Use Cases and Design of Web Applications in Decentralized Finance

Johannes Hüsers

BACHELORARBEIT

eingereicht am
Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Februar 2021

Advisor:

DI Martin Harrer

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere. This printed copy is identical to the submitted electronic version.

Hagenberg, February 1, 2021

Johannes Hüsers

Abstract

—0.5 to 1 page—

Kurzfassung

—0.5 to 1 page—

Contents

Chapter 1

Introduction

1.1 Motivation

—0.5 pages—

1.2 Goals

—0.5 pages—

1.3 Structure of the Thesis

—0.25 pages—

Chapter 2

Fundamentals

2.1 Cryptography

The term “cryptocurrency” already suggests, that blockchains rely heavily on cryptography. But cryptography is an even broader topic and exceeds by far the ranges of blockchains, cryptocurrencies and decentralization. Without crytography, our digital life would be very different today. Cryptography is the theory of data encryption, which means “secret writing” in Greek. Together with cryptoanalysis, which is the opposite (decoding data), cryptographic can be conflated as cryptology. Cryptography makes it possible to make data content unreadable for specific people, detect unauthorized data manipulation and guarantee the authenticity of a communication parter. What cryptography cannot achieve, is prevention of unauthorized data manipulation and the prevention that data is being read at all.

2.1.1 Public Key Cryptography

Public key cryptography are asymmetrical encryption methods that utilize both private and public keys. While symmetrical enryption methods such as the Data Encryption Standard (DES) and Advanced Encryption Standard (AES) use a single key to encrypt and decrypt a message, asymmetrical encryption methods use the public key of the recipient to decrypt the message and a private key to enrypt it. At the first glance it may seem that asymmetrical enryption methods are way better than symmetrical encryption methods and that they should be used all the time. But that is not the case. Asymmetrical encryption methods are less secure, more complex, more difficult to invent and slower than symmetrical encryption methods. Each type of encryption method has its own use cases so they actually complement each other. While symmetrical encryption methods are really good at encrypting data content, asymmetrical enryption methods are used to exchange the symmetrical keys and prove ownership and authenticity through digital signatures.

The most popular asymmetric enryption methods are RSA (Rivest, Shamii, Adelman), Diffie-Hellman and elliptic curve cryptography. RSA and Diffie-Hellman are the most common algorithms outside of the blockchain ecosystem and are used for example in banking, telecommunications and ecommerce. Because those are not used by blockchain technologies, we will focus on elliptic curve cryptography.

2.1.2 Elliptic Curve Cryptography

— 0.5 pages —

2.1.3 Hashing Functions

A hashing function maps a string of arbitrary size to a value of fixed size. But not all hashing functions are suitable for cryptographic problems. A good hashing function aims to have as few collisions as possible. A collision occurs when two different input strings lead to the same output hash. In order to prevent that as good as possible, each hash value should occur equally and a minimal modification of the input string should lead to a completely new hash value. This is important because the more collisions occur, the more vulnerable is the hashing function for hackers. If a hacker finds a string that generates the same hash value he can apply the real signature to his faked text [7].

— ethereum hash functions —

2.2 The Ethereum Blockchain

2.2.1 Blockchain Fundamentals

— 0.5 pages —

2.2.2 Clients

— 0.5 pages —

2.2.3 Wallets

— 0.5 pages —

2.2.4 Transactions

— 0.5 pages —

2.3 Smart Contracts

– 0.5 pages —

2.4 Decentralized Finance

— 0.5 pages —

2.4.1 State of the Art

— 0.25 pages — current applications, relevance on the market, technologies, ...

Chapter 3

Use Cases of Decentralized Finance

This chapter aims to introduce the five most relevant use cases of Decentralized Finance. Each use case is built on top of the previous one and rises in complexity. For example storing value in digital systems is quite easy nowadays but moving real physical assets such as gold or real estate into a blockchain is still pretty demanding. Note, that each type of financial service is not specific to Decentralized Finance, which means that they are applicable to every financial environment.

3.1 Classification of Financial Services

The digitalization changed the financial industry. A few years ago, a lot of new companies with a significant online presence[1], entered the market and changed the way how people interact with financial services. And now, the industry is experiencing again a new shift to more and more decentralized platforms and applications. But even tough the interaction interface changes from time to time, the underlying types of services we use in the financial sector stay the same. The fundamental service that is necessary to build a financial ecosystem is some kind of value storage. Once that is created, it is logical to have the ability to transfer this value to other people in order to compensate them for a product or service. The most popular and dominating use case of banks, which is lending builds on top of that. Once the ecosystem gets more mature, new types of value storage are introduced and people need a way to exchange two value storages for each other.

The last and the most sophisticated use case occurs, when people try to map complex real world examples in that financial ecosystem, such as securities, insurances and derivatives. In 2021, Decentralized Finance is about to enter exactly that use case. On a blockchain, this is called tokenization. A token is a new value store that is built on top of another cryptocurrency. At the time of writing this in January 2021, Chainlink[15] is the token with the highest market capitalization of $9 Billion[31] with the purpose of connecting real world data with the Ethereum blockchain[2].

[1] often referred as fintech

[2] known as oracle networks

3.2 Store of Value

No matter of the financial product or service, they all need one essential thing. In fact, our entire economic system depends on it: a currency, or in simpler terms, something to store value. Storing value is a fundamental trait of money alongside with being exchangeable and having a unit of account. The last two characteristics are comparatively easy to accomplish in the online world. Storing value in a digitalized way without having a central trusted authority, however, was a similar challenge as the people had to face in the early days when they started to switch from bartering to a real currency. A common consensus needs to be created where everyone can easily verify that a specific piece of money is authentic and wasn't created illegally. Even further, each person that uses this money needs a proof that it is storing real value. This wasn't possible until Satoshi Nakamoto, an anonymous person or group, published a specification [6] in 2008, on how digital money could be implemented. The first cryptocurrency was founded: Bitcoin.

