Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Taneli Hukkinen

Reducing blockchain transaction costs in a distributed energy market application

Master's Thesis
Espoo, January 17, 2018

Supervisor:	Professor Kari Smolander
Advisor:	Professor Timo Seppälä

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Taneli Hukkinen		
Title: Reducing blockchain transaction costs in a distributed energy market application			
Date:	January 17, 2018	**Pages:**	67
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Kari Smolander		
Advisor:	Professor Timo Seppälä		

This thesis explains the working of a previously undocumented blockchain application developed for the energy sector. The application enables distributed market coordination for small-scale decentralized energy systems. An Ethereum smart contract is employed as a core component of the application, facilitating a marketplace for transacting electrical energy.

A design science research methodology was applied to the application in an attempt to further develop it. The problem of high fees in the energy marketplace was identified, resulting from the smart contract's inefficient use of Ethereum gas. Two particular sources of inefficiency were identified, and solutions for fixing these inefficiencies were designed and implemented. Savings in transaction fees were created by replacing a function of the smart contract with off blockchain communication, and by editing the fund withdrawal mechanism of the smart contract so that it requires users to create fewer blockchain transactions. As a result, the smart contract's gas consumption was reduced by up to 11% in a certain use case.

The reduction in gas consumption was not sufficient to make the deployment and use of the application economically feasible on the canonical public Ethereum blockchain. A Plasma child chain or a dedicated Ethereum blockchain were suggested as potentially more feasible deployment environments for the application.

It was noted that the application relies on centralized components, and it is debatable whether its current blockchain-based implementation is justifiable.

Keywords:	blockchain, Ethereum, Solidity, decentralized applications, smart contracts, energy industry, distributed marketplace
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Taneli Hukkinen		
Työn nimi: Erään hajautetun energiamarkkinasovelluksen lohkoketjutransaktiokulujen vähentäminen			
Päiväys:	17. tammikuuta 2018	**Sivumäärä:**	67
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Professori Kari Smolander		
Ohjaaja:	Professori Timo Seppälä		

Tämä diplomityö selostaa erään energiasektorille kehitetyn, aiemmin dokumentoimattoman lohkoketjusovelluksen toiminnan. Sovellus mahdollistaa hajautetun markkinakoordinoinnin toteuttamisen vaihtoehtoisille mikrosähköverkossa toimiville energiajärjestelmille. Sovelluksen keskeisenä komponenttina on Ethereum-alustalle toteutettu älykäs sopimus, joka hallitsee kaupankäyntiä energiamarkkinapaikalla.

Työssä sovellettiin erästä *design science* -tutkimusmenetelmää lohkoketjusovellukseen, tarkoituksena kehittää sovellusta. Havaittiin, että vuorovaikutus sovelluksen älykkään sopimuksen kanssa on tarpeettoman kallista suurten lohkoketjutransaktiokulujen vuoksi. Älykkään sopimuksen toteutuksesta tunnistettiin kaksi lähdettä suurille kustannuksille. Näille ongelmakohdille suunniteltiin ja toteutettiin korjaukset. Korjaukset pohjautuivat älykkään sopimuksen erään funktion korvaamiseen lohkoketjun ulkopuolisella kommunikaatiolla, sekä älykkään sopimuksen varojen kotiutusmekanismin muokkaamiseen siten, että se vaatii käyttäjää tekemään vähemmän lohkoketjutransaktioita. Lopputuloksena älykkään sopimuksen kanssa toimimisesta syntyvät transaktiokulut vähenivät eräässä käyttötapauksessa 11%.

Transaktiokulut eivät pienentyneet niin merkittävästi, että julkisen Ethereum-lohkoketjun käyttö sovelluksen alustana olisi transaktiokustannusten kannalta järkevää. Mahdollisina vaihtoehtoisina alustoina sovellukselle ehdotimme Plasma-lapsiketjua tai omaa erillistä Ethereum-lohkoketjua. Näissä vaihtoehdoissa transaktiokulut saattaisivat olla siedettävämpiä.

Työssä huomattiin, että energiamarkkinasovellus tukeutuu keskitettyihin osiin, joten on kyseenalaista, onko sen nykyinen lohkoketjupohjainen toteutus perusteltu.

Asiasanat:	lohkoketju, Ethereum, Solidity, desentralisoidut sovellukset, älykkäät sopimukset, energia, markkinapaikka
Kieli:	Englanti

Acknowledgements

Blockchain technology, with its enormous disruptive potential, and the rate at which the technology evolves, feels like a deep rabbit hole. Researching the field has been exciting and absorbing, often keeping me up until the early hours. It does feel, however, great to finally write these last words of this thesis, and consider climbing out of the blockchain rabbit hole.

There are a few people who deserve a special mention for making the completion of this thesis possible.

First, I gratefully thank my advisor Timo Seppälä, for giving me the opportunity to join the BOND (Blockchains Boosting Finnish Industry) project and to pursue my Master's thesis in a research area that I have been fascinated by for years.

I would also like to thank my supervisor Kari Smolander for giving me constructive feedback and steering this thesis in the right direction.

Juri Mattila and Juuso Ilomäki I'd like to thank for collaborating with me to produce the ETLA report that laid the foundations for this work. Thank you also for all the inspiring and fruitful discussions we had about blockchain technology throughout the year.

Lastly, thank you mom for all the support and encouragement.

Espoo, January 17, 2018

Taneli Hukkinen

Contents

1 Introduction **7**
1.1 Problem statement . . . 9
1.2 Research goals . . . 9
1.3 Structure of the thesis . . . 9

2 Background **11**
2.1 Electricity markets today and in the future . . . 11
2.2 Bitcoin — the first blockchain-based distributed ledger . . . 13
2.2.1 Transacting . . . 14
2.2.2 Blockchain as a data structure . . . 16
2.2.3 Mining and supply control . . . 17
2.2.4 Permissioned and permissionless blockchains . . . 18
2.2.5 Off-chain transactions . . . 19
2.3 Smart contracts . . . 20
2.4 Decentralized applications . . . 21
2.5 Ethereum . . . 22
2.5.1 Control of accounts . . . 22
2.5.2 Smart contracts . . . 23
2.5.3 Transactions . . . 23
2.5.3.1 Fee system . . . 23
2.5.3.2 Optional data field . . . 24
2.5.3.3 Messages . . . 24
2.5.4 Ethereum Virtual Machine . . . 25

3 A blockchain energy market **26**
3.1 The process of transacting energy . . . 27
3.2 Nanogrid characteristics . . . 29
3.3 Sale and purchase agreements . . . 31
3.4 Smart meters . . . 32
3.5 Reputation system . . . 34
3.6 Smart contract . . . 35

3.6.1 Public functions 35
3.6.1.1 makeOffer 35
3.6.1.2 acceptOffer 35
3.6.1.3 buyerReport and sellerReport 36
3.6.1.4 withdraw 36
3.7 Scalability 37

4 Applying a design science research methodology 38
4.1 Problem identification and motivation 39
4.2 Objectives 40
4.2.1 Methods for reducing gas consumption of an Ethereum smart contract 40
4.2.2 Identifying inefficiencies 41
4.3 Design and development 42
4.3.1 Off-chain offers 42
4.3.2 Withdrawing from multiple trades 44
4.3.2.1 First implementation 45
4.3.2.2 Second implementation 45
4.3.3 Developed artifacts 45
4.4 Evaluation 46
4.4.1 Benchmark use case 46
4.4.2 Environment 47
4.4.3 Results 47
4.5 Communication 48
4.5.1 Feasibility 52
4.5.1.1 Proposed solutions 52

5 Discussion and conclusions 54
5.1 ETLA energy market as a decentralized application 54
5.2 Contributions 55
5.3 Limitations of this work 56
5.4 Future research 56

A Source code of Artifact 4 60

Chapter 1

Introduction

Digital platforms, services and applications have traditionally been built using a centralized architecture, in which a single party controls the system. Recently emerged blockchain technology has enabled the development of applications that have no single entity governing and controlling the application. Instead, these decentralized applications are collectively governed by a peer-to-peer network and its anonymous nodes.

The first blockchain-enabled decentralized application, from which the entire branch of technology originated from, was a digital cash and payment system named Bitcoin. Since Bitcoin's birth in 2009, a range of other blockchain-based decentralized applications have been developed. Early blockchain applications, such as the distributed cloud storage named Sia, needed to deploy a specialized blockchain network of their own, to facilitate the application's logic.

Perhaps the most remarkable use for blockchain technology thus far, along with digital cash, has been smart contract platforms. These platforms allow the deployment of arbitrary program code to the platform's database, and the execution of these pieces of program code in a distributed and replicated manner by the network's nodes. The pieces of code executed on the platform are referred to as smart contracts. The first and most notable such platform is Ethereum, that was launched in 2015.

Platforms like Ethereum made it possible to create decentralized applications in the form of smart contracts. Smart contracts were able to use the shared platform's network and database, so there was no more a need to deploy a separate specialized blockchain instance for each new application. As a result, decentralized applications could be more developed more rapidly, they could interact with each other and they were able to benefit from the shared network's strong security.

Some prominent smart contract use cases have already been discovered.

The most noteworthy examples include decentralized autonomous organizations (DAOs), initial coin offerings (ICOs) and decentralized marketplaces. DAOs are organizations that operate according to rules encoded in a smart contract (Chohan, 2017). ICOs are a startup fundraising model, where instead of company shares, investors receive tokens that may, for instance, grant the right to royalties in the funded project or have utility value in it. ICOs have largely taken the place of venture capital in blockchain related startups (Dell'Erba, 2017). Decentralized marketplaces enable trading of assets stored in a blockchain. A marketplace smart contract is capable of intermediating all trades, meaning that no trusted middleman is required. A token exchange named EtherDelta is possibly the most widely used blockchain-based smart contract thus far, having handled seven percent of all Ethereum transactions over the past ten months[1].

Applications like DAOs, ICOs and decentralized marketplaces give reason to believe that blockchain technology may be able to significantly lessen the role of central authorities and intermediaries in our society. The technology is expected to have this effect on fields such as communications, business, law and politics. Some see the potential implications of blockchain technology so significant that its emergence has even been compared to the birth of the Internet. (Wright and De Filippi, 2015)

In report number 71 of ETLA (The Research Institute of the Finnish Economy) Hukkinen et al. (2017) explain that they have developed a blockchain application that enables distributed market coordination for decentralized energy systems. The application's core element is an Ethereum smart contract that matches individual producers and consumers of energy and lets the matched pairs exchange energy for cryptocurrency. Essentially, the smart contract facilitates a decentralized marketplace for energy. As part of the report, Hukkinen et al. (2017) published the source code of the application under the MIT license.

In this thesis, we will create an understanding of the energy market blockchain application. We also apply a design science research methodology to the energy market application in an attempt to understand potential problems that may arise in the deployment of the application and to further develop the application.

[1]The seven percent is based on the EtherDelta smart contract's transaction count between February 9th, 2017 and December 8th, 2017 (data fetched from `https://etherscan.io/address/0x8d12a197cb00d4747a1fe03395095ce2a5cc6819`), and the total number of Ethereum transactions made between the same dates (data fetched from `https://etherscan.io/chart/tx?output=csv`)

1.1 Problem statement

The report by Hukkinen et al. (2017) explains the energy market application from a rather narrow perspective, leaving many questions unanswered. The publication explains motives for developing the application, and discusses regulatory issues around it and how legislators should react to possible technological disruptions. The report does not, however, describe how the developed application functions from a technology perspective, neither does it explain the requirements that the application sets for the physical world environment where it is meant to be used for trading energy.

The blockchain energy market is an expensive platform to operate on. If the application were deployed on the public Ethereum blockchain, according to our estimate, a single trade would cost around $0.48 in transaction fees.[2] We expect there to be room for optimization in how many and how costly blockchain transactions are needed to use the application.

1.2 Research goals

The goal of this thesis is to describe the architecture and operating principles of the blockchain energy market application introduced in the report by Hukkinen et al. (2017).

Additionally, the design science research methodology (DSRM) by Peffers et al. (2007) will be applied to the Ethereum smart contract that facilitates the energy market. The purpose is to explore ways to improve and further develop the Ethereum smart contract. We expect improvement possibilities to be found in the smart contract's high transaction costs.

As a result of the application of the DSRM, variations of the smart contract's source code will be produced. Any knowledge and understanding acquired in the making of these artifacts will be reported in this thesis.

1.3 Structure of the thesis

This Master's thesis is structured as follows. Chapter 2 provides background for the thesis. It explains how electricity markets are currently structured, and predicts how increasing use of renewables may change these markets.

[2]The transaction cost was calculated using a gas price of 4 Gwei per gas and ether price of $300. The gas price of 4 Gwei is the median gas price of blocks 4512310 to 4513809 and $300 is the market price of ether on November 8, 2017. A successful trade of energy was estimated to cost 400 000 Ethereum gas.

It also describes Bitcoin and blockchain technology, smart contracts and Ethereum. These are vital concepts and technologies when attempting to understand how decentralized applications, such as the blockchain energy market, work.

Chapter 3 describes, from a technology perspective, how the blockchain energy market application presented in ETLA report number 71 functions.

Chapter 4 presents the main contribution of this thesis. It describes the application of the DSRM by Peffers et al. (2007) to the blockchain energy market application's Ethereum smart contract. The chapter documents the design and implementation of the improvements made to the smart contract in an attempt lower its transaction costs. The impact of these improvements is evaluated and their significance is discussed in the chapter.

Chapter 5 concludes this Master's thesis. The chapter discusses the contributions and limitations of this work and proposes directions for future work on the subject.

