Open Blockchain-based Local Energy Market Simulation Platform

Master Thesis

Author: Niklas Reinhold (Student ID: 5377277)
Supervisor: Univ.-Prof. Dr. Wolfgang Ketter
Co-Supervisor: Philipp Kienscherf

Department of Information Systems for Sustainable Society
Faculty of Management, Economics and Social Sciences
University of Cologne

July 30, 2019

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 161 Abs. 1 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

Niklas Reinhold

Köln, den xx.xx.20xx

Abstract

[Abstract goes here (max. 1 page)]

Contents

List of Figures

List of Tables

Listings

1 Research Motivation

The generation from distributed renewable energy sources (RES) is constantly increasing (Mengelkamp, Gärttner, et al., 2018). In contrast to power plants which run by non-renewable fossil fuels, distributed RES produce energy in a decentralized and volatile way, which is hard to predict. These characteristics of the distributed RES challenge the current energy system (Ampatzis et al., 2014).

The existing electric grid is build for centralized generation by large power plants and the design of the current wholesale markets is not able to react in real-time to a significant amount of distributed RES (Mengelkamp, Gärttner, et al., 2018) (Ampatzis et al., 2014). Moreover, this way of energy generation is economically not ideal because of energy losses due to long physical distances between generation and consumption parties.

Therefore, new market approaches are needed, to successfully integrate the increasing amount of distributed RES (Mengelkamp, Notheisen, et al., 2018). A possible solution to the technical and market problems is Peer-to-Peer (P2P) energy trading in local energy markets (LEM) (Long et al., 2017).

LEM, also called microgrid energy markets, consist of small scale prosumers, consumers and a market platform which enables the trading of locally generated energy between the parties of a community. Due to the trading of locally generated energy within the related communities, LEM support sustainability and an efficient use of distributed renewable energy sources. Likewise, the need of expensive and inefficient transportation of energy through long physical distances can be reduced. The concept of LEM strengthens the self-sufficiency of communities and enables possible energy cost reductions. Moreover, profits remain within the communities whereby reinvestments in additional RES are promoted (Mengelkamp, Gärttner, et al., 2018).

However, P2P energy trading in LEM requires advanced communication and data exchanges between the different parties, which makes central management and operation more and more challenging. The implementation of LEM needs local distributed control and management techniques. (Andoni et al., 2019). Therefore, a new and innovative information communication technology (ICT) is required. A possible solution provides the emerging distributed ledger technology (DLT). It is designed to enable distributed transactions without a central trusted entity. Furthermore, an Ethereum-based blockchain allows the automated execution of smart contracts depending on vesting conditions, which suits the need of LEM for decentralized and autonomous market mechanisms. This offers new approaches and market designs. Accordingly, DLT can help addressing the challenges faced by decentralized energy systems. However, DLTs are not a matured technology yet and there are several barriers in using them, especially for researcher who do not have a technically background.

In addition, Ketter, Peters, Collins and Gupta introduce the approach of Competitive Benchmarking (CB) (Ketter et al., 2015). This approach describes a research method which faces a real world wicked problem that is beyond the capacity of a single discipline. This is realized by developing a shared paradigm which is represented in a concrete open simulation platform. In detail, it consists of the three principal elements *CB Alignment*, *CB Platform* and *CB Process*. The *CB Alignment* refers to the constant synchronization process between the shared paradigm and the wicked problem. Further, the *CB Platform* represents the medium, in which the shared paradigm is technically illustrated. In addition, the *CB Platform* provides the infrastructure for the third element *CB Process*. It describes the iterative development of new theories and design artifacts through independent researchers, which influence each other and improving their work in direct sight of each other.

Due to the plurality of involved parties and the interdisciplinary requirements for the implementation of new energy market approaches, the accessibility is of major importance. Therefore, the presence of such a open simulation platform depicting the shared paradigm of a LEM, would ensure the accessibility and could help to gain new valuable outcomes in the research field of LEM or new energy market designs in general.

Further, Guo, Koehler and Whinston developed a market-based optimization algorithm, which solves a distributed system optimization problem by self-interested agents iteratively trading bundled resources in a double auction market run by a dealer. The authors called it bundle trading market framework or short BTM (Guo et al., 2007). The central problem of the stated market-based optimization algorithm can interpreted as the welfare optimization of all participants in a LEM. The dealer, which runs the double auction market, maximizes the welfare through allowing agents to trade their preferable bundles of energy. Hence, the stated BTM implemented on basis of a blockchain as underlying ICT can depict the concept of a LEM.

Consequently, this research will bring all the introduced approaches together and develop an open blockchain-based LEM simulation, which enables the research approach based on the three stated elements of CB. The platform will be realized through the introduced optimization algorithm with a blockchain as the underlying ICT. That means, the smart contract takes the role of the market dealer and the self-interested agents represent the individual participants in a LEM. The focus of this paper will be on the implementation and software design of the open blockchain-based LEM simulation platform.

2 Literature Review

2.1 Blockchain-based Energy Markets

To start off, Mihaylov et al. were the first who adressed blockchain technology in energy markets. They present a new decentralized digital currency with the aid of which prosumers trade locally produced renewable energy (Mihaylov et al., 2014). In their introduced concept, the generation and consumption of renewable energy is direct transferable into virtual coins. However, the market value of the virtual currency is determined centrally by the distributed system operator. Further, Al Kawasmi et al. introduce a blockchain-based model for a decentralized carbon emission trading infrastructure (Al Kawasmi et al., 2015). Their model based on the bitcoin protocol and focus on privacy and system security goals. Besides, they provide a solution to the problem of anonymous carbon emission trading. Equally, Aitzhan and Svetinovic address the issue of transaction security in decentralized smart grid energy trading and implemented a proof-of-concept for a blockchain-based energy trading system including anonymous encrypted messaging streams (Aitzhan & Svetinovic, 2018). Concluding, they show that blockchains enable the implementation of decentralized energy trading and that the degree of privacy and security is higher than in traditional centralized trading platforms. Furthermore, Sikorski et al. present an proof-of-concept where a blockchain enables machine-to-machine (M2M) interactions depicting an M2M energy market (Sikorski et al., 2017). They pointed out that the blockchain technology has significant potential to support and enhance the 4th industrial revolution. Moreover, Mengelkamp et al. reveal the concept of a blockchain-based local energy market without the need of a thrusted third entity (Mengelkamp, Gärttner, et al., 2018). In addition, they deduce seven market components as a framework for building efficient microgrid energy markets. Consequently, the Brooklyn Microgrid project is introduced and evaluated according the market components. As a result, the Brooklyn Microgrid also shows that blockchains are a suitable technology to implement decentralized microgrid energy markets, though current regulation does not allow to run such a local energy markets in most of the countries. Later on, Mengelkamp et al. present an initial proof-of-concept of a simple blockchain-based concept, market design and simulation of a local energy market consisting of hundred households (Mengelkamp, Notheisen, et al., 2018). Finally, they conclude that the real-life realisation and technological limitations of such blockchain-based market approaches need to be investigated by further research. In addition, it is mentioned that regulatory changes will play an important role in the future of blockchain-based LEM.

2.2 Distributed Resource Optimization

To begin with, Fan et al. outline a new approach for the development of an information system which can be used for the problem of a supply chain. The concept demonstrate a decentralized decision making process that is realized through the design of a market-based coordination system which incite the pariticipants to act in a way that is beneficial to the overall systems (Fan et al., 2003). Further, Guo et al. revive this concept and develop a market-based decomposition method for decomposable linear systems, that can be easily implemented to support real-time optimization of dsitributed systems (Guo et al., 2007). They prove that the system optimality can be achieved under a dynamic market-trading algorithm in a finite number of trades. Moreover, the outlined alogrithm can be operated in synchronous and as well in asynchronous environments. Later on, Guo et al. extend their stated concept to a dynamic, asynchronous internet market environment (Guo et al., 2012). Additionally, they examine how various market design factors like dealer inventory policies, market communication patterns, and agent learning strategies affect the computational market efficiency and implementation. Finally, also this time, Guo et al. prove finite convergence to an optimal solution under all these different schemes.

2.3 Competitive Benchmarking

Firstly, March and Smith describe a two dimensional framework for research in IS (March & Smith, 1995). The two dimensions can be distinguished into behavioral-science and design-science. The behavioral-science paradigm is based on explaining or predicting human or organizational behavior. Whereas, the design-science paradigm is based on broad types of outputs produced by design research. It strives to extend the boundaries of human and organizational capabilities through the creation of new artifacts.

Furthermore, the research framework presented by Hevner et al. combines the both stated IT research paradigms and illustrates the interaction between these two (Hevner et al., 2008). They present that the technology and behavior are inseparable in an information system. Therefore, they argue that considering the complementary research cycle between design-science and behavioral-science is crucial to address fundamental problems faced in the productive application of information technology.

In addition, Ketter, Peters, Collins and Gupta introduce the IS research approach called Competitive Benchmarking (CB), which includes varios basic principles of the design science approach of Hevner et al. (Ketter et al., 2015). This IS research concept is designed for so called wicked problems and address the problems and needs of interdisciplinary research communities. CB focus on the interconnection of problems at the same and different levels to imitate real world. In repeated competitions, in which individual teams compete against each other, the developed environment and models can be tested and evaluated. This results in a diversity of outcoming designs, which are difficult to achieve in traditional design science frameworks.

3 Research Design

This research is inspired by the stated CB research approach and focus on the development of the second CB element, an open simulation platform depicting the shared paradigm of a LEM. The proposed design is presented in Figure 1. In contrast to the inital CB research design, the third element *CB Process* differs. The paltform enables research groups to design and test their artifacts in the field of the shared paradigm, using the developed platform. However, the designed artifacts in the form of autonomous software agents do not compete against each other in competitions. Rather, research groups are able to design and develop the behavior of all agents and the autonomous market mechanism. Hence, it is also possible to analyze and evaluate the designed artifacts and use them as a benchmark to compare different designs.

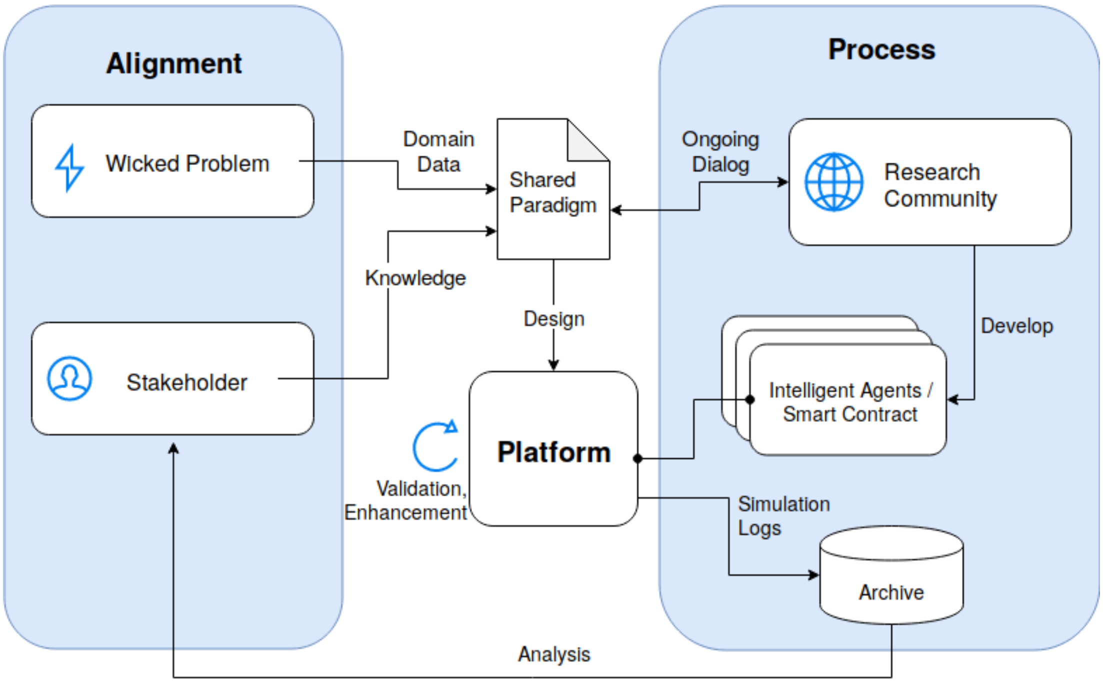

Figure 1: Research Design following Ketter, Peters, Collins and Gupta

According to the IS Design Science principles introduced by Hevner, March, Park, and Ram (Hevner et al., 2008), the presented research design produce a viable artifact in the form of an open simulation platform, which depict a relevant real-world wicked problem. Due to the outcoming data produced by the simulation platform, it is possible to develop methods to evaluate the utility, quality and efficacy of the artifact. Moreover, the research provide a varifable contribution through the artifact, the open simulation platform, itself. The platform enables the investigation of possible solutions of unsolved problems and further, it removes the technical barriers for the use of the complex distributed ledger technology.

Besides, the implementation of the platform is based on an appropriate selection of techniques. As stated in 2.1, blockchains are a suitable technology to implement decentralized microgrid energy markets. In addition, the in 2.2 introduced distributed optimization algorithm achieve evidentially system optimality under a dynamic market-trading algorithm. Therefore, this research relies upon the application of rigorous methods and comply the requirement of the fifth guideline by Hevner et al. (2008). In addition, the platform enables the iterative search for an optimal design through the comparison of different produced solutions and is valuable to technology-oriented as well as management-oriented audiences. As a result, these research design also fulfil the seven research guidlines of the research framework introduced by Hevner et al. (2008).

4 Expected Contribution

This research provides two different contributions. First, the fully decentralization of the introduced distributed optimization algorithm (Guo et al., 2007). The bundle-trading market grant access of any given trade to the market dealer. This, again, necessitates trust of the agents that it will use those resources according to the over-arching organizational goal. Due to the implementation of the market dealer by a smart contract, the technical implementation is public accesible. Moreover, all transactions of the market dealer to allocate the resources are transparent. Therefore, the behavior of the market dealer is comprehensible for every participant. Furthermore, it provides a high degree of security due to cryptographic encryption methods which are essential parts of the blockchain. Second, the open simulation platform itself. From the scientific perspective, the platform is relevant due to the removal of existing technical barriers for practicioners in using complex distributed ledgers technologies. Therefore the platform facilitates researchers without a deep technically background to use those DLTs and incentivises to design and test their artifacts by using this platform. On the other hand, from a business perspective, this research is relevant to policy makers and energy suppliers. The platform allows stakeholders to get a better understanding of the dynamics of decentralized LEM and enables the testing and evaluating of policy options and implications.