3.2.1 Proof of Work

Bitcoin uses a mechanism called proof-of-work in order to give each block its value. This utilizes a cost-function which is designed to be easily verifiable but quite expensive to compute [3]. In order to create a new block, a value needs to be found that matches with the correct number of zero bits of the block hash. Once the correct solution has been found, the new block represents real value, because operating this computationally intensive task on a CPU costs a lot of electricity. The integrity of this block is guaranteed as well, because in order to change the history of the blockchain, each changed block needs to be re-computed. The longest chain of blocks on the network is accepted as the "truth", so if dishonest people want to cheat and change the history of the blockchain, they would need to create the longest chain of blocks which can be only achieved by having more than 50% of the computing power of the whole network. While these so-called 51% attacks are a threat on blockchains with a smaller number of network members, it is very unlikely to happen on a well established network like the Bitcoin blockchain [9]. The concept of proof-of-work in order to achive consensus in a decentralized manner became very popular and can be used in other areas of application too, such as elections, lotteries, asset registries, digital notarization and more [1].

3.2.2 Proof of Stake

A different approach to storing digital value is proof-of-stake. Instead of making the participants of the network solve computational expensive problems, the consensus is created by proving that someone owns specific funds. A blockchain that uses a proof-of-stake algorithm to achieve consensus has a set of validators. These validators are running a master node which gives them the opportunity to vote. In order to do so, the validators need to prove that they own a specific amount of funds. This is done by sending a special transaction, which locks away their currencies for a specific time. If the validators are honest and their vote corresponds to the result of the majority, they will get their funds back and an additional reward proportional to their deposited stake. If their vote gets rejected, the dishonest validator risks loosing his money [2].

3.2.3 Switching the Consensus Algorithm

At the time of writing this in December 2020, the Ethereum blockchain uses a similar proof-of-work algorithm like Bitcoin, called *Ethash*. However, Ethereum is about to switch to a proof-of-stake algorithm called *Casper* in the near future [26]. Algorithms that are based on proof-of-stake have the big advantage of less energy consumption because they are not focused on computational intensive tasks. The Bitcoin network, for example, has an annual electrical energy consumption of approximately 72.18 terawatt hours which is comparable to the consumption of Austria [25]. But proof-of-stake comes also with a caveat. Implementing incentives on a voting system which are based on the amount each validator is willing to stake means that the rich validators get richer and the poor validators stay poor.

The process of changing a consensus algorithm of a blockchain is not easy to accomplish. Because the network is decentralized, there is no single entity which can force all members to change the algorithm from proof-of-work to proof-of-stake. Just letting the people choose what they prefer to use is also not a good idea, because it is very unlikely that all agree on one type of algorithm. That would probably trigger a hard fork of the blockchain resulting in two separate networks, similar to what happened to Bitcoin in August 2017.

Bitcoin's network is by design very slow in verifying transactions. A block which is mined approximately every 10 minutes has a size of about 1 MB. Due to the small block size, Bitcoin is only capable of processing 7 transactions per second. Some people wished for a Bitcoin network which is more suitable for day to day payments. That is why *Bitcoin Cash* was created, emerged out of a hard fork of the Bitcoin network [24] [5]. In order to prevent a hard fork on Ethereum due to the transition from proof-of-work to proof-of-stake, the Ethereum protocol has a built in mechanism called *difficulty bomb* which makes it more difficult over time to be profitable with mining[3] by increasing the size of the problem to solve [11]. This ensures a smooth transition of all members to switch to proof-of-stake.

For the majority of users on the blockchain which are neither miners nor validators it does not matter which consensus algorithm is being used. In fact, it is not even important when developing decentralized applications based on smart contracts like it is done in chapter 4. The only thing that makes an impact is, that the blockchain is able to store real value on the network by reaching consensus by its members.

3.3 Payments

3.3.1 Bitcoin

When thinking about a cryptocurrency that can be used for payments, usually Bitcoin is the one that comes first into mind. And that is justified. Bitcoin was the first cryptocurrency and has by far the highest market capitalization. But when it comes to payments, Bitcoin is a really poor choice.

Bitcoin is too volatile. It is not uncommon that the price of Bitcoin fluctuates a few thousand dollars per day. Merchants who try to sell their products with Bitcoin have a

[3] used synonymously to proof-of-work

really hard time if the revenue of yesterday is only worth the half today. There are a lot of reasons because Bitcoin is so volatile and that will not change in the near future.

Bitcoin is too slow. The block time on the Bitcoin blockchain is approximately 10 minutes. Transactions take a long time to be verified by the network. Too long for a cashier to wait for a proof that his customer made a correct transaction.

Bitcoin is too deflationary. By design, Bitcoin has a fixed supply of 21 million. As Bitcoin gets more popular over time, it is only logical that its price increases. While this is a pleasant thing for investors, it is an unwanted sideeffect for a financial economy, because everyone refuses to spend their coins as they increase in value.

3.3.2 Centralized Stable Coins

There are a few strategies to remove the volatility from a digital asset. The most popular way is by pegging a cryptocurrency or token to a reference fiat currency like the US dollar. This is achieved by collateralizing every unit of the digital asset with a unit in the reference currency. This works pretty well, but the problem is, that this is a quite centralized approach, as the collateralization is managed by a central authority. However, this is a very common way and the biggest stable coins such as Tether[30], USD Coin[33] and Gemini Dollar[27] operate using collateralization.

3.3.3 Dynamic Supply Protocols

The price of an asset is driven by supply and demand. If the supply is larger than the demand, the price of this asset falls and vice versa. Most assets have a fixed supply, or at least a fixed distribution schedule, which gives the market forces full control over the price. But if the supply is being controlled, the price can be controlled as well. Dynamic supply protocols do this by increasing the supply if the demand increases and take out supply if the market is experiencing a reduced demand. This results in an almost stable price. A token that was one of the first that implemented such a dynamic supply protocol is Ampleforth[13]. As the demand increases, the amount of coins in each wallet will increase as well, while keeping the same proportions of the market share.

3.3.4 Stability Protocols

Although dynamic supply protocols work pretty well, it can be really confusing for users, if the amount of that asset in their wallet fluctuates on a daily basis. Stability protocols take a different approach by controlling the price through incentives instead of changing the supply. If the demand increases, the protocol makes it less attractive to buy that asset and rewards users that buy when the demand is low. The bigger the market capitalization of that asset, the more accurate is the price. Decentralized exchanges work in a similar way as described later in section 3.5.2. A common token using stability protocols is Maker[29], which is by far the project with the most value locked[4] in Decentralized Finance.