Chapter 2

Background

2.1 Electricity markets today and in the future

Due to electricity's non-storable nature, supply and consumption of power being fed into an electrical grid must constantly be balanced. With multiple producers and consumers interacting with the grid, finding the price for each moment, where demand and supply meet, is vital for grid balancing. In most recently developed European markets, price formation happens at power exchanges. One such exchange is Nord Pool, the largest electrical energy market in Europe. In Nord Pool and other liberalized wholesale electricity markets, electricity is traded much like any commodity. (Weron, 2007)

A range of power delivery contracts are used to keep the supply and demand in balance. The contracts may either be physical or financial contracts. In physical contracts, power and cash are delivered on expiry of the contract. In financial ones, only cash is delivered in such a way that the difference between the contractual price and index price are settled. Financial contracts may be used for hedging or speculation. Physical contracts are classified to long-term contracts and spot contracts. Long-term contracts may have maturities ranging to as long as years, while spot contracts are settled almost immediately. Contracts may include many kinds of terms, but they always share four essential characteristics: delivery period, delivery location, size and price. (Weron, 2007)

Long-term contracts are typically sold in bilateral, over-the-counter transactions, and spot contracts in an organized day-ahead market. In a day-ahead market, market participants such as generators, distributor companies, traders and large consumers, match supply and demand to determine a publicly announced market clearing price (MCP), commonly known as spot price. Based on bids and offers set by buyers and suppliers, an MCP is set for each hour of the following day so that the supply and demand meet. The MCP is a result of a two-sided auction illustrated in figure 2.1. The MCP is at the equilibrium point where the demand curve (constructed from aggregated demand bids), and the supply curve (constructed from aggregated supply bids) intersect. (Weron, 2007)

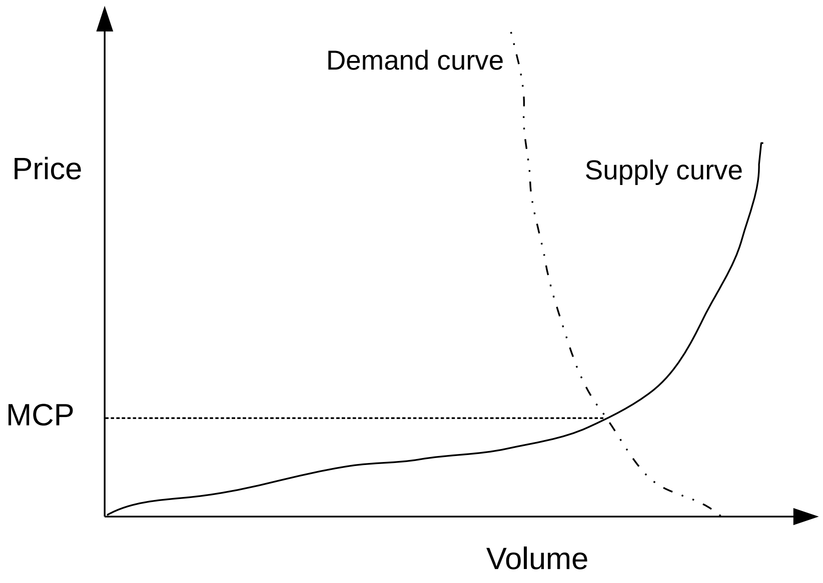

Figure 2.1: The two-sided auction of power exchanges

For balancing supply and demand for short time horizons, the transmission system operator operates a balancing market. The system operator is able to order extra production even in seconds, to keep the system in balance. (Weron, 2007)

Since electricity can not be economically stored, the described manifold of contracts is required to balance supply and demand in a transmission grid. (Weron, 2007)

An ongoing trend is that the share of energy generated by renewable methods, such as wind and photovoltaic energy systems, is increasing. In EU countries the percentage-share of renewable energy in gross final energy consumption has risen from 9 percent to 16.7 percent in a ten-year time span between the years 2005 and 2015 (European environment agency, 2017).

An advantage that solar energy and many other renewable systems provide is that they allow distributed generation near the points of demand, enabling reduced transmission losses (World Nuclear Association, 2017). The renewable energy trend may have the effect of transforming the current vertical centralized energy system structure into a more horizontal and distributed one.

Energy generation using nonrenewable energy resources is flexible compared to many renewable sources. Nonrenewable sources are, in general, less reliant on factors such as wind and sunlight conditions. This, accompanied by the price inelasticity of the demand side (Bye and Hansen, 2008), is why traditionally grid balancing has largely happened on the supply side. Future energy systems where supply consists largely of intermittent renewable energy sources may call for demand-side flexibility (Finn and Fitzpatrick, 2014). It has been shown that real-time pricing can cause a response in the demand side and be used to reduce curtailment (Finn and Fitzpatrick, 2014).

Assuming the renewable energy trend continues, future energy systems can be expected to be more distributed and flexible on the demand side. With photovoltaic systems and wind energy systems being accessible and affordable, generation of energy may shift from large energy companies to smaller organizations or consumers themselves. Small-scale electrical networks, microgrids and nanogrids, that can work islanded from the main grid, may grow more common. These small-scale electrical networks enable local generation, consumption and control.

2.2 Bitcoin — the first blockchain-based distributed ledger

Attempts to create a useful cryptography-based digital cash have been made for decades. In 1983 David Chaum invented blind signatures and as an application for them, he proposed a digital cash that allowed users to obtain money from banks and spend in a way that is untraceable to the bank and any other party (Chaum, 1983). Chaum's setting was a centralized system however, relying on a centralized intermediary to keep track of account balances and cryptographically sign the untraceable cash. Later, in 1998, Wei Dai proposed a concept of creating digital cash through solving computational work, and having a decentralized consensus of who owns the money (Dai, 1998). Dai did not, however, describe how the decentralized consensus could be achieved.

The digital currency named Bitcoin, introduced in 2008 by the pseudonym Satoshi Nakamoto (Nakamoto, 2008), was very similar to Dai's concept from ten years earlier. It was, however, revolutionary in the way that it proposed a working algorithm for decentralized, permissionless consensus referred to as "proof of work" or Nakamoto consensus. The algorithm was not only able to form a decentralized consensus on which transactions have occurred, but it also did it in a permissionless way with no admission control, letting anyone take part in forming the consensus. When previously a digital ledger required a centralized authority or a peer-to-peer network with a closed list of nodes to reach consensus, Bitcoin was capable of forming consensus without the help of either of those. This trait is invaluable for a payment system because it means that no single entity or a group of entities is in control of the users' money.

Bitcoin is the protocol and public ledger of a decentralized digital payment system. The system is based on a peer-to-peer network, where full nodes of the network store a copy of the history of transacted currency. Transactions created by users are propagated across the network and validated by the receiving nodes. The Bitcoin proof of work consensus algorithm is used to form a distributed consensus on which transactions are valid and included in the public ledger.

The unit of currency used in the system is called a bitcoin. A bitcoin is divisible to eight decimal places. The smallest unit that can be recorded in the Bitcoin ledger, a one hundred millionth of a bitcoin, is called a satoshi.

The term blockchain is used today when referring to the technology behind Bitcoin that enables decentralized, distributed and replicated digital ledgers (Swan, 2015). The technology consists of components such as a peer-to-peer network, public-key cryptography, digital cash, a decentralized consensus algorithm and an immutable chain of blocks used to store transactions. The inventor of Bitcoin, Satoshi Nakamoto, did not use the term blockchain in the Bitcoin whitepaper he wrote in 2008, but he did use it in the original source code of the first Bitcoin node.[1] In the source code, the term was used in the form "block chain" and was only used to refer to the data structure of Bitcoin transactions, that literally is a number of transaction blocks chained together sequentially. It is only later that the term has taken the broader meaning of the technology behind Bitcoin as a whole.

2.2.1 Transacting

The Bitcoin protocol uses public-key cryptography for controlling ownership of the Bitcoin cryptocurrency. Bitcoins are assigned to public keys, and knowing the corresponding private key allows transacting the bitcoins to

[1]The source code is available at `https://github.com/trottier/original-bitcoin` (accessed 4 January 2018).

another public key, thus spending the bitcoins.

A Bitcoin transaction has at least one transaction input and at least one transaction output. Each input adds some number of bitcoins to the transaction, thus increasing the number of bitcoins that can be spent in the outputs. The creator of the transaction decides how the input bitcoins are divided between the outputs. The number of bitcoins in transaction inputs must be equal or greater than the number of bitcoins in transaction outputs.

A transaction output contains two fields: a field that defines how many bitcoins belong to the output, and a field that holds a pubkey script. A pubkey script defines the rules that must be met for being allowed to spend bitcoins further. Typically the script is simply used to associate the bitcoins to a certain public key and allow the holder of a corresponding private key spend them further. Transaction outputs may be used as an input in one later transaction. Outputs that have not yet been spent are called ”unspent transaction outputs” (UTXOs).

In order to transact bitcoins, a user must create a valid Bitcoin transaction, broadcast it to the network and wait for the Bitcoin network to include it in the public ledger. New blocks of transactions are constructed and appended to the chain in a process called mining. In mining, network nodes that are participating in the block-creation process (called miners), race each other for who gets to create the next valid block and have it propagated to the network. The miner who succeeds in this task is awarded the transaction fees of all transactions included in the block, as well as an amount of newly minted bitcoins called the block reward.

Creating a valid block is a computationally difficult problem. A valid block must have a hash that starts with a certain number of zero bits. The number of zeros required depends on the network's global block difficulty. The network changes block difficulty every 2016 blocks in an attempt to keep the expected time for the miners to find a new block at ten minutes. To find a valid block, miners repetitively hash the block they are willing to create using a different nonce value. A nonce is an arbitrary 32-bit value in a block's header that miners can freely choose in order to alter a candidate block's hash. When a miner finally finds a block that satisfies the protocol's requirements, the block is propagated to the network, and the miners continue seeking for a block following the propagated one.

A transaction fee is often included in transactions, to incentivize miners to include the particular transaction in the blockchain. A block can only fit a limited amount of transactions so in case there are more unmined transactions than a miner is able to fit in the next block, the miner will likely only include transactions that pay the highest fees. The result is a fee market, in which transaction senders are able to control the time and probability of their

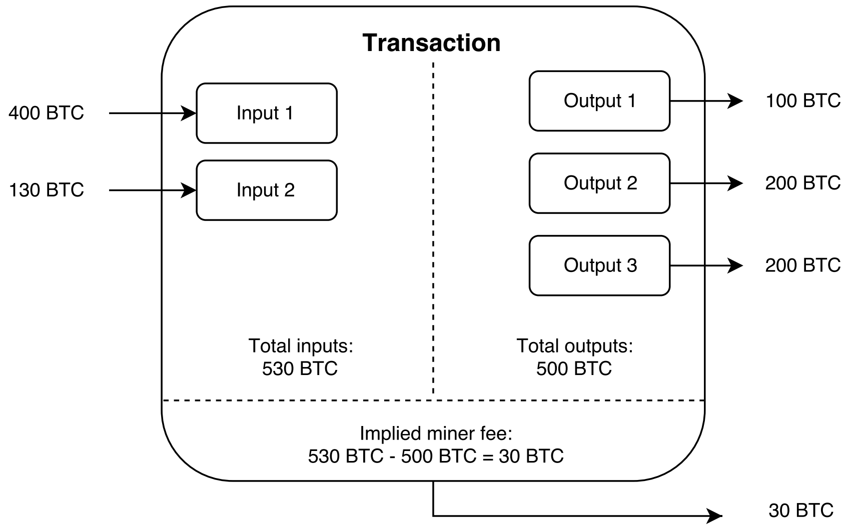

Figure 2.2: A Bitcoin transaction's inputs, outputs and the miner fee

transaction getting mined by changing the transaction fee. The amount of bitcoins in transaction inputs that exceeds the amount of bitcoins allocated to transaction outputs is considered the transaction fee, that the miner may direct to an address they control. Figure 2.2 illustrates a multi-input-multi-output transaction in Bitcoin, and how the implied miner fee is set.

2.2.2 Blockchain as a data structure

A blockchain can be viewed as a data structure that stores transactions. Transactions in a blockchain-based distributed ledger are aggregated into blocks and new blocks are appended into the blockchain when they are created through a process called mining. The blockchain data structure is replicated across nodes of the blockchain's network making it very resilient to node failures.

In Bitcoin, a block consists of two parts: a header and a payload. The header part, called the block header, consists of relevant pieces of information such as a timestamp and a cryptographic hash of the previous block. It also contains data related to the consensus mechanism and mining process, such as the difficulty target used for mining the block, and the nonce found by the miner to satisfy the difficulty target. The payload part consists of all transactions included in the block in question.

Each block is linked to its preceding block by storing the preceding block's hash in its header. The resulting chain of hashes is what ensures a blockchain's immutability. Each new block created confirms the validity of all of its preceding blocks. If any change were made in the content of any preceding block, it would invalidate the chain of hashes in the blocks following the modified block. Figure 2.3 represents a part of a simplified blockchain and the hash-based linkage from a block to its predecessor.

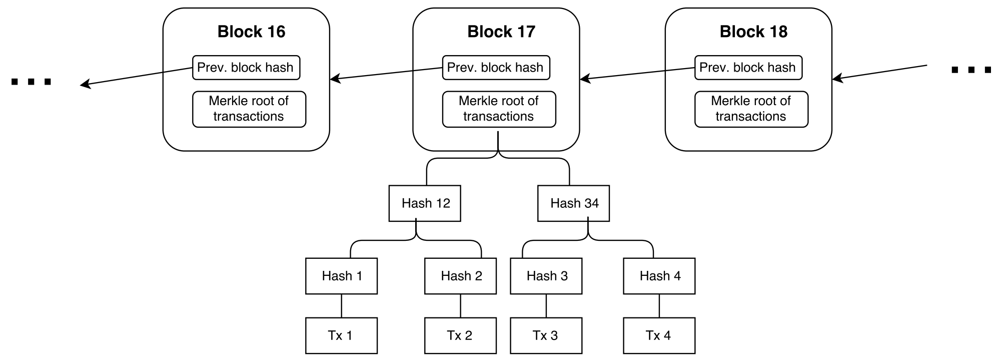

Figure 2.3: A simplified blockchain representation showing the hash-based linkage from a block to its predecessor, and the Merkle tree of a block's transactions.