5 Local Energy Markets

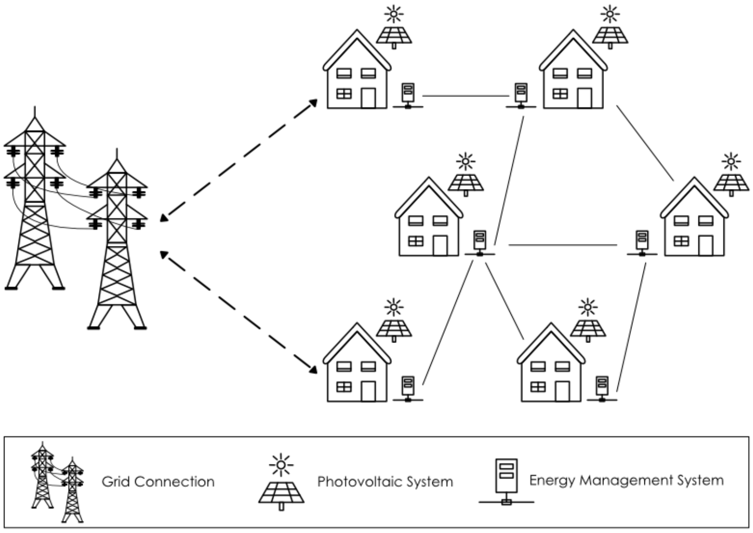

Figure 2: Example of Microgrid setup

To begin with, we revive the stated problem of a successful integration of the increasing amount of distributed renewable energy sources (RES), as already adressed in section 1. Due to the centralized generation by large power plants and the current design of the wholesale markets, the existing grid is not suitable designed to react in real-time to a significant growing amount of distributed RES (Mengelkamp, Gärttner, et al., 2018) (Ampatzis et al., 2014).

In order to successfully integrate and use these RES, new approaches are necessary (Mengelkamp, Notheisen, et al., 2018). A possible solution to those technical and market problems is Peer-to-Peer (P2P) energy trading in local energy markets (LEM) (Long et al., 2017).

This section will explain the general concept of local energy markets and present market components for building efficient LEM.

In the traditional centralized energy system, large power plants that operate according to a centralized coordination mechanism, supply a large amount of customer with energy, which are located within a wide area (for example a country or a state) (Mengelkamp, Gärttner, et al., 2018).

In contrast, decentralized energy systems consists of small-scale energy generators that are only used by a small number of people and located close to the energy consumption point (Mengelkamp, Gärttner, et al., 2018). Those local energy markets provide a market platform, market mechanism and market access for small-scale prosumer and consumer, to trade locally generated energy within their community. In this case, a community presents a group of geographically and socially close energy agents. Moreover, all participants buy or sell energy directly with each other by using the provided market platform without intermediation by conventional energy suppliers (Zhang et al., 2017).

In addition, if prosumer have a surplus in electricity, they have the opportunity to curtail it, store it in a energy storage device or export it back to the main power grid (Zhang et al., 2017). Considering, in the traditional centralized energy system, trading of energy is mainly unidirectional. As stated above, the electricity is usually transmitted from large-scale power plants to consumers over long distances, while the cash-flow goes the opposite way. On the contrary, the P2P energy trading in a LEM encourages multidirectional trading within a local geographical community (Zhang et al., 2017).

Due to this, LEM promote the consumption of energy close to its generation and, therefore, support sustainability and the efficient use of local resources (Mengelkamp, Gärttner, et al., 2018). Furthermore, local energy markets enable (near) real-time pricing and facilitate a local balance of supply and demand (Mengelkamp, Notheisen, et al., 2018).

Further, as stated by Mengelkamp, Gärttner, et al. (2018), the participants of a LEM do not necessarily have to be physically connected. A virtual microgrid describes the aggregated control of multiple energy producers, prosumers and consumers in a virtual community. Further, the revenue potential can be increased significantly by expanding a physical microgrid to inlude virtual participants.

Referring to Jimeno et al. (2011), a LEM has a specific characteristic that distinguish it from other aggregation of DER systems. A microgrid has the opportunity to operate either connected to the main grid or islanded from it. In other words, this allows a LEM to run disconnected from the main grid, in case it fails or the power quality is not satisfactory. Thus, participants of a LEM have a higher quality of supply for the loads within it. Additionally, it offers a way of obtaining cheaper and cleaner energy for all participants, if elements operated in a LEM taking into account by economic and emission policies.

Moreover, P2P energy trading in LEM requires advanced communication and data exchanges between the different parties, which makes central management and operation more and more challenging. The implementation of LEM needs local distributed control and management techniques. (Andoni et al., 2019). Zhang et al. (2017) stated that, P2P energy trading is often enabled by ICT-based online services. Moreover, Mengelkamp, Gärttner, et al. (2018) explain that the new and innovative blockchain technology as an emerging ICT, offers new opportunities for decentralized market designs. It is designed to enable distributed transactions without a central trusted entity. Accordingly, blockchain can help addressing the challenges faced by decentralized energy systems.

However, blockchain is not a matured technology yet and there are several barriers in using them, especially for researcher who do not have a technically background. Therefore, this research will use a blockchain in the LEM simulation platform as the underlying ICT and will give a detailed introduction to the blockchain technology in section 6. In conclusion, the developed simulation platform removes existing technical barriers for practicioners and enables the usage of blockchain technology for those.

5.1 Components of local energy markets

Mengelkamp, Gärttner, et al. (2018) developed in his research seven components for a efficient operation of blockchain-based local energy markets. This subsection will name and briefly illustrate each component. Further, the compliance of the developed open blockchain-based LEM simulation and the seven components will be examined later on in section [ref hinzufügen!!]

Microgrid setup (C1): In general, an explicit objective, a definition of the market participants and the form of the traded energy must be well defined. A LEM can have different, often contradictory objectives. Especially in the design of the market mechanism, the implementation of the objective plays an important role. Next, a significant number of market participants is needed, who trade energy among each other. Moreover, a part of the market participants needs to be able to produce energy. Finally, the form of the traded energy must be described, for example electricity, heat or a combination of them. Additionally, the way of energy transportation must be specified. Will be the traditional energy grid used or a physical microgrid build.

Grid connection (C2:) The connection points to the superordinate main grid must be well defined. These points measure the energy flows towards the main grid and evaluate the performance of the LEM and can help to balance the energy generation and demand within it. Besides, you have to distinguish between a physical microgrid and a virtual microgrid. A physical microgrid bring along a power distribution grid and is able decouple from the main grid, whereas a virtual microgrid simply connects all participants over an information system (C3). Consequently, a virtual microgrid does not have the opportunity to physically decouple from the main grid. Nevertheless, to operate in island-mode extensively, a physical microgrid need a large amount of own energy generation capacity and flexibility to ensure supply security and robustness.

Information system (C3): In addition, all participants must be connected to each other and a market platform that monitor all operations must be provided. Therefore, a information system is needed, which should enable equal access for every market participant to avoid discrimination. With reference to Mengelkamp, Gärttner, et al. (2018), these requirements can be implemented by blockchain technology based on smart contracts.

Market mechanism (C4): Besides, a market mechanism implemented through the information system is necessary. This market mechanism implies the allocation of the market and the payment rules. Further, a clear bidding format should be defined. It follows that the main objective of the mechanism is to provide an efficient energy allocation by matching the buy and sell orders of the participants appropriately. Finally, this should happen in near real time granularity.

Pricing mechanism (C5): The market mechanism (C4) includes the pricing mechanism and supports the efficient allocation of energy supply and demand. The traditional energy price is composed in large parts of taxes and surcharges. On the contrary, in a LEM different fees come to bear, for example in case of a physical microgrid. Hence, RES typically have almost zero marginal cost, prosumer can price their energy above all appropriate taxes and fees to make profit. Thus, the energy price should linked to the availability of energy. In other words, a surplus of energy should lower the LEM energy price while a lack of energy inreases the market price. From an economic point of view, LEM are beneficial to their participants as long as the average energy pirce is lower than the external grid price.

Energy management trading system (C6): The task and goal of the EMTS is to automatically ensure energy supply for a respective market participant. Therefore, the EMTS needs access to the energy-related data of the participant, like the real time demand and supply. The EMTS uses this data to forecasts consumption and generation and develops a bidding strategy accordingly. Moreover, the EMTS trades the predicted amounts on the provided market platform and aims to maximize the revenue and minimize the energy costs. For this reason, the EMTS needs to have access to the market participant's blockchain account to be capable to automatically perform energy transactions.

Regulation (C7): It needs to determined how a LEM fit into the current energy policy and which market design is allowed, how taxes and fees are distributed and billed. Likewise, it needs to determined in which way the local market is integrated into the traditional energy market and energy supply system. All these emerging issues are specified by the legislative regulation

6 Blockchain

To begin with, in section 5 we stated that the implementation of a LEM needs local distributed control and management techniques. Furhter, we emphasises that the blockchain technology as an emerging ICT offers new opportunities for decentralized market designs and helps adressing the challenges faced by decentralized energy systems. In conclusion, this research will use a blockchain in the developed LEM simulation platform as the underlying ICT.

Hence, this chapter will give an overview of the general purpose, the contained components and the fundamental functionality of a blockchain.

6.1 General Purpose

In general, a blockchain can be described as a digital data structure that can be understand as a shared and distributed database, containing a continuous expanding and chronological log of transactions (Andoni et al., 2019). Besides, various types like digital transactions, data records and executables can be stored in this digital data structure. The data transmission in a blockchain is comparable with copying data from one computer to another. However, the resulting challenge is that the system needs to ensure that the data is copied just once (Andoni et al., 2019). For example, in the domain of cryptocurrencies, this is equal to sending a coin from one wallet to another. In this case, the system needs to validate that this coin is spended just once and there is no double-spending. A conventional solution for this problem is a third intermediary. To come back to the stated example, the third intermediary is represented by a traditional bank, which store, protect and continuously update the valid state of the ledger (Andoni et al., 2019). But, in some cases central management is not practicable or reasonable. Reasons for this are possible intermediary costs or a high degree of trust of the users into the intermediary who operates the system. Further, central management has a significant disadvantage because of a single point of failure. Hence, the centralized system is fragile to technical problems as well to external malicious attacks (Andoni et al., 2019). Consequently, the main reason of bockchain technologies is the removal of such third trusted intermediaries through a distributed network of various users, who cooperating together to verify transactions and protect the validity of the ledger

6.2 Components

This subsection covers the architectural design of a blockchain and presents the contained components in detail. Due to the plurality of the blockchain technologies, each of the technology slightly differs in design and components. The following explanations are oriented torwards the Ethereum blockchain implementation, which is also used as the underlying ICT to implement the open simulation platform.

6.2.1 World State

Referring to the *Yellow Paper* by Wood (2014), Ethereum can be seen as a transaction-based state machine. What does that mean? At the beginning, Ethereums state machine starting with a so called *"genesis state"*. This is analogous to a blank sheet. On this state, no transactions have happened on the network. Next, transactions are executed and the state of the Ethereum world changes into a new state. Further, transactions are executed incrementally and morph it into some final state. Consequently, the final state is accepted as the canonical version of the world of Ethereum and represents at any times the current state.

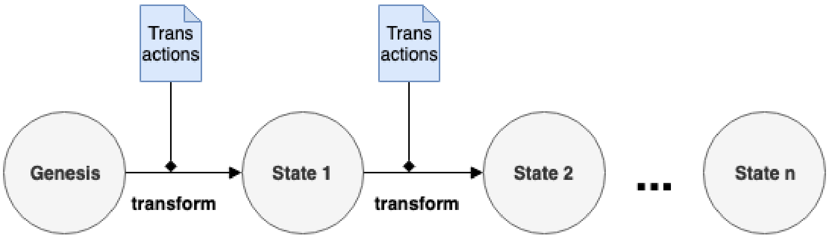

Figure 3: World State: Transition of States

In more detail, the world state arises out of a mapping of a key value pair for every account which exists on the Ethereum network (Wood, 2014). The key constitutes the address of an Ethereum account and the value presents the account's state, which contains detailed information of this account. However, the world state is not stored on the blockchain itself. This mapping is stored and maintained in a modified data structure called a *Merkle Patricia tree*. This tree is stored off-chain in a simple database backend (i.e. on a computer running an Ethereum client), also known as the *state database* in the Ethereum world (Wood, 2014). To get a better understanding of the operating principles of the blockchain, it is necessary to get an idea of how a *Merkle Patricia tree* works. A *Merkle Patricia tree* is a type of binary tree, which consists of a set of nodes. It has a large amount of *leaf nodes*, containing the underlying data. Further, a set of intermediate nodes, where each node is the hash of its two children, and finally, one single root node, representing the top of the tree which is also build out of its two child nodes (Buterin et al., 2013) (Wood, 2014). As mentioned before, the *leaf nodes* contain the stored data by splitting these data into chunks.

Afterwards, these chunks are splitted into buckets. Then, each bucket gets hashed and the same process repeats, traversing upwards the tree, until the total number of hashes remaining becomes only one and the root node is reached (*Merkling in Ethereum*, 2015).

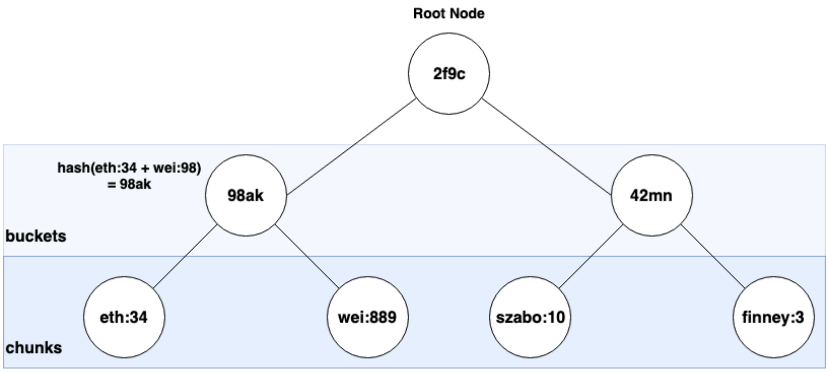

Figure 4: Example of a Patricia Merkle tree

Therefore, any change to the underlying data, stored in a *leaf node*, causes a change of the hash of the node. Each parent's node hash depend on the data of its children. Due to this, any change to the data of a child node causes the parent hash to change. This procedure repeats traversing upwards until the root node. Hence, any change to the data at the leaf nodes effects the root hash (*Modified Merkle Patricia Trie Specification*, 2015).

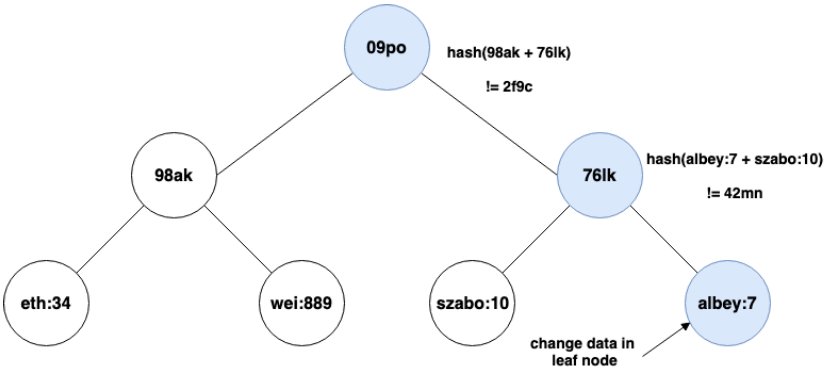

Figure 5: Example of a data change in a leaf node

Because of this characteristic, it is not necessary to compare the data of the entire tree. It is sufficient to compare the single root hash to ensure that all the data are the same. This property is very important, because it makes it possible to store only the hash of the root node to represent a state of the Ethereum world.