[4] $4.5 Billion (as of January 2021, https://defipulse.com/)

3.4 Lending & Borrowing

3.4.1 Collateralized Lending

Lending and borrowing is the main business model of banks in traditional finance. In order to assess their risks, banks force their customers to do an identification. Therefore, it is a lot easier to evaluate the probability to receive their funds back. And even if they don't, they can initiate legal ways to get their money back. In Decentralized Finance, this is a lot more difficult. All users are anonymous and not regulated in any way. The network needs to find new mechanisms to make sure, the users, behave as expected. Section 3.2 introduces two approaches to make it more attractive for users to play by the rules: incentives and punishments. The system of incentivization is used in the consensus protocol Proof of Work, where people receive a reward if they help to support the network. A punishment is used in Proof of Stake, as people will loose their collateral, if they vote for the wrong blocks. This system is used as well in lending applications. In order to borrow a specific asset, the user needs to deposit a collateral first, before he can withdraw the loan. If specific conditions are met, e.g. the price of the collateralized asset falls below a certain value, the user will get liquidated and looses his collateral.

While this system is the most popular right now, it only makes sense for users if they think that their collateral will grow in value as time goes by, otherwise they would just swap their collateral in the asset they need. Most cryptocurrencies are deflationary by nature, so they are supposed to increase in value. Stablecoins, however, such as Tether or Dai, are designed to have a constant value as good as possible. Therefore, collateralized lending works best with a deflationary asset and a stablecoin. This is exactly what Maker[29], the most popular lending project does, using Ether and Dai.

3.4.2 Alternative Lending Techniques

— 0.25 pages —

3.5 Exchanging

3.5.1 Characteristics of Decentralized Exchanges

With the arise of many new fintech enterprises in the last few years, online exchanges started to grew in importance on the market as well. While they all were very centralized in the first place, Decentralized Finance made it possible to establish new ways of exchanging money. Although each online exchange may behave very different, they can be all categorized into three big types when it comes to their structure: Centralized, decentralized and non-custodial exchanges. Centralized exchanges (CEXes) are the traditional approach where all the power is centralized to one specific organization. Decentralized exchanges (DEXes), however, are the complete opposite where there is no single entity in control and decisions are made completely based on Smart Contracts. Non-custodial exchanges are somewhere inbetween but are often confused with decentralized exchanges. It is important to note, that none of the types are superior in comparison to the others. Each type of exchange has its own benefits and risks. In order

to assess whether an exchange is suitable for a specific use case, it is crucial to know how to classify it.

Probably the most apparent indicator is the ownership of the private keys. On CEXes, the assets are coffered for the user. If someone buys Bitcoin on a centralized exchange, private keys are never an issue. On non-custodial and decentralized exchanges you have to manage your private keys by yourself. While this is usually a good thing, because you don't have to trust someone else[5], it comes also with the risk of losing all your assets if you forget your private key, since nobody can restore it for you.

Owning your private keys is a good indicator but by far not the only trait a decentralized exchange needs to have. When looking at the technical infrastructure of the exchange, many things could be structured in a centralized manner. While the code of a centralized exchange is usually proprietary, a DEX needs to have code that is licensed exclusively with an open-source licence, making it possible for anyone to fork the project in case the service is no longer available. That is also highly related to emergencies. How does an exchange react to bad or unexpected events such as hacking attacks? CEXes have the option to put their service temporarly offline until the issue is resolved, in order to save the funds of the users. On decentralized exchanges, there is no such thing and assets may be lost forever.

Another crucial topic to consider is censorship and geo blocking. If the service is not available in the whole world, it is not a DEX. Centralized exchanges always decide which coins are listed on the exchange, which is usually a pay to play model. Decentralized exchanges don't have such regulation. Anyone can add a new pair of coins to swap on a DEX, which often leads to a tremendously high amount of unknown and irrelevant listed coins.

The last key indicator on how to categorize an online exchange is the way how the order matching and the settlement works. Centralized exchanges use the same algorithm as stock exchanges, which utilizes an order book. Buy and sell orders are listed in a centralized ledger and the settlement happens when two orders match. Non-custodial exchanges usually use price oracles which try to get price information [12, p. 47]. Truly decentralized exchanges even go a step further and solve this problem without a third party price oracle. A DEX uses a concept that relies on Liquidity Pools, which will be discussed in detail in the next section.

3.5.2 Liquidity Pools

Every decentralized exchange needs liquidity. Because there is no order book, assets have to be already on the exchange before the user wants to swap two coins. In order to achieve that, the DEX depends on Liquidity Providers, which add their funds to the exchange. Liquidity Providers get rewarded[6] by the DEX because nobody provides liquidity for free. The rewards can be taken from the exchange fees, which are usually still lower than on CEXes. For example, Uniswap, one of the biggest decentralized exchanges rewards each Liquidity Provider with 0.3% of each transaction taking place on that Liquidity Pool proportional to their share of the pool [32]. If there is a yearly transaction volume on the pair Ether/Tether USD of $17.7 billion and the Liquidity

[5] often referred to as "not your keys, not your coins"

[6] known as Liquidity Mining

Provider owns a 0.1% share of this pool, he will be rewarded with $53.100 per year. The share is usually expressed through Pool Tokens, which can be traded as well.

Liquidity Providers can only provide liquidity in pairs. If someone wants to provide liquidity to the WBTC-ETH pair, which is Wrapped Bitcoin[7] and Ether, he needs to add the same value both in WBTC and ETH. People who want to swap their WBTC to ETH add only one asset of that pair and receive the other one, which creates an imbalance in that specific pool. The total value stays the same but the proportion of the assets change. In this example, there is more Ether compared to Wrapped Bitcoin after the swap. This makes it very attractive for the market to swap ETH back to WBTC, because the user will get proportional more WBTC for less ETH[8]. In high-volume markets this imbalance will be exploited very soon and the original proportion is restored.

If a Liquidity Pool of a specific trading pair is very small, there is a risk of clearing the entire pool by a single transaction, which would give away the associated asset almost for free in the next transaction. In order to prevent that, a user never gets the entire value in the swapped asset. The difference between the expected and the actual value is called slippage. That might seem annoying in the first place, but it is a crucial instrument for the exchange to work properly. This happens on exchanges with an order book too, because it is very unlikely that both a buy and a sell order have the exact same price. The deeper[9] the order book or the Liquidity Pool, the lower the percentage of the price slippage. If someone makes a high volume order on a small order book or Liquidity Pool, he will experience high slippage because his order wipes out a lot of opposite orders on a CEX and uses a high pool percentage on a DEX [28].

