TU WIEN Informatics

Smart Contract-based Cloud offloading

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Thomas Laner
Registration Number 11807845

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Ivona Brandic
Assistance: Dr. Vincenzo De Maio

Vienna, 1st July, 2022

Thomas Laner	Ivona Brandic

Erklärung zur Verfassung der Arbeit

Thomas Laner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juli 2022

Thomas Laner

Acknowledgements

In this segment of my work, I would like to thank all those people who have supported and promoted this work in one way or another.

First of all, I would like to thank my supervisors Univ.Prof.in Ivona Brandic and Dr. Vincenzo De Maio, who have supported and guided me in the course of this work, both in the practical part and in the writing of this thesis.

Secondly, I would like to thank the crypto community, from whom I could learn a lot and who made my entry into web3 development as easy as possible through helpful tutorials and blogs on the platforms *Youtube*, *Medium* and *stackExchange*, but also the developers of the numerous tools I used to implement my smart contract (see 5.1).

Finally, I would like to thank my partner and my parents for their support during my bachelor's degree and especially my parents for making my studies possible.

Abstract

Due to the ever-increasing demand of applications (e.g. IoT, cloud gaming) for computing resources, cloud computing and especially offloading have become one of the most important topics in computing. Thereby, cloud offloading describes the process of outsourcing resource-intensive tasks to remote surrogate machines (e.g., cloud servers). It has a number of significant advantages over traditional local data processing: it takes advantage of external computing resources that can be more powerful, faster and cheaper than their local counterpart. However, cloud offloading also has some drawbacks.

One of these drawbacks is a need to have trust into an external entity. Trust is necessary since users have to rely on a third party to ensure that their data which is exchanged over the network remains secret and is not not exploited by the third-party for its own benefit in any way (e.g., reselling data to interested enterprises, using the users' data against them). Another drawback of cloud offloading is the high degree of centralisation among a few parties. In this context, it should be mentioned that centralisation in itself is not a bad thing. In the case of cloud computing, it allows users to enjoy the convenience and ease of use that only the capabilities of a large company can provide. However, this type of centralisation can also be used by providers to restrict the free market, for example through practices that make it difficult for users to transfer their data to another type of data storage (e.g. opaque contract terms). Moreover, as centralisation increases, these few parties become potential 'single points of failure', making it increasingly attractive to attack them as a way to cause harm to a large number of people at the same time. The final drawback we would like to mention in this context is the high entry costs for providers of current cloud offloading services. This reduces the number of new providers that can enter the market and thus inhibits competition and development.

In the following thesis, we have addressed the drawbacks of cloud offloading and solved them with a smart contract running on the Ethereum blockchain. Here, the contract takes on the role of a resource trader, able to penalise participants for misbehaviour and react differently to the outputs provided by nodes depending on whether they are agreed to by both parties, thus solving the problems mentioned above.

When analysing the practical results of the work, we found that our proof-of-concept was successful. Our approach, which we implemented using a number of different tools (see 5.1), is able to provide an offloading framework for an arbitrary number of users on a private or public Ethereum (see 3.2) blockchain (see 2.2).

Contents

CHAPTER 1

Introduction

In this chapter, we first describe the issues we have identified in cloud computing in the use case of outsourcing computing resources to the cloud and the consequential research question of our thesis. The second part of the chapter is characterised by an outline of our solution and a brief description thesis structure.

1.1 Offloading in Cloud Computing

Many different new IT applications (e.g., virtual reality, cloud gaming, IoT-based applications) have emerged in recent years. Most of these new applications have a few things in common: strict latency requirements and a huge demand for computing power. As a result, the demand for computing resources has increased, creating a resource gap between the computing power required by the applications and the actual computing power available locally, which is limited by the on-device computing power and battery capacity.

This resource gap can be closed by using offloading in cloud computing, as described in Figure 1.1. A cloud offloading interaction starts with a user interacting with either a mobile device (i) or a device that is part of an IoT network (iv). Subsequently, this device then requests computing resources when it cannot process the requested operations itself (ii, v). These requests are then handled by a supervising entity that manages and allocates the resources made available to the cloud offloading network (iii) by enterprise cloud providers to the requesting devices. Compared to traditional computing, this technology has many advantages [WPE^{+}19]. It offers an external computing option that is made available to users following a pay-as-you-go model, but can be more powerful, faster and, for certain applications, more cost-effective than the traditional option of using local computing power. Therefore, offloading has recently become more and more important in the IT sector [AGH18], [AZH18]. This momentum is reinforced by technologies such

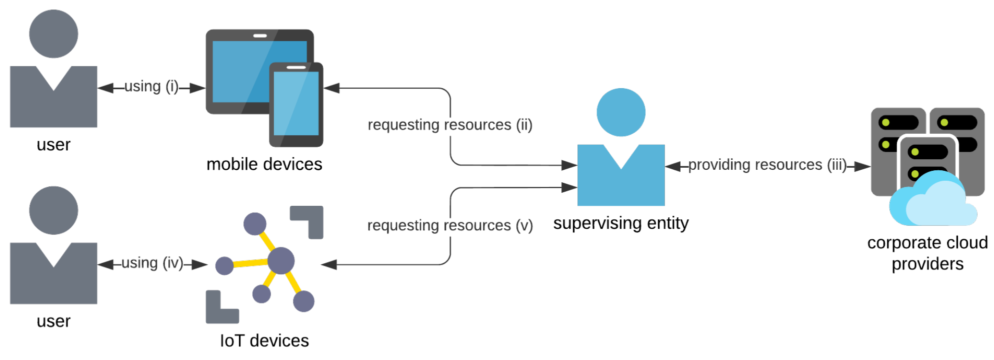

Figure 1.1: Conventional offloading

as 5G [GJ15] and Starlink [sta], which enable unprecedented speed and bandwidth in data transmission, reducing the bottleneck in communication between cloud and device.

There are several critical factors in offloading that need to be analysed in detail. To begin with, users have to pay for the resources they demand. This point should by no means be underestimated, as it determines to a large extent the success of a business model based on offloading. The lower the cost of resources can be set, the more users such a service will have and the bigger the market will be. The larger the market, the easier it is for providers to break even, which makes the market more attractive for new providers, which in turn leads to lower prices. Furthermore, trust plays a big role in cloud computing. Users entrust their data to an unknown company. They also have to trust the intermediary who oversees the execution of the contract to ensure that both parties meet the requirements placed on them. This intermediary usually needs to be compensated in some way for its services and therefore needs to be included in the cost breakdown alongside the payment for the associated resources.

The research question which is addressed in our work was defined based on the previously analysed critical factors of cloud offloading. It is defined by the query of whether it is possible to design and implement a real-market, autonomous supervising entity for cloud offloading that is cost-efficient and can be inherently trusted by both parties.

1.2 Smart contract-based offloading

In this work, we have solved the research question posed with the help of another technology that has become increasingly recognised and widespread in recent years: smart contracts.

Smart contracts are programmes that are able to store values, states and query conditions.

They represent autonomous agents that act according to their underlying program code and the inputs given to them. They are therefore completely predictable. To guarantee that the code of a smart contract cannot be changed and tempered with, a so-called blockchain must be used to store them and their state. A blockchain is a distributed and decentralised ledger that contains the chain of state changes processed on it (so called transactions), a certain number of which are grouped into blocks [SJD16] - hence the name blockchain. Blockchains are very difficult to permanently falsify (see 2.2) , which makes them perfect for storing smart contracts.

The aforementioned critical aspects of outsourcing in cloud computing can be addressed by using smart contracts. The aspect of a supervising party that needs to be trusted is covered by the code of the smart contract. Since the code is viewable by anyone, it is clear what principles are being acted upon, and since the contract cannot usually be changed once it has been migrated to a blockchain, everyone can be sure that the code will adhere to these principles for the duration of its use. The financial aspect can also be covered by a smart contract. Since the contract in this case takes the position of the supervising party, massive cost savings can be achieved. If the contract is written in such a way that even if it is successfully executed, the contract does not generate any profit, both parties only have to pay for the interactions they have with the contract. These costs are usually small amounts of cents. In contrast to traditional middleman-based offloading, these very low entry costs now also make it possible for individuals to make computing power they do not need available to others via a smart contract for a profit.

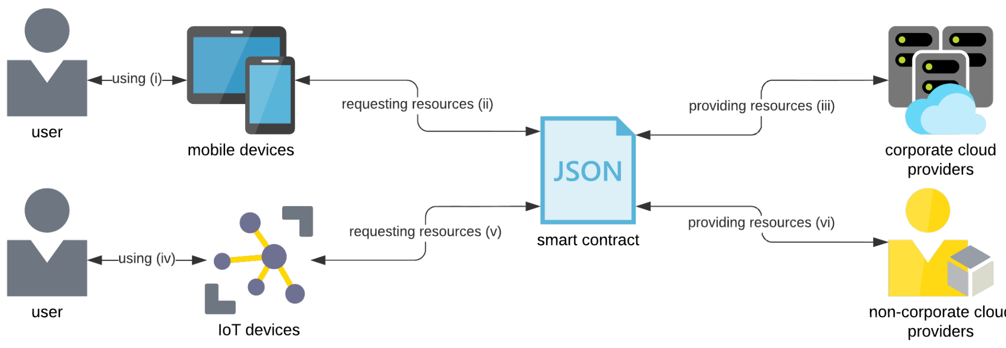

Figure 1.2: Smart contract-based offloading

Due to the aforementioned advantages offered by a smart contract, we decided to use this technology to answer our research question (see 1.1). Cloud offloading based on a smart contract is shown in Figure 1.2. The difference between conventional cloud offloading (see 1.1) and the one depicted here is the supervising entity, which is now represented by a smart contract, and an additional resource provider in the form of non-corporate cloud providers (vi). These new providers can now also participate in the market, as explained in the previous paragraph, due to the low entry costs. In doing so, we have focused on implementing a solution that has two main goals: low entry costs and high customisation potential. We set these goals as we wanted to give as many people as possible the opportunity to use our solution or customise it to their needs and then host their own version of the smart contract on a private or public blockchain. A secondary goal we set for our implementation was to maximise meaningful use of decentralisation. This goal was in addition to the two main goals, as we wanted to rely as much as possible on the benefits (e.g. security, transparency, cost reduction) of a blockchain (see 2.2).

In our work, firstly we addressed the survey of the current state of research. The results of which can be found in Chapter 2. On the basis of this we chose the blockchain on which to build our proof-of-concept (see 3.1). Next, we conceptualised our design, which is described in Chapter 4. Then we implemented our solution as described in Chapter 5. We then extensively tested and analysed our smart contract on a private blockchain. The results of this can be found in chapter 6 and chapter 7.

CHAPTER 2

Related Work

In this chapter, we provide an overview of the state-of-the-art of blockchain and cloud computing technologies to date, with a focus on offloading. Thereby, we first focus on the topics of blockchain and cloud computing separately and then on their interoperation by explaining various use cases and work already carried out.

