permissionless and totally transparent. It means that all the data is stored in one shared ledger that everyone has access to.

Fabric, on the other hand, provides modular and flexible solutions for private permissioned blockchains to allow security and confidentiality. The latter is brought to the Fabric using channels that provide independent ledgers accessible only to its users. So, it is possible to create multiple channels and connect only some of the users to it. In this case, the ledger is private (cannot be accessed by non-registered users) and it is possible to share confidential data without all the network noticing it.

Another difference in the architecture is the currency. Because of the proof-of-work consensus algorithm used by Ethereum, it has its own crypto-currency. In contrast to that, Fabric does not have it.

6.2 Consensus Algorithm

Currently, Ethereum uses mining based proof-of-work consensus algorithm, where all the nodes agree upon a common truth and thereby the ledger. Fabric, in contrast, is modular, allowing different algorithms to be used. Fabric also has different types of nodes on the consensus mechanism and these nodes are pre-defined on the network setup.

6.3 Ecosystem

Since Ethereum was one of the first blockchain platforms after Bitcoin, it has gained a lot of popularity. With that, many organizations and developers have built many tools and frameworks to make developing applications easier. For example, Truffle framework adds command line interface through what it is easy to compile and test smart contracts. Then,

Ganache provides a virtual Ethereum network, with dummy accounts, that is created automatically without any setup. Fabric, on the other hand, has less if any tools besides Docker that simplify the whole development process.

A good example to illustrate it would be the simple network setup. In Ethereum, we just need to run the Ganache and its done. In Fabric, however, the easiest way would be to use sample-network code in GitHub and Docker to get it running. This is also the path we followed to create the network as easy as possible.

However, as Fabric supports many general-purpose programming languages such as Go and JavaScript, in contrast to Ethereum that supports less popular contract oriented languages, it is important to mention the number of libraries deployed. More specifically, Go and JavaScript are more established and therefore have a lot of libraries built for a variety of needs. Solidity, on the other hand, has a really limited set of libraries.

6.4 Main use cases

As Ethereum is public and totally transparent it could be effectively used for most of the ownership storages such as real estate and crypto-currency. Currently, it is possible to use Ethereum as a payment method in many ways such as grocery stores, coffee shop, online store, etc. Another good use case for Ethereum could be online-gambling to make it more transparent while removing the need for trusted third party.

Fabric, on the other hand, provides a way to store confidential data that is required for any supply chain. It allows many applications to be built on top of private blockchain, for example, electronic health records or insurance, where data cannot be shared across the network but it should be accessible for specific participants so it cannot be totally private.

6.5 Language

One of the biggest differences is definitely the language support. Ethereum supports languages that are specifically designed to be used for writing Ethereum smart contracts, such as Solidity and Vyper. Hyperledger Fabric, on the other hand, supports multiple popular programming languages such as Go and JavaScript.

This brings us to a situation, where it is possible to write chaincode in Fabric without learning a new language. But, as Solidity is contract-oriented and designed specifically for Ethereum, it can be more effective to use it.

6.6 Collateral case

The biggest difference between the Ethereum and Fabric implementation is its architecture. In case of Fabric, we built a client using a Node.js runtime to handles certificates that are required to interact with a network. Even the network part is completely different, since we had to manually create and setup the network in case of Fabric but not for Ethereum. For a result, Fabric application codebase is much bigger and more complex than Ethereum.

Smart contracts were deployed and managed in similar way but creating the Ethereum smart contract took way less lines of code than Fabric while having the same methods. In case of Ethereum, it took 19 lines of code, in contrast to Fabric where it took almost 200. However, all the requirements were covered using one smart contract in both platform.

We also found out, that client libraries that are used in to connect to the network are similar.

7. Conclusion

In this thesis, an overview of distributed ledger and blockchain technology was given. The goal of this thesis was to compare Ethereum and Hyperledger Fabric theoretically and then practically using implemented applications.

We have managed to find out, that as Ethereum is public blockchain and therefore have all the data public, it suits well for applications that are designed to interact with a whole world such as insurance and peer-to-peer gambling. Hyperledger Fabric, on the other hand, is designed for private use cases such as supply chain, where every chain participant should have data only relevant for him. For example, it allows possibility to sell goods with different prices without participants knowing about these deals.

We implemented same use case with Ethereum and Hyperledger Fabric. As a result, we demonstrated how to build applications on top of these platforms and found out, that as Ethereum is the most popular framework, its ecosystem is rich in different development tools but the supported languages are not that well-established and are thereby really limited. Hyperledger Fabric, in contrast, has only crucial tools but it supports popular languages that have already a lot of libraries to make development easier.