3.5.3 Risks

While providing liquidity to an exchange pool might look very profitable at first, there are also certain risks the Liquidity Provider has to deal with. Smart Contract risk is something which you are automatically exposed to when trusting software instead of people. Because software is still written by people and people make mistakes, there is no guarantee that the Smart Contract on a DEX works completely the way it should be. On a turing complete network such as Ethereum with programming languages like Solidity the risk is even higher because you can literally do anything in a Smart Contract. Because a Smart Contract is deployed only once, it is impossible to fix errors once it is public on a blockchain. This risk is far less present on networks which are not turing complete such as Bitcoin.

Even if the Smart Contract risk is eliminated as good as possible by writing a lot of tests, there is still the project risk, that the developers might deliberately implement errors and backdoors into their code in order to disburse liquidity to themselves. This is often the case when the project has poor auditing and if the founders are not known to the public.

Impermanent Loss is the last and most present risk liquidity providers are exposed to. The explanation of a Liquidity Pool in the last section is missing an important factor.

[7] Bitcoin as a token on the Ethereum network

[8] this phenomenon is called arbitrage and applies to all efficient markets

[9] in terms of more orders and more liquidity

Even if there is no one trading an asset pair, an imbalance can still occur because each asset is volatile by itself unless the asset pair consists of two stable coins pegged to the same fiat currency. Impermanent Loss always exists if one of the two assets has large price fluctuations in a short amount of time. And while it is usually good to diversify an investment, Impermanent Loss has an even greater impact on non-correlating assets, because the price difference can be much higher. The only way to reduce the risk of Impermanent Loss is to have a larger time horizon where you can choose a good time to cash out the liquidity. Unfortunately there is no such thing as Impermanent Win because it is always a loss once one of the two assets fluctuates. The loss is expressed through the loss in the Liquidity Pool compared to the price gains by holding each asset separately.

3.6 Investing

3.6.1 Tokens

—0.5 pages—

3.6.2 Derivatives

—0.5 pages—

3.6.3 Synthetic Assets

—0.5 pages—

Chapter 4

Design and Architecture

This chapter builds on top of the concepts and use cases of the previous chapters and lays out the architecture and design of a typical decentralized application by using a more practical approach. After a short introduction into the used technologies there will be a detailed view on each important part of a modern decentralized web application. To keep this chapter more concise, only selected code snippets will be shown to describe certain aspects. However, the entire code base of the working application can be found on Github [16].

4.1 Technology Stack

4.1.1 Solidity

The main components of decentralized web applications are Smart Contracts. Smart Contracts can be written in a few different programming languages such as LLL, Serpent, Solidity, Vyper and Bamboo. Solidity is by far the most popular and currently the de-facto standard in writing Smart Contracts [2]. Solidity is statically typed, has a JavaScript-like syntax and supports various popular programming concepts such as libraries and inheritance [8]. Because there are no classes in Solidity, it is considered not as an object-oriented but as a contract-oriented language [20].

4.1.2 Truffle

Truffle is a development environment which helps to develop decentralized applications on the Ethereum Virtual Machine [21]. It handles all the difficult parts starting from Smart Contract compilation, testing, linking, deployment and continuous integration. When developing more than just some simple Smart Contracts, Truffle is an important tool to save a lot of time.

4.1.3 Ganache

Because testing a decentralized application on the main Ethereum network would not be a good idea (and is expensive too), some sort of testing blockchain is needed. Ganache is a local blockchain for developers which provides everything to test a decentralized

applications on the local machine [17]. It offers accounts with fake ether and works really well with Truffle, which is built by the same developers.

4.1.4 Web3

In order to link Smart Contracts to a graphical user interface such as a web client application, web3.js comes into play. It is a JavaScript library and provides all the necessary APIs in order to connect a blockchain backend to a web frontend [23]. It also manages connections to specific wallet providers such as MetaMask.

4.1.5 Mocha & Chai

Testing plays in blockchain development an even bigger role than in software development in general, because deployed Smart Contracts are immutable and bugs literally cost money. A flawed function with causes an endless loop whould burn the entire gas that was available for that function call without doing anything useful. Mocha is a JavaScript testing framework which makes it possible to test the functionality of each Smart Contract in a structured and automated way [18]. Together with the assertion library Chai [14] and the integration into Truffle, Mocha releases its full potential as a fully fledged blockchain testing environment.

4.1.6 React

What would be the best Smart Contract without a graphical user interfaces which gives people the opportunity to interact with the blockchain in an easy and intuitive way. React is a JavaScript library for building user interfaces for the web and by far the most popular web framework out there [19]. React is super fast because it uses a virtual DOM and provides the user a modern and intuitive experience.

4.1.7 TypeScript

While other frontend frameworks such as Angular come with TypeScript out of the box, you have to add additional dependencies in React yourself. When it comes to larger projects or whenever bugs could become expensive, it is a good idea to introduce some kind of type checking in JavaScript. TypeScript is a superset of JavaScript and helps to detect errors before they occur at runtime [22]. In financial applications like this, there is usually an even higher focus on reliable, bug-free code.

4.2 The SafeMath Library

Arithmetic operations in Solidity wrap on overflow. This can lead to bugs, because most programmers assume, that it would raise an exception when an overflow occurs, as it is done in other high level programming languages. By introducing the library `SafeMath`, an entire class of bugs is being eliminated so it should be used everywhere possible.

Program 4.1: The SafeMath library.

```
1 library SafeMath {
2   function sub(uint256 _a, uint256 _b) internal pure returns (uint256) {
3     assert(_b <= _a);
4     return _a - _b;
5   }
6
7   function add(uint256 _a, uint256 _b) internal pure returns (uint256) {
8     uint256 c = _a + _b;
9     assert(c >= _a);
10    return c;
11  }
12 }
```

4.3 The ERC20 Token Standard

The ERC20 standard makes it possible to build standardized tokens on the Ethereum protocol which can be integrated into the existing ecosystem [10]. This is achieved through a common interface each token needs to implement. The standard was initially introduced in 2015 as an Ethereum Request for Comments (ERC). It received its name due to the automatic issue assignment on Github, which had the issue number 20 [2] [4].