2.1 Cloud Computing: Offloading

As discussed in 1.1, the resource gap that has emerged in recent years due to an increasing divergence between the amount of locally available computing resources and those demanded by applications can be closed by offloading in cloud computing. This gap can be closed by outsourcing computing power and storage to external computers, so-called offloading in cloud computing. Currently, offloading is in the hands of a few large providers, namely Amazon Web Services[aws] , Microsoft Azure[msf] and the Google Cloud[goo]. Through this technology, customers can be provided with virtually unlimited computing power, network and storage resources. In recent studies [WPE^{+}19], models have been created to improve offloading processes. These models focus on the factors of offloading target, load balancing, mobility, partitioning and granularity and state that an optimal offloading algorithm should balance these factors to optimally solve offloading problems.

An example of a non-trivial use case of offloading is provided by the offloading algorithm described in [HWN12]. This has arisen from the need described in [LWX01] to save the battery life of mobile devices by offloading. The designed algorithm is based on the so-called Lyapunov optimisation and enables handheld devices to save battery while making sure that the execution time of the respective application meets the given time constraints.

2.1.1 Edge- & Fog Computing

In recent years, cloud computing has evolved out of the need to support even more resource-intensive tasks with even higher network requirements, such as augmented- or virtual reality. Since these kinds of application require stringent latency requirements have to be respected. This is difficult in conventional cloud computing and hence attempts are now being made to meet these through so-called "fog computing" and "edge computing" that have to be differentiated.

In Edge Computing [SD16], resources at the edge of the network and thus closer to the requesting devices are used for computing, which both increases bandwidth and reduces latency. The question of ideal workload offloading plays a major role here, because the more fitting the respective nodes are selected, the better the performance of the offloading [CLMS20]. One of the IT areas where edge computing will need to be used to solve latency issues related to cloud computing is the use of IoT devices [SJD16]. Edge computing and Fog computing must be distinguished since Fog computing works with the cloud and has a hierarchical layered architecture, while the Edge and Edge computing are defined by the minimum number of layers and are surrounded by peripheral devices [SWHL16]. The cloud integration of Fog Computing allows it to dynamically reconfigure itself for different applications. Therefore, Fog Computing is not tied to a hardware-softare combination, but rather consists of the fog nodes in the cloud that provide resources [SSHKMFM22].

2.2 Blockchain

A blockchain is a distributed and decentralised ledger that contains the chain of transactions processed on it, a certain number of which are grouped into blocks [Nak08]. The number of transactions that are combined in a block depends on the respective blockchain and its requirements [SJD16]. Blockchains cannot be permanently falsified (apart from a 51%-hack [SSN$^+$20]), as the transaction-verifying participants would recognise errors of this kind during the process of validating a new block [QYWW18]. Also, it is not possible to introduce a forged block into the blockchain, as the subsequent blocks would ignore the forged block and thus it would be of no value [B$^+$14].

The validation of a transaction is called mining and is carried out by participants in the blockchain, the so-called miners. These are encouraged by rewards in the form of the blockchain's own native currency, a so-called cryptocurrency, to be the first to validate the respective block [HR17]. In doing so, a blockchain's cryptocurrency serves as a medium of exchange on that blockchain and thus a certain value is attached to it [ethd]. What exactly can be done with a cryptocurrency is determined by the code of the respective blockchain. In the case of our proposed solution, for example, the underlying cryptocurrency is used as payment for computing power.

2.2.1 Limitation of blockchain-based approaches

Feasibility analysis

Since the approach that our solution takes, namely the use of smart contracts, depends on an underlying blockchain on which the contracts are stored, one can ask what disadvantages this kind of dependency entails and what limitations it imposes.

Costs are the first thing that comes to mind in this context. The costs of using a smart contract based on a blockchain are primarily caused by the mining process. All other costs are optional and are caused by the code of the respective smart contract. These mining fees usually have to be paid in the blockchain's own currency, a so-called cryptocurrency. These currencies are not inherently tied to a fiat currency and thus have a quite volatile character. This now opens up the question of how volatile the costs of an interaction with a blockchain and thus also with a smart contract can be in order to determine the usefulness of the solution we have outlined for the trust and cost problem associated with offloading. In the paper [ISFZ21] it has been shown that the supply-demand curve that can be modelled from past Bitcoin transactions has a "*relatively flat, downward-sloping demand curve and a much steeper, upward-sloping supply curve*" . This curve can be used to analyse the market equilibrium between users trying to avoid transaction delays and miners trying to maximise their transaction fee revenue, which the algorithmic nature of supply and demand that cryptocurrencies usually entail leads to a stabilisation of transaction fees in a regular operating environment.

Programmatic design is another factor, that could potentially limit the feasibility of a blockchain-based solution and hence has to be considered in the course of the feasibility analysis apart from the cost factor. In this sense, studies have shown that one of the main difficulties of blockchains is their resource limitation [SSHKMFM22]. This means that sophisticated algorithms and storage operations cannot be executed on the chain due to the computational effort and cost required to do so. Therefore, developers must always keep resource constraints in mind, and perform resource-intensive memory operations off-chain in some way.

User accountability

Since public blockchains are open databases that anyone can use with complete anonymity, the question of how participants can be held accountable for their actions arises.

In this context, the first thing to mention are the mechanisms that blockchains themselves use to achieve accountability of their miners, the so-called consensus mechanisms. In [NHN+19], the authors analyse different consensus mechanisms and conclude that there are various trade-offs between the mechanisms used.

Previous research has also taken other approaches to enforcing user accountability in different ways. One of these approaches were made by Yasin and Liu in the form of a framework that aggregates users' identities and ranks their reputation using a smart contract [YL16]. Another proposal was explored in [Gui20]. When analysing various blockchain-based online social networks (OSNs), the authors of the paper found that only a handful of them actually address the blockchain identification problem seriously. One approach taken by the *Sapien* platform was for users to use Ether in the Ethereum blockchain-based application to be able to use the service, while at the same time using a reputation system that represents the value a user creates for the platform with their contributions. Several other approaches have been taken in this area. An approach also worth mentioning was taken by the authors of [ZKS+19]. In this work, a smart-contract-based framework was developed to serve as an access management system for IoT devices. In this way, users are blocked from accessing the respective IoT device for a certain period of time in the event of misconduct. To prevent users from simply creating a new account, they are identified by their behaviour patterns.

2.3 Combination of Blockchain and Cloud Computing

Recent studies have shown that cloud computing and blockchain technologies, considered separately, have different shortcomings. In the case of conventional cloud computing, these are mainly defined by centralisation and monopolisation as well as security risks resulting from the concentration of huge amounts of user data in one instance. In the case of blockchain technologies, the shortcomings are mainly defined by the inability to perform large amounts of computation at a reasonable cost and by scalability issues. However, the combination of these two technologies, as described in [GN21] , can solve many of the problems that the two technologies considered individually. These benefits range from accessibility, trust, stability, scalability to data management.

Categorisation approaches presented in [ZHZ+21] are an important tool, that we used frequently in our research. In order to categorise different use-cases of using blockchain technologies to enhance the capabilities of cloud computing, two different approaches were made by the papers authors. On one hand they defined categorisation of using blockchains in combination with cloud computing:

1. *Cloud as a Blockchain Service* (*CAABS*), where a blockchain is used to manage cloud computing.
2. *Blockchain as a Cloud Service* (*BAACS*), where blockchain services are used in cloud computing platforms.
3. *MBC*, where a Blockchain is used to record, witness or verify the data of a single or multiple layers in a cloud computing model.

And on the other hand, the possible roles of of a cloud in combination with a blockchain were described as:

1. Client: cloud computing as a client of the underlying blockchain network using it to store, authenticate or witness data.
2. Tradable resources: cloud computing as a resource on the underlying blockchain network where cloud services are traded, delivered and managed
3. Management platform: cloud computing as a management platform for a blockchain that allows customers to obtain the provided blockchain through a cloud trading platform
4. Computing resources: cloud computing as a computing resource of a blockchain, whereby the cloud can perform the computing operations
5. Storage resources: cloud computing as a storage resource of the blockchain to store data.
6. Peers: cloud computing as a peer node of the underlying blockchain.

2.3.1 Applications

A number of different applications for using blockchian technologies in cloud computing have already been defined in various papers. In the following we will list some of them that seemed relevant for our approach.

Fog-Computing in Blockchain-Mining is the first use-case we want to cover in this regard. Since offloading can be used to shift computations and thus reduce the load on local devices, the question arises as to how this could best be applied in the area of blockchain mining to reduce the workload on individual computers and thus increase the output of an entire mining system. This practice of using cloud computing for blockchains falls into the previously mentioned categorisations of *BAACS* and *cloud computing as a computing resource* which were defined above.

The paper [JWNS19], for example, presents an auction-based market model, which could be a fog-computing-based solution for Blockchain's proof-of-work verification model.

An alternative approach was taken in [XFW$^+$19]. Herein, the authors propose a blockchain infrastructure where the computationally intensive part of the mining is moved to a Fog Computing Cloud. In order to optimise profits, computational resource management is formulated using the Stackelberg-Duopol economic model.

Blockchain-Based Resource Allocation Models are another possible application of a cloud computing application. Here, the blockchain is used to manage the resources of the respective cloud (for example in Fog computing). This application case falls under the previously defined categorisations of *CAABS* and *cloud computing as a tradable resource.*

In our research, we have differentiated two use cases of these resource allocation models: Iot in combination with edge computing and computing resources with Fog computing.

Resource allocation models for IoT in Edge-Computing are the first use case that we want to analyse in this context. To this end, the study [Liu21] categorises different approaches of blockchain-based resource allocation models for IoT Edge-Computing applications into three classes:

1. auction-based: resource allocation based on QoS factors for edge computing with a focus on increasing accuracy in predicting available resources.
2. optimisation-based: resource allocation with focus on meeting cloud provider service level agreements.
3. revenue-based: resource allocation with a focus on ensuring QoS metrics of data flows by maximising resource utilisation.

Moreover, the study has shown that in 1 and 2, mainly meta-heuristics are used to address the problem, while in 3. mainly machine learning is used to solve the given problem.

Another study, [ZKS+19], proposed a smart contract-based access control for IoT devices that can penalise users by blocking their access to the devices and identifies them based on their behavioural patterns.

Resource allocation models for Computing Resources in Fog-Computing present the second use case that we want to analyse. In the paper [TBOB18], *Cloudchain*, a "*blockchain-based cloud federation that enables cloud service providers to trade their computing resources via smart contracts*", running on the public Ethereum blockchain, was developed. In their implementation, the authors used a structure of three contracts:

1. *cloudchain registry*: used to register cloud providers by mapping their Ethereum addresses to cloud provider identifiers calculated from customer ratings.
2. *cloudchain profile*: containing a list of references to *cloudchain* contracts representing all past and current engagements of the participant.
3. *cloudchain contract*: issued between two participants in response to a service request and used to process that request (e.g.: closure, transfer of credit).

