Leveraging Blockchain Technologies to Build Short-Term Energy Flexibility Markets

Master's Thesis

by

Michell Boerger

Matriculation Number: 347864

submitted to

Technische Universität Berlin
Faculty IV - Electrical Engineering and Computer Science
Department of Telecommunication Systems
Open Distributed Systems

December 21, 2020

Supervisor: Prof. Dr. Manfred Hauswirth
TU Berlin - Open Distributed Systems

Co-Supervisor: Prof. Dr. rer. nat. Volker Markl
TU Berlin - DIMA Group

Contact details: Michell Boerger
Pichelsdorfer Str. 126
13595 Berlin
michell.boerger@campus.tu-berlin.de

Eidestattliche Erklärung / Statutory Declaration

Hiermit versichere ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Berlin, December 21, 2020 Michell Boerger

Acknowledgments

First of all, I would like to express my sincere thanks to Prof. Dr. Manfred Hauswirth, who made it possible for me to investigate this exciting topic in the context of a master's thesis. Prof. Dr. Manfred Hauswirth has agreed to supervise and evaluate this thesis. Again, I would like to warmly thank Prof. Hauswirth for his helpful recommendations and constructive criticism during the compilation of this thesis. Furthermore, I would like to thank Prof. Dr. rer. nat. Volker Markl for agreeing to assist as co-supervisor of this thesis.

I would also like to thank Dr. Nikolay Tcholtchev and Philipp Lämmel, who supported and supervised my master's thesis within the scope of their activities at the Fraunhofer Institute for Open Communication Systems (Fraunhofer FOKUS). I sincerely thank you for the always inspiring and exceptionally competent support and mentoring. Our many talks and discussions were always helpful and a great support for the successful realization of the thesis. I can't remember any problem for which you didn't have a constructive solution at hand.

Finally, I would like to thank my family and especially my girlfriend Ayanda from the bottom of my heart for her support throughout my studies. You have given me support and inspiration even in difficult times, without which, in retrospect, the completion of my studies seems impossible. I would also like to thank my friends for their mental, psychological, and professional support during this thesis. In particular, I would like to thank Sameer Kumar Subudhi for the many exciting discussions and recommendations during the course of this thesis. Sameer was always ready to lend an ear and always had good ideas, which definitely contributed to the success of this thesis. Thank you buddy!

Last but not least, I would like to mention that the thesis was realized in cooperation with the FOKUS Institute of the Fraunhofer Society in the context of the *WindNODE*[1] project. I would like to thank all participants for the great opportunity and the many helpful discussions. Again, I would like to express my special thanks to Prof. Dr. Manfred Hausfred, Dr. Nikolay Tcholtchev, and Philipp Lämmel. I hope that with the present thesis, I will be able to meet their high scientific standards and contribute to the success of the project.

1 https://www.windnode.de/

Abstract

Climate change and the resulting urgent need for energy transformation are some of the central challenges facing our society in the coming decades. Political decisions such as the plan of the German government to shut down coal-fired power plants by 2038 indicate that in this process, the use of renewable and distributed energy resources will play a central role in ensuring the generation of electricity. Many studies support this assumption and show that the use of these climate-neutral technologies is required to limit climate change. However, the integration of distributed energy resources into the existing energy grid poses enormous challenges for grid operators in terms of grid stability and security of supply. In order to counteract these problems, in the present thesis, we propose a blockchain-based solution which enables a lucrative and grid-serving integration of distributed energy resources as an extension of the existing energy system. This concept can contribute to increased grid stability and security of supply, and could also help to shape the energy system in a more future-proof, reliable, sustainable, scalable, and efficient way.

Specifically, in this thesis, we investigate whether it is feasible to use blockchain technologies to implement innovative digital markets in the energy sector in a secure and scalable way. Precisely, we explore the usage of this disrupting technology to allow trading of flexibilities between the operators of distributed energy resources and the grid operators for grid congestion management. In this context, grid operators integrate the available flexibility as an operating reserve into their grid congestion management processes while simultaneously, the resource operators benefit from receiving monetary compensation. Specifically, we will design short-term energy markets by developing intraday and day-ahead markets where operators of distributed energy resources and grid operators can trade flexibilities on an hourly or daily basis.

From a technological perspective, the main focus of the thesis is the question of whether and how we can design and implement such blockchain-based short-term energy markets by utilizing the concept of general-purpose smart contracts. For this reason, we utilize the Ethereum blockchain for the implementation of such markets and design an appropriate smart-contract-based architecture in this thesis. To ensure the integrity of market actors, we will investigate how to use cryptographic processes and infrastructures in order to integrate external information about actors from a regulating governmental institution into the blockchain-based energy markets. Furthermore, by using this cryptographic infrastructure, we design a pseudonymous usage of the markets, protecting the privacy of all participants and their trading data. Additionally, we present the application of a proprietary cryptocurrency in order to carry out a fully blockchain-based payment process based on available trading and smart meter data. As a proof-of-concept, we present the implementation of a prototype of the blockchain-based market system. Based on this prototype, we will evaluate the proposed market system in terms of security, scalability, and efficiency.

Zusammenfassung

Der Klimawandel und die daraus resultierende dringend notwendige Energiewende gehören zu den zentralen Herausforderungen, denen sich unsere Gesellschaft in den kommenden Jahrzehnten stellen muss. Politische Entscheidungen wie der Plan der deutschen Bundesregierung, Kohlekraftwerke bis zum Jahr 2038 abzuschalten, verdeutlichen, dass in diesem Prozess erneuerbare und dezentrale Energieressourcen eine zentrale Rolle bei der Stromerzeugung spielen werden. Viele Studien unterstützen diese Annahme und zeigen, dass der Einsatz dieser klimaneutralen Technologien notwendig ist, um den Klimawandel zu begrenzen. Jedoch stellt die Integration dieser in das bestehende Energienetz die Netzbetreiber vor enorme Herausforderungen in Bezug auf Netzstabilität und Versorgungssicherheit. Um dem entgegenzuwirken, schlagen wir in der vorliegenden Arbeit eine Blockchain-basierte Lösung vor, die eine lukrative und netzdienliche Integration von verteilten Energieressourcen in das bestehende Energiesystem ermöglicht. Dies kann zu einer erhöhten Netzstabilität und Versorgungssicherheit beitragen und helfen, das Energiesystem zukunftssicherer, zuverlässiger, nachhaltiger, skalierbarer und effizienter zu gestalten.

Konkret untersuchen wir in dieser Arbeit den Einsatz von Blockchain-Technologien zur Umsetzung innovativer digitaler Märkte im Energiesektor. Insbesondere untersuchen wir den Einsatz dieser Technologie zur sicheren und skalierbaren Realisierung eines Flexibilitätenhandels zwischen Betreibern von verteilten Energieressourcen und Netzbetreibern im Sinne des Netzengpassmanagements. Dabei integrieren Netzbetreiber die erworbene Flexibilität als Regelenergie in ihre Prozesse für das Netzengpassmanagement, während gleichzeitig die Ressourcenbetreiber monetäre Vorteile für die Erbringung genießen. Konkret stellen wir kurzfristige Intraday- und Day-Ahead-Märkte vor, in denen ein Handel von Regelenergie auf stündlicher oder täglicher Basis ermöglicht wird.

Aus technologischer Sicht liegt der Schwerpunkt der Arbeit auf der Frage, ob und wie wir solche Energiemärkte mit Hilfe von Smart Contracts gestalten und umsetzen können. Dazu untersuchen wir die Ethereum-Blockchain für die Realisierung der Märkte und entwerfen eine zugehörige Systemarchitektur, welche ausschließlich auf Smart Contracts basiert. Um die Integrität der Marktakteure zu gewährleisten, werden wir außerdem untersuchen, inwiefern kryptographische Verfahren und Infrastrukturen genutzt werden können, um externe Informationen über Akteure in Zusammenarbeit mit einer regulierenden Regierungsinstitution in die Blockchain-basierten Märkte zu integrieren. Darüber hinaus ermöglichen wir durch die Nutzung dieser kryptographischen Infrastruktur eine pseudonymisierte Nutzung der Märkte, wodurch die Privatsphäre aller Teilnehmer und ihrer Handelsdaten geschützt wird. Außerdem stellen wir die Verwendung einer proprietären Kryptowährung vor, mit deren Hilfe ein vollständig Blockchain-basierter Zahlungsprozess auf Grundlage von Handelsdaten und Smart-Meter-Messungen durchgeführt werden kann. Als Proof-of-Concept präsentieren wir außerdem einen Prototypen des Marktsystems. Mit Hilfe dessen werden wir das System abschließend im Hinblick auf Sicherheit, Skalierbarkeit und Effizienz evaluieren.

Contents

1 Introduction

The energy transformation has been one of the central goals and challenges in the energy sector in recent years. The increased use of renewable and distributed energy resources (DERs) confronts energy suppliers and grid operators (GOs) with challenges in terms of grid stability and security of supply [1]. Critics often argue that the excessive use of such technologies puts a strain on the stability of the grid and jeopardizes its operation [1]. On the other hand, studies show that the use of these climate-friendly technologies is imperative to limit climate change [2]. Up to now, GOs primarily use operating reserve from large, centralized, and often fossil energy plants to guarantee the security of supply and maintain grid stability. However, the climate-friendly flexibility potential of renewable and distributed energy resources has not played a central role yet [3]. Nevertheless, since Germany has decided to phase out coal energy by 2038, new approaches need to be developed [4]. To ensure a reliable, future-proof, and efficient energy system, an interaction between traditional and modern players should be considered. It may be advantageous to make the potential of DERs available to the GOs by marketing the available flexibility on new operating reserve markets. It would allow the former to integrate the available flexibility from small-scale conventional and renewable generation plants and shiftable loads into their grid congestion management processes. Thus, by establishing market-based acquisition and trading of decentralized flexibility between GOs and the owners of DERs, the potential of DERs can be fully exploited. In return, the owners of the DERs would receive monetary compensation. Following this approach would reduce the often argued risk of DERs putting a strain on the stability of the grid as they are now even used to maintain grid stability. In the long term, such integration would encourage the use of climate-friendly technologies to reduce carbon dioxide emissions and reassure critics. This concept could also help to shape the energy system in a more future-proof, sustainable, scalable, and efficient way.

In recent years, researchers have become increasingly interested in blockchain, and it is one of the most frequently discussed concepts in the information and communications technology (ICT) world in past years. The term blockchain first appeared in connection with Satoshi Nakamoto's 2008 paper, in which he presented Bitcoin as the first implementation of a blockchain [5]. In his novel and disruptive work, Nakamoto introduced Bitcoin as a digital currency without a central authority that provides means to prevent double-spending attacks [6]. After the great success of Bitcoin, alternative blockchain implementations followed quickly. Examples with a high reputation are the Ethereum blockchain presented by Buterin in 2013 [7] or Hyperledger Fabric [8]. Although these implementations differ from Bitcoin in some details and new features such as the realization of general-purpose smart contracts were added, they still follow the fundamental concept of Bitcoin. From a technical point of view, a blockchain is a practical approach to find a distributed consensus in an underlying untrusted peer-to-peer network with a dynamic number of nodes. In simple terms, it is a chain of blocks that is distributed decentralized across all nodes and in which cryptographic

procedures ensure the immutability of the blocks. These blocks record transactions between participants in which they can exchange digital assets directly without involving any intermediaries or central instance [5]. Typically, blockchain implementations use a consensus mechanism called proof of work to find agreement among all participants on which transactions will be recorded in a block [9]. In this mechanism, participants have to solve a computationally intensive cryptographic puzzle to create blocks. The notable characteristic of this mechanism is that an attacker requires more than fifty percent of the entire computational resources of the underlying network to tamper with the blockchain, rendering it hard to attack and tamperproof [6]. A further interesting aspect of blockchains is the concept of smart contracts that allow recording any formalisms and calculations as transactions, making it possible to implement economic processes, legal contracts, or any business logic using blockchain [7]. One blockchain implementation that implements the concept of general-purpose smart contracts is the Ethereum blockchain introduced by Buterin in the year 2013 [7]. In Chapter 2, we will discuss the concept of a blockchain in more detail.

1.1 Research Question and Contributions

The blockchain characteristics mentioned above predestinate this technology for the implementation of digital markets. Therefore, in the course of the thesis, we will investigate whether it is feasible to use blockchain technologies to implement innovative digital markets in the energy sector in a secure and scalable way. Specially, we intend to investigate the usage of this technology to allow trading of flexibilities between the operators of DERs and the grid operators for grid congestion management. Such markets could help mitigate the problems of DERs concerning grid integration mentioned above and promote the energy transformation. Accordingly, the main focus of the thesis is the question of whether and how we can design and implement secure and scalable energy markets by utilizing smart contracts. For this reason, we aim to employ the Ethereum blockchain for the implementation of such markets and design an appropriate smart-contract-based architecture based on this technology in the thesis.

Since the production and consumption of renewable DERs depend on current weather conditions, flexibility is often only available at short notice. Therefore, we will design short-term energy markets by developing intraday and day-ahead markets where operators of DERs and grid operators can trade flexibilities on an hourly or daily basis. In the intended markets, DER operators will offer the flexibility of their resources on either market. Grid operators can then retrieve, evaluate, and finally purchase these offers. Thereby, we will primarily design the markets for integration into the German electrical energy system, which we only refer to as energy system throughout the thesis.

The energy grid is a critical and highly regulated infrastructure, in which actors and operators often have to be verified and approved by governmental institutions. Therefore, as a further part of the thesis, we will investigate how to integrate external information about actors into the blockchain-based energy markets in a secure and trustworthy manner. In this way, we can ensure the integrity of all market participant operating on this critical infrastructure. Here, external data could be registration and market information of DER operators and grid operators, but also data of the DERs themselves. In the thesis, we assume that an external and official institution exists, which can interact with the blockchain-based markets. Furthermore, we assume an already established public

key infrastructure (PKI) between this institution and the market actors which we can leverage for cryptographic purposes. In the course of this thesis, based on these assumptions, we will investigate the technical realization of the integration of such external information in a cryptographic secure manner.

In general, energy trading data is classified as business-critical data. However, since blockchain data is open, a privacy-aware implementation of the proposed markets could be of particular interest. Thus, we will also elaborate on such a privacy-aware realization in the course of this thesis. Specifically, we will explore whether pseudonymous access to the markets is possible so that neither market participants nor their trading data can be tracked and interpreted by unauthorized parties. For this purpose, we aim to leverage the above mentioned PKI.

Assuming that consumption and production data of DERs is available through smart meters, another research aspect of the thesis is a blockchain-based and semi-automated payment processing between market actors. To this end, we will also investigate the application of a proprietary cryptocurrency in order to carry out such a payment process based on available trading and measurement data. The secure integration of smart meters and the transfer of meter readings to the blockchain is explicitly not part of the thesis. In our work, we assume that such an infrastructure already exists and can be used on the blockchain side.

In conclusion, our expected contributions are the following: the primary objective is to investigate whether we can utilize the blockchain to build short-term energy flexibility markets by leveraging the concept of smart contracts. We intend to take advantage of a dedicated cryptocurrency and smart meter readings to realize a fully blockchain-based payment process between market actors. Furthermore, we intend a collaboration with an external governmental institution to securely integrate external information about participants of the markets with the help of a PKI. In this way, we want to ensure the integrity of all market participant operating on the critical infrastructure of the energy system. Furthermore, we want to use the mentioned PKI to investigate a privacy-aware implementation of the markets, in which all participants can act pseudonymously on the markets, rendering their market behaviour hard to track.

1.2 Outline

The overall structure of the thesis takes the form of 7 chapters, including this introductory chapter. Following this, in Chapter 2, we will present some theoretical fundamentals for the thesis, and we will give a reasonably comprehensive overview of related work. In this context, we will explain the blockchain technology, smart contracts, required cryptographic basics, and the German energy system. Next, we will describe the short-term markets to be implemented in a formal and abstract manner in Chapter 3. Taking this as a basis, in Chapter 4, we will explain our concept for implementing the blockchain-based operating reserve markets by proposing a smart-contract-based architecture. Subsequently, in Chapter 5, we describe the prototypical implementation of the presented system. In Chapter 6, we will then evaluate the developed prototype in terms of efficiency and scalability. Besides, we will also discuss fulfilled requirements and resulting system properties. Finally, we will conclude the thesis and provide an outlook for future work in Chapter 7.

2 Background and State of the Art

Before we cover the concept and implementation of the thesis in more detail, we want to start with presenting technological fundamentals and prerequisites as well as currently relevant research in the following chapter. Since the present thesis has practical relevance, we will present not only academic work but also discuss exciting and relevant industrial projects. In this process, we will try to identify research gaps and compare the research with our work.

2.1 Background

In order to understand the presented thesis, it is essential to comprehend some technological fundamentals. Therefore, in the following, we will start by explaining the concept of a blockchain. We will deal with general principles and present various implementations of it. Subsequently, we will discuss a few required basics of cryptography. Especially the Rivest-Shamir-Adleman (RSA) cryptosystem and the X.509 standard applied in the thesis will be explained. Furthermore, we will also briefly explain the German energy system and discuss what is meant by operating reserve and distributed energy resources.

2.1.1 Blockchain

In 2008, Satoshi Nakamoto presented the Bitcoin network [5] and introduced the concept of a blockchain for the first time. Bitcoin is a digital peer-to-peer currency that runs without a central instance and does not require any trust. In Bitcoin, transactions of digital assets are recorded transparently, chronologically and immutably. Each network participant can access all the transaction data and all transactions are validated and finally accepted by the entire network. Five years later, in 2013, Buterin introduced a general-purpose blockchain called Ethereum [7]. Apart from the transfer of digital assets, any calculation can be represented and recorded as transactions on this blockchain implementation. It is particularly noteworthy that in both blockchains, although not a single node of the network has to be trusted, it is still possible to find a consensus within the network and even node failures and malicious behavior can be tolerated. Due to the great success of these two technologies, the blockchain is on everybody's lips and became the focus of many current research and industrial projects. Many variants of blockchains have been implemented since the publication of Bitcoin. Tschorch et al. [6] provided a comprehensive overview of current blockchain implementations in the area of digital currencies, whereas Cachin et al. present blockchain technologies without the limitation to digital currencies in [10]. Despite all interest and variety in this topic, there is still no officially established definition of the term blockchain.

Figure 2.1 illustrates the schematic and abstract structure of a blockchain. Simply speaking, a blockchain is a cryptographically secure and immutable chain of timestamped blocks. Each of these

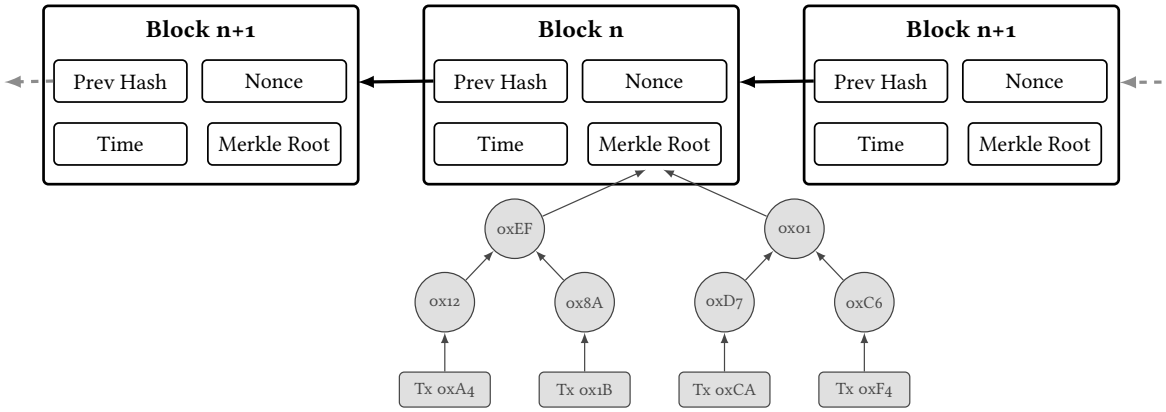


Figure 2.1: Simplified and abstract structure of a blockchain. Blocks are cryptographically securely linked and through a block containing the hash of the block header of the predecessor block. Thus, they form a cryptographically secure and immutable chain of blocks.

blocks contains a list of public and verifiable transactions with tracable origin. These transactions are combined into a block and linked to the list of previous transactions. This is done by including a hash of the header of the previous block in the current block. Via this mechanism, the blocks are cryptographically securely chained together. It should also be noted that the transactions are stored in a Merkle tree [11]. Here, transactions are stored in the leaves of the tree. All other nodes only contain the hash of their child nodes. This procedure is applied up to the root node of the tree, the so-called Merkle root. In this way, the root can be used to uniquely represent a list of transactions stored in the tree. In a blockchain, this property is used to uniquely assign transactions to block. Specifically, the Merkle root of a transaction tree is stored in the block header. Since the hash of the block header is used to link the blocks, the transactions are immutably assigned to a block. If an attacker tries to manipulate a transaction in the tree after it is included in a block, all hashes contained in the Merkle branch of the transaction will change, including the Merkle root. However, since the Merkle root is part of the block header, the corresponding block hash also changes. This would break the linkage and invalidate the chain of blocks. Thus, a blockchain can also be understood as a decentralized register, which chronologically records transactions that are verifiable and immutable.

In a blockchain, users are represented pseudonymously by using cryptographic techniques. For each user, a *wallet* is created, consisting of an asymmetric key pair. A unique identifier to represent the user and their account is calculated based on the public key. This identifier is often referred to as *address* and is used for the user's pseudonymous access to the blockchain network. The private key must be kept confidential and is used to digitally sign a transaction created by the user. This can be clarified by using an example: Alice is a user and wants to transfer a digital asset to Bob. To do so, Alice needs to create a blockchain transaction that represents the transfer of the digital asset. In order to create such a transaction, Alice must first know or receive the pseudonymous address from Bob. Afterwards, Alice creates a transaction locally and sets her address as the sender and that of Bob as the receiver. Furthermore, Alice adds the digital asset to be transferred as transaction data. Afterwards, she signs the transaction with her private key and publishes the transaction to the blockchain network. Here, the content and signature of the transaction are validated and then

recorded in a block. Once this is done, Bob can be considered as the owner of the transferred digital asset.

Consensus

On a blockchain, data is not administered centrally but stored distributed. Each participant of the underlying peer-to-peer network keeps a copy of the current data set and is authorized to read and modify it. Due to the highly distributed nature of the data and the decentralized execution of transactions, a blockchain system is characterized by increased availability, robustness, fault-tolerance and resilience. However, since several participants are processing the data records simultaneously, a consensus on the current distributed status has to be determined within the network. As the blockchain is a distributed peer-to-peer network without any central instance, the consensus problem has to be solved in a distributed manner as well. In the context of blockchain technologies, a number of consensus mechanisms were defined to solve this problem. In the following, we will introduce some of these. For a more extensive and detailed overview, we refer to the work of Cachin et al. [10].

Proof of Work A widely used consensus algorithm that is currently used by both Bitcoin and Ethereum is the so-called *proof of work (PoW)* [5][12]. A PoW algorithm can be interpreted as a probabilistic consensus strategy to determine a leader that is allowed to create new blocks and record transactions in them. Thereby, a computational complex cryptographic puzzle has to be solved to become the current leader. The process of solving the puzzle and creating the blocks is called *mining*, whereas the puzzle solvers are called *miners*. In order to solve the PoW, a hash value below a given *difficulty* threshold has to be found, which varies over time. To find a hash below this threshold is very complex and the miners have to invest a lot of computational resources. However, the verification of a found solution is simple. Specifically, to find a hash that solves the puzzle, miners vary a parameter called *nonce* until the resulting hash is below the difficulty threshold. When a miner solves the PoW, it uses the found nonce to create a new block and sends the newly created block to the entire network. Now each network node verifies the correctness of the PoW. If it is correct, the nodes add the block to their local copy of the blockchain. If two miners find a block simultaneously, a so-called *fork* occurs. In this case, a consensus conflict arises because two valid blocks exist which refer to the same predecessor block. In a PoW this problem is solved with the *longest chain rule*. First, all miners continue mining the next block based on the fork they received first due to different network propagation times. Once new blocks have been mined, the fork with the longest chain is judged as the authentic one since it has cost the most computational power. Subsequently, this fork represents the currently valid blockchain [13].

The high computational effort required to solve the PoW is intended to protect the network against malicious nodes and malicious behavior. Furthermore, the PoW effectively prevents Sybil attacks [14], since the probability of being determined as the next leader does not depend on the number of controlled network nodes but is proportional to the available computational power. Therefore, an attacker must control more than 50% of the total computational power of the network to corrupt the blockchain. This attack vector is known as the *51% attack* [6].

However, the complex PoW consensus mechanism causes several costs for the miners. First, high

energy costs are incurred for the operation of the hardware used for solving the PoW. Furthermore, the acquisition of this hardware is an additional cost factor. But since miners are natural profit seekers, an incentive system is needed to motivate them to still solve the PoW, accepting the high costs, and therefore contributing to the progress of the blockchain. For this reason, a miner receives a mining reward when it solves the PoW. Furthermore, users need to pay transaction fees to the miner for all their transactions which have been mined into their published blocks [6][9].

Proof of Stake An alternative to the PoW is the so-called *proof of stake (PoS)*. The fundamental assumption of this consensus mechanism is that those participants of the network who hold the largest share of the native cryptocurrency are most interested in the correctness of the blockchain. In this context, this share is referred to as *stake*. The leader responsible for creating a new block is also probabilistically chosen in PoS. However, unlike in PoW, in PoS the probability of being determined as the leader does not depend on the available computing power but on the size of the deposited stake. The larger a participant's stake, the higher the probability of being chosen as a leader [9]. Therefore, this mechanism links the block generation to the proof of ownership of a certain amount of digital assets - the stake. If it is determined in the PoS that an elected leader behaves maliciously, they are punished by losing coins from their stake. Through this direct and monetary penalty mechanism, all participants of the consensus mechanism are incentivized to behave correctly. From the described concept, it can also be concluded that an PoS blockchain is resident against attackers unless they hold a monopoly of at least 51% of the native cryptocurrency. In this case, the attackers would most likely be elected as leader more often than the honest nodes, allowing them to corrupt the blockchain [9]. However, since creating such a monopoly is extremely cost-intensive, it is uncertain whether this attack is feasible.

A considerable advantage of PoS in comparison to PoW is that it requires significantly less energy for determining consensus since no computing resources are wasted on finding a nonce. Furthermore, for this reason, the PoS consensus mechanism is often described as more efficient and scalable [13].

As described above, anyone that holds coins of the native cryptocurrency can participate in the PoS consensus mechanism. However, since there is a risk that those participants of the network who hold the largest stake will be chosen as leader with the highest probability, the described naive PoS approach tends to be unfair. Therefore, this aspect must be addressed when implementing a concrete PoS algorithm in order to guarantee fairness to all participants. For example, Peercoin [15] has introduced the concept of *coin age* to counteract this problem. Here, older and larger sets of coins have a greater probability of being chosen as the current leader. Thereby, Peercoin defines the coin age as the native currency times the holding period [6]. Therefore, the probability of being chosen as the leader does not depend entirely on the size of the stakes, which results in a higher probability of participants with lower stakes being chosen as leader.

Another example of a PoS implementation is Casper [16], which is planned to be deployed in Ethereum as a PoS layer on top of the existing PoW. This is supposed to represent the transition to *Ethereum 2.0* [17]. In Casper, dedicated *validator* nodes exist. Network peers can deposit an amount of coins as a stake to become such a validator node. Subsequently, blocks are still created via the already existing PoW algorithm. However, to eliminate forks of the PoW and determine the current valid chain, the validators in Casper vote every 100 blocks on the current valid block. If a block

receives a majority of two-thirds of the validators, it is considered to be the currently valid block and is henceforth immutable. In Casper, such blocks are called *checkpoints*. If a validator is found to act dishonestly, it is penalized by *slashing* its stakes. Due to the voting characteristic, Casper is also considered to be fair.

An extended form of the PoS is the *delegated proof of stake (DPoS)*. Here, stakeholders choose delegates who are responsible for creating and validating blocks. In DPoS, a majority of delegates must vote for the correctness of a block before it is added to the blockchain. If the stakeholders detect malicious behavior of a delegate, it can be voted out and replaced. Therefore, since the size of the stake does not have a direct influence on the consensus mechanism, DPoS is also regarded as fair [9].

Practical Byzantine Fault Tolerance Another alternative consensus mechanism is known as *Practical Byzantine Fault Tolerance (PBFT)* [18]. PBFT can be understood as a replication algorithm to tolerate Byzantine faults. In general, Byzantine faults are understood as any possible type of errors. These can include failures of individual network nodes, malicious behavior, or even consequences of programming errors. In this context, the distinctive characteristic of PBFT is that it is capable of tolerating Byzantine faults of up to one-third of all network nodes [18].

In the PBFT consensus mechanism, a dedicated and pre-known leader exists. This leader acts as a serializer and is intended to broadcast messages or transactions to all existing network nodes. Knowing this, PBFT can be roughly described with the following three phases. The first phase is called *pre-prepare*. In this phase, after a request by a client, the leader sends a pre-prepare message to all nodes informing them that the client wants to perform a certain transaction. The second phase is known as *prepare* phase. After receiving the pre-prepare message, all nodes send a prepare message to all other nodes including the leader. Thereby the prepare message of a node contains the confirmation of the node whether it agrees to the execution of the transaction or not. The third and final phase is then referred to as the *commit* phase. If a node receives confirming prepare messages from more than two-third of the network nodes, it sends a *commit* message to all other nodes. This message is the final notification that the node is now committed to execute the transaction. When a node receives commit messages from at least two-third of the network, it finally executes the transaction and informs the client about the execution. Therefore, the client receives the confirmation of execution from at least two-third of the network nodes. In this context, we want to emphasize that if the leader turns out to be faulty in this scenerio, a protocol exists which can be executed to elect a new leader [18].

A notable characteristic of PBFT is that it does not allow a rollback of the state and thus, once a block is accepted, it is final and cannot be reverted. However, a significant disadvantage of PBFT is that all network nodes must be known a priori. Furthermore, at least two-third of all nodes must be active at all time, since otherwise nodes cannot progress from one phase to the next and therefore cannot execute transactions [19]. As a result, we want to highlight that PBFT is not suitable for public blockchains. However, it can be applied in the context of a permissioned blockchains like Hyperledger Fabric [8]. In the following, we will explain the difference between a permissioned and permissionless blockchain.

Permissioning

Blockchain technologies can generally be classified into two categories, *permissionless* and *permissioned* blockchains. These primarily differ in terms of access, read, and write permissions. In the following, we will briefly introduce these two categories.

Permissionless Blockchain In a permissionless blockchain, access to the underlying peer-to-peer network is public and equal opportunities are provided to all nodes. A permissionless blockchain is often referred to as a *public blockchain* and examples of this category include Bitcoin and Ethereum. In a permissionless blockchain, everyone can access the network, read the blockchain data, and execute transactions in order to change the state of the blockchain. As a result, these can be described as decentralized [19]. Furthermore, every node can participate in determining the consensus on the current valid blockchain state. In a permissionless blockchain, consensus mechanisms such as PoW or PoS are typically employed [9]. Due to the already described characteristics of these consensus protocols, it is nearly infeasible to tamper with a permissionless blockchain. Furthermore, the use of these protocols enables a trustless execution of transactions since no network node needs to be trusted in a permissionless blockchain. However, since the consensus needs to be determined between all nodes, public blockchains suffer from scalability issues [19].

Permissioned Blockchain The second category, called permissioned blockchains, aims to achieve increased scalability and efficiency by restricting network permissions. In a permissioned blockchain, access to the network is restricted and only allowed for authorized participants, resulting in a centralized system [9]. In this context, it is assumed that participants know each other and are interested in the efficiency of the common network [19]. In a permissioned blockchain, it is predefined which group of participants are allowed to read the blockchain data and which group is allowed to modify the blockchain state. To this end, permissioned blockchain systems have the means to identify the nodes that can control and update the shared state via transactions [10]. In a permissioned blockchain, a consensus is determined by only an authorized set of nodes. Therefore, consensus mechanisms such as PoS and PoW are not meaningful to use in the context of permissioned blockchains, but the PBFT protocol can be employed [9]. Furthermore, since transactions are only executed by a subset of nodes that also determine the consensus on the current valid state of the blockchain, permissioned blockchains are considered more scalable [19]. However, a trustless execution of transactions is not enabled in a permissioned blockchain, as these authorized nodes must be trusted. Precisely, malicious behavior of a network node corresponds to a violation of trust and can lead to the network being disabled until the malicious participant is corrected or removed from the network. Finally, we want to mention that permissioned blockchains can be further differentiated between *consortium* and *private* blockchains [9]. The latter are only operated by a single organization within a single trust domain. On the other hand, consortium blockchains can be operated by multiple organizations and can be public or restricted. An example of a consortium blockchain is Hyperledger Fabric [8].

Smart Contracts and Ethereum

Nakamoto's Bitcoin blockchain is primarily intended to use transactions to record the transfer of digital money. While Bitcoin provides a stack-based scripting language that can be used to define conditions for the transfer of digital assets, it is strongly limited and not turing-complete [6]. In contrast, the Ethereum blockchain presented by Buterin in 2013 represents a general-purpose blockchain in which any calculation can be recorded in transactions by utilizing the concept of smart contracts [7]. Nick Szabo first introduced the idea of smart contracts in 1997 [20]. He suggests that these should be used to represent formalized contract rules in hardware or software. In the blockchain context, smart contracts are to be understood as self-executing scripts, which are stored in the blockchain along with a state assigned to the contract. The code stored in a smart contract is implemented in a dedicated scripting language, cannot be changed afterwards and is open to everyone. Within the smart contract, state transitions are defined, which change the assigned state based on its implemented rules. In Ethereum, a smart contract is deployed by executing a blockchain transaction and is internally represented as a contract account [12]. Therefore, a unique address is assigned to it, which enables users to interact with it and to execute the stored code. To do so, a user creates a transaction, defines the smart contract address as the recipient, adds the relevant smart contract function input to the transaction data, and then publishes the transaction to the network. The smart contract is now executed on all nodes of the network and the state transition is processed based on their local copy of the blockchain. Therefore, smart contracts have to be deterministic, because otherwise, each node calculates a different state and no consensus can be reached in the network [21][13]. We provide further information about the smart contract deployment and interaction process in Chapter 5.2.1.

The prerequisite for executing a smart contract is a runtime environment that can interpret and execute the stored code. To this end, Buterin presents the Ethereum Virtual Machine (EVM) for the Ethereum blockchain [7]. In the EVM a low-level, stack-based bytecode language is interpreted which is turing-complete and called *EVM code* [7]. Ethereum smart contracts are typically implemented in Solidity [22], a higher-level programming language developed by the Ethereum core team. Solidity code is compiled to EVM code and added to the smart contract.

As mentioned earlier, transaction fees have to be paid in blockchain systems. In Ethereum all transactions are executed in the EVM and execution costs are expressed in *gas*, which along with a *gas price* per gas unit defined by the transaction issuer can be converted into Ethereum's native currency *Ether (ETH)*. Thereby, a unique gas cost is assigned to each EVM operation, which can be found in the formalization of the EVM presented by Wood in 2014 [12]. In order to avoid infinite loops and denial of service attacks, there also exists an upper limit of maximum transaction costs per transaction, the so-called *gas limit*. Furthermore, there exists a *block gas limit*, which represents the maximum execution cost of all transactions contained in a block. Thereby, the gas limit can be arbitrarily chosen by the transaction issuer but is not allowed to exceed the block gas limit.