4.3.1 Example Implementation

The function `totalSupply` returns the maximum amount available of this token in *wei*, the smallest unit possible. The supply can be either static or dynamic. In this case, it is static assigned through the constructor when the smart Contract is created. The return type is a 256 bit unsigned integer which is the most common data type in Solidity.

```
1 uint256 private _totalSupply;
2
3 function totalSupply() public view returns (uint256) {
4   return _totalSupply;
5 }
```

Each token is responsible to keep track of all addresses holding that token. This can be stored using the data type `mapping` which is a key value pair of addresses and token amounts. The function `balanceOf` returns the token balance for a specific address.

```
1 mapping(address => uint256) private balances;
2
3 function balanceOf(address _owner) public view returns (uint256 balance) {
4   return balances[_owner];
5 }
```

The functionality of sending tokens to a specific address is implemented in the `transfer` function. It takes an address and an amount as parameters and returns a boolean if the transaction was successful. The `require` function checks for a specific condition to be true. If this is not the case, the function will be reverted and all the

gas goes back to the sender, which can be evaluated in Solidity using `msg.sender`. In order to prevent over- or underflow errors when performing additions and subtractions, a small library `SafeMath` 4.1 is being utilized. If everything worked well, the `Transfer` event will be emitted.

```
function transfer(address _to, uint256 _value) public returns (bool success) {
  require(_value <= balances[msg.sender]);
  balances[msg.sender] = balances[msg.sender].sub(_value);
  balances[_to] = balances[_to].add(_value);
  emit Transfer(msg.sender, _to, _value);
  return true;
}
```

The function `transferFrom` is very similar to `transfer` with the only difference that the caller is not the one who sends the tokens. This is usually not a physical person but rather a Smart Contract of a decentralized application. Tokens can be sent only on behalf of someone else if the sender accepts the amount first. All funds that are already released for transactions are stored in another mapping specifying for each address which addresses are enabled to send what amount.

```
mapping(address => mapping(address => uint256)) private allowed;

function transferFrom(address _from, address _to, uint256 _value) public returns (
    bool success) {
  require(_value <= balances[_from]);
  require(_value <= allowed[_from][msg.sender]);
  balances[_from] = balances[_from].sub(_value);
  allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
  balances[_to] = balances[_to].add(_value);
  emit Transfer(_from, _to, _value);
  return true;
}
```

If the caller is not the sender of a transaction, funds need to be approved first. That is exactly what the function `approve` does. It authorizes the spender to spend a specific amount of the tokens of the address which called the function. Old funds that are already authorized will not be accumulated but overwritten. This makes it possible to revert an approval by calling the `approve` function another time with the value of zero. This function emits the `Approval` event.

```
function approve(address _spender, uint256 _value) public returns (bool success) {
  allowed[msg.sender][_spender] = _value;
  emit Approval(msg.sender, _spender, _value);
  return true;
}
```

Because the data structure holding the allowed funds is usually private, a public interface is needed to find out how much remaining tokens can be spent on behalf of someone else. The function `allowance` takes two parameters and returns the amount of tokens that the spender is authorized to spend from the owner's address.

```
function allowance(address _owner, address _spender) public view returns (uint256
    remaining) {
  return allowed[_owner][_spender];
}
```

The events `Transfer` and `Approval` play an important role in the interface of an ERC20 token. The `Transfer` event is emitted every time when a transaction is about to be executed. The `Approval` event is being called every time some tokens need to be approved. Both events trigger some sort of action in the linked wallet application of the user. For example, the browser wallet MetaMask opens a popup window asking the user for confirmation, once an event is emitted.

```
1 event Transfer(address indexed _from, address indexed _to, uint256 _value);
2 event Approval(address indexed _owner, address indexed _spender, uint256 _value);
```

4.4 Liquidity Pool Implementation

There are a few different approaches on how to implement a lending functionality. One possibility would be the utilization of collateral like it is done by Maker[29]. However, this section will describe how to implement a lending application using liquidity pools as it builds the foundation for a decentralized exchange such as Uniswap[32].

As described in section 3.5.2, liquidity pools need to have a pair of cryptocurrencies or tokens. As we are building on top of the Ethereum network, we are limited to Ether and Ethereum tokens. In this example, the liquidity pool holds Ether and an ERC20 token called CherryToken, as implemented in section 4.3. Besides the token reference `cherryToken` and the exchange reference `cherrySwap`, the smart contract needs to keep track of the curren pool balance, its collected fees by the exchange and the balances each liquidity miner holds. The mapping `unresolvedEth` is needed to manage Ether that is sent to the contract without referencing a specific swap transaction.

```
1 string public name = "CherryLiquidity";
2 address public owner;
3 CherryToken public cherryToken;
4 CherrySwap public cherrySwap;
5 uint256 private ethBalance;
6 uint256 private ctnBalance;
7 uint256 private collectedFees = 0;
8 mapping(address => uint256) private balancesInEth;
9 mapping(address => uint256) private balancesInCtn;
10 mapping(address => uint256) private unresolvedEth;
```

The constructor initializes the smart contract, sets the contract references and seeds the pool with an initial amount. The getter functions `getEthBalance`, `getCtnBalance` and `getCollectedFees` provide the frontend with necessary information to implement a good user experience and to keep as much calculation off-chain.

```
1 constructor(CherryToken _cherryToken, CherrySwap _cherrySwap, uint256
      _initialEthSupply, uint256 _initialCtnSupply) public {
2   cherryToken = _cherryToken;
3   cherrySwap = _cherrySwap;
4   ethBalance = _initialEthSupply;
5   ctnBalance = _initialCtnSupply;
6   cherryToken.transferFrom(msg.sender, address(this), _initialCtnSupply);
7   owner = msg.sender;
8 }
9
10 function getEthBalance() public view returns (uint256 balance) {
```

```
11   return address(this).balance;
12 }
13
14 function getCtnBalance() public view returns (uint256 balance) {
15   return cherryToken.balanceOf(address(this)) - collectedFees;
16 }
17
18 function getCollectedFees() public view returns (uint256 fees) {
19   return collectedFees;
20 }
```

Adding liquidity works a little bit different for Ether and an ERC20 token, as Ether is a fundamental feature of Solidity and an ERC20 token is managed by the smart contract itself. At first, Ether needs to be sent to `addEthLiquidity`. As it is a payable function, it is able to receive funds. They are owned by the smart contract and are mapped to the user through his address, which is stored in `msg.sender`. Secondly, the function `addLiquidity` needs to be called to complete the pool deposit. After a few integrity checks, the unresolved Ether will be reduced, the tokens transferred and the pool balances updated.

```
1 function addEthLiquidity() payable {
2   require(msg.value > 0, "ETH amount cannot be 0");
3   unresolvedEth[msg.sender] = unresolvedEth[msg.sender] + msg.value;
4 }
5
6 function addLiquidity(uint256 _ethAmount, uint256 _ctnAmount) public {
7   require(unresolvedEth[msg.sender] >= _ethAmount, "ETH not sent yet");
8   require(_ctnAmount > 0, "CTN amount cannot be 0");
9   require(cherryToken.balanceOf(msg.sender) > _ctnAmount, "not enough CTN funds");
10  unresolvedEth[msg.sender] = unresolvedEth[msg.sender] - _ethAmount;
11  cherryToken.transferFrom(msg.sender, address(this), _ctnAmount);
12  balancesInEth[msg.sender] = balancesInEth[msg.sender] + _ethAmount;
13  balancesInCtn[msg.sender] = balancesInCtn[msg.sender] + _ctnAmount;
14 }
```