The link between the header and the transactions of a block is created by including the root hash of the Merkle tree[2] of the transactions in the block header. This is depicted in figure 2.3. The advantage of this approach is that it enables Simplified Payment Verification (SPV) nodes. SPV nodes are light clients that only store block headers. The Merkle tree structure allows SPV nodes to check the validity of any transaction by requesting only the branch of the Merkle tree that leads from the Merkle root to the transaction. SPV nodes can be useful for users that are, for instance, only interested in the validity of UTXOs they may spend.

2.2.3 Mining and supply control

Mining is the process of adding new transactions to a blockchain. Following the rules of the blockchain's consensus mechanism, network nodes are allowed to propose new blocks to be added to the chain. The permission to create a new block and have it added to the public chain is valuable, due to a block's creator being allowed to collect transaction fees and a block reward. There needs to be a fair way to decide, who is eligible for creating the next block.

[2]A Merkle tree is a tree structure in which every non-leaf node is labeled with the hash of the labels of its child nodes.

In Bitcoin, this is decided by a consensus mechanism called proof of work, where the permission to create a block is granted to the entity that first solves a computationally difficult problem.

Mining solves a handful of problems for digital decentralized payment systems. First, it enables a permissionless and Sybil-proof[3] consensus mechanism and immutability of the distributed ledger. If the blockchain forks, the network considers the chain with the greatest total difficulty to be the canonical chain. The total difficulty, that is the total amount of processing work spent to create the chain, can be deduced from the number of blocks and the difficulty target of these blocks. This simple rule is used to form a consensus in the network. The consensus rule together with the linked chain of block hashes is what guarantees a blockchain's immutability. To alter the blockchain's data, the altered block and all blocks following it need to be regenerated. The probability of making a chain the canonical chain after such an alteration, without controlling more hashing power than what is backing up a competing fork, becomes exponentially more improbable the deeper the altered block is in the chain (Nakamoto, 2008).

Mining is also a way to create a fair initial distribution of coins and control the supply in a perfectly transparent way. In the form of the block reward, mining provides a way to issue new coins in a controlled and predetermined way that maintains scarcity of the digital coin. Bitcoin uses a monetary policy in which the amount of new coins issued in a new block started from 50 bitcoins, and is halved every 210 000 blocks.

2.2.4 Permissioned and permissionless blockchains

The BitFury Group and Garzik (2015) have classified blockchains into four categories based on access to blockchain data:

Public blockchain
: A blockchain where everyone has read access to the blockchain and may submit transactions to it.

Private blockchain
: A blockchain in which only entities included in a predefined list are able to read the blockchain and submit transactions to it.

[3]Being Sybil-proof means being resistant to Sybil attacks. A Sybil attack is an attack in which the attacker attempts to control a peer-to-peer system by creating and controlling a large number of identities (Douceur, 2002).

Permissionless blockchain
A blockchain in which there are no restrictions to who is allowed to create blocks.

Permissioned blockchain
A blockchain in which only identities included in a predefined list are allowed to create blocks.

Bitcoin being a permissionless blockchain means that no permission is required to contribute in forming a consensus of the ledger's state. Anyone with an internet access and some processing power is free and capable of doing so. Permissioned blockchains or consortium blockchains, such as Hyperledger Fabric, are the opposite of permissionless ones. They only allow a certain set of "whitelisted" nodes to create and validate blocks.

A cryptocurrency is an essential component of any permissionless blockchain (Swanson, 2015). It is needed to create an economic incentive for the pseudonymous validators of the network to run full nodes and build new blocks on top of the canonical chain, thus securing its immutability.

Due to the whitelisting of validators, permissioned blockchains do not need a similar economically incentivized consensus protocol and cryptocurrency as permissionless blockchains do. Since validators are known legal identities, they may be rewarded and penalized by other means. Permissioned blockchains also do not need protection against Sybil attacks, so their consensus protocols may vastly differ from those used in permissionless chains. (BitFury Group and Garzik, 2015)

Permissioned blockchains offer a controlled and predictable environment for transactions, but they undermine the core aspects of blockchain technology: decentralization and trustlessness. (BitFury Group and Garzik, 2015)

2.2.5 Off-chain transactions

An off-chain transaction is a valid blockchain transaction that is created and cryptographically signed by its creator but is not initially broadcast to the blockchain network. Instead, the transaction is distributed to relevant parties in an off blockchain communication channel. The transaction is broadcast to the blockchain network only if a party that is in possession of the transaction has an incentive to broadcast it.

An important benefit of off-chain transactions is that often an off-chain transaction never gets broadcast to the network, but is still meaningful to the application it is used in. Hence, the load on the blockchain is reduced and fewer transaction fees are paid.

A notable application that uses off-chain transactions is the Lightning payment channel network proposed by Poon and Dryja (2015). In Lightning network, two parties may open a shared payment channel that holds a bonded deposit. The parties may alter the share of the deposit belonging to each other by exchanging off-chain transactions.

2.3 Smart contracts

Smart contract is a term coined by Nick Szabo in his article Smart Contracts from the year 1994 (Szabo, 1994). Szabo (1994) defines a smart contract as "a computerized transaction protocol that executes the terms of a contract". In another article, Szabo describes smart contracts as contractual clauses that are embedded in software or hardware in a way that makes violation of the contract expensive for the violator (Szabo, 1997a). Essentially, a smart contract can be seen as a contract that is able to monitor the fulfillment of its terms without human intervention and is able to penalize or prevent a breach of contract.

The concept of a smart contract is best explained by an example. In an article from 1997, Szabo proposes that a vending machine can be seen as a primitive ancestor of a smart contract. A vending machine takes coins and a push of a button as an input and dispenses change and a product as output. The vending machine always acts deterministically according to the same set of instructions. Inserting coins into a machine is seen as a sign of agreement with the terms of the vending machine's embedded contract. The vending machine is able to autonomously manage the process of handling a customer's money and selling a product without an external adjudicator. (Szabo, 1997b)

For many years after Szabo proposed the idea of smart contracts, there were no remarkable implementations or experiments around the use cases where Szabo envisioned smart contracts to be useful. This was likely due to technology not yet having reached the level of Szabo's advanced theories (Lauslahti et al., 2017). Szabo set certain design objectives for smart contracts. One of the objectives was to minimize the need for trusted intermediaries (Szabo, 1994). He also mentioned observability and verifiability as core objectives (Szabo, 1997a). Observability is required in the sense that the parties involved need to be able to observe their own and others' performance of the contract. Verifiability means that the adjudicator must be able to verify whether a contract has been performed or breached.

The objectives were difficult to achieve at the time, but in recent years technologies such as Internet of Things and blockchain have helped in overcoming the obstacles that smart contracts previously faced. Blockchain technology has not only offered decentralized trustless digital currencies but also, via blockchains with scripting capabilities, decentralized execution environments for smart contracts' code. Executing a smart contract's code on a blockchain creates full transparency of the contract's state and ensures observability and verifiability, without a trusted intermediary. In his article from 1997, Szabo admitted that smart contracts may often need to involve a trusted third party to perform or adjudicate a smart contract (Szabo, 1997a). Blockchain-based smart contracts that only need data from within the blockchain's network (such as account balances of digital assets) are able to get rid of the need for having a third party perform or adjudicate a smart contract's execution. A smart contract that requires data from outside the blockchain's network still requires a trusted party to feed that data to the smart contract. Internet of Things and the ability for arbitrary physical objects to send and retrieve data from the Internet have brought Szabo's vision of embedded smart contract closer to reality in this aspect as well.

In the blockchain context, the term smart contract often takes a more specific meaning compared to Szabo's definitions. When associated with blockchain technology, smart contracts can be defined as self-executing scripts that reside on the blockchain (Christidis and Devetsikiotis, 2016). The scripts are executed in a decentralized and replicated manner by the nodes of the blockchain's peer-to-peer network. In the blockchain context, a script representing a smart contract does not necessarily have anything to do with the legal concept of a contract.

2.4 Decentralized applications

Decentralized applications can be defined as applications that are hosted by a set of distributed nodes and administrative domains. Decentralization stems from the fact that there are more than one administrative domains and that no administrative domain has control over another one's behavior. According to this definition, applications such as Bitcoin, email and Tor can be considered as decentralized applications. (Coughlin et al., 2017)

As is the case with the term smart contract, the blockchain community seems to have adopted a more blockchain-specific meaning for decentralized applications. In the blockchain context decentralized applications, abbreviated as dapps, are expected to employ blockchain to perform tasks that have traditionally been executed on a centralized server. (Raval, 2016)

Blockchain technology has enabled new types of decentralized applications, that use a centralized server like component, that is run in a decentralized manner on a blockchain. This blockchain-based back-end component offers an execution environment for a dapp's logic, as well as a globally accessible state that serves the purpose of the dapp's storage space. (Buterin, 2014)

2.5 Ethereum

Ethereum is a blockchain-based protocol for building decentralized applications. It introduces a Turing-complete programming language for writing smart contracts and allows deploying the smart contracts to the blockchain. Ethereum smart contracts have a state and are aware of data in the blockchain. (Buterin et al., 2017)

Ethereum smart contracts can serve as a back-end for decentralized applications. The benefits of using an Ethereum smart contract instead of a new blockchain include faster and easier development, bootstrapped security, and being able to communicate with other decentralized applications deployed in the Ethereum blockchain (Buterin et al., 2017).

2.5.1 Control of accounts

In Ethereum, accounts and value stored in them can either be controlled and owned by a private key or a contract code. Accounts controlled by a private key are called externally owned accounts, and comparable to asset accounts in any blockchain-based digital ledger. The private key holder can send messages from an externally owned account by creating and signing a transaction. Externally owned accounts are typically used to store and transact ether, the native currency of the Ethereum blockchain, and ERC20[4] tokens, or send messages to smart contracts. (Buterin et al., 2017)

Contract accounts are accounts controlled by contract code instead of a private key. When a contract account receives a message from another account, its code runs, allowing it to read the received message, send further messages, read and write its own internal storage or create other contracts, according to the rules set in the account's contract code. (Buterin et al., 2017)

Contract accounts and the idea that arbitrary program code may own and control digital assets, and may be executed in a decentralized manner on a blockchain, is what sets Ethereum apart from more traditional permissionless blockchains, such as Bitcoin.

[4]ERC20 is a standard interface for Ethereum smart contracts that implement a custom token.

2.5.2 Smart contracts

A smart contract in the context of Ethereum means a piece of code implementing arbitrary rules that are in direct control of digital assets (Buterin et al., 2017). All contract accounts in Ethereum have a smart contract, called the contract code, that controls how that account manages its funds and responds to messages (Buterin et al., 2017). A smart contract can be contrasted with an object in a programming language, having a state and a set of methods to read or modify that state.

When a smart contract's method is called, execution happens on each full node of the network in a replicated fashion.

2.5.3 Transactions

A transaction in Ethereum means data that is signed by an external account, representing either a message or deployment of a new smart contract. An Ethereum transaction always includes three fields essential to any cryptocurrency transaction: the recipient of the message, the signature of the sender and the amount of funds being transferred from the sender to the recipient. In addition to these, there are three Ethereum specific fields in a transaction: the `STARTGAS` field, the `GASPRICE` field and an optional data field. (Buterin et al., 2017)

2.5.3.1 Fee system

The `STARTGAS` and `GASPRICE` fields are needed for Ethereum's transaction fee system that is needed to prevent denial-of-service attacks in the network. A transaction's fee is decided based on the amount of resources it consumes. Ethereum uses the concept of "gas" to calculate the amount of computational, bandwidth and storage resources a transaction consumes. The transaction cost of an Ethereum transaction is simply the amount of gas consumed multiplied by gas price (that is set in the `GASPRICE` field). (Buterin et al., 2017)

There are many ways in which a transaction may consume gas. To name a few, every transaction has a base fee of 21 000 gas, every byte of optional data in a transaction has a gas cost, and every executed operation of a smart contract has a gas cost with a size depending on the operation. Some operations, such as the `SELFDESTRUCT` operation used to delete a smart contract from the blockchain, are able to do gas refunds and may effectively have a negative gas consumption. (Wood, 2014)

The `STARTGAS` value indicates the maximum amount of gas that the transaction is allowed to use (Buterin et al., 2017). Specifying this is an important way for a transaction sender to protect themselves against unexpectedly high gas consumption and hence a high transaction fee. When sending a transaction to the network, the sender is not able to know in which block the transaction will be mined to the blockchain and what the state of the blockchain will be. Gas consumption of a transaction often depends on the blockchain's state. Therefore it may be impossible to know how much gas a transaction will eventually consume. By setting a `STARTGAS` value, a transaction sender is able to set an upper bound for gas consumption and transaction fee.

The described fee system, instead of a simpler model, such as the one in Bitcoin, is needed due to the Turing-complete programming language that Ethereum smart contracts are implemented in. It may be impossible to say whether execution of a program written in a Turing-complete language will eventually finish. The fee system must be able to charge on a per computational step basis in order to avoid execution of infinite loops in smart contracts that would spend an indefinite amount of an executing Ethereum node's resources. The Ethereum fee model also encourages efficient smart contract code in general.

2.5.3.2 Optional data field

The optional data field of an Ethereum transaction can be used to add arbitrary data to the transaction (Wood, 2014). When interacting with a smart contract, the field is used to specify the function that the transaction sender wishes to call, potentially along with its arguments. In the case of creating a new contract, the data field contains initialization code of the new contract (Wood, 2014).