The authors pursued several goals with this project: Firstly, they wanted to minimise the resource requirements of a provider by allowing their clients' requests to be outsourced to other members of the *cloudchain*. This approach further enables providers to rent their unused resources to other providers at cheaper rates instead of not using them. Secondly, the authors focused on maximising the profitability of using *cloudchain*. They tried to achieve this by implementing a differential game with the two parameters Gas cost and reputation value of a provider.

A different approach was taken by the authors of [HYYS20]. In their work, the authors introduce the concept of so-called "*parked vehicle assisted fog computing (PVFC)*", which uses smart contracts to create a safe and efficient way of offloading in order to use the unused computation resources of parked vehicles.

2.4 Discussion

Based on the research we have examined, several conclusions can be drawn.

Firstly, the feasibility of blockchain-based solutions has not only been demonstrated by their use in already completed projects, but has also been theoretically proven above (see 2.2.1).

Secondly, on the basis of the research analysed, we were able to conclude that the usefulness of blockchain technology and smart contracts has already been recognised by a large number of other researchers, as shown by the number of peer-reviewed papers on these topics we were able to find (see 2.3.1). In this regard, a large number of different topics have already been studied, but the topics that have been focused on the most are finance, energy and IoT [SJD16], as also described in the paper [Liu21].

Thirdly, we were able to conclude, that no approach to a Smart Contracts-based resource allocation model for cloud offloading perfectly meets our requirements for a low-entry cost solution with great adaptability potential and as much decentralisation as possible and sensible (see 1.2). It should be noted that *Cloudchain*[TBOB18] is quite close to the solution we envisioned and even has some features that could be used to extend our implementation (e.g. maximising the profitability of nodes through sophisticated algorithms). However, *Cloudchain* also uses a form of the *reputation-based* approach (see 3.3) to achieve accountability of its users. As we opted for the *staking-based* approach (see 3.3) for the reasons given in 4.1, this solution did not meet our requirements. Therefore, our solution had to be implemented from scratch.

CHAPTER 3

Methodology

This chapter describes the method we have chosen for this thesis. To this end, we first describe our approach to selecting a blockchain, followed by an explanation of the key concepts of the chosen blockchain. We then describe the approaches we identified that can be used to enforce user accountability for Blockchain-based solutions. The chapter concludes with a description of our implementation and testing methodologies.

3.1 Choosing a Blockchain

We focused on five key factors when selecting a blockchain for our implementation:

1. **Support for developer-defined smart contracts** is required as blockchains do not inherently support the migration of developer-defined smart contracts to the chain. Therefore, we had to choose a blockchain that does.

2. **Support for private & public blockchain** is necessary since it was our main goal to create a solution for the public market, while also ensuring that our solution can be used within closed systems. This private use could then be used, for example, to share computing resources that one department does not need at that time with another department of the same company. Since no outsider should participate in this case, a closed system is required.

3. **Low transaction costs** on the chosen blockchain, are necessary as we want to have low entry costs in order to be able to extend the solution to all types of markets and participants. If the entry costs were too high, it would not be possible to provide and request small amounts of resources from the network.

4. **High probability of blockchains long existence** is needed, since one of the goals of our implementation was adaptability, so that different entities can make

the changes they need to the system. If the blockchain on which the system is built did not last for years, our solution could no longer be used, and thus the goal of adaptability would have been in vain.

5. **Sophisticated development tools** for the chosen blockchain is the final factor we have chosen because we wanted to support the adaptability of our solution for others and also to simplify our initial development process.

There were many options[BB22] to choose from, but in the end we decided on the Ethereum blockchain[B$^+$14] as it seemed to best meet the requirements we stated. Other notable mentions include the original cryptocurrency Bitcoin[Nak08], which we did not select due to its limited support for smart contracts, and the cryptocurrency Solana[Yak18], which we did not select due to its smaller number of developer tools compared to Ethereum.

3.2 Chosen Blockchain: Ethereum

Ethereum was conceptualised by Vitalik Buterin in 2013. In the Ethereum White Paper [B$^+$14], he presented his idea of a blockchain that would allow developers to run their own decentralised applications on a blockchain. To achieve this, the Ethereum blockchain is designed with a built-in Turing-complete programming language that allows developers to create smart contracts and decentralised applications with individually definable rules and behaviour. The verification of interactions with smart contracts (the so-called Mining) in Ethereum is currently still done according to the so-called Proof-Of-Work principle, but with the "Ethereum Upgrade" [ethc] this will be changed to the more resource-friendly Proof-Of-Stake.

Ethereum Virtual Machine (EVM)

Is the name of the low-level, stack-based bytecode language in which code is written in Ethereum. The code consists of a series of bytes, where each byte represents an operation. Ethereum's "Hard Fork No.4: Spurious Dragon" update [Jam] introduced a size limit for contracts on the Ethereum blockchain to prevent denial-of-service attacks. This limit was set at 24576 bytes and must always be kept in mind when developing smart contracts, otherwise it is fairly easy to exceed it. For this reason, the code of an Ethereum application should always be kept as lean as possible and sensibly divided into several shorter smart contracts.

Accounts

Accounts in Ethereum are divided into the so-called externally owned- (EOW) and contract accounts. The former are under the control of users and the latter are under the control of a defined smart contract. Accounts are uniquely identified by their 20-byte addresses.

Transactions

The state of Ethereum is defined by the set of accounts. State transactions are therefore defined by transactions between two accounts, which must be recorded in the underlying blockchain. Each function call of a smart contract in Ethereum is actually to be seen as a transaction and for this reason is also called such.

Ether

Ether is the native currency of Ethereum. It is used to pay transaction fees, but can also be sent from one account to another. For this reason, Ether is the base currency of all Ethereum-based applications and all cryptocurrencies based on Ethereum are indirectly a certain amount of Ether. The basic unit of Ether is Wei. 1 Ether is equal to a quantity of 10^{18} Wei.

Gas

Gas is the term used to describe the cost of a transaction on the Ethereum blockchain. It's use is necessary to avoid denial-of-service attacks and it was already embedded in the original Ethereum code. Gas is defined in the unit Wei, where one computation step costs 1 Gas and therefore 1 Wei.

Smart Contracts

Smart contracts, as stated in the introduction (see Chapter 1), have fully predictable outcomes, are executed automatically, are publicly accessible due to the underlying blockchain, protect the privacy of users as they interact with the blockchain via anonymous accounts, and have visible terms[ethe]. Ethereum's smart contracts are, as already mentioned, not the only form of smart contracts. Before the creation of Ethereum, the original cryptocurrency Bitcoin[Nak08] already supported a type of smart contract simply called a "contract"[btc]. Like Ethereum's contracts, these are also "cryptographic "boxes" that hold value and unlock it only when certain conditions are met", minimising trust requirements. The difference between Bitcoins contracts and Ethereum's smart contract is the Turing completeness, value awareness, blockchain awareness and state that Ethereum's smart contracts add [B+14] along with the generally lower cost of transactions in Ethereum, making the use of its contracts more viable.

It is possible for anyone to write smart contracts and deploy them on the Ethereum blockchain. The only requirements are a smart contract compiled in a programming language that is supported by Ethereum and enough Ether to deploy the contract on the blockchain [ethe]. When a contract is compiled, the contract's bytecode and binary application interface are generated from the code of the written contract. The bytecode is a binary code that the Ethereum Virtual Machine (see above) can understand and thus execute. The Application Binary Interface (ABI), on the other hand, is a JSON file that describes the provided contract and its functions. This ABI is then translated by a

JavaScript client library and is necessary to call the functions of the contracts in a web application interface [etha]. This is the reason why not every programming language can be used to write Ethereum Smart Contracts, otherwise the bytecode and the ABI cannot be generated properly.

The specific steps that need to be followed to deploy a smart contract on a blockchain depend on the used tools. However, in general, the deployment of a smart contract can be thought of as the creation of a new account on the blockchain, which is done by sending an Ethereum transaction with the compiled code of the contract to the chain without specifying a recipient. As the deployment is essentially a mere transaction, transaction fees must be paid to validate the transaction. Since the validation of a deployment is more extensive than that of a normal transaction, it is also more costly for the issuer of the transaction [ethb].

Smart contracts are limited by their inability to send HTTP requests and by the limit on the size of the bytecode generated when compiling the contract [ethe] . The first limitation was introduced to reduce the dependency of smart contracts on information from outside the chain, as this kind of information could be forged, while the second limitation was described above (see "Ethereum Virtual Machine (EVM)"). Another important limitation of smart contracts is that they cannot initiate transactions, only respond to them, as the transaction fees required to process the transaction must first be paid from an external account. This can be a bit of a hurdle in development as all interactions have to be initiated by the user. Therefore, developers need to keep this aspect in mind when writing smart contracts.

3.3 Accountability in smart contracts

The approaches presented in 2.2.1 of previous research to solve the accountability problem in blockchain-based solutions can basically be divided into a *reputation-based* and a *staking-based* approach.

Reputation-based approach

This approach is representative of services where users need to be identified by some kind of service, which may also be off-chain. This is necessary because otherwise, due to the anonymity that blockchain-based solutions offer, users could exploit the system by using another account to interact with the contract and thus reset their reputation. However, a major advantage of this approach is that with each misconduct, the penalty the user receives can be increased. Of the papers we have analysed e.g. [YL16], [ZKS$^+$19] and [TBOB18] fall into this category.

Staking-based approach

This approach is the other group into which we have classified the approaches we have analysed. It requires that a user stakes a certain value in the contract (usually the cryptocurrency of the underlying blockchain), but does not require the user to be identified. In the event of an offence, the guilty user is punished by a deduction of the value staked. This approach is somewhat easier to implement than the *reputation-based* approach and does not require identification, which does not compromise the decentralisation of the solution, as would be the case in off-chain identification. However, it has the disadvantage that penalties cannot be increased for each misconduct, as in this approach a user could simply use another account to interact with the contract after receiving the remaining staking of the first account from the contract. Of the analysed papers, e.g. [Gui20] can be placed in this category.

3.4 Implementation methodology

When implementing Ethereum-based smart contracts (see 3.2), developers have two main options of programming languages: Solidity [sol] (see 5.1) and Vyper [vyp]. Both are contract-oriented object-oriented programming (OOP) languages optimised for the development of smart contracts.

Gartner, Inc. defines OOP as "*..a style of programming characterized by the identification of classes of objects closely linked with the methods (functions) with which they are associated...*" [gar]. Contract-oriented refers to the emphasis that both languages place on the creation of smart contracts. These contracts can be seen as the objects that underlie the object orientation of such programmes.

3.5 Testing methodology

Since more extensive smart contracts need to be split into several smaller contracts in order to keep the generated bytecode of each contract below Ethereum's contract size limit (see 3.2), functional unit testing seemed ideal to us. Here, each contract can be considered as a single unit and thus be tested mostly completely autonomously from the proceedings in other contracts.

The test cases themselves (see 6) were then derived based on the nature of a multi-tier state machine (see 4), which our implementation basically has. Thereby, the focus was on the states that the smart contracts can be in and the different paths through which these states can be reached.