Removing liquidity from the pool means withdrawing the lended funds and receiving an additional reward proportional to the pool share. As the reward calculation is quite computational intensive and Soldity is not good at managing divisions and percentages, the reward will be computed off-chain.

```
1 function removeLiquidity(uint256 _ethAmount, uint256 _ctnAmount, uint256 reward)
      public {
2   require(_ethAmount > 0, "eth amount cannot be 0");
3   require(_ctnAmount > 0, "ctn amount cannot be 0");
4   require(getEthBalance() >= _ethAmount, "not enough ETH in pool");
5   require(getCtnBalance() >= _ctnAmount, "not enough CTN in pool");
6   require(balancesInEth[msg.sender] > _ethAmount, "not enough ETH deposited");
7   require(balancesInCtn[msg.sender] > _ctnAmount, "not enough CTN deposited");
8   require(collectedFees >= reward, "invalid reward");
9   uint256 ctnPayout = _ctnAmount + reward;
10  cherryToken.transferFrom(address(this), msg.sender, ctnPayout);
11  msg.sender.transfer(_ethAmount);
12  balancesInEth[msg.sender] = balancesInEth[msg.sender] - _ethAmount;
13  balancesInCtn[msg.sender] = balancesInCtn[msg.sender] - _ctnAmount;
14 }
```

The functions `processEthToCtn` and `processCtnToEth` are being called by the exchange in order to withdraw the needed funds and deduct the fees. To make sure only the exchange can call this function, the address of the recipient will be checked at the beginning of each function.

```
1 function processEthToCtn(address recipient, uint256 ctnAmount, uint256 fees) public
      {
2   require(msg.sender == address(cherrySwap), "address not authorized");
3   addFees(fees);
4   cherryToken.transfer(recipient, ctnAmount);
5 }
6
7 function processCtnToEth(address recipient, uint256 ethAmount, uint256 fees) public
     {
8   require(msg.sender == address(cherrySwap), "address not authorized");
9   addFees(fees);
10  recipient.transfer(ethAmount);
11 }
12
13 function addFees(uint256 _amount) private {
14  collectedFees = collectedFees + _amount;
15 }
```

4.5 Exchange Implementation

A truly decentralized exchange is dependent of its own liquidity pools. While the functionality of a liquidity pool is really complex, the actual implementation of the exchange is a lot easier. It can be implemented in just two functions. oOne function for each exchange direction. Both of these functions follow the same structure: Receiving funds, calculating fees, deducting fees and sending funds.

When exchanging from Ether to an ERC20 token, the function needs to be payable. If a function is payable, network participants have the opportunity to send Ether to the contract address and specifying a specific function in the data component of the transaction. Most smart contracts have a fallback function which handles funds that are not dedicated to a specific function. Whenever the data component of a transaction addressed to the smart contract is empty, it will be handled by the fallback function, where the smart contract could keep track of the sender addresses.

```
1 function () external payable {}
```

In this case, the fallback function is empty, as another specific function handles the exchange mechanism. Incoming funds from the fallback function will be added into the liquidity pool, which contributes to the price stability of the exchange.

If someone wants to exchange Ether into an ERC20 token, the name of the function needs to be specified. The incoming payment is accessible in the function by using the value property of the message object. After checking if the payment is not empty, the function calculates the fees which will distributed to the liquidity miners. This calculation can be either fixed or dynamically. On more sophisticated exchanges, the percentage of the fees uses an approximation of an exponential curve. If the liquidity

pool is very small, the fees are really high[1]. As the pool gains more depth, the fees decrease into small fractions of percentages but never will be zero.

But even simple percentage calculations are a little bit complicated in Solidity, as the languages does not provide a floating point data type. All numeric asset representations are stored in wei, the smallest unit on the Ethereum Platform. One Ether is equivalent to one quintillion wei, which is a number with 19 digits.

```
function swapEthToCtn() public payable {
  require(msg.value > 0, "amount cannot be 0");
  uint256 fees = msg.value.mul(100).div(1);
  uint256 ctnAmount = msg.value.mul(1000).sub(fees);
  _addFees(fees);
  require(getCtnBalance() >= ctnAmount, "not enough funds available");
  _cherryToken.transfer(msg.sender, ctnAmount);
  }
```

After the fees are calculated, the function checks, if the smart contract has enough funds to pay the user back. If this is not the case, the transaction aborts and the money will be transferred back to the user. If there are still enough funds available, the amount deducted by the fees will be paid back using the `transfer` function from the ERC token standard.

The reversed function is not equipped with the `payable` modifier, as the incoming funds are an ERC20 token. Because the funds of the ERC20 tokens are managed by the token itself, the exchange contract has no direct access to it. If a user sends token funds to the address of the exchange, the exchange has no way to get notified about it. Therefore, the token transaction will take place in the contract function on behalf of the user. All the user has to do before calling the exchange function, is to grant the exchange permission to spend the funds on his behalf using the `approve` function from the ERC20 token standard.

Instead of just paying the desired amount to the exchange, the amount needs to be passed to the function as a parameter. As before, the amount is being checked if it is empty to save the gas which would be wasted when executing a swap with zero funds. After that, the exchange rate and fees will be calculated. After another safety check, the tokens will be sent from the user to the exchange and the exchange sends the Ether amount back to the user using the built-in `transfer` function.

```
  function swapCtnToEth(uint256 amount) public {
    require(amount > 0, "amount cannot be 0");
    uint256 ethAmount = amount.div(1000);
    uint256 fees = ethAmount.mul(1).div(10);
    _addFees(fees);
    require(getEthBalance() >= ethAmount.sub(fees), "not enough funds available");
    _cherryToken.transferFrom(msg.sender, address(this), amount);
    msg.sender.transfer(ethAmount);
  }
```

[1] up to 50% and more

4.6 Testing Smart Contracts

Testing plays a crucial part in the development of decentralized applications. Once deployed, the smart contracts are immutable on the blockchain. Fixing bugs after the initial deployment can be expensive and tedious, as the affected smart contracts need to be re-deployed and might be incompatible with the previous version. When working with financial assets of other people, it is even more vital, to minimize the number of bugs. Therefore, most applications use one or more testing frameworks and use a three-phase deployment cycle, as described in more detail in section 4.8.