2.5.3.3 Messages

A message in the Ethereum context means data and value (ethers) that are transferred from an account to another. A message can be transferred in two ways. Firstly, transactions that do not deploy a new smart contract always include a message. Secondly, smart contracts may send messages to other smart contracts while executing. This is their way of making function calls to other smart contracts. (Wood, 2014)

2.5.4 Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) is the virtual machine where contract code of Ethereum smart contracts is executed. Running contract code on the EVM ensures that the execution environment is always identical and hardware independent. The way Ethereum consensus model works means that every operation executed in the EVM is executed on every full Ethereum node, as nodes validate new blocks. Though Ethereum smart contracts are typically written in a high-level language, such as Solidity or Serpent, the high-level language always needs to be compiled to a low-level bytecode language called Ethereum Virtual Machine code before execution.

The state of a running EVM can be defined by the tuple `(block_state, transaction, message, code, memory, stack, pc, gas)`. The tuple element `code` is the contract code: a sequence of bytes where every byte represents an operation. Code execution works iteratively so that the instruction at the `pc`$^{\text{th}}$ (program counter) byte of the contract code is executed. The program counter is initialized to zero. At each iteration, the program counter is increased by one. Code execution stops only when the end of the code is reached, an error occurs, or `STOP` or `RETURN` instruction is found. (Buterin et al., 2017)

EVM code has access to three types of storage for data: storage, memory and stack. Storage is where smart contracts store their state variables. It is an unlimited size data storage and the only storage that is persistent between function calls. The downside of using storage is that it is by far the most expensive place to store data. The storage of all smart contracts resides in `block_state`. Memory is a much cheaper data storage but it is erased between function calls. Memory is unlimited in size, but it costs gas to expand it. Stack is a limited sized, non-persistent data storage and the cheapest of the three EVM data storages to use. (Buterin et al., 2017)

The `gas` tuple element defines the amount of gas that the code execution has left to spend. All instructions, when executed, reduce an amount of gas. If `gas` value falls below zero, an error is thrown. (Buterin et al., 2017)

Chapter 3

A blockchain energy market

ETLA report number 71 published the source code of an Ethereum smart contract that functions as a core component of a blockchain-based energy market, hereafter referred to as the ETLA energy market (Hukkinen et al., 2017). The smart contract is implemented in the Solidity programming language. For the smart contract to be useful for energy trading, a system composed of the following four components, including the smart contract component itself, needs to be built and integrated:

- A smart contract facilitating the marketplace
- A small-scale electrical network for delivering electricity, hereafter referred to as a nanogrid
- Smart meters serving as access points to the nanogrid
- A reputation system for assessing the trustworthiness of smart meters

The purpose of this chapter is to explain how the ETLA energy market smart contract is designed to work and what kind of an environment it requires to be usable for energy trading. The smart contract software and the workflow of interacting with the contract will be explained in detail.

This chapter does not offer a full specification of the non-software parts of the system, i.e. the nanogrid and the smart meters acting as access points to the nanogrid. Also, this chapter does not give an in-detail specification of the reputation system for assessing the trustworthiness of smart meters. This chapter does, however, attempt to describe and discuss characteristics required from the non-software parts and the reputation system on a high-level, in order to create an understanding of the smart contract and its requirements.

The blockchain energy market is intended to be used by communities that share a nanogrid. The application allows nanogrid participants that have excess electrical energy, to sell that energy to other users of the nanogrid. The application provides a decentralized marketplace for transacting electrical energy. Since the marketplace is implemented as an Ethereum smart contract, it is able to have no central authority that could, for instance, censor or cancel offers, steal users' deposits, or do front running. In addition to providing a platform on which to trade energy and make contracts of transmitting electricity, the blockchain application also inherently offers a payment platform, using Ethereum's native cryptocurrency, ether.

The energy market facilitated by the ETLA smart contract carries out the following tasks:

- Receives offers to sell energy and allows buyers to accept them
- Manages agreements to sell and purchase energy
- Facilitates an escrow service, and payment handling in ether cryptocurrency
- Balances generation and consumption in the nanogrid

3.1 The process of transacting energy

The process of purchasing energy at the blockchain energy market is depicted in figure 3.1.

First, a user with excess energy expresses willingness to sell the energy by writing an offer to sell to the blockchain. Any other user may then decide to buy the offered energy. If a buyer submits a transaction by which they accept the offer, a sales and purchase agreement is made. The terms of the agreement are set by the seller in the initial offer creating transaction. A buyer may only agree to the terms, but not influence them in any way.

The sale and purchase agreement is a contract that obligates the seller to sell and the buyer to buy energy. The contract accepting transaction made by the buyer must include a prepayment of the energy. The prepayment is at this stage not yet transacted to the seller, but temporarily to the smart contract facilitating the trade. The purpose of the prepayment is that the buyer shows their commitment to the trade. It would be unwise for the buyer to back off after having already paid.

The sale and purchase agreement specifies two timestamps: one for the start of the energy transfer and one for the end of it. When the start time arrives, the seller's and the buyer's smart meters allow the buyer and the seller

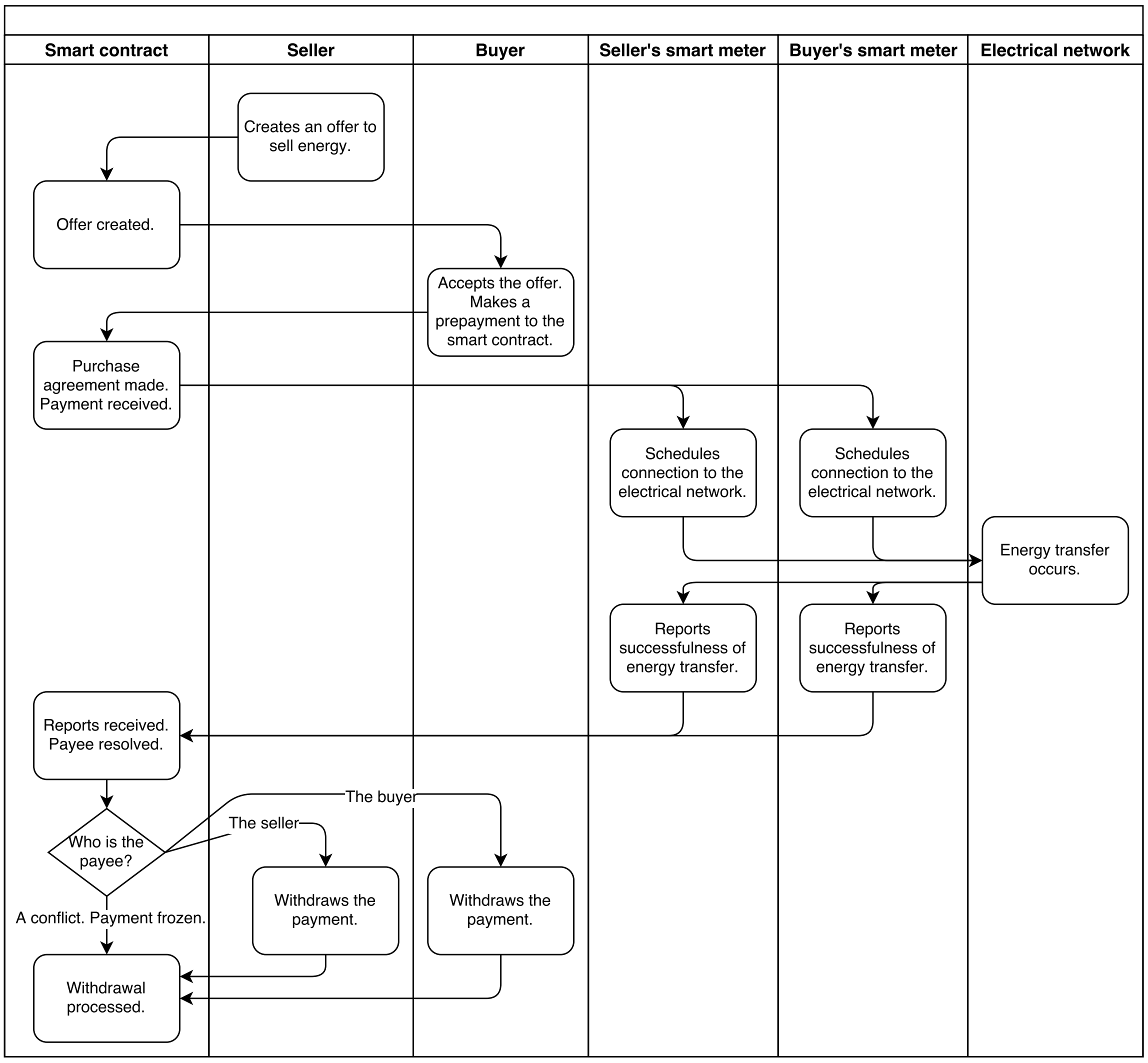
Smart contract
Seller
Buyer
Seller's smart meter
Buyer's smart meter
Electrical network
Creates an offer to sell energy.
Offer created.
Accepts the offer. Makes a prepayment to the smart contract.
Purchase agreement made. Payment received.
Schedules connection to the electrical network.
Schedules connection to the electrical network.
Energy transfer occurs.
Reports successfulness of energy transfer.
Reports successfulness of energy transfer.
Reports received. Payee resolved.
Who is the payee?
The buyer
The seller
A conflict. Payment frozen.
Withdraws the payment.
Withdraws the payment.
Withdrawal processed.

Figure 3.1: A flowchart of the process of trading energy on the marketplace provided by the smart contract. Each actor's actions are grouped to a column of their own.

to access the nanogrid and transfer electric energy. The smart meters act as gatekeepers of the nanogrid, preventing unauthorized energy consumption. The smart meters also record electric current and voltage during transmission, and using that data, they are able to tell whether the energy transfer was successful.

After the period of time allocated for energy transfer is over, the two smart meters autonomously create a blockchain transaction, reporting on the successfulness of the energy transfer on their owner's side. Based on the reports, the smart contract decides who is allowed to withdraw the prepayment made earlier by the buyer. The desirable outcome is that both reports implicate a successful trade, and the seller may withdraw the payment.

Sellers are capable of populating the energy market smart contract with multiple offers and buyers are allowed to accept any offer that suits them best. All offers and agreements are processed fully transparently. As a result, a fair market price should be discoverable by users of the system.

3.2 Nanogrid characteristics

The ETLA energy market application assumes that its users have access to a nanogrid where every access point to the nanogrid is interconnected. Figure 3.2 depicts such topology. The nanogrid needs to have a commonly agreed upon voltage and current type (alternating or direct current).

All participants of the nanogrid are connected to the nanogrid via a smart meter. Smart meters function as gatekeepers, only allowing users to draw or supply power when they have an ongoing scheduled energy transfer. Connecting to the nanogrid without a valid smart meter should be disallowed.

The example nanogrid topology of figure 3.2 shows a battery attached to each access point to the nanogrid. Though any means of energy storage is not required from users, an ample use of batteries is expected from users willing to make successful trades reliably.

For suppliers, failing to provide electricity at the rate defined in a contract results in a loss of payment and reputation. Many sources of distributedly generated energy are highly unpredictable, so suppliers may want to use a battery as a buffer, to be sure they can provide as much energy as they have promised in a contract.

Consumers, on the other hand, have an economic incentive to draw power at the maximum power they are allowed to. Unless consumers are very certain that they can consume all of the electricity delivered to them during the timeframe of an appointed energy transfer, a battery may be a rational investment for them as well.

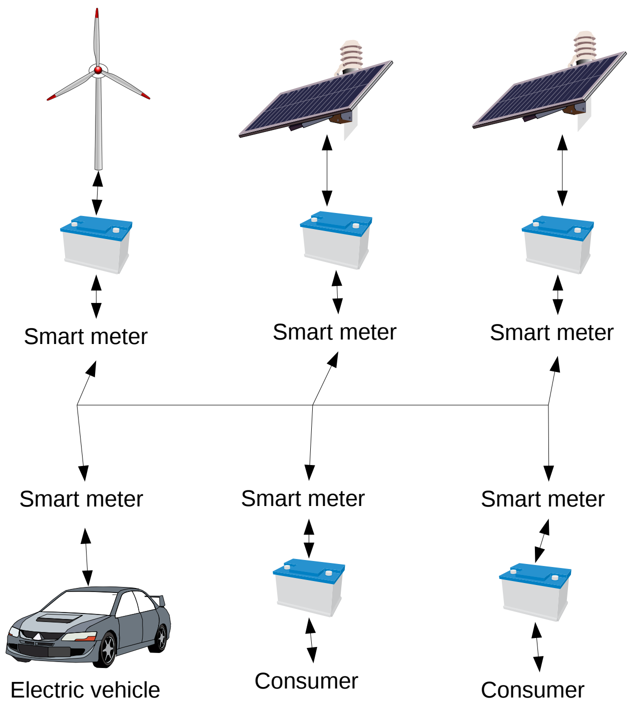

Figure 3.2: Example nanogrid topology

Each individual electricity transmission in the nanogrid must happen at an amperage and power predetermined in the sale and purchase agreement. More precisely, throughout the timeframe allocated for electricity transmission, the supplier of energy is obliged to supply electricity at least at the predetermined power and the consumer is allowed to draw at that same power at most. This convention ensures that when the system is working as expected, supply in the nanogrid equals to or exceeds consumption at all times.

A less strict approach, such as allowing the supplier to occasionally supply at a lower power than what is agreed upon in the contract, and then compensate by supplying at a higher power at other times, would result in conflicts in an interconnected nanogrid, where concurrent electricity transfers are allowed.

The ETLA energy market smart contract has no notion of transmission costs or power loss in the electrical network. Therefore it is only suited for a compact network where transmission costs of electricity are small enough to render them irrelevant when considering who to trade energy with.