6.2.2 Block

As described in section 6.2.1, whenever transactions are executed, the state of the Ethereum world changes into a new state. Every state and the belonging transactions, which transform the prior state into the new state, is cumulated in a so called block. That means, states are represented by blocks. As you can see in figure 3, the history of the Ethereum world is a linkage of states, or in other words, blocks. That is where the name blockchain comes from. A blockchain is a sequence of blocks, which holds a complete list of transaction records (Zheng et al., 2017). Moreover, a block is a collection of different relevant informations and consists of the *block header* and the *block body*, which contains the list of transactions. Following *Ethereums Yellow Paper* by Wood (2014), the subsequent pieces of information are contained in the *block header*:

Parent Hash: This is the hash of the parents block's header. Therefore, every block points to his ancestor. Due to this contained attribute, a chain arises out of the single blocks.

Beneficiary: The miners address to which all block rewards from the successful mining of a block are transferred.

State Root: This is the hash of the root node of the state tree, after a block and its transactions are finalized. As mentioned in section 6.2.1, the state tree is the one and only global state in the Ethereum world. It is used as a secure unqiue identifier for the state and the state root node is cryptographically dependent on all internal state tree data.

Transactions Root: This is the hash of the root node of the transaction tree. This tree contains all transactions in the block body. In contrast to the state tree, there is a separate transactions tree for every block.

Receipts Root: Every time a transaction is executed, Ethereum generates a transaction receipt that contains information about the transaction execution. This field is the hash of the root node of the transactions receipt tree and like the transaction tree, there is a seperate receipt tree for every block.

Difficulty: This is a measure of how hard it was to mine this block – a quantity calculated from the previous block's difficulty and its timestamp

Number: This is a quantity equal to the number of blocks that precede the current block in the blockchain.

Gas Limit: This is a quantity equal to the current maximum gas expenditure per block. Each transaction consumes gas. The gas limit specifies the maximum gas that can be used by the transactions included in the block. It is a way to limit the number of transactions in a block.

Gas Used: This is a quantity equal to the total gas used in transactions in this block.

Timestamp: This is a record of Unix's time at this block's inception.

Nonce: This is an 8-byte hash that verifies a sufficient amount of computation has been done on this block. Further, it is a number added to a hashed block that, when rehashed, meets the difficulty level restrictions. The nonce is the number that blockchain miners are solving for.

So far, we introduced Ethereum as a transaction-based state machine, transforming one state into another through the execution of transactions. Further, we explained how these individual states are stored and emphasises the meaning of these storage. Moreover, we pointed out that an Ethereum state is represented by blocks and described all relevant informations contained in it. However, we didn't introduce how many transactions be part of a block and who build and validate a block? To answer this, we introduce the transaction object and his life cycle in depth.

6.2.3 Transaction

To start with, a transaction is the basic method for Ethereum accounts to interact with each other. Further, in the Ethereum world exists two different types of transactions. Those, which result in a so called *message call*, which can be seen as a traditional transaction, and those which result in a *contract creation* (Wood, 2014). Though, both different types of transaction share the following common attributes (Wood, 2014):

Nonce: This attribute is discontiguous of the block attribute. In contrast to the block attribute, the nonce of a transaction keep track of the total number of transactions that an account has executed.

Gas Price: As mentioned in section 6.2.2, each transaction consume gas. Gas can be seen as fees. This attribute presents the price per unit of gas for a transaction [1].

Gas Limit: This attribute is similiar to the block attribute of the same name. In this case, it is a quantity equal to the current maximum gas expenditure per transaction.

[1] A website to see current gas prices: https://ethgasstation.info/index.php

To: In case of a *message call*, this attribute contains the address of the recipient. Otherwise, in case of a *contract creation* this value remains empty.

Value: In case of a *message call*, this attribute contains the amount of Wei, which will be transferred to the recipient. In the case of a *contract creation*, this value contains the amount of Wei as a intial endowment of the contract.

6.2.4 Transaction Life Cycle

Next, the life cycle of a transaction will be outlined. We will take a simple *message call* as an example and will pass through the entire flow of how this transaction gets executed and permanently stored on the blockchain. During this process, many basic concepts of a blockchain will be introduced.

To begin with, someone want to send an abitrary amount of ether to someone else. Consequently, a transaction with the respective attributes, which are stated in section 6.2.3, will be created. As a first step, this transaction will be signed. That means, the one who wants to execute the transaction needs to proof the ownership of the account. Otherwise, someone else could execute a transaction on your behalf. The way to proof this is by signing the transaction with the corresponding private key of this account (*Medium, Life Cycle of Transactions*, 2017). Next, the signed transaction is submitted to the related Ethereum node. Then, the node will validate the signed transaction. After a successful validation, the node will send the transaction to it's peer nodes, who again send it to their peer nodes and so on (*Medium, Life Cycle of Transactions*, 2017). As soon as the transaction is broadcast to the network, the related node will declare a transaction id, which is the hash of the signed transaction and can be used to track the status of the transaction [2].

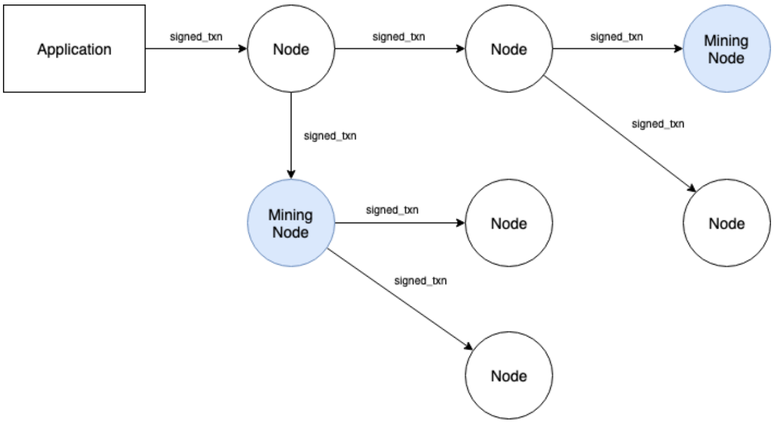

Figure 6: Node network demonstrating a transaction broadcast

[2] A Website to track transactions: https://etherscan.io/

Further, figure 6 describes a Ethereum network. As you can see, it contains a mix of miner nodes and non miner nodes. The non miner nodes can be distinguished into different types of nodes with different behavior, which are explained in detail in section 6.2.6. The miner nodes are the ones who processing transactions into blocks (*Medium, Life Cycle of Transactions*, 2017).

Mining nodes contain and maintain a pool of transactions where they collect incoming transactions. They sort the transactions by gas price. However, there are no specific rules how the nodes sort the transactions. A common configuration is to sort the transaction by gas price in a descending order to optimize for a higher pay (*Medium, Life Cycle of Transactions*, 2017).

Signed Transaction	Gas Price
hashOfTransaction1	15 Gwei
hashOfTransaction2	13 Gwei
hashOfTransaction3	11 Gwei
hashOfTransaction4	10 Gwei
hashOfTransaction5	7 Gwei

Table 1: Transaction Pool of Mining Node

Further, the mining node takes transactions from the pool and processes them into a pending block. Moreover, a block can only contain a certain number of transactions due to the block attribute *gas limit*, mentioned in section 6.2.2. That means, the mining node can only process so many transactions into one block until the gas limit is reached [3]. For example, imagine that the current *gas limit* of a block is 28 Gwei. Next, we look at the transaction pool of a mining node in table 1. As you can see, two blocks are required to process all transactions from the pool. The first two transactions are contained in the first block. Furhtermore, the second block contains the remaining three transactions. So far, it is described how a pending block will be created and the transaction limit of a block is explained. In the next section 6.2.5, the process of a pending block to a valid block is outlined in depth. Finally, after the validation process of a block is finished successfully, the life cycle of a transaction is terminated. In other words, the transaction is anchored in the blockchain and modified the world state as mentioned in section 6.2.1.

6.2.5 Mining

In this section, the steps of the validation process of a pending block will be described. In general, each mining node in the network can generate or propose a

[3] A website to see current attributes: https://ethstats.net/

block. The emerging question is, which node build the new block and how will the newly generated block accepted by the remaining network members? This process is called *consensus mechanism*. It exists a lot of various consensus algorithms, each of them provides different features, advantages and disadvantages.

To a large extent, the key perfomance drivers of a blockchain like transaction speed, scalability and security depending on the embedded consensus algorithm (Andoni et al., 2019). In this section, we will expound the *Proof of Work (PoW)* mechanism, which is at the time of this writing the current strategy in the Ethereum blockchain implementation for reaching consensus. *Proof of Work* enables a high scalability in terms of the number of nodes and clients (Vukolić, 2015), whereas it provides a poor transaction speed and a high power consumption due to the solving of a cryptographical puzzle, which requires significant computational effort (Andoni et al., 2019).

The *PoW* is a random process which is not predictable and therefore only solvable through a trial and error approach (*Proof of Work*, 2019). All mining nodes compete with each other and the goal is to achieve a hash output that is lower than a given specified target (Andoni et al., 2019). The hash output includes and comprises out of the *Parent Hash*, *Transaction Root* and *Nonce*, which are all part of the block headers data, mentioned in 6.2.2. However, the *Parent Hash* and *Transaction Root* are given and immutable, thus the *Nonce* is the value that all the mining nodes are solving for [4]. Consequently, the mining nodes modify the *Nonce* until the hash output is lower than the required target (Andoni et al., 2019). The following pseudocode in listing 1 will illustrate the *PoW* procedure:

```
targetValue = 00005cdf6d384113165841052dfd4638eaf756ac
parentHash = abbecf2d59eacbde676c3f1f0bfcf124f8c68209
transactionRoot = 917c631e5ee59596859910759c8ed76a21252010
nonce = 1
valid = False

while(not valid):
        hashOutput = sha256(parentHash+transactionRoot+nonce)

        if(hashOutput <= targetValue):
                valid = True
        else:
                nonce += 1
```

Listing 1: Pseudocode for PoW mechanism

[4] A interactive blockchain demo: https://anders.com/blockchain/

Finally, when one mining node successfully calculated the *Nonce* and therefore reaches the target value, it broadcasts the block to all other mining nodes in the network. Now, all other nodes mutually validate the correctness of the hash value by recalculating the hash output and verifying if it is lower then the target value. If the block is validated through all other mining nodes, all nodes will append this new block to their own blockchains (Zheng et al., 2017).

Considering, the network is decentralised and the simultaneous generation of valid blocks is possible when multiple nodes find the suitable *Nonce* at a nearly same time (Zheng et al., 2016). In this case, branches will be generated as shown in figure 7

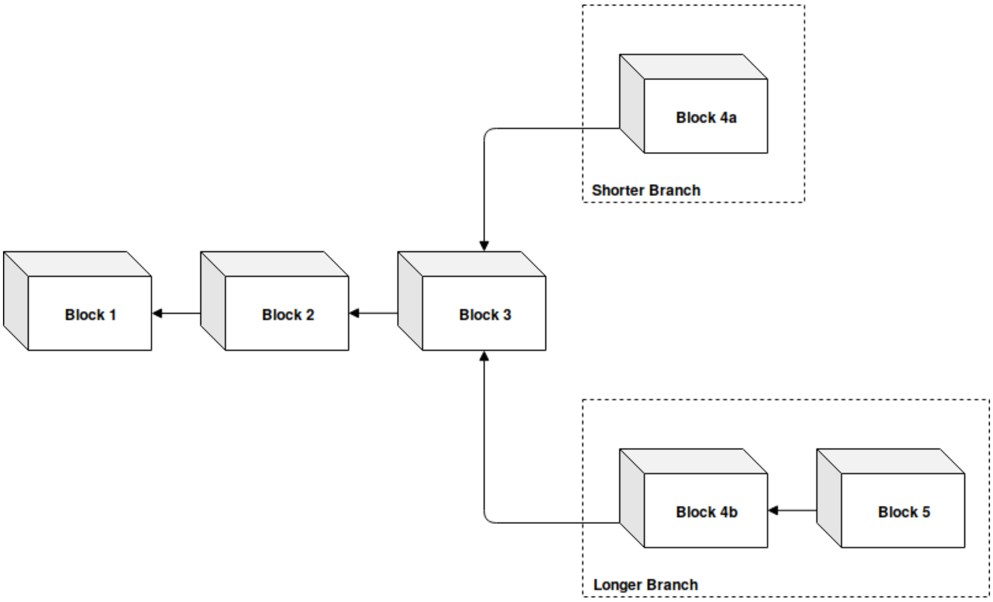

Figure 7: Scenario of Blockchain Branches

Nevertheless, it is unlikely that two competing branches will generate the next block simultaneously again. In the *PoW* protocol, the longer branch is judged as the authentic one. In figure 7, two branches arised out of two simultaneous valided blocks *Block 4a* and *Block 4b*. After that, all mining nodes accept both branches and start working on the next consecutive block until a longer branch is found (Andoni et al., 2019). As you can see in figure 7, *Block 4b* and *Block 5* forms the longer chain, therefore all miners on the branch with *Block 4a* will switch to the longer branch (Zheng et al., 2017).

To sum up, this section joins and completes the previous section 6.2.4. The consensus mechanism *PoW* is described and outlined in pseudocode, whereby the validation process of new generated blocks is explained and illustrated.

6.2.6 Ethereum Clients

First of all, the expressions Ethereum client and Ethereum node can be used interchangeable. As mentioned earlier in the previous section 6.2.4 and illustrated in the figure 6, clients can be distinguished into different types. This section will describe the concept of an Ethereum client in general and will give a brief overview of the different types.

In general, an Ethereum client is a software application that implements the Ethereum specification, which is specified in the Ethereum yellow paper (Wood, 2014). A client communicates over the peer-to-peer network with other Ethereum clients. Although these clients can be implemented in different programming languages, they all communicate through the standardized Ethereum protocol, wherefore they are able to operate and interact with the same Ethereum network. (Antonopoulos & Wood, 2018).

Since the Ethereum blockchain is not officially implemented in a particular programming language, following a list of the current main implementations of the Ethereum protocol:

- `Parity`, written in Rust
- `Geth`, written in Go
- `cpp-Ethereum`, written in C++
- `pyethereum`, written in Python
- `Harmony`, written in Java

However, while all clients differ from each other, they share some fundamental features (*Medium, Ethereum-Clients*, 2019).

First, each is able to join the peer-to-peer Ethereum network. Next, they all synchronizing a local copy of the blockchain. There are a few modes of synchronizing, which comes with miscellaneous advantages and disadvantages, which will be outlined later on. Moreover, it makes a significant difference for the health and resilience of the decentralised network. Lastly, each client is also capable of broadcasting new transactions to the network and creating and managing accounts (*Medium, Ethereum-Clients*, 2019).

As mentioned above, there are differences in client behavior regarding to the synchronizing modes. In general, the clients can be distinguished into two different types, the so called *full node* and *light node*.

Full Node: A full node will download all history data peer-to-peer from another full node. This requires a significant amount of hardware and bandwith resources (Antonopoulos & Wood, 2018). Afterwards, it will simulate every transaction in the ledger and execute the whole deployed source code to recalculate the state of each existing block (*Medium, Ethereum-Clients*, 2019). Therefore, the amount of independently operating and geographically dispersed full nodes is a crucial and very important indicator for the health, resilience and censorship resistance of a blockchain (Antonopoulos & Wood, 2018). However, a full node is not necessarily a mining node. It authoritatively validates all transactions, but to participate in the mining competition, an additional software called *ethminer* is needed [5]

Light Node: To begin with, a light node is not a separate piece of software from the clients that run a full node. The difference lies in the *light* mode for synchronization. In contrast to a full node, a light node only synchronize the block headers and the current state of the chain (*Light-Client-Protocol*, 2019). Anyway, a light node lacks the ability to run transactions throughout the history of the whole blockchain. For this reason, it does not contribute to the health and resilience of the network like a full node and traditionally cannot act as a mining node (*Medium, Ethereum-Clients*, 2019). In addition, a light node has some further limitations. For instance, they are not capable to monitor pending transactions from the network and the hash of a transaction is not sufficient to locate the transaction. Moreover, to perform certrain types of operations, a light node relies on requests to full nodes, whereby they can be far slower to query the chain (*Light-Client-Protocol*, 2019).