CHAPTER 4

Design

In this chapter, we describe the design we have conceptualised for our solution. In the course of this, firstly a description of how accountability of participants is achieved can be found. Subsequently, a description of the resulting processes of users and nodes is given.

4.1 Participant Accountability

A key aspect of a resource management system is to find a way to ensure that both parties involved in the contract do everything possible to achieve a mutually beneficial outcome. Therefore, a way must be found to punish participants who act in a malicious manner. In 3.3 we have categorised the approaches from related work, which we divided into a *reputation-based* and a *staking-based* approach in 2.2.1.

As stated in 1.2, the goals of our implementation were low entry cost, high adaptability and maximising the meaningful use of decentralisation. Since a reputation-based system requires either some form of off-chain identification, which limits decentralisation by creating a potential single point of failure, or a sophisticated contract, which complicates the code and thus limits adaptability and increases the cost of deploying the contract, we chose the *staking-based* approach to achieve user accountability.

4.1.1 Staking Economy

In our solution, adhering to the chosen *staking-based* approach, participants need to stake Ether with the contract in order to use the resource management system. This Ether, which we will refer to as staking, can then be deducted in the event of a participant's misconduct and sent to a predefined Ethereum address or, under certain conditions, withdrawn by the participant.

A distinction is made between three types of staking. Firstly, there is staking ether which can be moved back into the participant's EOW at any time and thus withdrawn from the contract. On the other hand, there is staking ether which cannot be reclaimed for a limited period of time. This occurs during the execution of an assignment and is reduced by the punishment policy if necessary. In the case of nodes, there is a third type of staking. This type is bound to resources that are currently made available to the network but not yet used. If the associated resources are deactivated by a node, this type of staking is converted back into the first type of staking, which can be reclaimed at any time.

When designing our system, we decided that each participant must have a certain amount of some type of staking stored in the contract. Otherwise, the participant's account in the contract is closed. This mainly serves to simplify the scheduling algorithm and to encourage participants to use the network service a lot, although the practical benefits of the latter still need to be investigated in depth.

In our eyes, this staking approach fully exploits the advantages of smart contracts in that the contract controls the flow of the staking provided completely autonomously and thus cannot be tricked.

4.1.2 Punishment Policy

Since we use the *staking-based* approach to ensure that participants do not misbehave, the punishment of a participant is formed by a deduction of the staked Ether. This involves deducting the staking of the penalised participant or participants if more than one needs to be penalised. This stake is then transferred either to the other party participating in the assignment that necessitated the punishment, who suffered from the misconduct, or to another recipient address if both parties to a stake have misbehaved and both are to be punished. This approach could even be used to incentivise the use of the smart contract by randomly distributing the collected penalty money to chosen participants upon successful completion of an assignment.

Currently, only one punishable offence is implemented in the contract: When a task is completed, the statements of the two parties (user and node) about the success of the respective task do not match. Since the contract is not able to identify who is telling the truth and who is not, both parties are penalised by having a certain amount of their staking deducted while the assignment's payment is returned to the user.

4.2 Process

4.2.1 User

In this section, the process followed by the User, whose role in our solution is to requests the resources that nodes provide, is described.

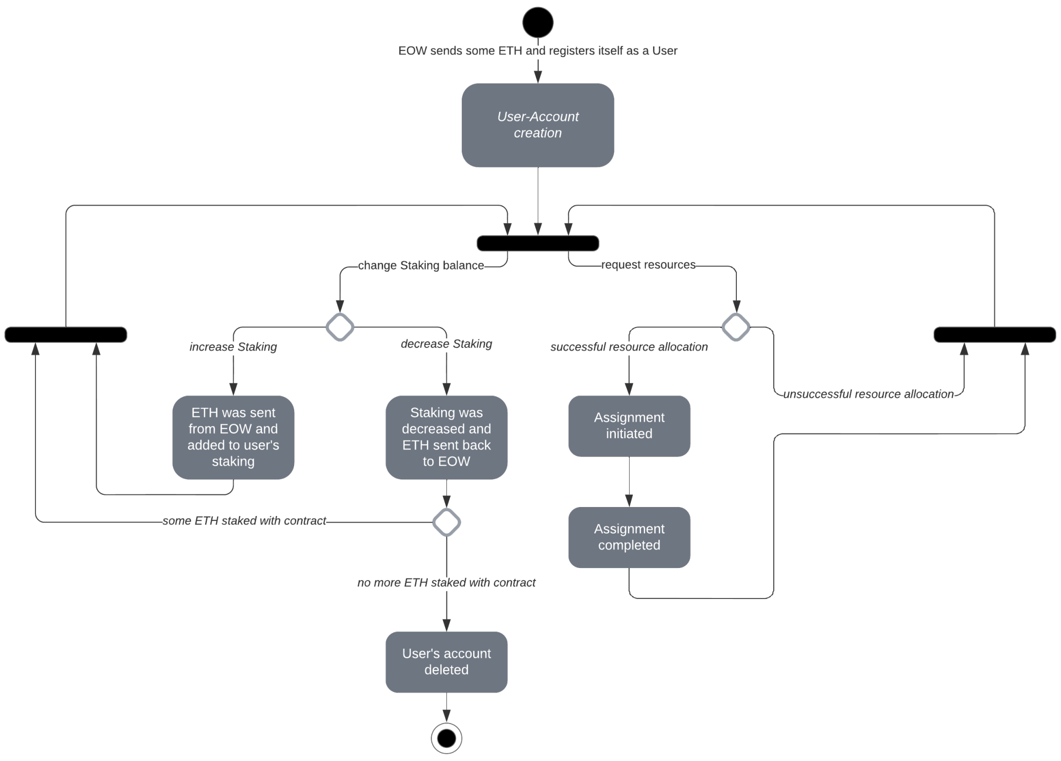
EOW sends some ETH and registers itself as a User
User-Account creation
change Staking balance
request resources
increase Staking
decrease Staking
successful resource allocation
unsuccessful resource allocation
ETH was sent from EOW and added to user's staking
Staking was decreased and ETH sent back to EOW
Assignment initiated
Assignment completed
some ETH staked with contract
no more ETH staked with contract
User's account deleted

Figure 4.1: Activity Diagram showing user's Process

The user's process in our smart contract (see Figure 4.1) begins with the registration of the user. In the course of the registration, the new user must deposit a certain amount of ether as staking in the contract. However, the user can withdraw this staking from the contract at any time, with the exception of the amount of staking needed to cover currently active assignments. If the user withdraws the entire amount of his staking from the contract (i.e. no current assignments), his account in the contract is closed. The user can also increase the amount of his Staking by adding external Ether to the contract as staking.

The core of the user process is requesting and using node resources. For this, it is checked whether the user currently has enough unbound staking available to cover the requested resources. If this is the case, a query is made for an available node, and if a node with sufficient resources can be found, an assignment is created. In an assignment (see Figure 4.2), the user must first pay a certain amount of ether for the processing of the assignment. The amount of this payment depends on the amount of resources requested. The contract retains the payment means during the processing of the assignment. After the payment, the user must send his input data to the selected node. This is done off-chain. The user can receive the address for this from the contract after the successful payment.

Thereupon, the user receives the nodes output, as before, over an off-chain channel, once the node is done processing the users input. Hereby, the output is send to the address from which the user initially sent his input. If the user subsequently accepts the output of the node, the payment funds previously sent by the user to the contract are transmitted to the node and the user receives his staking back in full. The assignment is considered successfully completed. Otherwise, the funds sent by the User are returned to the User and both parties are penalised with a deduction of their staking. The assignment is deemed to be unsuccessfully completed.

The funds sent back to the user by an assignment in both completion types are reassigned to the user's free Staking. It is therefore possible for the user to retrieve this staking from the contract or to use it for a new assignment.

4.2.2 Node

In this section, the process followed by the Node, whose role in our solution is to provide computation-resources to the users, is described.

The process that nodes follow in our smart contract (see Figure 4.3) begins with the registration of the nodes. Here, a new node specifies an amount of resources that it wants to make initially available to the network and must cover this with a certain amount of ether, its initial staking, which it sends to the contract in the registration transaction. If the transaction contains more than the required Ether, this excess Ether is also stored in the contract.

If a node wants to increase its staking, it can do so at any time. The Ether sent is then added to the excess Ether. However, if a node wants to reduce the staking stored in the contract, enough surplus staking must be available. Otherwise this is not possible.

A node can also change the number of resources made available to the network. In order to increase it, it must either have already stored enough surplus staking in the contract or have sent the required funds in the transaction that initiated the increase in the number of resources. Reducing resources is only possible if they are not currently allocated to a user.

The core of the process is the provision of resources to the user (see Figure 4.2). If a user makes a request, the first node from the queue of activated nodes that has enough available resources is selected. This forms is a very simple scheduling algorithm in the form of an adapted LIFO queue. If a node is selected for an assignment, it must define its address to which the user shall sent his input to after the user has sent his payment to the contract. It is not possible to change this address afterwards. As soon as the user has sent the input data off-chain to the specified address, the node announces this to the contract and starts processing the order. Once the node has completed processing the user's input, it sends its output back to the user via the same off-chain path and notifies the contract that it has finished processing. Depending on the user's response to the output, it then receives the payment and all of its staking tied to the provided resources

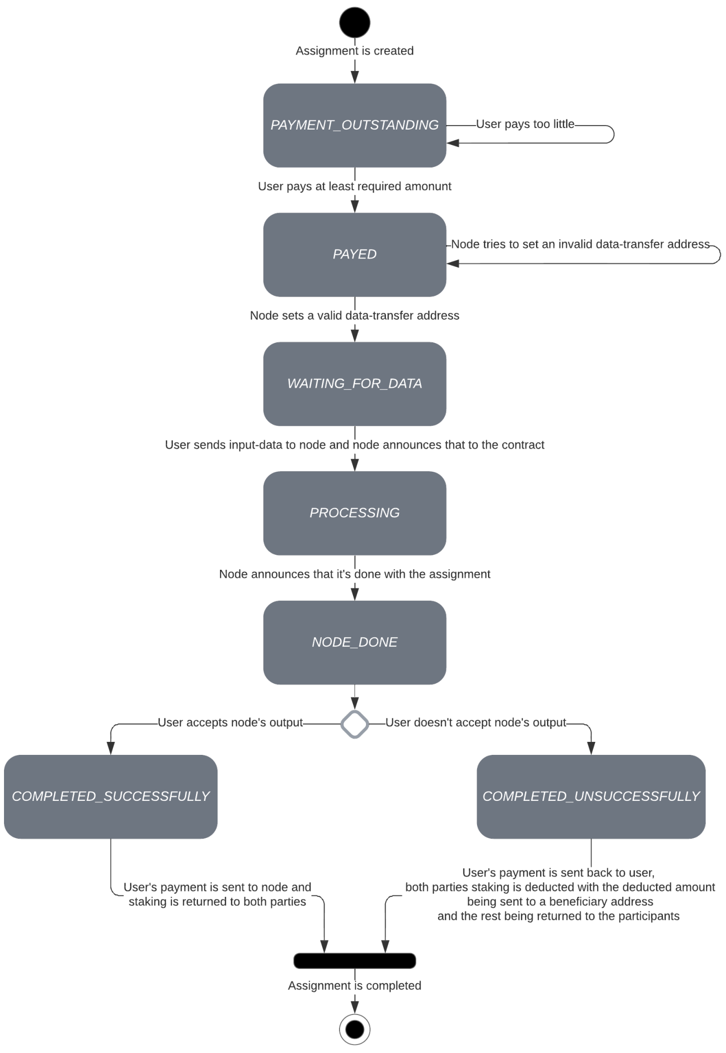
Assignment is created
PAYMENT_OUTSTANDING
User pays too little
User pays at least required amonunt
PAYED
Node tries to set an invalid data-transfer address
Node sets a valid data-transfer address
WAITING_FOR_DATA
User sends input-data to node and node announces that to the contract
PROCESSING
Node announces that it's done with the assignment
NODE_DONE
User accepts node's output
User doesn't accept node's output
COMPLETED_SUCCESSFULLY
COMPLETED_UNSUCCESSFULLY
User's payment is sent to node and staking is returned to both parties
User's payment is sent back to user, both parties staking is deducted with the deducted amount being sent to a beneficiary address and the rest being returned to the participants
Assignment is completed

Figure 4.2: Activity Diagram showing assignment's Process

back, or it does not receive the payment and only receives a portion of the staking back, as the remainder was withheld by the contract as a penalty for non-compliance between the two parties.