3.3 Sale and purchase agreements

The trade of energy in the ETLA energy market is based on sale and purchase agreements that are binding contracts between a buyer and a seller. In an end-user application, the making of these contracts would likely be abstracted away from the user and delegated to the user's software. The contracts are enforced by the energy market smart contract and the users' independently acting smart meters. This section describes the flow of these contracts, as well as the data structure representing these contracts in the energy market smart contract.

When creating an offer to sell energy, a seller promises to provide electrical energy to the nanogrid with the terms defined in their offer in case a buyer were to accept the offer. An offer can be seen as a sale and purchase agreement that is lacking the identity and signature of a buyer.

A buyer accepting the offer needs to make a prepayment to the Ethereum smart contract, that is acting as an escrow agent. By making the prepayment, the buyer agrees to the sale and purchase agreement and gets permission to draw power from the nanogrid as soon as the energy transfer is ongoing.

The buyer is not able to violate the agreed-upon terms due to having fulfilled their obligation already when committing to the agreement by transferring the payment to the smart contract. The seller, however, may fail to provide enough energy, thus breaching the contract. A reputation system, that is able to penalize economically, is needed to prevent producers from doing so.

Table 3.1 shows all terms that are included in a sale and purchase agreement when a seller initially creates an offer to sell energy. The contract includes the four essential characteristics of any power delivery contract: delivery period, delivery location, size and price. Delivery location is expressed by the attribute *seller's smart meter*, which as its value, contains the Ethereum address of the smart meter through which the seller will be delivering power to the nanogrid. The *id* field contains a 256 bit unique identifier for the contract.

As the table shows, the amount of energy to be transferred is included in the contract. The power at which electrical energy will be delivered can be derived from this value and the amount of time allocated for the energy transfer.

Table 3.2 shows the two fields added to a sale and purchase agreement when a buyer accepts a seller's offer and an agreement is made. The buyer announces their Ethereum address and the smart meter's Ethereum address through which the buyer will draw power from the nanogrid.

Table 3.1: The attributes of a sales and purchase agreement not yet accepted by a buyer, along with an example value for each attribute.

Attribute	Example value
ID	9824138
Seller	0xde0B295669a9FD93d5F28D9Ec85E40f4cb697BAe
Seller's smart meter	0xbB1fef4e9B9db1aEAbBD474438a09F5825Aea110
Start time	2017-08-22T19:25:33+00:00
End time	2017-08-22T19:27:45+00:00
Amount of electricity	500 000 joules
Price	0.65 ether

Table 3.2: The attributes added to a sale and purchase agreement, when a buyer accepts it.

Attribute	Example value
Buyer	0xC2c1Eded5Cdf42d944C872295202CeE84B5fd0C0
Buyer's smart meter	0x7727E5113D1d161373623e5f49FD568B4F543a9E

Finally, when the scheduled electricity transmission is over, the seller's and the buyer's smart meters report whether the transmission was successful. Table 3.2 depicts these final fields added to the sale and purchase agreement's data structure residing in the energy market smart contract.

Table 3.3: The attributes added to a sale and purchase agreement when the participants' smart meter's report whether the participants have respected the contract.

Attribute	Example value
Seller's smart meter's report	OK
Buyer's smart meter's report	Not OK

3.4 Smart meters

Ethereum smart contracts can only fully rely on the correctness of the blockchain's internal data that is validated by miners, for instance, the distribution of the ether cryptocurrency. Applications that interact with the physical world often need access to data from the physical world. In the case of the blockchain energy market, the smart contract needs to know about the successfulness of energy transfers. This data can be fed into the blockchain in a transaction, but it can only be trusted as much as the party that creates the data feeding transaction.

Trusted smart meters are used as an interface between the blockchain energy market application and the physical world. A smart meter is required for every user to nanogrid connection. A smart meter serves three purposes:

- Acts as a trusted data feed that reports about the successfulness of energy transfers.
- Controls when its owner is connected to the nanogrid.
- Acts as a ”smart fuse”, limiting the current at which its owner can draw power from the nanogrid.

By default a smart meter keeps users disconnected from the nanogrid. A smart meter listens to events in the energy market smart contract. When it detects that a sale and purchase agreement has been made where the smart meter itself is declared as an endpoint for electricity transmission, it schedules itself to connect to the nanogrid for the duration of the transmission.

Besides connecting to and disconnecting users from the nanogrid, a smart meter acts as a smart fuse. If the current flowing through a smart meter is higher than what is agreed-upon in the contract, the smart meter disconnects its user from the nanogrid. This prevents users from consuming more energy than the producer they have a contract with is expected to produce.

During a scheduled transmission, smart meters record electric current and voltage. Based on these readings they are able to report to the blockchain, whether a transfer of electricity fulfilled the terms set for it in a contract.

A smart meter needs to either run its own Ethereum node or connect to an external trusted Ethereum node to be able to carry out its tasks. It needs to be able to submit transactions to be able to act as an oracle. It also needs to be able to read blockchain events to know the terms of sale and purchase agreements where it is involved. For creating transactions, a smart meter must have its own Ethereum address and have access to the addresses private key. The owner of a smart meter is expected to top up their smart meter’s Ethereum account with ether to maintain its ability to pay transaction fees.

Users of the energy market application should not have full control of their smart meter, the software installed on it, or the private key of the smart meter’s Ethereum address. Users should also not be able to bypass the smart meter and draw electricity directly from the nanogrid. Having these powers would allow users to steal energy and manipulate results of smart meter reports.

Smart meters should be issued by a trusted party, presumably a company that manufactures or installs the smart meters. Having a trusted party issue and sell the smart meters at least partly eliminates the concern of Sybil attacks. When users have to pay for a valid smart meter, any fraud they attempt to make should be worth at least the price of negative reputation inflicted on their smart meter to be profitable.

3.5 Reputation system

A reputation system for smart meters is required to disincentivize misbehavior of smart meters and users, leading to unsuccessful trades of energy. Even though the smart meters are expected to be trusted devices, it is possible for them to malfunction or be tampered by users, thus sending false reports to the blockchain. Also, dishonest or careless sellers of energy may fail to provide the amount of electricity to the nanogrid, that they have promised in a contract.

The blockchain energy market application does not implement an on-chain reputation system, nor does it oblige users to use an off-chain one. The smart contract lets market participants make trades with anyone, regardless of how successful their past trades have been. It is, however, expected that any rational buyer will consider the trade history of their counterparty before committing to a trade. That is, the buyers will employ a reputation system that is based on on-chain smart meter report data.

A full-fledged reputation system will not be proposed in this thesis. Table 3.4, however, depicts general guidelines for how we suggest reputation to be distributed in different outcomes of an energy trade. In a successful trade, the reputation of both participants increases. In case the producer's smart meter reports not having supplied enough energy, then regardless of what the buyer's smart meter reports, it can be concluded that the seller breached the contract and deserves a reputation decrease. The only case of dispute is when the seller's smart meter reports that it has distributed the agreed-upon amount of energy, but the buyer's smart meter states that it wasn't able to draw enough power from the nanogrid. In case of a dispute, it is impossible to trace which party is being honest, or if the cause of the issue is a third party or a malfunction in the nanogrid. Therefore both of the transacting participants' reputation needs to be decreased.

Seller's of energy are expected to transmit slightly more electricity to the nanogrid than what they have agreed in sale and purchase agreements. This is to avoid disputes caused by transmission losses or other inefficiencies in the electrical network.

Table 3.4: Suggested smart meter reputation changes.

Seller's smart meter's report on power transfer successfulness	Buyer's smart meter's report on power transfer successfulness	Reputation changes
OK	OK	Positive for both
OK	Not OK	Negative for both
Not OK	OK or not OK	Negative for the seller

3.6 Smart contract

The energy market involves two smart contracts written in the Solidity programming language: `ElectricityMarket` and `SmartMeters`. The `SmartMeters` contract is very simple. It contains a mapping that maps smart meter addresses to their owners. An issuer specified in the contract is able to issue new smart meters and transfer existing ones to new owners.

The rest of this section will discuss the `ElectricityMarket` contract that defines the logic of the energy market. The `ElectricityMarket` contract is able to look up owners of smart meters from `SmartMeters` contract.

3.6.1 Public functions

There are five public, state changing functions in the energy market smart contract.

3.6.1.1 makeOffer

The `makeOffer` function is used to signal willingness to sell energy. As parameters, the caller needs to specify ID of the offer, the price they charge for the energy, the amount of energy being offered, the start and end times of the electricity transmission, and the Ethereum address of a smart meter that the function caller owns, through which energy will be delivered to the nanogrid.

3.6.1.2 acceptOffer

After an offer has been made, a buyer may accept it by calling `acceptOffer`. As parameters, the buyer needs to specify the ID of the offer, and an address of a smart meter that the function caller owns. This is the smart meter that will act as the buyer's access point to the nanogrid at the time of electricity transmission. The transaction calling `acceptOffer` must also transfer the amount of ether specified in the offer to the smart contract as a collateral.

The function must be called in a block that is mined at least 60 seconds before the beginning of the electricity transmission. Having this period of time in between approval of the contract and the beginning of electricity transmission is important for two reasons. Firstly, there must be enough time for the transaction to propagate to the seller. Secondly, as a result, a few blocks are likely to be mined on top of the block where `acceptOffer` was called, increasing the certainty of the block where the function was called to be part of the longest chain.

3.6.1.3 buyerReport and sellerReport

The `buyerReport` and `sellerReport` functions take an offer ID as their first parameter and a boolean value expressing successfulness of the energy transfer as the second parameter. These functions can only be run by the buyer's or the seller's smart meter specified in the sale and purchase agreement. The smart meters have 30 minutes to submit their reports to the blockchain after the transmission of electricity has ended. If a report is not submitted within this time, the smart contract assumes a report of failed energy transfer.

3.6.1.4 withdraw

The `withdraw` function takes an offer ID as a parameter and is used to withdraw the deposit of the specified offer. The function can be successfully executed when the offer is in `ReadyForWithdrawal` state or is allowed to be moved to that state, meaning that both smart meter reports have been made or the deadline to make them has expired.

The function sets the trade to `Resolved` state and transfers the deposit made to it earlier by the buyer. The receiver of the deposit is decided based on reports received from the buyer's and the seller's smart meters, according to logic expressed in table 3.5. If both smart meters send positive reports, the deposit goes to the seller and the trade can be considered a success. In case the seller's smart meter sends a negative report and admits that the seller has failed to deliver enough energy, the buyer gets their deposit back. A clear dispute can only take place if the seller's smart meter reports positively and the buyer's sends a negative report. In this case, there is no way of telling which side is being honest, so the smart contract freezes the deposit to discourage unsuccessful trades.

Table 3.5: The table shows the rightful payee according to the smart contract's logic with all possible smart meter report combinations.

Seller's smart meter's report on power transfer successfulness	Buyer's smart meter's report on power transfer successfulness	Payee
OK	OK	The seller
OK	Not OK	The deposit stays frozen in the smart contract
Not OK	OK or not OK	The buyer

3.7 Scalability

A successful trade of energy on the ETLA energy market smart contract requires five Ethereum transactions, consuming a total of over 400 000 gas. As of 13th October 2017, the gas limit per block in the public Ethereum chain is 6 712 392 and the average block time circa 31 seconds. Were the public Ethereum chain to exclusively process transactions that call functions of the ETLA energy market smart contract, the blockchain would be able to handle about one trade per two seconds. A throughput like this is easily enough for a single community's nanogrid, but it is not enough for widespread adoption of the application, and processing trades of multiple nanogrids along with the transactions of all other smart contracts of the public Ethereum chain.

Some means of increasing scalability of the system are clearly needed. The application needs either a significant reduction in gas consumption, or it needs to be executed in an environment other than the public Ethereum blockchain.

Chapter 4

Applying a design science research methodology

This thesis employs the design science research methodology (DSRM) presented by Peffers et al. (2007) in order to improve and further develop the ETLA energy market smart contract. For their methodology, Peffers et al. (2007) have formed a process model of the following six activities:

1. Problem identification and motivation
2. Define the objectives for a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

The sequential order of the activities is only nominal. Researchers may enter the process at virtually any of the activities, and progress non-linearly, possibly iterating back to previous steps as new discoveries are made. (Peffers et al., 2007)

This thesis uses the first activity of the DSRM process, problem identification and motivation, as an entry point to the DSRM process. The main focus of the work is, however, on design and development.

In their DSRM process, Peffers et al. (2007) include separate activities for demonstration and evaluation of the developed solution. Peffers et al. (2007) do point out, however, that often in design science research, only one of these activities is carried out. This work takes the liberty of omitting the demonstration activity and only performing a more formal evaluation activity.

The sections of this chapter report the application of the DSRM to the ETLA energy market smart contract. The sections are structured and ordered according to the activities of the DSRM. Despite this thesis reporting the activities in a linear fashion, iteration did occur in the DSRM process.

4.1 Problem identification and motivation

The utility of any blockchain application, including the ETLA blockchain energy market, is limited by its operating costs. In any public blockchain, some sort of transaction fees are needed to create transaction priority, prevent denial-of-service attacks, and create an incentive for running network nodes that are building new blocks and consensus on which blocks are accepted in the chain.

To complete a successful trade in the ETLA energy market, both the seller and the buyer need to make transactions. According to our measurements, the seller needs to spend roughly 290 000 gas on three transactions (calling the functions `makeOffer`, `sellerReport` and `withdraw`) and the buyer roughly 110 000 gas (calling the functions `acceptOffer` and `buyerReport`). If the ETLA energy market were deployed on the public Ethereum blockchain, the cost of one trade today would be around $0.35 for the seller and $0.13 for the buyer [1].

The cost of performing a trade sets a lower bound for how small amounts of energy can be transmitted and thereby limits the possible use cases of the market. For instance, it can never be profitable to sell $0.30 worth of energy, if the seller needs to pay $0.35 in transaction fees. Implementing an energy market as a blockchain application brings a number of advantages compared to a centralized service but these advantages are easily outweighed by high operating costs in case the blockchain application is not competitive in that regard.