6.2.7 Smart Contract

This section deals with smart contracts in the Ethereum world and based entirely on the book *Mastering Ethereum* by Antonopoulos and Wood (2018). First of all, the term smart contract is a bit misleading, since Ethereum smarts contract are neither smart nor legal contracts. However, how can a Ethereum smart contract be defined? A Ethereum smart contract is an immutable comuputer program that runs deterministically in the context of an Ethereum Virtual Machine (EVM) as part of the networkprotocol. To go into the definition in more detail, immutable means, that once the contract is deployed into the blockchain, it is not possible to make any changes in the source code. The only way to modify the contract is to deploy a whole new instance. Further, deterministic means that the result of the execution is the same for all who run it, depending on the context of the transaction that initiated its execution and the state of the blockchain at the

[5] GPU Mining Worker: https://github.com/ethereum-mining/ethminer

moment of execution. As already mentioned in section 6.2.3, in the Ethereum world exists two different types of transactions. Those which result in a message call, and those which result in a contract creation. That means, smart contracts are deployed through special contract creation transactions into the blockchain. The special thing about these transactions is that the recipient of the transaction remains empty, whereby the transaction will be send to a special destination address called *zero address (0x0)*. Each contract is identified and reachable via a common Ethereum address, which can be used in a transaction as the recipient to send funds to the contract or to call one of the contract functions. Anyway, in contrast to an externally owned account, there are no keys associated with an account of a smart contract. For this reason, the creator of a smart contract does not have any special rights at the protocol level. Although, it is possible to get those special rights if you explicitly specified them in the source code of the contract. Furhtermore, contracts are only active and run if they called by a transaction. A contract is able to call another contract and so on, but nevertheless, the first call have always come from an externally owned account. A contract will never run on his own or in the background. Additionally, contracts are not executed in parallel. They will always executed consecutively, hence the Ethereum blockchain can be considered as a single-threaded machine. Finally, it should be noted that transactions are atomic regardless of how many contracts they call. Transactions execute in their entirety and any changes in the global state are only recorded if all executions terminates successfully. If the executions fails, all the changes in state are reverted as if the transaction never ran. Anyhow, a failed execution of a transaction is recorded as having been attempted. Moreover, the gas fees spent for the execution is withdrawn from the originating account.

7 Linear Programming

This section will give an introduction into the basics of mathematical linear programming and linear programming duality. It will present the purpose and necessity of optimization models and will give an example how an optimization model is constructed.

To begin with, optimization models try to define in mathematical terms, the goal of solving a problem in the best or optimal way. This can be applied in many different areas, for example, it might mean running a business to maximize profit, designing a bridge to minimize weight or selecting a flight plan for an aircraft to minimize time or fuel use (Griva et al., 2009). The use case of solving an optimization problem optimally is so ubiquitous, that optimization models are used in almost every area of application (Griva et al., 2009). Often it is not possible or economically feasible to make decisions without the help of such a model. Due to the excellent improvements in computer hardware and software in the last decades, optimization models became a practical tool in business, science and engineering (Griva et al., 2009). Finally, it is possible to solve optimization problems with a huge set of variables.

Further, linear optimization, also known as a *linear program*, refers to the optimization of a linear function with the decision variables, $x_1, ..., x_n$ subject to linear equality or inequality constraints. As presented by Bichler (2017), in canonical form a linear optimization model is given as:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad && i = 1, ..., m \\ & x_j \geq 0, && j = 1, ..., n \end{aligned}$$

However, a linear program can be modeled in different forms. It is also possible to write a model in the matrix-vector notation. To represent the above model in this form, letting $x = (x_1, ..., x_n)^T$, $c = (c_1, ..., c_n)^T$, $b = (b_1, ..., b_m)^T$ and name the matrix of the coefficients a_{ij} by A. As introduced by Griva et al. (2009), the model becomes:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

In general, the objective function of a linear model is either minimized or maximized. Additionally, the constraints may include a combination of inequalities and equalities and the variables are unrestricted or restricted in sign (Bichler, 2017).

7.1 Duality Theory

In addition, for every linear program exists another problem, which is called the *dual* of the original linear problem, also known as the *primal* (Bichler, 2017). In the *dual*, the roles of variables and constraints are reversed. That means, every variable of the *primal* becomes a constraint in the *dual*, and every constraint in the *primal* becomes a variable in the *dual* (Griva et al., 2009). For instance, if the *primal* has n variables and m constraints, the *dual* will have m variables and n constraints. Further, the *primal* objective coefficients are the coefficients on the right-hand side of the *dual*, and vice versa. Finally, the transposed coefficient matrix of the *primal* constitutes the matrix in the *dual* (Griva et al., 2009). To give an example of a *primal* and the corresponding *dual* linear program, the representation by Bichler (2017) is considered:

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \geq b \\ & x \geq 0 \end{aligned} \tag{primal}$$

$$\begin{aligned} \max \quad & b^T y \\ \text{s.t.} \quad & A^T y \leq c \\ & y \geq 0 \end{aligned} \tag{dual}$$

Next, the motivation of the duality theory can explained by the issue of trying to find bounds on the value of the optimal solution to a linear programming problem. If the *primal* is a minimization problem, the *dual* represents a lower bound on the value of the optimal solution, otherwise, if the *primal* is a maximization problem, the *dual* represents an upper bound (Bichler, 2017). To illustrate the concept and motivation of duality theory more tangible, an example stated by Bichler (2017) is given. In an application, the variables in the *primal* linear problem stand for consumer products and the objective coefficients represent the profits linked to the production of those consumer products. In this case, the objective in the *primal* describes directly the relation between a change in the production of the products and profit. Whereas, the constraints in the *primal* describe the availability of raw materials. Consequently, an increase in the availability of raw materials needed for the production of the consumer products, enables an increase in production, and finally an increase in the overall profit. However, the *primal* problem does not outline this relationship reasonable. Hence, one of the benefit of the *dual* is to properly show the effect of changes in the constraints on the value of the objective. Due to this, the variables in a *dual* linear program often called *shadow prices*, since they describe the hidden costs associated with the constraints.

7.1.1 Weak Duality Theorem

As stated in the section before (7.1), the *dual* problem provides, depending on maximization or minimization problem, lower and upper bounds for the *primal* objective function. Referring to Vanderbei et al. (2015), this result is true in general and is described as the *Weak Duality Theorem*:

Theorem 1. *If x is feasible for the primal and y is feasible for the dual, then:*

$$c^T x \leq b^T y$$

Regarding to Griva et al. (2009), there are some logical corollaries of the *Weak Duality Theorem*:

Corollary 1. *If the primal is unbounded, then the dual is infeasible. If the dual is unbounded, then the primal is infeasible.*

Corollary 2. *If x is a feasible solution to the primal, y is a feasible solution to the dual, and $c^T x = b^T y$, then x and y are optimal for their respective problems.*

7.1.2 Strong Duality Theorem

Concerning to corollary 2, it is possible to verify if the points x and y are optimal solutions, without solving the corresponding linear programs. Furthermore, this corolary is also used in the proof of strong duality (Griva et al., 2009). The *Strong Duality Theorem* describes, that for linear programming there is never a gap between the *primal* and the *dual* optimal objective values (Vanderbei et al., 2015).

Theorem 2. *If the primal problem has an optimal solution x^*, then the dual also has an optimal solution y^*, such that:*

$$c^T x^* = b^T y^*$$

7.1.3 Simplex-Method

This subsection will give a brief introduction to the *simplex-method*. To start with, in the field of linear programming, the *simplex-method* is the most widely used algorithm. This method was developed around 1940 and had several competitors at that time. However, the algorithm was able to assert itself due to its efficiency. (Griva et al., 2009) In general, the *simplex-method* is an iterative algorithm, used to solve linear programming models (Griva et al., 2009). To apply the algorithm to a model, it must be in standard form (Luenberger, 2008). According to Griva et al. (2009), we consider the following linear progamming model:

$$\begin{aligned} \text{min} \quad & z = -x_1 - 2x_2 \\ \text{s.t.} \quad & -2x_1 + x_2 \leq 2 \\ & -x_1 + 2x_2 \leq 7 \\ & x_1 \leq 3 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Next, we give an example to illustrate how to convert a linear program into standard form. To achieve that, the so called *slack variables* will be added, which represents the difference between the right-hand side and the left-hand side.

$$\begin{aligned} \text{min} \quad & z = -x_1 - 2x_2 \\ \text{s.t.} \quad & -2x_1 + x_2 + x_3 = 2 \\ & -x_1 + 2x_2 + x_4 = 7 \\ & x_1 + x_5 = 3 \\ & x_1, x_2, +x_3, +x_4, +x_5 \geq 0 \end{aligned}$$

Finally, with the linear progamming model in standard form, the idea of the *simplex-method* is to iteratively proceed from one basic feasible solution to another (Luenberger, 2008). Thereby, it always looks for a feasible solution which is better then the one before. The definition of better in this case depends on the nature of the linear problem. The process is performed until a solution is reached which cannot be improved anymore. In the end, this final solution is then an optimal solution (Vanderbei et al., 2015).

8 The Bundle-Trading-Market Framework

To start with, we explained in section 5 that local energy markets enable the trading of locally generated energy through a market platform, market mechanism and market access for small-scale prosumer and consumer. In addition, we presented in section 5.1 a market mechanism as one of seven core components for a efficient operation of blockchain-based local energy markets.

On account of that, this chapter covers and explain in detail the applied market mechanism in the developed open blockchain-based LEM platform.

As already briefly introduced in the sections 1 and 2.2, the applied market mechanism is implemented through the developed market-based optimization algorithm, which is also called the *bundle trading market framework* or short *BTM*. In the proposed framework by Guo et al. (2012), independent, self-interested agents trading bundled resources in a double-auction market, which run by a dealer. In this case, the dealer replaces the central authority and agents represent distributed entities. Generally, the *BTM* consists of a master problem that solves the market-matching problem, which is managed by the dealer, and subproblems that solve the bundle-determination problem of the agents. Further, the solution process is an iteration between the market-matching problem and the the bundle-determination problems (Guo et al., 2007). Considering, the *BTM* constitutes a market-based decomposition method for decomposable linear systems, which can be easily implemented to support real-time optimization of distributed systems (Guo et al., 2007). Likewise, the central problem of the stated market-based optimization algorithm can interpreted as the welfare optimization of all participants in a LEM. The dealer, which runs the double auction market, maximizes the welfare through allowing agents to trade their preferable bundles of energy. Therefore, the *BTM* is a suitable market mechanism for the concept of a LEM and will be used as such in the developed blockchain-based LEM simulation platform.

8.1 Problem Overview

This subsection will give an introductory description to the overall problem. With reference to Guo et al. (2012), we consider a distributed system with k independent agents. In addition, we have a central problem and individual problems of the k individual agents, which both can be expressed as the following linear programs. Besides, the *BTM* presented by Guo et al. (2007) requires a nondegenerate central problem with a bounded solution. In contrast to discrete markets where only integer number of units can be traded, the proposed framework facilitate continuous trade amounts.

Central Problem

$$\begin{aligned} Z(c) = \min_{x_j \geq 0} \quad & \sum_{j=1}^{k} d_j^T x_j \\ \text{s.t.} \quad & N_j x_j \leq n_j, \qquad i = 1, ..., k \\ & \sum_{j=1}^{k} C_j x_j \leq c \end{aligned} \tag{1}$$

Agent Problem $(j = 1, ..., k)$

$$\begin{aligned} z_j(c_j) = \min_{x_j \geq 0} \quad & d_j^T x_j \\ \text{s.t.} \quad & N_j x_j \leq n_j, \\ & \sum_{j=1}^{k} C_j x_j \leq c_j \end{aligned} \tag{2}$$

In the table below, the applied variables in the stated equations are described:

$d_j \in R^{b_j}$	vector of agents $j's$ costs
$x_j \in R^{b_j}$	decision variables controlled by agent j
$N_j \in R^{a_j \times b_j}$	activity matrix
$C_j \in R^{m \times b_j}$	activity matrix
$n_j \in R^{a_j}$	capacity vector of agent $j's$ independent resources that are managed locally
$c_j \in R^{m}$	agent $j's$ vector of shared resources that can be exchanged with other agents

Table 2: Applied variables in BTM Framework

Further, the overall objective of the central problem is to minimize the operating costs of the overall system. The minimization of the central problem have to be implemented in consideration of the operational constraints (first set of constraints) and the total shared resources capacity constraints (second set of constraints) of each individual agent. Anyhow, the optimal solution to the central problem cannot be directly calculated, due to the lack of access to all relevant information for decision making $(d_j, n_j, c_j, N_j, C_j, forj = 1, ..., k)$. Additionally, Guo et al. (2007) stated that, for an allocation of $c_j, j = 1, ..., k$ such that $C_j x_j^* \leq c_j, \sum_{j=1}^{k} C_j$, solving the agents' problems is equivalent to solving the central problem.

Hence, a market-based resource allocation mechanism to coordinate decentralized decision making from agents is used by the *BTM*. For any intially given $c_j, j = 1, ..., k$, these mechanism enables an indirectly acquisition of an optimal solution to the central problem through an direct, iterative and wealth-improving bidding process among the self-interested agents.

8.2 Market Environment

Generally, a dealer and k independent agents constitute the whole market economy. Each of the agent have the opportunity to trade the shared resources c_j in a double auction market, which is operated by the dealer. Moreover, each agent only has local perspective and knowledge, that is to say, they don't know anything about the prodcution decisions of the others. Further, the market prices to match the trades, will be determined by the dealer. Subsequently, the roles of the market participants and what they know is described. Equally, the market operations like the agent bidding, market matching and settlement is outlined and a summary of the market-based algorithm is presented.

8.2.1 Agent bidding

First of all, the initialization takes place. That means, each of the $j = 1, ..., k$ agent receives an endowment of the m shared resources c_j , and a cash endowment e_j. The wealth of an agent at any point is defined as $e_j - z_j(c_j)$ (Guo et al., 2007). Next, agents have the opportunity to buy additional resources or to sell some of their own resources to lower their operating costs. Therefore, each agent knows the current market prices $p \in R^m$ for the shared resources, which are published by the dealer. Further, for trading the shared resources in the market, a bundle referring to Guo et al. (2012), is defined:

Bundle A bundle $w \in R^m$ is an m-dimensional vector of shared resources. Each of the m elements corresponds to an amount of one specific shared resource. A negative sign of an element signify a sell amount, and contrary, a positive sign signify a buy amount.