Hyperledger Fabric

Another blockchain implementation that supports the concept of general-purpose smart contracts is Hyperledger Fabric [8]. Fabric is founded by the Linux Foundation [23] and designed as a modular

and extensible permissioned blockchain for enterprise organizations to enable them to develop a permissioned blockchain via plug-and-play. In Hyperledger Fabric, three types of nodes exist [24]. The first type of nodes is called *clients*. These propose the execution of transactions in the network. Thereby, each transaction represents the execution of a *chaincode*. In Fabric, chaincodes represent smart contracts and can be developed in standard programming languages like Java[25], Go[26], or JavaScript[27]. The developed chaincode is then executed in isolated Docker[28] environments[24]. In order to execute a transaction in Fabric, a client first sends an execution proposal to the second type of nodes - the *peers*. In this step, only a particular type of peers receive the proposal, the so-called *endorsement peers*. Subsequently, these simulate the execution of the transaction based on their current state of the ledger and send the simulation result as an *endorsement* back to the client. The client then forwards all received endorsements to the last node type - the *orderers*. These now determine a consensus on the actual execution result based on all the submitted endorsements. Thereby, the consensus mechanism used in Fabric is not predefined but can be chosen modularly. After determining a consensus and the chronological order of transactions, the orderers send a batch of several transactions combined as a block to all peers. The peers then validate the transactions contained in the block and add the latters to their local append-only copy of the blockchain [24].

Since in Fabric every transaction is executed only by a subset of authorized endorsement peers, it is considered to be more scalable than Ethereum. Another differentiating factor is that Fabric does not implement a native cryptocurrency [24]. However, a proprietary cryptocurrency can be implemented via chaincodes to represent digital assets as coins.

In addition to Bitcoin, Ethereum, and Hyperledger Fabric, other blockchain implementations exist, such as Corda[29] or Tendermint [30]. However, in order not to exceed the scope of the thesis, we will not discuss more examples and refer to the work of Cachin et al. [10].

2.1.2 Cryptography

In the following section, we will discuss some essential basics in the field of cryptography, which are crucial for the comprehension of this thesis. Since cryptography is only a minor part of the thesis and since we do not want to exceed the scope of the thesis, we refrain from mathematical details as much as possible. For detailed information and mathematical descriptions, we refer the interested reader to the work of Katz et al. [31].

Symmetric and Asymmetric Cryptography

Katz et al. define cryptography as *"the study of mathematical techniques for securing digital information, systems, and distributed computations against adversarial attacks"* [31]. In general, cryptography aims to enable two parties to communicate secretly via an insecure channel in the presence of an eavesdropper with bounded computational resources. However, modern cryptography also addresses mechanisms for ensuring integrity, techniques for exchanging secret keys, protocols for authenticating users, electronic auctions and elections, digital cash, and much more [31]. Thereby, two alternative approaches exists in modern cryptography.

The first and more classical approach is called *symmetric* or *private-key cryptography*. Here, two parties share a single secret key, which is a bit sequence of fixed size. Both parties use this key for

encryption and decryption purposes. The shared key can be used to encrypt the plaintext as well as to decrypt the original plaintext from the encrypted ciphertext. In this context, only the ciphertext is sent over the insecure channel and can therefore be read by the eavesdropper. However, since the eavesdropper does not possess the shared key, they cannot decrypt the message, which allows both partners to communicate secretly. Although symmetric cryptography can be implemented efficiently and securely, it has a few significant disadvantages. First, private-key cryptography does not allow both parties to exchange the shared key initially over the insecure channel. Furthermore, symmetric cryptography requires the management of a large number of keys. To communicate with n partners, $n-1$ keys must be managed in symmetric cryptosystems.

The alternative to symmetric cryptography was introduced in 1976 by Whitfield Diffie and Martin Hellman and is called *asymmetric* or *public-key cryptography* [32]. In contrast to symmetric cryptography, two different keys exist in this approach. A receiver who wishes to communicate secretly generates a pair of keys called public key pk and private key sk. For the key pair holds that one key can invert operations of the respective other key. Therefore the following applies to a message m:

$$pk(sk(m)) = pk(pk^{-1}(m)) = m = sk(sk^{-1}(m)) = sk(pk(m)) \tag{2.1}$$

In an asymmetric cryptosystem, the receiver must keep the private key secret, while the public key can be published and shared with multiple senders. If a specific sender wants to send a secret message to the receiver, it uses pk as encryption key to generate the encrypted ciphertext c from the plaintext message m using $c = pk(m)$. This ciphertext is then sent to the receiver via the insecure channel. Subsequently, the receiver can use its private key sk for decryption by applying $sk(c) = sk(pk(m)) = m$ to recover the plaintext m. In this setting, the secrecy of encrypted messages is preserved even against an adversary who knows the encryption key pk. In contrast to private-key encryption, public-key encryption also allows the distribution of keys over an insecure channel by using *digital signatures* and *digital certificates*, which we will discuss later. Furthermore, in a public-key setting, less keys have to be managed because several communication partners can use the same public key for secure communication with a specific receiver. However, since public-key cryptosystems are two to three magnitudes slower than symmetric cryptosystems, hybrid approaches are frequently used. For example, asymmetric cryptosystems are often used to exchange a secret key of a symmetric cryptosystem. Afterward, this key is used for secure and secret communication [33].

The RSA Cryptosystem

One of the first and still most used public-key cryptosystems was introduced in 1977 by Rivest, Shamir, and Adleman and is named RSA in their honor. RSA is based on the integer factorization problem in which given two large primes, it is easy to compute the product, but it is very difficult to factor the resulting product. Again, we want to emphasize that we refrain from complex mathematical modeling and refer to the work of Katz et al. [31].

Key Generation To generate a key pair in the RSA cryptosystem, two random and large prime numbers p and q are chosen in the first step. Subsequently, their product is calculated as:

$$n = p \cdot q \tag{2.2}$$

Thereby $n \in \mathbb{N}$ represents a k-bit integer number. The prime numbers p and q are typically chosen with a length of $\frac{k}{2}$ bit each. Then a natural number $e \in \mathbb{N}$ with $1 < e < \varphi(n)$ that it is *relatively prime* to $\varphi(n)$ is chosen and for which therefore applies:

$$gcd(e, \varphi(n)) = 1 \tag{2.3}$$

Here $\varphi(n)$ represents the *Euler totient function* and *gcd* the greatest common divisor. Since p and q are prime, $\varphi(n) = (p-1)(q-1)$ applies [34]. Based on this, the number $d \in \mathbb{N}$ with $1 < d < \varphi(n)$ is calculated, which represents the multiplicative inverse to e modulo $\varphi(n)$ and for which the following must apply:

$$d \cdot e \equiv 1 \ mod \ \varphi(n) \tag{2.4}$$

which defines $d \cdot e$ as congruent to 1 $mod \ \ \varphi(n)$ and for which therefore $(1 - d \cdot e) \ mod \ \ \varphi(n) = 0$ holds. The public key pk is now defined by the tuple (n, e). In contrast, the parameter d constitutes the private key sk. Although the parameter n is published, the factors p and q will be effectively hidden due to the enormous difficulty of factoring n. This also hides the way d can be derived from e.

Encryption A k-bit message $m \in \mathbb{Z}_n$ can now be encrypted using the public key (n, e). Thereby $\mathbb{Z}_n$ represents the integer ring over the number set $\{0, 1, ..., n-1\}$ [33]. Specifically, the encrypted ciphertext $c \in \mathbb{Z}_n$ is calculated as follows:

$$c = m^e \ mod \ n \tag{2.5}$$

Decryption The plaintext m can now be retrieved from the ciphertext c using the private key d. The plaintext m is calculated in the following way:

$$m = c^d \ mod \ n \tag{2.6}$$

The correctness of the RSA method can be proved using the theorem of Euler-Fermat, which the interested reader can review in the original RSA paper [34].

The method just described is called *plain RSA*. Nowadays, various extensions exists, which are intended to enhance the security of the RSA cryptosystem, e.g. by applying random *padding* [33]. This and more extensions are defined in the *Public-Key Cryptography Standards (PKCS) #8* [35].

Digital Signatures, Digital Certificates, Public Key Infrastructures, and X.509

The arithmetic of asymmetric cryptography can be used to create *digital signatures*. These can uniquely identify the originator of a message, provide *nonrepudation*, and guarantee the integrity of a message. For digital signatures, the property shown in Equation 2.1 is used, according to which messages encrypted with the private key sk can only be decrypted with the corresponding public key pk. Therefore, a sender could create a simple and illustrative digital signature for the message m using any hash function $h(\cdot)$ and their private key sk_{sender} as follows :

$$sig(m) = sk_{\text{sender}}(h(m)) \tag{2.7}$$

Afterward, the sender sends the tuple $(m, sig(m))$ to the receiver. The receiver can then verify the correctness of the digital signature if they are in possession of the sender's public key pk_{sender}. By applying $pk_{\text{sender}}(sig(m)) = pk_{\text{sender}}(sk_{\text{sender}}(h(m))) = h(m)$ the receiver extracts the signed hash value $h(m)$ of the message. Finally, the sender can calculate the hash of the sent message m independently and verify if both values match. In this case, it is ensured that the sender has indeed created the signature for the sent message m, which guarantees the integrity of the message.

In praxis, the primary application of digital signatures is the secure distribution of public keys via *digital certificates*. These bind a public key to a certain identity or proof that a public key belongs to a specific person. Thereby, a *public key infrastructure* describes how digital certificates and public keys are managed and securely distributed. A PKI assures valid and correct registration, defines how the bootstrapping of certificates works, and how the verification of certificates can be performed [33].

One of the most used PKI concepts is based on the principle of the *chain of trust*. In this context, authorized *certificate authorities (CAs)* exist, which issue digital certificates to identities and persons. Thereby the CAs themselves also have digital certificates, which are called *root certificates* and represent the *trust anchor* in the chain of trust. All digital certificates created by a CAs refer to the root certificate of the respective CA. If a receiver wants to verify the authenticity of a sender's digital certificate, it examines the certificate chain. First, the receiver verifies the signature of the sender certificate and confirms its correctness. Subsequently, the receiver checks whether the certificate was issued by a CA and if it refers to a valid and known root certificate of this CA. If this is the case, the receiver can assume that the identity specified in the certificate is bound to the public key contained in it. According to this principle, the chain of trust can be extended to any length. Specifically, users can issue certificates themselves, which in turn refer to their own certificate issued by a CA. Again, the certificate chain is validated as described above and it is verified that all certificates in the chain are correct and that the last element of the chain refers to a root certificate [33].

One of the most popular and a widely used standards that determine the format of digital certificates is the *X.509* standard. The X.509 standard is created by the ITU-T [36] and also known as ISO/ICE 9594-8 [37]. Today, X.509 certificates are used for the Transport Layer Security (TLS) variants of various transmission protocols, such as the hypertext transfer protocol secure (HTTPS). The most important fields of a digital certificate standardized in X.509 are the following:

- **Version:** Specifies the version of the X.509 standard used. The current version is 3.

- **Subject:** This field specifies the distinguished name of the identity or person for whom the certificate was issued. The notation is based on the X.500 naming convention [38].
- **Issuer:** Distinguished name of the CA or identity that issued the certificate. Again, the notation is based on the X.500 standard.
- **Serial Number:** An integer value that unambiguously describes the originating issuer.
- **Period of Validity:** This field describes in which time period the certificate is valid and can be trusted. The field consists of the first and last date on which the certificate is valid.
- **Public Key:** This field contains the public key, which the certificate binds to the described subject. In addition to the binary string of the public key, the algorithm used (e.g. RSA) and its parameters are specified here.
- **Signature Algorithm:** The algorithm used to sign the certificate and its associated parameters.
- **Extensions:** A collection of one or more optional extension fields.
- **Signature:** The signature over all other fields of the certificate.

2.1.3 German Energy Sector

There are two important aspects of the German energy sector - 1) the energy grid and 2) the energy market. The aim of the former is to transfer energy from the producer to the consumer as efficiently as possible. The latter focuses on trading energy as a commodity and supplies energy to the grid. The operators of the energy grid need to make sure that it is operating continuously and the security of supply is guaranteed [39]. The German electricity grid is hierarchically structured and we differentiate between two types of grids - the transmission grid and the distribution grid [39]. The former is operated by transmission system operators (TSOs) and the latter by the distribution system operators (DSOs). The transmission grid operates on a nationwide level and is responsible to deliver energy from energy production plants to consumption hot-spots like cities. From there on different types of transmission grids are used to distribute energy to industrial facilities and households [39].

Energy market participants supply the grid with energy. For this purpose, they buy and sell energy in various energy markets. In Germany, three major important markets should be highlighted. On the one hand, the long-term trading on the European Energy Exchange (EEX) [40] exists. Here, energy can be traded from one week up to several years before the planned supply, but up to a maximum of six years in advance [41]. It is interesting to know that all energy market participants are expected to balance their consumption and production. For this purpose, balancing groups are assigned to them, which represent virtual energy accounts in which feed-ins and feed-outs are listed and they are expected to balance [3]. Therefore, short-term markets must also exist in order to be able to react at short notice to long-term trades done on the EEX. For this purpose, the day-ahead market and the intraday market exist. Here, energy is also traded on an exchange but in shorter periods. For energy traded on the day-ahead market, the supply will take place the next day [41]. On the intraday market, energy is even traded on an hourly basis [41].

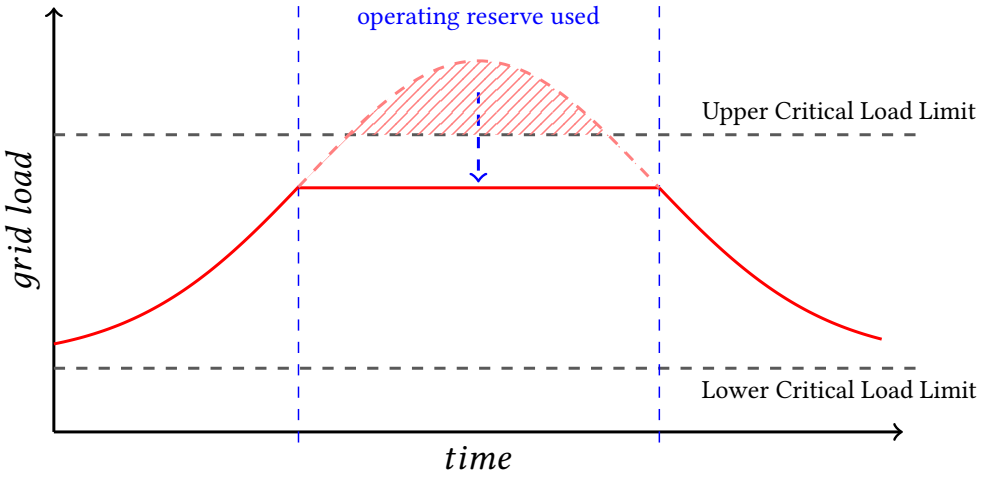


Figure 2.2: An exemplary use of (negative) operating reserve to stabilize the power grid. The red graph describes the current actual load on the grid. The red dotted line shows the estimation of the TSO without the use of operating reserve. Since the estimation would exceed the upper critical load limit, the TSOs use negative operating reserves within the blue interval in order not to endanger the grid stability.

Grid Balancing and Operating Reserve

To ensure the stable and smooth operation of the energy grid, consumption and production must coincide at any time. For this purpose, Germany is divided into four control areas, in each of which one TSO is responsible for maintaining a grid frequency of 50Hz and thus keeping the grid stable. If feed-ins and feed-outs do not coincide within a control area, the grid frequency will deviate from its target value, causing the grid to become unstable, compromising energy supply and risking a grid blackout. If the production is higher than the consumption, the grid frequency increases. The other way around, it drops to a value less than 50Hz. Therefore, the TSOs have to maintain a balance within the control area at any time. As already mentioned, energy market participants have to manage a balancing group for feed-ins and feed-outs. Furthermore, they must report the planned energy schedule of these groups to the TSOs 24 hours in advance [41]. If the sum of all schedules, power flow forecasts, or current measurements of the grid frequency indicates that less energy is consumed than is fed into the grid, the energy production must be reduced or the consumption increased immediately. Vice versa, if the consumption is too high, the feed-in must be increased or the consumption reduced. The TSOs employ to positive and negative operating reserve to accomplish this [3]. Thereby, this energy originates from plants which operators have offered and sold their flexibility in energy production and consumption on operating reserve markets. At the request of the TSOs, they must provide their flexibility promptly [41]. In this context, a positive operating reserve corresponds to an increase in energy production or a reduction in consumption. In contrast, negative operating reserve corresponds either to an increase in consumption or a reduction in production. Figure 2.2 illustrates the exemplary request of negative operating reserve due to an overload of the grid. In the example given, the load forecasts created by the TSO exceed the upper grid load limit in the blue interval. This means that significantly more energy is fed into the grid than is currently

consumed. As a result, the TSO requests negative operating reserves to either increase consumption or reduce production within the geographical control area, thus reducing the grid load and keeping the grid frequency stable.

Distributed Energy Resources

In the present thesis, distributed energy resources are considered as small, local systems providing controllable generation, flexible loads, or energy storage capabilities. They are connected to the distribution grid and the generated energy is primarily consumed locally. Typical examples of DERs are solar plants, wind power plants, combined heat and power plants but also electric cars, heat pumps, and other energy storage systems. Other studies have already shown that the flexibilities in production and consumption of energy given by DERs can be used for grid control while providing climate-friendly advantages [42]–[44]. The range of flexibility available to DERs ranges from a few kilowatts to several megawatts and is therefore particularly suitable for use as a tertiary operating reserve, which must be provided 15 minutes after requested by the TSOs [3]. It should be emphasized that in contrast to large power plants, in the context of DERs flexibility can only be planned and provided at short notice.

2.2 State of the Art

In recent years, several studies and projects have been conducted in which blockchain technologies have been used in the energy domain. The studies of Chitchyan et al. [45] and Goranović et al. [46] provide a comprehensive overview of current expectations and efforts to establish these technologies in this domain. These works clearly show that much of the research is primarily focused on peer-to-peer electricity trading while research on grid services has received less attention so far.

2.2.1 Academic Approaches

One of the first studies that utilize blockchain technologies to realize flexibility services was conducted by Horta et al. in [47]. In their work, they propose a blockchain-based transactive platform and market mechanism, in which households can exchange energy blocks and flexibility services with neighbors and grid operators on local energy markets with the aim to locally balance renewable energy while satisfying economic goals. Their presented system is designed for the future low voltage distribution grid. In contrast to our work, integration of already deployed distributed energy resources into the existing infrastructure is not promoted.

In 2018, Pop et al. [48] investigated how the Ethereum blockchain can be used to use energy flexibilities by utilizing a demand response program. In detail, grid operators request energy flexibilities from prosumers by sending them energy demand requests. Subsequently, they respond with the amount of energy they are willing to shift and an expected price. Based on this information, the grid operator performs analyses and selects some prosumers for the flexibility service, which will then be accepted by a final signal. Finally, the prosumers will apply the requested load shift. In contrast to the present thesis, Pop et al. expect the grid operators to request energy flexibilities pro-actively and

on-demand. On the one hand, this makes it more difficult the resource operators to plan the provision of operating reserve. On the other hand, real-time price negotiations could be unfair. Additionally, Pop et al. focus on energy flexibilities from prosumer households. The flexible loads from industrial facilities are disregarded here.

Similar to the work of Pop, Danzi et al. introduce a demand response program in [49]. This program aims to incentivize users that are assigned to balancing groups to utilize their available flexibilities to avoid imbalance within these groups in order to reduce the costs for balancing these groups. In contrast to our work, the only enable provision of operating reserve at very short notice.

Very related to this thesis, Blom et al. [50] present a day-ahead and real-time market for trading energy blocks and flexibility services based on the Ethereum blockchain. In contrast to this thesis, not only flexibility but also electricity are traded in their solution. First, electricity is traded on local energy markets and it is attempted to cover supply and demand. If this is not possible, flexibility is traded in order to fill the gap. Unfortunately, the implementation and design of markets is not described very extensively in their published paper.

In [51], Nieße et al. propose a fully automated grid congestion management approach. Here, a combination of blockchain technologies and autonomous and automated agents are used to utilize local energy as an operating reserve. To this end, distribution system operators negotiate available flexibilities and expected prices with the agents off-chain. Afterwards, only the settled price is documented on-chain. Subsequently, the flexibility is scheduled, and a billing process is initiated In contrast to the present thesis, only day-ahead trading is possible. Therefore, there is no option to integrate flexibility available at very short notice.

An alternative and often discussed aspect regarding grid stability is the concept of transactive energy. Here, information about energy production and consumption is exchanged between all participants of a smart or microgrid. In contrast to the direct employment of operating reserve, energy is traded and exchanged between all participants to balance the grid autonomously based on the transferred information. This results in increased grid resilience, which should eliminate the need for an operating reserve. There already exist first academic studies describing this concept in connection with the blockchain technology. In particular, we would like to mention the work of Lombardi et al. [52], Mylrea et al. [53], and Nakayma et al. [54].

2.2.2 Industrial Projects

Also, some industrial projects are related to the concept discussed in this thesis. The transmission system operator TenneT TSO GmbH is collaborating with the German company Sonnen GmbH to investigate within a pilot project to what extent battery storage systems in combination with blockchain technologies can contribute to grid stabilization [55]. Additionally, TenneT is carrying out a similar pilot project with the Dutch company Vandebron to investigate how the blockchain and the flexibility potential of electric vehicles can be used to balance the energy grid [55]. Last but not least the pilot project Gridchain of the company PONTON GmbH has to be mentioned. In cooperation with a group of Austrian distribution system operators, they explore the possibilities of leveraging blockchain technologies for grid process integration [56].

3 Market Definition

In the following, we will present a high-level description of the short-term markets in which operators of distributed energy resources can offer their resources' flexibilities to grid operators for the use as an operating reserve. In this description, we do not yet require any specific technologies. Instead, the described markets can be realized using any blockchain that implements the concept of general-purpose smart contracts and proprietary cryptocurrencies. We start by describing the actors in the markets and defining their registration process. In doing so, we assume a superordinate governmental institution that regulates the market participants to guarantee the integrity of all market actors. We will then describe how we consider using cryptocurrencies to implement an automated payment process based on trading data and smart meter readings on the markets. Finally, we will explain the semantics and characteristics of the two presented market variations intraday and day-ahead.

3.1 Roles and Actors

In the proposed operating reserve markets, two different types of user roles exist, *grid operators* and *resource operators (ROs)*. The latter are the operators of distributed energy resources (DERs), which offer flexibility in the energy production and consumption of their resources for trading. In the proposed markets, they represent the supply side. Opposite to this, the GOs constitute the demand side and represent the TSOs and DSOs. In the markets designed, they purchase the flexibility offered in order to utilize it for grid stabilization by using it as an operating reserve.

In both proposed market variants, ROs can only offer flexibilities and access their placed offers, but not acquire third-party flexibilities. On the other hand, GOs can retrieve all offers and can accept and purchase them. However, they cannot sell the products acquired as flexibility to third parties. We want to mention that we will use the formulation *accept an offer* and *purchase an offer* synonymously throughout the thesis.

In addition to these two market participants, a single superior authority, which we call the *data and grid authority (DGA)*, exists. This identity is unique in the system and represented by an official governmental institution. It cooperates with the blockchain-based markets and validates registration information of market participants and DERs using external proprietary knowledge. Furthermore, this actor is entitled to resolve conflicts in the blockchain-based payment process. These can occur, for example, in the event of incorrect or illegal market behavior by users or the lack of smart meter measurement data.

3.1.1 Registration

Before we start explaining the two market variants and the blockchain-based payment process, we want to explain how users can register to participate in the proposed markets. In this regard, we

expect that the just mentioned DGA is willing to cooperate with the proposed market system. As already mentioned, we assume that this governmental institution regulates participants and resources in the real-world energy system and can validate information about them. In the following, we will briefly explain how the proposed system can leverage these competencies by collaborating with the DGA to guarantee the integrity of the participants of the blockchain-based markets. In doing so, we will also explain what technical requirements we demand of the DGA in order to achieve this.

The assumption of a regulatory institution in the energy system

The energy system is a critical element in any country and requires special regulation and integrity. In our use case, we, therefore, assume that the DGA as a central governmental institution regulates which actors are allowed to participate in this system. This is intended to guarantee the integrity and security of the energy system. Precisely, the DGA assigns roles to persons and determines which plants and DERs to integrate into the grid. It has access to proprietary data on actors, roles, and DERs in the already established energy system and is therefore capable of validating information about them. We assume that users must first register to the DGA in the real world in order to participate in the established energy system. The DGA then verifies them and decides whether to integrate the respective person or identity into the energy system as the role applied for. Using this approach, the integrity and security of the energy system can be ensured. In this context, we expect that the DGA assigns a unique identifier to each actor and DER during the registration process. They later use this identification number for all energy system related communication and identification towards other parties. Furthermore, to ensure the integrity of communication between parties in the energy system, we also assume that the DGA deploys a PKI where it acts as a CA by issuing digital certificates. In this regard, each actor creates a cryptographic key pair consisting of private and public keys during registration. The DGA then certifies the ownership of the created public key by issuing a digital certificate to the user. Subsequently, this certificate allows users to authenticate themselves to the institution and third parties in a cryptographically secure manner. In the thesis, we assume that the PKI is based on the X.509 standard and that the RSA cryptosystem is used to create the key pairs. Finally, we assume that in addition to the digital certificate and the identifier, the institution also provides a registration code to each actor. As we will explain later in this section, the users use this code as an additional pseudo-random component in our designed registration process.

The described real-world registration procedure is independent of our blockchain-based approach and is instead the fundamental prerequisite for its implementation described in the following. We explicitly want to emphasize that it will not be part of the thesis to implement such an institution and real-world registration process. Instead, we assume the responsibilities just discussed and aim to ensure the integrity of our market participants and DERs with the help of an already established PKI.

Blockchain-based registration of users and DERs

As already mentioned, we perceive the necessity of a collaboration with the regulating DGA for the proposed operating reserve markets. This is motivated by the fact that the energy network is a critical infrastructure. In particular, the application of operating reserve for grid stability is an essential and critical service for the successful operation of the grid, in which it is crucial to guarantee the

integrity of all involved actors. Therefore, in order to participate in the proposed blockchain-based markets, users must first register in our designed system, which will then collaborate with the DGA to verify the users' integrity. In order to achieve this, we require some fundamental prerequisites before users can perform the blockchain-based registration. Precisely, we assume that the GOs and resource operators (ROs) have already successfully registered at the DGA in the real world and were assigned an identifier, an X.509 based digital certificate, and a registration code. Furthermore, we demand that ROs have already had their DERs validated and registered at the DGA. As a result, we can now perform blockchain-based registration of users and DERs with guaranteed integrity. Additionally, we can now define the registration processes in a privacy-aware manner. This means that no user has to reveal information about themselves or their DERs in the openly accessible blockchain. Specifically, we will calculate a unique pseudonym for each user and DER, which can be used within the blockchain-based markets to represent these identities. The resulting pseudonymous usage of the markets prevents real-world users' market and trading behavior from being traced while their privacy is protected.

User registration When a user wants to participate in the blockchain-based markets, they can do so by invoking a smart contract function, which triggers a blockchain transaction using their blockchain account. In this function, the user specifies the market role they wish to perform (either GO or RO) and a calculated pseudonymous hash identifier based on their identifier provided by the DGA. Furthermore, they must provide a specific digital signature calculated using their private key. Therefore, the smart contract function input triple reg_{user} is defined as:

$$reg_{\text{user}} := (role,\ h(ID_{\text{user}},\ n),\ \underbrace{pk_{\text{DGA}}(ID_{\text{user}},\ sk_{\text{user}}(ID_{\text{user}},\ n)))}_{=:\ u} \tag{3.1}$$

where $pk_{\text{actor}}(\cdot)$ defines the ciphertext encrypted with public key pk_{actor} of the respective actor. Accordingly, $sk_{\text{actor}}(\cdot)$ defines the ciphertext encrypted using the actor's corresponding secret key sk_{actor}.[1] The parameter n is the user's registration code provided by the DGA and can be defined as

$$n := sk_{\text{DGA}}(h(ID_{user})) \tag{3.2}$$

In both equations, $h(\cdot)$ represents the SHA-256 hash function [57]. The parameter ID_{user} represents the user's unique real world identifier, which was provided by the DGA. By assigning the data from reg_{user} to the user's blockchain account, it is now stored on the blockchain that the account that issued the transaction wants to participate in the markets by performing the provided role $role$. However, the user behind the account is not yet successfully registered and must wait for approval from the DGA which guarantees the integrity of the user data. For this purpose, the parameters u and $h(ID_{\text{user}},\ n)$ (see eq. 3.1) are used, which are also stored on the blockchain and mapped to the user's blockchain account address. They are serving two purposes.

1 These keys are based on the PKI established by the DGA. In the described use case we will assume RSA keys. We also want to recall that the used placeholder *actor* can represent markets users like the ROs or GOs but also the DGA.

First, the parameter $h(ID_{\text{user}},\ n)$ is used as a pseudonymous identifier of the user and prevents the user from revealing their real-world identity on the blockchain, which protects their privacy. Only the user can create this parameter, as only they know the assigned registration code n. The uniqueness characteristic of this parameter has the additional advantage that the system can guarantee that no user can register twice. No other blockchain account can register using the same $h(ID_{\text{user}},\ n)$. If the system detects a duplicated usage of this parameter, the corresponding registration transactions would be reverted.[2] Thereby, we want to highlight that the parameter is only used during the registration process to identify a user in order to prevent double registration. For all subsequent market actions, the system uses the pseudonymous blockchain account address to identify registered users. In the definition of the parameter $h(ID_{\text{user}},\ n)$, we introduced the pseudo-random registration code n. The reason for doing so is to prevent attackers who either already know the user ID or are using brute-force to reveal the user's identity and thus link the blockchain account to real-world users, which would violate their privacy. Therefore, the users have to ensure that the parameter n gets not leaked to third parties. As the definition of the registration code (see eq. 3.2) implies, there is no additional effort for the DGA to provide the registration code. The DGA does not have to store the registration code because it can verify it at any time using its public key and the parameter ID_{user}.

Second, the parameter u (see eq. 3.1) is used to prove the users integrity to the DGA. When a user has performed a registration transaction, the smart contract emits an event to notify the DGA that a user needs to be verified. Subsequently, the DGA reads the parameters u, $h(ID_{\text{user}},\ n)$, and the stored role from the blockchain to prove the integrity of the data in a three step verification process. In the first step, it uses its private key sk_{DGA} and the parameter u to identify the real-world user. Specifically, the DGA can extract the identifier ID_{user} and the user's signature $sk_{\text{user}}(ID_{\text{user}},\ n)$ by applying the arithmetic of asymmetric cryptography to u:

$$sk_{\text{DGA}}(u) = sk_{\text{DGA}}(pk_{\text{DGA}}(ID_{\text{user}},\ sk_{\text{user}}(ID_{\text{user}},\ n))) = (ID_{\text{user}},\ sk_{\text{user}}(ID_{\text{user}},\ n)) \tag{3.3}$$

Then it checks the validity of the provided signature by using the public key pk_{user} of the user known to it and extracts the signature input data by applying:

$$pk_{\text{user}}(sk_{\text{user}}(ID_{\text{user}},\ n)) = (ID_{\text{user}},\ n) \tag{3.4}$$

The DGA now verifies that the extracted ID_{user} and the registration code n are valid. For this purpose, it uses its public key pk_{DGA} to decrypt the registration code n in order to extract the encrypted value $h(ID_{\text{user}})$ (see equation 3.2). Then, it uses the parameter ID_{user} just extracted from $sk_{\text{user}}(ID_{\text{user}},\ n)$ and applies the SHA-256 hash to it. Finally, it checks whether the calculated and extracted hash matches to prove the validity of the registration code. Then, it also verifies that the inner and outer ID_{user} parameters extracted from u are identical. If these parameters also match, the DGA assumes that the signature and parameter u are correct and were created by the user ID_{user}. As a next step, the DGA verifies the parameter $h(ID_{\text{user}},\ n)$ read from the blockchain. Therefore, it simply recalculates

2 Although only the users themselves can create the parameter u, we cannot use it as an identifier. Due to the randomized padding of RSA encryption, this parameter is not unique. Thus, a double registration could not be prevented.

the parameter based on the parameters ID_{user} and n just extracted from the already verified parameter u. Then, it validates whether the calculated hash value and the value read from the blockchain match. If this is the case, it is guaranteed that the user also used the correct pseudonym identifier.[3] In the third and last step, after the signature and pseudonymous identifier are verified, the DGA finally checks whether the user is allowed to operate as the claimed role in the energy system using its proprietary data set. If all three verification steps were successful, it issues a blockchain transaction to unlock the user, which grants the user access to the blockchain-based markets. On the contrary case, it blocks the user in the markets with a different kind of transaction. A visualization of the user registration process containing some implementation details is depicted in Figure 4.3.

In the described user registration process, the DGA is the only instance that can decrypt and interpret the parameter u and identify real-world users behind the pseudonymous identifiers $h(ID_{\text{user}},\ n)$. Therefore, it is also the only instance that can link blockchain accounts to real-world users.[4] Through this mechanism, we can guarantee the real-world user's integrity behind the blockchain account used for registration while protecting the user's privacy.

DER registration In a similar way, we can define the registration process for DERs. If an RO wants to trade flexibilities of one of its DERs on the markets, they must first register them on those. We impose the apparent condition that they have previously registered as RO and were successfully verified by the DGA based on the process described above. To guarantee the integrity of the markets here as well, we demand that the DGA must also verify DERs before the RO can offer flexibility for them on the blockchain-based markets. This verification is needed to confirm that the RO is the actual operator and the DER really exists and can provide operating reserve. To register a DER on the markets, the RO initiates a blockchain transaction by invoking a smart contract function with the following input:

$$reg_{\text{DER}} := (\text{type},\ h(ID_{\text{DER}},\ n),\ h(ID_{\text{smartMeter}}),\ r) \tag{3.5}$$

where we define the encrypted ciphertext r as:

$$r\ :=\ pk_{\text{DGA}}(ID_{\text{user}},\ ID_{\text{DER}},\ ID_{\text{smartMeter}},\ sk_{\text{user}}(ID_{\text{user}},\ n)) \tag{3.6}$$

The tuple reg_{DER} is stored on the blockchain and assigned to the blockchain account address of the transaction inititating user. The input parameters consist of the *type* of the DER and the SHA-256 hash of the smart meter identifier $ID_{\text{smartMeter}}$ assigned to the DER's, which we notate as $h(ID_{\text{smartMeter}})$ and which will we use later to fetch production and consumption readings related to this DER (see Chapter 3.2). In addition, the RO must also provide the input parameters $h(ID_{\text{DER}},\ n)$ and r, which

3 The DGA does not check whether the hash identifier is already used on the blockchain, because this is already done by the smart contracts during execution of the registration transaction to prevent double registration. In this case, the transaction would have been reverted.

4 As mentioned above, the parameters u and $h(ID_{\text{user}},\ n)$ are only used during the registration process to ensure the integrity of a user and to prevent a user from registering multiple times. For all other market actions, the blockchain account of an already verified user is used for identification (which is still linked to u).

are again used to verify the integrity of the DER data. Similar to the user registration process, they serve two purposes.

The parameter $h(ID_{\text{DER}},\ n)$ is used as a pseudonymous identifier of the DER. For all market activities, users must specify this parameter to identify DERs uniquely. Using this parameter prevents the RO from revealing the real-world identifier of their DER to the blockchain, protecting the privacy of their trading and production data. Additionally, since $h(ID_{\text{DER}},\ n)$ is also unique for each DER and operating RO pair, we can use it to guarantee that no RO can register DERs twice by reverting registration transactions when duplicated $h(ID_{\text{DER}},\ n)$ parameters are detected.

The ciphertext r is used to guarantee the integrity of the DER and is understood as a signature. The parameter can only be decrypted and interpreted by the DGA, which makes its involvement necessary. In this parameter, the RO encrypts information about the DER they want to register using the public key pk_{DGA} of the DGA. Mainly, the RO uses the identifier ID_{DER}, which was assigned to the resource by the DGA during the registration in the real world. Furthermore, they provide the identifier $ID_{\text{smartMeter}}$. Additionally, the RO includes their identifier ID_{user} and their corresponding signature to prove and guarantee their integrity. Similar to the user registration process, the DGA receives an event when a DER was registered. This notification serves as a request to review the DER data and verify its integrity. Therefore, the DGA retrieves the parameters contained in reg_{DER} from the blockchain and verifies them in a four-step verification process.