As a developer, you have the choice to write tests natively in Solidity, or using an arbitrary testing framework based on JavaScript. When working with Truffle, the testing framework Mocha[18] and the assertion library Chai[14] are preinstalled as the default. All tests that are in the testing subdirectory, will be executed using the command `truffle test`.

At the top of each test file, the artifacts of the used smart contracts, as well as Chai, need to be imported:

```
1 const CherryPool = artifacts.require('CherryPool');
2 const CherryToken = artifacts.require('CherryToken');
3
4 require('chai')
5   .use(require('chai-as-promised'))
6   .should();
```

The specific tests are to be implemented in a contract block, which takes two parameters: The name of the tested contract and an array of test accounts as parameters. As the first account is used to deploy the contract, it is labelled as the contract owner.

```
1 contract('CherryPool', ([owner, user]) => {
2   // tests are here
3 }
```

While the contract compilation takes place before the tests are being executed, the smart contracts still need to be instantiated. This can be done using a `before` block, where all initialization tasks for the tests take place:

```
1 before(async () => {
2   // load contracts
3   cherryToken = await CherryToken.new();
4   cherryPool = await CherryPool.new(cherryToken.address);
5   await cherryToken.transfer(cherryPool.address, '100000000000000000000'); // 1000
      CTN
6   await web3.eth.sendTransaction({ from: owner, to: cherryPool.address, value: web3.
      utils.toWei('1') });
7 });
```

The actual tests are grouped together to tests that are similar to each other or that test the same functionality into `describe` and `it` blocks for the concrete unit tests. The estimated results of the test can be checkt with assertions or by appending `should.be.rejected` to a function call.

```
1 describe('CherryPool deployment', async () => {
2   it('has no collected fees yet', async () => {
3     const collectedFees = await cherryPool.getCollectedFees();
4     assert.equal(web3.utils.fromWei(collectedFees), 0);
```

```
  });

  it('has initial Ether pool', async () => {
    const ethBalance = await cherryPool.getEthBalance();
    assert.equal(web3.utils.fromWei(ethBalance), 1);
  });

  it('has initial CherryToken pool', async () => {
    const ctnBalance = await cherryPool.getCtnBalance();
    assert.equal(web3.utils.fromWei(ctnBalance), 1000);
  });
});
```

4.7 Connecting Smart Contracts to Web Applications

The connection between the web frontend and the smart contracts is done using the JavaScript framework Web3[23], which is maintained by the Ethereum core team. It contains a lot of useful utility functions as seen before such as `web3.eth.sendTransaction()` for making transaction or `web3.utils.fromWei()` to convert a number from wei into Ether.

When the user wants to connect to a wallet, Web3 needs to determine the network provider first. Once detected, a new instance of Web3 will be instantiated and is ready for use.

```
function loadWeb3() {
  if ((window as any).ethereum) {
    (window as any).web3 = new Web3((window as any).ethereum);
    (window as any).ethereum.enable().then();
    return true;
  } else if ((window as any).web3) {
    (window as any).web3 = new Web3((window as any).web3.currentProvider);
    return true;
  }

  window.alert('Non-Ethereum browser detected. You should consider trying MetaMask
    !');
  return false;
}
```

The accounts array can be accessed via `web3.eth.getAccounts()` and the network id is retrieved using `web3.net.getId()`. Each contract can be instantiated afterwards based on the compilation artifacts. In addition to the tests, the correct network needs to selected first.

```
const cherryPoolData = (CherryPool as any).networks[networkId];
if (cherryPoolData) {
  cherryPool = new web3.eth.Contract((CherryPool as any).abi, cherryPoolData.address
    );
} else {
  alert('CherryPool contract not deployed to detected network!');
  return null;
}
```

After these steps, the API is ready to be used by any UI component. Smart contract functions can be invoked using `call()` or `send()`. The function `call()` is used if the function does not alter the state of the blockchain. A typical use case for that is retreiving an account balance:

```
1 account.cherryToken.methods.balanceOf(account.address).call().then((ctnBalance:
      number) => {
2   setCherryTokenBalance(web3.utils.fromWei(ctnBalance.toString()));
3 });
```

If a feature needs to alter the state of the blockchain, the function `send()` needs to be used. As every state modification costs gas, this will usually lead to a confirmation dialog provided by the wallet of the user. Typical use cases for that are adding liquidity, exchanging assets or approving funds for other accounts:

```
1 account.cherryToken.methods.approve(
2   account.cherryPool._address,
3   web3.utils.toWei(ctnToSupply.toString())
4 ).send({ from: account.address }).then();
```

4.8 Smart Contract Deployment

A typical release cycle of a blockchain application consists of three testing environments. At first, the smart contracts will be deployed to a local blockchain such as Ganache[17]. That is the environment where most of the testing will take place. Once the application is quite stable, it will be migrated to a public test network. These networks provide more realistic conditions, as it behaves more like the main network. Assets on the test networks have no value, which is really difficult to implement in reality. In order to prevent a value increase, the test networks need to be redeployed from time to time.

The most important test networks on Ethereum are Ropsten, Kovan (using the Aura consensus protocol) and Rinkeby (using the Clique consensus protocol). These conensus protocols are specifically designed to prevent the creation of value, which is exactly the opposite what the conensus protocols Proof of Work and Proof of Stake are trying to achieve.

Chapter 5

Closing Remarks

5.1 Criticism

—0.5 pages— * power consumption * criminalism and money laundering *

5.2 Risks

—0.5 pages— * smart contract risk * volatility * regulation * black swans * scams

5.3 Prospective Impact

— 0.5 pages —

Appendix A

Technical Details

Appendix B

Supplementary Materials

List of supplementary data submitted to the degree-granting institution for archival storage (in ZIP format).

B.1 PDF Files

Path: /

thesis.pdf . . . . . Master/Bachelor thesis (complete document)

B.2 Code-Files

Path: /media

. . . . . .

B.3 Reference Files

Path: /online-sources

. . . . . .