At the end of an assignment, the provided resources are always deactivated and must be reactivated by the node if the node wishes to provide them again. This decision was made to check that a node does indeed want to make resources available again, and to make it easier for a node to deactivate them.

Deactivating a node is only possible if it has no active assignments at the moment the deactivation is requested. If a node is successfully deactivated, all its staking is returned

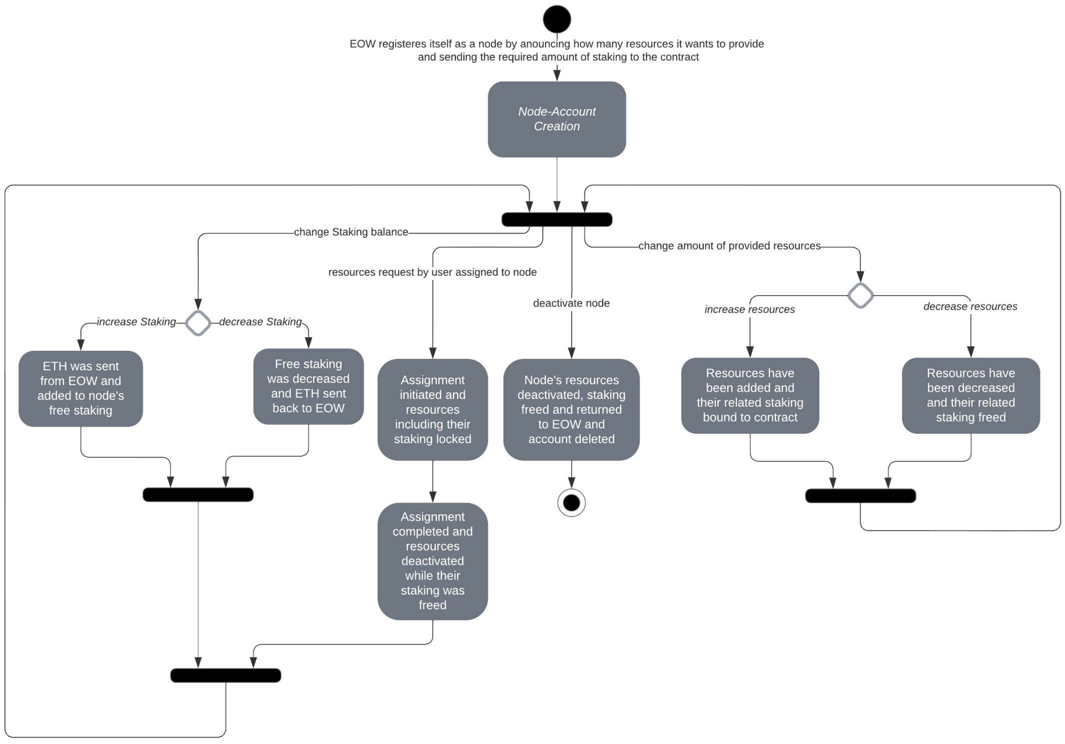
EOW registeres itself as a node by anouncing how many resources it wants to provide and sending the required amount of staking to the contract
Node-Account Creation
change Staking balance
resources request by user assigned to node
deactivate node
change amount of provided resources
increase Staking
decrease Staking
increase resources
decrease resources
ETH was sent from EOW and added to node's free staking
Free staking was decreased and ETH sent back to EOW
Assignment initiated and resources including their staking locked
Node's resources deactivated, staking freed and returned to EOW and account deleted
Resources have been added and their related staking bound to contract
Resources have been decreased and their related staking freed
Assignment completed and resources deactivated while their staking was freed

Figure 4.3: Activity Diagram showing Node's Process

to its EOW and its entry in the contract is deleted. If a node did not have to reactivate resources it wanted to make available again, as described in the previous paragraph, it could be that a node could not deactivate itself in the event of a high number of user requests because it would have continuously active assignments.

It should be mentioned here that this problem could also be addressed by two other approaches. On one hand, having to accept assignment would solve this problem, but would increase the gas consumption of nodes and the bytecode of the contracts. Another approach would be to allow nodes to initiate deactivation while still having active assignments. Then only assignments that were already active would be completed, whereby the nodes could no longer receive any new assignments and the node would then be deactivated as soon as all active assignments were completed. However, this would also drive up the bytecode. For this reason, we decided to use the resource reactivation method.

CHAPTER 5

Implementation

In this chapter we present the most important aspects of the implementation of the design presented in the previous chapter to give the reader a better up-front understanding of the implementation, which can be found in this Github Repository.

5.1 Description of utilised Tools

5.1.1 Visual Studio Code (Text Editor)

Visual Studio Code[vsC] is a desktop application for editing source code. Available for Windows, macOS and Linux, it has built-in support for JavaScript, TypeScript and Node.js. In addition, it has an extensive ecosystem of extensions for other programming languages. We have used the editor in our development process mainly as a text editor, but also to initiate deployment of contracts with Truffle and Ganache via the built-in console and to test the contracts with Remix, all of which have a visual studio code extension to facilitate development.

5.1.2 Solidity (Programming language)

For the implementation of the smart contract, we used the programming language Solidity. It is described as "*...a high-level object-oriented language for implementing smart contracts*"[sol]. The language was developed to work on the Ethereum Virtual Machine (EVM). It was influenced by the programming languages C++, Python and JavaScript. Solidity also supports a lot of features from other programming languages for this reason, such as inheritance, libraries and complex user-defined types. All of these features are used in our implementation.

5.1.3 Truffle Suite (Deployment of contracts)

The Truffle Suite is an ecosystem for smart contract development that includes several development tools. The tools that we used in the development of the implementation are:

Truffle

Truffle is described by the creators as a "*...development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM)...*"[trub]. Among other things, this gives developers the ability to automatically compile, link, deploy and test smart contracts. In addition, it gives them the ability to interact with contracts via a console, for which we used Remix [Subsection: 5.1.4] instead in our development process for the sake of simplicity.

Ganache

Ganache is a UI and CLI tool for creating a local blockchain that can be used to develop, deploy and test Web3 applications[gan]. During development, we used this tool to deploy our contracts, check the addresses of deployed contracts and issue events. In addition, the Ganache-UI tools allow developers to have a good overview of the status of the blockchain and a selection of EOWs (including their Ether balances) that are used to interact with the blockchain for testing purposes.

5.1.4 Remix (Testing)

Remix is an open source web and desktop application that can be used for the entire development process of a smart contract[rem] . Therefore, writing code, deploying contracts, interacting with deployed contracts and testing can all be done within the respective GUIs in Remix. In addition, Remix also has a Visual Studio Code extension that allows the Remix tools to be used within the text editor. In our development, we used Remix mainly for interacting with the contracts and for testing. However, we also used the Remix IDE web application to test ideas in a quick and easy way.

5.1.5 OpenZeppelin (Code-Library)

OpenZeppelin [ope] is an open-source platform that provides developers with various tools for the creation and automation of Web3 applications and security audits of smart contracts. The OpenZeppelin developers use the Solidity programming language mentioned above. In our implementation, we used OpenZeppelin to avoid having to develop a uint256-to-string conversion-function ourselves, but to simply import it from the OpenZeppelin library.

5.2 Splitting of the Contract

To save bytecode and to make the contracts more comprehensible, we have divided the smart contract into three smaller smart contracts that interact with each other. Therefore, there is one contract for the user side, one contract for the node side and one contract for the execution of assignments. As can be seen in Figure 5.1 , a *UserContract* and a *NodeContract* are directly connected with each other and can only have one such connection. This connection is created at the time of migration and cannot be changed afterwards. The *AssignmentContract* is migrated from the *NodeContract* when a new assignment is created, which is initiated by a user in the *UserContract*. A new *AssignmentContract* is created for each Assignment. The *AssignmentContract* is connected to exactly one *UserContract* and one *NodeContract*. Furthermore, in the current implementation only the connection with one user and one node is possible.

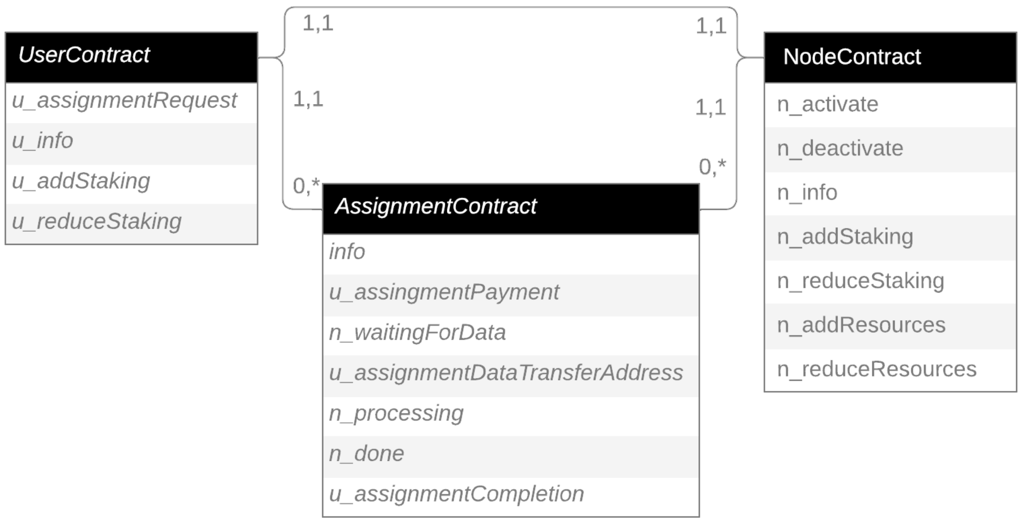

Figure 5.1: ERD Diagram showing the 3 contracts and their functions

5.3 Libraries & Interfaces

In the course of the implementation, we used the possibility which Solidity gives developers to define their own libraries and interfaces to guarantee our goal of a high adaptivity of the contract.