The public Ethereum blockchain would not be a viable deployment environment for the energy market application at the estimated operating costs.

The problem identified in the ETLA energy market smart contract is its high operating costs, that result from the contract's high gas consumption.

[1]The transaction costs were calculated using a gas price of 4 Gwei per gas and ether price of $300. The gas price of 4 Gwei is the median gas price of blocks 4512310 to 4513809 and $300 is the market price of ether on November 8, 2017.

4.2 Objectives

In section 4.2.1, we propose a set of guidelines we believe to be useful for decreasing gas consumption of any Ethereum smart contract. In section 4.2.2, we identify inefficiencies in the ETLA energy market smart contract and set the remedying of these inefficiencies as our objective.

4.2.1 Methods for reducing gas consumption of an Ethereum smart contract

In Ethereum, transaction cost is simply gas price multiplied by the amount of gas consumed by the transaction (Wood, 2014). To reduce transaction costs, one or both of these two values should be lowered.

Gas price should generally not be controlled by Ethereum smart contracts, but instead selected by the user at the time of creating a transaction. This allows users to maintain the ability to balance between cheap transactions and getting their transaction included in the blockchain quickly.

The amount of gas consumed on the other hand is a variable that can and should be optimized by smart contract developers. There are various ways in which a smart contracts gas consumption may be reduced. Ethereum's transaction fee system is built with the idea, that any use of computational, bandwidth or storage resources costs gas. Conversely, savings in any of the said resources will lessen gas consumption. For nearly every means that a smart contract has to charge gas, there is a way for developers to reduce the amount being charged.

We have formulated a list of guidelines that we believe to be efficient ways of optimizing gas consumption of an Ethereum smart contract. The list focuses on practices that reduce the use of the most expensive EVM operations.

- Avoid a design pattern where many new smart contracts need to be deployed, for instance, on a per-user basis. At a cost of 32 000 gas, contract creation is the most expensive EVM operation.

- Keep the amount of transactions needed to interact with the smart contract low, in order to diminish the impact of the transaction base fee of 21 000 gas. Design an interface with fewer functions that do more actions, rather than more functions that do fewer actions.

- Optimize the smart contract's use of storage space. Whenever possible, use memory instead of persistent storage. Storing a word in persistent storage costs 20 000 gas, whereas storing a word in memory only costs 3 gas plus a memory expansion fee, whenever more memory is required. The memory expansion fee scales quadratically as more memory is needed, so memory should be used densely. When the use of persistent storage is necessary, consider if the stored data could be replaced with its cryptographic hash on-chain, and the data itself could be stored off-chain.

- Delete contracts and data stored in persistent storage that are not needed, in order to gain gas refunds.
- Make use of off-chain transactions, using the blockchain only as an arbiter in case disputes happen.

The optimizations performed in this thesis will focus on migrating from on-chain execution to off-chain execution and combining actions executed in multiple transactions to a single transaction.

4.2.2 Identifying inefficiencies

When trying to find inefficiencies in the smart contract's gas consumption, we approached each of the contract's five public functions individually. We first considered if there is a viable way of executing the function off-chain. If not, we researched if another method of the ones listed in section 4.2.1 would be applicable.

We identified that the `offer` function can be removed from the smart contract entirely. Instead of announcing sales offers in the blockchain, offers can be cryptographically signed by their creator and sent to potential buyers off-chain. If a buyer later decides to accept the offer, the buyer must then include the sales offer along with the seller's digital signature as a parameter in their `acceptOffer` function call.

A major implication of the proposed change is that sales offers not accepted by a buyer are no longer stored in the blockchain. There should be no requirement for storing these offers on-chain because there is no need for having a consensus on whether a sales offer has been created or what was its time of creation. It is sufficient that a buyer can prove the authenticity of the offer via its digital signature when accepting it on-chain, and that the offer has not yet expired when it is accepted. Making this proposed change should not essentially change the smart contract's functioning.

The functions `acceptOffer`, `buyerReport` and `sellerReport` do not seem as straightforward to execute off-chain. An on-chain `acceptOffer` call is needed make sure that only one buyer can accept a given offer and to prove the network that the offer was accepted in time. The smart meter report functions have a strict deadline before which they must be submitted, and an on-chain function call is a simple way to prove that the deadline has been met.

The `withdraw` function needs to be executed on-chain in order for the funds to be transferred on-chain. We identified, however, that the current design, in which a separate call needs to made for each transfer of energy, is likely not optimal. If users were allowed to withdraw funds from multiple energy transfer contracts using a single `withdraw` call, fewer transactions would be created and less gas would be spent on transaction base fees. We expect this change to reduce overall gas consumption of the smart contract.

To summarize, we expect to be able to lower the amount of gas consumed by the smart contract in the following two ways:

1. Moving the `offer` function off-chain and reworking `acceptOffer` so that it takes an off-chain offer as a parameter and verifies its validity.

2. Reworking the `withdraw` function so that funds from multiple energy transfer contracts can be withdrawn using a single function call.

The objective for a solution to the identified problem is to implement these changes in the ETLA energy market smart contract.

4.3 Design and development

4.3.1 Off-chain offers

The idea behind off-chain sales offers is that instead of calling the smart contract's `makeOffer` function, the sellers form their offers, cryptographically sign them, and send them to potential buyers in an off-chain communication channel. A peer-to-peer network built for the needs of the blockchain energy market application would likely be an applicable communication channel. The communication channel should be accessible to anyone willing to trade in the marketplace.

The structure of an off-chain offer is portrayed in figure 4.1. This data sequence of 209 bytes must be signed by the seller of energy, in order to make the offer valid. The first 29 bytes are a prefix indicating that the signature is an Ethereum specific signature. The prefix must equal to the byte sequence `\x19Ethereum Signed Message:\n180`, where the number 180 represents the number of bytes in the actual message following the prefix. The `eth_sign` method of the Ethereum JSON RPC API forces a prefix of the named format.

Using JSON RPC calls is the standard way for off-chain components of a decentralized application to communicate to an Ethereum node. Asking a node to cryptographically sign a message is done by calling the `eth_sign` method of the Ethereum JSON RPC API.

Table 4.1: The sequential structure of an off-chain offer

Length	Content
29 bytes	The Ethereum Signed Message prefix ("\x19Ethereum Signed Message:\n180")
32 bytes	Offer ID
32 bytes	Price
32 bytes	Amount of electricity
32 bytes	Start time
32 bytes	End time
20 bytes	Address of the seller's smart meter

In the original smart contract code published in the ETLA report, a buyer, when accepting a sales offer of energy, would call the `acceptOffer` function of the smart contract providing the ID of the offer, and the address of their own smart meter as arguments. Terms of the sales offer would already have been stored in the smart contract as a result of an earlier `makeOffer` call made by the seller.

The new implementation no longer has a `makeOffer` function, due to its functionality having been moved off-chain. The buyer still calls the `acceptOffer`, but the function now takes full terms of the sales offer as arguments in addition to the seller's signature, proving the authenticity of the offer. The new `acceptOffer` implementation does the validity checks for the offer that were previously done in the `makeOffer` function. Using the sales offer and its signature provided as arguments, the `acceptOffer` function recovers the signer of the sales offer. The function then verifies that the signer actually owns the smart meter reported in the sales offer as the seller's smart meter.

The expected reduction in gas consumption from implementing off-chain offers is approximately 21 000 gas per a successful trade. This is due to not having to execute the `makeOffer` function anymore, and not needing to pay its transaction base fee. In addition to this saving, off-chain offers have the effect that offers that are never accepted by a buyer also never create a blockchain transaction, rendering them entirely free. Being able to create free offers enables use cases that are otherwise not feasible. For instance, consider a seller who knows that he is capable of selling and transmitting a certain amount of energy in one hour. The seller may create an initial offer with a price higher than what he expects to be the market price. To minimize the risk of not finding a buyer, the seller may then repeatedly send new offers to the network with a slightly lower price, until finally a buyer agrees with the price. The transaction cost of such behavior is exactly the same as if the seller's initial offer was accepted.

4.3.2 Withdrawing from multiple trades

The objective of the second improvement to the smart contract is to remove the need for active market participants to create a separate blockchain transaction for each resolved trade they want to withdraw funds from. The `withdraw` function will be modified so that it can withdraw from multiple trades in one function call.

We believe that implementing the proposed change to the `withdraw` function will create gas savings due to the need for fewer transactions and fewer transaction base fees. We estimate the saving per successful trade to be $21000 - \frac{21000}{n}$ gas, where 21 000 is the Ethereum base transaction cost, and `n` is the number of trades from which a user is able to withdraw using a single `withdraw` call. Therefore, using the redesigned `withdraw` function for withdrawing funds from a single is not expected to result in any savings. However, the more trades a user is able to withdraw from using a single function call, the closer the savings approach to 21 000 gas per trade.

Two implementations for withdrawing ether from multiple resolved trades using a single transaction were created. The first implementation offers a simple interface with no function parameters. In the first implementation, the smart contract keeps track of each user's open energy trades, and allows a parameterless function call to withdraw funds from all trades from which the user is entitled to do so.

The second implementation keeps the smart contract code simpler but does not offer an interface as clean as the first one. In the second implementation, the `withdraw` function takes a list of trades, from which the user wishes to withdraw, as a parameter. The second implementation requires users (or the application they are using to interact with the smart contract) to keep track of their ongoing trades and their IDs.

The second implementation was created only after having evaluated the first implementation. Due to the first implementation's high gas consumption observed in the evaluation phase of the DSRM, we iterated back to the design and development activity and developed the second implementation.

4.3.2.1 First implementation

A mapping was created in the smart contract, that links Ethereum addresses (representing market participants) to a list of energy trades, where the Ethereum address may be eligible for receiving the deposit. This mapping is named `contractsByPotentialWithdrawer`. When a user creates an offer or accepts it, the offer ID is added to the list found in `contractsByPotentialWithdrawer` mapping using the user's Ethereum address as a key.

The `withdraw` function was modified so that it no longer has any parameters. Instead of having to specify the ID of a trade from which to withdraw, the function withdraws from all trades where the function caller is entitled to collect funds. The function iterates through the list `contractsByPotentialWithdrawer[callerAddress]` where `callerAddress` is the Ethereum address of the function caller. If a trade is found in the list where the caller is entitled to collect funds, it will do so and change the state of the trade to resolved. Also, trades in which the funds have been frozen by the smart contract will be set to resolved state. Finally, the function removes all trades that are in the resolved state from the `contractsByPotentialWithdrawer[callerAddress]` list to mitigate redundant work for the next time the function is called. Removing resolved trades should also accumulate gas refunds for the transaction, due to freeing space in the persistent storage of the contract.

4.3.2.2 Second implementation

The second implementation takes a slightly different approach to the issue of being able to withdraw funds from a single trade per transaction. Instead of having the `withdraw` function withdraw from every trade where it is allowed to do so, the function was modified so that users must specify the trades from which they want to withdraw.

The implementation is very similar to the one in the original smart contract. The `withdraw` function of the original implementation only received the ID of one trade as a parameter and attempted to withdraw ether from that trade. The new implementation takes a list of trade IDs as a parameter, iterates that list and attempts to withdraw from each trade in the list.

4.3.3 Developed artifacts

As result of the design and implementation work, four artifacts were produced. Each artifact is a variant of the ETLA energy market smart contract with some attempted improvements implemented.

Artifact 1
Implemented off-chain offers (i.e. the changes described in subsection 4.3.1).

Artifact 2
The `withdraw` function was modified so that it withdraws from all trades where the caller is permitted to do so (i.e. implemented the changes described in section 4.3.2.1).

Artifact 3
The `withdraw` function was modified so that it takes an array of trade IDs as argument and attempts to withdraw funds from all of the listed trades (i.e. implemented the changes described in section 4.3.2.2).

Artifact 4
A combination of the changes implemented in artifacts 1 and 3. Artifact 4 was an attempt to create a version of the ETLA energy market smart contract that has both off-chain offers and an improved `withdraw` function implemented. Appendix A contains the source code of Artifact 4.

4.4 Evaluation

The gas consumption of each developed artifact was evaluated by executing a benchmark use case against the artifacts and recording the artifacts' gas consumption in the use case. The same benchmark use case was also executed against the original ETLA energy market smart contract, and the measurements collected from that execution were used as reference values.

The benchmark use case, described in section 4.4.1, was crafted so that it represents a typical use case of the smart contract, where a number of trades of energy are completed successfully.

4.4.1 Benchmark use case

The following use case was formulated to compare gas consumption of the original energy market smart contract implementation to the artifacts developed:

1. As a seller, create `n` number of offers to sell energy.

2. As a buyer, accept all created offers.
3. As the seller's smart meter, report all trades to have been successful.
4. As the buyer's smart meter, report all trades to have been successful.
5. Withdraw all deposits to the seller's address.

The variable `n` in step one translates to the number of energy trades completed in the use case. The test case was executed with different values of `n` to see how different implementations perform when varying amounts of transactions are created.

The combined gas consumption of all transactions created in execution of the use case was the measured result.

Gas consumption of transactions was inquired from the TestRPC instance using the `eth_estimateGas` function of the Ethereum JSON RPC API before the sending of each transaction. The `eth_estimateGas` function makes a transaction and returns its gas consumption, but does not add the transaction to the blockchain. In a TestRPC configuration like the one used, the `eth_estimateGas` call is made to a blockchain of exactly the same state as its corresponding actual transaction, so the returned gas consumption estimate is equal to the true consumption.