In the first step, the DGA uses its private key sk_{DGA} to decrypt r and to extract the contained encrypted parameters $(ID_{\text{user}},\ ID_{\text{DER}},\ ID_{\text{smartMeter}},\ sk_{\text{user}}(ID_{\text{user}},\ n))$ by applying:

$$sk_{\text{DGA}}(r) = (ID_{\text{user}},\ ID_{\text{DER}},\ ID_{\text{smartMeter}},\ sk_{\text{user}}(ID_{\text{user}},\ n)) \tag{3.7}$$

Then, it checks the validity of the provided signature $sk_{\text{user}}(ID_{\text{user}},\ n)$ by using the public key pk_{user} exactly as described in the user registration process. Here, the *RO* used the same registration code n (see eq. 3.2) as in the user registration. Furthermore, the DGA also checks whether the found user ID_{user} is a registered RO and assigned to the specified resource ID_{DER} based on information from its proprietary database. If all requirements are fulfilled, the DGA assumes that the parameter r is correct. In the next verification step, the integrity of the pseudonym identifier $h(ID_{\text{DER}},\ n)$ gets verified. For this purpose, the DGA recalculates the parameter based on the parameters ID_{DER} and n just extracted from the verified parameter r. Subsequently, it validates whether the calculated hash value and the value read from the blockchain match. If this is the case, the system can guarantee that the RO also used the correct pseudonym identifier for their DER. As the third step, the DGA checks whether the smart meter $ID_{\text{smartMeter}}$ is allocated to this DER ID_{DER} using its proprietary database. It also checks whether the RO provided the correct smart meter hash by just recalculating it and comparing it with the value read from blockchain. In the last step, the DGA must ensure that a user with multiple identities registers the DER for the correct blockchain account.[5] To do this, the DGA

5 This case can occur when a user multiple roles in the real-world energy system. For example, grid operators could also operate DERs and therefore be ROs. In this case, they would have to register with two distinguished blockchain accounts for each role on the blockchain-based markets. Consequently, they know the identifiers, attachments and keys of both accounts. This knowledge would enable a RO to register its DERs not with the associated RO blockchain account, but with the GO account, which is prohibited.

uses the parameters ID_{user} and n extracted from r to calculate the pseudonymous hash identifier $h(ID_{\text{user}},\ n)$ of the RO (see eq. 3.1). Then it reads from the blockchain the account address of the user with this calculated pseudonymous identifier. Next, the DGA checks whether this address matches the address that issued the transaction for registering the DER. This procedure ensures that the user is using their correct blockchain identity. If all these four verification steps succeeded and all requirements are met, the DGA approves the DER by issuing a blockchain transaction that unlocks the DGA. Henceforth, the RO can offer flexibilities for it on the markets now. In case that at least one of the requirements is not met, the disapproval of the registration request is submitted using a different blockchain transaction.

In the two registration processes presented, we can guarantee the integrity of users and DERs in the markets. At the same time, we have enabled pseudonymous access to the markets. This protects the privacy of the users and their trading data and rendering them anonymous. Since only the DGA can link blockchain accounts to real-world users and DERs, the market and trading behavior cannot be traced. This enables the implementation of free, protected, non-discriminatory, and privacy-aware markets while guaranteeing the integrity and privacy of all participants. Once again, we would like to clearly emphasize that it is not part of the thesis to implement the described off-chain behavior and the required data infrastructure of the DGA.[6]

3.2 Payment Processing and Fees

Before we finally explain the semantics and characteristics of the two market variants intraday and day-ahead, we will explain how we define the payment process between market participants. In the proposed markets, we implement an semi-automated blockchain-based payment process based on trading data and smart meter readings. For this purpose, we exploit the concept of cryptocurrencies and introduce a proprietary coin to implement such billing procedures. However, before we talk about the actual payment process, we would like to briefly explain the necessity of smart meter readings for such semi-automated mechanisms.

The need for smart meter readings

In the proposed markets, ROs offer only hypothetical forecasts of the flexibilities available for the DERs. GOs then purchase these speculative products before the ROs generate and provide the sold energy. This hypothetical trading data alone is, therefore, not sufficient for an automated payment process. Instead, we need to know whether the services sold were delivered and the energy generated. For this purpose, we would like to propose the use of smart meters. As already indicated in Chapter 3.1.1, we assume that each DER is assigned to a smart meter through which consumption and production data is made available on the blockchain. That is why a smart meter with the blockchain identifier $h(ID_{\text{smartMeter}})$ is assigned to each DER during the blockchain-based registration process of the latter. The DGA knows the real-world installations and has already

6 Rather, we take this for granted and assume that the DGA is willing to cooperate with the blockchain-based markets in order to ensure the integrity of the actors and, thus, the markets. For this purpose, we only provide a blockchain interface that enables the DGA to accomplish its duties

approved this assignment during registration (see Chapter 3.1.1). Therefore, only the availability of and access to the measurement data on the blockchain is missing to perform the semi-automated payment process. However, the secure integration of smart meters and such measurement data is explicitly not part of this thesis. Instead, we assume an already existing infrastructure, through which we can access readings of the smart meters via smart contracts using the identifier $h(ID_{\text{smartMeter}})$.[7] Thereby, the measured data is available in 15-minute intervals, in which each interval contains the absolute measured consumed and produced energy values of the smart meter as well as the relative values representing just the increase or decrease of this specific interval. Therefore, we can define a single smart meter reading $m_{t,h(ID_{\text{smartMeter}})}$ for the interval t of the smart meter $h(ID_{\text{smartMeter}})$ as:

$$m_{t,h(ID_{\text{smartMeter}})} := (e^{+}_{\text{absolute}},\ e^{+}_{\text{relative}}, e^{-}_{\text{absolute}},\ e^{-}_{\text{relative}}) \tag{3.8}$$

Here, e^{+}_{absolute} and e^{+}_{relative} correspond to the absolute and relative measured energy production values which were supplied to the grid. On the contrary, e^{-}_{absolute} and e^{-}_{relative} express the energy consumed from the grid during the given time interval. All four values are given in *kilo watt hours (kWh)*. In contrast, t is understood as a Unix timestamp and is a divisor of 15 min multipliers.

Furthermore, we assume a *smart meter manager (SMM)*, which is responsible for managing and installing smart meters in the real world. Additionally, it carries out the correct transmission of metering data to the blockchain and an accurate assignment to smart meters. We assume that the SMM collaborates with the DGA to ensure the correct assignment of smart meters to DERs. In this thesis, we will only prototypically implement a metering smart contract, which provides measurement data and over which we can read data of individual smart meters identified by $h(ID_{\text{smartMeter}})$. Validation, integration, and implementation of the SMM, as well as the secure and reliable transmission of smart meter readings, is out-of-scope of this thesis.

The payment process

In the thesis, we leverage the concept of cryptocurrencies to implement a semi-automated and blockchain-based payment process. Although some blockchain technologies like Ethereum offer a native cryptocurrency that could be used for payment purposes, others like Hyperledger Fabric do not (see Chapter 2.1.1). Therefore, in order to be technology agnostic, we introduce a proprietary cryptocurrency called *Smart Energy Coin (SEC)*. We designate it as a system-wide payment method for all trading volumes and fees, which we charge for dedicated market operations. The market participants will have to acquire the currency in advance and independently in order to be able to execute trades on the markets. Thereby, the users themselves have to ensure that their used blockchain account has enough balance of the SEC currency. We describe the payment process as semi-automated since the system does not execute it fully automated. Instead, the process must be initiated via a corresponding blockchain transaction by any involved trading partner.

We would like to recall once again that the flexibilities traded on the markets are purely hypothetical and must be validated by measurement data after the virtual purchase of the service. Therefore, an instant payment mechanism on the markets is not appropriate. The service providers would have

7 Since this parameter is assigned to DERs, we can map production and consumption data to these resources.

already received the money for their service to provide and would no longer be forced to serve it. On the other hand, a pure delayed payment process, in which the purchasers pay immediately after they have received the service is also unsuitable since the service providers have no guarantees of monetary compensation for the already provided service. This dilemma illustrates the necessity of a combination of both approaches. For these reasons, we introduce an escrow mechanism to perform the blockchain-based payment processes. In this mechanism, users must immediately transfer the value of the purchased and traded services as a deposit to a dedicated escrow account, which is represented by a smart contract. If the delivery of the service is proofed at a later date, the deposit is transferred to the service provider. Otherwise, the amount is refunded to the purchasers. In our use case, the ROs are the service providers that offer flexibility as a service to the GOs. Therefore, based on the escrow mechanism, they are paid delayed after a proof of delivery of their services has been performed. However, before this is possible, the GOs must first provide the corresponding value of their purchased flexibility to the escrow account during a purchase. For this purpose, the system will automatically transfer this value from the blockchain account of the GOs to the escrow account, provided the account has a sufficient balance of SECs. Otherwise, the system will reject the purchase. After a predefined time limit, any involved trading partner can initiate the automatic payment process. This time limit is supposed to ensure that the SMM successfully transferred all smart meter readings to the blockchain. These are an essential element of the escrow mechanism since they are used to prove the delivery of the service. To this end, the system will calculate how much flexibility was delivered by the RO to the GO based on the readings and the stored trading data. If the RO provided all offered flexibilities, the escrow account automatically transfers the entire deposited amount to the RO as payment for the service. Otherwise, this amount is only paid proportionally to the delivered energy. The escrow mechanism refunds the remaining deposited amount to the GO. As we will explain in Chapter 3.3, multiple GOs can just acquire individual shares of the flexibility proposed in an offer. Now the exceptional case may arise where the RO has not delivered enough flexibility to satisfy the need of all GOs. If the system determines such a case based on the meter readings, we assign the supplied energy to the GOs according to the *first come, first serve (FCFS)* principle. In this approach, the partially delivered energy is first assigned to the first purchaser until their demand is satisfied. Then the RO is paid according to the mechanism described above. Subsequently, the system assigns the remaining unassigned energy to the second purchaser, and so on. The system will refund all services not delivered to the unsatisfied GOs. In this process, the purchasers are ordered based on the block numbers of the blocks that contain their blockchain transaction representing the purchase of an offer.

To allow the formal definition of the escrow mechanism at a later point in this chapter, we would like to introduce some useful functions related to the payment process here:

- $address(user)$: Returns the blockchain account address of the market actor $user$
- $balance(addr)$: Calculates the number of SECs the blockchain account with the address $addr$ holds.
- $transfer(sender, receiver, n)$: Transfers n SECs from blockchain account with address $sender$ to the given account $receiver$.

Fees and Rewards

In the markets presented, we require the actors to pay fees for all performed market operations. There are several reasons for this provision. Firstly, we intend to use the deposited fees to compensate for the effort of third parties potentially involved in an operation. For example, the DGA is compensated with the fees paid for the approval of market participants during registration. Another reason for using fees is due to the already mentioned fact that trading processes presented here are not atomic but depend on delayed actions. For instance, although the markets do record which RO is supposed to provide how much flexibilities to which GOs at a later time, the delivery of the offered service is independent of the trading process. Therefore, we aim to use the required fees to encourage users to behave according to the terms stipulated. Concretely, we charge fees in the form of SECs for the registration of users and DERs. Beyond, we will charge fees for placing and accepting offers. We define the function $fees(\cdot)$ that calculates the fees for a given market operation as follows:

$$fees(operation) = \begin{cases} 10\ \text{SEC} =: f_{\text{userRegistration}} & , \text{ if } operation = \text{“userRegistration”} \\ 10\ \text{SEC} =: f_{\text{derRegistration}} & , \text{ if } operation = \text{“derRegistration”} \\ 5\ \text{SEC} =: f_{\text{placeOffer}} & , \text{ if } operation = \text{“placeOffer”} \\ 5\ \text{SEC} =: f_{\text{acceptOffer}} & , \text{ if } operation = \text{“acceptOffer”} \\ 0\ \text{SEC} & , \text{ else} \end{cases} \tag{3.9}$$

During the execution of the trading operation, the fees are transferred from the user's blockchain account to the escrow account for later distribution. As already indicated, the markets charge the fees $f_{\text{userRegistration}}$ and $f_{\text{derRegistration}}$ during the blockchain-based registration processes. After successful approval by the DGA, they are transferred to the latter to compensate for the effort. This mechanism also helps to prevent fake registration requests that attempt to attack the system and try to overload the DGA. On the other hand, the markets use $f_{\text{placeOffer}}$ and $f_{\text{acceptOffer}}$ to encourage users to follow the correct trading behavior and to initiate the semi-automated payment process later on. We already motivated the necessity of the former above. The latter, however, is motivated due to the fact that the payment process is a blockchain transaction that has to be initiated manually by a user. If a user involved in a trade initiates the process, we use $f_{\text{placeOffer}}$ and $f_{\text{acceptOffer}}$ to pay them a reward of 1 SEC. The costs for this reward are equally distributed among all other involved trading partners and deducted from their fees deposited. However, if the payment process cannot be processed automatically due to missing measurement data, the DGA is involved in resolving the conflict. After an off-chain consolidation of all trading partners, the DGA decides how to distribute the deposited funds for the trading volume among them. Thereby, the actual process depends on the individual case and is not formally defined here. Finally, the DGA uses the fees $f_{\text{placeOffer}}$ and $f_{\text{acceptOffer}}$ to pay itself an expense allowance of 4 SEC. In this case, the trading partner that initiated the semi-automated trading process which raised an escrow conflict due to missing metering data still receives a reward of 1 SEC. The remaining deposited fees will be refunded equally among all trading participants.

Algorithm 3 portrays the semi-automated payment distribution process based on the trading data, available smart meter readings, and the described escrow policies. In order to be able to understand this formal process in detail, we recommend that the interested reader first reads the following Section

3.3, because it contains further formal definitions of the offers, which are essential for understanding the algorithm. In the following, we will finally introduce the semantics and rules of the two market variations intraday and day-ahead.

3.3 Market Description

In the following, we finally introduce and explain the markets and the trading processes themselves. For this purpose, we will introduce the two short-term market variants *intraday* and *day-ahead*. As already discussed in Chapter 2.1.3, the flexibilities of the DERs are only available at short notice and therefore, can only be scheduled at short notice. For this reason, we think that especially these short-term market variants are best suited for the trading of their flexibilities. In the two markets presented, ROs can offer the flexibility of their operated DERs to GOs for the use as operating reserves, which the latter can integrate into their grid congestion management processes. In this context, flexibility refers to the schedulable and controllable production and consumption of energy. In particular, the ROs offer two types of flexibility products - *positive* and *negative* flexibility. The former corresponds to an increase in production or a reduction in consumption. Therefore, it is used by the GOs for grid stabilization in the event of grid bottlenecks. On the other hand, the offered negative flexibility corresponds to an increase in consumption or a reduction in production, which the GOs can utilize when the grid is underloaded. As we will explain in more detail later, flexibilities are traded on the day-ahead market on a daily basis. Therefore, the GOs can use it to purchase an operating reserve for the long-term planned load balancing processes. As will be shown, the intraday market is suitable for compensating for short-term planning deviations, since here flexibilities are traded on an hourly basis.

For each offer, in both markets, the time is divided into 15-minutes intervals. Therefore, the RO must provide a value for the offered positive and negative flexibility for each 15-minutes interval of the corresponding trading period in each offer. The flexibility values are expressed as power values in unit *kilo watt (kW)*. In addition, for each of these intervals, the RO must specify the desired price for each flexibility value that they expect for the supply of the defined power. Thereby, throughout the markets, prices per interval are expressed in $\frac{SEC}{kWh}$ using our proprietary cryptocurrency SEC. If no positive or negative flexibility is available for an interval, the RO has to set the related flexibility value and the associated price to zero. Based on this information, we can now broadly describe trading in both markets as follows:

1. For one of their registered DERs, an RO places an offer on the market in order to advertise flexibility available by it. For each of the discretized time intervals, they specify a negative and positive flexibility value as power values and additionally defines a corresponding price for each of these values. Furthermore, the system will transfer the fee $f_{\text{placeOffer}}$ charged for the market operation from the RO's blockchain account to the escrow account. Algorithm 1 illustrates this process schematically.

2. After a predefined deadline for placing offers has expired, all GOs can retrieve the placed offers of a trading period from the markets and can accept them. Thereby, we emphasize that a GO does not need to acquire all the flexibility advertised in an offer. For each of the 15-minutes intervals

represented in an offer, the GO can choose how much of the positive and negative flexibility offered they intend to purchase. However, while doing so, they must accept the price per interval determined by the RO. The remaining flexibility is still available for purchase to the other GOs. Therefore, it is clear that no auction takes place at the markets, but direct purchases of flexibility where we apply the principle FCFS. Before a predetermined deadline for accepting offers expires, a GO can edit or withdraw already purchased offers. The specific time constraints for placing and accepting offers vary in both markets and we will explain them in detail later. As discussed above, the GO also has to provide the value of the purchased trading volume and the charged fee $f_{\text{acceptOffer}}$ as security. Therefore, the system will automatically transfer these values from the blockchain account of the GO to the escrow account during the purchase. Algorithm 2 formally represents the process of accepting and purchasing an offer.

3. At the end of the accepting period, it is determined to which GOs the RO has to deliver the sold flexibilities. When the represented trading period starts, for each 15-minutes interval the RO must deliver the flexibility sold to all purchasing GOs. Parallel to this process, the installed smart meters measure the consumed and produced energy for each DER. While doing so, they transmit this reading to the SMM, which verifies them and stores them on the blockchain. Later we use these readings to prove the delivery of the service in the payment process.

4. After the expiry of a final deadline, anyone involved in a trade can now initiate the payment process based on the escrow mechanism described above. We introduce this deadline to ensure that the SMM has sufficient time to receive and store all smart meter readings on the blockchain. Once a trading partner initiated the payment process, the system automatically distributes the deposited security based on the escrow mechanism described in Chapter 3.2. Otherwise, the assets remain in the escrow account. Algorithm 3 explains the process of distributing the funds formally.

The trading process is the same for both the presented markets. Only the time constraints for the submission of offers, the purchase of these, the delivery of the flexibilities, and the initiation of the blockchain-based payment process differ. Therefore, we will define these conditions more precisely for both markets in the following.

3.3.1 Day-Ahead Market

As mentioned above, on the short-term day-ahead market flexibility is traded on a daily basis. Thereby the ROs offer the flexibility of their resources one day before it is available and can be provided. Offers in this market are discretized in 15-minutes intervals. Therefore, an offer consists of a total of 96 time intervals. As described above, the ROs define a positive and negative flexibility value for each of these intervals as well as the prices to be charged in each interval. Formally, a day-ahead offer o for the trading period t, placed by the resource operator ro for its registered DER r can be defined as a 5-tuple as follows:

$$o := (t, r, ro, F_{96}^{+}, F_{96}^{-}) \tag{3.10}$$

where the lists F_{96}^{+} and F_{96}^{-} for the positive and negative flexibility series are defined as follows:

$$F_k^+ := ((f_0^+,p_0^+),....,(f_{k-1}^+,p_{k-1}^+)) \tag{3.11}$$

$$F_k^- := ((f_0^-,p_0^-),....,(f_{k-1}^-,p_{k-1}^-)) \tag{3.12}$$

Thereby, the i-th tuple (f_i^+,p_i^+) of the list F_k^+ is to be understood as the offered positive flexibility with power value f_i^+ in kW at the price of p_i^+ in $\frac{SEC}{kWh}$. Here the power f_i^+ has to be hold and generated for the entire time interval $[t + i \cdot 15min, t + (i + 1) \cdot 15min[$. The same applies to the negative flexibility series F_k^-. The trading period t is defined as the Unix timestamp of the beginning of the trading day.

In the same manner, a_o can be defined to describe the acceptance of the day-ahead offer o for the DER r in the trading period t accepted by the grid operator go:

$$a_o := (t, r, go, S_{96}^+, S_{96}^-) \tag{3.13}$$

Here, the lists S_{96}^+ or S_{96}^- represent the positive or negative shares accepted for purchase by the GO go from the positive or negative flexibilities offered in F_{96}^+ or F_{96}^- and are defined as follows:

$$S_k^+ := (s_0^+, ..., s_{k-1}^+) \tag{3.14}$$

$$S_k^- := (s_0^-, ..., s_{k-1}^-) \tag{3.15}$$

In this case, the i-th element s_i^+ of the list S_k^+ is to be interpreted as the share of the power f_i^+ from the i-th tuple (f_i^+,p_i^+) of the positive flexibility series F_k^+ of the offer o which the GO intends to purchase. The value s_i^+ is expressed as power value in kW. The same applies to the list S_k^- and the negative flexibility series F_k^- of the offer o.

For the proposed day-ahead market, fixed time constraints apply concerning which of the trading operations described above are allowed when. Table 3.1 provides an overview of these time constraints and the corresponding intervals. If a RO wants to place an offer for the trading day d, they must place the offer no later than 6 p.m. the day before. They can edit a placed offer at any time up to this deadline. Thereby, the RO can also withdraw the offer by providing zero filled power series while editing the offer within this deadline. On the day before between 6 p.m. and 10 p.m. all GOs can retrieve the placed offers and accept them. After that, a 2-hour offline processing phase takes place until the beginning of the trading day, in which the ROs can prepare the delivery of the services. During the trading day itself, the RO must now deliver the sold flexibilities to the respective GOs for each 15-minutes interval. At the same time, the SMM transfers measurement data on the energy produced or consumed to the blockchain. In order to allow the SMM a sufficient time for this activity, the payment process for the flexibility supplied can only be initiated after 48 hours of the beginning of the trading period d. Thus the SMM can delay the transmission of the measurement data for at most 24 hours after the last reading received. Finally, any trading partner involved can initiate the payment process after this period expired.

Market Action	**Day Ahead** *Intervals for day d*	**Intraday** *Intervals for hour t*
ROs place, edit, and withdraw offers	$[0, d - 6h[$	$[0, t - 2h[$
GOs accept offers	$[d - 6h, d - 2h[$	$[t - 2h, t - 1h[$
ROs provide flexibilities	$[d, d + 24h[$	$[t, t + 1h[$
ROs or GOs initiate payment process	$[d + 48h, \infty[$	$[t + 24h, \infty[$

Table 3.1: An overview illustrating the time constraints of which market operations are allowed in which time intervals on the proposed markets.

3.3.2 Intraday Market

The intraday market is a very short-term market in which flexibility products can be traded on an hourly basis up to 2 hours before delivery. Here, an offer consists of four 15-minutes intervals. Analog to the formal definition of day-ahead offers, an intraday offer o for the trading period t placed by the resource operator ro for its registered resource r can be defined as a 5-tuple as follows:

$$o := (t, r, mp, F_4^+, F_4^-) \tag{3.16}$$

In the same way, a_o can be defined to accept an intraday offer o by grid operator go for the resource r in the trading period t:

$$a_o := (t, r, go, S_4^+, S_4^-) \tag{3.17}$$

Thereby the lists F_4^+, F_4^-, S_4^+ and S_4^- refer to the definitions in Eq. 3.11-3.15. In contrast to the day-ahead offers, the trading period t represents the beginning of the traded trading hour as Unix timestamp here. At this point, we would like to introduce function $costs(\cdot)$, which determines the cost of a purchase a_o of the corresponding intraday or day-ahead offer in SECs. This function will be valuable in the formal description of the processes for placing and accepting an offer as well as the payment process in the algorithms 1, 2, and 3. The function is therefore defined as follows:

$$costs(a_o) := \sum_{i=0}^{n \in \{4-1, 96-1\}} ((a_o.S^+[i].s^+ * o.F^+[i].p^+) + (a_o.S^-[i].s^- * o.F^-[i].p^-)) * 0.25h \tag{3.18}$$

Time constraints for the execution of market operations also apply on the intraday market. These are well defined and depicted in Table 3.1. Up to two hours before the flexibility is to be delivered, a RO can place, edit ,and withdraw an offer. Afterwards, the GOs have one hour to review the placed offers and purchase the flexibility products. After that, a one-hour offline processing phase takes place until the flexibility is to be delivered. Subsequently, the RO must supply the flexibility sold to the GOs and an offline payment transaction takes place for the delivered operating reserve. Afterward, there is again a waiting period of 24 hours in which the SMM has sufficient time to store the measured data on the blockchain. Once this has expired, any trading partner involved can initiate

the semi-automated payment process and thus officially close the trade.

Finally, based on the formalisms introduced in this chapter, we want to define some of the essential blockchain-based market operations formally. Therefore, Algorithm 1 and 2 represents the processes and corresponding requirements of placing and accepting an offer. In contrast, Algorithm 3 represents the semi-automated payment process based on the escrow mechanism described in this chapter. We will not discuss them again because the algorithms illustrated here are just the formal definition of these three processes discussed in this chapter in detail using the formalisms specified.

In conclusion, in the last sections, we introduced and formally described the markets, processes, and actors. We have also clearly stated which technical and behavioral prerequisites we expect from market actors in order to implement the described markets. We designed the markets in a privacy-aware manner in which users do not reveal their real-world identity or their DERs. We can guarantee real-world users' and DERs' integrity while protecting their privacy in the presented market mechanisms. Therefore, we enabled the implementation of free, protected, non-discriminatory, and privacy-aware blockchain-based operating reserve markets.

Algorithm 1 Algorithm to place, edit, and withdraw offer $o = (t, r, ro, F_i^+, F_i^-)$.

Require: $i \in \{4, 96\} \land t \ mod \ 15min = 0$
Require: User ro is approved as resource operator
Require: DER r registered and approved as well as assigned to user ro
Require: Time requirements for placing offer fulfilled

1: **function** PLACEOFFER(t, r, ro, S_i^+, S_i^-)
2: $operationCosts \leftarrow fees$("placeOffer")
3: $o_{old} \leftarrow readPlacedOffersOfUser(t,r,ro)$
4: **if** o_{old} is not null **then** ▷*ro has already placed offer for r in t*
5: **if** $F_i^+ and F_i^-$ are filled with zeros **then** ▷*ro wants to withdraw offer*
6: $remove(o_{old})$
7: transfer(escrow, address(ro), operationCosts) ▷*refund fees*
8: return
9: **else**
10: $operationCosts \leftarrow 0$ ▷*fees already paid*
11: **end if**
12: **end if**
13: **if** $balance(address(ro)) < operationCosts$ **then**
14: revert
15: **end if**
16: $o \leftarrow (t, r, ro, F_i^+, F_i^-)$
17: $transfer(address(ro), escrow, operationCosts)$ ▷*deposit fees*
18: $remove \ o_{old}$ ▷*no effect if o_{old} is null*
19: $store \ o$
20: **end function**

Algorithm 2 Algorithm to create $a_o = (t, r, go, S_i^+, S_i^-)$ representing the acceptance of offer o.

Require: $i \in \{4, 96\} \land t \ mod \ 15min = 0$
Require: User go is approved as grid operator
Require: There exists a offer o for DER r for trading interval t
Require: Enough flexibility for all i intervals in o still available and not purchased yet
Require: Time requirements for accepting offer fulfilled

function ACCEPTOFFER(t, r, go, S_i^+, S_i^-)
  $a_o \leftarrow (t, r, go, S_i^+, S_i^-)$
  $operationCosts \leftarrow costs(a_o) + fees(\text{“acceptOffer”})$
  $depositToRefund \leftarrow 0$
  $a_{b,old} \leftarrow getAcceptedOffersOfUser(t, r, go)$
  if $a_{o,old}$ is not null **then** ▷*go has accepted offer o in t already*
    if $S_i^+ and S_i^-$ are filled with zeros **then** ▷*go wants to withdraw offer*
      $depositToRefund \leftarrow costs(a_{o,old}) + fees(\text{“acceptOffer”})$
      $transfer(escrow, address(go), depositToRefund)$ ▷*refund deposited fees*
      remove $a_{o,old}$
      return
    else ▷*go wants to update their shares*
      $depositToRefund \leftarrow costs(a_{o,old})$
      $operationCosts \leftarrow operationCosts - fees(\text{“acceptOffer”})$ ▷*fees already paid*
    end if
  end if
  if $balance(address(go)) < (operationCosts - depositToRefund)$ **then**
    revert
  end if
  $transfer(escrow, address(go), depositToRefund)$
  remove $a_{o,old}$ ▷*no effect if* $a_{o,old}$ *is null*
  $transfer(address(go), escrow, operationCosts)$
  store a_o
end function

Algorithm 3 Algorithm that initiates the semi-automated escrow mechanism to distribute the payments deposited for offer $o = (t, r, ro, F_i^+, F_i^-)$.

Require: $i \in \{4, 96\} \wedge t \ mod \ 15min = 0$
Require: Time requirements for initiating payment process fulfilled
Require: Escrow not in state *cleared* or *conflict*

function CLEARESCROW(o)
  $h_{smartMeter} = getHashOfSmartMeterAssignedToDER(r)$
  $M_{t,h_{smartMeter}} = getReadingsOfSmartMeterInInterval(h_{smartMeter}, t, t + i * 15min)$
  if $|M_{t,h_{smartMeter}}| < i$ **then**
    change escrow state to *conflict*
    inform DGA to solve escrow issue due to missing meter readings
    return
  end if
  $A_o \leftarrow getAcceptancesOfOffer(o)$
  $rewardShare \leftarrow 1 \text{ SEC} * (|A_o| + 1)^{-1}$
    ▷*ro can have placed offers for r in both trading variants overlapping in time.*
  $SoldFlex^+ \leftarrow getAlreadySoldFlexOfOtherTradingVariant(o, +, t, t + i * 15min)$
  $SoldFlex^- \leftarrow getAlreadySoldFlexOfOtherTradingVariant(o, -, t, t + i * 15min)$
  for $a_o \in sortedByFCFS(A_o)$ **do**
    $costsForDeliveredFlex^+ \leftarrow 0$
    $costsForDeliveredFlex^- \leftarrow 0$
    for $k \leftarrow 0$ to $i - 1$ **do**
      for $series \in \{+, -\}$ **do**
        $flexLeft^{series} \leftarrow M_{t,h_{smartMeter}}[k].e_{relative^{series}} - SoldFlex^{series}[k]$
        $flexRequested^{series} \leftarrow a_o.S_i^{series}[k].s^{series} * 15min$
        $flexAssigned^{series} \leftarrow 0$
        if $flexLeft^{series} \geq flexRequested^{series}$ **then**
          $flexAssigned^{series} \leftarrow flexRequested$
        else if $flexLeft^{series} \leq flexRequested^{series} \wedge flexLeft^{series} \geq 0$ **then**
          $flexAssigned^{series} \leftarrow flexLeft^{series}$
        end if
        $costsForDeliveredFlex^{series} \leftarrow o.F_i^{series}[k].p^{series} * flexAssigned^{series}$
        $SoldFlex^{series}[k]$ += $flexAssigned^{series}$
      end for
    end for
    $totalCosts \leftarrow costsForDeliveredFlex^+ + costsForDeliveredFlex^-$
    $refund \leftarrow (costs(a_o) - totalCosts) + fees(\text{“}acceptOffer\text{''}) - rewardShare$
    $transfer(escrow, \ address(o.ro), \ totalCosts)$
    $transfer(escrow, \ address(a_o.go), \ refund)$
  end for
  $transfer(escrow, \ address(o.ro), \ fees(\text{“}placeOffer\text{''}) - rewardShare)$
  $transfer(escrow, \ address(callingUser), \ 1 \text{ SEC})$ ▷*distribute clear escrow reward*
  change escrow state to *cleared*
end function

3.4 Requirements Analysis

In the following section, we will identify the functional and non-functional requirements of the system to be developed based on the abstract modeling of the markets from the previous section. In this context, functional requirements are specific functionalities that the system must satisfy in order to enable the technical implementation of the markets [58]. Automated tests are often used to verify these. In contrast, non-functional requirements are unspecific characteristics that describe the quality of the system under development [58]. These, in turn, often cannot be tested automatically and are more challenging to evaluate. For clarity, we define the requirement levels according to the definition of the RFC 2119 standard [59]. Therefore, the term *MUST* defines requirements that the system must satisfy. *MUST NOT* are negative requirements, which must not be realized. Requirements that are defined by the terms *SHOULD* and *SHOULD NOT* are concrete recommendations. The term *MAY* is used to define optional requirements.

3.4.1 Functional Requirements

Requirement 1 The SMM MUST be able to register smart meters in the system. The smart meters are stored and identified on the blockchain via the SHA-256 hash of their real-world identifier.

Requirement 2 The SMM MUST be able to store consumption and production readings of registered smart meters in the system, as described in Chapter 3.2. Subsequently, only the system, the DGA, and the SMM itself MUST be allowed to read the stored data.

Requirement 3 The system MUST provide a proprietary cryptocurrency, which is used for all payment processes in the system. This is called Smart Energy Coin (SEC). All actors MUST be able to acquire and spend coins. The SEC cryptocurrency SHOULD be compatible with the ERC-20 token standard [60].

Requirement 4 The system MUST be able to distinguish between two types of market users: the resource operators and grid operators. They MUST be able to register as described in Chapter 3.1.1. After registration, the DGA MUST be informed via an event about the registration process. Finally, the DGA MUST be able to store verification information of the users in the system to unlock or block them on the markets. In the beginning of the registration, the users have to deposit a registration fee of 10 SEC, which will be transferred to the DGA after verification by the DGA to compensate the effort.

Requirement 5 ROs that have been successfully registered and verified by the DGA MUST be able to register their operated DERs in the system. The registration process MUST be implemented as described in Chapter 3.1.1. If a DER is registered, the DGA is informed by an emitted event about an upcoming verification. The DGA MUST then be able to store the results of this verification of the DER in the system. The ROs MUST also deposit a registration fee of 10 SEC at the beginning of the registration process, which is then transferred to the DGA after verification (positive and negative) to compensate for the effort.

Requirement 6 Users who have not yet been successfully verified by the DGA MUST NOT be allowed to participate in the markets. Also, for DERs that have not yet been successfully verified by the DGA, it MUST NOT be permitted to offer operating reserve in the markets.

Requirement 7 Requirements 8-13 MUST be implemented for the two market variants intraday and day-ahead, as described in Chapter 3.3.

Requirement 8 For the requirements 9-13, before processing the corresponding market action, the system MUST ensure that the timing constraints shown in Table 3.1 are met. If this is not the case, the requests MUST be rejected.

Requirement 9 Registered ROs MUST be able to offer operating reserve of their registered DERs on the markets. This process must be implemented as described in Chapter 3.3 and Algorithm 1. Thereby, a 5 SEC place offer fees MUST be deposited in an escrow mechanism, which the system MUST initialize for every offer. Additionally, ROs MUST be able to edit and withdraw the placed offers.

Requirement 10 GOs MUST be able to accept offers and purchase shares of them. The acquisition process MUST be implemented as described in Chapter 3.3 and Algorithm 2. In the process, the GOs MUST deposit an accept offer base fee of 5 SEC in the escrow mechanism. Furthermore, they MUST also deposit the monetary costs of the operating reserve to be acquired from the offer. Additonally, GOs MUST be able to edit and withdraw the requested shares of accepted offers.

Requirement 11 ROs MUST be allowed to only read information about their own placed offers. In contrast, GOs MUST be permitted to access information about all offers.

Requirement 12 The DGA and each trading partner involved in an offer MUST be able to initiate the settlement of the trade. The system will assign operating reserves to each GO based on the stored trading data and smart meter readings in this process. Afterward, the deposited escrow funds will be distributed according to the costs of the assigned operating reserve. This escrow clearing process MUST be implemented as described in Chapter 3.2 and Algorithm 3 and is based on real smart meter readings. If at least one of these is missing, the escrow MUST be marked as conflicting and the DGA must be informed to solve the conflict. The user who initiated the clearing process receives a reward of 1 SEC, which will be equally deducted from all involved trading partners' deposited funds.