An *improving bundle set* concerning to lower operating costs is defined as the following:

Improving Bundle Set

$$\begin{aligned} &W_j(c_j|p) = \{w : \exists x_j \geq 0 \ni d_j^T x_j + p^T w \leq z_j(c_j), \\ &N_j x_j \leq n_j, C_j x_j \leq c_j + w\} \end{aligned} \tag{3}$$

To remember that $z_j(c_j)$ is the optimal value of the agent $j's$ problem depending on the amount of the shared resources c_j. That means, an agent looking for bundles that satisfy $d_j^T x_j + p^T w \leq z_j(c_j)$. If $p^T w \geq 0$, the term $p^T w$ is interpreted as the payment to receive the bundle w. On the opposite, if $p^T w \leq 0$, the term $p^T w$ constitutes the revenue for selling the bundle w.

Accordingly, a rational agent will choose an improving bundle for trading, which leaves his wealth level on the same level or better off. In addition, the basic market mechanism of Guo et al. (2007) assume no strategic actions in the bundle selection and pricing. However, strategic actions can be relaxed and Guo et al. (2012) also incorporate strategic factors in their extended *BTM*. Nevertheless, the applied market mechanism in the developed open blockchain-based LEM simulation implemented nonstrategic bidding, wherefore an agents bundle selection can be defined as the following *bundle determination problem:*

Bundle Determination Problem

$$\begin{aligned} \min_{x_j \geq 0, w} \quad & d_j^T x_j + p^T w \\ \text{s.t.} \quad & N_j x_j \leq n_j, \\ & C_j x_j \leq c_j + w \end{aligned} \tag{4}$$

According to Guo et al. (2007), the *bundle determination problem* have either a bounded, or an unbounded solution. A bounded solution is called *limited bundle* $w \in R^m$, whereby an unbounded solution is called *unlimited bundle* $u \in R^m$.

Furthermore, each agent needs to determine a limit price, that indicates the maximum an agent is willing to pay for the bundle. As already explained above, nonstrategic pricing is implemented. Therefore, an agent will always submit a limit price equal to the valuation of the bundle. That is to say, $l(w) = v(w)$.

For a *limited bundle* w, the value $v(w)$ is defined as:

$$z_j(c_j) - z_j(c_j + w) = z_j(c_j) - d_j^T x_j.$$

Whereas for a *unlimited bundle* u, the value $v(u)$ describes the unit-incremental value that u contributes to the objective change. It is defined as:

$$-d_j^T \hat{x}_j.$$

Potentially, an agent has multiple optimal solutions in his bundle-determination problem. Hence, it is allowed to submit more than one bundle at a time.

Finally, old orders and bids of prior rounds without any trades will be treated as open orders and bids, because the valuation of an agent for a bundle is independent of the market price p. It only depends on the respective resource level c_j.

8.2.2 The Dealer's Market Clearing Mechanism

First of all, the developed open blockchain-based LEM platform applied a synchronous call market where the market prices p are announced periodically by the market dealer. However, the synchronous requirement can be relaxed. Therefore, Guo et al. (2012) introduced a ansynchronous market trading environment in their e-companion.

Furthermore, the market has a dealer who trades on her own account. The dealer has some initial cash endowment e_0 and resource endowment c_0. Anyway, the sum of all resource endowments needs to fulfil the following equation:

$$\sum_{j=1}^{k} c_j + c_0 \leq c$$

Next, agents submit sealed bids to the dealer. The dealer maintains an individual order book for each agent. Each individual order book contains two different order sets. On the one hand, the set for limited orders I_j, on the other hand, the set for unlimited orders H_j. If an agent submits orders and no trade is executed, the orders are collected and accumulated for the respective agent. Otherwise, any trade from an agent will clear the respective order book. Lastly, to determine the maximal trade surplus the dealer solves the following *market matching problem*:

Market Matching Problem

$$\begin{aligned} \max_{y_j^i \geq 0, t_j^h \geq 0} \quad & \sum_{j=1}^{k} \left(\sum_{i \in I_j} l_j(w_j^i) y_j^i + \sum_{h \in H_j} l_j(u_j^h) t_j^h \right) \\ \text{s.t.} \quad & \sum_{j=1}^{k} \left(\sum_{i \in I_j} w_j^i y_j^i + \sum_{h \in H_j} u_j^h t_j^h \right) \leq c_0 \\ & \sum_{j=1}^{k} y_j^i \leq 1, \quad j = 1, ..., k \end{aligned} \tag{5}$$

The objective of the *market matching problem* maximizes the trade surplus.

Considering the first set of constraints, if $c_0 = 0$, it is required that any buy amounts be met by sell amounts. On the opposite, if $c_0 \neq 0$, the dealer supplies additional resources from the inventory to meet buys.

With respect to the second set of constraints, it is indicated that the market handles limited and unlimited bundles differently. The trades of unlimited bundles in H_j are unrestricted, whereas limited bundles in I_j are restricted. In that case, limited trades are restricted to be convex combinations of bundles in I_j.

Besides, the *market matching problem* always has a solution, because setting all variables to zero is always a feasible solution. As described by Guo et al.

(2007), the market closes when no trade takes place and the market prices remain unchanged. Finally, if the *market matching problem* has a nonzero solution for $y_j^{i^*}$, for $i \in I_j$ and $t_j^{h^*}$, for $h \in H_j$, then agent j will have traded the following:

$$w_j^* = \sum_{i \in I_j} w_j^i y_j^{i^*} + \sum_{h \in H_j} u_j^h t_j^{h^*}.$$

Thus, the market clearing prices $p \in R^m$ are derived from the dual values of the first set of constraints in the *market matching problem.* To remember, the concept of duality is explained in section 7.1. Consequently, the settlement price for an agent who have traded resources are the following:

$$p^T w_j^*.$$

In the following, the whole settlement process is summarized and the calculation of the new values of agent j and the dealer is outlined:

Agent j:

$$\begin{aligned} c_j &\leftarrow c_j + w_j^* \\ e_j &\leftarrow e_j + p^T w_j^* \end{aligned}$$

Dealer:

$$\begin{aligned} c_0 &\leftarrow c_0 + \sum_{j=1}^{k} w_j^* \\ e_0 &\leftarrow e_0 + \sum_{j=1}^{k} p^T w_j^* \end{aligned}$$

8.2.3 Summary of Market-Based Algorithm

Concluding, this subsection will give an overview of the overall, decentralized and synchronous call market-based algorithm presented by Guo et al. (2007). On the one hand, the process of the algorithm is outlined, on the other hand, a simplified flowchart of the *BTM* framework, inspired by Guo et al. (2012), is presented.

Input

Initial allocations: e_j, c_j, for $j = 1, ..., k$, and
initial market prices: $p \geq 0$.

Output

Optimal solutions x_{j^*} to the central problem,
optimal allocations c_{j^*}, for $j = 1, ..., k$, and
optimal market prices p^*.

Step 0

Set $I_j = \emptyset, H_j = \emptyset$, for $j = 1, ..., k$, and
the last market-price vector $\pi = 0$.

Step 1

The dealer saves the current market prices $\pi \leftarrow p$. Each agent solves his *bundle determination problem* and adds new limited bundle orders to I_j and unlimited bundle orders to H_j.

Step 2

The dealer solves the *market matching problem* (w_j^* represents a matched bundle for agent j).

Step 3

If no nonzero solution exists, this step will be skipped and continued with *Step 4*. If a nonzero solution to the *market matching problem* exists and the trades includes items from at least one agent and the dealer, the settlement takes place. The dealer publishes the shadow prices p as the new market prices. The new allocations are calculated as follows: $c_j \leftarrow c_j + w_j^*$, for $j = 1, ..., k$ and $c_0 \leftarrow c_0 + \sum_{j=1}^{k} w_j^*$. Further, the new endowments are calculated as follows: $e_j \leftarrow e_j + p^T w_j^*$ for $j = 1, ..., k$ and $e_0 \leftarrow e_0 + \sum_{j=1}^{k} p^T w_j^*$. Reset $I_j = \emptyset$ and $H_j = \emptyset$ for each agent j with $w_j^* \neq 0$. Now, go to *Step 1*.

Step 4

The dual values of the clearing constraints from the *market matching problem* form the new market prices p. If the new market prices $p \neq \pi$, go to *Step 1*, otherweise, stop.

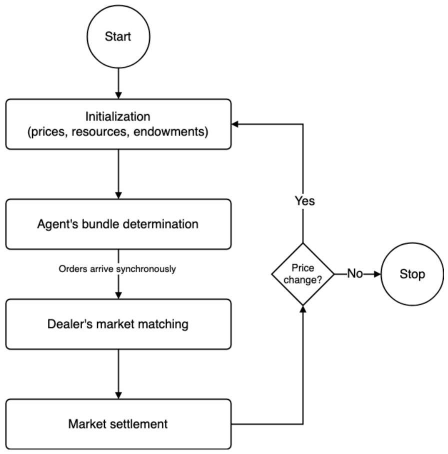
Start
Initialization
(prices, resources, endowments)
Agent's bundle determination
Orders arrive synchronously
Dealer's market matching
Market settlement
Yes
Price change?
No
Stop

Figure 8: Flowchart of the BTM Framework

9 Concept of LEM Simulation Platform

To start off, this section brings together the previous sections 6 and 8, and explains how the bundle trading market framework *(BTM)* and a blockchain interact to implement a simulation of a local energy market, which are introduced in section 5. Finally, this section gives an example for a linear programming problem in terms of energy efficient demand side management of households.

First, in the presented market mechanism in section 8, independent, self-interested agents trading bundled resources in a double-auction market. In case of the developed open-blockchain based LEM simulation, the agents of the *BTM* respresent households. These households are trading independently and self-interested bundles of energy resources to minimize the monetary expense through optimally scheduling the operation and energy consumption of all energy consuming appliances.

Next, blockchain is introduced as the applied ICT of the developed LEM simulation. As already mentioned earlier, the implementation of LEM needs local distributed control and management techniques, which can addressed by the blockchain technology. This implies that the respective households submit and receive all relevant data, informations and payments via a blockchain. Moreover, the double-auction market which enables households to trade their energy bundles, is operated by a dealer. In turn, the dealer is implemented by a smart contract, which are introduced in section 6.2.7, and a conventional software client. The dealers' smart contract contains all relevant information regarding the market mechanism, like submitted orders, trades, dealers' inventory and market prices. Therefore, the households exclusively communicate with the dealers' smart contract.

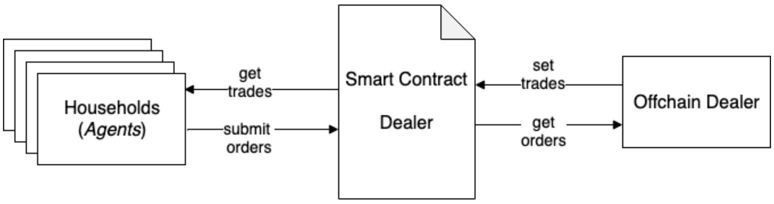

Figure 9: Simplified presentation of the applied communication

With reference to figure 9, the households submit their orders to the dealers' smart contract. Afterwards, the dealers' conventional software client, designated as *off-chain* dealer in the figure, fetches all existing orders contained in the smart contract and solves the *market matching problem*, which is introduced in section 8.2.2. Secondly, the *off-chain* dealer updates all relevant information in the smart contract, for instance the settled orders, the trades, the new market prices and the inventory. Finally, the households get their respective trades from the dealers' smart contract.

Anyway, the *off-chain dealer* is necessary due to the complex and costly calculation of the *market matching problem.* As mentioned in section 6.2.3, each transaction consume gas. Additionally, each mathematical operation in a smart contract increases the amount of gas to be paid for the respective transaction. As a consequence, implementation and execution of the *market matching problem* in the smart contract itself would be hard to realize and associated with high transaction costs.

Hence, a well designed smart contract should move computational complexity *off-chain* and focus more on updating state in the contract (*Calculating Costs in Ethereum Contracts - HackerNoon.com*, 2019). A balance between *on-chain* and *off-chain* complexity should be found.

In addition, we mentioned in section 8 that the developed open blockchain-based LEM platform applied a synchronous call market. That means, the dealer only executes the *market matching problem* after all agents have submitted their orders. Likewise, we mentioned in section 6.2.7 that smart contracts are only active and run if they called by a transaction. They will never run on his own or in the background. For this reason, a mechanism is necessary which triggers the execution of the *market matching problem.* The implementation of an *off-chain* dealer is also suitable for this.

So far, this section presented the basic idea and concept of the developed open blockchain-based LEM platform and explained the necessity of all exisiting components. In the following section 10, the setup and technical implementation of the simulation will be described in detail.

However, we stated earlier in this section that the agents of the *BTM* respresent households and that these households trading bundles of energy resources to minimize the monetary expense. In section 8.1, we introduced the individual linear progams of the agents. Further, we explained in section 8.2.1 that a rational agent will choose an improving bundle for trading which leaves his wealth level on the same level or better off and defined the selection of such a bundle as the *bundle determination problem.* Finally, to embed the *BTM* into the topic of local energy markets, we need to define a linear programming problem which depicts the energy efficient demand side management of households.

9.1 Demand side management of a household

This subsection will embed the BTM into the topic of local energy markets. Therefore, we will develop a exemplary and simplified linear programming problem in terms of energy efficient demand side management of households. That means, the objective of the respective households is to minimize their monetary expense regarding energy consumption. However, the minimization of the costs due to energy consuming depends on a number of constraints. It must be ensured that sufficient energy can be provided at all times.

Chen et al. (2013) developed such a linear programming model for demand side management of households, on which the following presented model is based.

First, we introduce all relevant variables. Let a denote an appliance, and A the set of all existing appliances in a household. In that case, an appliance constitutes energy consuming items, e.g. washers, refrigerators, plug-in hybrid vehicles, etc. Next, the variable x_a denotes the energy consumed by an appliance a to a given point in time. Further, each appliance $a \in A$ has a maximum energy level that is defined as rated power and described by P_a. Additionally, it exists a limit on the total energy consumed by various appliances of a household. This total energy consumption limit is described by L_A. In case the total energy consumption limit is exceeded, the power network of the household will be tripped out. Moreover, a household consumes on the one hand solar energy from photovoltaic systems (PV) and on the other hand energy from the electrical grid. If households do not consume the solar energy directly, they have the opportunity to store it in batteries, or export it back to the main power grid if the battery is full. Let e_g denote the energy used from the electrical grid, and e_s the energy produced by the photovoltaic systems. In addition, let p_g and p_s denote the unit price of the energy from the electrical grid and solar energy. Furthermore, we describe the solar energy used by all appliances of a household as y_s, the consumed battery energy as y_b and the remaining solar energy which is not used by the household appliances and stored in the battery as z_b.

Finally, we developed the following linear program for optimization of the household monetary expense.

$$\begin{aligned} \min \quad & p_g \cdot e_g + p_s \cdot e_s \\ \text{s.t.} \quad & \sum_{a \in A} x_a \leq L_A \\ & x_a \leq P_a, \quad \forall a \in A \\ & \sum_{a \in A} x_a = y_s + y_b + e_g \\ & y_s + z_b \leq e_s \end{aligned} \tag{6}$$

In the table below, all applied variables of the developed linear program are listed:

a	energy consuming appliance of household
A	set of all existing energy consuming appliances of household
x_a	consumed energy of an appliance a
P_a	maximum energy level (rated power) of an appliance a
L_A	total energy consumption limit of a household
e_g	energy used from the electrical grid
e_s	energy produced by PV
p_g	unit price of energy from electrical grid
p_s	unit price of energy from PV
y_s	solar energy used by all appliances of a household
y_b	consumed batterie energy by all appliances of a household
z_b	remaining unused solar energy stored in the batterie

Table 3: Applied variables in linear program for optimization of the household monetary expense

As mentioned earlier, the objective of the respective households is to minimize their monetary expense regarding energy consumption. The term $p_g \cdot e_g$ denotes the costs for energy from the electrical grid, whereas the term $p_s \cdot e_s$ denotes the costs for energy from photovoltaic systems.