The comparison was done using rather simple solutions, implementing one smart contract and two methods for each of them. For the future work on the comparison, more complex use cases can be chosen. For example, an application that requires multiple smart contracts to be written for any asset management, not just storing but transferring ownership as well.

8. References

[1] S. Baghla, "Origin of Bitcoin: A brief history from 2008 crisis to present times," *Analytics India*, 2017. [Online]. Available: https://analyticsindiamag.com/origin-bitcoin-brief-history/. [Accessed: 18-Mar-2018].

[2] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[3] V. Gupta, "A Brief History of Blockchain," *Harvard Business Review*, 2017. [Online]. Available: https://hbr.org/2017/02/a-brief-history-of-blockchain?referral=03758&cm_vc=rr_item_page.top_right. [Accessed: 06-Apr-2018].

[4] M. Walport, "Distributed ledger technology: Beyond block chain," 2015.

[5] M. Valenta and P. Sandner, "Comparison of Ethereum, Hyperledger Fabric and Corda," 2017.

[6] N. Singh, "Ethereum or Hyperledger Fabric?," *Medium*, 2018. [Online]. Available: https://medium.com/quillhash/ethereum-or-hyperledger-fabric-259f3c9b8da6. [Accessed: 02-May-2018].

[7] N. Harish, S. Krause, and H. Gradstein, "Distributed Ledger Technology (DLT) and Blockchain," 2017.

[8] H.-H. Buerger, "Ethereum White Paper," *Github.Com*, 2016. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper. [Accessed: 18-Feb-2018].

[9] A. Hertig, "What is a Decentralized Application?," *Coindesk*, 2017. [Online]. Available: https://www.coindesk.com/information/what-is-a-decentralized-application-dapp/. [Accessed: 14-May-2018].

[10] P. Sajana, M. Sindhu, and M. Sethumadhavan, "On Blockchain Applications: HyperledgerFabricAnd Ethereum," *Int. J. Pure Appl. Math.*, vol. 118, pp. 2965–2970, 2018.

[11] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, "Performance Analysis of Private Blockchain Platforms in Varying Workloads," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, 2017, pp. 1–6.

[12] D. T. T. Anh, M. Zhang, B. C. Ooi, and G. Chen, "Untangling Blockchain: A Data Processing View of Blockchain Systems," *IEEE Trans. Knowl. Data Eng.*, pp. 1–1, 2018.

[13] P. Bajpai, "Bitcoin Vs Ethereum : Driven by Different Purposes," *Investopedia*, 2018. [Online]. Available: https://www.investopedia.com/articles/investing/031416/bitcoin-vs-ethereum-driven-different-purposes.asp. [Accessed: 02-May-2018].

[14] C. Jagers, "What is Ethereum?," *Investopedia*, 2017. [Online]. Available: https://www.investopedia.com/articles/investing/022516/what-ethereum.asp. [Accessed: 02-May-2018].

[15] Y. N. Aung and T. Tantidham, "Review of Ethereum: Smart Home Case Study," 2017.

[16] D. Vujičić, D. Jagodić, and S. Ranđić, "Blockchain Technology, Bitcoin, and Ethereum: A Brief Overview," 2018.

[17] “Hyperledger Fabric documentation, Introduction.” [Online]. Available: http://hyperledger-fabric.readthedocs.io/en/release-1.1/membership/membership.html. [Accessed: 09-May-2018].

[18] “Hyperledger Fabric documentation, Channels.” [Online]. Available: http://hyperledger-fabric.readthedocs.io/en/release-1.1/channels.html. [Accessed: 06-May-2018].

[19] “Hyperledger Fabric documentation, Chaincode.” [Online]. Available: http://hyperledger-fabric.readthedocs.io/en/release-1.1/membership/membership.html. [Accessed: 09-May-2018].

[20] “Hyperledger Fabric documentation, Architecture Explained.” [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-1.1/arch-deep-dive.html. [Accessed: 06-May-2018].

[21] “Ethereum Pet Shop.” [Online]. Available: http://truffleframework.com/tutorials/pet-shop. [Accessed: 02-May-2018].

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Karl Veskus**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

   of my thesis

   Ethereum versus Fabric - A Comparative Analysis,

   supervised by Fredrik Payman Milani and Luciano García-Bañuelos

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **14.05.2018**