Requirement 13 The DGA MUST be able to solve escrow conflicts. Thereby, it MUST be capable of storing settlement data of the traded operating reserve in the system, which initiates the distribution of the deposited escrow funds. In this process, the DGA manually assigns an operating reserve to each involved GO. Afterward, the DGA receives a reward of 4 SEC for compensating for the effort. The reward will be equally deducted from the deposited funds of all involved trading partners. The process of solving escrow conflicts is described in Chapter 3.2.

3.4.2 Non-functional Requirements

Requirement 14 The system SHOULD be implemented with the help of a blockchain. The blockchain technology used has to implement the concept of general-purpose smart contracts. The functional requirements SHOULD then be implemented using smart contracts wherever possible.

Requirement 15 The system is intended to protect the privacy of users and their trading data. Thereby, actors and DERs MUST only be identified pseudonymously. In this way, it SHOULD not be possible to associate the trading behavior of blockchain users and DERs to real-world users.

Requirement 16 Additionally, the system SHOULD be secure in terms of confidentiality and data integrity, although comprehensive security cannot be guaranteed.

Requirement 17 A stable operating behavior of the system is expected. This means that the system SHOULD able to process requests in a reliable manner, even if the number of requests is increased. Furthermore, invalid requests MUST also be processed reliably.

Requirement 18 The system SHOULD be performant and SHOULD be able to process requests in a reasonable time frame. Furthermore, the resource requirements SHOULD be kept as low as possible.

4 Architecture

After describing the semantics of the markets in the last chapter in an abstract manner, we want to begin to model the technical realization of these in the following chapter. To this end, we will describe the architecture of the proposed system, identify system components, and explain how they interact. Thereby, we will provide a detailed specification of these dynamic aspects represented as processes.

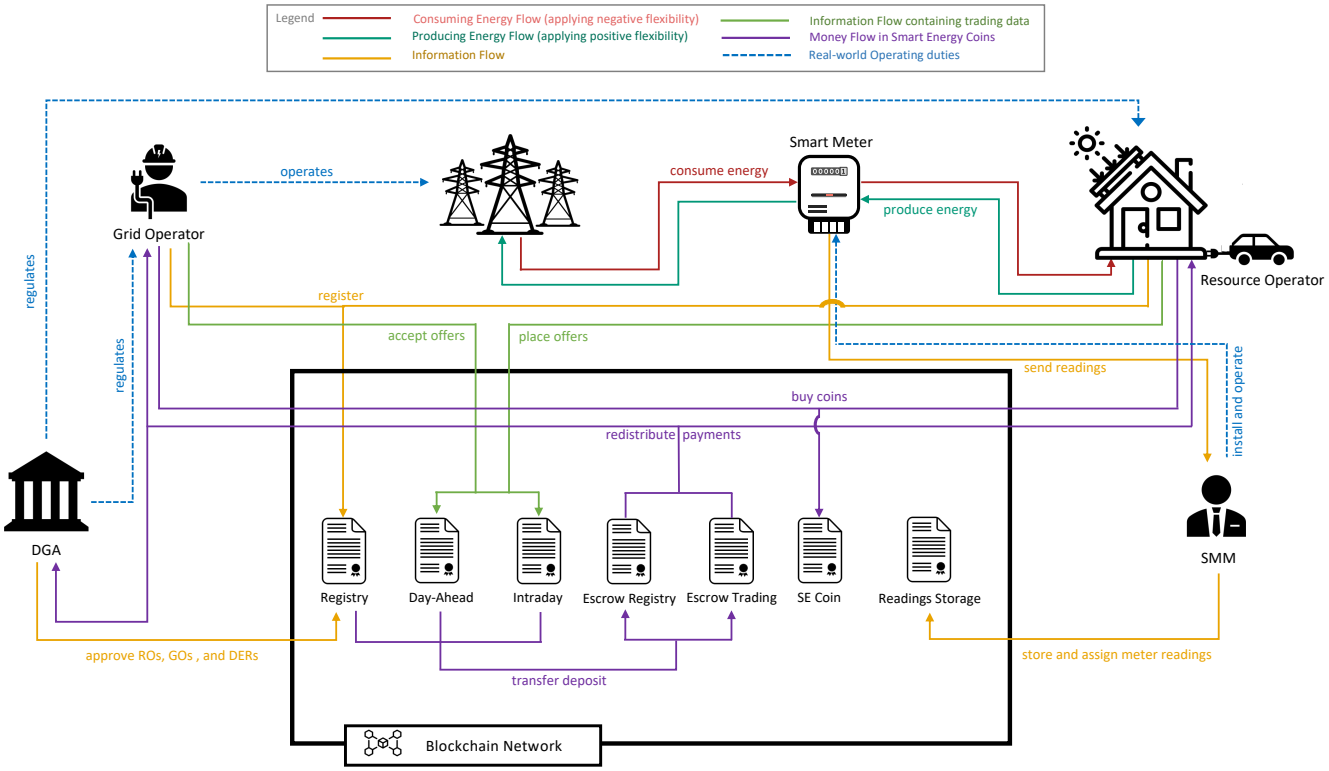


Figure 4.1: Architecture of the proposed system showing all components and actors. The diagram also shows their blockchain-based and real-world interaction.

4.1 System Components and Actors

Based on the description in Chapter 3, Figure 4.1 represents the architecture of the system. It depicts all system components and actors. Furthermore, their blockchain-based and real-world interactions are graphically illustrated. The main technical component of the system is a peer-to-peer blockchain network. On this blockchain, we will provide various smart contracts, which represent the technical

implementation of the markets and are to be understood as system components as well. Precisely, we will provide seven seperate smart contracts: the *Registry*, *Day-Ahead*, *Intraday*, *Escrow Registry*, *Escrow Trading*, *SE Coin*, and *Readings Storage* contract. Each of these implements a different aspect of the proposed markets. They provide interfaces to users which allow them to perform operations at the markets, which will be explained later. For the concrete system implementation, no specific blockchain technology is required. The only requirement we demand from the technology used is that it implements the concept of general-purpose smart contracts.

Furthermore, the system consists of four real-world actors representing the users of the system: the *grid operators*, *resource operators*, the *data and grid authority*, and the *smart meter manager*. These actors either participate in the markets or are required to enable a reliable realization of these. In the following, we will explain the actors, system components, and their interaction in more detail.

4.1.1 Actors

As can be deduced from the requirements analysis of the previous chapter, we can identify four actors and users of the system. Figure 4.2 illustrates them and lists their use cases, activities, and duties in the designed system. In the following, we will explain these and their interaction with system components. However, we try to keep it short and focus on the relevant information required for the system implemenation, as we have already described the actors in detail in Chapter 3.1.

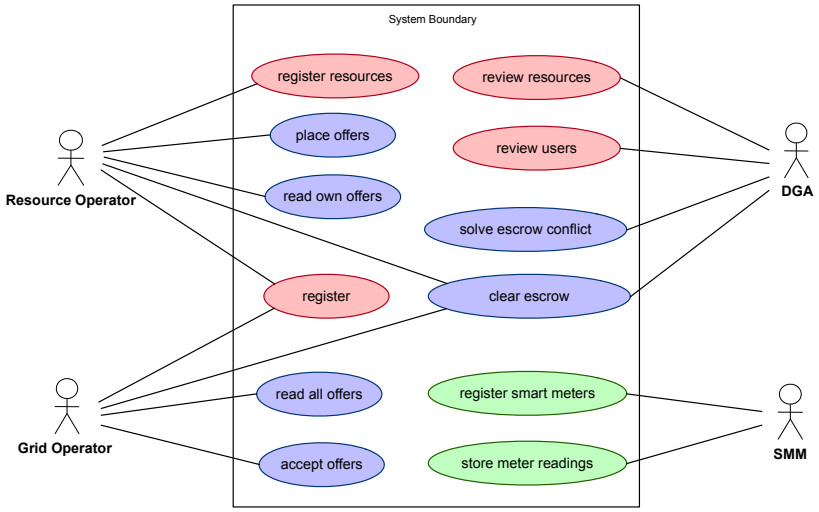


Figure 4.2: Diagram showing the use cases, activities, and duties of the 4 identified actors in the designed system. We distinguish between trading (blue), registering (red) and metering (green) related use cases and processes.

Resource Operators

The resource operators represent the supply side in the markets. As the name implies, they own or operate DERs. The latter are capable of providing operating reserve, which the ROs will offer at the blockchain-based markets. In this context, the ROs are allowed to place offers, as well as edit and withdraw them. Furthermore, in order to get paid for the service provided, they can initiate

the escrow clearing process of one of their offers in the system. Depending on the trading variant used, they interact with the smart contracts Day-Ahead, Intraday, Escrow Trading, and SE Coin. Furthermore, the ROs can only obtain information about their own placed offers from the markets.

Before ROs are allowed to place offers, they must register themselves and their DERs in markets and need to be verified by the DGA. To this end, they also interact with the Registry and Escrow Registry smart contracts.

Grid Operators

As can be seen in Figure 4.1 and 4.2, the grid operators represent the demand side of the markets. They use the acquired operating reserve to balance their operated grids and maintain grid stability. In our system, the GOs can retrieve and review all placed offers from the markets. After inspecting them, they are able to purchase shares of the offered operating reserve by accepting offers. In this context, they can also edit or withdraw the allocated shares later on. Furthermore, the GOs are allowed to initiate the escrow clearing process for all offers in which they have purchased shares. This is useful to receive a refund if the RO has not provided all of the acquired operating reserve. For the aforementioned purposes, they interact with smart contracts Intraday, Day-Ahead, Escrow Trading, and SE Coin.

As a prerequisite, the GOs must first register at the markets and need to be reviewed by the DGA. For this reason, they also interact with the Registry and Escrow Registry smart contracts.

Data and Grid Authority (DGA)

The data and grid authority constitutes a central actor in our proposed system. It is represented by a governmental institution. The task of DGA is to ensure the integrity and security of the energy system and to resolve conflicts between trading partners. As shown in Figure 4.1 and 4.2, the DGA is responsible for reviewing and verifying registration information of users and DERs in our system. To this end, it uses the Registry smart contract to read registry information of users and DERs and to store the review results in our blockchain-based system. In this context, we want to mention that our system will regularly trigger events to inform the DGA about upcoming tasks.

The DGA fulfills another essential aspect in the context of the already mentioned escrow mechanism. On the one hand, the DGA is also authorized to initiate the escrow clearing process to avoid orphaned deposited funds. On the other hand, it is responsible for solving escrows in conflicting states. These can occur in case of missing meter readings. In this case, no automated assignment of the operating reserve to GOs is possible. Therefore, the payment process cannot be executed. That is why we expect the DGA to contact all GOs and ROs involved in the real world in order to clarify who has received or consumed how much operating reserve. It then preprocesses the gathered information and use the Escrow Trading to store it on the blockchain. Finally, the payment process can be performed as described in Chapter 3.2.

Smart Meter Manager (SMM)

In our system, the smart meter manager is responsible for managing the smart meters and their generated readings. To this end, the SMM uses the Readings Storage smart contract to store information

about registered smart meters. Besides, it uses the same contract to store readings of the registered smart meters. These measurements are essential for the semi-automated payment process.

4.1.2 System Components and Smart Contracts

After having just explained the actors and their interaction with the system components, we will describe these components in more detail in the following section. For this purpose, we will outline the individual smart contracts represented in Figure 4.1. Each of the depicted contracts fulfills a separate logical tasks in the developed market system. Later, these will be deployed to the main system component - the blockchain network. Subsequently, the actors can use the markets by interacting with the deployed smart contracts via blockchain transactions. In the following, we will briefly describe the smart contracts and the blockchain network.

Blockchain Network

In our proposed system, a blockchain network represents the main technical component. On the blockchain managed by the underlying peer-to-peer network, the designed smart contracts are deployed and executed by all network nodes. In order to perform market operations, the actors of our system create blockchain transactions addressed to the respective smart contracts and broadcast them to the network. Afterward, the nodes will process the transactions, which results in the execution of the stored smart contract code. This execution modifies the status stored on the blockchain and persists market actions in this way.

By using the blockchain technology, our system also benefits from the characteristics of this novel concept (see Chapter 2.1.1). On the one hand, all transactions and market activities are processed decentralized. Therefore, thanks to the used consensus mechanism, it is possible in our system to guarantee the validity of the transaction executions without a central instance and without trusting the network. Furthermore, the cryptographic security measures provided by the blockchain technology render all transactions and data immutable, which leads us to the conclusion that our system is also tamper-proof, trustworthy, and secure. Finally, since all data is processed and stored decentralized, where each network node holds a complete copy of the blockchain and executes all transactions, the system is also considered fault-tolerant and fail-safe. As we have already mentioned, we are not demanding any specific blockchain technology. The only requirement we demand is that the used blockchain implements the concept of general-purpose smart contracts.

In the following, we want to explain the particular smart contracts, which are deployed in this blockchain network and executed there in a decentralized manner.

Smart Energy Coin

The Smart Energy Coin smart contract represents a proprietary cryptocurrency by implementing the ERC-20 token standard [60]. We intend to use this currency as a payment instrument for all trading actions and fees in the described market system. The smart contract provides an interface to transfer SEC coins between users and other contracts. Furthermore, the Smart Energy Coin component provides functions to delegate other users with the transfer of such coins.

Registry

The Registry smart contract provides an interface for any registration related logic of the system. By using this interface, GOs and ROs can register to participate in the markets. The latter can also register their DERs here. As a result, the Registry component stores information about registered GOs, ROs and DERs and verifies that the registration fees were deposited. As described in Chapter 3.1.1, all privacy-related user data is either stored encrypted or pseudonymous. Besides, the contract also administers an assignment of DERs to smart meters.

Furthermore, since the users must be verified by the DGA before participating in the markets, the Registry smart contract is also utilized by the DGA. Precisely, the DGA is authorized to obtain registration information of users and DERs from this contract. Furthermore, the Registry smart contract offers an interface through which the DGA can provide the verification results and thus either unlock or block users on the markets. Finally, other smart contracts can use the Registry interface to access registration information of market participants and involved DERs.

Escrow Registry

In the Escrow Registry smart contract, we realized an escrow mechanism for the charged registration fees (see Chapter 3.1.1). At this system component, market participants can deposit the required user and resource registration fees. Since the registration process consists of several time-delayed actions of several actors, we cannot perform the payment process instantaneously. Therefore we have introduced this contract to temporarily store and manage the funds. During registration, the Registry contract can then interact with this component to obtain information on whether specific users indeed have deposited the required fees. Furthermore, after verification by the DGA, the Escrow Registry contract is in the position to transfer the deposited funds to the DGA, in order to compensate it for the effort. However, if the DGA does not perform the verification or significantly delays it, users can apply here for a refund of their deposited funds and thus cancel the registration process.

We have decided to outsource the escrow mechanism to a dedicated smart contract to separate the business logic (Registry contract) from the actual payments and to safely keep them in this contract. By this separation of duties, the system is supposed to be more extendable, verifiable, and transparent.

Readings Storage

The Readings Storage smart contract provides an interface for the SMM to register smart meters and store their readings in the system (see Chapter 3.2). The stored readings represent a crucial part of the semi-automated payment process described later. They are transmitted and stored exclusively by the SMM. Thereby, the readings are assigned to the smart meters via the identifier $h(ID_{smartMeter})$. Subsequently, other smart contracts can use the Readings Storage to read the production and consumption values of DERs by specifiying the associated smart meters. Furthermore, they can also use the interface of this contract to check whether smart meters are registered.

Intraday

The Intraday smart contract is one of the most important components in our system. It represents the technical realization of the intraday market variant by implementing the intraday market semantics described in Chapter 3.3.2. The contract provides an interface to the ROs and GOs which they can use for trading purposes. Precisely, ROs can offer operating reserve of their registered and verified DERs according to the intraday description of the markets. Furthermore, GOs can review these offers and purchase shares of the offered operating reserve. Before executing a market action, the Intraday smart contract ensures that the necessary fees and trading volumes required for the semi-automated payment process have been deposited in an escrow as described in Chapter 3.2. For this purpose, the Intraday component interacts with the Escrow Trading smart contract.

Furthermore, the Intraday smart contract interacts with the Registry contract to ensure that market participants are registered and verified, hold the necessary role, and indeed operate involved DERs.

Day-Ahead

We have implemented the second market variant proposed in the Day-Ahead smart contract. The specification of this component is analogous to that of the Intraday smart contract just described above. Only the trading regulations differ from the intraday description and follow the definition of the day-ahead market semantics described in Chapter 3.3.1.

This smart contract also uses the Registry to guarantee the authenticity of market participants and DERs. Furthermore, the Day-Ahead component interacts with the Escrow Trading component to ensure that the users deposited all required fees.

Escrow Trading

The Escrow Trading smart contract is a central component for the realization of the aforementioned semi-automated payment process. As the name already indicates, the trading related escrow mechanism is implemented here. The presented smart contract is able to initiate and manage an escrow data structure for each individual offer. This data structure stores information about the place and accept offer fees as well as the deposited trading volumes of the GOs (see Chapter 3.2). Furthermore, the smart contract offers an interface through which ROs and GOs can deposit the corresponding funds by providing identifying meta information of an offer. The smart contract will then transfer these funds in the form of SEC from the user's blockchain account to itself for later distribution. In this context, the Escrow Trading component is used by the Intraday and Day-Ahead smart contract to retrieve information about the deposited funds for a specific offer. If these fulfill the requirements described in Chapter 3.2, the trading contracts allow the respective user to execute the particular market action.

Furthermore, the Escrow Trading smart contract implements the realization of the payment process and the allocation of actually consumed and allocated operating reserve. To this end, any involved trading partner or the DGA can initiate the payment process by using the interface of this component. Subsequently, the Escrow Trading reads the stored trading data from the trading contracts. Furthermore, it reads the measurement data of the DER represented in the offer from the

Readings Storage smart contract. Based on the descriptions in Chapter 3.2, the Escrow Trading then use the gathered information to assign operating reserve to the GOs. Subsequently, these assignments are stored in the escrow data structure. Finally, the corresponding deposited funds are distributed as payments or refunds. However, if measurement data is missing in this process, the escrow is marked as conflicting, and the DGA must then manually assign the operating reserve to GOs. Afterward, the payments are distributed based on this information provided by the DGA. Finally, the escrow is resolved.

4.2 Process View

In the following, we want to define the identified use cases, activities, and duties of the actors depicted in Figure 4.2 by describing these dynamic aspects as processes. For each of these, we will discuss the realization in detail and explain which system components are involved in which manner. As illustrated in Figure 4.2, we distinguish between three types of processes in this section. The processes highlighted in red refer to the registration and verification processes of market participants described in Chapter 3.1.1. On the other hand, the processes displayed in blue represent trading related market actions, which enable the proposed functionality of the markets. In contrast, the green processes refer to metering related processes for managing and handling measurement data. Before we describe these dynamic aspects in detail in the following, we want to emphasize that the system will identify the actors by their blockchain account addresses in all specified processes.

4.2.1 Registration Processes

In the following section, we want to begin to discuss the processes of registering users and distributed energy resources. These two processes compromise the four use cases highlighted in red in Figure 4.2. In this context, we will combine the registration and verification of a user or DER into one process description, as these are inseparably related. That is why we conclude it is only reasonable to examine them together.

Register Users

Figure 4.3 illustrates the process of user registration. The provided sequence diagram represents the practical implementation of the abstract description from Chapter 3.1.1. Prior to participating in the proposed markets, users must register as RO or GO in the system. To do so, they must first register in the Registry smart contract component. Thereby the presented process consists of four time-delayed steps, whereby each step constitutes a dedicated blockchain transaction.

As illustrated in Figure 4.3, the users must first use the Smart Energy Coin smart contract to record that the Escrow Registry contract is entitled to transfer the required user registration fee of 10 SEC from the users' blockchain account. To this end, they use the smart contract function `approve()` by initiating a blockchain transaction and broadcast it to the blockchain network. After decentralized processing, the users will receive a first transaction hash as a response. This performed delegation follows the ERC-20 token standard [60] and is required because the transfer of tokens and the execution of a smart contract function cannot be reflected in a single blockchain transaction.

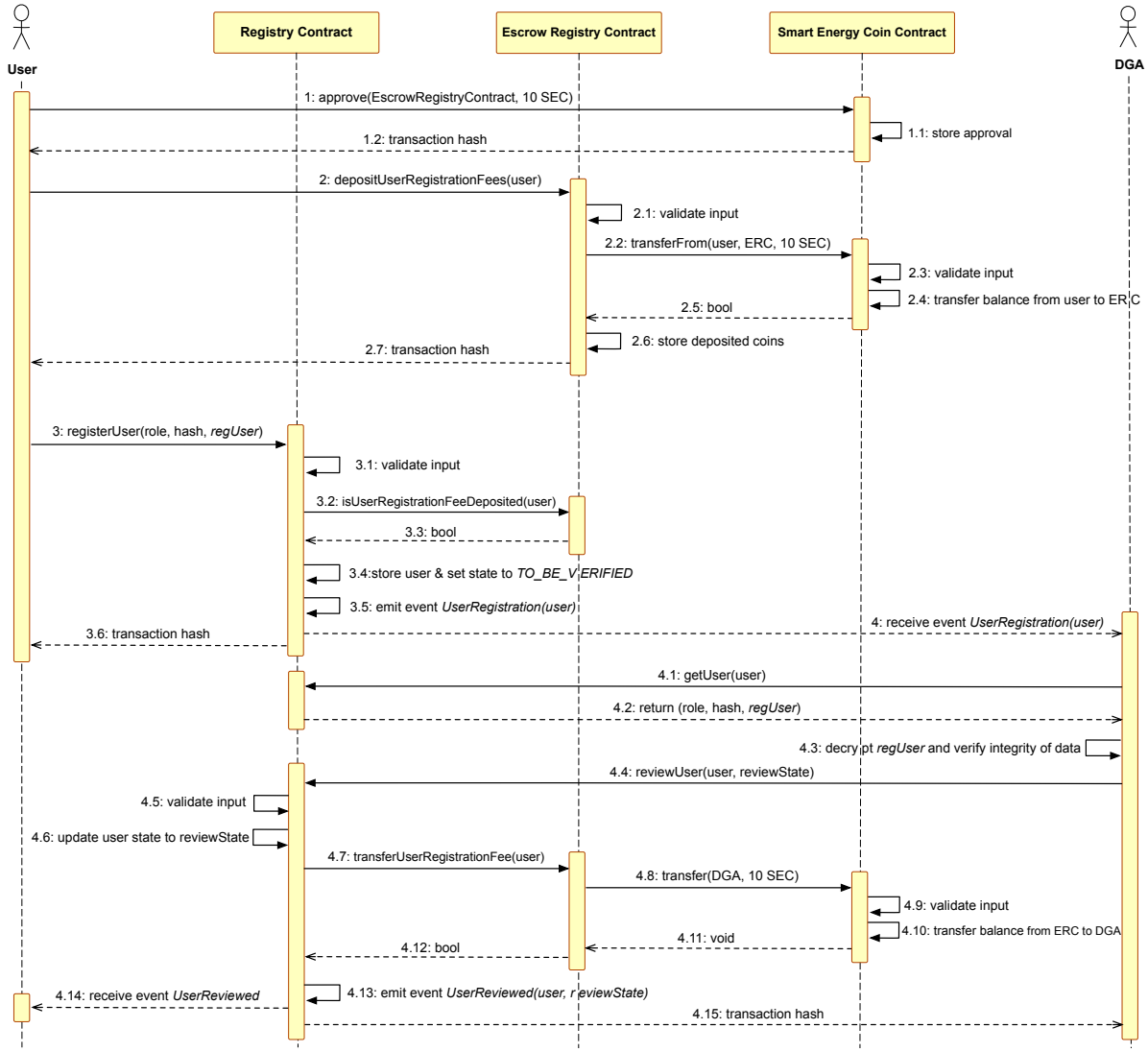


Figure 4.3: Sequence diagram visualizing the user registration process.

Therefore, this delegation authorizes a later called smart contract function to transfer the approved tokens during its execution and thus to combine both operations in a single blockchain transaction.

As a second step, the users have to invoke the function `depositUserRegistrationFees()` of the Escrow Registry smart contract to deposit the required user registration fee in an escrow mechanism. Upon the request, the Escrow Registry uses the Smart Energy Coin contract to transfer the fee of 10 SEC from the blockchain account of the calling user to itself. It is important to note that this is only allowed due to the performed delegation in step 1. Afterward, the Escrow Registry stores that this specific user has deposited the required user registration fees.[1] If the transfer of the fees is not possible, for example because of an insufficient SEC balance, the transaction is reverted. Otherwise, the users receive the hash of the executed blockchain transaction as a response.

1 In this mapping, the user's blockchain account address used during this process is stored to identify them later.

(a) State Machine for User States

(b) State Machine for Resource States

Figure 4.4: State machine for the user and resource states maintained in the system. The state transitions are performed by executing the respective the Registry smart contract functions.

In the third step, the users invoke the function `registerUser()` of the Registry contract to finally register in the system. As discussed in Chapter 3.1.1, the users have to provide their respective market role, their pseudonymous hash identifier used to avoid double registration, as well as the encrypted parameters from the input *regUser* (see Eq. 3.1). As shown in step 3.2, after processing the input, the Registry calls the Escrow Registry smart contract to check if the invoking user has deposited the required 10 SEC registration fee. If this is not the case, the transaction is reverted. In the positive case, however, the Registry Contract creates and stores a new user object representing the user and their role (step 3.4). Later, the stored users will be identified in the system by their blockchain account address used during this registration process. Furthermore, the system will manage a user state for each user. In this initial case, the user is stored in the state *TO_BE_VERIFIED*. This state defines that the user data is stored, but the DGA must still verify the user and their data. Therefore, no market operations are allowed to be executed in the system by the user yet. In step 3.5, the Registry smart contract emits a *UserRegistration* event to inform the DGA that a new user has been registered and needs to be verified.

In the final step, the DGA performs this verification procedure. As shown in step 4.1, the DGA first reads the user data from the Registry. Then it verifies the data as stated in Chapter 3.1.1. Afterward, it stores the result of the verification by invoking the function `reviewUser()` of the Registry contract, which updates the user state. Thereby the DGA specifies the blockchain account address as well as the new state of the user just verified. If the verification was successful, the provided state equals *VERIFIED*. In this state, the users are allowed to use the markets and to perform market actions. If the verification failed, the DGA must provide the state *REJECTED*, which blocks the users on the markets. Afterward, in step 4.7, the Registry smart contract informs the Escrow Registry about the verification by the DGA, which then transfers the deposited 10 SEC user registration fee to the well-known blockchain account of the DGA. This is intended to compensate the DGA for the verification effort. Subsequently, in step 4.13, a *UserReviewed* event is emitted to inform the user about the verification. Finally, the DGA receives the transaction hash and the user registration process is terminated.

User States As already mentioned, the system maintains a state for each user. These states describe which kind of operations the users are allowed to perform in the system. In Figure 4.4 (a), the state machine of the possible user states is depicted. Furthermore, the figure illustrates which functions of

the Registry contract can modify the user state. The initial state after the registration of a user is *TO_BE_VERIFIED*. Users in this state can not yet perform market operations but have to wait for the verification by the DGA. By performing the `reviewUser()` function, the DGA can put the users either in the state *VERIFIED* or *REJECTED*. If the users are in the final state *VERIFIED*, they are successfully verified and can participate in the markets and are allowed to execute market operations. In the state *REJECTED*, however, the users are blocked and cannot participate in the markets. However, by performing the registration process again, they can update their stored registration information and return to the state *TO_BE_VERIFIED*. This is useful in cases where verification has failed only due to incorrect data submitted. In this case, only the data and status would be updated in step 3.4. The last state transition can be accomplished by the function `refundUserRegistrationFee()`. Here, users can proceed from state *TO_BE_VERIFIED* to state *REFUNDED*. The users can execute this function if they are still in the state *TO_BE_VERIFIED* after 75 mined blocks after registration. In this case, the DGA has still not verified them. By invoking the mentioned function, the users can counteract the delayed verification and cancel the registration process which implies a refund of their deposited user registration fee. Afterward, they can register again by performing the process just described.

Register Resources

Before an already registered and verified RO can offer operating reserve for one of their operated distributed energy resources on the markets, they must first register it in the system. This process is similar to the user registration process just described and is formally described in Chapter 3.1.1. Figure 4.5 illustrates the sequence diagram of the technical implementation of this process. Again, the entire registration process is composed of 4 blockchain transactions that have to be executed individually.

First of all, the RO has to use the Smart Energy Coin contract again in order to authorize the Escrow Registry to transfer the 10 SEC resource registration fee from their blockchain account. As a response, the RO receives the hash of this blockchain transaction.

In the second step, the RO now performs the actual depositing of the resource registration fee. As shown in step 2, the RO uses the function `depositResourceRegistrationFees()` of the Escrow Registry smart contract and specifies the computed pseudonymous hash identifier of the DER (see Eq. 3.5). The Escrow Registry component then validates the request and uses the Smart Energy Coin contract to transfer the just approved 10 SEC resource registration fee from the RO's account to itself. Afterward, it stores that the invoking RO has deposited the resource registration fee for the given DER.[2] Subsequently, the RO receives the transaction hash of this operation in step 2.7.

Next, the RO uses the Registry function `registerResource()` to perform the registration and to store information about the DER in the system. As described in Chapter 3.1.1, the RO must provide the resource type, the pseudonymous hash identifier of the resource, and encrypted verification data (see Eq. 3.5). Subsequently, the Registry validates the input and checks if the DER's hash identifier is not already used in the system. Then it invokes the function `isResourceRegistrationFeeDeposited()` of the Escrow Registry smart contract to guarantee that the RO has indeed deposited the 10 SEC resource registration fee for the specified DER. If this is not the case, the transaction is reverted,

2 Here, the RO is identified by their blockchain account address.

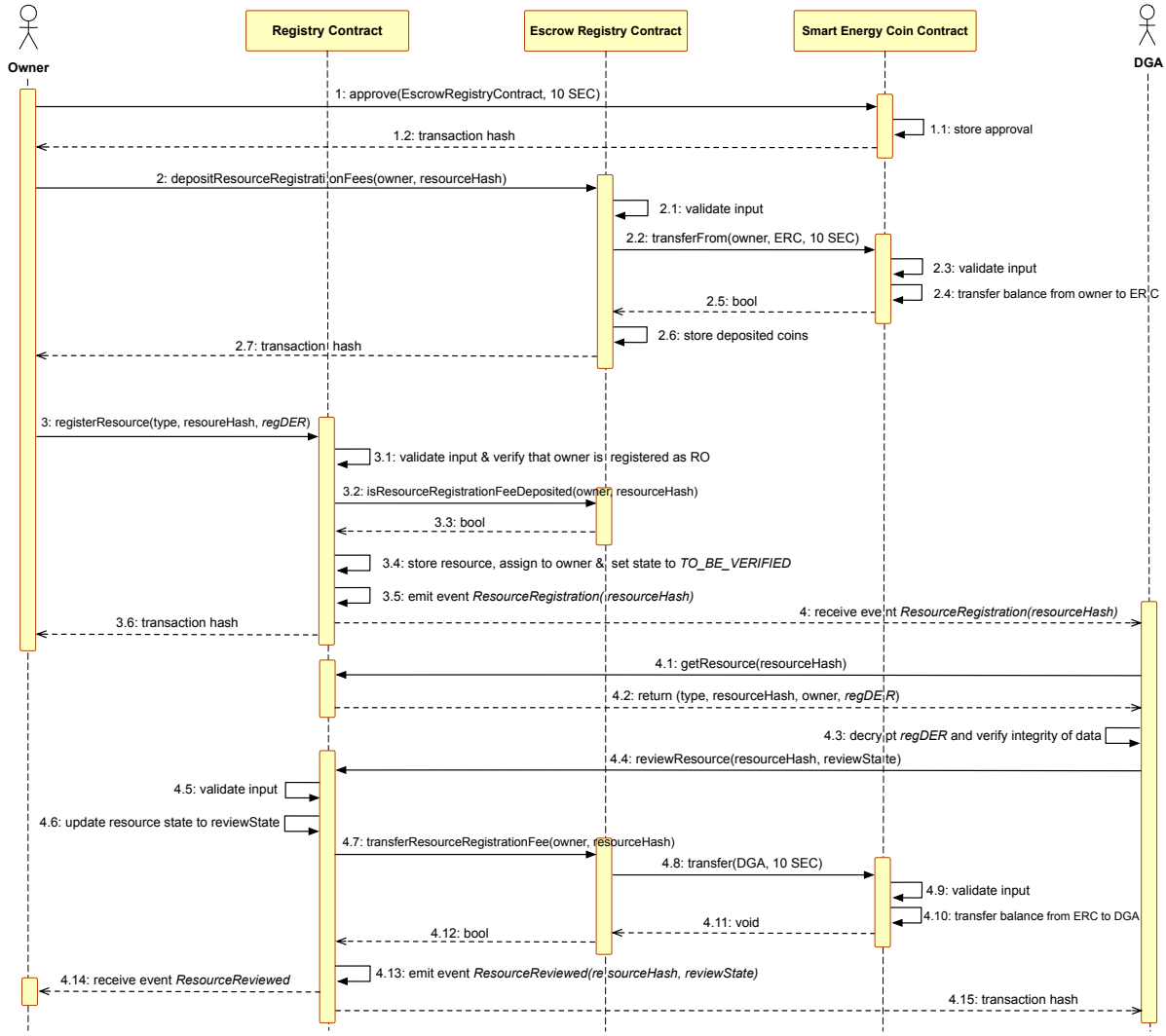


Figure 4.5: Sequence diagram illustrating the process of registering a DER.

and no resource information is stored. Otherwise, the Registry creates a resource object to store the provided DER data. Similar to the users, the system also maintains a state for each registered DER (see Figure 4.4 (b)). Right after this initialization phase, the DER is in the initial state *TO_BE_VERIFIED*. Although the DER is already stored in this state, the RO cannot yet offer an operating reserve of the DER on the markets. The DGA must first verify that the RO indeed operates the DER and that all information stored is trustworthy. To inform the DGA about the need for a verification, a *ResourceRegistration* event is emitted in step 3.5. Afterward, the RO receives a transaction hash as a response.

In the final step, the DGA performs the mentioned verification in order to conclude the registration process. After notification by the event, it uses the retrieved DER's pseudonymous hash identifier to read the stored registration information from the Registry by invoking the function `getResource()`. Afterward, it verifies this information as we discussed in Chapter 3.1.1. In step 4.4, the DGA then

informs the system about the verification result. To this end, it uses the Registry smart contract function `reviewResource()`. Here, the DGA provides the hash identifier and the new state of DER representing the verification result. If the verification was successful, the DGA provides the state *VERIFIED*. The DER is then unlocked in the system and the RO can offer operating reserve of it. However, if the verification fails, the DGA specifies the state *REJECTED*, whereafter the DER is blocked in the system and none of its operating reserves can be offered. In step 4.7, the Registry immediately informs the Escrow Registry smart contract about the completed verification by the DGA. Thereupon the deposited 10 SEC resource registration fee is transferred from the Escrow Registry to the blockchain account of the DGA. Thus it was compensated for the verification effort. Finally, the Registry contract issues a *ResourceReviewed* event to notify the RO about the verification. The DGA then receives a transaction hash as a response, and the resource registration process terminates.