Besides, the first constraint ensures that the total energy consumption of all energy consuming appliances of a household not exceeds the limit L_A.

Next, the second constraint describes the requirement that the consumed energy for all existing appliances a in a household is less or equal to the respective rated power P_a.

Further, the third constraint makes sure that the energy consumed by all appliances of a household is the sum of the energy from the electrical grid, the solar energy from the PV system and the battery energy. This implies, if the amount of solar and battery energy is not sufficient to cover the energy consumption of all appliances, external energy from the electrical grid will be used. Therefore, it is ensured that sufficient energy is available at all times.

Finally, the fourth constraint describes that the solar energy produced by the PV system is greater or equal to the sum of the consumed and stored solar energy. At times when the battery is fully charged and the energy consumption of the appliances is covered by solar energy, it is possible to sell the surplus of wasted solar energy. Referring to the *BTM*, the first two constraints of the proposed linear programming model constitute the independent resources that are managed locally. The shared resources are represented by the energy resources e_g and e_s. Therefore the respective households trade the following limited bundles.

$$w = \begin{pmatrix} e_g \\ e_s \end{pmatrix}$$

Consequently, the objective of the linear programming model written in matrix-vector notation.

$$\min \quad p^T \cdot w$$

where p^T denotes the transposed price vector.

$$p^T = \begin{pmatrix} p_g & p_s \end{pmatrix}$$

To sum up, this subsection presented a simplified linear programming model for demand side management of a household that can be integrated into the proposed *BTM*.

Furthmore, the developed model should only serve as an inspiration and example and has no claim to completeness. It is intended to illustrate how the presented framework can be adapted to enable the trading of energy resourcres in a local energy market. Finally, researchers without a deep technically background are able to use this platform and develop and investigate their concepts.

10 Implementation

This section deals with the technical implementation of the LEM simulation and gives insights into the developed source code. First, a brief introduction to the applied software technologies and libraries is given. Next, the individual components and their attributes and functions are presented. Finally, the whole simulation process is outlined and the implementation of the fundamental parts of the simulation process are described in detail.

10.1 Applied technologies

Above all, this subsection give a brief introduction to the used software technologies and libraries.

Python An interpreted, object-oriented, high-level programming language with dynamic semantics. Additionally, it is simple and comes with a easy to learn syntax [6]. The programming language of choice, which is used to develop the respective components that communicate via the blockchain.

Web3.py A Python library for interacting with a Ethereum blockchain [7]. It enables the developed Python components to communicate via the blockchain.

Ganache A personal blockchain for Ethereum development [8]. It simulates a full client behavior and make developing Ethereum applications faster, easier and safer. Ganache is provided on the one hand as a software tool with a graphical user interface (GUI) and on the other hand as command line tool (CLI). In the develop LEM simulation the command line version of Ganache is used.

Solidity A statically typed, contract-oriented, high-level programming language for implementing smart contracts on the Ethereum blockchain [9]. It is used to implement the smart contract dealer.

[6] https://www.python.org/

[7] https://github.com/ethereum/web3.py

[8] https://github.com/trufflesuite/ganache-cli

[9] https://github.com/ethereum/solidity

10.2 Components

To start off, this subsection presents the respective main components that interact with each other during the runtime of the simulation. However, the individual components are considered in isolation in this subsection.

10.2.1 Agent

The agents are implemented by a Python class. Moreover, the naming is oriented on the proposed *BTM* and, as mentioned earlier in section 9, the agents represent the respective household in the LEM simulation. To begin with, the class attributes are presented. The associated source code can be seen in the following listing 2.

```
class Agent(object):

def __init__(self, agent_number, account_address, web3, dealer_contract):
    self._name = 'AGENT{}'.format(agent_number)
    self._account_address = account_address
    self._web3 = web3
    self._dealer_contract = dealer_contract
    self._optimization_problem = None
    self._bundle_set = None
    self._bid = None
    self._mkt_prices = None
    self._trade = None
    self._objective = None
    self._wealth = None
    self._accept_trade = None
```

Listing 2: Overview of the agent class attributes

The variables `agent_number`, `account_address`, `web3` and `dealer_contract` are initialized when the class is instantiated. The variable `agent_number` is attached to the name of the respective agent and serves as an identifier. Otherwise the variable `account_address` could be used to identify the agents, but it would be cumbersome due to the cryptic hexadecimal representation of the blockchain account addresses. Therefore a more understandable name for each agent is established. Furthermore, each agent is assigned an Ethereum blockchain account defined by the variable `account_address`. It is required to submit and receive transactions via the blockchain. Moreover, the variable `web3` represents the Python library for interacting with a Ethereum blockchain, which is already introduced in section 10.1. The last variable which is initialized on instantiation is the `dealer_contract`. This variable contains a object of the smart contract dealer. This object knows all implemented functions of the dealer's smart contract. Hence, this variable is needed to call the functions of the smart contract out of this Python class. All other class attributes are set immediately after the instantiation or during the running simulation.

Next, some of the most important functions of the class are introduced and explained. After the instantiation, each agent get their optimization problem. In the listing 3 below, the implementation of the setter is shown.

```
@optimization_problem.setter
def optimization_problem(self, value):
    self._optimization_problem = value
    self._objective = self._optimization_problem.solve().fun
    self._wealth = self.balance + abs(self._objective)
```

Listing 3: Setter of optimization problem

As you can see in the listing, after setting the optimization problem, the class attributes `objective` and `wealth` are determined. In this case, the attribute `wealth` is defined as the sum of the current `balance` of the account and the absolute value of the current `objective`. Equally, it is defined in the *BTM*, outlined in section 8. However, `balance` is not a class attribute, but a class function. It uses the class attributes `web3` and `account_address` to retrieve the current balance of the account from the blockchain.

In addtion, the function for determining the improving bundle set and the associated bid is introduced in the listing 4.

```
def determine_bundle_attributes(self):
    result = solve_bundle_determination(self._optimization_problem, self.↩
        _mkt_prices)
    self._bundle_set = result.x
    self._bid = self._objective - result.fun
```

Listing 4: Determining of bundle attributes

As shown in the listing, first the bundle determination problem is solved. It is solved in a separate function which requires the optimization problem and the current market prices as parameters. Further, the class attributes `bundle_set` and `bid` are initialized. We stated in section 8.2.1 that the applied market mechanism implemented nonstrategic bidding and pricing, wherefore an agent always submits a limit price equal to the valuation of the bundle. This is reflected in the definition of the class attribute `bid`. The true value of the improving `bundle_set` is the current `objective` minus the objective of the optimization problem with the resources of the `bundle_set`. In this case, the minus sign reflects the fact that we are minimizing costs. After presenting the determination of all necessary bundle attributes, the submitment of an order is shown in the listing 5 below. First of all, the variables `bundle_set` and `bid` are prepared for sending to the dealer's smart contract. In Solidity, fixed point numbers are not fully supported yet. It is possible to declare them, but they cannot be assigned to or from (*Solidity Documentation*, 2019).

```
    def set_order(self):
        bundle_set = utils.prepare_for_sending(self._bundle_set)
        bid = utils.prepare_for_sending(self._bid)

        if(bid > 0):
            prepayment = utils.from_ether_to_wei(self._bid)
        else:
            prepayment = 0

        self._dealer_contract.contract.functions.setOrder(bundle_set, bid, ↩
            prepayment).transact({'from': self._account_address, 'value': ↩
            prepayment})
```

Listing 5: Submitment of order

Therefore, we developed a work around that shift the decimal place two digits to the right and cuts off the remainder, so that you receive integer values.

Additionally, the variable `prepayment` is assigned. The condition that `bid` is greater than zero identifies if this order represents a buy order. In case of a buy order, the agents have to pay in advance. Therefore the value of `prepayment` is set equally to the value of variable `bid`. This procedure is implemented to prevent the strategic placing of orders and to ensure that the financial resources are available. If it is a sell order, no payment in advance is required and the value of `prepayment` is set to zero. Furthmore, before the value of `prepayment` is equated with the value of `bid`, a function is called that converts the value of `bid`. This function converts the input values from the unit Ether to the unit Wei. The implementation of this function is required because Wei is the base unit for currency in Solidity and all monetary values within a contract are given in it. However, on the client side we decided to specify all monetary values in Ether for better clarity.

Finally, the already introduced class attribute `dealer_contract` is used to call the function of the dealer's smart contract which is responsible for setting the orders.

Furthermore, the class provide a function to evaluate the published market prices of the dealer. This function is implemented to verify if the dealer has calculated the prices of the respective trades correctly.

The related source code of this function can be seen in the listing 6 above. To begin with, the function retrieves the values of the primal *market matching problem* and the belonging dual problem from the dealer's smart contract. For the verification of the correctness, the introduced *Strong Duality Theorem* in section 7.1.2 is applied. Due to this, it is possible to verify if the solution of the dealer is optimal, without solving the corresponding *market matching problem*. To apply the *Strong Duality Theorem*, the solution of the primal problem (represented by variable `primal_solution`) is calculated on the one hand, and the solution of

```
def verify_strong_duality(self):
    mmp_values, mmp_duals, mmp_target_coefs, mmp_bounds = self.↩
        get_mmp_attributes()
    primal_solution = (-np.sum(mmp_values * mmp_target_coefs))
    dual_solution = np.sum(mmp_duals * mmp_bounds)
    primal_solution = math.floor(primal_solution * 10) / 10
    dual_solution = math.floor(dual_solution * 10) / 10

    if primal_solution == dual_solution:
        self._accept_trade = True
    else:
        self._accept_trade = False
```

Listing 6: Verification of trades

the dual problem (represented by variable `dual_solution`) on the other hand. Additionally, both solutions are rounded to one decimal place. This procedure is implemented due to the issues and limitations of floating point arithmetic [10]. We want to make sure that correct trades are not rejected on the basis of those limitations. Finally, it is examined if both solutions are identical. If this is the case, the class variable `accept_trade` of type boolean is set to `True`, otherwise to `False`.

Likewise, the class provides a function to notify the dealer's smart contract if the respective trade is accepted or not. The implementation of this function is shown in listing 7.

```
def accept_trade(self):
    if(self._accept_trade):
        trade = self._dealer_contract.contract.functions.acceptTrade(self.↩
            _accept_trade).call({'from': self._account_address})
        self._trade = utils.prepare_for_storing(trade)

    self._dealer_contract.contract.functions.acceptTrade(self._accept_trade↩
        ).transact({'from': self._account_address})
```

Listing 7: Notification of trade acceptance

If the verification of the *Strong Duality Theorem* was successful and the variable `acccept_trade` was thus set to the value `True`, the function retrieves the trade and stores it in the class attribute `trade`. Besides, before the trade is stored, it is prepared for storing. This is due to the already mentioned work around that shift the decimal place two digits to the right before sending values to the dealer's smart contract. When retrieving and storing values, exactly the opposite is done and we shift the decimal place two digits to the left.

Furthermore, as you can see in the listing 7 above, an almost identical function invocation of the dealer's smart contract function `acceptTrade()` is indicated twice. The difference lies in the expressions `call()` and `transact()`. The `call()` method of the Web3.py library enables to invoke any smart contract func-

[10] https://docs.python.org/2/tutorial/floatingpoint.html

tion on a read-only basis and returns values sent by the smart contract return statement. Those read-only invocations run much faster than transactions that require network verification. On the contrary, the `transact()` method submits a verified transaction that potentially change the state of the blockchain. Hence, network verification is needed that causes a significant delay due to the mining procedure. Consequently, this method does not return values of the smart contract return statements, but the transaction hash of the submitted verified transaction. Regarding the implementation, the first invocation only serves to preserve the respective bundle, whereas the second call of the function serves to notify the dealer whether the trade has been accepted or not. For this reason, the first call of the dealer's smart contract function `acceptTrade()` is only executed if the trade is accepted.

Finally, a last class function is described. This function adds the received trade to the shared resources as it is defined in the *BTM*, introduced in the section 8.2.2.

```
    def add_trade_to_shared_resources(self):
        if(self._accept_trade):
            self._optimization_problem.shared_resources = np.add(self.↩
                _optimization_problem.shared_resources, self._trade)
            self._objective = self._optimization_problem.solve().fun
            self._wealth = self.balance + abs(self._objective)
```

Listing 8: Adding of trade to shared resources

As shown in the listing 8 above, the code of the function is only executed if the trade has been accepted. In addition, the new `objective` of the linear programming model is calculated after getting the new resources. Since a new `objective` has been calculated, the class attribute `wealth` is also recalculated accordingly.

10.2.2 Smart contract dealer

In this part, the dealer's smart contract is introduced. As already mentioned, the contract is implemented by the programming language Solidity. First of all, attributes of the contract are presented in the following listing 9.

First, general attributes are declared. As you can see, the contract contains an attribute called `owner`. This variable is set in the constructor of the contract and initialized with the address of the contract creator, which is in this case the off-chain dealer. It is being used in a modifier which ensures that certain functions can only be executed by the owner of the contract. Moreover, a modifier can be seen as an extension of a function and is mainly used to to check if certain conditions are met before executing the rest of the source code in the body of a

```
contract Dealer{
    // general attributes
    address private _owner;
    int256[] public resource_inventory;
    int256[] public mkt_prices;

    // attributes for order management
    uint32 public order_count;
    uint32[] public order_indices;
    mapping(uint32 => Order) public orders;

    // attributes for trade management
    mapping (address => uint256) account_index;
    address[] public trades_accounts;
    mapping(address => int256[]) public trades;
    mapping(address => uint256) public bills;
    mapping(address => uint256) public prepayments;
    mapping(address => uint256) public refunds;
```

Listing 9: Attributes of Smart Contract

function.

In addition, attributes for managing the orders and trades of the agents are declared. The orders, trades and additional properties of the trades are predominantly managed in mappings. A mapping is a collection of key value pairs. All of the keys have to be of the same data type, and all values must be of the same data type. However, Solidity does not provide an iteration function to retrieve all the lists that are stored in a mapping. That is to say, it is not possible to read the values from the mapping without knowing the respective keys. Therefore, we developed a work around and store the keys of mappings in additional arrays which serves as look up tables. With reference to the listing 9, the array `order_indices` functions as a look up table for the mapping `orders`, and the array `trades_accounts` for the mapping `trades` and all other trades related mappings.

Next, some of the most important functions of the dealer's smart contract are introduced and explained.

To begin with, the function for receiving the agents orders is outlined in the listing 10. The function header contains on the one hand the Solidity built-in modifier `payable`, and on the other hand, the custom modifier `checkPrepayment`. A function declared with `payable` is one that can accept incoming payments. Without the declaration, the function would reject payments and throw an exception. In this case, the modifier is required to receive the prepayments of the agents. Further, we developed the second modifier `checkPrepayment` to verify whether the incoming payment match the amount of required prepayment.

Hence, if the prepayment is not sufficient, the function call of the agents is rejected and the order is not accepted.