4.4.2 Environment

The measurements were run on an Ubuntu 16.04.3 LTS machine. The Solidity smart contracts were compiled using the Solidity compiler solc version 0.4.18. TestRPC is an Ethereum client built for testing and development. It can be used to create a local Ethereum test network. A local instance of TestRPC version 6.0.1 was used to simulate an Ethereum blockchain. TestRPC was configured to create a separate new block for each transaction. A block gas limit of 90 000 000 was configured.

4.4.3 Results

Table 4.2 shows the measured gas consumption of the ETLA energy market smart contract when executing the benchmark use case described above using `n` values progressing first linearly from 1 to 8, and then exponentially using values 16, 32, 64 and 128. The collected measurements are used as reference values that the developed artifacts are compared to.

The same benchmark use case was executed against each of the four artifacts. Table 4.3 contains the measurements collected from Artifact 1, table

4.4 contains measurements from Artifact 2, table 4.5 from Artifact 3 and table 4.6 from Artifact 4. The third column in these four tables shows the percentage difference in gas consumption between the ETLA smart contract and the developed artifact. The fourth column displays the per trade difference in gas consumed compared to the reference.

Table 4.2: Reference measurements from the original ETLA energy market smart contract.

Amount of trades (`n`)	Gas consumed
1	400 318
2	787 210
3	1 175 676
4	1 565 716
5	1 957 330
6	2 350 518
7	2 745 280
8	3 141 616
16	6 368 968
32	13 125 496
64	27 848 152
128	62 128 792

4.5 Communication

The results collected from the first artifact show that the implementation of off-chain offers did reduce gas consumption of the smart contract in the selected use case. In measurements where a few trades were made, roughly a 5 percent decrease in gas consumption was achieved. When more trades were made, this percentage gradually decreased. That is, however, due to the smart contract becoming more populated with data and certain phases in its execution having to spend gas on iterating that data. The absolute gas consumption savings achieved from off-chain offers do not seem to be reliant on the number of trades made, ranging narrowly from 20 784 to 20 816 gas per trade depending on the number of trades made. This roughly equals to the base transaction fee of 21 000 gas, which was the expected saving from not having to call the `makeOffer` function.

Implementing the ability to withdraw funds from multiple trades using a single transaction also led to savings in gas consumption, however not at

Table 4.3: Measurements from Artifact 1 that has off-chain offers implemented.

Amount of trades (`n`)	Gas consumed	Difference to reference (%)	Per trade difference to reference (gas)
1	379 534	-5.19	-20784
2	745 578	-5.29	-20816
3	1 113 260	-5.31	-20805
4	1 482 516	-5.31	-20800
5	1 853 346	-5.31	-20797
6	2 225 750	-5.31	-20795
7	2 599 728	-5.30	-20793
8	2 975 280	-5.29	-20792
16	6 036 168	-5.23	-20800
32	12 459 832	-5.07	-20802
64	26 517 208	-4.78	-20796
128	59 466 520	-4.29	-20799

Table 4.4: Measurements from Artifact 2 that implements the first iteration of the renewed `withdraw` function.

Amount of trades (`n`)	Gas consumed	Difference to reference (%)	Per trade difference to reference (gas)
1	477 671	+19.32	+77353
2	897 102	+13.96	+54946
3	1 318 107	+12.11	+47477
4	1 740 686	+11.18	+43743
5	2 164 839	+10.60	+41502
6	2 590 566	+10.21	+40008
7	3 017 867	+9.93	+38941
8	3 446 742	+9.71	+38141
16	6 934 407	+8.88	+35340
32	14 211 625	+8.27	+33942
64	29 975 536	+7.64	+33240
128	66 338 698	+6.78	+32890

Table 4.5: Measurements from Artifact 3 that implements the second iteration of the renewed `withdraw` function.

Amount of trades (`n`)	Gas consumed	Difference to reference (%)	Per trade difference to reference (gas)
1	402 564	+0.56	+2246
2	762 007	-3.20	-12602
3	1 123 024	-4.48	-17551
4	1 485 615	-5.12	-20025
5	1 849 780	-5.49	-21510
6	2 215 519	-5.74	-22500
7	2 582 832	-5.92	-23207
8	2 951 719	-6.04	-23737
16	5 959 480	-6.43	-25593
32	12 276 826	-6.47	-26521
64	26 121 121	-6.20	-26985
128	58 645 051	-5.61	-27217

Table 4.6: Measurements from Artifact 4 that implements off-chain offers and the second iteration of the renewed `withdraw` function.

Amount of trades (`n`)	Gas consumed	Difference to reference (%)	Per trade difference to reference (gas)
1	381 780	-4.63	-18538
2	720 397	-8.49	-33407
3	1 060 652	-9.78	-38341
4	1 402 481	-10.43	-40809
5	1 745 884	-10.80	-42289
6	2 090 861	-11.05	-43276
7	2 437 412	-11.21	-43981
8	2 785 537	-11.33	-44510
16	5 627 010	-11.65	-46372
32	11 611 844	-11.53	-47302
64	24 791 563	-10.98	-47759
128	55 985 573	-9.89	-47994

first attempt. The first implementation, Artifact 2, used more gas than the reference implementation in every measurement. This is due to its need to be able to iterate over each user's contracts, where the user is potentially eligible to withdraw funds. To implement this, the smart contract makes extensive use of expensive persistent storage space. It seems clear that having to call `withdraw` function only once did bring gas savings, since the more trades are made, the closer Artifact 2 comes compared to its reference in gas consumption. The use of persistent storage was, however, a more significant factor than these savings.

At the cost of increased gas consumption, Artifact 2 was able to have a simpler interface for the `withdraw` function. It could be argued that the significance of this interface is not very high, since in an end user application, users would likely not be interacting with the smart contract directly. There could be a program layer above the smart contract, that executes off-chain and calls the smart contract functions on behalf of the user.

The second attempt at improving the `withdraw` function, in the form of Artifact 3, was much more successful. When `n` value was set to one, the artifact performed about half a percent worse than the reference implementation. With all other `n` values, the artifact consumed less gas than its reference. At best, it was able to reduce gas costs by over 6 percent. Surprisingly, with all `n` values other than one, Artifact 3 created larger savings than the estimated $21000 - \frac{21000}{n}$ gas per trade. With `n` values greater than or equal to 5, the savings of the artifact exceeded the base transaction fee of 21 000 gas, which we expected to be the maximum gas saving for the artifact. We believe that the extra savings at least partly originate from Artifact 3 only calling the Solidity `send` function once, while the reference implementation calls it `n` times. As a result, slightly less EVM code needs to be executed.

Artifact 4 was a combination of the changes that were found to be working in Artifacts 1 and Artifact 3. It has both off-chain offers and the improved `withdraw` function. A noteworthy remark about Artifact 4 was that its gas consumption savings were almost exactly equal to the sum of savings gained in Artifacts 1 and Artifact 3. There was virtually no overhead in combining the two improvements.

In the formulated use case, both of the implemented improvements were able to generate savings of approximately the same size, that is approximately the size of the Ethereum transaction base fee per trade. The changes made to the `withdraw` function created more savings than off-chain offers. It should be noted, however, that the selected benchmark use case does not demonstrate the savings achievable from off-chain in the best way. Off-chain offers have the effect, that offers that are never accepted by a buyer do not require a blockchain transaction and are thus completely free. Were the use case formulated in, for instance, such a way that sellers create a hundred offers and only one of them gets accepted by a buyer, the gas savings would have been substantially larger.

4.5.1 Feasibility

While we were able to reduce the gas consumption of the energy market smart contract in this work, the reduction was a little over ten percent at best. Assuming that users of the energy market are using batteries to assure their capability of making successful trades, a typical trade on the energy market could be estimated to be in the order of magnitude of the size of large car battery's capacity. A 12 volt 100 Ah battery could theoretically output 1.2 kWh of energy. Assuming a price of 0.1 $/kWh, the energy of a typical trade would be worth $0.12. In section 4.1 we made an estimate that a trade in the original smart contract, deployed in the public Ethereum blockchain, would cost approximately $0.48. Even with the gas savings achieved, it seems that transactions costs would exceed the value of energy transferred in a typical trade. We can conclude that the implemented optimizations are not adequate to make the application feasible on the public Ethereum chain costwise, at least not for energy trades as small as suggested.

4.5.1.1 Proposed solutions

We see potential in Plasma child blockchains proposed by Poon and Buterin (2017), as a viable platform for the energy market application. Plasma chains are essentially child chains of the Ethereum main chain, that have their own validators. Plasma chains are represented in the main chain by smart contracts, to which the state of the child chain is periodically submitted. Fraud proofs are used to penalize dishonest validators. Native coins and tokens of the main chain may be deposited to and withdrawn from a child chain. A Plasma chain should provide significantly smaller transaction fees than the main chain at the cost of decreased security. (Poon and Buterin, 2017)

A Plasma child chain could provide an execution environment similar to and connected to the Ethereum main chain, but with a lower demand for transactions, implying lower transaction costs.

For the time being, there are no finished Plasma implementations, but development efforts are ongoing[2].

[2]See for example the beta phase implementation by BANKEX: `https://blog.bankex.org/bankex-developed-the-worlds-first-private-blockchain-that-supports-plasma-protocol-adaa1459039d` (accessed 16

Another option more feasible than deploying the application on the public Ethereum chain, would be to create a separate Ethereum blockchain instance. Since the application requires trusting the smart meters and their issuers, it should be considered whether the blockchain can be made permissioned and the trusted parties the whitelisted nodes. In this scenario, the blockchain could be entirely transaction feeless. Even as a permissionless blockchain, transaction fees would likely be a fraction of what they are in the canonical Ethereum chain, due to a smaller demand for transactions.

An issue with having a separate blockchain isolated from the canonical Ethereum chain is the lack of support for the ether currency and the ERC20 tokens of the canonical chain. The isolated chain would need its own coin.

A large share of the application's gas consumption originates from the use of persistent storage. Significant gas savings, and thus lower operating costs in any execution environment, could be achieved by storing the hash of a sales and purchase agreement instead of its full details in the blockchain. The actual data could be hosted outside the blockchain, for instance using the IPFS protocol.

December 2017).

Chapter 5

Discussion and conclusions

5.1 ETLA energy market as a decentralized application

Blockchain applications that require access to such data from outside the blockchain, that is not generated by the identity feeding the data to the blockchain, are problematic. These types of applications need a trusted middleman, or a group of middlemen, to feed the data. The main benefit of blockchain applications is decentralization, so adding centralized components to a blockchain application raises the question, if using blockchain technology is beneficial at all.

The ETLA energy market application suffers from the described problem. In the application, smart meters make reports about electricity data, that they do not generate themselves, but measure from the electrical network. As a result, the application relies on the trustworthiness of the smart meters and the authorities that are allowed to issue these smart meters. Despite using a blockchain as a decentralized execution environment, the application is not fully decentralized.

Another issue is that the application is reliant on the assumption that the trusted smart meters are the only access points to the electrical network. The application has no means for enforcing this rule. A real-world application should somehow either make the electrical network inaccessible without using an authorized smart meter or resort to an authority, likely a centralized one, with the ability to penalize for breaking the rule.

Despite the aforementioned flaws, the blockchain does bring decentralized qualities to the application. The marketplace for energy is decentralized and uncontrollable by any authority, up until the point when trades need to be settled in the physical world. Transactions that create offers to sell, or accept those offers, are as difficult to censor, fake or manipulate as any blockchain transaction. It is only the settlement of trades, i.e. exchange of energy in the physical world and reporting the outcome of that exchange, where decentralization is lost.

5.2 Contributions

As a result of this work, the ETLA blockchain energy market has a technical documentation in the form of chapter 3. The documentation is valuable because it enables reviewing and evaluation of the application and its feasibility. The documentation also lays the groundwork for possible physical implementations and experiments with the concept and further development of it.

The design science part of this thesis yielded a number of results and learnings. First, we produced the guidelines for optimizing gas consumption of Ethereum smart contracts listed in section 4.2.1. The guidelines were successfully used to find and fix inefficiencies in the ETLA energy market smart contract. We believe them to be applicable in similar optimization tasks for other Ethereum smart contracts as well.

Four artifacts resulted from the design science part of this research. The first three artifacts produced were revisions of the ETLA energy market smart contract with changes that attempted to reduce gas consumption of the smart contract. The last artifact produced, referred to as Artifact 4, combined working improvements from the earlier artifacts. Artifact 4 fixed two inefficiencies and performed up to over 11% more cost-effectively than the original smart contract, in the use case presented in section 4.4.1. Artifact 4 also enables sellers of energy to make fee less off-chain offers.

The measured gas consumption reduction was not significant enough to enable novel use cases or deployment environments for the application. The added ability to create off-chain offers, however, while had little impact on gas consumption in the benchmark use case, enables new types of offer strategies for sellers. In a use case involving heavy use of sell offers that are never accepted, off-chain offers alone could reduce gas consumption much more than the measured 11%.

Artifact 2 and its failure to improve gas efficiency of the energy market smart contract concretely showed that an Ethereum smart contract should perform the absolute minimum set of tasks required from it. Decentralized applications should implement program layers on top of the smart contract to enable keeping the smart contract as simple and low resource-consuming as possible.

5.3 Limitations of this work

Two inefficiencies of the energy market smart contract were identified and fixed in this. It is likely that there are more fixable inefficiencies in the smart contract that were simply not identified in this thesis.

The gas consumption optimizations implemented in this work are evaluated using a single benchmark use case. It is clear that one use case cannot perfectly represent the typical use of the application. For instance, no such offers to sell are made in the use case, that do not get accepted by a buyer. This is an occurrence that would likely happen in real-world use, and in which Artifact 1 and Artifact 4 would have had a significant advantage over the reference implementation, due to them implementing off-chain offers.

The communication channel for propagating off-chain offers was not implemented in the scope of this thesis.

5.4 Future research

We recognize that it is debatable, whether the energy market application is a viable blockchain application, due to its need for centralized components. We encourage future research to more extensively evaluate the application's viability as a decentralized application and seek ways to purge its centralized components.