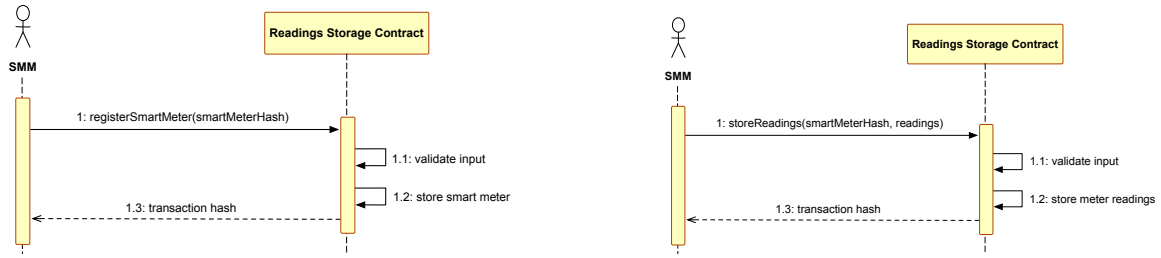
Resource States For every DER, the system also manages a state describing the current status of it. Depending on the respective state, certain operations are allowed or blocked for the DER in the system. Figure 4.4 (b) represents the state machine of the DER states. This state machine almost behaves identically to the already described state machine of the users. The initial state is *TO_BE_VERIFIED*. In this state, the data of the DERs are stored and initiated, but no operating reserve can be offered. First, the DER must be verified by the DGA. During this verification process the DGA can set the DERs either to the state *VERIFIED* or *REJECTED* by invoking the function `reviewResource()`. In the former case, the DERs are successfully verified and the ROs are allowed to offer operating reserve of them on the markets. In contrast, in the state *REJECTED* the verification has failed, the DER is blocked in the system, and no operating reserve can be offered. Nevertheless, by executing the registration process again, the ROs can update the DERs' stored data and request a new verification. In this context, the state of the DER transitions back into the state *TO_BE_VERIFIED*. The last state is named *REFUNDED*. The DER can get into this state when the RO calls the function `refundResourceRegistrationFee()`. They are allowed to execute this function after 75 mined blocks after DER registration in order to counteract the significantly delayed verification by the DGA. In this case, the ROs abort the registration process. However, by registering the DERs again, they can put them back into the state *TO_BE_VERIFIED*.

4.2.2 Smart Meter Processes

In the following, we want to discuss dynamic aspects related to the registration of smart meters and the management of their generated meter readings. To this end, we describe the two processes highlighted in green in Figure 4.2.

Register Smart Meter

In the Chapter 3.2 we explained that smart meters are assigned to DERs in order to measure data about the energy production and consumption of the DERs. Later, the system will use this information for executing the semi-automated payment process. For this reason, the SMM is authorized to register



(a) Registering Smart Meters

(b) Store Smart Meter Readings

Figure 4.6: Sequence diagrams depicting the process registering smart meters and store readings of them.

smart meters in our system.[3] This process is illustrated as sequence diagram in Figure 4.6 (a). As we can see, the SMM interacts with the Readings Storage smart contract to perform this operation. The SMM issues a blockchain transaction that invokes the `registerSmartMeter()` function of the Readings Storage smart contract. Thereby, it provides the SHA-256 hash of the real-world identifier of the smart meter (see Chapter 3.1.1). Here we use the hash of the identifier to pseudonymize the stored information about the smart meters and thus prevent the measurement data from being traced by unauthorized parties. After invoking, the Readings Storage verifies the input and guarantees that the provided hash is not yet stored in the system. Then it stores this smart meter in the system. Finally, the SMM receives the hash of the blockchain transaction.

With the smart meter now being registered in the system, we can finally store the relevant measurement data of it in the system. In the following, we will describe this process in detail.

Store Smart Meter Readings

Measurement data on the actual generated positive and negative operating reserve of a DER is an essential part of the proposed semi-automated payment process. As described in Chapter 3.2, in our system the SMM is responsible for providing readings of the smart meters assigned to the DERs. Figure 4.6 (b) illustrates the concrete technical implementation of this process.

In order to store production and consumption readings of a smart meter already registered in our system, the SMM uses the Readings Storage smart contract. To do so, it calls the function `storeReadings()` and provides the SHA-256 hash of the smart meter identifier as input. As an additional input parameter, the SMM provides a variable list of meter readings. These readings must be formatted as defined in Equation 3.8. After invoking, the Readings Storage smart contract validates the input and proves that a smart meter with the provided hash is indeed registered in the system. If this is the case, the component also verifies the provided list of meter readings and examines the plausibility of these. If the measurement data is successfully verified, the Readings Storage smart contract stores the readings and assigns them to the smart meter. Since the process described is represented by a blockchain transaction, the SMM finally receives its transaction hash as a response.

Subsequently, as in our system a smart meter is assigned to each DER during the resource registra-

3 In our system, the blockchain account address of the SMM is known and we will use it to identify the SMM during requests.

tion, we can allocate the stored production and consumption data to the DERs and thus carry out the semi-automated payment process.

4.2.3 Trading Processes

The aim of the following is to describe the last category of identified processes in our system. Specifically, we will describe the trading related procedures depicted in blue in Figure 4.2. These are probably the most important ones in our system. They enable the technical implementation of the markets according to the description in Chapter 3.3. All other processes described so far just create the framework for implementing these market processes and can be regarded as prerequisites for the implementation of these.

Place Offer

Figure 4.7 represents the process in which registered and verified ROs can finally offer operating reserve of their registered and verified DERs on the markets. The realization of this process follows the specifications in Chapter 3.3 and is formally represented in Algorithm 1. The process is the same for both market variants intraday and day-ahead. Only the concrete trading smart contract to be used differs. In total, the process consists of three blockchain transactions to be issued.

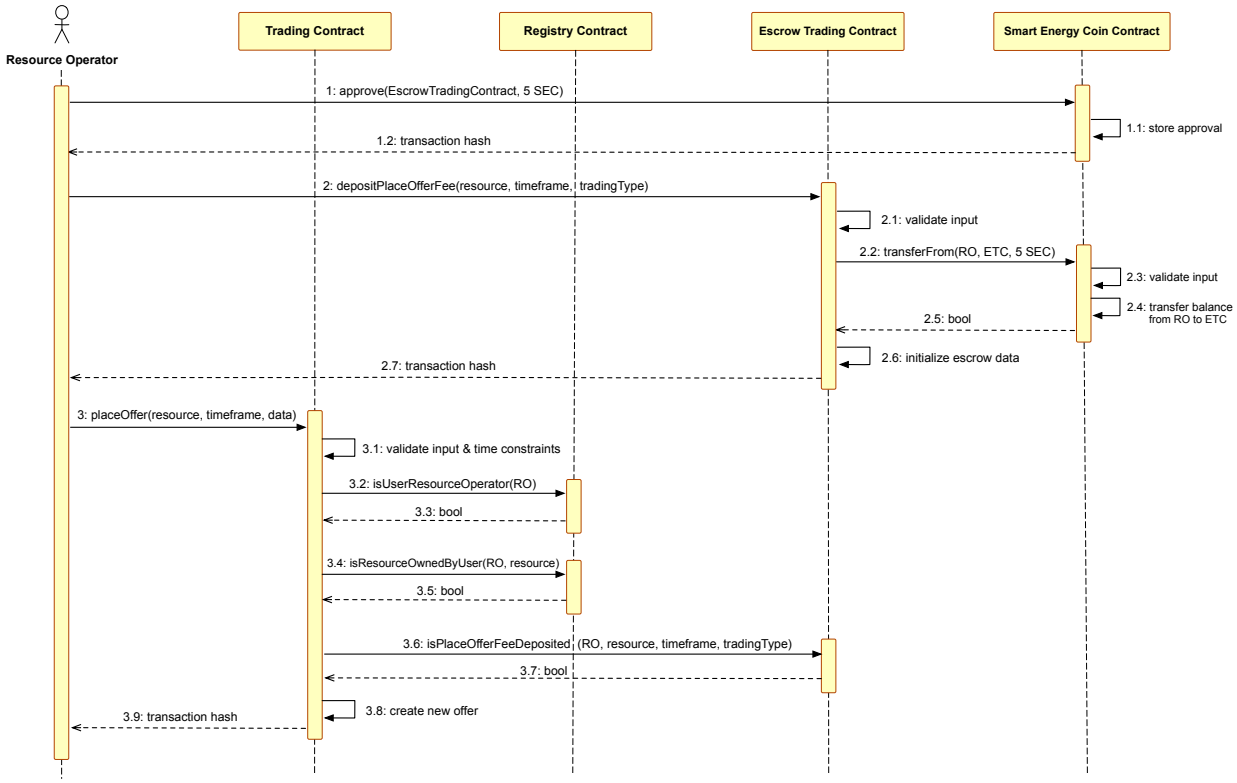


Figure 4.7: Sequence diagram visualizing the process of a RO placing an offer for one of their operated DERs.

In the first step, the ROs must use the Smart Energy Coin. As already described twice in the registration processes, they invoke the function `apporove()` to declare that the Escrow Trading Contract is entitled to transfer the required 5 SEC place offer fee from their blockchain account (see Chapter 3.2). As already mentioned, the Escrow Trading contract implements all trading related escrow mechanisms and later executes the semi-automated payment process. In response to this operation, the ROs receive the transaction hash of this first blockchain transaction.

In the second step, they invoke the function `depositPlaceOfferFee()` of the Escrow Trading smart contract to deposit the required 5 SEC place offer fee. In doing so, the ROs must specify the DER's hash identifier for which they want to offer the available operating reserve. Furthermore, they must specify the trading type to be used. As explained in Chapter 3.3 they can choose between intraday or day-ahead. The last input parameter has to be the trading timeframe for which the offer should be created on the specified market variant. After processing the request, in step 2.2, the Escrow Trading contract uses the function `transferFrom()` of the Smart Energy Coin to transfer the approved 5 SEC place offer fee on behalf of the RO to itself. If this is successful, the Escrow Trading smart contract initiates a data structure representing the trading escrow for this offer (step 2.7). In this escrow data structure, all deposited place and accept offer fees, as well as the trading volume funds to be deposited by the GOs, are stored. Later on, even the metering-data-based allocation of the operating reserve to the GOs is stored here, which is a prerequisite for the payment process. The initiated escrow data structure is identified by the hash identifier of DER, the used trading variant (intraday or day-ahead), and the selected trading timeframe. For each escrow instance, the system manages a dedicated state. After the initialization in this process, the escrow is in the state *OWNER_DEPOSITED* (see Figure 4.11 (a)). At this point, we do not want to elaborate on the meaning of this state, since we will discuss all escrow states and there meanings in Chapter 4.2.3. Finally, the ROs receive a blockchain transaction hash in response.

In the final step, the ROs submit the offer to finally advertise the available operating reserve on the markets. Depending on the trading type used in the second step, they use either the Intraday or Day-Ahead smart contract. Since the procedure is the same for both market variants, we will abstract from these and call the contract invoked henceforth Trading smart contract. In order to place an offer, the ROs invoke the function `placeOffer()` of the Trading smart contract. In this request, they specify the pseudonymous hash identifier of the DER and the trading timeframe during which the offer is valid (see Chapter 3.3). Furthermore, they provide information about the positive and negative operating reserve to be offered. Depending on the concrete trading variant used, this input must be either formatted as defined in Equation 3.10 or 3.16. Next, the Trading contract validates the input, reviews it for plausibility, and guarantees that no offer matching this DER, trading timeframe, and trading variant has already been placed. In this context, we want to add that in our system, an offer of a trading variant is uniquely identified by the hash identifier of the DER and the trading timeframe. If the input is valid, the Trading contract additionally guarantees that the time constraints for the place offer operation shown in Table 3.1 are satisfied. If this is also the case, the Trading contract calls the function `isUserResourceOperator()` of the Registry in step 3.2 to check whether the invoking user is indeed registered and verified as RO. Thereby the user is identified by the used blockchain account address. Afterward, in step 3.4, the Trading contract utilizes the Registry again. It executes the function `isResourceOwnedByUser()` to ensure that the DER represented by the provided hash

identifier is registered, verified, and assigned to the invoking RO. Subsequently, the Trading Contract contacts the Escrow Trading contract in step 3.6 to check whether the RO has deposited the expected 5 SEC place offer fee for the specified DER, trading timeframe, and market variant. If this applies, the Trading contract creates a new offer object for the chosen market variant. In step 3.8, this offer is then stored and assigned to the RO. Finally, the ROs receive a transaction hash as a response. Henceforth, the GOs can fetch and review the offer.

For the sake of completeness, we want to mention that in order to withdraw the deposited place offer fee, the RO can use the function `withdrawPlaceOfferFee()` of the Escrow Trading contract. This is useful if the RO decides not to place an offer on the markets for the represented trading timeframe.

Edit Offer

After explaining how a RO can place an offer, we will now describe how they can edit or withdraw it. This process is illustrated as a sequence diagram in Figure 4.8. The execution of the process is only possible if the deadline for editing the offer as shown in Table 3.1 is not yet expired. To perform this process, the RO initiates a blockchain transaction addressed to one of the two existing Trading contracts. Depending on the market variant the offer was placed for, the RO uses either the Intraday or Day-Ahead smart contract. The issued blockchain transaction represents the invocation of the function `editOffer()` of the respective Trading contract. In this request, the RO identifies the offer to be edited by specifying the DER's hash identifier and the trading timeframe. Furthermore, they

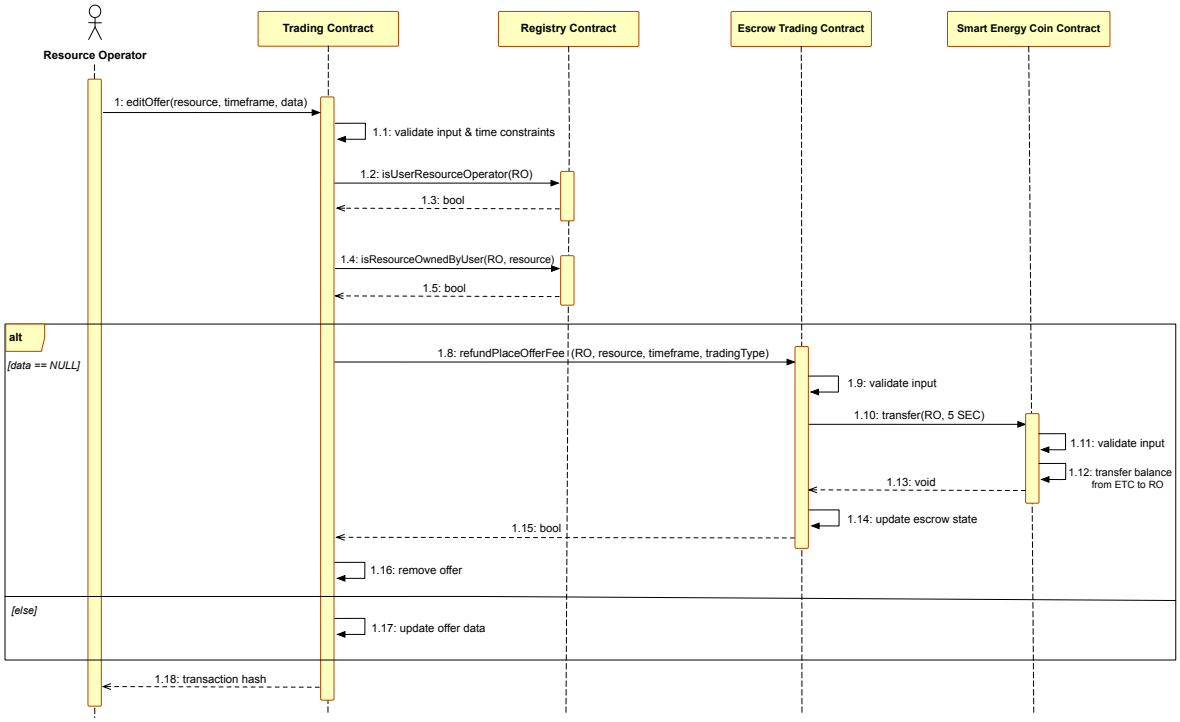


Figure 4.8: Sequence diagram depicting the process of editing or removing an already placed offer.

provide the new operating reserve to be stored as input. Similar to the process of placing offers and depending on the concrete market variant, the operating reserve must be either formatted as defined in Equation 3.10 or 3.16. After the RO invokes the Trading contract, the latter first checks whether an offer for the specified DER and trading timeframe exists. Then, it checks if the deadline for editing offers as described in Table 3.1 is still met. Next, the Trading contract uses the Registry component and executes the functions `isUserResourceOperator()` (step 1.2) and `isResourceOwnedByUser()` (step 1.4). In these steps, the Trading contract ensures that the invoking user is indeed a verified RO and assigned to the DER. Depending on the provided operating reserve to update, there are two options to proceed:

If the provided operating reserve input indicates that the RO does not want to offer operating reserve for this DER, trading timeframe, and market variant anymore, the offer will be removed. In order to indicate this, the RO must still provide the operating reserve input as defined in Equation 3.10 or 3.16. However, both the series F^+ and F^- must be filled with zeros in this case. Through this mechanism, the RO informs the system that they want to withdraw and remove the offer. As described in step 1.8, the Trading contract first instructs the Escrow Trading contract to refund the deposited place offer fee. Specifically, it calls the function `refundPlaceOfferFee()`. To describe the concrete escrow unambiguously, it provides the RO, the hash identifier of the DER, the trading timeframe, and the trading type as input. After processing the request, the Escrow Trading contract transfers the 5 SEC place offer fee to the blockchain account of the RO in step 1.10. Afterward, the RO records the refund of the fees in the stored escrow data structure and modifies the escrow state to *REFUNDED* (see Figure 4.11 (a)). To understand the meaning of this state, we refer to Chapter 4.2.3. Finally, the Trading contract removes the offer and the RO receives a transaction hash as a response.

However, if the operating reserve provided in the input is not filled with zeros, the Trading contract only updates the stored data. As shown in step 1.17, the Trading contract first filters the concrete offer and then just updates the operating reserve. Finally, the RO receives a transaction hash as a response.

Accept Offer

In our system, the GOs can now retrieve and analyze the placed offers of the ROs from the Trading contracts. If they intend to employ the offered operating reserve to stabilize their grids, they can purchase shares of it in the system. This process is called *accept offer* and is shown in Figure 4.9 as a sequence diagram. The process is formally represented in Algorithm 2 and implemented as described in Chapter 3.3. The whole process consists of three blockchain transactions.

As described in Chapter 3.2, the GOs must first deposit an accept offer fee in the offer's dedicated escrow mechanism. Furthermore, they must also deposit the costs of the operating reserve to be purchased as a pre-payment and security in the escrow mechanism. Later, we will use this to pay the ROs for the delivered service. We want to emphasize that as described in Chapter 3.3, the GOs do not have to purchase the entire operating reserve offered. Instead they can only acquire fragments of it, which we will call operating reserve shares. As already explained, the GOs must first approve in the Smart Energy Coin contract that the Escrow Trading Contract is entitled to transfer the fees and costs described above on their behalf. They can either calculate the operating reserve share costs

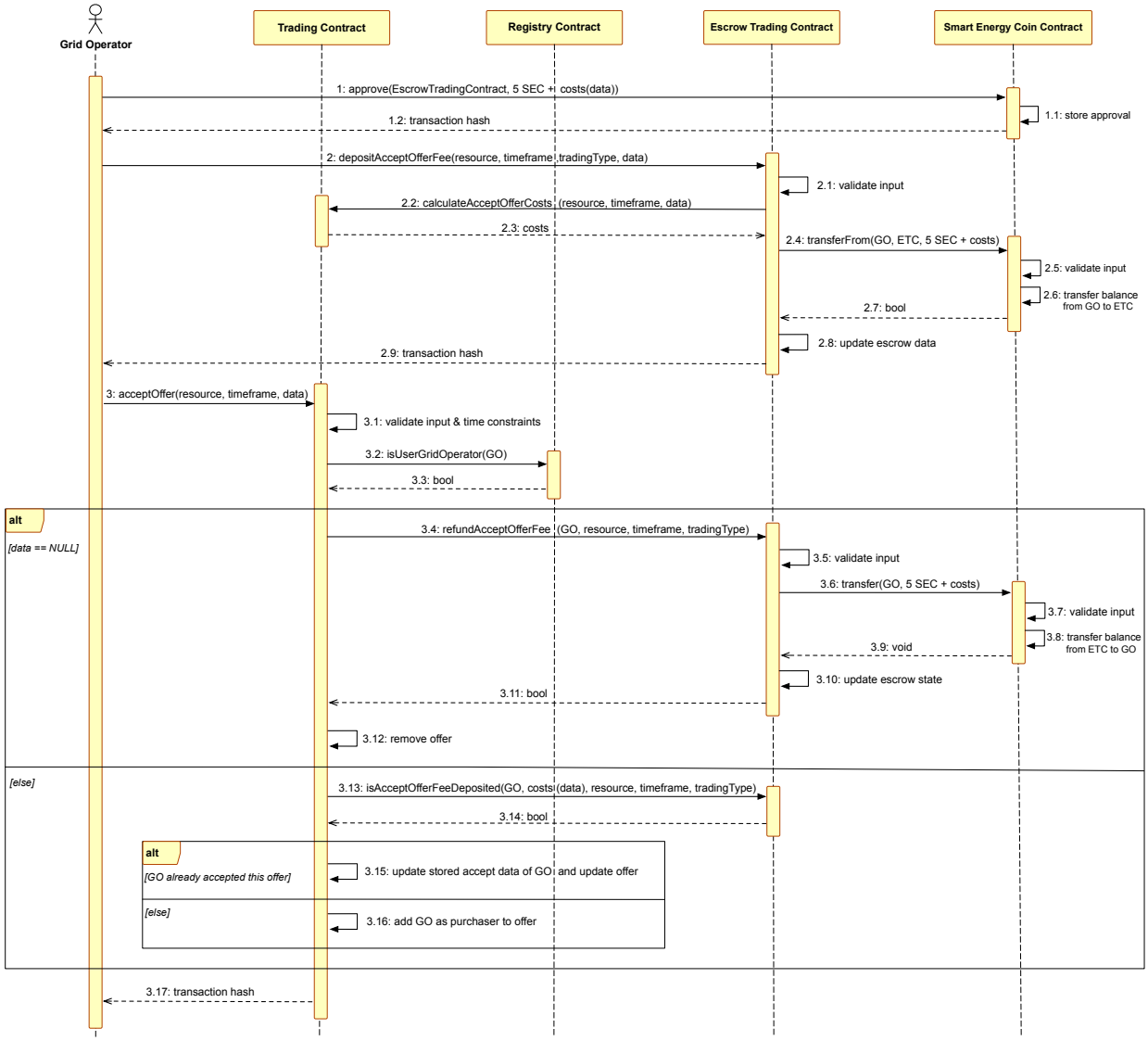


Figure 4.9: Sequence diagram visualizing the process of accepting an offer.

themselves or use the function `calculateAcceptOfferCosts()` of the Trading contracts. After executing the `approve()` function, the GOs receive a blockchain transaction hash as a response.

In the second step, the GOs invoke the function `depositAcceptOfferFee()` of the Escrow Trading contract to deposit the described fees and costs effectively. To do so, they must provide the identifying information of an offer as input. This includes the pseudonymous hash identifier of the DER, the trading timeframe, and the trading type. As further input, they have to specify the shares they want to purchase from the offered positive and negative operating reserve. Depending on the trading type used, they must format the share series as defined in Equation 3.13 or 3.17. Subsequently, after validating the input, the Escrow Trading contract calculates the costs of the specified shares to be acquired. To do so, it calls the function `calculateAcceptOfferCosts()` of the Trading contract and provides the shares and the meta-information required to identify the offer. Since the offer and the

concrete prices for each interval are stored in the Trading contract, the latter can now calculate the costs of the given shares as described in Equation 3.18. Afterward, the Escrow Trading contract uses the Smart Energy Coin and transfers the sum of 5 SEC accept offer fees and the calculated costs from the blockchain account of the GO to itself. If this fails, for example because the GO has approved an insufficient amount of coins due to a calculation error in the first step , the transaction is reverted and the process is terminated. However, if the operation was successful, the Escrow Trading contract stores this in the escrow instance assigned to the offer. For this purpose, the smart contract initiates a specially designed data structure in which only the escrow data of this GO is managed for this specific offer. This object is then stored in the escrow instance of the offer and assigned to the GO. Each of these special objects also contains a corresponding state. All state configurations are shown in Figure 4.11 (b) and are described in detail in Chapter 4.2.3. The initial state after this operation is *DEPOSITED*. Furthermore, since the GO's accept offer data has been stored in the escrow instance, its state is also updated to *GRID_OPERATORS_DEPOSITED*. Finally, the GO receives a transaction hash in response.

In the final third step, the actual purchase of the shares is carried out. To this end, the GO calls the function `acceptOffer()` of the Trading smart contract. As already mentioned, it depends on the concrete market variant used whether they have to invoke the Intraday or Day-Ahead smart contract. However, the procedure is almost identical for both variants. In order to identify the offer, the GO gives the DER's hash identifier and the trading timeframe as input to the request. Besides, the GO identifies the shares of the operating reserve that they want to purchase. Here, the exact same data as in step 2 must be used. Afterward, the Trading contract checks if the offer exists and if the time requirements for accepting an offer as shown in Table 3.1 are fulfilled. If this is the case, the Trading contract calls the function `isUserGridOperator()` of the Registry and checks whether the invoking user is indeed a verified GO. Then, depending on the provided operating reserve share input, there are two options to proceed:

If the operating reserve shares given in the input are only filled with zeros, the system assumes that the GO had already accepted the offer and now wants to withdraw it. Trivially, the Trading contract first checks whether the GO has already accepted the offer and shares are stored. If this is not the case, the blockchain transaction is reverted and the process is aborted. Otherwise, the Trading contract can withdraw the purchase. As depicted in step 3.4, it first instructs the Escrow Trading to refund the deposited fees and operating reserve costs. To do so, it calls the function `refundAcceptOfferFee()` and specifies the GO, the DER hash identifier, the trading timeframe, and the trading type as input. The Escrow Trading contract then transfers all funds deposited by the GO back to their blockchain account. Since the object created in step 2 manages the GO's accept offer data in the escrow, the refund is recorded here, and its state is changed to `REFUNDED`. Finally, the Trading contract removes the deposited shares of the GO in the offer object and the GO receives the transaction hash as a response. Henceforth, the released operating reserve is again available for other GOs.

In the alternative scenario, the shares of operating reserve provided by the GO were not exclusively filled with zeros. In this case, the system assumes that the GO wants to purchase the provided shares. As described in step 3.13, the Trading contract first checks whether the GO has deposited the accept offer fees and the costs of the provided shares. To do so, it invokes the function

`isAcceptOfferFeeDeposited()` of the Escrow Trading contract and provides the GO, the offer, and the minimum expected deposited costs as input. If the GO has deposited the accept offer fees and the specified costs in the Escrow Trading contract, it responds with a positive answer. Subsequently, the Trading contract checks whether there is still enough operating reserve left in the offer and whether the GO's shares can be satisfied.[4] If this is not the case, the transaction is reverted. Otherwise, again two alternatives how the Trading contract continues exist. If the invoking GO had already purchased shares of this offer, the Trading contract just updates the stored shares by overwriting these with the provided ones.[5] Otherwise, if the GO calls the `acceptOffer()` for the first time, no shares have been deposited yet, and the system assumes the GO is purchasing operating reserve of this offer for the first time. In this case, the Trading contract simply stores the provided shares in the offer. Henceforth, the Trading contract blocks the operating reserve shares for other GOs. Finally, the GO receives a transaction hash as a response.

For the sake of completeness, we want to state that a GO can withdraw the accept offer fees and costs deposited in the escrow if they have not yet accepted the offer in the Trading contract. For this purpose, the GOs can use the function `withdrawAcceptOfferFee()` of the Escrow Trading contract.

Parallel Requests and Conflicting States Furthermore, we want to highlight a special characteristic of our system due to the usage of the blockchain technology. In the process just described, an operation is executed in which several users can change and manipulate the status of a shared object (the offer) simultaneously. Since our system is a distributed system, we would typically have to make sure that no conflicts occur and that our system reaches a consensus at some point. In the case of parallel accept offer requests, for example, we would have to determine which purchase operation of which GO should be executed first. As a result, the acquired operating reserve would no longer be available for the other parallel requesting GOs, which might invalidate their previously valid requests. This is where the advantages of the blockchain technology come in very handy. By using the blockchain technology, we can consider this problem as not given or already solved in our system. Per design, a blockchain is a practical approach to find a consensus in a distributed system. The consensus mechanism used in a blockchain automatically guarantees that a consistent status is established and conflicting states are resolved in the system. Although transactions are processed in a parallel and decentralized manner in the underlying peer-to-peer network, the network automatically agrees on the chronological order of the execution of the transactions and therefore automatically resolves conflicts [6].

Clear Escrow

Using the processes described above, ROs and GOs can already offer and purchase operating reserve at the markets. We have also described how metering data about the actual produced and consumed energy of a DER can be stored and assigned in our system. Based on the trading and metering data now recorded in the system, we can finally clear the escrow and distribute the deposited funds. By doing so, we realize and perform the often mentioned semi-automated payment process. This

4 It is evident that operating shares already assigned to GOs are blocked for others

5 In this case, when checking if there is enough operating reserve left in the offer to satisfy the GOs requested shares, the old and still stored shares of the GO are not considered and seen as not existing.

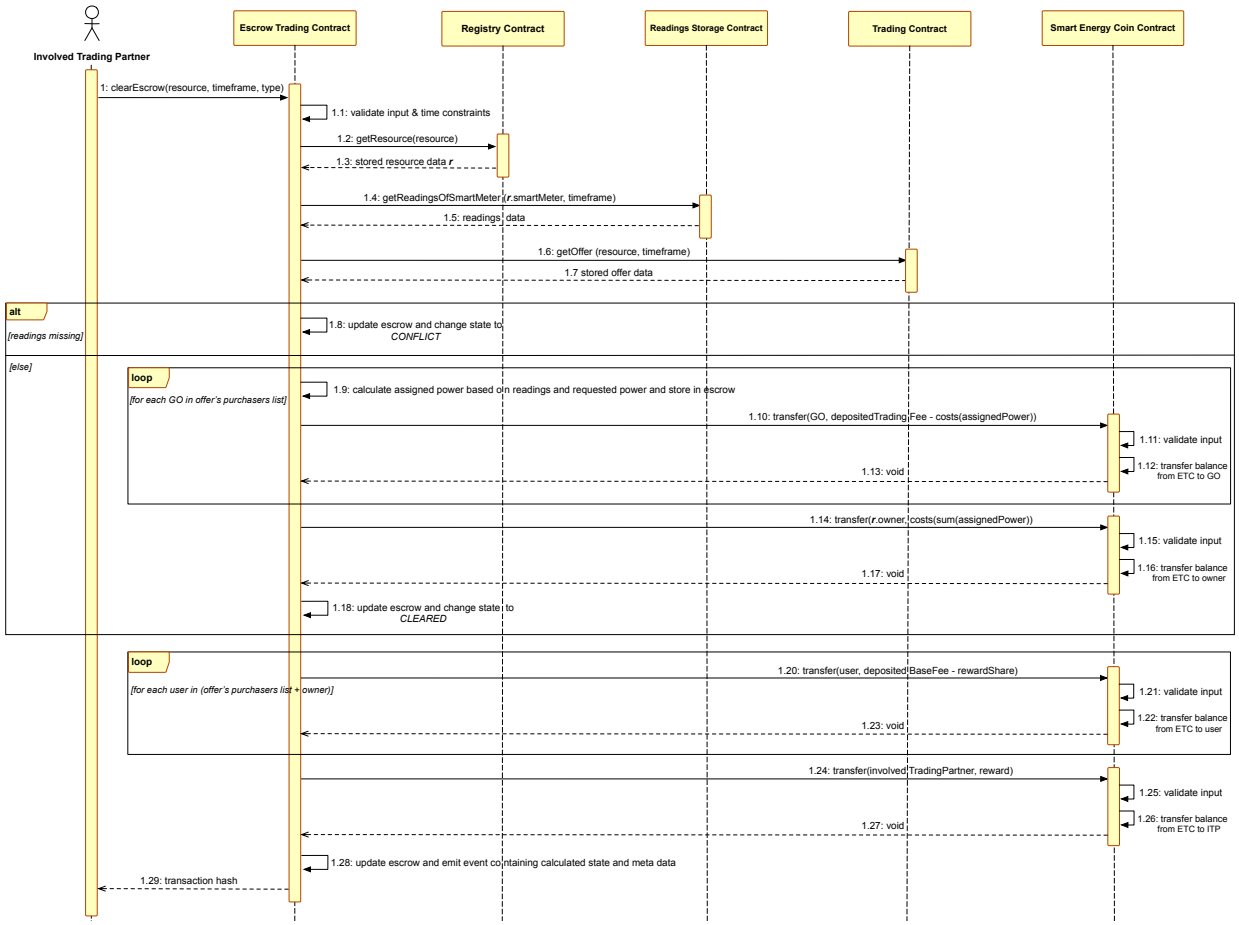


Figure 4.10: Sequence diagram depicting the process of clearing an escrow and executing the semi-automated payment process.

process is shown in Figure 4.10 as a sequence diagram. The implementation of the process follows the descriptions in Chapter 3.2 and is outlined in Algorithm 3. To initiate the payment process, each trading partner involved in an offer and the DGA are authorized to initiate the clear escrow process. Since this process is realized by a blockchain transaction, the system cannot execute it automatically and autonomously. Therefore, since a one-time involvement of a user is necessary, we describe the payment process as semi-automated.

As shown in Figure 4.10, in order to initiate the payment process and clear the escrow, a involved trading partner invokes the function `clearEscrow()` of the Escrow Trading smart contract. In this context, the trading partner provides the hash identifier of the DER, the trading timeframe, and the trading type to identify the offer and the escrow instance. Subsequently, the Escrow Trading contract validates the input and ensures that the escrow exists at all. Furthermore, the Escrow Trading contract first checks whether the time constraints for clearing the escrow shown in Table 3.1 are met. The constraints are supposed to ensure that the SMM has enough time to store the required measurement data in the system. The contract then uses the Registry to read the stored data of the DER. To this end, it calls the function `getResource()`. The stored DER information is required because it contains

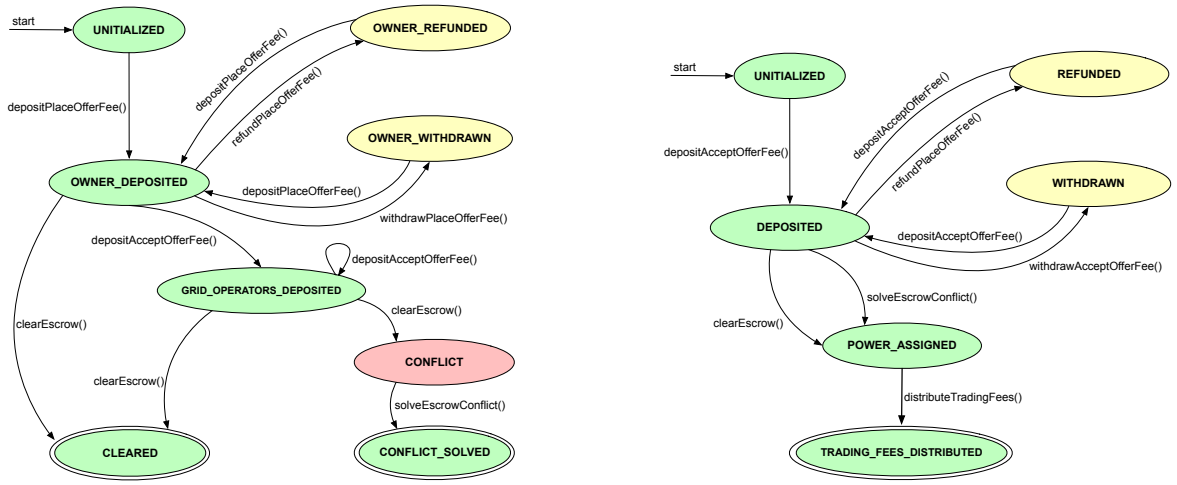
the smart meter assigned to DER. In step 1.4, the Escrow Trading then uses this extracted smart meter to execute the function `getReadingsOfSmartMeter()` of the Readings Storage smart contract in order to fetch the metering data of the DER. In this context, it also specifies the time interval of which it requires metering data in order to clear the escrow. In the last preparatory step, the Escrow Trading contract calls the function `getOffer()` of the Trading contract to extract the offer's stored information. This information contains details about the offered operating reserve and the expected prices, which are essential for the payment process to be executed. Now there are two different possible alternatives to continue:

If not all required measurement data for the specified timeframe are available and readings are missing for some reason, the escrow cannot be cleared. In step 1.8, the Escrow Trading then records this in the escrow instance and updates its state to *CONFLICT*. Additionally, a *EscrowConflictDetected* event is issued to notify the DGA about the conflict and the request solve it (see Chapter 4.2.3).