Next, an `Order` object is declared and initialized. The order object is imple-

```
function setOrder(int256[] _bundle, uint256 _bid, uint256 _prepayment) ↩
    public payable checkPrepayment(_prepayment) {

    Order memory new_order = Order(
        msg.sender,
        _bundle,
        _bid
    );
    orders[order_count] = new_order;
    order_indices.push(order_count);

    // increment order count
    order_count ++;
}
```

Listing 10: Receiving agents orders

mented by the data type struct. It is a user-defined data container for grouping variables. This data type is well suited to bundle and manage all necessary information regarding an order. In this case, the object `Order` contains the variables `msg.sender`, `bundle` and `bid`. The variables `bundle` and `bid` are passed as function parameters. Besides, when a contract is executed, it has access to a small set of global objects. One of these objects represents the `msg` object. It is the transaction call that launched the contract execution. Thereofore, the account address of the agent who executed the function call can be reached via the `msg.sender` variable contained in the `msg` object.

Finally, the `Order` object is added to the mapping `orders`. The variable `order_count` is an incremental number which serves as the key for the mapping `orders`. After each incoming order this number is increased by one. Additionally, the variable `order_count` is added to the array `oder_indices` that serves as a look up table as mentioned earlier.

Furthermore, the contract function which serves to set the trades of the agents is introduced in listing 11. With reference to the function header, the modifier `onlyByOwner()` is applied. This custom modifier examines if the caller of the function is also the owner of the contract. This means, that only the owner of the contract has the permission to call this function. The owner of the contract is the off-chain dealer. Accordingly, the applied modifier ensures that no one else can set the trades.

```
function setTrade(address _account, int256[] _trade, uint256 _prepayment, ↩
    uint256 _bill, uint256 _refund) public onlyByOwner() {
    addToArray(_account);
    trades[_account] = _trade;
    bills[_account] = _bill;
    prepayments[_account] = _prepayment;
    refunds[_account] = _refund;
}
```

Listing 11: Setting the trades

Next, the function `addToArray()` is called and the variable `account` is passed. This function checks whether the account to which the trade belongs is already contained in the array `trades_account`. If not, the function adds the account to the array, which again serves as the look up table for the mapping `trades` and all other trades related mappings. Concluding, all the required informations for a trade are placed in the corresponding mappings. In all mappings the account address is used as the key.

At the end, a last function of the dealer's smart contract is introduced. The implementation of the function is shown in listing 12 above and can be seen as a kind of collection point for the trades.

```
function acceptTrade(bool accept_trade) public returns (int256[]) {
    if(accept_trade) {
        msg.sender.transfer(refunds[msg.sender]);
        return trades[msg.sender];
    } else {
        msg.sender.transfer(prepayments[msg.sender]);
        delete trades[msg.sender];
    }
}
```

Listing 12: Collection point of trades

The function enables agents to receive or reject their trades. In case of an acceptance, the function pay the agents their refund and returns their belonging granted trade. As explained earlier, the agents prepay the true value of the entire requested bundle. However, the agents often only get shares of their requested bundles due to the solution of the *market matching problem.* For this reason, the agents have often prepaid more than they have to pay, which leads to excess payments. Because of this, the dealer calculates the difference between the prepayment and the price of the granted trade and refunds it. In case of a rejection, the dealer refund the whole prepayment and deletes the granted trade. The deletion of a rejected trade is necessary to calculate the dealer's resource inventory correctly after the settlement process.

10.2.3 Off-chain dealer

We mentioned earlier in the section 9 that the dealer is implemented by a smart contract and a conventional software client. In the previous subsection 10.2.2, the dealer's smart contract was described. In turn, this subsection deals with the counterpart, the off-chain dealer.

Firstly, the off-chain dealer is implemented by a Python class and the class attributes are introduced and presented in the listing 13 below. The variables `account_address`, `web3`, `dealer_contract`, `shared_resource_size` and `mkt_prices` are initialized when the class is instantiated. Again, the variables `account_address`, web3 and dealer_contract have the same purpose as those used in the class implementation of the agents, introduced in subsection 10.2.1. Further, the variable shared_resource_size is required to determine the vectorsize of the traded bundles and the related market price vector. Additionally, the market price vector is initialized using the shared_resource_size and presented by the variable mkt_prices. All other class attributes are set immediately after the instantiation or during the running simulation.

```
class Dealer(object):

    def __init__(self, account_address, web3, dealer_contract, ↩
        shared_resource_size):
        self._account_address = account_address
        self._web3 = web3
        self._dealer_contract = dealer_contract
        self._resource_inventory = None
        self._trade = None
        self._shared_resource_size = shared_resource_size
        self._mkt_prices = np.zeros(self._shared_resource_size)
        self._order_handler = None
```

Listing 13: Overview of the off-chain dealer's class attributes

Next, some of the most important functions of the class are introduced and explained. To begin with, the function to retrieve the orders of the agents from the dealer's smart contract is introduced in the listing 14.

```
def get_orders(self):
    self._order_handler = utils.OrderHandler()
    order_indices = self.get_order_indices()

    # get all orders from contract and store in order handler
    for order_id in order_indices:
        order = self._dealer_contract.contract.functions.getOrder(order_id)↩
            .call()
        self._order_handler.add_order(order_id, order)
```

Listing 14: Retrievement of orders

The variable order_handler is initialized with an object of the class OrderHandler. We developed the class OrderHandler to store and manage the orders of the agents' more easily on the part of the off-chain dealer. It contains a pool for the account adresses of the agents and the respective orders. In addition, the object provides functions to calculate all relevant attributes regarding the trades of the belonging orders, like the prepayment and the refunds. Then, the function of the smart contract is called to retrieve the order indices, which is stored in the variable of the same name. This variable is used to access the individual orders of the smart contract. Afterwards, all orders from the contract are retrieved iteratively and stored in the order_handler.

Furthermore, the class also provides functions for creating and solving the *market matching problem*. Due to the focus on the architectural design and implementation of the simulation and the high complexity of this functions, they are not presented in detail. However, the corresponding source code can be found in the appendix A.1.

Finally, the function to set the trades of the agents in the contract is presented in the listing 15. At the beginning, several function are invoked which initiate the calculation of trades and their related attributes. All these calculations take place in the object `OrderHandler`, which has already been presented. First, the calculation of the trade share is called. As stated earlier, due to the solution of the *market matching problem*, the agents often just get shares of their requested bundles. In this step, these shares for each settled order are set. Then, the calculation of the respective trades is initiated. Then again, the calculation of the remaining attributes, such as the prepayment, the bill, and the refund, is initiated. After that, all required attributes of the trades are calculated.

```
    def set_trades(self):
        self.set_trade_share()
        self.initiate_trade_calculation()
        self.initiate_prepayment_calculation()
        self.initiate_bill_calculation()
        self.initiate_refund_calculation()

        for order in self._order_handler.get_all_orders():
            account, trade, prepayment, bill, refund = order.↩
                get_trade_information()
            prepayment = utils.from_ether_to_wei(prepayment)
            bill = utils.from_ether_to_wei(bill)
            refund = utils.from_ether_to_wei(refund)

            self._dealer_contract.contract.functions.setTrade(account, trade, ↩
                prepayment, bill, refund).transact({'from': self.↩
                _account_address})
```

Listing 15: Submitment of trades

Finally, the respective orders are set iteratively in the dealer's smart contract. For each existing order, the related values of `prepayment`, `bill` and `refund` are again converted from Ether to Wei. Then, the object of the smart contract dealer is used to invoke the contracts function to set the trades.

10.2.4 Blockchain

In this part the used blockchain is described. At the beginning, in section 10.1, we already stated that we used a personal blockchain called Ganache. This personal Ethereum blockchain acts like a full client, which are described in section 6.2.6. However, Ganache has even more features that allows you to customize the properties of the Blockchain. First, you can specify the number of generated accounts and the assigned amount of ether. Further, it is possible to adjust the blocktime. The blocktime defines the time it takes to mine a new block. Additionally, it provides the feature to instantly mine a new block for every transaction. This is very useful for testing and debugging applications like the developed simulation. Likewise, it is possible to specify the price of gas in Wei and the gas limit of a block. Hence, the used blockchain Ganache allows you to recreate many scenarios of different blockchain behavior and is therefore perfectly suited for the application of a simulation.

10.3 Simulation process

To begin with, this subsection presents the whole simulation process. In contrast to the previous subsection 10.2, that considered the individual components in isolation, this subsection focus on the interaction of the various components.

Further, we divided the simulation process into the two parts *Initial Setup*, and *Main simulation loop*, which are described in the following.

10.3.1 Initial setup

Above all, the initial setup part is shown in the listing 16. The displayed source code is executed directly before the source code from listing 17.

```
# general setup of simulation
var = mem.Variables()
lp.decompose(var)

# set initial inventory and market prices
var.dealer.set_resource_inventory()
var.dealer.set_mkt_prices()
```

Listing 16: Initial setup of the simulation

Firstly, an object of the class `Variables` is instantiated. This class object can be seen as a kind of global memory. It contains all fundamental variables which are required to run the simulation. These variables are described in more detail in the following list.

`central_problem`: Represents the specified central optimization problem of the presented *BTM*. It is divided into subproblems and distributed to the existing agents. It is important to note that the number of variables present in the problem can be exactly divided by the number of agents. Otherwise an exception is thrown.

`web3`: Represents the object of the Python library, which enables the interaction with an Ethereum blockchain. It is the same object that is passed to the agent classes and the off-chain dealer class.

`dealer_contract`: Represents the object of the dealer's smart contract. It is the same object that is passed to the agent classes and the off-chain dealer class. It contains all implemented functions of the dealer's smart contract. When the contract is deployed and thus anchored in the blockchain, a JSON file is created. This file is denoted as the Contract Application Binary Interface (ABI). The Contract ABI is used to instantiate the Python object.

`agent_pool`: Represents a list containing all existing agent objects. Therefore all existing accounts are read from the blockchain and used to instantiate the agents.

`dealer`: Represents the object of the off-chain dealer, which is instantiated with the remaining account.

Next, the function for decomposing the central problem is called. The class object `Variables` is passed to it as parameter. This is needed to reach the central problem, the agent pool and thus the amount of all existing agents, and the object of the off-chain dealer. Then, the central problem is decomposed into subproblems which are distributed among the agents. The distrituion step also includes the distribution of the shared resources. Further, the allocation of the shared resources is implemented in such a way that all parties receive equal shares.

Furthermore, the off-chain dealer sets the initial values of the resource inventory and the market prices in the dealer's smart contract. As stated before, the the initial resource inventory is calculated in the decomposition step of the central problem. We also mentioned earlier in 10.2.3, that the market price vector is initialized when instantiating the off-chain dealer class. The size of the market price vector is equal to the size of the shared resources and is initially specified with zeros.

Finally, a few simulation parameter are set, which are needed to determine the termination of the simulation. These are self-explanatory and not essential for understanding the simulation process, wherefore they are not shown in the listing 16. Afterwards, the mail simulation begins.

10.3.2 Main simulation loop

First of all, the source code of the *Main simulation loop* is outlined in the listing 17. These loop represents the main process of the simulation and takes place after the part *Initial Setup*. The simulation loop is implemented by a while loop, which only terminates if the market prices from the previous round are equal to the market prices of the current round. Conversely, this means that the simulation runs as long as the market prices of the last two rounds are different. Additionally, there are comments in the source code of listing 17. The inline comments to the right of the function calls indicate if the invoked function constitutes a `call()` or a `transact()` method. As described earlier, the `call()` method invokes a smart contract function on a read-only basis, whereas the `transact()` method submits a verified transaction that change the state of the blockchain.

```
while(not equal_market_prices):

    # agent bidding
    for agent in var.agent_pool:
        agent.get_mkt_prices()  # call
        agent.determine_bundle_attributes()
        agent.set_order()  # transact

    utils.wait_for_new_block(var)

    # dealers market matching mechanism
    var.dealer.get_orders()  # call
    var.dealer.create_mmp()
    var.dealer.solve_mmp()
    var.dealer.set_trades()  # transact
    var.dealer.delete_order()  # transact
    var.dealer.set_mkt_prices()  # transact
    var.dealer.set_mmp_attributes()  # transact

    utils.wait_for_new_block(var)

    # agent trade verification
    for agent in var.agent_pool:
        agent.verify_strong_duality()  # call
        agent.accept_trade()  # transact
        agent.add_trade_to_shared_resources()

    utils.wait_for_new_block(var)

    var.dealer.recalculate_resource_inventory() # transact
```

Listing 17: Main loop of the simulation

The behavior of a blockchain differs from that of a conventional client-server architecture. In a blockchain environment, the process of a `transact()` method call is not immediately completed. The network verification requires some time due to the mining procedure. Consequently, before the values written into the blockchain by a `transact()` method are accessible, a certain time passes. Hence, it is important to take this into account if you want to read certain values from the blockchain that you set in previous steps. Therefore, we implemented a function called `wait_for_new_block()`. This function simply waits for the generation (mining) of a new block and is invoked whenever it is needed to wait for new values to be finally anchored in the blockchain.

In addition, the *Main simulation loop* can be divided into the three main activities *agent bidding*, *dealer's market matching mechanism*, and *agent trade verification*, which are also marked with a comment in the listing 17. These main activities are thematically summarised function calls. Besides, the above described function `wait_for_new_block()` is called between these activities to ensure an faultless process. In the following each of the three activities are described.

Agent bidding: To start with, this activity refers to the section 8.2.1 of the same name and represents the technical implementation. Further, a sequence of all function calls for this activity is provided in figure 10.

Above all, the steps described below are performed for all existing agents. This is indicated by the specified for-loop, shown in the beginning of listing 17.

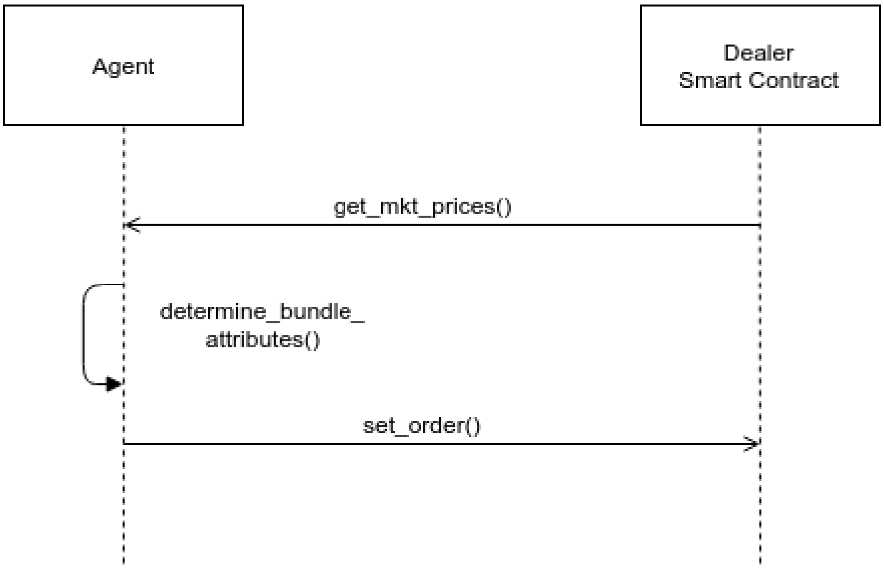

Figure 10: Sequence of agent bidding

First, the agent get the current market prices from the dealer's smart contract. To obtain the market prices, the function `getMktPrices()` of the dealer's smart contract is utilized. The market prices are required for the determination of the prefered bundle set.