The library "*ParticipantsLibrary*" defines the states that an assignment can take, possible event types, constants, custom errors and some functions that do not require memory access and are used by more than one contract. When using and adapting the contract, there are a few things to keep in mind when changing constants.

1. the constant *PERCENTAGE_RESOURCES_TO_STAKING* must be set significantly larger than the constant *PERCENTAGE_RESOURCES_TO_PAYMENT*, since the former defines the number of the necessary staking per resource in percent and the latter the necessary payment amount per resource. If the first constant were not chosen significantly larger than the second, a user would benefit more from getting his payment back than from getting his staking back. In this case, a user would be tempted to accept every assignment as correct even if an assignment was not executed correctly.

2. the constant *PERCENTAGE_STAKING_PUNISHMENT*, that defines the amount of staking that is to be deducted if a punishment is to be enacted, should also be chosen with care, because in the same way as in the previous case, in the worst case, it invalidates any sense of checking a successful assignment execution.

The necessary interfaces are listed in the interface-file "*Interfaces*". The interfaces with ending with *..._internal_interface* are used by the contracts to call contract functions from each other, while those ending with *..._external_interface* are to be understood as user interfaces and describe and define the functions that EOWs can call.

5.4 Data structures

5.4.1 User

The users are stored in the *UserContract.* A single user is defined through a mapping (which is an implementation in solidity of a hash-table) of the user's EOW-address to the user's structure inside of the *UserContract.* This structure contains the number of *freeStaking*, which is the amount of staking currently not locked in an assignment, and *lockedStaking*, which is the amount of staking currently locked in an assignment, that the user currently has stored in the contract. In addition, the structure also contains an array of addresses representing the addresses of the user's currently active assignments. Any number of users can be stored with the contract, as mappings in solidity have an arbitrary size. This does not affect the cost of operations on the users data structure, as entries from mappings can be addressed directly and do not have to be searched for first.

5.4.2 Node

Just as with the user, nodes are also stored with a mapping from address to struct. The underlying struct, however, is somewhat larger. Since the node offers resources, currently unbound resources and resources currently bound by assignments are stored here as *freeResources* and *lockedResources.* Another difference to the user structure is the so-called *boundStaking.* This is used to describe the amount of staking that is bound to resources currently offered by the node but not used by users. The remaining variables in the node-struct are identical to those of the user: a list of active assignments in the form of an array of their addresses, the *freeStaking*, which here describes the amount of staking not bound to resources, and *lockedStaking*, which describes the amount of staking bound to currently active assignments.

5.4.3 Assignment

The data structure of an assignment can be found in the *AssignmentContract*. Since there can only be one assignment per *AssignmentContract* and therefore no mapping is necessary to store any number of assignments, we have implemented this data structure without a struct. Instead, it consists only of several variables, which (with the exception of *nodeDataTransferAddress*) are all set in the constructor when the contracts are migrated. *UserContractAddress*, *NodeContractAddress*, *UserContractInstance* and *NodeContractInstance* are used to access functions that handle the completion of an assignment in the connected *UserContract* and *NodeContract*. The current state in which the assignment is located is stored in the variable state. This is of the enum type *assignmentState*, which is defined in the *ParticipantsLibrary* (see 5.3). In addition, the data structure stores the previously mentioned *nodeDataTransferAddress*, i.e. the address to which a user should send the data that the node has to process. This is set by the node during the assignment-process. The remaining variables store the amount of resources, the amount of payment and the insurance cost (participants' *lockedStaking*) necessary to pay and cover the requested resources in the variables *resources*, *payment* and *staking*.

5.5 Scheduling of Assignments

In the current implementation, the scheduling of assignments is still kept quite simple. The scheduling is done by a queue of the nodes currently stored in the *NodeContract*, which is iterated over and thus an entry for which the *freeResources* are greater than or equal to the requested resources is identified.This queue can be found in the *NodeContract* under the name *resourceSchedule* as an array of addresses. When a node is activated, the address of the new node is stored at the end of the array and therefore of the queue. If a user requests a certain amount of resources, the array is iterated through from index 0 until a node address points to a node-struct with sufficient free resources. Once such an entry is found, the required resources are subtracted from freeResources and added to *lockedResources*. Furthermore, the address of the selected node is moved to the end of the queue and the gap is closed by shifting the following entries to the left. If no such entry could be found, that information is returned to the calling user but otherwise nothing is changed. A node is only completely deleted from the queue if it deactivates itself (by calling the function *n_deactivate*) and thus terminates its connection to the *NodeContract*. Of course, this has the disadvantage that nodes with no free resources are also contained in the queue and drive up the number of necessary iterations. On the other hand, this helps the nodes to save Gas if a node can only offer no resources for a short time. For more in-depth explanations of the design decisions made, see the chapter Design (see Chapter 4).

5.6 Assignment Execution

The following figure 5.2 shows a possible flow of an assignment from the registration of both parties with their respective contracts.

It should be noted that the function *u_assignmentRequest*() calls the function *c_resource-Request*() and thus the user indirectly interacts with the *nodeContract*. If this request can be processed by a node due to the currently available resources (as shown in Figure 5.2) an instance of the *AssignmentContract* is migrated. Both parties then interact directly with this migrated contract.

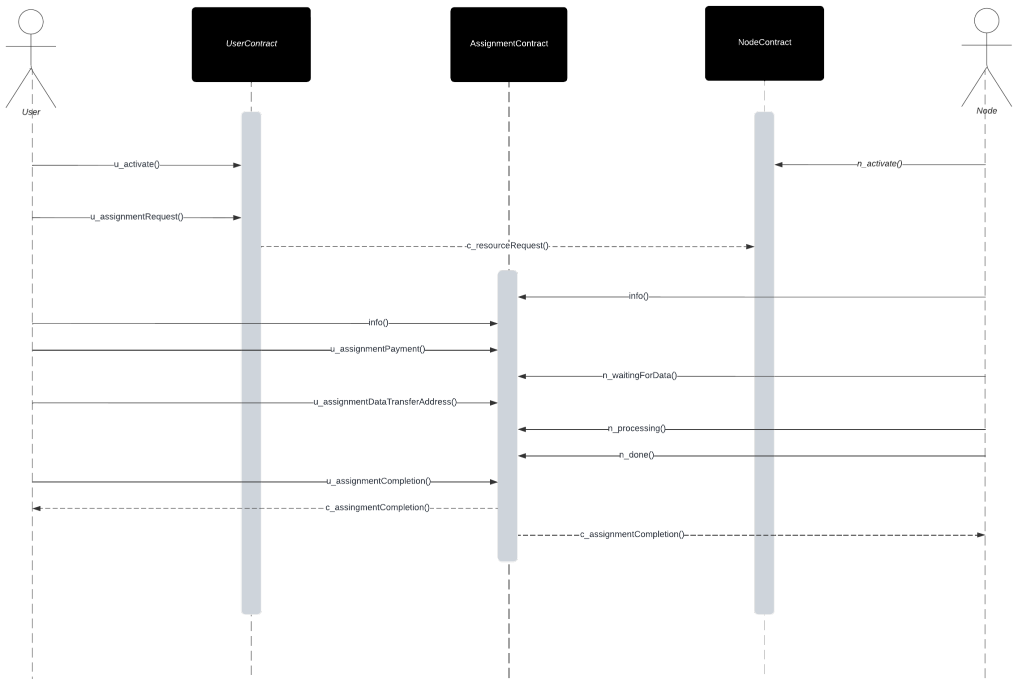

Figure 5.2: UML Sequence Diagram showing an execution scene of an assignment from request to completion

Both participants' contracts are only used again when an assignment is completed to adjust the staking- and resource- balances of both parties according to the successful or unsuccessful completion of that assignment, which is done by the transaction *c_assignmentCompletion*() that is triggered by *u_assignmentCompletion*(). In the course of this function, ether of the deducted staking is sent to the BENEFICIARY_ADDRESS, if it was created due to an unsuccessful assignment completion. Otherwise, however, the staking-ether remains in the respective contracts of both parties in order to save gas by avoiding unnecessary transfers to and from the *assignmentContract*.

CHAPTER 6

Testing

In this chapter we describe the procedure we adopted to test our implementation. Starting with a description of the testing process, we then briefly describe each of the individual tests that were carried out.

6.1 Process

For testing, we used a combination of Visual Studio Code (vscode), the Truffle Suite and Remix, which were described in [Section: 5.1]. The contracts were deployed to a local Ganache network in Visual Studio Code using the vscode truffle extension, to which we then interacted with the deployed contracts via the Remix extension for Visual Studio Code and monitored the output of the transactions therein. We monitored the events emitted by the contracts and the balances of the simulated EOWs in Ganache' GUI.

6.2 Test Cases

Below a brief description of the individual test cases of functions that can be called externally and the internal functions that they call can be found. The values of the constants used and the exact inputs and outputs can be found here.

6.2.1 Test Cases for UserContract

u_assignmentRequest

UserContract functions:

- *i_assignmentCheckStaking*
- *i_assignmentCheckResources*

- *i_assignmentReturnStaking*

NodeContract functions:

- *c_resourceRequest*
  internal functions:
  - *i_scheduleLookForAvailableEntry*
  - *i_scheduleDeleteEntry*
  - *i_resourceManagement*
  - *i_stakingManagement*)

Test cases:

1. EOW calls with parameter resources = 0, which is invalid and hence the transaction has to be reverted with the custom error: *invalidInput*()
2. Unregistered node calls with msg.value = 0, which is invalid and hence the transaction has to be reverted with the custom error: *insufficientFundsSent*()
3. Unregistered user calls with not enough msg.value to cover the staking of the requested resources which leads to the user being registered with the sent value added to that user's *freeStaking* but no assignment being initiated
4. Unregistered user calls with enough msg.value to cover the staking of the requested resources but not enough resources being currently available which leads to the user being registered with the sent value added to that user's *freeStaking* and an unsuccessful search for available resources.
5. Registered user calls with no msg.value contained in the call to cover the staking of the requested resources but having already enough *freeStaking* staked in the contract while no resources being currently available which leads to no change in the user's storage entry and an unsuccessful search for available resources.
6. Registered user calls with msg.value > 0 contained in the transaction, but requesting more resources than the staking contained in user's *freeStaking* and msg.value together, which leads to the msg.value being added to that user's *freeStaking* but no search for available Resources
7. User requests resources after providing enough staking (either through msg.value or through *freeStaking*) while enough resources are currently available. This leads to a search for available resources, the allocation of those resources by locking user's *freeStaking* and node's *boundStaking* and the *freeResources* that user uses now.

u__info

Test cases:

1. Unregistered user calls functions which leads to a revert with the error message "This function is restricted to registered users"
2. A registered user without active assignments calls the function which leads to that user's *freeStaking*, *lockedStaking* and an empty array in place of that user's *activeAssignments* being returned
3. A registered user with active assignments calls the function which leads to that user's *freeStaking*, *lockedStaking* and addresses of the currently active assignments of the user being returned.

u__addStaking

Test cases:

1. Unregistered user calls functions which leads to a revert with the error message "This function is restricted to registered users"
2. Registered user calls with no msg.value contained in the call which is invalid and hence the transaction has to be reverted with the custom error: *insufficient-FundsSent*
3. Registered user calls with a msg.value > 0 contained in the call which leads to that value being added to that user's *freeStaking*

u__reduceStaking

Test cases:

1. Unregistered user calls functions which leads to a revert with the error message "This function is restricted to registered users"
2. Registered user calls with the parameter *amount* being set to 0 which leads to a custom error: *invalidInput* being triggered
3. Registered user calls the function with the parameter *amount* set higher than the amount of user's *freeStaking*, which leads to a custom error: *insufficientFundsSent* being triggered.
4. Registered user calls the function with the parameter *amount* set lower than the amount of user's *freeStaking* which leads to that specified amount of staking being returned to the user's EOW-address.