However, if all measurement and trading data is available, the escrow can be cleared and the payment process executed. For this purpose, the Escrow Trading contract first calculates how much operating reserve has been provided by the DER and then allocates it to the individual GOs. As described in Chapter 3.2, this operation follows the *first-come-first-serve* principle. In the first place, the provable measured operating reserve is used to satisfy the demands of the GO that acquired shares first. The remaining operating reserve is then used to satisfy the second GO's demands, and so on. Since there can even exist simultaneous intraday and day-ahead offers of a DER on the markets, the already allocated operating reserve of such offers is deducted from the available operating reserve if their respective escrows are already marked as cleared. This is trivial since operating reserve cannot be allocated twice. Subsequently, the Escrow Trading stores the assigned operating reserve of the GOs in the escrow instance. For this purpose, it uses the initialized objects that describe the accept offer data of each GO in the escrow. Here, it adds the allocated operating reserve as a series to it and changes its status to *POWER_ASSIGNED* (see Figure 4.11 (b)). Then, the Escrow Trading contract calculates the costs for the allocated operating reserve as defined in Equation 3.18. Next, it uses the deposited funds of the GO to transfer the costs to the RO's blockchain account in order to pay them for the service provided. The transferred funds are then deducted from the deposited funds in the escrow and the remainder is refunded to the GO. Next, the status of the GO's accept offer data instance in the escrow is updated to `TRADING_FEES_DISTRIBUTED`. In this alternative, the state of the escrow instance is finally modified to *CLEARED*.

As described in Chapter 3.2, no matter which of the two alternatives is executed above, the invoking user gets rewarded with a reward of 1 SEC for initiating the payment process. This reward is intended to encourage users to initiate the payment process as soon as possible and prevent orphaned funds and fees. For this purpose, the place and accept offer fees of all users stored in the escrow are used. The reward is distributed equally among all users and deducted from their deposited fees uniformly. The remaining amount of fees will be refunded to the users in step 1.20. In step 1.24, the reward of 1 SEC will be transferred to the invoking trading partner's blockchain account. For transparency reasons, in step 1.28, the calculated status (*CONFLICT* or *CLEARED*) and other metainformation is updated in the escrow instance. Additionally, a `EscrowCleared` event is emitted to inform all trading partners about the progress.

Finally, the user receives a transaction hash as a response.



(a) State Machine for Escrow States (b) State Machine for Escrow Accept Offer Data

Figure 4.11: State machine for the esrow states and escrow accept offer data states maintained by the system. The state transitions are performed by executing the respective the Escrow Trading smart contract functions.

Escrow States As already mentioned, the system manages a state for each escrow instance created in the Escrow Trading smart contract. The corresponding state machine is shown in Figure 4.11 (a). Furthermore, we have already mentioned that each escrow instance also contains a list of objects that manage the escrow related accept offer data of a GO. Even for each of these objects, a state is managed. The related state machine is shown in Figure 4.11 (b). In both cases, the states are used to describe the current state of the escrow and the data contained. This is intended to simplify the implementation and increase transparency. Furthermore, the system can derive from each state which operations are still allowed or required for an escrow.

After a RO has deposited the place offer fee for an offer in the Escrow Trading contract, an escrow instance is created and stored in the initial state *OWNER_DEPOSITED*. This state indicates that the initial fees have been deposited, but the offer has not yet been accepted by GOs. In this state, the RO is allowed to withdraw the deposited fees under certain circumstances. If the RO has still not placed the offer, it can use the function `withdrawPlaceOfferFee()` to withdraw the fees. Then the state of the escrow changes to *OWNER_WITHDRAWN*. On the other hand, the system can also initiate a refund of the place offer fees in the state *OWNER_DEPOSITED*. This can be required if the RO decides to remove the offer (see Chapter 4.2.3). In this case, the state changes to *OWNER_REFUNDED* by invoking the function `refundPlaceOfferFee()`. In these two states, no more funds are stored in the escrow and the offer does not exist in the system. In both cases, the escrow can change back to the state *OWNER_DEPOSITED* when the RO deposits the place offer fees again. If a trading partner invokes the function `clearEscrow()` in the state *OWNER_DEPOSITED*, the system concludes that the offer was not accepted. In this case, no operating reserve needs to be allocated in the payment process and the place offer fee minus the reward share is refunded to the RO. The escrow then changes to the final state *CLEARED*. If a GO deposits the accept offer fee by calling the function `depositAcceptOfferFee()` in the state *OWNER_DEPOSITED*, the escrow

immediately transitions to the state *GRID_OPERATORS_DEPOSITED*. In this state, the Escrow Trading contract is aware of the facts that the offer has probably been accepted, the operating reserve costs have been deposited, and that the payment process must be executed. Afterward, the escrow state can only be changed by invoking the function `clearEscrow()`. After executing this function, the payment process is performed. If it succeeds, the escrow changes to the final state *CLEARED*. In this state, no more operations on the escrow instance are allowed, and all fees and funds are already distributed. If the clearing process was not successful, the escrow changes to the state *CONFLICT*. This state indicates the presence of a conflict. In this state, only the DGA can invoke the function `solveEscrowConflict()` to put the escrow into the final state `CONFLICT_SOLVED`. In this state, the escrow is considered resolved and all fees and funds are distributed.

Figure 4.11 (b) illustrates all states of the objects that describe the accept offer related data of the GOs within the escrow. These objects are stored in a list in the escrow instance and assigned to GOs. Since they are only used as an internal tool for the administration of the escrow and its operation and do not contribute to the understanding of the system and its processes, we will not describe them in detail here. However, only for the sake of completeness, we have presented them here. For interested readers, we refer to the source code in Appendix 1 for more details.

Solve Escrow Conflict

As described in the process above, it can appear that due to missing metering data, an escrow cannot be cleared, and the payment process cannot be executed directly. In this case, the DGA is responsible for resolving the conflict in our system. This process is shown in Figure 4.12 and implemented as described in Chapter 3.2. We assume that in case of a conflict, the DGA contacts the involved GOs and RO to determine how much operating reserve was effectively generated and consumed by which GOs to what extend. The DGA then compiles the gathered information and submits it to our system. Using this information about the allocated operating reserve for the GOs, the payment process can finally be executed.

To perform the described process, the DGA invokes the function `solveEscrowConflict()` of the Escrow Trading smart contract. Thereby, it specifies the DER's hash identifier, the trading timeframe, and the trading type to identify the offer. Furthermore, as described above, it provides the allocation of the operating reserve to the GOs. For each GO involved, it must specify a series of assigned positive and negative operating reserve. The Escrow Trading contract then validates the input and confirms that the escrow exists and is in a conflicting state. Subsequently, it uses the Registry to execute the function `getOffer()`. Here, the Escrow Trading contract reads the stored information of the offer to get the defined prices of the operating reserve. These are required to calculate the cost of the allocated operating reserve later. Afterward, in step 1.4, for each GO included in the offer, the Escrow Trading stores the operating reserve allocated by the DGA in the escrow. In this context, the Escrow Trading will ensure that the assigned operating reserve does not exceed the operating reserve demanded by the GO. Next, the smart contract calculates the costs of the allocated operating reserve as defined in Equation 3.18. It then transfers these costs to the RO's blockchain account and deducts the amount from the GO's deposited funds. Subsequently, the remaining amount of funds is refunded to the GO. In step 1.13, the Escrow Trading contract updates the status of the escrow to

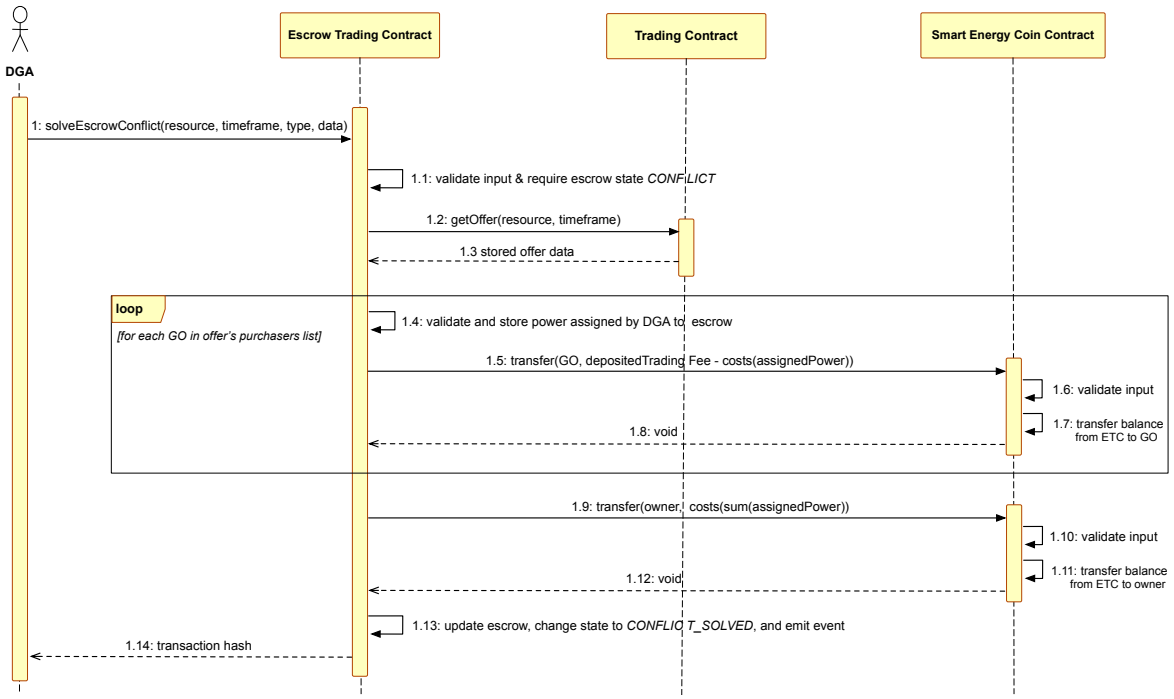


Figure 4.12: Sequence diagram depicting the process of solving an escrow in conflicting state.

CONFLICT_SOLVED and emits a *EscrowConflictSolved* event to inform all trading partners about the procedure. Finally, the DGA receives a blockchain transaction hash as a response.

In conclusion, we have presented the proposed architecture of our system in this chapter. We have explained all actors and system components in detail. Furthermore, we have defined the relevant dynamic aspects of the system and presented them as processes. In this context, we have described which actors interact with which system components to realize these. Summarized, our system is capable of realizing the markets as described in Chapter 3.

In addition, we have found that our system also benefits from the advantages of the blockchain technology. By using a blockchain, we can state that our system is tamper-proof, fault-tolerant, and overall more secure. Furthermore, a consensus is automatically established in our distributed system through the blockchain. Thus we do not have to consider potential conflicts in the parallel execution of the described processes, rendering the system easier to implement.

5 Implementation

In the course of this thesis, we developed a prototypical implementation of the proposed system. In the following chapter, we will describe the realization of this prototype using the Ethereum blockchain. As a proof of concept, this provided implementation aims to realize the system specifications defined in the Chapters 3 and 4. Additionally, we will discuss the smart contract development process and the lessons learned while using the Ethereum technology.

5.1 Protoype Components and Used Technologies

Figure 5.1 provides an overview of the implemented components of the provided prototype. Furthermore, the figure illustrates the underlying technologies used to implement each of the components. As an extension, Figure 5.2 illustrates the used technology stack of our developed application. Moreover, Table 5.1 provides the precise technologies and there corresponding versions used.

As we can see on the right side in Figure 5.1, the yellow-highlighted main components of our proposed system - the smart contracts - are provided on a *Ganache* [61] blockchain during development. Ganache is a local running private Ethereum blockchain that is used as a tool for developing distributed applications for this platform. It can be understood as a test environment in which the development, deployment, and testing of smart contracts can be performed locally and deterministically. Specifically, we use the *Ganache command line interface (CLI)* in version 6.9.1 and configured it to employ the Ethereum *Muir Glacier* fork [62]. The provided smart contracts are developed in the

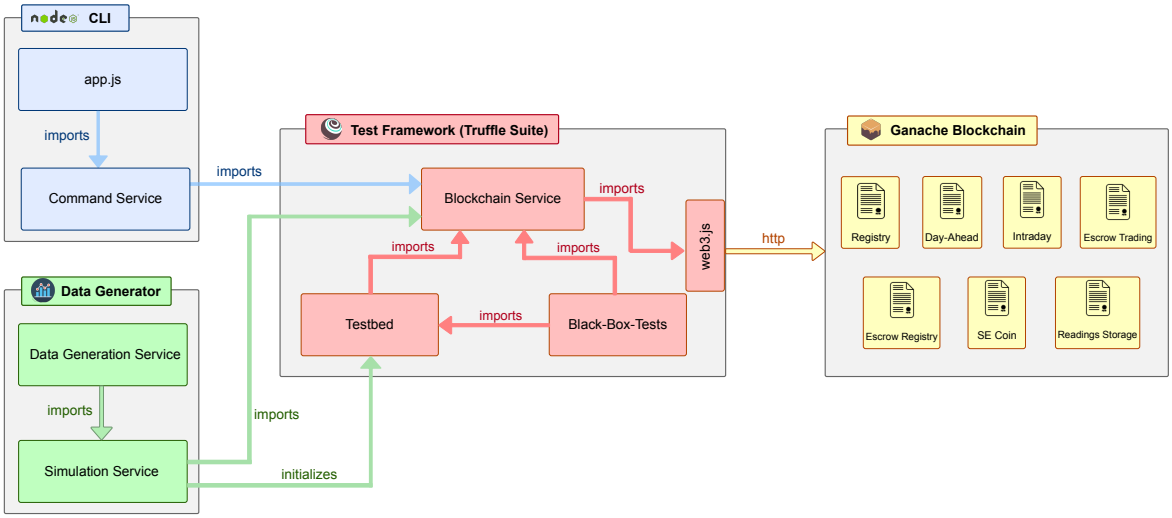


Figure 5.1: Overview of implemented components of the provided prototype.

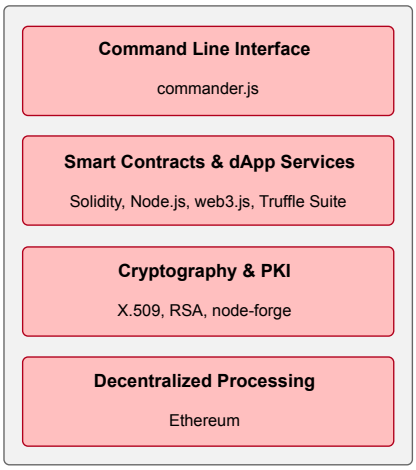


Figure 5.2: Technology stack used to realize the implementation of the provided prototype.

Solidity [22] programming language and compiled to native EVM bytecode using the compiler *solc* [63]. The Ganache application then executes this assembled bytecode. We will discuss precise details about the development of the smart contracts in Chapter 5.2. In Chapter 2.1.1 we explained that several blockchain implementations exist that realize the concept of general-purpose smart contracts. Therefore, all of these alternatives could potentially be used to implement the specifications of the system defined in Chapter 3 and 4. However, by using Ganache it is evident that we have decided to use the Ethereum blockchain. We have not identified any criteria that argue against the use of the other implementations mentioned. Instead, we decided to use the Ethereum blockchain because it seems to have the most significant traction, the first-mover advantage with many developers, and a significantly higher amount of libraries that facilitate the development process. Thus, without much effort and additional infrastructure, it is possible to rapidly access the Ethereum network, provide smart contracts in it, and finally utilize them to implement our proposed markets. Furthermore, we hope that the widely-used Ethereum tools and the superior community support will also benefit our end users in their use of the system.

For the purpose of deploying the developed smart contract on the blockchain and interacting with them we use the *Truffle Suite* [64]. All components highlighted in red in Figure 5.1 are implemented by utilizing this suite. The Truffle Suite is one of the most widely used development frameworks for the implementation of Ethereum-based distributed applications. Truffle provides components that make it easy to develop, compile, test, and finally deploy Solidity-based smart contracts. Thereby, the Truffle Suite uses the *web3.js* library [65] to connect and interact with an Ethereum client using the hypertext transfer protocol (HTTP). The web3.js library provides functions to create and sign Ethereum transactions, as well as to broadcast them to the connected Ethereum network. Furthermore, web3.js can be used to read information from the blockchain. Therefore, we also use web3.js to interact with our smart contracts through a HTTP-based network connection. In the developed prototype, we use version 5.1.25 of the Truffle Suite and version 1.2.1 of the web3.js library.

As shown in Figure 5.1, we provide a Truffle-Suite-based module named *Blockchain Service*. This module serves as an abstraction layer for the developer and user to ease using the provided smart

Technology	Version
Node.js	10.22.0
npm	6.14.6
web3.js	1.2.1
Truffle Suite	5.1.25
Ethereum	Muir Glacier fork
Ganache CLI	6.9.1
Solidity (solc)	0.5.16
node-forge	0.9.1
commander.js	6.1.0

Table 5.1: Overview of used technologies and libraries and their required versions.

contracts. Since smart contract functions can only be executed by an associated blockchain transaction and since it can be quite complex to create them, the Blockchain Service offers a simplified alternative. The provided interface is very similar to the one of the smart contracts and expects the same inputs. However, the Blockchain Service automatically creates blockchain transactions using the input and Ethereum wallet provided by the user. Subsequently, it uses the web3.js library to execute the created transactions in the connected Ethereum network by broadcasting them. Thus, the Blockchain Service enables users to execute smart contract functions without requiring them to understand the complexity of the underlying system. Furthermore, the interface of the Blockchain Service can also be used to read information stored on the blockchain.

As described above, since the manual use of the smart contracts can be very complex and time-consuming, manual testing of the developed smart contract code is inefficient and unsuitable. Furthermore, since smart contracts cannot be altered after deployment, we state that automated testing is a key element for the development of smart contracts. Therefore, we have developed a collection of automated tests that ensure the functionality of the developed smart contracts. To implement the tests, we used the Truffle Suite, which provides a testing framework based on *Mocha* [66] and *Chai* [67]. As illustrated in Figure 5.1, we have developed black-box tests. The developed tests are implemented in JavaScript and utilize the developed Blockchain Service to interact with the smart contracts. The black-box tests verify two things. On the one hand, they guarantee that the interface of the public smart contract functions behaves as defined. On the other hand, the black-box tests verify that the system satisfies all identified functional requirements (see Chapter 3.4). To simplify the development of the automated tests, we have also provided a *Testbed*. The Testbed allows the automatic provision and registration of all actors, users, DERs, and smart meters. Furthermore, trading and measurement data is automatically generated and assigned to trading partners and DERs according to the definition in Chapter 3. The X.509-based PKI and RSA keys required for the registration of users and DERs are also simulated in the Testbed. Thereby, all X.509 certificates and RSA keys[1] are generated using the node-forge [68] library. This library is also used for all decrypting and encrypting porpuses performed in the system. The data and functions now provided by the Testbed can then be used to implement complex black-box tests. Using the Testbed, it is now

1 For completeness, we want to mention that we use RSA keys with a length of 2048 bits.

possible to map a state required for a specific test-case to the smart contracts, manipulate it during the test, and then verify the result afterward. Here, it should be mentioned that the Truffle Suite test framework is based on the *clean-room* principle. This means that after each test case, the data stored in the smart contract is removed, and the state is reset. Therefore, utilizing the provided Testbed to establish a desired state automatically is particularly convenient.

Furthermore, in order to enable an easy and user-friendly usage of our prototype and to demonstrate the functionality of the system, we provide a terminal-based *CLI*. This component is highlighted in blue in Figure 5.1. The CLI is based on the Node.js framework [69] and is implemented using the commander.js [70] library. As shown in the figure, the CLI uses the Blockchain Service to interact with the blockchain-based system. Therefore, the underlying complexity of the communication is transparent for the users of the system.

The software component highlighted in green in Figure 5.1 is called *Data Generator*. This component is considered to be isolated from the developed prototype and does not contribute to the functionality of the proposed system. Instead, this component executes the market simulation used for the evaluation in Chapter 6. We will describe the performed market simulation in detail in Chapter 6.1.1.

If not explicitly mentioned yet, we would like to add that all provided code fragments, as well as the Truffle Suite, are based on the JavaScript-based framework Node.js [69]. We use Node.js version 10.22.0 and version 6.14.6 of the corresponding packet manager *npm* [71].

5.2 Realization of the Smart Contracts

In the following section, we will explain the technical implementation of the smart contracts, which realize the proposed market system. We start by describing the general development process for the implementation of Solidity-based smart contracts. Afterward, we will provide implementation details of the smart contract architecture provided.

5.2.1 Development Process for Smart Contracts

Figure 5.3 illustrates the development process of a smart contract implemented with the Solidity programming language and provided on the Ethereum blockchain. As we can see, smart contracts are provided in the form of *.sol* files. In an intermediate step, the Solidity compiler solc is then used to compile the contracts and store the results in *.solc* files. These contain the bytecode representation of the smart contract logic, which is later executed by the EVM. Afterward, using the Truffle Suite, the compiled contract bytecode is then transformed into *application binary interface (ABI)* files, which are stored in the *JavasScript object notation (JSON)* format. The ABI files contain all information relevant for using the developed smart contracts. They are the standard way to interact with contracts in the Ethereum ecosystem [72]. On the one hand, ABI files contain the compiled bytecode. Later, the information about the Ethereum network used and the Ethereum contract addresses of the deployed smart contracts is also stored in these files.

After developing and compiling the smart contracts locally, we can now use the Truffle Suite and web3.js to deploy and test the contracts on a local Ganache based Ethereum blockchain as described in the previous section. Precisely, we can execute the developed tests locally and deterministically in

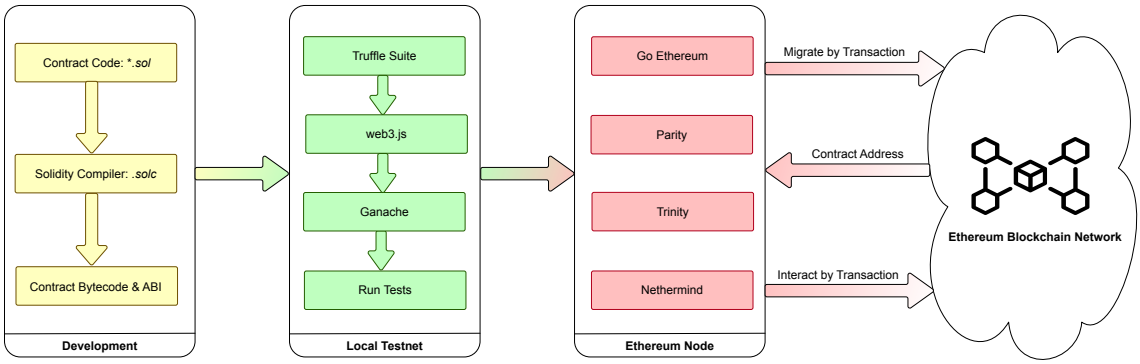


Figure 5.3: The development process for smart contracts based on the Solidity programming language.

a test environment in order to ensure that the smart contracts model the correct process logic.

Once the tests have verified that the smart contracts are correctly implemented and thus ready for the production environment, we can finally provide them on the original Ethereum blockchain. To this end, we must first establish a connection to the underlying peer-to-peer network of the Ethereum blockchain. As shown in Figure 5.3, we have to connect to an Ethereum node. Using the established connection to this Ethereum client, the web3.js library then allows a HTTP-based interaction with the network according to the Ethereum protocol. As indicated in Figure 5.3 in red, a variety of Ethereum client implementations exist. The most prominent ones are *Go Etherum* [73], *Parity* [74], *Trinity* [75], and *Nethermind* [76]. It is up to the developer to decide which client to use.

With the aid of the Truffle Suite, the connected Ethereum client can now be instructed to deploy the developed smart contracts to the network. In the Ethereum context, this step is called migration. For the migration, an Ethereum account is required to create an Ethereum transaction representing the smart contract code's migration to the network. After migration, the smart contract is available on each network node. Furthermore, a unique Ethereum contract address was assigned to the deployed smart contract during migration. As mentioned before, users can then use this contract address to invoke the functions of the smart contract. To do so, they create Ethereum transactions addressed to the contract and broadcast them to the network. For completeness, we would like to remind the reader that developers cannot modify the stored smart contract code. If the code is incorrect or the requirements evolve, the smart contract must be migrated again, which changes its contract address. Afterward, the data of the old smart contract is unavailable to the new deployed one.

5.2.2 The Implemented Smart Contract Architecture

In Figure 5.4, the architecture of the developed Solidity-based smart contracts is represented as a Unified Modeling Language (UML) class diagram [77]. Since it is not trivial to model smart contracts using the UML standard, we followed the recommendations of Marchesi et al. [78]. Thereby, the illustration of the smart contracts is simplified and limited to the aspects necessary for understanding the thesis. Further details can be found in the source code provided in Appendix 1. For a better visual interpretation, we have highlighted the smart contracts in yellow, struct data types in red, enumerations in green, and event types in blue in Figure 5.4.

As can be seen, we have indeed implemented all the smart contracts identified in Chapter 4.1. Also, the illustrated relationships between the smart contracts can be derived directly from the tasks, interactions, and deduced processes described in Chapter 4.1. Therefore, we do not want to discuss the relationships again. Among the smart contracts presented, the implementation of the Intraday and Day-Ahead smart contracts is of particular interest. As shown in Figure 5.4, both smart contracts are instances of the abstract smart contract interface *AbstractOfferManagement*. This is due to the fact that the behavior and implementation of both variants do not differ significantly. As formally defined in Chapter 3.3, both variants differ only in the length of the operating reserve series contained in an offer (see Equation 3.10 and 3.16) and the time requirements for the individual market operations (see Table 3.1). Therefore, these two aspects are defined abstractly in *AbstractOfferManagement* and have to be implemented individually by the Intraday and Day-Ahead trading instances. All other operations are identical and implemented in AbstractOfferManagement.

In Figure 5.4, we can also see that the ROs and GOs are represented as instances of the *User* struct of the Registry smart contract and managed there. Correspondingly, the DERs are also implemented as instances of the *Resource* struct data type here. Furthermore, we would like to mention that we can determine from the mapping *registeredUsers* that the users in the system are uniquely identified by their Ethereum blockchain account address. The User struct attribute *userAddress* represents this value. The pseudonymous hash identifier of the users defined in Equation 3.1 is reflected in the User attribute *hashIdentifier*. As already explainedn, this parameter is not used to identify the users but to avoid double-registration attempts. In contrast, the DERs are indeed identified by their pseudonymous hash identifier defined in Equation 3.5, as we can deduce from the depicted mapping *registeredResource*. The attribute *hashIdentifier* of the Resource struct represents this identifier. Both discussed structs also manage the states of a user or DER represented in Figure 4.4.

The individual placed intraday and day-ahead offers are represented by instances of the struct data type *Offer* and stored in the mapping *offersPerTimeframe*, both provided by the AbstractOfferManagement smart contract. The offers are then assigned to the ROs using the mapping *offersPerTimeframeOfResourceOperator*. As can be seen, an offer is uniquely identified by the trading timeframe and the hash identifier of the DER specified in the offer. The former is represented by the uint64 keys in the mappings just mentioned. The latter is represented by the Offer struct attribute *resourceID*. The positive and negative operating reserve advertised and purchased of an offer is represented in the offer elements by instances of the struct *Series*.

The mappings *depositedUserRegistrationFees* and *depositedResourceRegistrationFees* of the EscrowRegistry contract stores which users have deposited how many user and resource registration fees. As described above, users are identified by the Ethereum address and DERs by their hash identifier. The deposited SEC are represented as uint256 values.

As illustrated in Figure 5.4 and described in Chapter 4.1, the EscrowTrading contract manages all fees and funds necessary for trading purposes and realizes the proposed escrow mechanism. As illustrated, the EscrowTrading Contract stores all escrow information related to a specific offer in an instance of the struct *Escrow*. Here, the deposited place and accept offer fees as well as the stored trading funds are listed. In elements of the struct *AcceptOfferData*, Escrow instances also store how much operating reserve was assigned to the GO after clearing the escrow. In the mapping *escrows* of the EscrowTrading smart contract, all escrow instances are stored and assigned to a concrete

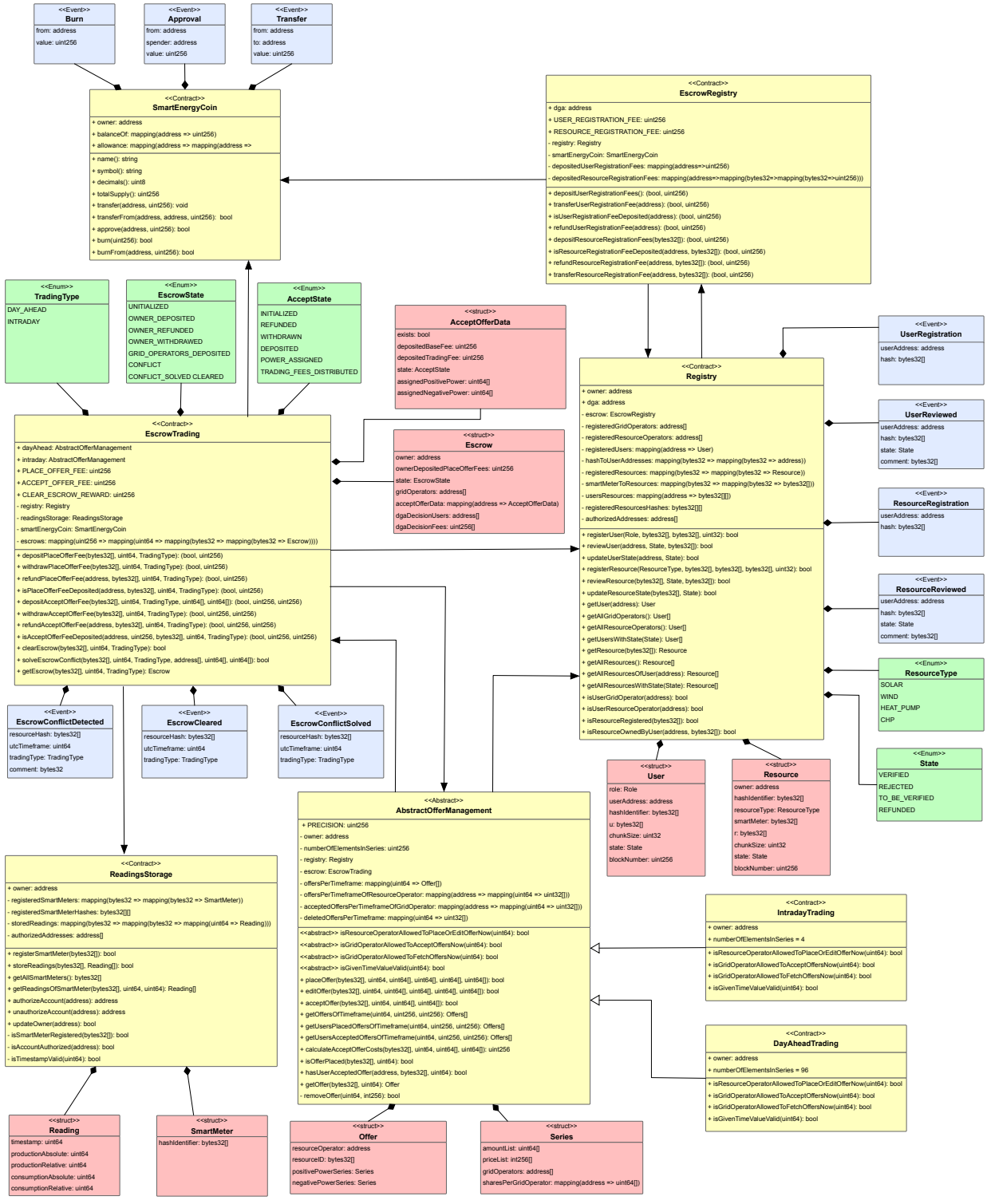


Figure 5.4: Architecture of the developed Solidity-based smart contract represented as UML class diagram.

offer. Thereby, an escrow is uniquely identified by the trading type, the hash identifier of the DER represented in the offer, and the concrete trading timeframe. The state of the individual escrows is also managed in these instances by the attribute *state* (see Figure 4.11 (a)).

The smart meters are stored in the ReadingsStorage contract in instances of the struct *SmartMeter*. On the other hand, the associated readings are represented in instances of the struct *Reading* and assigned to the smart meters in the mapping *storedReadings*.

Finally, we want to emphasize that the smart contracts identify the DGA and the SMM via their respective Ethereum account addresses wherever necessary. These are stored as variables in the contracts and are therefore known to them.

In order to not exceed the scope of the thesis, we do not want to discuss further details of the provided implementation of the smart contracts. For more in-depth information, we refer to the provided source code in Appendix 1.

5.3 Lessons Learned

In the following, we want to address some interesting aspects and challenges in using the still emerging Ethereum blockchain technology and the Solidity programming language. We hope that these insights might be relevant and helpful for other developers in realizing their systems.

Tests are Essential In the previous section, we have explained the development process of smart contracts. It is evident from these explanations that the utilization and the associated invocation of smart contract functions is only possible via blockchain transactions. However, these are complex and time-consuming to create. Besides, developers do not receive the return value of the smart contract function after the execution of the blockchain transaction. Therefore, for each function that modifies the status of a variable, a corresponding function that reads the variable must be used in order to be able to verify the modifications at all. Because of these reasons, we conclude that automated tests are essential for the development of smart contracts and that manual tests are fundamentally impractical. As a best practice, we recommend that developers implement automated tests from the beginning, as described in Chapter 5.1. This is also an excellent way to verify the requirements of the system to be developed.

Solidity and Ethereum Solidity is an object-oriented programming language based on the syntax of the established programming languages JavaScript, C++, and Python [22]. In our opinion, Solidity is easy to understand for developers experienced in these languages, and their learning curve will be steep. However, we think that the features of Solidity are still very limited in some areas and need to be improved for broader use.

For example, the Solidity language does not support floating-point numbers. Instead, it allows the usage of integer values up to a size of 256 bit. This allows circumventing the lack of floating-point numbers by representing them with a selected, preferably high, precision in large 256-bit integer numbers. For example, in our application, we have multiplied all values defined in Chapter 3 by the factor 10^{10} before storing them in the smart contracts. Accordingly, the values were divided by this factor during reading. This enables our users to specify floating-point numbers following the units

defined in Chapter 3. Nevertheless, depending on the required number range, the necessary precision might not be available through 256-bit integers and might render Solidity impractical in this case.

Another major shortcoming of the programming language is the missing support of structs and complex dynamic arrays as return values in functions. Although Solidity allows to define and store complex objects using structs, instances of these cannot easily be passed on to other smart contracts or the user. In such cases, the developer has to decompose the complex objects into their base components and serialize these into native data types. On the receiving side, the complex object must then be deserialized from the native data types provided. This makes the development process unnecessarily complicated and error-prone.

In addition, a limitation concerning the Ethereum EVM is to be mentioned. The EVM only allows a maximum stack depth of 16 variables. However, if a function needs more local variables and parameters, a *stack to deep* error needs to be circumvented. As a countermeasure, the developer can split the function into several single functions or hold several variables in an extra defined struct datatype, which is then only regarded as a single variable. However, this limitation makes the development of complex problems more challenging.

Furthermore, we would like to mention that Ethereum currently only supports a maximum smart contract size of 24 KB [79]. However, since the Registry smart contract exceeds this limit, the Ganache blockchain must be used using the flag – *allowUnlimitedContractSize*. This configuration eliminates this limitation locally. To allow the implementation of complex smart contracts, we recommend Ethereum developers to increase the predefined limit.

Community and Development Support In the Ethereum ecosystem, several libraries exist that support and accelerate the development process. Especially the already mentioned Truffle Suite and web3.js library are worth mentioning. However, error messages from these tools, the Solidity compiler, or the EVM are often very general, meaningless, or even misleading. In developer forums, we could often find tips and tricks for possible solutions to problems, but often complex manual debugging was necessary to isolate and fix concrete errors. We assume that as the technology matures and the developer community grows, the support increases and error messages are improved.

Missing Standardization Another reason for the problem just described may be some lack of standardization in the Ethereum ecosystem. Although the Ethereum protocol is well defined, there seems to be a lack of standards and specifications for any aspects based on the Ethereum implementation. For example, it has appeared that due to an update of the web3.js library from version 1.2.0 to 1.2.1, we had to modify the codebase. Concrete, the way of processing blockchain responses and setting default values has changed in this minor update. Well designed and certified standards and specification could avoid such problems.

In conclusion, in this chapter, we have presented the implementation of the system described in Chapter 4. Thereby, we described the realization using the Ethereum blockchain and the Solidity programming language. Furthermore, we highlighted important aspects and difficulties that arose while using these technologies.

6 Evaluation

After we have presented and discussed the system in detail in the last chapters, we want to evaluate it more precisely in the following chapter. For this purpose, we will describe how we have analyzed the system in terms of efficiency and scalability based on a market simulation and present the results obtained. Then we will discuss which of the requirements identified in Chapter 3.4 are fulfilled by the developed system. Thereby, we want to summarize the resulting characteristics of the system.

6.1 Gas Costs Analysis

In the following, we want to evaluate the performance of the developed system in terms of scalability and efficiency. As described in the previous chapters, the system is realized as a blockchain application. Thereby, the functionalities of the system are implemented by utilizing smart contracts, which are provided in the Ethereum blockchain and executed through corresponding blockchain transactions which are broadcasted to the Ethereum network. These transactions are processed decentralized and are eventually added to the blockchain. Therefore we conclude that the throughput of our system depends on the throughput of the used blockchain technology, which is defined by the implemented consensus mechanism and the associated properties like block time, transactions per block, and the resulting transactions per second. For the Ethereum blockchain used a block time of about 15 seconds and a corresponding throughput of up to 15 transactions per second applies [80]. Since the transactions are executed decentralized, the resource utilization in terms of CPU and memory usage is also not meaningful to evaluate the performance of our application. However, there exists a metric in the Ethereum ecosystem to examine the complexity of executing the smart contracts in the EVM - the gas costs (see Chapter 2.1.1). The gas costs are an approximate indicator for the efficiency and resource utilization of the developed smart contracts in the decentralized system[12]. Since the concrete gas costs of a smart contract function can depend on individual input parameters or already stored data, we can also make statements about the scalability of our system by varying these in a controlled manner. Thereby, we would like to remind the reader that each Ethereum block has a maximum block gas limit, which the sum of the gas costs of all transactions contained in a block must not exceed. Currently, the block gas limit for the Ethereum mainnet is 12487205 gas.[1] Therefore, in the following, we used this value to determine the maximum number of transactions per block in case of using the Ethereum mainnet blockchain. We consider this as an estimation of the maximum throughput of the developed system in case of operation in the Ethereum mainnet.

For the reasons mentioned above, we will therefore examine the gas costs of the essential smart contract functions in the following in order to be able to evaluate the efficiency and scalability of our system. To compare the virtual gas costs of the EVM with a physical metric for the efficiency and

1 Block gas limit of the Ethereum mainnet on the 08/17/2020 taken from `https://ethstats.net/`

Configuration Parameter	Value
Number of users to register	60 (30 ROs and 30 GOs)
Number of resources to register	60
Number of smart meters to register	60
Max number of offers per timeframe	30
Max number of purchasers per offer	29
Max number of readings to store	50
Ethereum mainnet block gas limit	12487205 gas[1]
Gas price	50 Gwei[2]
ETH price	326.43 €[3]

Table 6.1: The evaluation tests configuration parameters used for plots and analysis.

scalability of a system, we have also measured the time required to mine each transaction. These transaction times allow us to estimate whether the gas cost correlates with the required computing time and complexity. However, for the reasons mentioned above, we do not collect additional hardware-related metrics.

6.1.1 Evaluation Test Setup

As already indicated, we performed a market simulation to collect empirical data on the gas costs of the developed smart contract functions. Thereby, we examined the smart contract code in advance and identified on which data and input parameters the complexity of the functions depends. These parameters were then varied in the conducted market simulation to investigate the scalability of the system. For example, the effort to execute an accept offer operation depends on the number of GOs that have already accepted the offer because the system must ensure that sufficient operating reserve shares are still available. Therefore the number of purchasers per offer was varied in the market simulation. In Table 6.1, the configuration parameters of the simulation and evaluation are shown. The exact workflow and interesting details of the performed market simulation will be explained gradually in the course of this chapter

As we have already mentioned in Chapter 5.1 and illustrated in Figure 5.1, the market simulation is performed by the *Data Generator* component. During the market simulation, the smart contracts are provided in a Ganache blockchain. Therefore, all simulated transactions are executed locally and deterministically by Ganache, making the evaluation reproducable. Since the Ganache blockchain also utilizes the EVM, we can transfer the observed gas costs to the Ethereum blockchain. In order to research the limits of the system, we have increased the block gas limit of the Ganache blockchain to 99900000000 gas. We also want to mention that we only considered functions that modify the state of smart contracts because only for these gas costs are charged in the Ethereum blockchain. In contrast, reading operations are free of charge and highly scalable because every network node can keep and read a local copy of the blockchain. The evaluation was performed on a MacBook Pro with the operating system macOS Catalina version 10.15.7, a 3 GHz dual-core Intel Core i7 processor, and 16 GB DDR3 memory. In the following, we will now present the evaluation results of the market simulation.

Smart Contract	Transaction Costs		
	[gas $\cdot 10^3$]	[ETH]	[€]
Registry	6729.201	0.336460	109.83
IntradayTrading	5265.794	0.263290	85.95
DayAheadTrading	5328.608	0.266430	86.97
EscrowRegistry	2271.966	0.113598	37.08
EscrowTrading	9030.269	0.451513	147.39
ReadingsStorage	1914.682	0.095734	31.25
SmartEnergyCoin	772.757	0.038638	12.61

Table 6.2: Overview showing the deployment costs of the smart contracts.

6.1.2 Deployment Costs

We have already explained in Chapter 5.2 that smart contracts are deployed to the Ethereum blockchain via a blockchain transaction. Since fees must be paid for these transactions, we will begin with examining the deployment costs of the developed smart contracts. In Table 6.2, the gas costs of the smart contract migration transactions are listed. We have also calculated and presented the actual costs to be paid for each value in the native Ethereum currency ETH. In addition, we have converted and provided this value in the fiat currency Euro for better understanding. We can calculate the ETH costs of a blockchain transaction as follows:

$$transactionCosts_{ETH} := transactionCosts_{gas} \cdot price_{gas} \cdot 10^{-9} \tag{6.1}$$

Based on this value we can then calculate the corresponding Euro value using the equation

$$transactionCosts_{Euro} := transactionCosts_{ETH} \cdot price_{ETH} \tag{6.2}$$

As can be seen in Table 6.1, we assumed a gas price of 50 Gwei[2] and an ETH price of 326.43 €[3] for all calculations in this chapter. Table 6.2 shows that the SmartEnergyCoin smart contract with 772.757 gas $\cdot 10^3$ and the ReadingsStorage with 1914.682 gas $\cdot 10^3$ have the most cost-effective deployment costs. This is due to the fact that they implement the least complex smart contract code. In general, in the Ethereum blockchain it applies that the larger the smart contract size, the higher the deployment costs. Therefore, the Registry contract with 6729.201 gas $\cdot 10^3$ and the EscrowTrading contract with 9030.269 gas $\cdot 10^3$ have the highest deployment costs because their corresponding smart contract code is significantly more complex. As explained in Chapter 5.2, both trading contracts implement the same abstract interface. Therefore, the deployment costs of these two are very similar. Differences in the deployment costs can be explained by the varying implementation of the few non-abstract functions (see Chapter 5.2). In concrete terms, the IntradayTrading smart contract has a deployment cost of

2 Recommended average gas price for the 11/4/2020 according to `https://etherscan.io/chart/gasprice`
3 Price taken on 11/4/2020 from `https://www.finanzen.net/devisen/ethereum-euro-kurs`

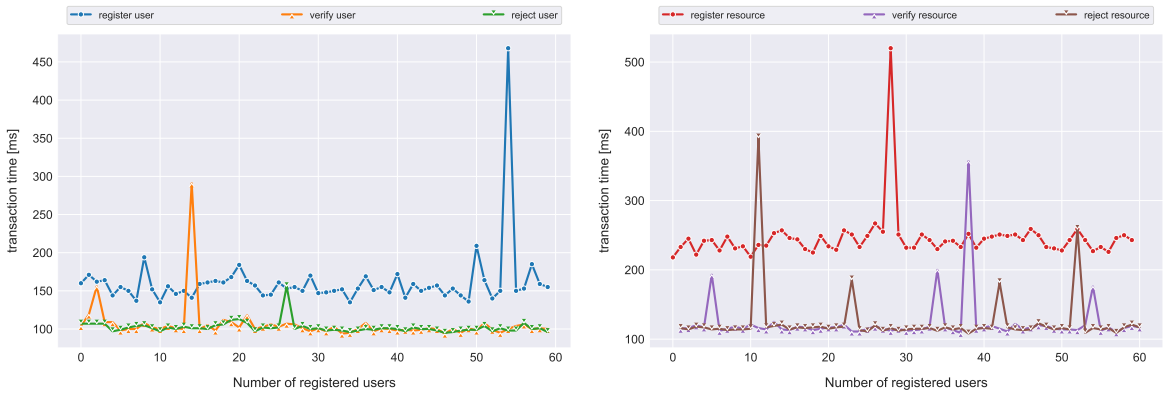
5265.794 gas · 10^3, whereas 5328.608 gas · 10^3 is required for the migration of the DayAheadTrading contract. Finally, the EscrowRegistry contract requires a deployment cost of 2271.966 gas · 10^3.

Table 6.2 also shows the deployment costs represented in the fiat currency Euro. Here we can observe that the deployment costs are in the range of 12.61€ to 147.39€. In order to deploy all contracts, a summed cost of 511.08€ is charged. Therefore, a single system deployment is very cost-intensive.

6.1.3 Registry Transactions

In the following, we want to begin evaluating the first smart contract functions in terms of efficiency and scalability. We start by discussing the essential functions of the Registry smart contract. Specifically, we will discuss the functions for registering and verifying a user or DER. The data statistics of all our measurements discussed in this chapter are presented in Table 6.3.

As described in Chapter 3.1.1, the double-registration of users and DERs is prevented by using a pseudonymous hash identifier. Therefore, all already stored hash identifiers must be considered during registration. For this reason, we have evaluated these functions in relation to the number of users and DERs already registered in the system. As can be concluded from Table 6.1, we have stepwise registered 60 users in the market simulation. Thereby we registered a total of 30 ROs and GOs in the system. Furthermore, each of the 30 ROs registered two DERs in the simulation, resulting in a total of 60 DERs. From the function `registerUser` listed in Table 6.3 we can infer that the transactions costs for registering a user are almost constant. The function manifests an average gas cost of 908.716 gas · 10^3 with a standard deviation of just 2.716 gas · 10^3. Similary, the costs for performing DER registration using the `registerResource` function are relatively stable with an average gas cost of 1388.954 gas · 10^3 and a standard deviation of 10.877 gas · 10^3. As discussed in Chapter 5.2, the reason for this is that hash identifiers of both objects are stored in a Solidity mapping. This enables us to verify if the respective hash identifier is already used in a constant time, rendering the complexity of the whole registration process constant. Small variations in the costs of these functions can be explained by the initial initiation of different data instances and helper objects. Therefore, we can state that both variants' computational complexity does not depend on the number



(a) Transaction times for user transactions

(b) Transaction time for resource transactions

Figure 6.1: Illustration of measured transaction times for registring and verifying users and resources.

of already registered users or DERs. As shown in Table 6.3, we can represent a maximum of 13 user registration and up to 9 resource registration transactions in one Ethereum mainnet block.

Furthermore we can deduce from the statistics of the functions `reviewUser` and `reviewResource` listed in Table 6.3 that also the transactions for positive or negative verification of the users or DERs are to be regarded as constant. As listed, the average transaction cost for positive user verification is 131.308 gas $\cdot 10^3$ with a standard deviation of 0.003 gas $\cdot 10^3$. For a negative verification, the average transaction cost is 131.403 gas $\cdot 10^3$ with the same standard deviation. For DERs, the transaction costs are 152.517 gas $\cdot 10^3$ in the positive case and 152.588 gas $\cdot 10^3$ in the negative case. Both cases have no standard deviation. Furthermore, we can observe from the table that we can place a maximum of 95 user and 81 resource review transactions in an Ethereum mainnet block.

As illustrated in Figure 6.1, the transaction times of the performed transactions to register and verify users and DERs also contribute to the observation that the complexity to perform the functions is nearly constant and does not depend on the number of already registered users or DERs. Fluctuations of the measurements shown can be explained with resource allocation by the operating system.

In summary, we conclude that the Registry functions have been implemented in an efficient and scalable fashion. All discussed functions demonstrate an almost constant complexity and do not depend on the already registered objects of the same kind. We have also shown that in the Ethereum mainnet up to 13 user or 9 resource registration transactions per 15 seconds are possible. Therefore, we consider the possible throughput for registration purposes as sufficient. However, we want to mention that the registration costs for users with a maximum of 15.07€ and for resources with a maximum of 23.09€ are quite expensive.

6.1.4 Intraday and Day-Ahead Trading Transactions

In the following section, we will now examine the essential trading functionalities of our developed system. First, we will discuss the functions for placing and editing an offer. Then we evaluate the functions for accepting the offers. Finally, we discuss the efficiency and scalability of the payment process. For all functions, we differentiate between the two market variants intraday and day-ahead.

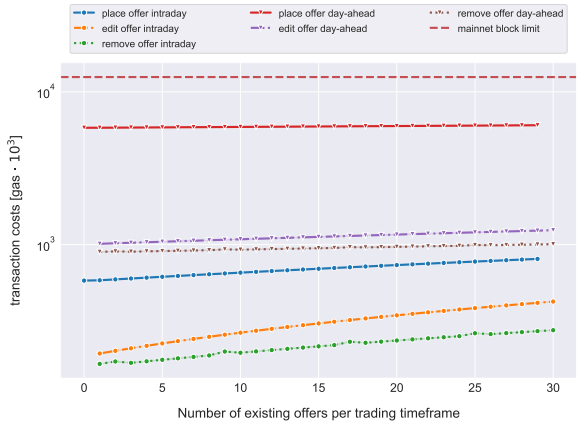
Place Offers

In the market simulation performed, each of the 30 registered ROs placed offers in both market variants for their two registered DERs. Thereby offers for different successive trading timeframes were placed in both market variants. Additionally, each of the ROs edited the placed offers. Therefore 60 intraday and day-ahead offers were placed and edited for each trading timeframe. Moreover, the measured data were analyzed in relation to the number of existing offers in a trading timeframe. Figure 6.2 and Table 6.3 present the gathered measurement results.

In Figure 6.2 (a), we can observe that the gas costs of all displayed functions increase linearly with the number of existing offers per trading timeframe. In addition, Figure 6.2 (d) shows the statistical distribution of the measured gas costs. Please note that both figures use a logarithmic scale. As a reason for the linear correlation to the number of existing offers, we can state that we have to consider all existing offers for the concrete timeframe to check whether an offer already exists. Therefore the computational effort increases linearly with the number of stored offers. Furthermore, we can

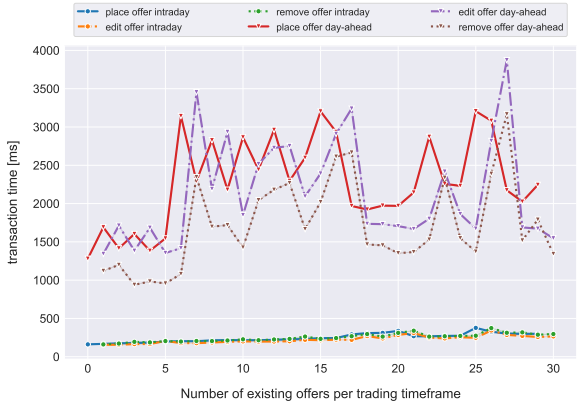
observe in all plots that the computational effort for the day-ahead functions is significantly higher than for the intraday functions. This is mainly due to the fact that the day-ahead functions require much more data to be processed and stored, which results in higher gas costs. Concretely we can infer from Table 6.3 that the function `placeOffer` has a minimum gas cost of 570.463 gas $\cdot 10^3$ and a maximum gas cost of 819.889 gas $\cdot 10^3$ for the intraday market. The calculated average value is 691.389 gas $\cdot 10^3$. As already mentioned, this function is significantly more expensive for the day-ahead case. As we can see, in this case, a minimum value of 5805.164 gas $\cdot 10^3$ and a maximum value of 6054.590 gas $\cdot 10^3$ is observed. The average value here is 5918.737 gas $\cdot 10^3$.

Furthermore, we investigated the function `editOffer` for both market variants. Here we distinguish between two alternatives. On the one hand, it is possible to withdraw an offer using this function, which we refer to as *remove offer* in the plots. On the other hand, it is also possible to edit the stored data by invoking this function. As we can conclude from the presented data and plots, a linear dependency on the number of existing offers can be identified in both cases. The reason for this is that when reading the offer to edit, all stored offers must be considered again. For the pure editing of an offer, gas costs of at least 193.843 gas $\cdot 10^3$ and at most 423.900 gas $\cdot 10^3$ are required in

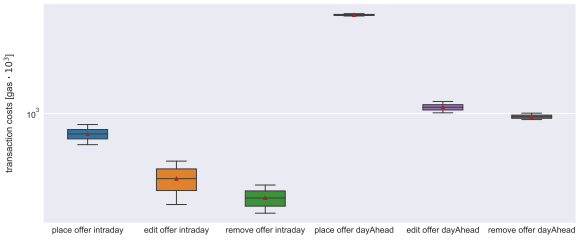


(a) Gas costs

(b) Transactions per Ethereum mainnet block



(c) Transaction time



(d) Boxplot showing distribution of gas costs

Figure 6.2: Visual evaluation of transaction metrics for placing and editing offers.

the intraday case. The average is 308.972 gas · 10^3. For the withdrawal of an offer, minimum and maximum costs of 165.759 gas · 10^3 and 275.412 gas · 10^3 are charged. The average is 219.075 gas · 10^3. In contrast, for the day-ahead contract, the minimum cost for editing is 1012.537 gas · 10^3, and the maximum is 1242.594 gas · 10^3. On average, 1120.199 gas · 10^3 is required here. For the removal, the minimum value is 896.724 gas · 10^3, whereas the maximum value is 1006.377 gas · 10^3. The average cost can be found at 946.441 gas · 10^3.

In Figure 6.2 (c), we can further see that for all discussed functions, the transaction time also increases slightly with the number of existing offers. However, we can recognize that for all cases, the transaction time is in a range from 155 ms to 3815 ms, which we describe as very good.

Besides, Figure 6.2 (b) illustrates the number of transactions that can be placed in one Ethereum mainnet block for each discussed function in relation to the number of existing offers. Since the gas cost of the transactions increases with the number of existing offers, it is trivial that the number of transactions per block decreases with the number of offers. As we can see, in the Ethereum mainnet, a maximum of 21 intraday and 2 day-ahead offers can be submitted every 15 seconds. However, up to 64 intraday and 12 day-ahead offers can be edited in one block.

In conclusion, we want to summarize that none of the described functions exceed the Ethereum mainnet limit. Besides, we observed an increasing linear dependency on the number of offers existing in a trading timeframe for all functions. Since the growth is linear and minimal, we describe the implementation of the functions as scalable and efficient. However, we want to state that the possible throughput per Ethereum block is limited by the corresponding block gas limit. It depends on the actual market dynamics of the users if this is sufficient for our system. Finally, we would like to mention that the cost of executing the functions with average values from 3.58€ to 96.60€ is very cost-intensive.

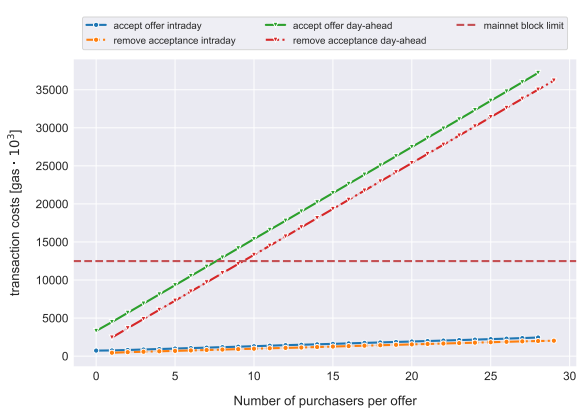
Accept Offers

After we have discussed the placing of the offers, we would like to examine the acceptance of them in the following. In the market simulation, the 30 registered GOs accepted the offers placed by the ROs. Since each accept offer operation requires to check whether there is still enough operating reserve share available or whether the other listed GOs have already purchased everything, we examined the function `acceptOffer` in relation to the number of purchasers per offer. As shown in Table 6.1, a maximum of 29 GOs accepted the same offer in the simulation. Figure 6.3 and Table 6.3 show the results of our measurements. Please note that we have investigated two different options for the `acceptOffer` functions. On the one hand, it is possible to use this function to accept a new offer or to change the stored data. On the other hand, a GO can use this function to withdraw an accepted offer and thus withdraw as a purchaser, which we refer to as *remove acceptance* in the data and plots.

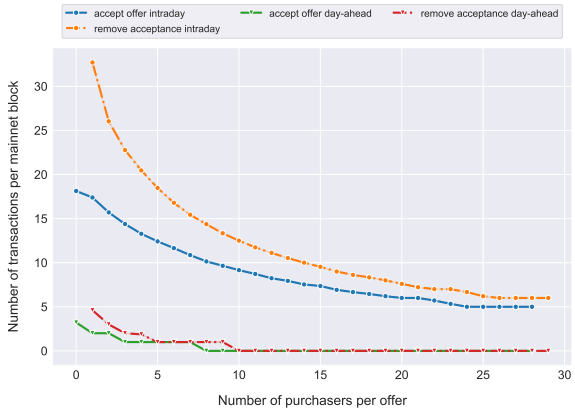
In Figure 6.3 (d) the statistical distribution of the measured gas costs is shown as a boxplot. In addition, Figure 6.3 (a) depicts the gas costs in relation to the number of purchasers per offer. As we can see, the gas costs are linear dependent on the number of purchasers per offer. The higher the number of purchasers, the higher the gas costs. In general, we can observe that for the day-ahead case, the gas costs are significantly higher than for the intraday case. Again, in the intraday case, less data has to be verified, processed, and stored. As can be seen in Table 6.3, the intraday accept offer

gas costs are at a minimum value of 405.025 gas · 10^3 and a maximum value of 2445.868 gas · 10^3. The average is 1266.469 gas · 10^3. For the day-ahead market, the costs are at least 3013.668 gas · 10^3 but not more than 37215.871 gas · 10^3. The average value is 13858.795 gas · 10^3. For withdrawing from an already purchased offer, the intraday costs are in a range from 165.814 gas · 10^3 to 2024.196 gas · 10^3 with an average of 986.341 gas · 10^3. For the day-ahead market, the costs are in a range from 2171.430 gas · 10^3 to 36205.358 gas · 10^3. The average is 12997.215 gas · 10^3.

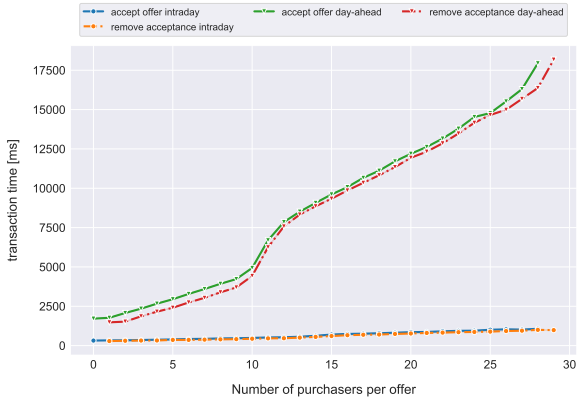
Furthermore, Figure 6.3 (b) illustrates the number of transactions per mainnet block depending on the number of purchasers per offer. As we can see, for the intraday market, we can mine up to 18, but not more than 5 accept offer transactions in one Ethereum mainnet block and thus execute them every 15 seconds. For the day-ahead case, a maximum of 4 such transactions are supported. However, we can see in Figure 6.3 (a) that starting from a value of 8 purchasers per offer, the gas costs exceed the Ethereum mainnet block gas limit. The corresponding transaction can, therefore, no longer be represented in such a block. This clearly causes scaling problems when operating our system in the Ethereum mainnet, as the block gas limit strongly limits the maximum number of day-ahead purchasers per offer.



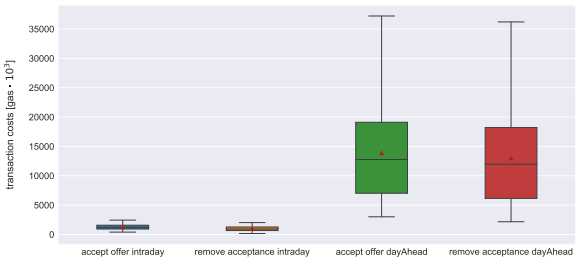
(a) Gas costs



(b) Transactions per Ethereum mainnet block



(c) Transaction time



(d) Boxplot showing distribution of gas costs

Figure 6.3: Visual evaluation of transaction metrics for accepting offers.

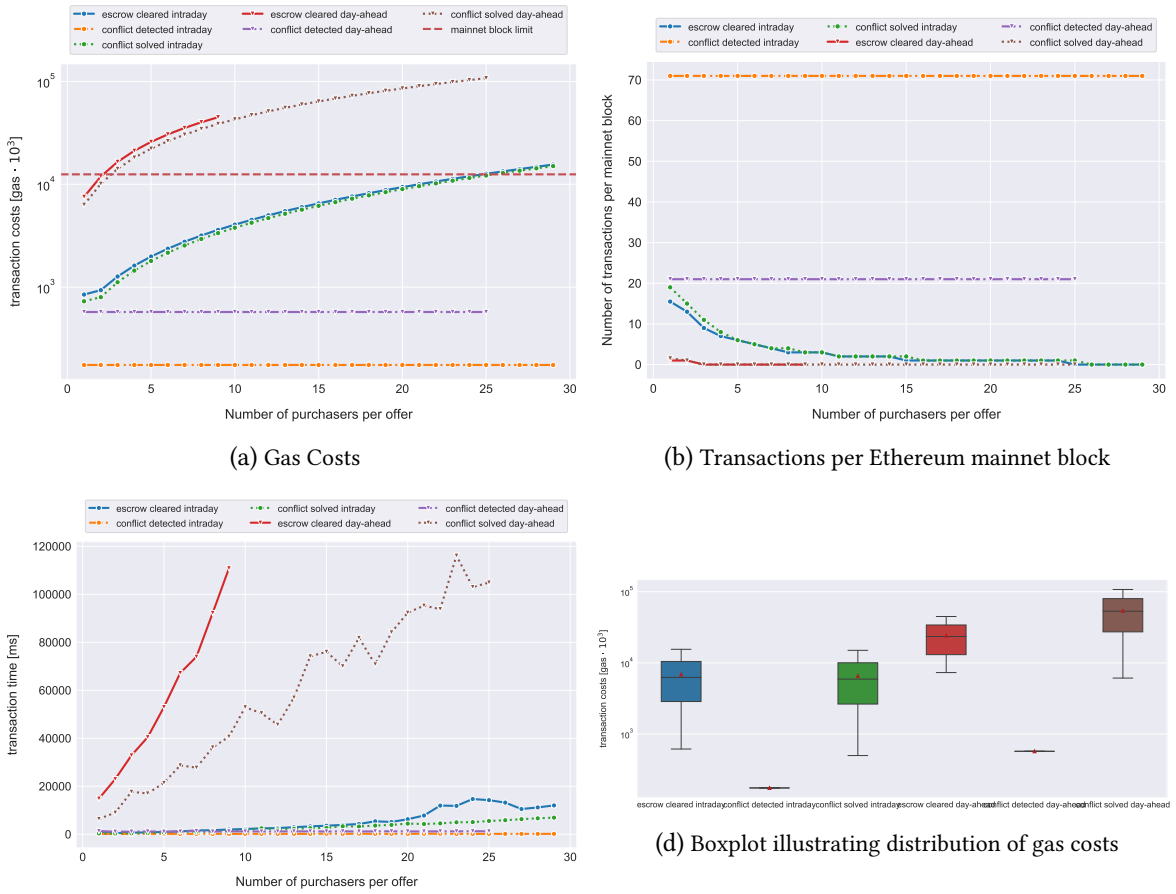
Finally, we want to investigate the transaction times shown in Figure 6.3 (c). Again, the required transaction time correlates strongly with the gas costs and increases with the number of purchasers per offer. The values lie between 134 ms and 1335 ms for the intraday market variant, which can be regarded as good. However, for the day-ahead case, the values are already in the range of 1003 ms and 18359 ms, which is significantly higher but sufficiently fast.

In summary, we can conclude that the accept offer operations for intraday are sufficiently efficient and scalable. However, scaling problems can be observed in the current implementation of the day-ahead market. Furthermore, we would like to mention that the average costs of 20.67€ for intraday and 226.20€ for day-ahead are again very cost-intensive (see Table 6.3).

Clear Escrows and Solving Conflicts

In order to conclude the evaluation of the trading functions, we now want to discuss the costs for clearing an escrow and solving escrow conflicts. We recall that these functions allocate operating reserves to the GOs based on the meter readings and execute the payment process. It should be clear that the complexity of these operations depends on the number of GOs that have acquired shares in a specific offer. Therefore, we examined these functions in relation to the number of purchasers per offer. For this purpose, all offers already accepted in the market simulation were used, resulting in a maximum number of 29 purchasers per offer. The results of our measurements are shown in Figure 6.4 and listed in Table 6.3 as the functions `clearEscrow` and `solveEscrowConflict`. For the former function, we additionally distinguish between the two cases where the payment process was either executed successfully, or a conflict was simulated due to missing measurement data.

In Figure 6.4 (d) the statistical distribution of the measured gas costs is shown as a boxplot. In addition, Figure 6.4 (a) illustrates the measured gas costs in relation to the number of purchasers per offer. We want to mention that a logarithmic scale is used in the plots. First of all, we want to examine the function `clearEscrow`. We will start with the cases where a conflict was detected during execution. As we can see, in these cases, the gas costs are constant for both market variants. This is due to the fact that in these cases, the detected conflict is stored and the function is terminated directly after the attempted fetching of the meter readings. As can be seen in Table 6.3, the intraday case results in gas costs of 175.183 gas $\cdot 10^3$, which means that 71 such transactions can be mapped in one Ethereum block. For day-ahead, the gas costs are constant at 573.360 gas $\cdot 10^3$, so only 21 transactions can be mapped in an Ethereum mainnet block. Next, we will consider the cases in which the payment process was successfully executed by the function `clearEscrow`. In Figure 6.4 (a), we can see that in these cases, the gas costs increase significantly with the number of purchasers per offer. Precisely, we can observe a linear relationship. In Table 6.3, we can see that for the intraday market variant the values lie in a range from 617.022 gas $\cdot 10^3$ to 15532.701 gas $\cdot 10^3$, with an average value of 6873.077 gas $\cdot 10^3$. On the other hand, day-ahead gas costs of at least 7314.456 gas $\cdot 10^3$ but not more than 44863,195 gas $\cdot 10^3$ are charged, which are significantly higher. The average here is 24126.871 gas $\cdot 10^3$. Finally, we consider the function to solve an escrow conflict. Again, Figure 6.4 (a) shows a linear relationship to the number of purchasers. Our measurements indicate that gas costs of at least 499.385 gas $\cdot 10^3$ and a maximum of 15018.371 gas $\cdot 10^3$ are accumulated in the intraday case. For day-ahead, the values are already in the range between 6120.840 gas $\cdot 10^3$ and 107508.669



(a) Gas Costs

(b) Transactions per Ethereum mainnet block

(c) Transaction time

(d) Boxplot illustrating distribution of gas costs

Figure 6.4: Presentation of metrics of the clear escrow and solve escrow conflict transactions.

gas · 10^3. The average for intraday is 6546.006 gas · 10^3, while for day-ahead, it is 53973.662 gas · 10^3.

The transaction times measured and plotted in Figure 6.4 (c) also increases linearly with the number of purchasers per offer. For the intraday case, the values are in a range from 370 ms to 14682 ms, which can be described as good. However, the transaction times of the day-ahead variant are between 13633 ms and 116104 ms, which is very high. Furthermore, we can see in Figure 6.4 (a) and 6.4 (c) that in our simulation the execution of the `clearEscrow` function for the day-ahead market was only possible up to a maximum of 9 purchasers per offer. However, for the function `solveEscrowConflict` an execution was feasible up to a maximum of 25 purchasers per offer. If the number of purchasers exceeded the respective values, the Ganache blockchain used for the simulation could no longer execute the respective transaction because the memory of the hardware used was insufficient. As a result, we want to point out that we are facing scaling problems of the day-ahead market variant regarding the payment process.

Figure 6.4 (b) shows the number of possible transactions of the respective functions per Ethereum mainnet block in relation to the number of purchasers per offer. Here we would like to point out that in the day-ahead case, a transaction to clear as well as to solve an escrow conflict with 3 or more

purchasers per offer cannot be included in an Ethereum mainnet block. For the intraday case, this applies starting from 26 purchasers per offer. Therefore, scaling problems of the functions regarding the operation in the Ethereum mainnet due to the block gas limit have to be mentioned here as well.

In summary, we can conclude that the implemented payment process is linearly dependent on the number of purchasers per offer. Nevertheless, we have encountered scaling problems in both the intraday and day-ahead variants when using the Ethereum mainnet. Furthermore, we can see in Table 6.3 that depending on the executed operation, average costs of 2.86€ to 880.93€ are incurred, which critically questions the economic efficiency of the system in use with the Ethereum blockchain.

6.1.5 Readings Storage Transactions

Finally, we also want to examine the gas costs of the essential functions of the ReadingsStorage smart contract.

Register Smart Meters

First, we want to examine the cost of the `registerSmartMeter` function, which registers smart meters in the system. In the market simulation, we have analyzed the costs of this function in relation to the smart meters already registered. As shown in Table 6.1, we registered a total of 60 smart meters in the market simulation.

As listed in Table 6.3, the gas costs for the smart meter registration are on a constant level with an average value of 154.911 gas $\cdot 10^3$ and a standard deviation of 1.936 gas $\cdot 10^3$. This fact is again due to the fact that the smart meter identifiers are managed in a mapping, and therefore the double registration can be verified in constant time. Small deviations can again be explained by the extra effort for the initiation of objects and counters. Based on the measured values, we can perform up to 80 smart meter registration transactions in one Ethereum mainnet block. In Figure 6.5 (d) we can also see that the transaction time for the smart meter registration is constant.

Store Smart Meter Readings

In the market simulation, we also stored production and consumption readings of the registered smart meters. These transactions executed the function `storeReadings`, which expects a list of smart readings to be stored as input. Thereby the length of the input list is variable. As shown in Table 6.1, in the simulation, we increased the number of readings to be stored in a single transaction continuously up to a maximum value of 50 consecutive readings. Thereby we increased the number of readings in steps of 5.

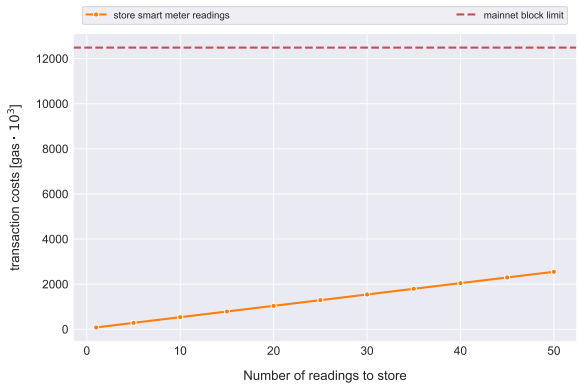
As a result, Figure 6.5 (a) illustrates the gas costs for the `storeReadings` function in relation to the number of reading data points to store. We can observe that the gas costs increase linearly with the number of reading data points to store. The growth is to be expected because the amount of storage required to store the readings grows with the increasing number of provided readings. But since storage allocation in the EVM has to be paid with gas [12], the observed growth can be explained by this fact. However, we want to state that the costs increase within a reasonable range and never exceed the Ethereum mainnet limit. Another interesting aspect regarding this function is shown in

Table 6.2. As we can see, the smallest measured value for storing a single data point is 78.257 gas $\cdot 10^3$, which is equivalent to 1.28€. In contrast, the maximum value for storing 50 data points is 2546.905 gas $\cdot 10^3$, which is 41.57€. Since it is obviously cheaper to store 50 data points once than 50 single data points several times, we recommend storing readings in batches.

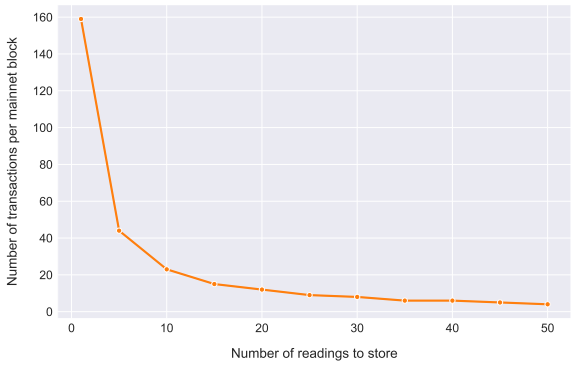
Figure 6.5 (b) illustrates the maximum number of store readings transactions to be placed in a mainnet block in relation to the number of readings to store. As we can see, when storing a single data point, we can mine a maximum of 159 transactions into a block, which corresponds to 159 stored data points. On the other hand, we can see that with a batch size of 50 readings, we can mine up to 4 blocks in one block, which corresponds to 200 stored data points. For efficiency reasons, we recommend again to use larger batch sizes for storing smart meter readings.

Besides, the visualization of the measured transaction times in Figure 6.5 (c) and 6.5 (d) demonstrates again that the transaction time of the transactions correlates positively with the gas costs. Furthermore, we can state that the transaction time with a median value of about 300 ms is within an appropriate range.

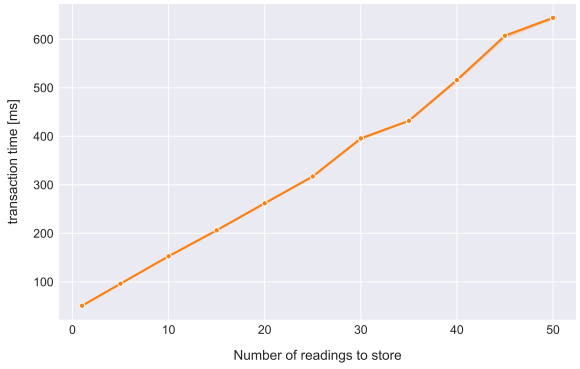
In conclusion, we can state that the ReadingsStorage functions are also implemented in an efficient and scalable manner. Again, we want to point out that the execution of the individual transactions in the Ethereum mainnet would be very cost-intensive.



(a) Gas costs



(b) Transactions per Ethereum mainnet block



(c) Transaction time

(d) Boxplot visualizing distribution of transaction time

Figure 6.5: Transaction data metrics visualization for storing meter readings.

Function	Transaction Costs [gas $\cdot 10^3$]				Transaction Costs [€]				Transactions per Block		
	min	avg	max	std	min	avg	max	std	min	avg	max
Registry											
registerUser	908.166	908.716	923.267	2.716	14.82	14.83	15.07	0.04	13	13.00	13
reviewUser [positive]	131.297	131.308	131.309	0.003	2.14	2.14	2.14	0.00	95	95.00	95
reviewUser [negative]	131.392	131.403	131.404	0.003	2.14	2.14	2.14	0.00	95	95.00	95
registerResource	1365.194	1388.954	1414.406	10.887	22.28	22.67	23.09	0.18	8	8.58	9
reviewResource [positive]	152.517	152.517	152.517	0.000	2.49	2.49	2.49	0.00	81	81.00	81
reviewResource [negative]	152.588	152.588	152.588	0.000	2.49	2.49	2.49	0.00	81	81.00	81
IntradayTrading											
placeOffer	570.463	691.389	819.889	69.969	9.31	11.28	13.38	1.14	15	17.75	21
editOffer	193.843	308.872	423.900	69.243	3.16	5.04	6.92	1.13	29	42.13	64
editOffer - remove	165.759	219.075	275.412	34.157	2.71	3.58	4.50	0.56	45	57.90	75
acceptOffer	405.025	1266.469	2445.868	450.346	6.61	20.67	39.92	7.35	5	10.87	30
acceptOffer - remove	165.814	986.341	2024.196	411.367	2.71	16.10	33.04	6.71	6	15.21	75
DayAheadTrading											
placeOffer	5805.164	5918.737	6054.590	66.416	94.75	96.60	98.82	1.08	2	2.00	2
editOffer	1012.537	1120.199	1242.594	65.642	16.53	18.28	20.28	1.07	10	10.71	12
editOffer - remove	896.724	946.441	1006.377	32.438	14.64	15.45	16.43	0.53	12	12.61	13
acceptOffer	3013.668	13858.795	37215.871	7988.311	49.19	226.20	607.42	130.38	0	0.79	4
acceptOffer - remove	2171.430	12997.215	36205.358	7960.956	35.44	212.13	590.93	129.93	0	1.07	5
EscrowTrading											
clearEscrow [intraday]	617.022	6873.077	15532.701	4616.206	10.07	112.18	253.52	75.34	0	3.40	20
clearEscrow - conflict detected [intraday]	175.183	175.183	175.183	0.000	2.86	2.86	2.86	0.00	71	71.00	71
solveEscrowConflict [intraday]	499.385	6546.006	15018.371	4493.423	8.15	106.84	245.12	73.34	0	3.90	25
clearEscrow [dayAhead]	7314.456	24126.871	44863.195	13412.086	119.38	393.79	732.23	218.91	0	0.30	1
clearEscrow - conflict detected [dayAhead]	573.360	573.360	573.360	0.000	9.36	9.36	9.36	0.00	21	21.00	21
solveEscrowConflict [dayAhead]	6120.840	53973.662	107508.669	31967.518	99.90	880.93	1754.70	521.76	0	0.15	2
ReadingsStorage											
registerSmartMeter	154.661	154.911	169.661	1.936	2.52	2.53	2.77	0.03	73	79.88	80
storeReadings	78.257	1293.141	2546.905	790.124	1.28	21.11	41.57	12.90	4	26.45	159

Table 6.3: Table showing the data statistics of the transactions cost data of the smart contract functions based on the performed evaluation tests.

6.2 Analysis of Fulfilled Requirements

In the following, we want to review the requirements defined in Chapter 3.4 and analyze which of them were fulfilled in the developed system. For this purpose, we first consider the functional requirements. Then we discuss the compliance with the non-functional requirements by discussing system properties of the developed system.

Functional Requirements In the developed system, users can register themselves and their DERs pseudonymously according to requirements 3 and 4. In cooperation with the DGA, the system is able to guarantee the integrity of their provided registration data.

After successful registration on the markets, ROs can place and edit offers for their registered DERs on the intraday and day-ahead markets according to requirements 7, 8, and 9. Subsequently, pseudonymously registered GOs can retrieve and accept these offers as described in requirements 10 and 11. Following requirement 11, the system provides an escrow mechanism for all trading processes, enabling a semi-automated payment process based on the consumption and production measurement data of DERs. In this context, the SMM is capable of providing the system with the measurement data of the smart meters assigned to the DERs according to requirements 1 and 2. If the automated payment process is not feasible due to missing measurement data, the DGA can solve the conflict in the system as described in requirement 13. As defined in requirement 3, the system uses a dedicated ERC-20 compatible cryptocurrency called SEC for all payment processes.

Furthermore, as we explained in Chapter 5.1, black-box tests were developed during the implementation of the prototype. These tests are designed to verify that the system is implemented correctly and that all functional requirements are satisfied as defined. Specifically, they also verify all aspects and functionalities mentioned above. Thus, we want to declare that the prototype meets all defined functional requirements 1-13.

Non-functional Requirements Concerning the non-functional requirements, we can first of all state that requirement 14 has been fulfilled. The system was implemented using the Ethereum blockchain, which implements the concept of general-purpose smart contracts. All functionalities of the system were implemented by utilizing smart contracts, which are then made available to the users in the Etherum blockchain for open access.

We also consider requirement 15 as fulfilled. In the implemented registration process, users can register themselves pseudonymously. By providing additional encrypted information, the system is able to verify the integrity of the users and their pseudonymous identifier with the help of the DGA. The same applies to the DERs. Since the users act pseudonymously in the markets and no entity other than the DGA can link the pseudonymous identities to real-world users, we conclude that the users' privacy and the confidentiality of their trading data are protected in our system.

In this context, we also classify requirement 16 as fulfilled. On the application layer, the aspects just discussed for requirement 15 contribute to the general security, confidentiality, and data integrity of the system. On the underlying technology layer, the used Ethereum blockchain guarantees the security and integrity of the system. As described in Chapter 2.1.1, the blockchain ensures the security and integrity of the managed data and executed transactions through cryptographic methods.

Additionally, the data stored on the Ethereum blockchain is tamper-proof. Due to the decentralized data administration and processing of transactions, we also want to emphasize the increased reliability, availability, robustness, and fault-tolerance of our system. By using the blockchain, users can also openly access our markets and do not have to trust any third party for the trading processes. They only need to trust the DGA and the SMM during the registration process and submission of the measurement data. However, we believe that this restriction is justifiable concerning the required integrity and security of the energy system and the resulting need for governmental regulation.

Furthermore, we consider requirements 17 and 18 as partially fulfilled. On the one hand, we have verified in the black-box tests that the system also processes invalid requests in a reliable manner. In the evaluation just discussed, we also demonstrated that most of the system's functions were implemented in an efficient and scalable manner with respect to gas costs and transaction time. We want to point out that especially the intraday market seems to be extremely scalable with an increasing number of users. However, we would also like to mention that we have observed scaling problems regarding the day-ahead market in the provided implementation. Under the assumption of operation in the Ethereum mainnet, problems arise primarily during the accept offer operations and the execution of the payment process. One of the limiting factors that restrict the throughput of the system is the block time and the block gas limit of the Ethereum mainnet blockchain. It depends on the actual market dynamics of the users, whether the achieved and reported throughput is sufficient. If this is not the case, we recommend the use of an alternative blockchain technology with higher potential throughput, such as Hyperleder Fabric. Furthermore, we have shown that for the Ethereum mainnet, the monetary transaction costs in Euro are very cost-intensive due to the current high ETH price. This puts into question the economic efficiency of the developed market system. The use of a private blockchain, in which no monetary transaction costs have to be paid, could solve this problem. However, a private blockchain is usually not tamper-proof which requires all market participants to trust the operator of the private blockchain.

In conclusion, in this chapter, we have evaluated the developed system in terms of efficiency and scalability by performing a market simulation. To this end, we used the gas costs and transaction time of the executed market operations as metrics for the evaluation. We have found that primarily the intraday market variant is implemented efficiently and scalable. However, for the day-ahead variant, we observed scalability problems when using the Ethereum mainnet blockchain. In general, we can state that the respective characteristics of the used blockchain technology limit the possible throughput and the scalability of the developed system. In particular, we would like to mention the block gas limit and the block time as limiting factors. It depends on the market dynamics of the users, whether the throughput of the Ethereum blockchain is sufficient or whether an alternative blockchain technology needs be employed for production operation.

Furthermore, we have shown in the chapter that the system fulfills all functional requirements. Besides, we discussed the implementation of the non-functional requirements by discussing the system properties. On the one hand, we found that the system is secure in terms of confidentiality and data integrity. Additionally, we can state that our system protects the privacy of users and their trading data by enabling pseudonymous usage of the markets.

7 Conclusion and Final Thoughts

In this last chapter, we will summarize the thesis and review the main findings in a concluding way. To complete the thesis, we will also give a short outlook for possible future work.

7.1 Conclusion

In this thesis, blockchain-based intraday and day-ahead operating reserve markets were presented, on which energy grid operators and operators of local distributed energy resources can trade energy flexibilities in the schedulable energy production and consumption of their resources. Since, in return, operators of DERs benefit from monetary advantages, the use and acquisition of these climate-friendly energy resources could be promoted simultaneously. In this thesis, all aspects of the proposed markets were implemented by utilizing the concept of general-purpose smart contracts. Since deployed smart contracts cannot be modified, we have defined the markets formally. This is intended to ensure that all details of the markets are described in an interpretation-free manner so that the requirements for the smart contracts can be defined precisely, thus avoiding later modifications.

Based on the abstract and formal definition of the markets, we have developed a corresponding smart-contract-based system architecture, in which all system components are provided on the blockchain. In addition to the intraday and day-ahead based trading of the operating reserve, we have defined a semi-automated payment process. For this purpose, we employ a proprietary and ERC-20 compatible cryptocurrency called *Smart Energy Coin*, which enables the payment process based on a presented escrow mechanism and available smart meter readings. The latter is provided to the system by the accredited *smart meter manager* and assigned to resources. Furthermore, in collaboration with a governmental institution named *data and grid authority*, we also managed to guarantee the integrity of all market participants in the system. Thereby we assume a X.509 based PKI already established by the DGA and demonstrate that by utilizing cryptographic procedures based on this PKI, it is possible to guarantee the integrity of all actors. Additionally, we utilized this PKI to design the markets in a privacy-aware manner in which users do not need to reveal their real-world identity or their DERs. In concrete terms, we enabled pseudonymous access to the markets. Therefore we can guarantee real-world users' and DERs' integrity while protecting their privacy in the presented market mechanisms. Hence, we enabled the implementation of free, protected, non-discriminatory, and privacy-aware blockchain-based operating reserve markets.

In this thesis, we also developed a prototype based on the theoretical concept by using the Ethereum blockchain. Thus we have demonstrated that it is feasible to realize a concrete technical implementation of the theoretically discussed short-term operating reserve markets. During the realization of the prototype, we also developed black-box tests to verify that the presented prototype satisfies all requirements identified in the thesis. Additionally, we want to highlight that our system

benefits from using the blockchain technology. Concrete, there is no central authority in our system, which the users have to trust for any trading processes. This enables a trustless operation of the markets. Furthermore, the data stored in the system is tamperproof by using the Ethereum blockchain. Thus we conclude that the presented system is secure in terms of confidentiality and data integrity. Besides, the decentralized data administration and processing of transactions in the Ethereum network renders our system more resilient, robust, and fault-tolerant.

Finally, we evaluated the developed prototype in terms of efficiency and scalability and observed that especially the intraday market variant was implemented in an efficient and scalable manner. In contrast, we observed scaling problems of the day-ahead market regarding the accept offer process and the execution of the payment process. In general, we can state that the respective characteristics of the used blockchain technology limit the possible throughput and the scalability of the developed system. Especially the block gas limit and the block time needs to be mentioned as limiting factors. It depends on the market dynamics of the users, whether the throughput of the developed system in combination with the Ethereum blockchain is sufficient or whether an alternative blockchain technology needs to be employed for production operation.

In conclusion, in the present thesis, we have demonstrated that it is technically feasible to implement short-term intraday and day-ahead operating reserve markets by utilizing the Ethereum blockchain. However, we also found that the technology is still in the early stages, making the development challenging due to missing features and standardization. Furthermore, we have shown that for the Ethereum mainnet, the monetary transaction costs of up to 732.23€ are very cost-intensive. This puts into question the economic efficiency of the developed market system.

7.2 Future work

As future work, the proposed system could be extended to provide additional features. As an extension of the present thesis, it could be investigated to what extent the concrete network topology of the energy grid can be modeled in the system. At the moment, all grid operators are authorized to accept all offers available on the markets and thus to acquire the advertised operating reserve. However, for the real deployment of operating reserve, the geolocation of a DER is important. In particular, it is essential to which subgrid operated by which GO a DER is assigned since the available operating reserve is to be employed there for grid stability. Therefore it should be investigated whether the network topology and the geolocation of DERs can be integrated into the system. Based on this, GOs should be restricted to accept only operating reserve from DERs assigned to their grids.

Furthermore, a field study with real users could be conducted to determine the concrete market dynamics of the users. Based on this, it could be evaluated whether the analyzed throughput is sufficient when using the Ethereum blockchain. In this context, it could also be examined whether the analyzed gas costs can be reduced, and thus, the efficiency and scalability of the proposed system increased. Alternatively, the system could be implemented and evaluated using other blockchain technologies in order to evaluate whether increased efficiency, scalability, and throughput can be achieved. In this context, the use of Hyperledger Fabric [8] might be advantageous.

Finally, providing a graphical user interface could support market participants in using the developed platform by increasing the usability, which could contribute to the acceptance of the markets.

List of Figures

List of Tables

List of Abbreviations

ABI Application Binary Interface 67

CA Certificate Authority 14, 15, 20
CLI Command Line Interface 64, 67

DER Distributed Energy Resource 1–3, 17, 19–21, 23–26, 28–30, 36, 37, 40, 41, 43–45, 48–60, 62, 66, 69, 71, 76, 77, 90
DGA Data and Grid Authority 19–26, 28, 36, 37, 41, 43, 47–50, 59, 60, 62, 63, 71
DPoS Delegated Proof Of Stake 8
DSO Distribution System Operator 15, 19

EEX European Energy Exchange 15
ETH Ether 10, 75
EVM Ethereum Virtual Machine 10, 65, 73, 74, 83

FCFS First Come, First Serve 27, 30

GO Grid Operators 1, 19, 21, 25, 27–32, 36, 41, 43–45, 53–58, 60–62, 69, 74, 76, 79, 81

HTTP Hypertext Transfer Protocol 65, 68
HTTPS Hypertext Transfer Protocol Secure 14

ICT Information and Communications Technology 1

JSON JavasScript Object Notation 67

kW Kilo Watt 29, 31
kWh Kilo Watt Hours 26

PBFT Practical Byzantine Fault Tolerance 8
PKCS Public-Key Cryptography Standards 13
PKI Public Key Infrastructure 2, 3, 14, 20, 21, 66

PoS Proof Of Stake 7–9
PoW Proof Of Work 6, 7, 9

RO Resource Operator 19, 21, 23–25, 27–32, 36, 40, 41, 43–45, 48–50, 52–55, 58, 60–62, 69, 74, 76, 77, 79, 90
RSA Rivest-Shamir-Adleman 4, 12, 13, 15, 20, 21, 66

SEC Smart Energy Coin 26–29, 32, 36, 42, 45–48, 50, 53–55, 57, 60
SMM Smart Meter Manager 26, 27, 30, 31, 36, 41, 43, 50, 51, 59, 71

TLS Transport Layer Security 14
TSO Transmission System Operator 15–17, 19, 90

UML Unified Modeling Language 68, 70, 91

Bibliography

[1] BMWi, *Systemsicherheit und Systemdienstleistungen*, de. [Online]. Available: https://www.bmwi.de/Redaktion/DE/Artikel/Energie/netz-und-systemsicherheit.html (visited on 07/05/2020).

[2] U. Förstner and S. Köster, "Klima und Energie," de, in *Umweltschutztechnik*, U. Förstner and S. Köster, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 159–218, ISBN: 978-3-662-55163-9. [Online]. Available: https://doi.org/10.1007/978-3-662-55163-9_4 (visited on 07/30/2020).

[3] BMWi, *Was ist eigentlich Regelenergie?* de. [Online]. Available: https://www.bmwi-energiewende.de/EWD/Redaktion/Newsletter/2017/12/Meldung/direkt-erklaert.html (visited on 07/05/2020).

[4] Bundesnetzagentur, *Kohleausstieg*, de. [Online]. Available: https://www.bundesnetzagentur.de/DE/Sachgebiete/ElektrizitaetundGas/Unternehmen_Institutionen/Kohleausstieg/kohleausstieg_node.html (visited on 12/07/2020).

[5] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," en, p. 9, Oct. 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf (visited on 05/21/2020).

[6] F. Tschorsch and B. Scheuermann, "Bitcoin and Beyond: A Technical Survey on Decentralized Digital Currencies," *IEEE Communications Surveys Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016, ISSN: 1553-877X.

[7] V. Buterin, *A Next-Generation Smart Contract and Decentralized Application Platform*, Nov. 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper (visited on 06/23/2020).

[8] *Hyperledger Fabric*, en-US. [Online]. Available: https://www.hyperledger.org/projects/fabric (visited on 07/01/2020).

[9] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, Jun. 2017, pp. 557–564.

[10] C. Cachin and M. Vukolić, "Blockchain Consensus Protocols in the Wild," *arXiv:1707.01873 [cs]*, Jul. 2017, arXiv: 1707.01873. [Online]. Available: http://arxiv.org/abs/1707.01873 (visited on 12/15/2020).

[11] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," en, in *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed., ser. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1988, pp. 369–378, ISBN: 978-3-540-48184-3.

[12] G. Wood, "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER," en, 2014, Edition: d6ff64f - 2019-06-13. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf (visited on 06/24/2020).

[13] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba, "A Taxonomy of Blockchain-Based Systems for Architecture Design," in *2017 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2017, pp. 243–252.

[14] J. R. Douceur, "The Sybil Attack," en, in *Peer-to-Peer Systems*, P. Druschel, F. Kaashoek, and A. Rowstron, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2002, pp. 251–260, ISBN: 978-3-540-45748-0.

[15] *Peercoin*, en. [Online]. Available: https://www.peercoin.net/ (visited on 12/16/2020).

[16] V. Buterin and V. Griffith, "Casper the Friendly Finality Gadget," *arXiv:1710.09437 [cs]*, Jan. 2019, arXiv: 1710.09437. [Online]. Available: http://arxiv.org/abs/1710.09437 (visited on 12/16/2020).

[17] *The Eth2 upgrades*, en. [Online]. Available: https://ethereum.org (visited on 12/16/2020).

[18] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," en, p. 14,

[19] N. Teslya and I. Ryabchikov, "Blockchain Platforms Overview for Industrial IoT Purposes," in *2018 22nd Conference of Open Innovations Association (FRUCT)*, ISSN: 2305-7254, May 2018, pp. 250–256.

[20] N. Szabo, *The Idea of Smart Contracts*, en, 1997. [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html (visited on 07/01/2020).

[21] K. Christidis and M. Devetsikiotis, "Blockchains and Smart Contracts for the Internet of Things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016, ISSN: 2169-3536.

[22] *Solidity — Solidity 0.5.16 documentation*, en. [Online]. Available: https://docs.soliditylang.org/en/v0.5.16/index.html (visited on 11/24/2020).

[23] *The Linux Foundation*, en-US. [Online]. Available: https://www.linuxfoundation.org/ (visited on 12/17/2020).

[24] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," en, in *Proceedings of the Thirteenth EuroSys Conference*, Porto Portugal: ACM, Apr. 2018, pp. 1–15, ISBN: 978-1-4503-5584-1. [Online]. Available: https://dl.acm.org/doi/10.1145/3190508.3190538 (visited on 12/15/2020).

[25] *Java Programming Language*, en. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html (visited on 12/17/2020).

[26] *The Go Programming Language*, en. [Online]. Available: https://golang.org/ (visited on 12/17/2020).

[27] *JavaScript | MDN*, en. [Online]. Available: https://developer.mozilla.org/de/docs/Web/JavaScript (visited on 12/17/2020).

[28] *Docker*, en. [Online]. Available: https://www.docker.com/ (visited on 12/17/2020).

[29] *Corda | Open Source Blockchain Platform for Business*, en. [Online]. Available: https://www.corda.net (visited on 12/17/2020).

[30] *Tendermint*, en-US. [Online]. Available: https://tendermint.com (visited on 12/17/2020).

[31] J. Katz and Y. Lindell, *Introduction to modern cryptography*, eng, Second edition, ser. Chapman & Hall/CRC cryptography and network security. Boca Raton London New York: CRC Press, Taylor & Francis Group, 2015, OCLC: 900419026, ISBN: 978-1-4665-7026-9.

[32] W. Diffie and M. Hellman, "New directions in cryptography," en, *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976, ISSN: 0018-9448. [Online]. Available: http://ieeexplore.ieee.org/document/1055638/ (visited on 12/09/2020).

[33] C. Paar, J. Pelzl, and B. Preneel, *Understanding cryptography: a textbook for students and practitioners*, eng, 2nd corrected printing. Berlin: Springer, 2010, OCLC: 845804174, ISBN: 978-3-642-04101-3 978-3-642-04100-6 978-3-642-44649-8.

[34] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," en, p. 15,

[35] B. Kaliski <kaliski_burt@emc.com>, *RFC5208 - Public-Key Cryptography Standards (PKCS) #8*, en. [Online]. Available: https://tools.ietf.org/html/rfc5208 (visited on 12/14/2020).

[36] *ITU-T*, en. [Online]. Available: https://www.itu.int/ITU-T/recommendations/rec.aspx?rec=X.509 (visited on 12/09/2020).

[37] 14:00-17:00, *ISO/IEC 9594-8:2017*, en. [Online]. Available: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/25/72557.html (visited on 12/09/2020).

[38] T. Howes, S. Kille, and W. Yeong, *X.500 Lightweight Directory Access Protocol*, en. [Online]. Available: https://tools.ietf.org/html/rfc1487 (visited on 12/12/2020).

[39] BMWi, *Ein Stromnetz für die Energiewende*, de. [Online]. Available: https://www.bmwi.de/Redaktion/DE/Dossier/netze-und-netzausbau.html (visited on 07/07/2020).

[40] *EEX*, de. [Online]. Available: https://www.eex.com/en/ (visited on 07/05/2020).

[41] D. R. Graeber, "Handel mit Strom aus erneuerbaren Energien," de, in *Handel mit Strom aus erneuerbaren Energien: Kombination von Prognosen*, D. R. Graeber, Ed., Wiesbaden: Springer Fachmedien Wiesbaden, 2014, pp. 7–58, ISBN: 978-3-658-03642-3. [Online]. Available: https://doi.org/10.1007/978-3-658-03642-3_2 (visited on 06/05/2020).

[42] C. Eid, P. Codani, Y. Perez, J. Reneses, and R. Hakvoort, "Managing electric flexibility from Distributed Energy Resources: A review of incentives for market design," *Renewable and Sustainable Energy Reviews*, vol. 64, pp. 237–247, Oct. 2016, ISSN: 1364-0321. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364032116302222 (visited on 06/12/2020).

[43] M. Paulus and F. Borggrefe, "The potential of demand-side management in energy-intensive industries for electricity markets in Germany," *Applied Energy*, The 5th Dubrovnik Conference on Sustainable Development of Energy, Water and Environment Systems, held in Dubrovnik September/October 2009, vol. 88, no. 2, pp. 432–441, Feb. 2011, ISSN: 0306-2619. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306261910000814 (visited on 06/13/2020).

[44] P. Palensky and D. Dietrich, "Demand Side Management: Demand Response, Intelligent Energy Systems, and Smart Loads," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 3, pp. 381–388, Aug. 2011, ISSN: 1551-3203.

[45] R. Chitchyan and J. Murkin, "Review of Blockchain Technology and its Expectations: Case of the Energy Sector," *arXiv:1803.03567 [cs]*, Mar. 2018, arXiv: 1803.03567. [Online]. Available: http://arxiv.org/abs/1803.03567 (visited on 06/05/2020).

[46] A. Goranović, M. Meisel, L. Fotiadis, S. Wilker, A. Treytl, and T. Sauter, "Blockchain applications in microgrids an overview of current projects and concepts," in *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, Oct. 2017, pp. 6153–6158.

[47] J. Horta, D. Kofman, D. Menga, and A. Silva, "Novel market approach for locally balancing renewable energy production and flexible demand," in *2017 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, Oct. 2017, pp. 533–539.

[48] C. Pop, T. Cioara, M. Antal, I. Anghel, I. Salomie, and M. Bertoncini, "Blockchain Based Decentralized Management of Demand Response Programs in Smart Energy Grids," en, *Sensors*, vol. 18, no. 1, p. 162, Jan. 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/1/162 (visited on 06/05/2020).

[49] P. Danzi, S. Hambridge, Č. Stefanović, and P. Popovski, "Blockchain-Based and Multi-Layered Electricity Imbalance Settlement Architecture," in *2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, Oct. 2018, pp. 1–7.

[50] F. Blom and H. Farahmand, "On the Scalability of Blockchain-Supported Local Energy Markets," in *2018 International Conference on Smart Energy Systems and Technologies (SEST)*, Sep. 2018, pp. 1–6.

[51] A. Nieße, N. Ihle, S. Balduin, M. Postina, M. Tröschel, and S. Lehnhoff, "Distributed ledger technology for fully automated congestion management," *Energy Informatics*, vol. 1, no. 1, p. 22, Oct. 2018, ISSN: 2520-8942. [Online]. Available: https://doi.org/10.1186/s42162-018-0033-3 (visited on 06/12/2020).

[52] F. Lombardi, L. Aniello, S. De Angelis, A. Margheri, and V. Sassone, "A Blockchain-based Infrastructure for Reliable and Cost-effective IoT-aided Smart Grids," en, in *Living in the Internet of Things: Cybersecurity of the IoT - 2018*, London, UK: Institution of Engineering and Technology, 2018, 42 (6 pp.)–42 (6 pp.) ISBN: 978-1-78561-843-7. [Online]. Available: https://digital-library.theiet.org/content/conferences/10.1049/cp.2018.0042 (visited on 01/16/2020).

[53] M. Mylrea and S. N. G. Gourisetti, "Blockchain for smart grid resilience: Exchanging distributed energy at speed, scale and security," in *2017 Resilience Week (RWS)*, Sep. 2017, pp. 18–23.

[54] K. Nakayama, R. Moslemi, and R. Sharma, "Transactive Energy Management with Blockchain Smart Contracts for P2P Multi-Settlement Markets," in *2019 IEEE Power Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, ISSN: 2167-9665, Feb. 2019, pp. 1–5.

[55] *TenneT unlocks distributed flexibility via blockchain*, en. [Online]. Available: https://www.tennet.eu/news/detail/tennet-unlocks-distributed-flexibility-via-blockchain/ (visited on 06/19/2020).

[56] *Gridchain*, en. [Online]. Available: https://enerchain.ponton.de/index.php/16-gridchain-blockchain-based-process-integration-for-the-smart-grids-of-the-future (visited on 06/19/2019).

[57] T. Hansen and D. E. E. 3rd, *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*, en. [Online]. Available: https://tools.ietf.org/html/rfc6234#section-4.1 (visited on 10/15/2020).

[58] "ISO/IEC/IEEE 29148:2018 - International Standard - Systems and software engineering – Life cycle processes – Requirements engineering," *ISO/IEC/IEEE 29148:2018(E)*, pp. 1–104, Nov. 2018, Conference Name: ISO/IEC/IEEE 29148:2018(E).

[59] S. Bradner <sob@harvard.edu>, *RFC 2119*, en. [Online]. Available: https://tools.ietf.org/html/rfc2119 (visited on 12/04/2020).

[60] *EIP-20: ERC-20 Token Standard*, en. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20 (visited on 11/09/2020).

[61] *Ganache CLI*, original-date: 2016-01-08T17:40:36Z, Nov. 2020. [Online]. Available: https://github.com/trufflesuite/ganache-cli (visited on 11/24/2020).

[62] *EIP-2384: Muir Glacier Difficulty Bomb Delay*, en. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2384 (visited on 11/24/2020).

[63] *Solc*, en. [Online]. Available: https://www.npmjs.com/package/solc (visited on 11/24/2020).

[64] *Truffle Suite*, en. [Online]. Available: https://trufflesuite.com/truffle (visited on 11/24/2020).

[65] *Web3.js - Ethereum JavaScript API*, en. [Online]. Available: https://web3js.readthedocs.io/en/v1.2.1/ (visited on 11/24/2020).

[66] *Mocha*, en. [Online]. Available: https://mochajs.org/ (visited on 11/24/2020).

[67] *Chai*, en. [Online]. Available: https://www.chaijs.com/ (visited on 11/24/2020).

[68] *Node-forge*, en. [Online]. Available: https://www.npmjs.com/package/node-forge (visited on 11/24/2020).

[69] Node.js, *Node.js*, en. [Online]. Available: https://nodejs.org/en/ (visited on 11/24/2020).

[70] *Commander.js*, en. [Online]. Available: https://www.npmjs.com/package/commander (visited on 11/24/2020).

[71] *Npm*, en. [Online]. Available: https://www.npmjs.com/ (visited on 11/24/2020).

[72] *Contract ABI Specification — Solidity 0.5.16 documentation*, en. [Online]. Available: https://docs.soliditylang.org/en/v0.5.16/abi-spec.html#index-0 (visited on 11/25/2020).

[73] *Go Ethereum*, en. [Online]. Available: https://geth.ethereum.org/ (visited on 11/25/2020).

[74] *Parity Ethereum Client - OpenEthereum*, en, Sep. 2018. [Online]. Available: https://www.parity.io/ethereum/ (visited on 11/25/2020).

[75] *Trinity*, en. [Online]. Available: https://trinity.ethereum.org/ (visited on 11/25/2020).

[76] *Nethermind - CLIENT*, en. [Online]. Available: https://nethermind.io/client (visited on 11/25/2020).

[77] *UML 2.5.1*, en. [Online]. Available: https://www.omg.org/spec/UML/ (visited on 11/25/2020).

[78] M. Marchesi, L. Marchesi, and R. Tonelli, "An Agile Software Engineering Method to Design Blockchain Applications," en, in *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia on ZZZ - CEE-SECR '18*, Moscow, Russian Federation: ACM Press, 2018, pp. 1–8, ISBN: 978-1-4503-6176-7. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3290621.3290627 (visited on 11/25/2020).

[79] *EIP-170: Maximum Contract Size*, en. [Online]. Available: https://github.com/ethereum/EIPs (visited on 11/25/2020).

[80] M. Bez, G. Fornari, and T. Vardanega, "The scalability challenge of ethereum: An initial quantitative analysis," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, ISSN: 2642-6587, Apr. 2019, pp. 167–176.

Appendices

Appendix 1: Source Code

In this appendix, we provide the source code of the implemented system to the reader. The provided source code not only contains the smart contracts and corresponding tests, but also the gathered evaluation data and scripts to visualize and analyze these.

The source code is provided as a compressed *zip* file via the cloud service *tubCloud* of the TU Berlin. The corresponding link and password are listed below.

In the root directory of the provided source code exists a *README* file, which contains all necessary instructions to run the system and the execute tests.

Password: mastersThesisBoerger2020

TubCloud Link: https://tubcloud.tu-berlin.de/s/Ky2MN7AS3f3e7dS