We propose future research to make further efforts in making the application economically feasible in a public and permissionless setup. We believe that significant progress is unlikely to be made by simply further optimizing gas consumption of the smart contract. Instead, we propose more radical approaches, such as experimenting with Plasma child chains or a separate standalone Ethereum blockchain instance.

We also encourage researchers and industry professionals to experiment in combining the energy market smart contract with physical devices and actual transactions of energy. Such experiments could confirm whether the system is functional, and point out its potential flaws and issues.

Bibliography

BitFury Group and Jeff Garzik. Public versus Private Blockchains. Part 1: Permissioned Blockchains, 2015. URL `http://bitfury.com/content/5-white-papers-research/public-vs-private-pt1-1.pdf`.

Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Ethereum*, (January):1–36, 2014. URL `http://buyxpr.com/build/pdfs/EthereumWhitePaper.pdf`.

Vitalik Buterin et al. Ethereum white paper — a next-generation smart contract and decentralized application platform. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2017. Accessed: 2017-12-11.

Torstein Bye and Petter Vegard Hansen. How do spot prices affect aggregate electricity demand? 2008.

David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

Usman W. Chohan. The decentralized autonomous organization and governance issues. 2017. URL `https://ssrn.com/abstract=3082055`.

K Christidis and M Devetsikiotis. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access*, 4:2292–2303, 2016. ISSN 2169-3536. doi: 10.1109/ACCESS.2016.2566339.

M. Coughlin, K. Kaoudis, and E. Keller. Augmenting cloud architectures to support decentralized applications. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 544–547, May 2017. doi: 10.23919/INM.2017.7987325.

Wei Dai. b-money, 1998. *URL http://www. weidai. com/bmoney. txt*, 1998.

Marco Dell'Erba. Initial coin offerings. a primer. 2017. URL `https://ssrn.com/abstract=3063536`.

John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

European environment agency. Eurostat — share of renewable energy in gross final energy consumption. `http://ec.europa.eu/eurostat/tgm/table.do?tab=table&plugin=1&language=en&pcode=t2020_31`, 2017. Accessed: 2017-11-02.

Paddy Finn and Colin Fitzpatrick. Demand side management of industrial electricity consumption: Promoting the use of renewable energy through real-time pricing. *Applied Energy*, 113(Supplement C): 11–21, 2014. ISSN 0306-2619. doi: https://doi.org/10.1016/j.apenergy.2013.07.003. URL `http://www.sciencedirect.com/science/article/pii/S0306261913005692`.

Taneli Hukkinen, Juri Mattila, Juuso Ilomäki, and Timo Seppälä. A Blockchain Application in Energy. ETLA Reports 71, The Research Institute of the Finnish Economy, 2017. URL `https://pub.etla.fi/ETLA-Raportit-Reports-71.pdf`.

Kristian Lauslahti, Juri Mattila, and Timo Seppälä. Smart contracts–how will blockchain technology affect contractual practices? ETLA Reports 68, The Research Institute of the Finnish Economy, 2017. URL `https://pub.etla.fi/ETLA-Raportit-Reports-68.pdf`.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

Joseph Poon and Vitalik Buterin. Plasma: Scalable Autonomous Smart Contracts. *White paper*, 2017.

Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *Technical Report (draft)*, 2015.

Siraj Raval. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology.* " O'Reilly Media, Inc.", 2016.

Melanie Swan. *Blockchain: Blueprint for a new economy.* " O'Reilly Media, Inc.", 2015.

Tim Swanson. Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. *Report, available online, Apr*, 2015.

Nick Szabo. Smart Contracts. 1994.

Nick Szabo. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9), 1997a. ISSN 13960466. doi: 10.5210/fm.v2i9.548. URL `http://ojphi.org/ojs/index.php/fm/article/view/548`.

Nick Szabo. The idea of smart contracts. 1997b.

Rafal Weron. *Modeling and forecasting electricity loads and prices: A statistical approach*, volume 403. John Wiley & Sons, 2007.

Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.

World Nuclear Association. Renewable energy and electricity. `http://www.world-nuclear.org/information-library/energy-and-the-environment/renewable-energy-and-electricity.aspx`, 2017. Accessed: 2018-01-03.

Aaron Wright and Primavera De Filippi. Decentralized blockchain technology and the rise of lex cryptographia. 2015.

Appendix A

Source code of Artifact 4

```
pragma solidity 0.4.18;

import "./SmartMeters.sol";


contract ElectricityMarket {

   enum ContractState { NotCreated, Created, Accepted,
      WaitingForBuyerReport,
      WaitingForSellerReport, ReadyForWithdrawal, Resolved,
         TimedOut }

   struct Contract {
      address seller;
      address sellerSmartMeter;
      address buyer;
      address buyerSmartMeter;
      uint price;
      uint electricityAmount;
      uint startTime;
      uint endTime;
      bool sellerReport; // true if everything went OK
      bool buyerReport; // true if everything went OK
      ContractState state;
   }

   modifier contractInState(uint id, ContractState state) {
      require(contracts[id].state == state);
      _;
```

```
}

modifier waitingForSellerReport(uint id) {
    if ((contracts[id].state != ContractState.Accepted) &&
        (contracts[id].state !=
        ContractState.WaitingForSellerReport)) {
        throw;
    }
    _;
}

modifier waitingForBuyerReport(uint id) {
    if ((contracts[id].state != ContractState.Accepted) &&
        (contracts[id].state !=
        ContractState.WaitingForBuyerReport)) {
        throw;
    }
    _;
}

modifier ownsSmartMeter(address owner, address smartMeter) {
    require(owner == smartMeters.owner(smartMeter));
    _;
}

modifier buyerSmartMeterOnly(uint id) {
    require(msg.sender == contracts[id].buyerSmartMeter);
    _;
}

modifier sellerSmartMeterOnly(uint id) {
    require(msg.sender == contracts[id].sellerSmartMeter);
    _;
}

modifier condition(bool c) {
    require(c);
    _;
}

modifier costs(uint price) {
    require(msg.value == price);
    _;
```

```
}

mapping (uint => Contract) contracts;
mapping (address => uint[]) contractsBySmartMeter;
SmartMeters smartMeters;


// Minimum time from block.timestamp to startTime. The time
   needs to be long
// enough, so that a few blocks are created in between, so that
   the smart
// meters can be sure that the transmission is approved by the
   blockchain
uint minTimeFromAcceptedToStart = 60; // 1 minute
// Maximum time from endTime to smart meters reporting about
   how the
// transmission went.
uint maxTimeFromEndToReportDeadline = 1800; // 30 minutes

event LogOffer(address seller, uint id, uint price, uint
   electricityAmount, uint startTime, uint endTime, address
   sellerSmartMeter);
event LogAcceptOffer(address seller, uint id, uint price, uint
   electricityAmount, uint startTime, uint endTime, address
   sellerSmartMeter, address buyer, address buyerSmartMeter);
event LogResolved(uint id, address seller, address buyer,
   address recipient);

function ElectricityMarket() {
    smartMeters = new SmartMeters();
}

function acceptOffer(bytes32 r, bytes32 s, uint8 v, uint256 id,
   uint256 price, uint256 electricityAmount, uint256
   startTime, uint256 endTime, address sellerSmartMeter,
   address buyerSmartMeter) payable
    condition(startTime < endTime)
    costs(price)
    ownsSmartMeter(msg.sender, buyerSmartMeter)
{
    contractNotCreatedCheck(id);
    noOverlappingContractsCheck(sellerSmartMeter, startTime,
       endTime);
```

```
        contractTimeoutCheck(startTime);

        address seller = getOfferSigner(r, s, v, id, price,
            electricityAmount, startTime, endTime, sellerSmartMeter);
        require(seller == smartMeters.owner(sellerSmartMeter));

        storeNewOffer(id, price, electricityAmount, startTime,
            endTime, sellerSmartMeter, seller, buyerSmartMeter,
            msg.sender);
        LogOffer(seller, id, price, electricityAmount, startTime,
            endTime, sellerSmartMeter);
        LogAcceptOffer(seller, id, price, electricityAmount,
            startTime, endTime, sellerSmartMeter, msg.sender,
            buyerSmartMeter);
    }

    function getOfferSigner(bytes32 r, bytes32 s, uint8 v, uint256
        id, uint256 price, uint256 electricityAmount, uint256
        startTime, uint256 endTime, address sellerSmartMeter)
        private constant returns (address) {
        // The prefix used by web3.eth.sign method. The number in
            the end is the
        // amount of bytes in the message.
        bytes memory prefix = "\x19Ethereum Signed Message:\n180";

        bytes32 hash = keccak256(prefix, id, price,
            electricityAmount, startTime, endTime, sellerSmartMeter);
        return ecrecover(hash, v, r, s);
    }

    function sellerReport(uint id, bool report)
        sellerSmartMeterOnly(id)
        waitingForSellerReport(id)
    {
        if (hasReportDeadlineExpired(id)) {
            contracts[id].state = ContractState.ReadyForWithdrawal;
            return;
        }
        contracts[id].sellerReport = report;
        contracts[id].state = (contracts[id].state ==
            ContractState.Accepted) ?
            ContractState.WaitingForBuyerReport :
            ContractState.ReadyForWithdrawal;
```

```
}

function buyerReport(uint id, bool report)
    buyerSmartMeterOnly(id)
    waitingForBuyerReport(id)
{
    if (hasReportDeadlineExpired(id)) {
        contracts[id].state = ContractState.ReadyForWithdrawal;
        return;
    }
    contracts[id].buyerReport = report;
    contracts[id].state = (contracts[id].state ==
        ContractState.Accepted) ?
        ContractState.WaitingForSellerReport :
        ContractState.ReadyForWithdrawal;
}

// Attempt to withdraw from all ids listed in parameter. If a
    contract is
// not ready for withdrawal, throws.
function withdraw(uint[] ids)
{
    uint withdrawSum = 0;

    for (uint i = 0; i < ids.length; i++) {
        uint id = ids[i];

        tryToMakeReadyForWithdrawal(id);
        if (contracts[id].state ==
            ContractState.ReadyForWithdrawal) {
            address recipient = getRecipient(id);

            if (recipient == msg.sender) {
                withdrawSum += contracts[id].price;
                contracts[id].state = ContractState.Resolved;
                LogResolved(id, contracts[id].seller,
                    contracts[id].buyer, msg.sender);
            }
            else if (recipient == address(this)) {
                contracts[id].state = ContractState.Resolved;
                LogResolved(id, contracts[id].seller,
                    contracts[id].buyer, address(this));
            }
        }
    }

    msg.sender.transfer(withdrawSum);
}

// Return address of the rightful deposit recipient of a
    contract in Resolved state
function getRecipient(uint id) private constant returns
    (address) {
    if (!contracts[id].sellerReport) {
        return contracts[id].buyer;
    }
    else if (contracts[id].buyerReport) {
        return contracts[id].seller;
    }
    else {
        return address(this);
    }
}

// Assume that startTime < endTime for both timestamp pairs
function doTimeslotsOverlap(uint startTime1, uint endTime1,
    uint startTime2, uint endTime2) private constant returns
    (bool) {
    if ((endTime1 < startTime2) || (endTime2 < startTime1)) {
        return false;
    }
    return true;
}

function hasReportDeadlineExpired(uint id) private constant
    returns (bool) {
    if ((contracts[id].endTime +
        maxTimeFromEndToReportDeadline) > now) {
        return false;
    }
    return true;
}

function storeNewOffer(uint id, uint price, uint
    electricityAmount, uint startTime, uint endTime, address
    sellerSmartMeter, address seller, address buyerSmartMeter,
    address buyer) private {
    contracts[id].seller = seller;
    contracts[id].price = price;
    contracts[id].electricityAmount = electricityAmount;
    contracts[id].startTime = startTime;
    contracts[id].endTime = endTime;
    contracts[id].sellerSmartMeter = sellerSmartMeter;
    contracts[id].state = ContractState.Accepted;
    contracts[id].buyer = buyer;
    contracts[id].buyerSmartMeter = buyerSmartMeter;

    contractsBySmartMeter[sellerSmartMeter].push(id);
}

function noOverlappingContractsCheck(address smartMeter, uint
    startTime, uint endTime) private constant {
    for (uint i = 0; i <
        contractsBySmartMeter[smartMeter].length; i++) {
        Contract c =
            contracts[contractsBySmartMeter[smartMeter][i]];
        if (doTimeslotsOverlap(startTime, endTime, c.startTime,
            c.endTime)) {
            throw;
        }
    }
}

function contractTimeoutCheck(uint startTime) private constant {
    require((now + minTimeFromAcceptedToStart) <= startTime);
}

function contractNotCreatedCheck(uint id) private constant {
    require(contracts[id].state == ContractState.NotCreated);
}

function tryToMakeReadyForWithdrawal(uint id) private {
    if ((contracts[id].state == ContractState.Accepted
        || contracts[id].state ==
            ContractState.WaitingForSellerReport
        || contracts[id].state ==
            ContractState.WaitingForBuyerReport)
        && hasReportDeadlineExpired(id))
    {
        contracts[id].state = ContractState.ReadyForWithdrawal;
    }
}

function getSeller(uint id) constant returns (address) {
    return contracts[id].seller;
}

function getBuyer(uint id) constant returns (address) {
    return contracts[id].buyer;
}

function getBuyerReport(uint id) constant returns (bool) {
    return contracts[id].buyerReport;
}

function getSellerReport(uint id) constant returns (bool) {
    return contracts[id].sellerReport;
}

function isCreated(uint id) constant returns (bool) {
    return contracts[id].state != ContractState.NotCreated;
}

function getState(uint id) constant returns (ContractState) {
    return contracts[id].state;
}

function getSmartMetersContract() constant returns (address) {
    return smartMeters;
}

}
```