5. Registered user calls the function with the parameter *amount* set to the exact amount of that user's *freeStaking* while that user's *lockedStaking* has the value 0. This leads to the closure of that user's account since no ETH of that EOW is stored with the contract anymore.

6. Registered user calls the function with the parameter *amount* set to the exact amount of that user's *freeStaking* while that user's *lockedStaking* does not have the value 0. This causes that user's *freeStaking* to be set to 0, while that user's entry in the storage remains active.

6.2.2 Test Cases for NodeContract

n_activate

NodeContract functions:

- *i_resourceManagement*

Test cases:

1. EOW calls with parameter resources = 0, which is invalid and hence the transaction has to be reverted with the custom error: *invalidInput*()

2. Unregistered node calls with a msg.value that is lower than the required amount of staking derived by the resource-to-staking conversion of the *resources*-parameter. Hence the transaction has to be reverted with the custom error: *insufficient-FundsSent*()

3. Unregistered node registers itself successfully by calling the function with a msg.value that is higher than the required amount of staking derived by the resource-to-staking conversion of the *resources*-parameter

4. Registered node calls the function while already being registered, which leads to a revert with the custom error: *alreadyRegistered*()

n__deactivate

NodeContract functions:

- *i__scheduleDeleteEntry*

Test cases:

1. Unregistered node calls functions which leads to a revert with the error message "This function is restricted to registered nodes"
2. Registered node with active Assignments calls function which returns a custom Error *deactivateActiveAssignments*()
3. Registered node without active Assignments calls function while no other node is registered in order to test deletion of Node from *resourceSchedule* if that leads to *resourceSchedule* being empty
4. Registered node without active Assignments calls function while other nodes are registered in order to test correct deletion of Node from *resourceSchedule* while the queue is not empty.

n__info

Test cases:

1. Unregistered node calls functions which leads to a revert with the error message "This function is restricted to registered nodes"
2. Registered node without active Assignments to test return of empty *activeAssignments*
3. Registered node without active Assignments to test return of non-empty *activeAssignments*

n__addStaking

Test cases:

1. Unregistered node calls functions which leads to a revert with the error message "This function is restricted to registered nodes"
2. Registered node calls with no msg.value contained in the call which is invalid and hence the transaction has to be reverted with the custom error: *insufficientFundsSent*()
3. Registered node calls with a msg.value > 0 contained in the call which leads to that value being added to that user's *freeStaking*

n__reduceStaking

Test cases:

1. Unregistered node calls functions which leads to a revert with the error message "This function is restricted to registered nodes"

2. Registered node calls with the parameter *amount* being set to 0 which leads to a custom error: *invalidInput* being triggered

3. Registered node calls the function with the parameter *amount* set higher than the amount of node's *freeStaking*, which leads to a custom error: *insufficientFundsSent* being triggered.

4. Registered node calls the function with the parameter *amount* set lower than the amount of node's *freeStaking* which leads to that specified amount of staking being returned to the node's EOW-address.

5. Registered node calls the function with the parameter *amount* set to the exact amount of that node's *freeStaking* while that node's *lockedStaking* or *boundStaking* does not the value 0. This causes that node's *freeStaking* to be set to 0

6. Registered node calls the function with the parameter *amount* set to the exact amount of that node's *freeStaking* while that node's *lockedStaking* and *boundStaking* have the value 0. This causes the custom error *invalidInput*() to be thrown.

n__addResources

Test cases:

1. Unregistered node calls functions which leads to a revert with the error message "This function is restricted to registered nodes"

2. Registered node calls with the parameter *amount* set to 0 which is invalid and hence the transaction has to be reverted with the custom error: *invalidInput*()

3. Registered node calls with the parameter *amount* set to a value that would require a greater amount of *freeStaking* to be added to that node's *boundStaking* than the amount of staking currently available in *freeStaking* which causes a custom error *insufficientFreeStaking*() to be thrown.

4. Registered node calls with the parameter *amount* set to a value that would require less or exactly the amount of *freeStaking* to be added to that node's *boundStaking* that is currently available in *freeStaking* which leads to the operation being executed.

n__reduceResources

Test cases:

1. Unregistered node calls functions which leads to a revert with the error message "This function is restricted to registered nodes"
2. Registered node calls with the parameter *amount* being set to 0 which leads to a custom error: *invalidInput* being triggered
3. Registered node calls the function with the parameter *amount* set higher than the amount of node's *freeResources*, which leads to the currently available amount of resources being deactivated.
4. Registered node calls the function with the parameter *amount* set lower than the amount of node's *freeResources* which leads to that specified amount of resources being deactivated and the *boundStaking* connected to those resources being returned to *freeStaking*.

6.2.3 Test Cases for AssignmentContract

info

Test cases:

1. Some EOW calls the function which leads to the assignment's resources, staking, payment and currentState being returned.

u__assingmentPayment

AssignmentContract functions:

- *c__updateState*
- *c__checkInputAddress*

Test cases:

1. Someone else than user calls functions which leads to a revert with the error message "This function is restricted to the connected user"
2. User calls function while *state* is set to *PAYMENT__OUSTANDING* with msg.value $<$ *requiredValue* which leads to the transaction being reverted with the custom error: *insufficientFundsSent*()
3. User calls function while *state* is set to *PAYMENT__OUTSTANDING* and msg.value $>=$ *requiredValue* which leads to the assignment's *state* being changed to *PAYED*.

4. User calls function while *state* is not set to *PAYMENT_OUTSTANDING* which leads to the custom error *assignmentHasDifferentState*() being triggered.

n_waitingForData

AssignmentContract functions:

- *c_updateState*
- *c_checkInputAddress*

Test cases:

1. Someone else than node calls functions which leads to a revert with the error message "This function is restricted to the connected node"
2. Node calls function while *state* is set to *PAYED* with the parameter *DataTransfer-Address* being set to an invalid String which leads to the transaction being reverted with the custom error: *invalidInput*()
3. Node calls function while *state* is set to *PAYED* with a valid *DataTransfer-Address*-string which leads to the assignment's *state* being changed to *WAIT-ING_FOR_DATA* and the *nodeDataTranferAddress* being set to the passed *Dat-TranferAddress.*
4. Node calls function while *state* is not set to *PAYED* which leads to the custom error *assignmentHasDifferentState*() being triggered.

u_assignmentDataTranferAddress

Test cases:

1. Someone else than user calls functions which leads to a revert with the error message "This function is restricted to the connected user"
2. User calls function while *state* is set to *WAITING_FOR_DATA* which leads to *nodeDataTranferAddress* being returned to the user.
3. User calls function while *state* is not set to *WAITING_FOR_DATA* which leads to the custom error *assignmentHasDifferentState*() being triggered.

n_processing

AssignmentContract functions:

- *c_updateState*

Test cases:

1. Someone else than node calls functions which leads to a revert with the error message "This function is restricted to the connected node"
2. Node calls function while *state* is set to *WAITING_FOR_DATA* which leads to the assignment's *state* being changed to *PROCESSING*.
3. Node calls function while *state* is not set to *WAITING_FOR_DATA* which leads to the custom error *assignmentHasDifferentState*() being triggered.

n_done

AssignmentContract functions:

- *c_updateState*

Test cases:

1. Someone else than node calls functions which leads to a revert with the error message "This function is restricted to the connected node"
2. Node calls function while *state* is set to *PROCESSING* which leads to the assignment's *state* being changed to *NODE_DONE*.
3. Node calls function while *state* is not set to *PROCESSING* which leads to the custom error *assignmentHasDifferentState*() being triggered.

u_assignmentEnd

AssignmentContract functions:

- *c_updateState*
- *c_remainingStakingCalculation*

UserContract functions:

- *c_assignmentCompletion*

NodeContract functions:

- *c_assignmentCompletion*

Test cases:

1. Someone else than user calls functions which leads to a revert with the error message "This function is restricted to the connected user"
2. User calls function while *state* is set to *NODE_DONE* with the parameter *outputCorrect* set to 1 which leads to the node's *lockedResources* being deactivated, both parties staking being returned to their *freeStaking* and the assignment's payment being sent to the node's EOW-address.
3. User calls function while *state* is set to *NODE_DONE* with the parameter *outputCorrect* set to 0 which leads to the node's *lockedResources* being deactivated, both parties staking being deducted by a certain amount while the rest is being returned to their *freeStaking* and the assignment's payment being sent back to the user's EOW-address.
4. User calls function while *state* is not set to *NODE_DONE* which leads to the custom error *assignmentHasDifferentState*() being triggered.

CHAPTER 7

Discussion

In this chapter, the length of the bytecode of the implement Smart Contract and the gas costs of an exemplary use of the contract are analysed.

7.1 Bytecode

Table 7.1 shows the sizes of the bytecodes of the different contracts. These were determined with the Truffle plugin "contract-size" [trua]. It is visible, that the *ParticipantsLibrary* has a quite short bytecode, since it only contains constants and definitions. The *AssignmentContract* also has a rather short bytecode, as it basically only implements a simple state machine. It is noticeable that the bytecodes of the *UserContract* and the *NodeContract* are quite different in size. This is due to the fact that, on the one hand, the entire resource management is implemented in the *NodeContract* and, on the other hand, that the entire *AssignmentContract* has to be imported into the *NodeContract*, as it is migrated to the Blockchain in the *NodeContract* when an assignment is successfully created. This leads to the fact that the bytecode of the *NodeContract* is quite close to the contract size limit of 24576 bytes (see 3.2). For this reason, it is important to be careful

Contract	**Bytecode**
Migrations	0.79 KiB
ParticipantsLibrary	0.08 KiB
AssignmentContract	5.47 KiB
NodeContract	21.29 KiB
UserContract	9.46 KiB

Table 7.1: Bytecode-Size of Contracts

Description of the processed Operation	**Gas Cost**
Migrations	
Initial Migration of the Contracts	0.14423502 Ether
UserContract	
Successful assignment request of 50 resources and 100000 Wei contained in transaction as staking	1657708 Wei
Reduce Staking of User by 20000 Wei	48081 Wei
Increase Staking of User by 20000 Wei	38294 Wei
User deducts 100000 Wei and thereby closes it's account in the UserContract	42140 Wei
NodeContract	
Registration of Node with 100 Resources and 100000 Wei as Staking	155670 Wei
Reduce Staking of Node by 20000 Wei	48074 Wei
Increase Staking of Node by 20000 Wei	38290 Wei
Increase Resources of Node by 30	63637 Wei
Decrease Resources of Node by 30	64527 Wei
Node deactivates itself	35927 Wei
AssignmentContract	
User pays 50000 for an successful assignment request of 50 resources (150 Wei would have been needed)	55354 Wei
Node sets it's *dataTransferAddress* to "abcd"	52841 Wei
Node changes the assignment's state to PROCESSING	30273 Wei
Node changes the assignment's state to NODE_DONE	30295 Wei
User announces that node completed assignment correctly	90212 Wei
User announces that node completed assignment incorrectly	96843 Wei

Table 7.2: Exemplary Gas-Usage of the contracts' externally callable functions

when implementing possible additional functions and to possibly consider redividing the contract.

7.2 Gas costs

Table 7.2 shows a number of exemplary interactions with the contract and their Gas costs. It should be noted that these can vary depending on the Ether received in transactions or the current state of the contract (for example, deleting a node from an otherwise empty scheduling array vs. deleting it from a non-empty array), as other computing operations may be necessary. The most costly operations are the registration of users and nodes, as mappings and arrays have to be edited, and the *u_assignmentCompletion* function, which is used to complete assignments, as in the course of this function the *AssignmentContract*, in which it is called, is terminated and staking- and resource- balances have to be managed for both participants.

CHAPTER 8
Conclusion

In conclusion, our solution has fulfilled the goal of proving that a blockchain-based resource allocation system for cloud computing can work reasonably well, and our goal of creating an adaptable solution with a low barrier to entry has been achieved.

However, there are also aspects that our solution does not cover and that need to be implemented in order to make our implementation viable for real market use. One of these aspects is the perception of time when it comes to processing assignments. Without this, it is possible that the execution of an assignment would be starved if one of the partners of the assignment does not call the functions needed to advance the execution of the assignment. A perception of time could be achieved by using the number or timestamp of the current block. A timestamp would be the more sensible solution in our eyes, as gives more meaningful information about the state of the outside world. However, it should be noted that timestamps can be falsified by miners within a certain time frame and therefore a certain minimum granularity must be maintained when using timestamps.Checking that participants perform their tasks in an assignment within a certain time frame results in the possibility that they fail to do so.For this reason, in this case, more penalties need to be introduced, which can be enforced if time conditions are not met.

Another aspect that real market use could require is the ability for users to set a price that they are willing to pay for resources, or a supply and demand driven pricing model to set a global price for a given point in time.

Our solution could therefore be seen as a first step on an even longer path.

CHAPTER 9

References

List of Figures

List of Tables

Bibliography

[AGH18] Khadija Akherfi, Micheal Gerndt, and Hamid Harroud. Mobile cloud computing for computation offloading: Issues and challenges. *Applied computing and informatics*, 14(1):1–16, 2018.

[aws] Amazon webservices. `https://aws.amazon.com`.

[AZH18] Mohammad Aazam, Sherali Zeadally, and Khaled A Harras. Offloading in fog computing for iot: Review, enabling technologies, and research opportunities. *Future Generation Computer Systems*, 87:278–289, 2018.

[B⁺14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.

[BB22] Chaimade Busayatananphon and Ekkarat Boonchieng. Financial technology defi protocol: A review. In *2022 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunications Engineering (ECTI DAMT & NCON)*, pages 267–272, Jan 2022.

[btc] Bitcoin wiki's "contract" entry. `https://en.bitcoin.it/wiki/Contract`.

[CLMS20] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE access*, 8:85714–85728, 2020.

[etha] Compiling smart contracts. `https://ethereum.org/en/developers/docs/smart-contracts/compiling/`.

[ethb] Deploying smart contracts. `https://ethereum.org/en/developers/docs/smart-contracts/deploying/#steps-to-deploy`.

[ethc] Etherum upgrades website. `https://ethereum.org/en/upgrades/`.

[ethd] Intro to ether. `https://ethereum.org/en/developers/docs/intro-to-ether/#what-is-a-cryptocurrency`.

[ethe] Introduction to smart contracts. `https://ethereum.org/en/smart-contracts/`.

[gan] Ganache documentation. `https://trufflesuite.com/docs/ganache/`.

[gar] Object-oriented programming - definition by gartner, inc. `https://www.gartner.com/en/information-technology/glossary/oop-object-oriented-programming`.

[GJ15] Akhil Gupta and Rakesh Kumar Jha. A survey of 5g network: Architecture and emerging technologies. *IEEE access*, 3:1206–1232, 2015.

[GN21] Jianhu Gong and Nima Jafari Navimipour. An in-depth and systematic literature review on the blockchain-based approaches for cloud computing. *Cluster Computing*, pages 1–18, 2021.

[goo] Google cloud services. `https://cloud.google.com`.

[Gui20] Barbara Guidi. When blockchain meets online social networks. *Pervasive and Mobile Computing*, 62:101131, 2020.

[HR17] Garrick Hileman and Michel Rauchs. Global cryptocurrency benchmarking study. *Cambridge Centre for Alternative Finance*, 33:33–113, 2017.

[HWN12] Dong Huang, Ping Wang, and Dusit Niyato. A dynamic offloading algorithm for mobile computing. *IEEE Transactions on Wireless Communications*, 11(6):1991–1995, June 2012.

[HYYS20] Xumin Huang, Dongdong Ye, Rong Yu, and Lei Shu. Securing parked vehicle assisted fog computing with blockchain and optimal smart contract design. *IEEE/CAA Journal of Automatica Sinica*, 7(2):426–441, March 2020.

[ISFZ21] Noyan Ilk, Guangzhi Shang, Shaokun Fan, and J Leon Zhao. Stability of transaction fees in bitcoin: a supply and demand perspective. *MIS Quarterly*, 45(2), 2021.

[Jam] Hudson Jameson. Hard fork no. 4: Spurious dragon. `https://blog.ethereum.org/2016/11/18/hard-fork-no-4-spurious-dragon/`.

[JWNS19] Yutao Jiao, Ping Wang, Dusit Niyato, and Kongrath Suankaewmanee. Auction mechanisms in cloud/fog computing resource allocation for public blockchain networks. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):1975–1989, 2019.

[Liu21] Xing Liu. Towards blockchain-based resource allocation models for cloud-edge computing in iot applications. *Wireless Personal Communications*, pages 1–19, 2021.

[LWX01] Zhiyuan Li, Cheng Wang, and Rong Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246, 2001.

[msf] Microsoft azure. `https://azure.microsoft.com/en-gb/`.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[NHN$^+$19] Cong T. Nguyen, Dinh Thai Hoang, Diep N. Nguyen, Dusit Niyato, Huynh Tuong Nguyen, and Eryk Dutkiewicz. Proof-of-stake consensus mechanisms for future blockchain networks: Fundamentals, applications and opportunities. *IEEE Access*, 7:85727–85745, 2019.

[ope] Openzeppelin website. `https://www.openzeppelin.com`.

[QYWW18] Rui Qin, Yong Yuan, Shuai Wang, and Fei-Yue Wang. Economic issues in bitcoin mining and blockchain research. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 268–273. IEEE, 2018.

[rem] remix documentation. `https://remix-ide.readthedocs.io/en/latest/#`.

[SD16] Weisong Shi and Schahram Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, 2016.

[SJD16] Mayra Samaniego, Uurtsaikh Jamsrandorj, and Ralph Deters. Blockchain as a service for iot. In *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 433–436, Dec 2016.

[sol] Solidity documentation. `https://docs.soliditylang.org/en/v0.8.14/`.

[SSHKMFM22] Maryam Sheikh Sofla, Mostafa Haghi Kashani, Ebrahim Mahdipour, and Reza Faghih Mirzaee. Towards effective offloading mechanisms in fog computing. *Multimedia Tools and Applications*, 81(2):1997–2042, 2022.

[SSN$^+$20] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang, and David Mohaisen. Exploring the attack surface of blockchain: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 22(3):1977–2008, 2020.

[sta] Starlink webpage. `https://www.starlink.com`.

[SWHL16] Ivan Stojmenovic, Sheng Wen, Xinyi Huang, and Hao Luan. An overview of fog computing and its security issues. *Concurrency and Computation: Practice and Experience*, 28(10):2991–3005, 2016.

[TBOB18] Mona Taghavi, Jamal Bentahar, Hadi Otrok, and Kaveh Bakhtiyari. Cloudchain: A blockchain-based coopetition differential game model for cloud computing. In *International Conference on Service-Oriented Computing*, pages 146–161. Springer, 2018.

[trua] truffle-contract-size plugin. `https://github.com/IoBuilders/truffle-contract-size`.

[trub] Truffle documentation. `https://trufflesuite.com/docs/truffle/`.

[vsC] Visual studio code documentation. `https://code.visualstudio.com/docs`.

[vyp] Vyper documentation. `https://vyper.readthedocs.io/en/stable/`.

[WPE$^+$19] Jianyu Wang, Jianli Pan, Flavio Esposito, Prasad Calyam, Zhicheng Yang, and Prasant Mohapatra. Edge cloud offloading algorithms: Issues, methods, and perspectives. *ACM Comput. Surv.*, 52(1), feb 2019.

[XFW$^+$19] Zehui Xiong, Shaohan Feng, Wenbo Wang, Dusit Niyato, Ping Wang, and Zhu Han. Cloud/fog computing resource management and pricing for blockchain networks. *IEEE Internet of Things Journal*, 6(3):4585–4600, June 2019.

[Yak18] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper*, 2018.

[YL16] Affan Yasin and Lin Liu. An online identity and smart contract management system. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 192–198. IEEE, 2016.

[ZHZ$^+$21] Jinglin Zou, Debiao He, Sherali Zeadally, Neeraj Kumar, Huaqun Wang, and Kkwang Raymond Choo. Integrated blockchain and cloud computing systems: A systematic survey, solutions, and challenges. *ACM Comput. Surv.*, 54(8), oct 2021.

[ZKS$^+$19] Yuanyu Zhang, Shoji Kasahara, Yulong Shen, Xiaohong Jiang, and Jianxiong Wan. Smart contract-based access control for the internet of things. *IEEE Internet of Things Journal*, 6(2):1594–1605, April 2019.