Next, the agent solves the bundle determination and initialize the class attributes `bundle_set` and `bid`. In turn, they are required for the next function call. The corresponding implementation of this function is presented in listing 4.

Finally, the agent sets the order in the dealer's smart contract. To place the order in the contract, the function `setOrder()` of the contract is utilized. The implementation of the dealer's smart contract function is shown in the listing 10. Moreover, the implementation of the function `set_order()` of the agent is also introduced in the listing 5.

Dealer's market clearing mechanism: To begin with, this activity refers to the section 8.2.2 and can be seen as the technical implementation. Further, a sequence of all function calls for this activity is provided in figure 11.

First, the off-chain dealer gets the orders from the dealer's smart contract. Therefore, the function `get_orders()` of the agent (introduced in listing 14) utilizes the function `getOrder()` of the contract.

Then, the off-chain dealer creates and solves the market matching problem. The implementation of both functions is outlined in appendix A.1.

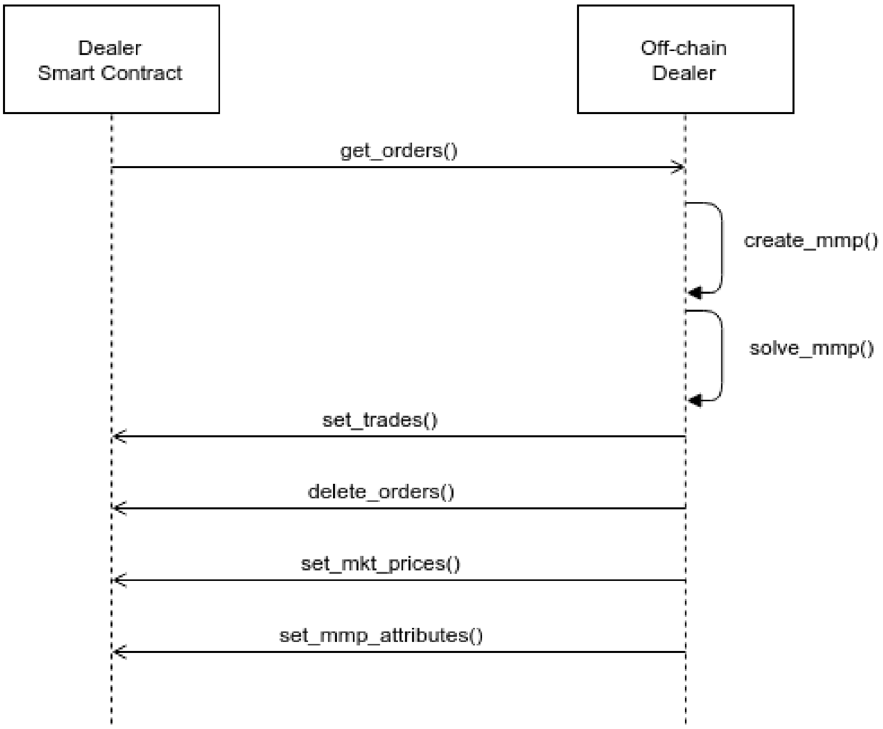

Figure 11: Sequence of dealer's market matching mechanism

After solving the market matching problem and the calculation of the respective trades, the off-chain dealer sets the trades in the dealer's smart contract. Therefore, the function `set_trade()` of the off-chain dealer (introduced in listing 15) utilizes the function `setTrade()` of the dealer's smart contract.

Afterwards, the off-chain dealer deletes the settled orders out of the dealer's smart contract. To do this, the function `delete_order()` of the off-chain dealer utilizes the function `deleteOrder()` of the dealer's smart contract.

In addition, the off-chain dealer sets the new market prices in the dealer's smart contract. To set the new market prices, the function `set_mkt_prices()` of the off-chain dealer utilizes the function `setMktPrices()` of the dealer's smart contract.

Finally, the values of the market matching problem are set in the dealer's smart contract. They are used by the agents to verify the correctness of the calculated market prices. Therefore, the function `set_mmp_attributes()` of the off-chain dealer utilizes the function `setMMPAttributes()` of the dealer's smart contract.

Agent trade verification: To start off, this activity covers the entire trade verification step of the agent. Likewise, a sequence of all function calls for this activity is provided in figure 12.

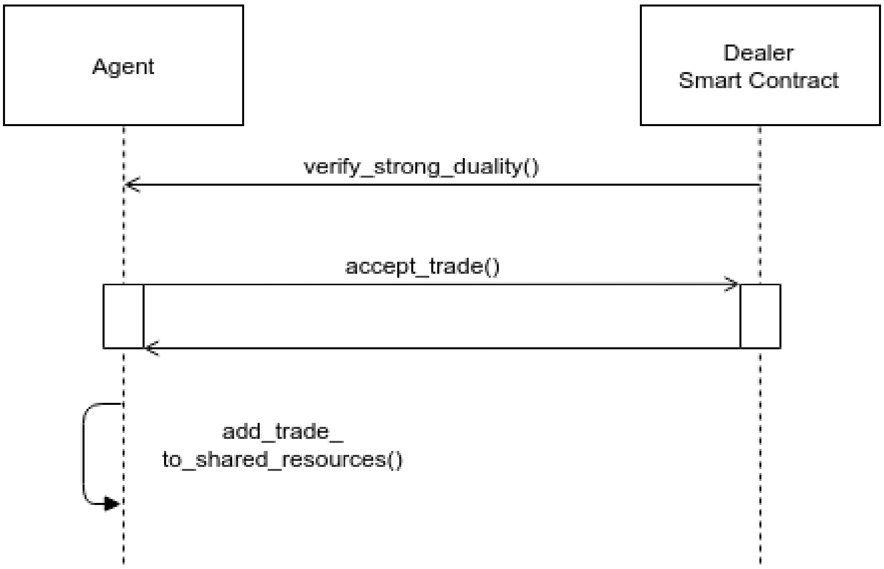

Figure 12: Sequence of agents trade verification

First, the agent verifies if the conditions of the *Strong Duality Theorem* are filfilled. Therefore, the function `verify_strong_duality()` of the agent (introduced in listing 6) utilizes the function `getMMPAttributes()` of the dealer's smart contract. As mentioned ealier in section 8.2.1, this function determine the class variable `accept_trade` of type boolean, which indicates whether the trade is accepted or not.

Afer that, the agent informs the dealer's smart contract whether the trade is accepted or rejected. For this, the function `accept_trade()` of the agent (introduced in listing 7) utilizes the function `acceptTrade()` of the dealer's smart contract (introduced in listing 12). In case of acceptance, the dealer's smart contract transfer the refunds to the agents and returns the respective bundle. Otherwise, in case of rejection, dealer's smart contract transfer the whole prepayment and deletes the respective trade.

Finally, the agent adds the trade to the shared resources. The invokation of this function only has an effect if the trade was previously accepted.

At the end of each iteration, the resource inventory of the dealer is recalculated. Therefore the off-chain dealer calculates how much of the inventory is used for the trades and settles this with the previous inventory. Finally, the new calculated inventory is set in the dealer's smart contract.

A Appendix

A.1 Additional Off-chain Dealer Implementation

```
    def create_mmp(self):
        bundles = [order.get_concatenated_bundles() for order in self.↩
            _order_handler.get_all_orders()]
        bids = [order.get_concatenated_bids() for order in self._order_handler.↩
            get_all_orders()]

        try:
            TARGET_COEFS = np.hstack(bids) * (-1)  # create target coef vector

            self._mmp_amount_variables = np.size(TARGET_COEFS)  # set amount of↩
                 variables
            mmp_coefs = np.hstack(bundles)
            var_leq_one_coefs = np.identity(self._mmp_amount_variables, dtype=↩
                float)  # create constraint matrix for y<=1
            var_geq_zero_coefs = np.identity(self._mmp_amount_variables, dtype=↩
                float) * (-1)  # create constraint matrix for y>=0
            mmp_bounds = self._resource_inventory
            var_leq_one_bounds = np.ones(self._mmp_amount_variables, dtype=↩
                float)
            var_geq_zero_bounds = np.zeros(self._mmp_amount_variables, dtype=↩
                float)

            CONSTRAINT_COEFS = np.concatenate((mmp_coefs, var_leq_one_coefs, ↩
                var_geq_zero_coefs), axis=0)  # create final constraint matrix
            CONSTRAINT_BOUNDS = np.concatenate((mmp_bounds, var_leq_one_bounds,↩
                 var_geq_zero_bounds))  # create final bounds matrix

            self._mmp_constraint_coefs = CONSTRAINT_COEFS
            self._mmp_constraint_bounds = CONSTRAINT_BOUNDS
            self._mmp_target_coefs = TARGET_COEFS

        except ValueError as error:
            print('Creation of MMP failed!')
            print(error)
```

Listing 18: Creation of MMP

```
    def solve_mmp(self):
        solvers.options['show_progress'] = False
        sol = solvers.lp(matrix(self._mmp_target_coefs), matrix(self.↩
            _mmp_constraint_coefs), matrix(self._mmp_constraint_bounds))
        self._mmp_values = np.array([float('%.2f' % (sol['x'][i])) for i in ↩
            range(self._mmp_amount_variables)])
        self._mmp_duals = np.array([float('%.2f' % entry) for entry in sol['z'↩
            ]])
        self._mkt_prices = np.array([float('%.2f' % (sol['z'][i])) for i in ↩
            range(self._shared_resource_size)])
```

Listing 19: Solving of MMP

References

Aitzhan, N. Z., & Svetinovic, D. (2018). Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. *IEEE Transactions on Dependable and Secure Computing*, *15*(5), 840–852.

Al Kawasmi, E., Arnautovic, E., & Svetinovic, D. (2015). Bitcoin-based decentralized carbon emissions trading infrastructure model. *Systems Engineering*, *18*(2), 115–130.

Ampatzis, M., Nguyen, P. H., & Kling, W. (2014). Local electricity market design for the coordination of distributed energy resources at district level. In *Innovative smart grid technologies conference europe (isgt-europe), 2014 ieee pes* (pp. 1–6).

Andoni, M., Robu, V., Flynn, D., Abram, S., Geach, D., Jenkins, D., ... Peacock, A. (2019). Blockchain technology in the energy sector: A systematic review of challenges and opportunities. *Renewable and Sustainable Energy Reviews*, *100*, 143–174.

Antonopoulos, A. M., & Wood, G. (2018). *Mastering ethereum: building smart contracts and dapps.* O'Reilly Media.

Bichler, M. (2017). *Market design: A linear programming approach to auctions and matching.* Cambridge University Press.

Buterin, V., et al. (2013). A next-generation smart contract and decentralized application platform. *Ethereum White Paper*.

Calculating Costs in Ethereum Contracts - HackerNoon.com. (2019). Retrieved 2019-07-10, from `https://hackernoon.com/ether-purchase-power-df40a38c5a2f`

Chen, X., Wei, T., Hu, S., & Member, S. (2013). Uncertainty-Aware Household Appliance Scheduling Considering Dynamic Electricity Pricing in Smart Home. *IEEE TRANSACTIONS ON SMART GRID*, *4*(2). Retrieved from `http://ieeexplore.ieee.org.` doi: 10.1109/TSG.2012.2226065

Fan, M., Stallaert, J., & Whinston, A. B. (2003). Decentralized mechanism design for supply chain organizations using an auction market. *Information Systems Research*, *14*(1), 1–22.

Griva, I., Nash, S. G., & Sofer, A. (2009). *Linear and nonlinear optimization* (Vol. 108). Siam.

Guo, Z., Koehler, G. J., & Whinston, A. B. (2007). A market-based optimization algorithm for distributed systems. *Management Science*, *53*(8), 1345–1358.

Guo, Z., Koehler, G. J., & Whinston, A. B. (2012). A computational analysis of bundle trading markets design for distributed resource allocation. *Information Systems Research*, *23*(3-part-1), 823–843.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2008). Design science in information systems research. *Management Information Systems Quarterly*, *28*(1), 6.

Jimeno, J., Anduaga, J., Oyarzabal, J., & de Muro, A. G. (2011). Architecture of a microgrid energy management system. *European Transactions on Electrical Power*, *21*(2), 1142–1158.

Ketter, W., Peters, M., Collins, J., & Gupta, A. (2015). Competitive benchmarking: an is research approach to address wicked problems with big data and analytics.

Light-Client-Protocol. (2019). Retrieved from `https://github.com/ethereum/wiki/wiki/Light-client-protocol`

Long, C., Wu, J., Zhang, C., Cheng, M., & Al-Wakeel, A. (2017). Feasibility of peer-to-peer energy trading in low voltage electrical distribution networks. *Energy Procedia*, *105*, 2227–2232.

Luenberger, D. G. (2008). *Linear and Nonlinear Programming 4TH* (Tech. Rep.). Retrieved from `http://www.springer.com/series/6161`

March, S. T., & Smith, G. F. (1995). Design and natural science research on information technology. *Decision support systems*, *15*(4), 251–266.

Medium, Ethereum-Clients. (2019). Retrieved from `https://medium.com/@eth.anBennett/ethereum-clients-101-beginner-geth-parity-full-node-light-client-4bbd87bf1dee`

Medium, Life Cycle of Transactions. (2017). Retrieved from `https://medium.com/blockchannel/life-cycle-of-an-ethereum-transaction-e5c66bae0f6e`

Mengelkamp, E., Gärttner, J., Rock, K., Kessler, S., Orsini, L., & Weinhardt, C. (2018). Designing microgrid energy markets: A case study: The brooklyn microgrid. *Applied Energy*, *210*, 870–880.

Mengelkamp, E., Notheisen, B., Beer, C., Dauer, D., & Weinhardt, C. (2018). A blockchain-based smart grid: towards sustainable local energy markets. *Computer Science-Research and Development*, *33*(1-2), 207–214.

Merkling in Ethereum. (2015). Retrieved from `https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/`

Mihaylov, M., Jurado, S., Avellana, N., Van Moffaert, K., de Abril, I. M., & Nowé, A. (2014). Nrgcoin: Virtual currency for trading of renewable energy in smart grids. In *European energy market (eem), 2014 11th international conference on the* (pp. 1–6).

Modified Merkle Patricia Trie Specification. (2015). Retrieved from `https://github.com/ethereum/wiki/wiki/Patricia-Tree`

Proof of Work. (2019). Retrieved from `https://en.bitcoin.it/wiki/Proof_of_work`

Sikorski, J. J., Haughton, J., & Kraft, M. (2017). Blockchain technology in the chemical industry: Machine-to-machine electricity market. *Applied Energy*, *195*, 234–246.

Solidity Documentation. (2019). Retrieved from `https://solidity.readthedocs.io/en/develop/types.html#fixed-point-numbers`

Vanderbei, R. J., et al. (2015). *Linear programming.* Springer.

Vukolić, M. (2015). The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International workshop on open problems in network security* (pp. 112–125).

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, *151*, 1–32.

Zhang, C., Wu, J., Long, C., & Cheng, M. (2017). Review of existing peer-to-peer energy trading projects. *Energy Procedia*, *105*, 2563–2568.

Zheng, Z., Xie, S., Dai, H., Chen, X., & Wang, H. (2017). An overview of blockchain technology: Architecture, consensus, and future trends. In *Big data (bigdata congress), 2017 ieee international congress on* (pp. 557–564).

Zheng, Z., Xie, S., Dai, H.-N., & Wang, H. (2016). Blockchain challenges and opportunities: A survey. *Work Pap.–2016*.