Appendix C

Glossary

References

Literature

[1] Andreas Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. 2nd ed. Sebastopol: O'Reilly Media, Inc., 2017 (cit. on p. 5).

[2] Andreas Antonopoulos and Gavin Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. Sebastopol: O'Reilly Media, Inc., 2018 (cit. on pp. 5, 12, 14).

[3] Adam Back. *Hashcash - A Denial of Service Counter-Measure*. Aug. 2002. URL: http://www.hashcash.org/papers/hashcash.pdf (cit. on p. 5).

[4] Badr Bellaj, Richard Horrocks, and Xun Wu. *Blockchain By Example*. Birmingham: Packt Publishing Ltd., 2018 (cit. on p. 14).

[5] Brenn Hill, Chopra Samanyu, and Valencourt Paul. *Blockchain Quick Reference*. Birmingham: Packt Publishing Ltd., 2018 (cit. on p. 6).

[6] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Oct. 2008. URL: https://bitcoin.org/bitcoin.pdf (cit. on p. 5).

[7] Klaus Schmeh. *Kryptografie: Verfahren, Protokolle, Infrastrukturen*. Heidelberg: dpunkt.verlag GmbH, 2007 (cit. on p. 3).

[8] *Solidity Documentation*. Version 0.7.5. Oct. 2020. URL: https://docs.soliditylang.org/ (cit. on p. 12).

[9] Melanie Swan. *Blockchain: Blueprint for a New Economy*. Sebastopol: O'Reilly and Associates, 2015 (cit. on p. 5).

[10] Fabian Vogelsteller and Buterin Vitalik. *ERC-20 Token Standard*. A standard interface for tokens. Mar. 8, 2019. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md (cit. on p. 14).

[11] Gavin Wood. *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Version 3e2c089. Sept. 2020. URL: https://ethereum.github.io/yellowpaper/paper.pdf (cit. on p. 6).

[12] Xun Wu and Weimin Sun. *Blockchain Quick Start Guide*. Birmingham: Packt Publishing Ltd., 2018 (cit. on p. 9).

Software

[13] *Ampleforth*. URL: https://github.com/ampleforth/uFragments (visited on 01/10/2021) (cit. on p. 7).

[14] *Chai*. URL: https://github.com/chaijs/chai (visited on 11/26/2020) (cit. on pp. 13, 20).

[15] *Chainlink: Blockchain Oracles for connected Smart Contracts*. URL: https://github.com/smartcontractkit/chainlink (visited on 01/10/2021) (cit. on p. 4).

[16] *CherryPool*. URL: https://github.com/johannesdominik/cherrypool (visited on 11/26/2020) (cit. on p. 12).

[17] *Ganache*. URL: https://github.com/trufflesuite/ganache (visited on 11/26/2020) (cit. on pp. 13, 22).

[18] *Mocha*. URL: https://github.com/mochajs/mocha (visited on 11/26/2020) (cit. on pp. 13, 20).

[19] *React*. URL: https://github.com/facebook/react/ (visited on 11/26/2020) (cit. on p. 13).

[20] *The Solidity Contract-Oriented Programming Language*. URL: https://github.com/ethereum/solidity (visited on 11/26/2020) (cit. on p. 12).

[21] *Truffle*. URL: https://github.com/trufflesuite/truffle (visited on 11/26/2020) (cit. on p. 12).

[22] *TypeScript*. URL: https://github.com/Microsoft/TypeScript (visited on 01/09/2021) (cit. on p. 13).

[23] *web3.js - Ethereum JavaScript API*. URL: https://github.com/ethereum/web3.js (visited on 11/26/2020) (cit. on pp. 13, 21).

Online sources

[24] *Bitcoin Cash*. Nov. 2020. URL: https://www.bitcoincash.org/ (visited on 11/24/2020) (cit. on p. 6).

[25] *Bitcoin Energy Consumption Index*. Oct. 2020. URL: https://digiconomist.net/bitcoin-energy-consumption (visited on 11/24/2020) (cit. on p. 6).

[26] *Ethereum Casper*. Sept. 2018. URL: https://twitter.com/i/events/1036281460704112645 (visited on 11/24/2020) (cit. on p. 6).

[27] *Gemini Dollar*. Jan. 2021. URL: https://www.gemini.com/dollar (visited on 01/10/2021) (cit. on p. 7).

[28] *Liquidity Mining, risks and exchange function on a DEX explained*. Nov. 2020. URL: https://julianhosp.com/liquidity-mining-risks-and-exchange-function-on-a-dex-explained/ (visited on 11/24/2020) (cit. on p. 10).

[29] *MakerDAO: An Unbiased Global Financial System*. Jan. 2021. URL: https://makerdao.com/en/ (visited on 01/10/2021) (cit. on pp. 7, 8, 16).

[30] *Tether: Stable digital cash on the Blockchain*. Jan. 2021. URL: https://tether.to/ (visited on 01/10/2021) (cit. on p. 7).

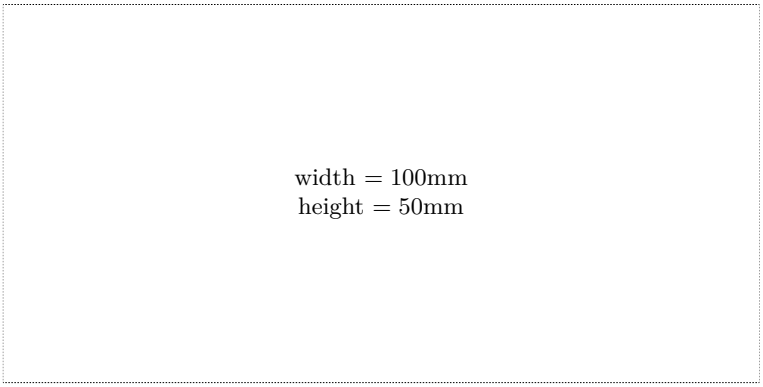
[31] *Top DeFi Tokens by Market Capitalization*. Jan. 2021. URL: https://coinmarketcap.com/defi/ (visited on 01/09/2021) (cit. on p. 4).

[32] *Uniswap Pools*. Nov. 2020. URL: https://uniswap.org/docs/v2/core-concepts/pools/ (visited on 11/24/2020) (cit. on pp. 9, 16).

[33] *USD Coin (USDC) Stablecoin*. Jan. 2021. URL: https://www.circle.com/en/usdc (visited on 01/10/2021) (cit. on p. 7).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —