University of Kaiserslautern

Department of Electrical Engineering and Information Technology

Microelectronic Systems Design Research Group

Bachelor Thesis

Design and Implementation of a Blockchain-Based Smart Outlet Concept

Entwurf und Implementierung eines Blockchain-basierten Smart Outlet Konzepts

---

Presented:	June 11, 2019
Author:	Daniel Gretzke (392488)
Research Group Chief:	Prof. Dr.-Ing. N. Wehn
Tutor:	M.Sc. Frederik Lauer

Statement

I declare that this thesis was written solely by myself and exclusively with help of the cited resources.

Kaiserslautern, June 11, 2019

Daniel Gretzke

Abstract

City residents rarely park in front of their own home. In the future, when more and more cars on the road will be electric, charging the car might become a problem. This bachelor thesis develops a peer to peer concept that allows every homeowner to sell electricity with the help of a smart socket to an electric car owner without the need for a middleman. Furthermore, a payment system was developed with the aid of a blockchain-based, decentralized computing platform, which can run on embedded systems like a smart socket or plug. This concept and payment system were implemented successfully and enable a secure exchange of electricity for monetary value on embedded platforms.

Abstract

Für Stadtbewohner ist es häufig schwierig, direkt vor dem eigenen Haus einen Parkplatz zu finden. In Zukunft, wenn immer mehr Elektroautos auf den Straßen fahren, kann das Laden des Autos zu einem Problem werden. Diese Bachelorarbeit entwickelt ein Peer-to-Peer Konzept, mit dem jeder Hausbesitzer über eine intelligente Steckdose Strom an Elektroautobesitzer ohne Mittelsmann verkaufen kann. Darüber hinaus wurde ein Zahlungssystem mit Hilfe einer auf der Blockchain-Technologie basierenden, dezentralen Rechenplattform entwickelt, welches auf eingebetteten Plattformen, wie einer intelligenten Steckdose und einem intelligentem Stecker, läuft. Dieses Konzept und das daraus hervorgehende Zahlungssystem wurden erfolgreich implementiert und erlauben den sicheren Austausch von Elektrizität gegen einen finanziellen Geldwert auf eingebetteten Plattformen.

Contents

1 Introduction

> Whereas most technologies tend to automate workers on the periphery doing menial tasks, blockchains automate away the center. Instead of putting the taxi driver out of a job, blockchain puts Uber out of a job and lets the taxi drivers work with the customer directly.
> — *Vitalik Buterin, co-founder of Ethereum*

A cryptocurrency based on a blockchain was first implemented in 2009 by Satoshi Nakamoto (pseudonym) and was called Bitcoin. Since then, it has steadily gained importance every year. Meanwhile, thousands of cryptocurrencies and tokens were built on this technology. The hype in the year 2017 called the attention of many companies to blockchain and even last year, when the value of cryptocurrencies fell as far as 95%, the interest in this field did not drop.

Compared to traditional payment methods like Visa, Banks and PayPal, cryptocurrencies are built decentralized, meaning that there is no central organization that controls transactions, the issuance of new money, et cetera. The validity of the blockchain *P2P* (peer to peer) network is secured through cryptographic protocols. This brings several benefits. Traditional payment methods usually go with high transactions costs, most commonly in the amount of a few percent. On the contrary, the cost of a single transaction on a blockchain averages out at just a few cents[1]. Some cryptocurrencies even work without any fees.

Because of this, they are suited for micro transactions really well. There are some disadvantages, though. The blockchain technology is still at an early stage and really immature. Compared to traditional electronic payments, it only manages to achieve very few *TPS* (transactions per second) and has long transaction times. E.g., Bitcoin manages 4-5 TPS[2] as opposed to Visa, which manages to process almost 4,000 TPS on average[3].

As stated in the quote above, the key strength of blockchain and cryptocurrencies is the decentralization aspect. For many, it will reshape various markets we know today, potentially revolutionize the financial industry and even disrupt monopolies in the future.

Another trend regarding the future are electric cars. It's expected that in a few years most cars on the road and almost all cars sold will be electric. Often these need to be charged overnight. Unfortunately, most city residents are familiar with the problem that they rarely park in front of their own house, let alone own a garage. It's foreseeable that recharging a car might bring difficulties.

This bachelor thesis is devoted to this problem. It examines whether a smart electrical

socket, which is placed outside the house by a homeowner, can be used to efficiently sell electricity and which payment method is suited best for this task. Based on an initial concept, a prototype is to be developed that implements the previously worked out features. It will serve as an example on how to implement *M2M* (machine to machine) payments on a microcontroller level.

2 Theory

The purpose of this chapter is to explain the most important terms about blockchain and the key underlying components of the technology. Most explanations will be kept superficial, as only a broad understanding of the term is required for this thesis.

Cryptocurrency

Traditional, so-called fiat currencies like the Euro or US Dollar have been around for hundreds of years in form of coins or paper money. They are usually issued and regulated by a government. With the rise of the internet, a new form of currencies, the so-called cryptocurrencies emerged. They are digital currencies that are usually decentralized, meaning they work without a central authority controlling and regulating the flow of monetary value. The rightfulness of transactions is ensured by mathematical functions like digital signatures and often a technology called the blockchain.

Hash

In cryptocurrencies, hashes are used to verify data. The cryptographic hash function is a mathematical function that usually fulfills the following requirements[4]:

1. *Determinism*: the same input always produces the same output
2. *Preimage resistance*: it is nearly impossible to get the input from the output
3. *Second-preimage resistance*: it is nearly impossible to find a second input that produces the same output as the first input

The output of this function is called hash. Bitcoin uses the SHA-256[5], Ethereum uses the Keccak-256[6] hashing algorithm which produces an output hash with a length of 32 bytes.

Public Key Cryptography

Public key cryptography[7] is the backbone of the blockchain technology and consists of a key pair – a private and a public key. The private key needs to be secret and the public key can be shared with anyone. A message can be signed using the private key resulting in a signature. This signature can then be verified using the public key to prove that the message was, in fact, signed by the corresponding private key. Bitcoin and Ethereum use the *ECDSA* (Elliptic Curve Digital Signature Algorithm) for signatures. They are used to prove that transactions were indeed sent by a specific account and not manipulated.

Wallet
Everything that is needed to send and/or receive cryptocurrencies is a private key[5], which is a wallet in its simplest form. More complex forms of a wallet encrypt a private key with a password or store it on a dedicated hardware device.

Address
An address is used to identify participants on the blockchain. Usually, it is derived from a public key belonging to the private key of an account, e.g., Ethereum uses the 20 least significant bytes of the Keccak-256 hash of the public key as the address[6].

Ledger
The ledger is the equivalent of a traditional record of all addresses and their balances. Every participant of the network has a copy of this ledger.

Node
A node is a participant on the blockchain. When a new node joins the network, it downloads the ledger currently accepted by the majority. A so-called full node has the entire history of the ledger – from the first block to the latest.

Transaction
A transaction updates the ledger, e.g., reducing the balance of an address by a specific amount and adding it to the balance of another account.

Blockchain
A list of transactions is bundled together into a block. Also included in this block is the hash of the previous block, which contains the hash of the previous block and so on. This creates a chain of blocks[5] which always point to their predecessor as demonstrated in Figure 2.1, thus the name blockchain. This is also the reason why a blockchain is considered immutable: modifying any information in a block would automatically change the hashes of every block after it and would be rejected by the network.

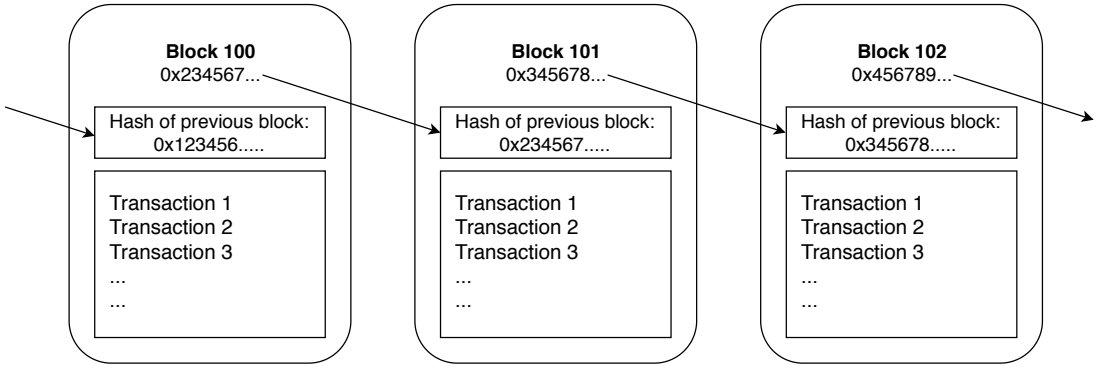

Figure 2.1: Visualization of a blockchain

Miner / Validator

A miner is a participant on the network whose task it is to secure the network through validating or mining new transactions and blocks.

Fees

For most cryptocurrencies, a fee has to be paid with every transaction. These go to the miner as a reward for including a transaction in a block. Additionally, they prevent an attacker from spamming the network with transactions. In contrast to traditional payment methods, which usually have a variable fee based on the total transaction value, cryptocurrencies have a fixed fee. Miners can freely choose which transactions they want to include in a block, therefore the more a user is willing to pay, the higher the chance, that the transaction will be included in the next block. Fees differ from cryptocurrency to cryptocurrency and can be as low as a fraction of a cent, but also went up to 55$ for a single transaction on average[8] for Bitcoin once, when the network couldn't scale to handle all transactions.

Proof of Work

There are different approaches to reach consensus over which block should be added next to the network, PoW (Proof of Work) is one of them and is used by most cryptocurrencies. A *nonce* (number used once) is included in the block which is adjusted by a miner. Every different nonce produces a different block hash. The goal is to find a nonce which produces a hash that is numerically lower than a certain threshold value (also called difficulty). A miner who finds this number can submit their block to the network. All miners race to find a nonce as fast as possible, so their block will be accepted by all other participants in the network first and considered valid. The fastest submission gets a so-called block reward, a reward minted by the protocol to the fastest miner, as well as all transaction fees from that block.

Block Time
The time it takes to add a new block to the network is called block time. On the Bitcoin network, the block time is 10 minutes[5]. Ethereum wants to archive a block time of 12 seconds[9], realistically it is averaging at about 15 seconds[10]. The difficulty is adjusted automatically to keep the block time roughly the same, no matter how much computing power is currently mining.

Confirmation Time
The confirmation time is the time it takes until a transaction is mined. The higher the transaction fee the faster the transaction is confirmed.

Ethereum Virtual Machine
The *EVM* (Ethereum Virtual Machine) is a Turing complete virtual machine that can execute computer code on the Ethereum blockchain. As a result, the Ethereum network acts like a decentralized computer with all the features of a blockchain: the storage is entirely public and every computation or code execution is recorded on the blockchain.

Smart Contract
Ethereum smart contracts are written in special programming languages, the most popular being called "Solidity", and are compiled into bytecode afterwards. This bytecode is then publicly stored on the blockchain and can be executed by everyone. The advantage of smart contracts is, that the code is public and immutable, thus can theoretically be used as a binding, programmable contract that can control monetary value on its own. As a drawback, every computational step and byte stored on the blockchain costs money in form of transaction fees.

DApp / Web3
DApp stands for decentralized application. Web3 is also called the decentralized web, where websites are powered by smart contracts and DApps.

Account
There are two types of accounts on the Ethereum blockchain: An *EOA* (externally owned account) and a contract account. Each of those accounts has a transaction count and a balance[6] associated with them. An EOA is controlled by a private key, whereas a contract account has code and storage associated with it.

Off-chain Transactions / Payment Channels
For every transaction on the blockchain, fees have to be paid. One way to save transaction fees is to have multiple transactions that are not recorded on the blockchain. They are called off-chain transactions and are settled on the blockchain, once the payment process is finished.

Ether - GWei - Wei
ETH (Ether) is the currency used on the Ethereum network. It can be divided into smaller fractions. The most important are Wei and GWei. Wei is the smallest unit: 1 Ether equals 10^{18} Wei[6]. GWei is primarily used for calculations of transaction fees: 1 Ether equals 10^9 GWei, 1 GWei equals 10^9 Wei. Ether is always used in Wei format in smart contracts.

Mainnet / Testnet
The main network of a blockchain is called the mainnet. Testnets are blockchain networks that behave similarly to the official mainnet and are used for testing purposes only. This allows developers to develop smart contracts without the risk of losing any monetary value. Additionally, improvements to the mainnet are often tested on a testnet prior to their official implementation.

3 Concept

This chapter describes the requirements for the overall concept and goals of this bachelor thesis, analyzes different payment methods to evaluate which meet said requirements and defines a concept and its hardware requirements for a possible implementation.

3.1 Definition of Prerequisites

The following prerequisites are required to develop a concept for the sale of electricity between two parties who do not know each other:

1. A potential customer Alice owns an electric car and wants to purchase electricity to charge it overnight.
2. A supplier Bob who owns a smart electrical socket wants to sell electricity.
3. Alice and Bob do not meet, Alice does not trust Bob to supply the electricity she paid for, Bob does not trust Alice to not wrongfully revert the payment after the electricity has been supplied. It has to be assumed that there are bad actors, who want to steal from the other party (in the form of monetary value or electricity).
4. For all calculations throughout this thesis the following values are assumed:
   a) Charging power: 3.7 kW
   b) Electricity price: 0.3 €/kWh
   c) Total transferred energy: 40 kWh
   d) Charging duration: 11 hours
   e) Electricity cost: 12 €
   f) Price for the customer: 18 € (profit margin of 50%)

The objective of the next section is to evaluate which payment method is suited best for the described use case. Thus, the goals of the M2M payment system have to be defined:

1. Value has to be transferred from Alice to Bob.
2. Electricity has to be supplied in return for a payment, whereby the risk or impact of not getting electricity in return has to be minimal.
3. Alice needs to be able to stop paying for electricity when it is not longer needed, e.g., when the battery is fully charged, without overpaying.
4. Bob should only start supplying electricity after a valid payment was received and thus the risk of losing electricity is minimized.

5. Value should be transferred from Alice to Bob at least once per minute to continuously pay for electricity to reduce risk, but transaction fees should stay at a minimum at the same time. For this thesis a payment interval of 15 seconds is assumed resulting in 0.007 € per payment.

6. All calculations relevant for the payment system should be able to be processed on an embedded platform with low computational power and memory.

3.2 Analysis of Different Payment Methods

In this subsection, the advantages and disadvantages of traditional payment methods, established electronic and online payments and cryptocurrencies are compared and a payment method for the implementation is chosen.

3.2.1 Traditional Currency

The simplest form of monetary value is cash, it could be imagined that money could be paid through a coin slot and electricity would be returned as a result. It's not suitable for this use case, as the risk of not receiving any electricity in return for the customer and the risk of theft for the supplier, e.g., someone violently stealing the cash, is too high.

Traditional electronic or online payments like VISA or PayPal pose a low risk of theft for the buyer because of customer protection methods, but the risk for the seller is non-negligible, e.g., in the form of credit card fraud or customer protection exploitation. Another disadvantage can be seen in the high fees, PayPal, for example, has a pricing of 0.10 € + 10% for micro-transactions[11] and wouldn't be economically feasible for the predefined goals. The key strength is a fast, near-instant transaction time.

3.2.2 Cryptocurrency

Cryptocurrency payments have some major advantages over the previously analyzed payment methods. Compared to electronic and online payments, many cryptocurrencies have lower fees and some even work without any fees at all. Payments can be broken down into fractions, as mentioned above, with payment intervals of a minute or less. Because all payments are immutable, the supplier has no risk and the customer risks just losing less than one cent for a first payment. Additionally the currency can be stored right on an embedded device and the machine itself is in charge of sending transactions. One of the biggest disadvantages of cryptocurrencies for M2M payments are long transaction times, but as the technology is still in its infancy, this might change in the future.

Over a thousand different cryptocurrencies exist[12] and every one of them has their own rules that can drastically differ from the next one. Each underlying blockchain technology has their benefits and drawbacks. In the next paragraphs, some of these digital currencies will be evaluated whether or not they are suitable to reach the set goals.

Bitcoin

Because Bitcoin is the earliest cryptocurrency, it suffers from problems other cryptocurrencies could improve upon. As demonstrated in Figure 3.1, scalability is one of these issues. The block time is 10 minutes[5] and during peak times the average confirmation time can take more than two hours. Thus bitcoin should not be considered for this use case.

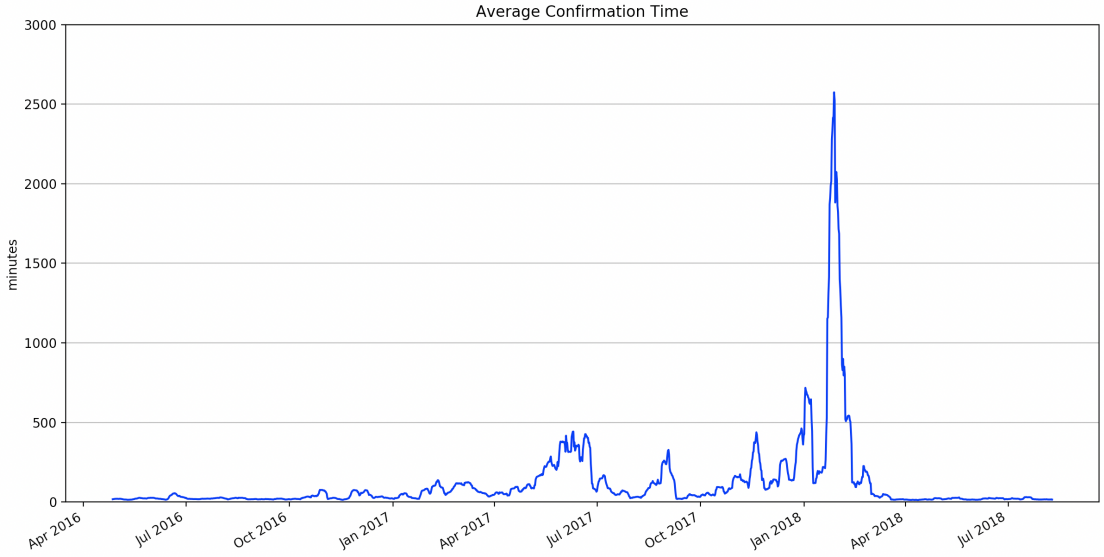

Figure 3.1: Average confirmation time of a transaction on the Bitcoin blockchain[13]

IOTA

As previously mentioned, there are some cryptocurrencies that work without any fees that are worth to be considered. One of these currencies is IOTA. It was designed especially with M2M payments in mind and the underlying technology, called Tangle, differs from traditional blockchains. The way it works is that before someone's transaction can be validated, they need to validate other transactions first[14]. In theory, this makes the network scale especially well — the more transactions are broadcasted, the shorter the transaction times get. In practice, transaction times are at 2.6 minutes at the time of writing[15] and therefore not fast enough for this use case.

Nano

Another feeless cryptocurrency is called Nano, formerly known as RaiBlocks. It is built upon a technology called block-lettuce. Each account on the network is an independent blockchain which can update itself asynchronously from the rest of the accounts. This makes Nano not only have zero transaction fees but also allows it to have transaction times of less than a second. It can also handle way more TPS than Bitcoin and Ethereum, namely around 100 TPS over a longer period of time with peaks up to 300 TPS[16].

Unfortunately, at the time of development of the prototype, the Nano network faced some issues which resulted in transaction times in up to 20 seconds, as seen in Figure 3.2. After the V18 update these problems were solved and the transaction times returned back to less than a second[17].

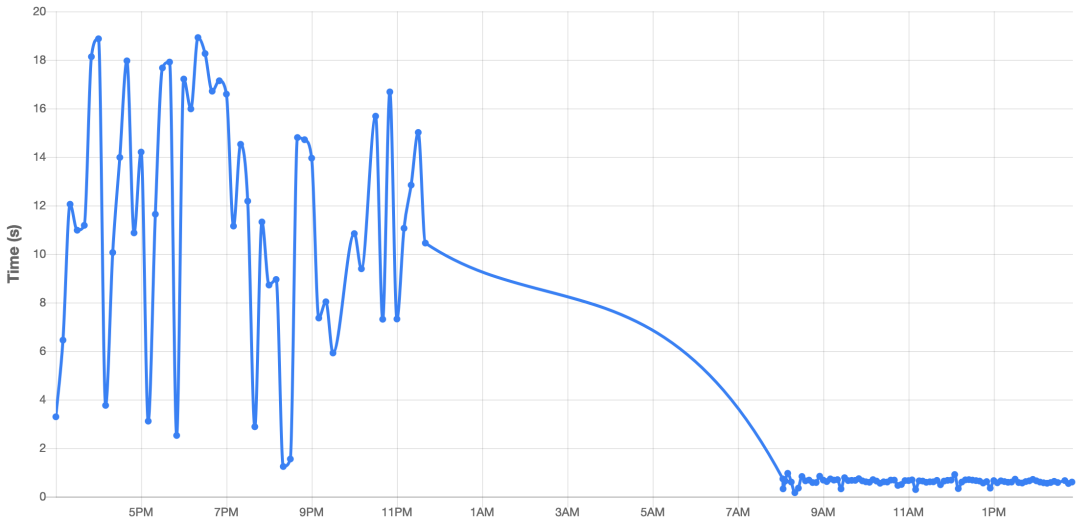

Figure 3.2: Improvements of transaction times after the update to V18 on Feb. 22nd, 2019[17]

Although the currency was considered at first, the main reason Nano was not chosen for the implementation is that it was not built with M2M transactions in mind. To prevent spamming, i.e., attacking the network through congesting it with transactions, PoW needs to be generated in order to send or receive transactions. According to the Nano white paper[18] an Intel Core i7-4790K processor with 4.00 GHz can handle up to 0.33 TPS. A microcontroller would not be able to generate the PoW required for the transactions in a reasonable amount of time. Instead, a dedicated server would be required for each supplier and customer, which would store the private keys and handle the transmission and reception of transactions. The micro controllers would merely communicate with said servers and therefore the currency did not meet the defined requirements.

Ethereum

Ethereum has a block time of 15 seconds. At the time of writing the fee to be included in the next block is 0.01 €[1]. In this case, the transaction would be fast enough to meet the goal of 4 transactions per minute, but the transaction fees would exceed the payment, raising the total price as much as 143%.

One way to cut transaction fees is the use of off-chain transactions. As previously mentioned, these work very similarly to on-chain transactions: information like a beneficiary and value can be signed by a sender, so it can be proven where the transaction originates from. They differ from each other, as off-chain transactions are not recorded on the blockchain. They can be sent from one participant to another directly, therefore there is no transaction fee or transaction time involved. A number of these off-chain transactions can be bundled in a payment channel. In the end, these transactions have to be settled on the blockchain, actually modifying the ledger and updating the balances of all participants. Thus, the 2,460 transactions required during a charging period of 11 hours can be bundled into a few transactions on the blockchain. Sending on- and off-chain transaction requires computations like encoding, hashing or signing of data which can be achieved on an embedded platform. Therefore this payment method meets all the requirements listed above.

Ethereum was chosen for the implementation of this prototype, as it implements a virtual machine with the ability to execute code, therefore a payment channel can be built using smart contracts. The following steps briefly explain how the payment concept works:

1. The customer places a deposit (max. purchase value) into the smart contract and initializes the payment channel.
2. The customer signs an off-chain transaction and sends it to the supplier.
3. Once the supplier receives the transaction, the delivery of electricity begins. Steps 2 & 3 are repeated until the customer or the supplier want to discontinue the exchange.
4. As soon as any party wants to close the payment channel, the supplier submits the off-chain transactions to the smart contract. Each party is now able to withdraw their share from the smart contract.

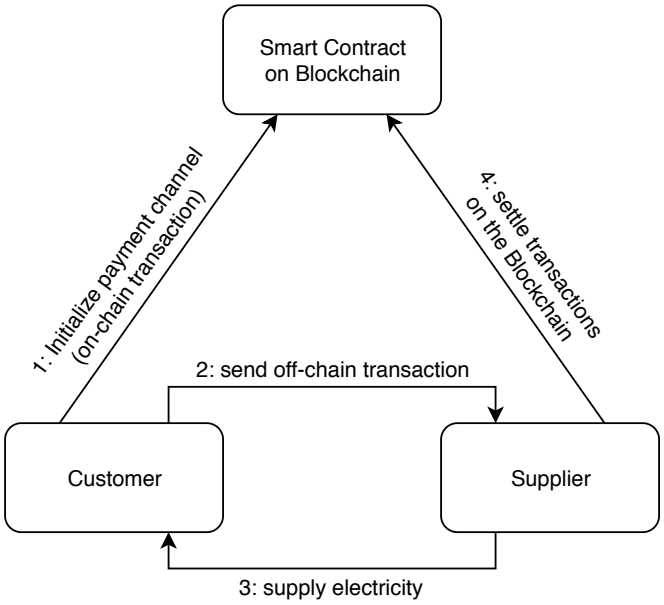

Figure 3.3: Sketch of a payment channel

With payment channels, the risk of the supplier is minimal, as electricity must only be provided when a valid payment was received. The customer only risks losing one transaction for the initialization of the payment channel and one off-chain transaction if no electricity is provided in return.

In total, only 4 on-chain transactions have to be made. One for the payment channel initialization, one for the settlement, and one by each party to withdraw the balances, totaling all transaction fees at a couple cents.

3.3 Component Requirements

As demonstrated in Figure 3.4 the concept is based on four components. A supplier, a customer, a node in form of a server and a smart contract on the Ethereum blockchain.

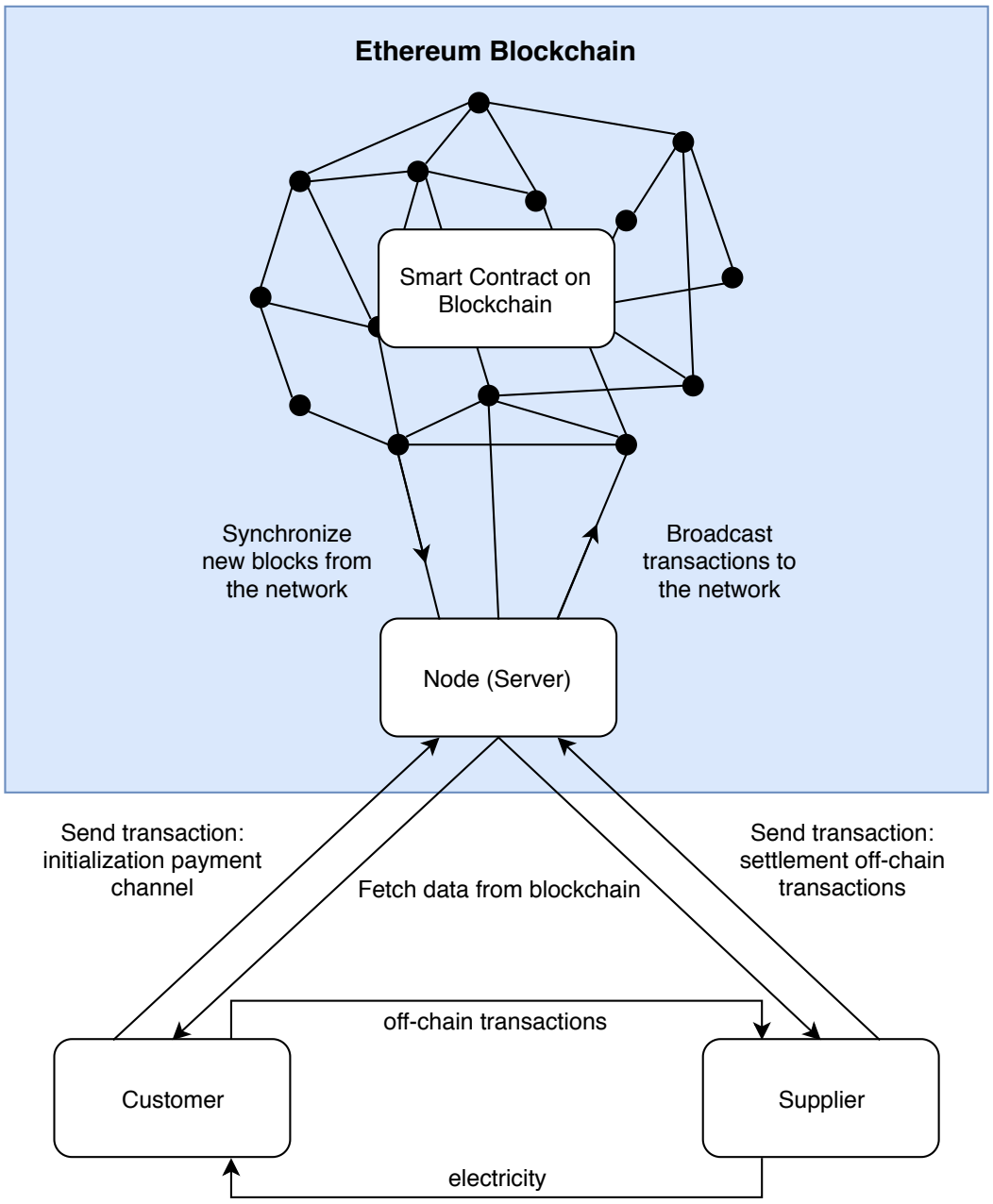

Figure 3.4: Sketch of the concept

The supplier should have a microcontroller which is integrated directly into an electrical socket. This way it can be powered directly by the supplier's circuit and can be protected against external damages, e.g., caused by weather or human influences. A relay should be placed between the circuit and the socket, which can be switched on and off

by the microcontroller to start and stop the supply of electricity to the customer. For the communication with the Ethereum network an internet connection is required. It can be either established by connecting to a cellular network using a SIM card or to a WiFi network, preferably of the supplier itself.

The customer should have a microcontroller which is integrated into an electrical plug. If it is integrated into a short extension cord, any electrical device can be connected to it and be charged. The microcontroller should be able to measure whether electricity is flowing through the cord using a current sensor, e.g., a hall effect-based sensor. The plug is required to communicate with the Ethereum network as well, therefore a SIM card can be integrated to connect to a cellular network or the plug can use the supplier's WiFi connection for the communication. The microcontroller can be powered directly by the supplier's socket itself, as the electricity output can be limited to sufficiently power the customer's microcontroller but not enough to charge any device.

Both parties are required to communicate with each other, as payment information needs to be exchanged and off-chain transactions need to be sent and received. The microcontrollers can send information to each other over the circuit itself using *PLC* (power-line communication) or, if the customer and supplier are using the same WiFi network, it can be used to handle the communication between both parties.

The design and functionality of the off-chain transactions is defined by the implementation of the smart contract, which should have the following features. It should act in the interest of the customer and the supplier, securing both parties from fraud. A payment channel should be initialized by the customer by depositing an amount of Ether into the smart contract. The smart contract should act as a trustee, managing the funds until the payment channel is closed again. The customer can then start signing off-chain transactions with its private key and sending them to the supplier. When the payment channel is ready to be closed, the supplier should submit the off-chain transactions to the smart contract, as it is in its interest to submit all transactions. The smart contract is required to verify whether the off-chain transactions were indeed signed by the customer. If the transactions are valid, the smart contract can then pay out the total transaction volume to the supplier and refund the remainder to the customer. To ensure that both parties are protected, the payment channel should have an expiration date. In case the supplier fails to submit a valid signature in a certain amount of time, the customer should be able to withdraw the entire deposit from the smart contract.

To initialize and close the payment channel, the customer and the supplier need to be able to communicate with the Ethereum blockchain, which is achieved with a node. It is the only component required to run on a server, as the entire blockchain history has to be downloaded which takes up several gigabytes of data. When relevant pay-

ment information, e.g., the price per second stored in the smart contract, needs to be fetched from the blockchain, the server sends the information stored in its synchronized local copy. When data is sent to the blockchain in form of a transaction, the entire process of generating the transaction computes on the microcontroller. The transaction is then sent to the node which forwards it to the other blockchain participants. Therefore, the node acts as a gateway between the Ethereum network and the microcontrollers.

3.4 Risks & Concerns

Although choosing payment channels implemented on a blockchain over traditional payment methods brings some advantages in low transaction fees and every party being in charge of their own money, there are some drawbacks that are worth noting:

The reason no party has to trust another when using smart contracts is immutability. Once the code has been deployed to the blockchain, it cannot be changed after the fact. This poses a risk, as bugs found in the smart contract code cannot be fixed either. This already resulted in losses of multiple hundreds of millions of dollars on the Ethereum blockchain alone[19].
As soon as funds are stored inside a smart contract or on a physical device there is a financial incentive for hackers to try to steal the money. As the private keys of the customer and supplier are stored on the embedded devices themselves, every part of the microcontroller has to be carefully secured, so under no circumstances the private key can be extracted to steal funds. Therefore, PLC should be chosen over a WiFi connection for the communication between the customer and the supplier.
Because the concept is designed as a P2P, middleman free system, not only 3rd parties have an incentive to steal funds. If the customer uses the supplier's WiFi network to communicate with the blockchain, a man in the middle attack should be prevented by using a HTTPS connection secured over TLS.
Transactions are the most secure part of the concept, as they are always digitally signed. Therefore, even if they are intercepted by a malicious actor, they can only be used for their intended purpose.

Another concern is that the concept would not work economically at the time of writing. Cryptocurrencies and the blockchain technology are still in its infancy and therefore the price is extremely volatile. A sudden rise or drop of 40% in price is not rare and would heavily influence the customer experience. Additionally, only very few people actually own cryptocurrencies and are willing to pay or receive payments using cryptocurrencies. Because of the low adoption rate, the concept would not be economically feasible as a real product at the time of writing.

3.5 State of the Art

The exploratory research conducted for this thesis focused on studying concepts and design choices of similar projects completed in the commercial field.

A few companies already started researching and experimenting with blockchain technologies and even went as far as combining it with e-mobility. A company called accessec GmbH built a prototype of a car wallet that works with the IOTA cryptocurrency and integrates a point of sale allowing to conduct transactions with vendors seamlessly[20]. Bosch teamed up with the energy supplier EnBW to build a prototype of a blockchain based charging station[21].

The project that came closest to the topic of this bachelor thesis is called Share&Charge, formerly known as Blockcharge[22][23], which was founded by Innogy, a subsidiary of the energy company RWE[24]. It launched at the end of April 2017 with close to 1,500 charging stations, but the project was closed merely a year later[25]. It claimed to be a P2P charging network calling itself the "AirBnB of Charging Stations". It was running on the Ethereum mainnet where anyone could become a charging station owner by purchasing a smart electrical socket to sell electricity. At first glance it seemed like the solution to this thesis' question, but upon further investigation the decentralization aspect had to be questioned. The electrical socket was communicating with a smartphone app to manage the purchase of electricity. This app had to be preloaded with fiat currency via PayPal, etc., the transaction was conducted in fiat and a charging station owner could only withdraw fiat currency as well. There is no evidence that the monetary transaction itself was handled on the blockchain and not just the record aspect of it. Additionally, it was claimed that the system worked without a middleman but the fee that had to be paid to Share&Charge with every charging process contradicts that claim. All this information led to the conclusion that the private keys probably were not handled by the users themselves which is a crucial point in building decentralized applications.

To summarize, all projects combining blockchain with e-mobility seem to be bringing existing centralized business models to the blockchain and could work just as well with traditional payment methods. Furthermore, no technical information, let alone source code, could be found that could be used or improved upon for this bachelor thesis.

4 Implementation

This chapter will go into the details of the implementation of the previously worked out concept, explain why certain design choices were made as well as all challenges that were faced to successfully implement the payment system on an embedded platform.

4.1 State Machine

The concept was implemented on the microcontrollers as finite state machines. As seen in Figure 4.1, the implementation of the payment system is divided into four phases. Both, the customer and the supplier follow this flowchart for their state machine.

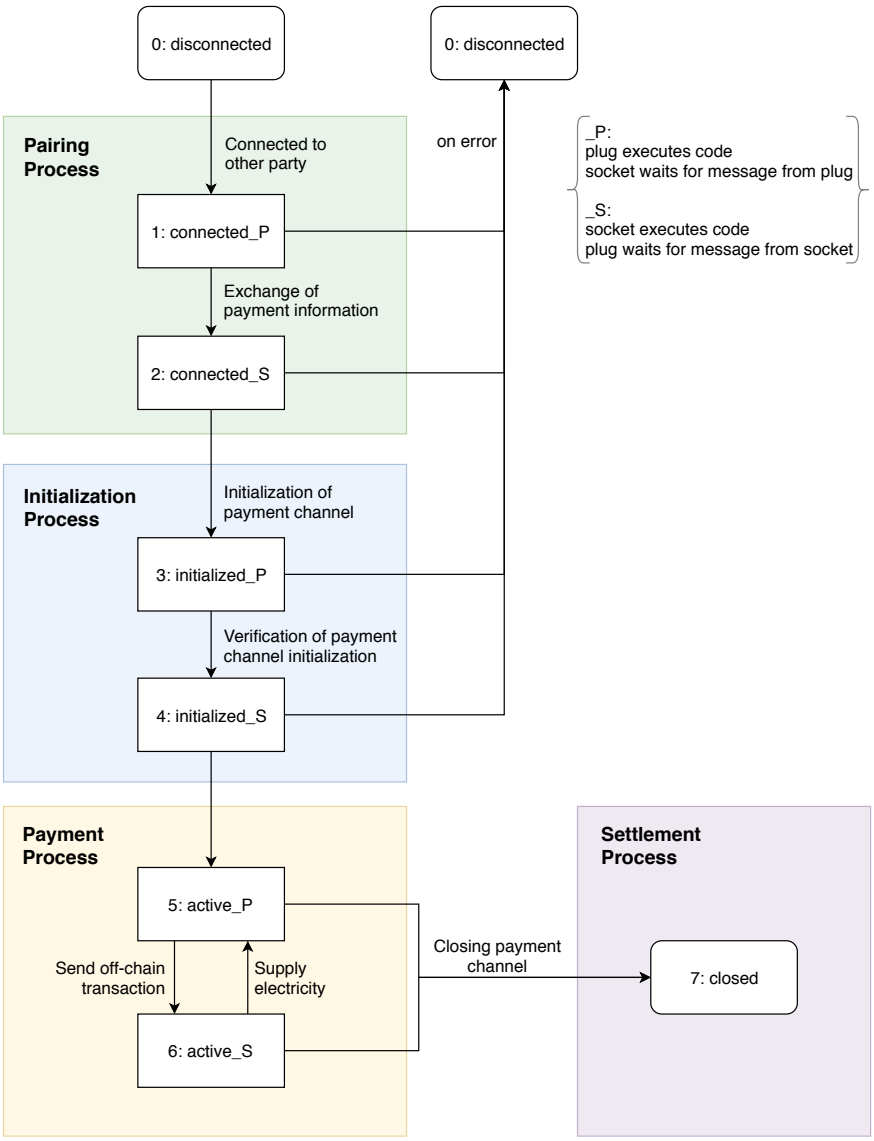

Figure 4.1: Concept of the state machine

According to the different states, different code is executed – the suffix *_P* means that the plug is currently executing code and the socket is expecting a message from the plug, *_S* means the opposite.
During the pairing process, both parties exchange all required payment information with each other. The payment channel is initialized and all payment information is verified by both parties in the initialization process. During the third phase, the plug and the socket exchange off-chain transactions for electricity until any party wants to stop the payment process. Then the settlement process is initialized closing the payment channel and finalizing the transactions on the blockchain.

Pairing Process

The goal of the pairing process is to establish a connection between the customer and the supplier and exchange all necessary payment information.

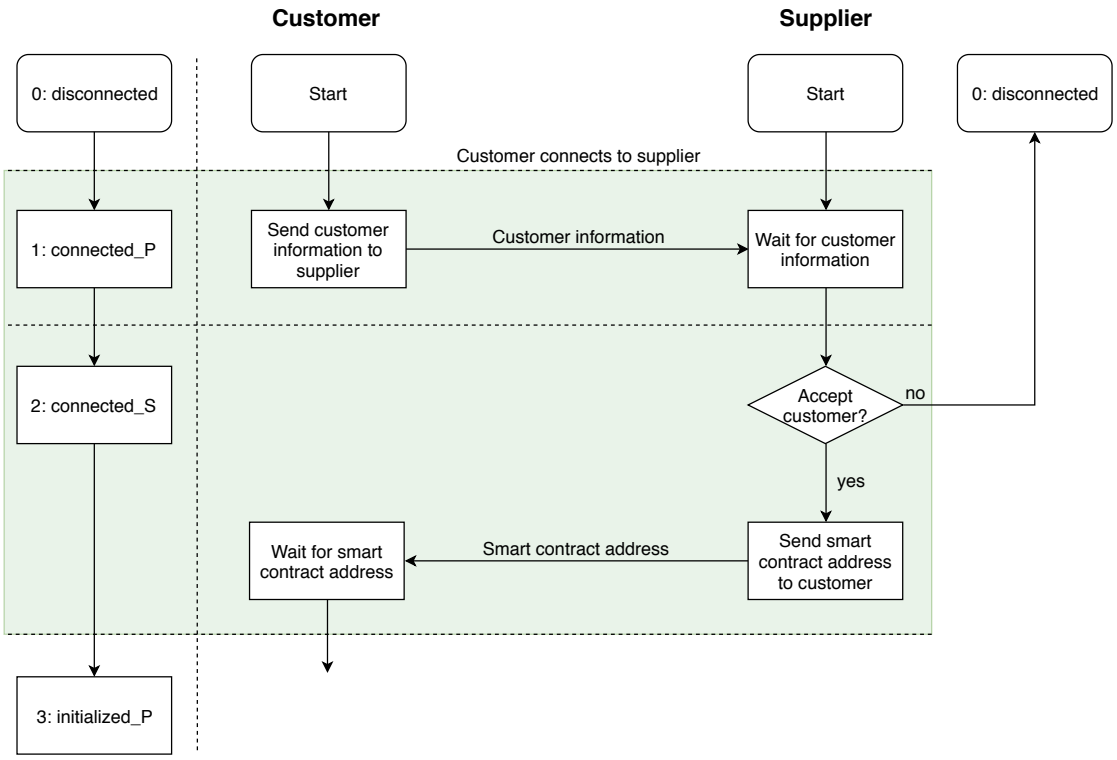

Figure 4.2: Pairing process

Both, the customer in form of the plug and the supplier in form of the socket start in the *disconnected* state.

As soon as a new customer connects to the socket and wants to purchase electricity, both parties transition to the *connected_P* state. The socket now expects the

customer information, in this case the Ethereum address, from the plug. The address is important to know as it will be later used to verify off-chain transactions and whether the payment channel was initialized correctly.

When the customer sends the required information, the plug and the socket enter the *connected_S* state. The supplier can check the address against white- or blacklists and decide whether to accept payments from this address or not. If the customer is accepted, the socket transfers the address of the smart contract, as it is required by the customer to fetch the rest of the payment information. Both parties now enter the initialization process.

Initialization Process

During the initialization process demonstrated in Figure 4.3 all the necessary payment information is fetched from the smart contract and the payment channel is initialized by the customer.

Both parties start in the *initialized_P* state. The plug is required to retrieve all relevant payment information which is stored in the smart contract, where it can be easily updated if necessary. Payment information required for the transaction is the price per second and the payment interval to calculate the value of each off-chain transaction sent. Optional payment information could be the owner of the smart contract, a minimum deposit value, minimum and maximum charging durations and the expiration date, depending on the implementation. The plug should also verify that there is no other payment channel currently active.
After the price was fetched from the smart contract, the customer can decide whether to accept the price and continue with the initialization process or to disconnect.
If the price is accepted, an on-chain transaction is signed on the microcontroller and submitted to the Ethereum network. This transaction calls a function inside the smart contract and initializes the payment channel. A maximum amount of Ether that the customer is willing to spend during the charging process is passed alongside the smart contract call which will be deposited into the contract. The customer now has to wait until the transaction was mined. If the payment channel was initialized correctly, the socket is notified about the initialization and both devices enter the *initialized_S* state.

If the socket can confirm that the payment channel was initialized correctly, it notifies the customer that it is ready to accept off-chain transactions and the payment process begins.

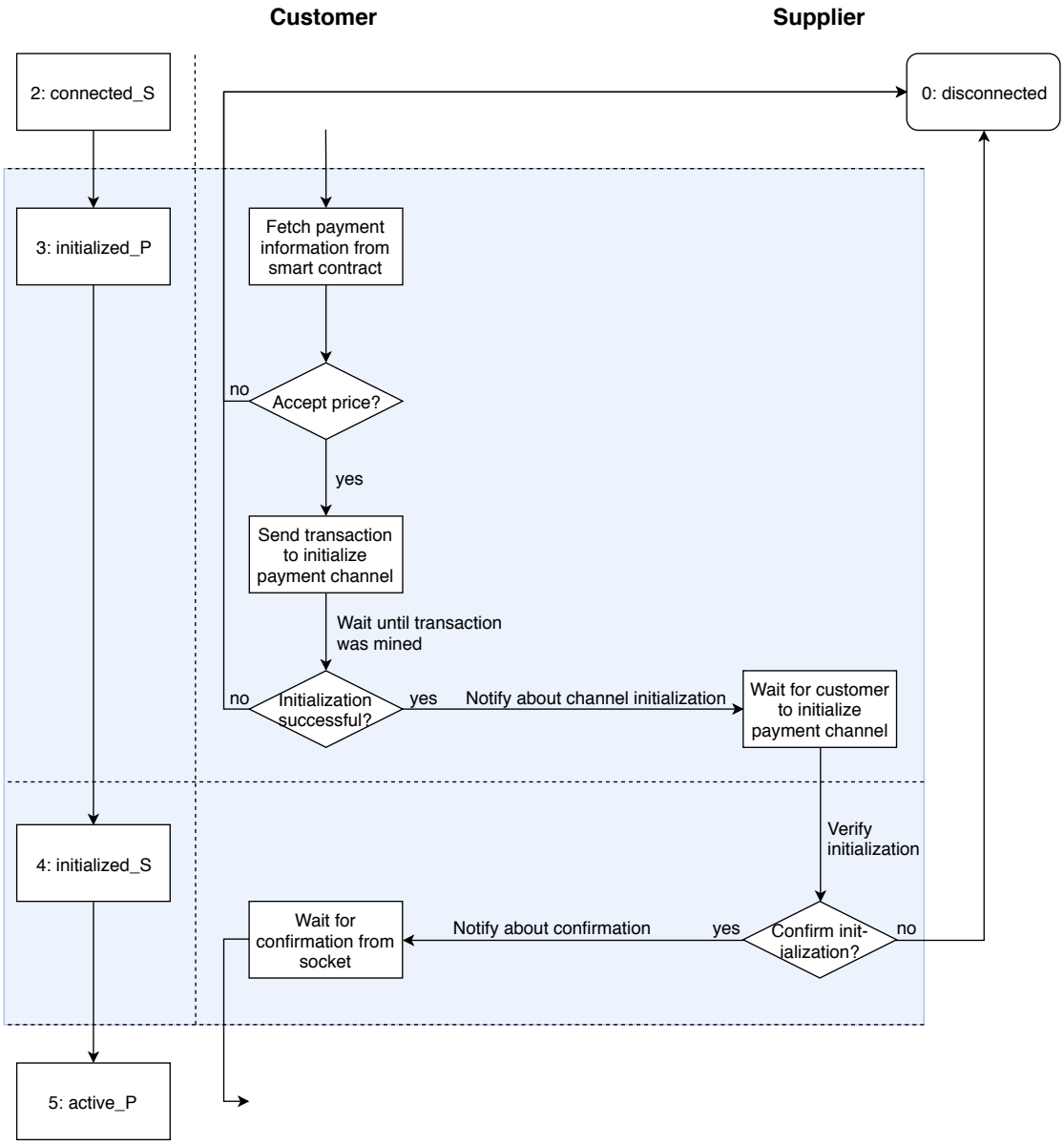

Figure 4.3: Verification process

Payment Process

The goal of the payment process is to exchange off-chain transactions for electricity.

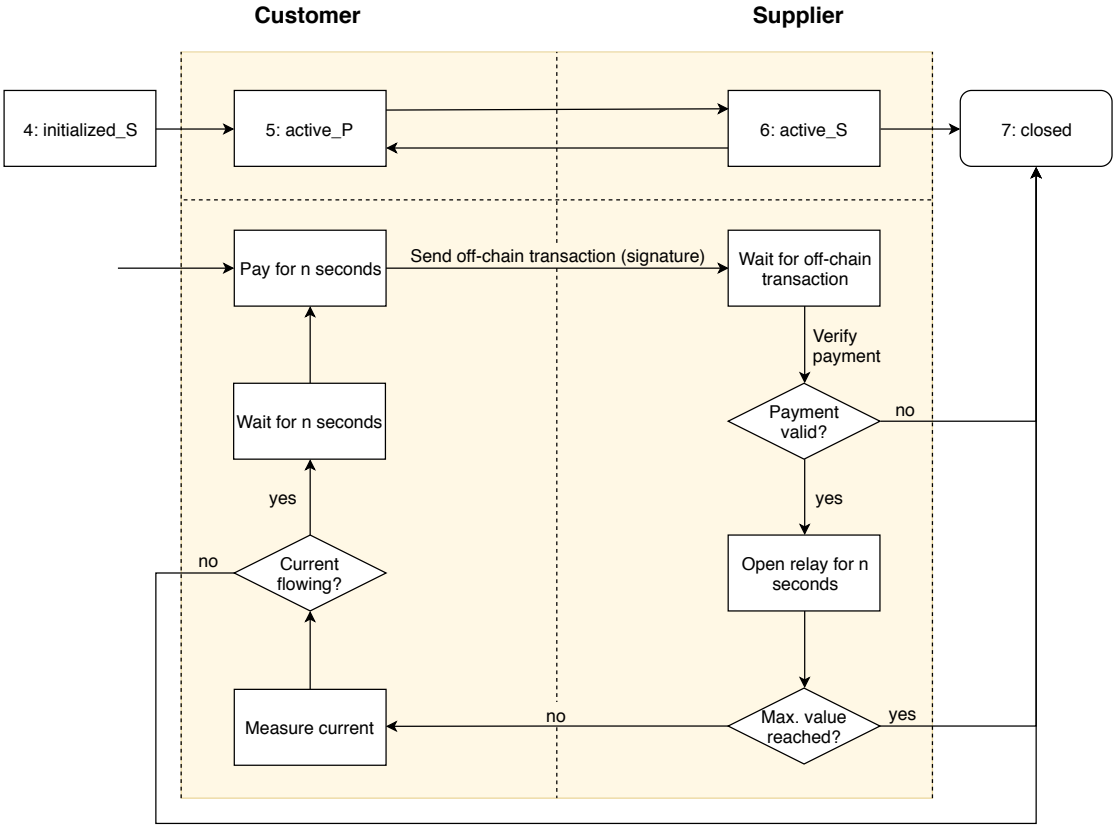

Figure 4.4: Payment process

The payment process starts when the customer sends the first off-chain transaction to the supplier. As it would be inconvenient for the socket to store and submit hundreds of off-chain transactions, only one signature can be submitted to the smart contract. With every transaction, the plug increases the value of the off-chain transaction. Every signature is valid, but it is in the interest of the supplier to submit the latest signature with the highest value. The value of the transaction is calculated as follows:

$$value = number\_of\_transactions * price\_per\_second * seconds\_between\_transactions$$

With the assumed values from Chapter 3 (a total price of 18 € over a duration of 11 hours) roughly 0.007 € are transmitted with each off-chain transaction. With the formula above, this means that the value hashed and signed is 0.007 € for the first transaction, 0.014 € for the second, and so on. As the smart contract operates on ETH and not €, the values are converted to Wei first.

When the socket receives an off-chain transaction, it should verify that the payment is actually valid. Both parties keep track of the number of transactions sent and know the price per second, therefore the data that was digitally signed by the customer can

be recreated. Using the data and the signature received from the plug the signer's Ethereum address can be recovered. It should match the Ethereum address received during the pairing process for the transaction to be valid.
If the off-chain transaction is valid, the microcontroller can close the relay for the specified amount of seconds, supplying the plug with electricity.

To ensure that the socket doesn't deliver more electricity than the plug can pay for, the supplier has to check whether the maximum value, that was deposited into the smart contract, was reached. If that is the case, the channel is closed, ending the exchange of electricity.

After the plug sends a transaction that pays for the specified amount of seconds, it starts measuring the current. As soon as it is detected, the plug starts a timer and waits an appropriate amount of time to send the next transaction. For the implementation, a new transaction is sent to the socket when there is 5 seconds of paid electricity left to ensure a steady flow without interruptions. If the customer paid for electricity, but no current is measured in return, it can immediately interrupt the payment process by closing the payment channel, without losing another payment.

Settlement Process

The settlement process closes the payment channel taking the value of the off-chain transactions and paying out the balances to each participant on the blockchain, finalizing the exchange.

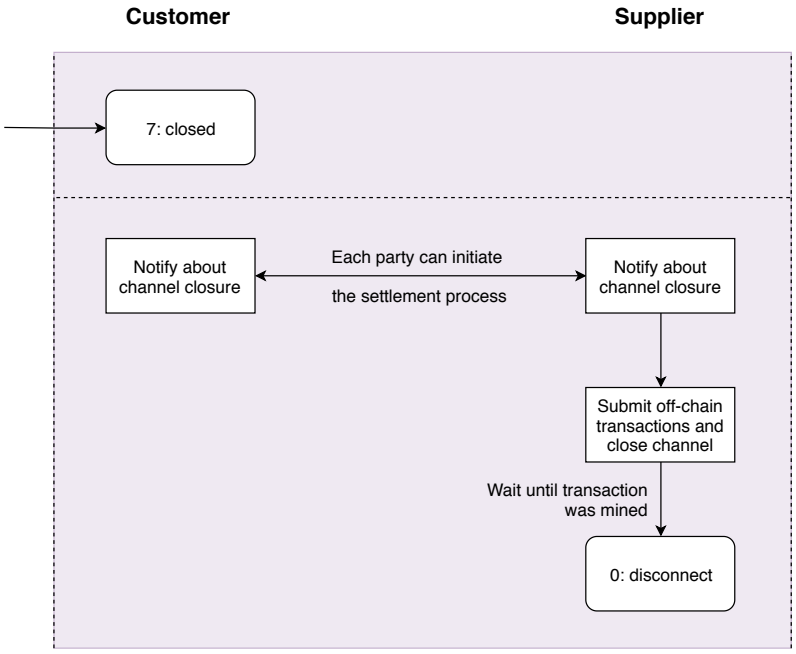

Figure 4.5: Settlement process

As soon as a party wants to stop the payment process, e.g., when the battery of the customer is fully charged or the socket received an invalid signature, it can notify the other party about it. The socket signs an Ethereum transaction which calls a function inside the smart contract that takes the latest off-chain transaction, which contains the total value spent by the plug up to this point, as an argument. The smart contract verifies the submitted signature and pays out the amounts accordingly.
The smart contract also protects the customer, as it was mentioned before. If the supplier fails to submit a valid signature in a certain amount of time and the payment channel expires, the deposited amount can be withdrawn again.

4.2 Transactions

During the initialization, payment and settlement process transactions have to be sent by the plug and socket. This section will go into detail on how transactions were implemented for this thesis.

When researching existing implementations of the generation of on-chain transactions for the Ethereum network on a microcontroller level, it was discovered that these implementations are far from production ready, as opposed to implementations with high level programming languages. In an effort to find existing code that could be used as a foundation for the implementation some utility libraries were found, but many of them were not compatible with the microcontroller and the ones that were, often had to have issues fixed or additional functions implemented.

Payment channels and off-chain transactions were implemented in this bachelor thesis as well. They are a subset of state channels, which currently are a heavily researched topic on their own, therefore no standard exists that could be followed for the implementation. As far as exploratory research went, every state channel implementation was still under development and not a single implementation of a payment channel on an embedded level could be found[26]. Therefore an entire concept for the payment channel had to be developed, that could run on microcontrollers.

The process of sending an on-chain transaction can be broken down into the following steps:

1. Gather data
2. Encode data
3. Hash data
4. Sign hash
5. Submit signature

Off-chain transactions were implemented in a way to mimic the functionality of on-chain transactions. The following paragraphs will go into detail of the implementation of on- and off-chain transactions, why certain design choices were made and which challenges were faced.

1. Gather Data

The following information is necessary to send an on-chain transaction on the Ethereum blockchain:

1. *to*: Receiving address of the transaction.
2. *value*: Value of the transaction in Wei.
3. *data*: Hexadecimal data can be passed with the transaction. When sending a transaction to a smart contract, the data is used to define which function is called and to pass arguments. The first four bytes are the function identifier followed by the function arguments which are encoded according to the "ABI specification"[27]. As the signature, which is a dynamic byte array, is passed to the smart contract alongside other variables, the encoding according to the specification had to be implemented.
4. *nonce*: The nonce is important, as it protects a user against replay attacks and guarantees that transactions are executed in the correct order. The nonce starts at zero and increments by one after a transaction was sent. This means that transactions, where the nonce is less than the total amount of transactions sent by the account, are rejected by the Ethereum network. Without the nonce an attacker could resubmit a transaction, that was already mined, over and over again to drain the balance of a victim.
5. *gas price*: The fee of a transaction that gets paid out to a miner in Wei. The higher the fee, the faster the transaction will be mined. The Rinkeby testnet uses a *PoA* (Proof of Authority) mining algorithm, which only allows a few accounts to act as miners, as in a test environment without monetary incentives, stability is more important than decentralization. Therefore there are no competing miners and high transaction fees are not relevant. A transaction with a gas price of 1 GWei will be usually included in the next block resulting in a confirmation time of less than 15 seconds. On the mainnet the gas price should be checked carefully, so the transaction is mined in a reasonable amount of time to not interfere with user experience.
6. *gas limit*: The gas limit specifies how much code can be executed with a transaction. The base limit of any transaction, whether it is a smart contract call or not, is 21,000 gas – it covers the storage of the transaction on the blockchain as well

as the elliptic curve operation to recover the sender of the transaction[28]. Each additional computational step costs more gas, e.g., an addition (opcode: ADD) costs 3 gas, loading a variable from storage (opcode: SLOAD) costs 200 gas and even goes as far as 20,000 gas if a storage value (variable stored on the blockchain) is set from zero to non-zero. As it can be seen, storing data on the blockchain is one of the most expensive operations to discourage storing vast amounts of information on-chain. If the gas limit is set too low and there is not enough gas to finish the smart contract execution, the transaction is unsuccessful and reverts any changes made. If the limit was set too high, any unused gas is refunded to the sender. Thus, when implementing smart contract calls on the microcontroller, it is important to provide enough gas for all computations. The total transaction fee is calculated by multiplying the gas price with the gas limit.

7. *chain ID*: Each Ethereum blockchain has a unique chain ID to differentiate between different chains. For example the mainnet uses the ID 1 and the Rinkeby testnet uses the ID 4. With the Ethereum Improvement Proposal 155 (EIP-155)[29] the chain ID should be included into a transaction to prevent so-called cross chain replay attacks, where a transaction on one Ethereum chain can be resubmitted to another Ethereum chain.

Although the functionality of off-chain transactions mimic on-chain transactions, not every parameter listed above is required. Because the payment channel was implemented between two parties only, the *to* parameter can be left out. The value, as demonstrated in the previous chapter, is defined as the total value spent up to a specific point in Wei and is included in the off-chain transaction. There is no need to pass any data with the off-chain transactions, therefore no data is included. No transaction fees are paid either, so the gas limit and gas price can be dropped from the off-chain transaction. A nonce is mandatory to protect the customer. Without it, a supplier could just resubmit an off-chain transaction with a higher value from a previous exchange, stealing money from the customer. The smart contract keeps track of the nonce of each customer and needs to be fetched by both parties during the initialization process. Although the *chain ID* is not required to be included, a similar kind of attack can be conducted across different smart contracts. If every socket has its own smart contract, an off-chain transaction with a higher value originally submitted on one smart contract, can be wrongfully resubmitted for a different exchange on another smart contract. Therefore, the address of the smart contract managing the payment channel is included inside the off-chain transaction.

2. Encode Data

On-chain transactions are encoded with the *RLP* (Recursive Length Prefix)[6]. Before the data listed above is hashed, it is converted into hexadecimal and therefore byte-

arrays. The benefit of RLP encoding is that parameters of variable length are encoded in a way that all parameters can be extracted from the resulting byte array in the end. This is mandatory, so miners can extract the required data from the encoded byte array to fulfill the transaction. RLP was especially designed for Ethereum to efficiently encode these values with the following rules:

- If a single byte is in the range of 0x00 and 0x7f, the byte is its own encoding.
- If a byte array is 0 to 55 bytes long, the byte array is prefixed with a single byte with the value 0x80 plus the length of the byte array.
- If a byte array is longer than 55 bytes, it is encoded by a first byte that has the value 0xf7 plus the length of the length prefix of the byte array, followed by the bytes describing the length of the byte array and finally the byte array itself.
- A list of byte arrays can be encoded as well, as it is required for the transaction data. All items are encoded individually first and then encoded again as a total byte array. This allows to encode data, no matter how nested the information is.

For the implementation, code from a RLP C++ library was used, which was written by Takahiro Okada[30]. The following changes had to be made to the library so it could be used for the implementation.

- The library had to be rewritten, to implement encoding of transactions into one utility library and to make it compatible with the microcontroller.
- The library now supports the EIP-155[29] when encoding transactions with the chain ID.
- A wrong calculation was fixed that would incorrectly encode byte arrays longer than 55 bytes.
- The memory allocation of the library had to be adjusted, as it did not allocate enough memory to handle all data that needed to be encoded for the transactions.

The smart contract validates an off-chain transaction by recreating the signed data and using it to recover the signers address from the signature. As the nonce and the contract address are already stored inside the smart contract, only the value and the signature have to be passed to the smart contract function. The Solidity smart contract programming language uses so-called encoding in packed mode before hashing as a standard[31]:

```
abi.encodePacked(
  _value,
  contractAddress,
  customerNonce
)
```

To ensure consistency, the same encoding had to be implemented on the microcontrollers with the following rules:

- all variables are converted to bytes and concatenated together
- "types shorter than 32 bytes are neither zero padded nor sign extended"[31]
- "dynamic types are encoded in-place and without the length"[31]

3. Hash data

The encoded data of both, on- and off-chain transactions, is hashed using the Keccak-256 hashing algorithm. This algorithm is almost identical with the official SHA-3 implementation, but Ethereum uses Keccak-256 instead, which was "the winning entry to the SHA-3 contest"[6].

Before signing a signature that is used on the Ethereum blockchain, e.g., an off-chain transaction, it is recommended that the data is prefixed with an Ethereum specific prefix[32]. The prefix makes the signature recognizable as an Ethereum specific signature and protects users from attackers letting them unknowingly sign a transaction, instead of a message.
The encoded data is hashed, then prefixed according to the following rules:
"\x19Ethereum Signed Message:\n" + len(message) + message
As a 256 bit hash is the message that is signed, the length of the prefix is always 32 bytes. The prefixed message is then hashed again:

```
bytes32 message = keccak256(
  abi.encodePacked(
    value,
    contractAddress,
    customerNonce
  )
);

bytes32 prefixedMessage = keccak256(
  abi.encodePacked(
    "\x19Ethereum Signed Message:\n32",
    message
  )
);
```

The plug mirrors the smart contract implementation demonstrated above during the generation of off-chain transactions. It needs to produce the same hash for the same input data as the smart contract to make it verifiable for the closure of the payment channel.

4. Sign data
Although Ethereum relies on ECDSA elliptic curve signatures, there are some additional rules and features that differ the signature process from its standard implementation:

- It takes up to two guesses to recover the public key / address from the r & s values of an ECDSA signature. Therefore, a third value v, also called the *recovery ID* is calculated, that enables the immediate extraction of a public key from a signature. According to the Ethereum yellow paper[6] "the recovery identifier is a 1 byte value specifying the parity and finiteness of the coordinates of the curve point for which r is the x-value". The recovery ID is determined during the signature process by looking at the y value on the elliptic curve with the following formula:
  $v = 27 + (y\%2)$
  i.e., if y is even the recovery ID equals to 27, if it is odd, it equals to 28. The formula is used for signatures of all kinds. With the aforementioned EIP-155, optionally the following formula can be used for on-chain transactions, securing them against cross-chain replay attacks:
  $v = chain\_id * 2 + 35 + (y\%2)$
  This means that for transactions on the Rinkeby network, the recovery ID is either 43 or 44. For the generation of off-chain transactions the first and for on-chain transactions the second formula was implemented on the microcontrollers.

- According to the Ethereum yellow paper[6] the s part of the signature has to be less or equal than half of the numeric value of the elliptic curve n.

As a foundation, a standard implementation of the ECDSA algorithm[33] was used. Both additional features of the Ethereum specific signatures had to be implemented extending the library.

5. Submit signature
To submit an on-chain transaction, the transaction data and the signature has to be RLP encoded again. The resulting byte array can then be submitted to the Ethereum network by sending the transaction to the node.

As previously mentioned, both the plug and the socket keep track of all transaction data themselves. Therefore, when the plug is paying for electricity with an off-chain transaction, it simply sends the signature to the socket.

Additional challenges
Smart contracts mainly work with 256 bit unsigned integers for numbers, but most microcontrollers are not capable of working with integers this big. When fetching a *uint256* value from a smart contract, the value returns as a hexadecimal string from

the node. The largest fixed width integer type *uint64_t* or *unsigned long long* can only store 64 bits of data which equals roughly 18.44 ETH. If the value exceeds exceeds 64 bit, parsing, storing and calculating with the value becomes a lot more challenging, as the data has to be processed as a byte array.

As all the communication with the Ethereum blockchain is processed through the node and http requests, all data is mostly handled as hexadecimal represented by char arrays. These use more space than byte arrays. During the entire process of generating on-chain transactions, the data for the smart contract call has to be encoded, then encoded with the rest of the transaction data, hashed, signed and the resulting signature has to be encoded with the transaction data again. The generated transaction then has to be put into a JSON http request body and the response that returns from the node has to be JSON parsed to receive required data. This generates a lot of data mainly in the form of char arrays which can get long, depending on the arguments passed to a smart contract. On an embedded platform with limited memory a lot of optimization and memory management is required to successfully generate a transaction.

4.3 Smart Contract

Security is extraordinarily important for smart contract development, as possibly large sums of money are handled and the code is immutable, and can't be changed once the smart contract has been deployed to the Ethereum network. This section will go over the smart contract implementation, some security best practices and design choices. The entire source code can be found within the appendix, listings 4 & 5.

The smart contract was designed in a way that only one contract is responsible for one socket which can only handle one payment at the time. One of the most important rules for smart contract development is to keep the code as simple as possible. Adding complexity only increases the risk of a critical issue. When the customer initializes the payment channel, a maximum transaction value is deposited which is kept inside the smart contract until the channel is closed or expires. Additionally, the address of the customer is stored inside the smart contract as a global variable which will be used to verify the off-chain transactions:

```
// set channel customer to the address of the caller of the transaction
channelCustomer = msg.sender;
```

The channel can be timed out by any participant on the network after the expiration date has been reached, returning the deposited funds to the customer. The function to close the channel can only be called by the supplier. The total transaction value and the signature of the final off-chain transaction is passed to the function as an argument.

From the three values inside the off-chain transaction, the transaction data is regenerated. The nonce and the address of the smart contract cannot be forged, as these values are not passed by the supplier. Instead, they are taken directly from the storage of the smart contract itself. The smart contract repeats the same steps to generate the transaction data as the plug – the data is hashed and prefixed with the Ethereum specific prefix:

```
bytes32 message = keccak256(
  abi.encodePacked(
    // value passed as an argument
    _value,
    // variable to receive the smart contract address
    address(this),
    // gets the nonce of the current channel customer
    customerNonces[channelCustomer]
  )
);

bytes32 prefixedMessage = keccak256(
  abi.encodePacked(
    "\x19Ethereum Signed Message:\n32",
    message
  )
);
```

The computed *prefixedMessage* should result in the same data that the customer used to sign the off-chain transaction. Solidity offers a function to recover the address of the signer, if provided with the signed data and the signature.

```
return ecrecover(prefixedMessage, v, r, s) == channelCustomer;
```

The function returns true if the recovered address matches the current payment channel customer. The verification of the off-chain transaction is tamper proof. If the supplier tried to provide a false value with the signature, the recreation of the data would result in data that the customer did not sign. Therefore, the *ecrecover* function would return an address that does not match the current channel customer. Only the correct data both parties agreed to can be used to close the payment channel.

If the signature was valid, the nonce of the customer increments by one, making all off-chain transactions up to this point invalid. Then the transaction value is paid out to the supplier and the rest is refunded to the customer, closing the payment channel.

4.3.1 Best practices

The smart contract implements the so-called pull over push method[34]. When closing the payment channel, the according transaction values could be immediately paid out to the customer and the supplier. An attacker could write a smart contract that would initialize a payment channel and would always revert the transaction when it received Ether during the settlement. In this case the payment channel would be locked forever, as the customers balance could be never paid out. Instead, a balance variable is used, from which each party can withdraw their share without the risk of blocking the smart contract.

Solidity has no built in checks for integer under- and overflows, which can become a major security issue, especially when dealing with balances, allowing an attacker to drain a smart contract. A so-called SafeMath library, as the one found under Listing 4 can help in preventing integer under- and overflows.

Another security concern that comes up when withdrawing Ether from balances is the reentrancy attack. In traditional programming, an amount would be paid out and upon success, the balance would be adjusted to reflect the new balance. In smart contract programming this practice resulted in a hack where an attacker recursively withdrew the balance from a smart contract, resulting in theft of over 3.6 million ETH, which is worth 792 million € at the time of writing[19]. Therefore, it is very important to set the new balance before transferring Ether. If the transfer fails, the entire smart contract call is reverted, so the new balance only comes into effect if the transaction was successful.

5 Conclusion

The goal of this thesis was to examine whether M2M payments could be implemented on a microcontroller level and which payment method was suited best for this task. A concept of an electrical plug and an electrical socket was developed, which exchange electricity for monetary value. This worked out concept was successfully implemented on embedded devices.

An underlying payment system was developed to enable fast and secure transfers of monetary value between two participants. The system uses the Ethereum blockchain to initialize and settle a payment channel, which enables instant and feeless transactions between parties inside this channel. The transactions are secured through digital signatures and the smart contract acts as a trustee managing the monetary value while protecting both parties from fraud and minimizing the risk of theft dramatically. The payment system enables a true P2P market, allowing everyone to become a charging station provider and to securely sell electricity to electric car owners. As there is no need for a third party as a middleman that oversees the transaction, the total transaction fees for a charging process of any length and value should average at less than 0.10 € at the time of writing. This bachelor thesis can not only be used for this specific concept, but many different P2P business models can be developed using the payment channel as a base as well.

This implementation is most likely the first implementation of a payment channel on an embedded platform, as far as research went. For Ethereum specific functionalities, e.g., signatures and encoding that were especially created for the blockchain technology, libraries exist in high level programming languages, but had to be implemented on embedded devices for this purpose. Additionally, many computational steps and memory allocation had to be optimized to make the payment system work on microcontrollers. This means that the code that was written for this bachelor thesis can be used in the future not only for payment channels, but as the groundwork for basically any communication between an embedded device and the Ethereum blockchain and smart contracts.

5.1 Outlook

As cryptocurrencies are still in their infancies, most of the development is still ahead and nobody can imagine what they will look like in 10 years. Similarly, when the internet was in its infancy, no-one imagined that something like Facebook could even exist. The technology will scale to process more transactions at even lower fees and will be steadily integrated more and more into our daily lives. For example, in its latest flagship, the

Galaxy S10, Samsung included an Ethereum and Bitcoin wallet, meaning that millions of people are now able to transact with cryptocurrencies securely without any big entry barriers.

Eventually, the price of cryptocurrencies will stabilize, but until that happens, some suggestions for future work that can build upon this thesis are proposed. Currently, the microcontrollers are communicating with each other over a WiFi connection and WebSockets and the plug has to be powered separately to function. An implementation that would improve upon this, would have the socket and the plug communicate over PLC, i.e., over the electricity that is transmitted. Before the actual electricity delivery, the socket could limit the output so that the microcontroller of the plug can be powered and communicated with, without the need for an external energy source. Another interesting topic would be to implement the payment system with other cryptocurrencies, such as IOTA or Nano to analyze the key strengths and weaknesses of the different payment methods. As fast as the crypto-ecosystem develops, there could be new cryptocurrencies in a few months that couldn't even be considered during this thesis. In about a year at the time of writing, Ethereum plans to upgrade to a new blockchain, also called Ethereum 2.0[35] which will implement a new mining algorithm and lots of different scaling solutions. This might bring potential new ways to implement an even more efficient payment system.

6 Appendix

6.1 Setup Instructions

This section will focus on the technical setup of the hardware components of a functioning prototype and the installation and setup of all required software.

6.1.1 Socket

The Sonoff S20 smart socket was used for the technical implementation of the concept. A microcontroller is automatically powered by the socket it is plugged into. The S20 also has a WiFi module and a relay, which can be switched on and off by the microcontroller. The microcontroller can be reprogrammed via a serial port. To program the Sonoff S20 the device has to be opened, revealing the logic board with the relay and the serial port.

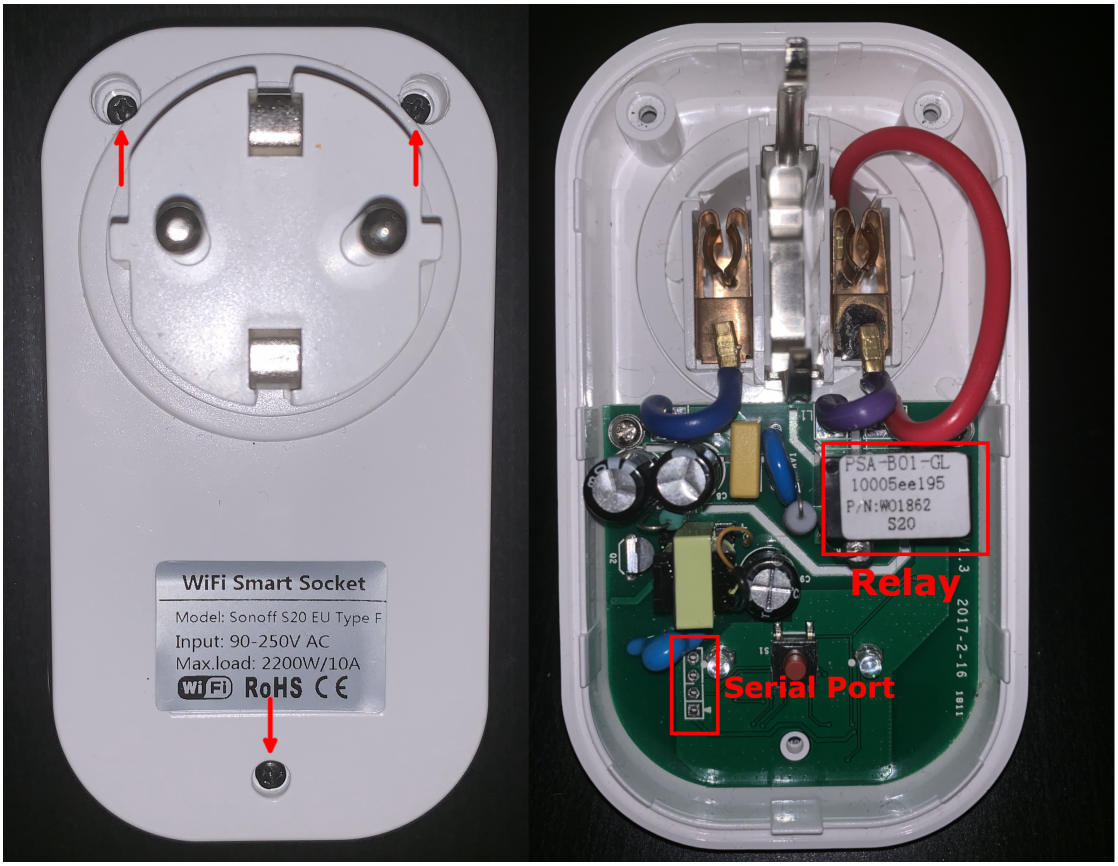

Figure 6.1: Sonoff S20

To program the microcontroller with a computer a FTDI USB to serial converter is used. The converter has to be plugged into the S20 as follows:

FTDI Converter	Sonoff S20
GND	GND
TX	RX
RX	TX
3.3V	3.3V

Table 6.1: Connection between serial converter and microcontroller pins

It's important to notice that the FTDI converter must operate at 3.3V entirely. Caution: some converters only switch the TX and RX pin to 3.3V while the VCC remains at 5V. This can fry the internals of the S20. To program the microcontroller, the button has to be pressed before plugging the pins into the serial port to put it in programming mode. After the pins have been inserted, the button can be released shortly after.

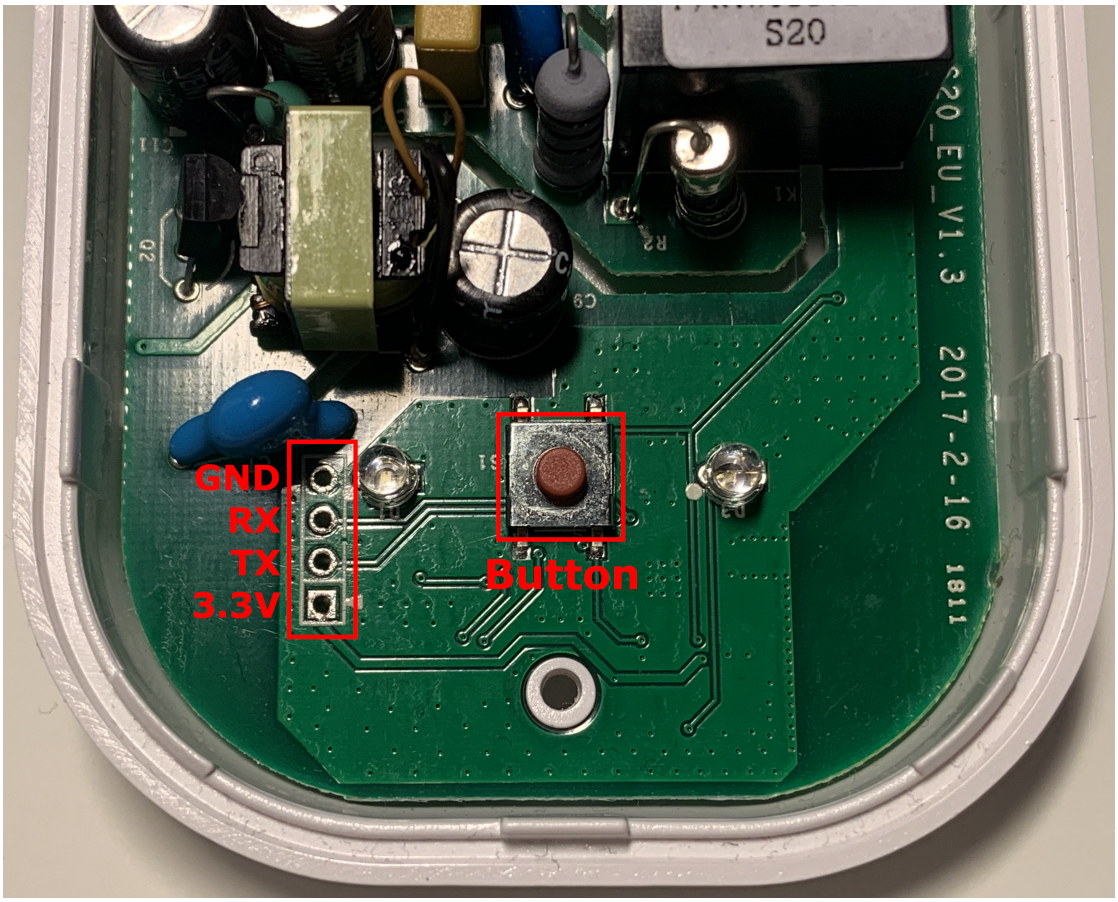

Figure 6.2: Serial ports of the Sonoff S20

Both, the socket and the plug, are programmed using the Arduino IDE. The following steps need to be followed to install the ESP8266 Board, which the S20 is based on:

- Inside the Arduino IDE open "Preferences"
- Enter `http://arduino.esp8266.com/stable/package_esp8266com_index.json` under "Additional Boards Manager URLs"
- Open Tools → Board → Boards Manager
- Search and install "esp8266" by "ESP8266 Community"

After connecting the FTDI converter to the computer, it should appear under Tools → Port. To successfully flash code to the S20 the following settings have to be set:

- *Board*: "Generic ESP8266 Module"
- *CPU Frequency*: "80 MHz"
- *Flash Size*: "1M (no SPIFFS)"

6.1.2 Plug

The device to control the measurement of current in the plug and handle the communication with the socket is the Heltec WiFi Kit 8, which is based on an ESP8266 as well and has a 0.91 inch OLED display.

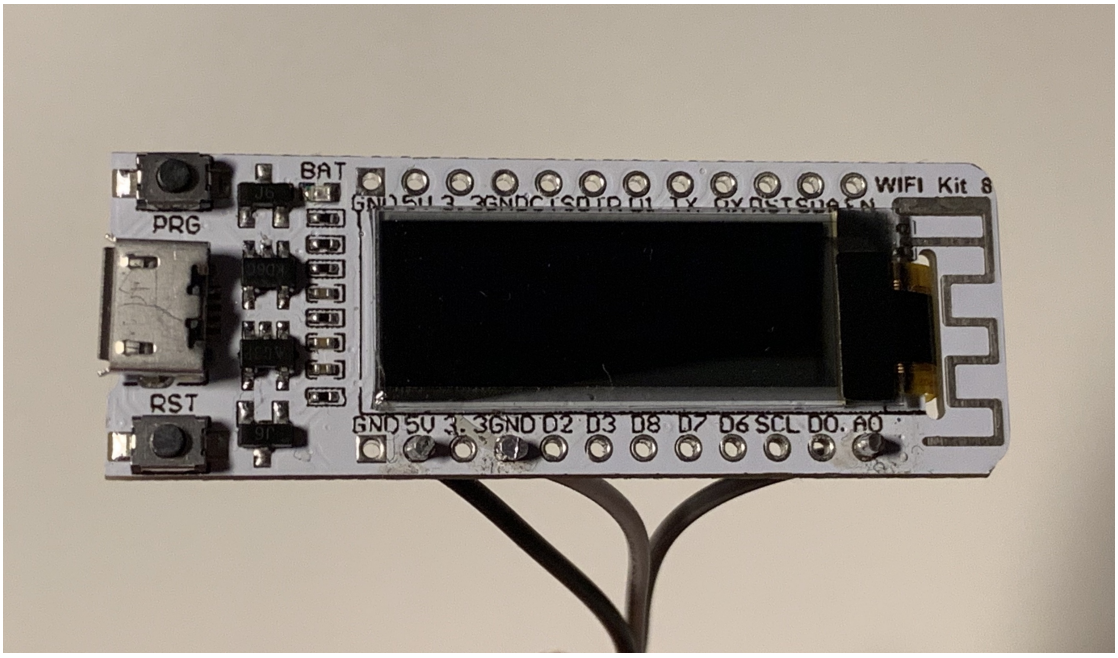

Figure 6.3: Heltec WiFi Kit 8

The ACS712 20A current meter is used to measure the current. It's hall effect-based and provides galvanic isolation up to a minimum of 2.1 kV (RMS)[36]. To connect the current meter, a part of the hot wire leading to the plug has to be cut and stripped. Both ends have to be inserted into the screw terminal of the ACS712. The current from the plug is now redirected underneath the hall sensor.

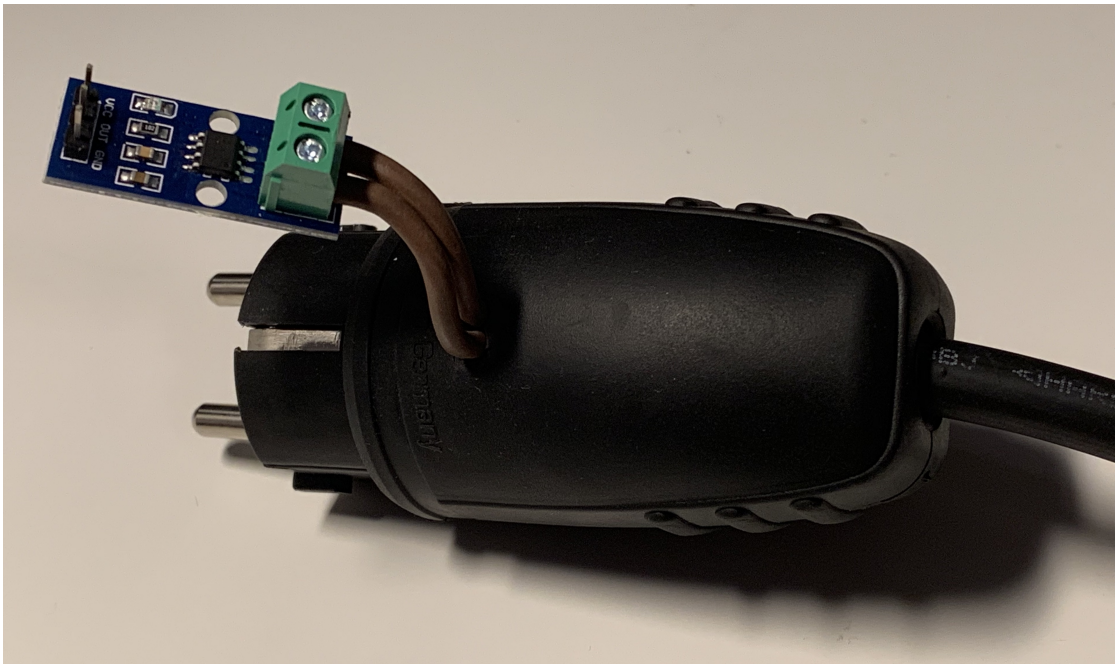

Figure 6.4: ACS712 current meter connected to plug

The pins of the current meter have to be soldered to the Heltec board as follows:

ACS712	Heltec WiFi Kit 8
VCC	5V
OUT	A0
GND	GND

Table 6.2: Connection between current meter and microcontroller pins

To program the Heltec WiFi Kit 8, USB to UART drivers need to be installed first. The download link can be found under "References"[37]. Next, if the ESP8266 was not installed inside the Arduino IDE yet, the instructions found in the subsection above can be used to install the board.
After a successful driver installation, the board should be found under Tools → Port when the device is connected to the computer.

The following settings are required to ensure a successful flash of the Heltec WiFi Kit:

- *Board*: "NodeMCU 1.0 (ESP-12E Module)"
- *CPU Frequency*: "160 MHz"
- *Flash Size*: "4M (3M SPIFFS)"

6.1.3 Libraries

The prototype relies on some external libraries that have to be installed additionally:

Display
To use the display of the Heltec WiFi Kit a special library needs to be installed. Inside the Arduino IDE, under Sketch → Include Library → Manage Library search for and install the "U8g2" library by "oliver".

WebSocket server
The communication between the plug and the socket relies on WebSockets. Download the library as a zip file from the git repository[38].
Install it via Sketch → Include Library → Add .ZIP Library.

Keccak-256 Library
The implementation relies on a library that was written by Aleksey Kravchenko and was found in the source code of the firefly DIY hardware wallet[39]. This library is included with the source code.

ECDSA Library
Ethereum relies on the Elliptic Curve Digital Signature Algorithm, although there are some differences to the standard implementation of that algorithm. The zip library is provided with the source code of this prototype and extends the "micro-ecc" library by Kenneth MacKay[33].

6.1.4 Node

A server is mandatory to act as a gateway to the Ethereum blockchain. Geth is the official "Golang implementation of the Ethereum protocol"[40] and is used to run a full node. After the installation it will be used to send transactions and make smart contract calls.

A *VPS* (Virtual Private Server) was used for this implementation running Ubuntu 18.04. The server has a six core CPU, 16 GB of RAM, 400GB SSD and 400 MBit/s

unlimited traffic. The following instructions can be used to install the program under Ubuntu. For other environments the link to the instructions can be found under "References"[41]. To install geth add the repository first:

```
$ sudo add-apt-repository -y ppa:ethereum/ethereum
```

Next, install geth:

```
$ sudo apt-get update
$ sudo apt-get install ethereum
```

To interact with the Ethereum blockchain the entire chain history has to be downloaded first. This can take up to 40 GB of disk space and will take several hours of synchronizing. Geth will be started via this console command:

```
$ geth console --rinkeby --rpc --rpcapi="db,eth,net,web3,
personal,txpool" --rpcaddr X.X.X.X --rpcport 8545 --cache=2048
```

The launch options have the following purposes[42]:

- *rinkeby*: synchronizes the Rinkeby testnet
- *rpc*: enables the HTTP-RPC server, allows to receive JSON RPC requests
- *rpcapi*: exposed APIs, a listing of all APIs can be found under "References"[43][44]
- *rpcaddr*: IP address of RPC interface, replace "X.X.X.X" with the IP address of the server, defaults to "localhost". Exposing the RPC interface without any restrictions is not advisable, especially on the mainnet, as it is a severe security concern.
- *rpcport*: listening port of the RPC server
- *cache*: memory allocated in MB, a minimum of 1024 MB is advisable for a faster synchronization

The console parameter starts a JavaScript console, allowing to interact with the blockchain using the web3 library. Geth currently comes with web3 version 0.20.1[45] but might be upgraded to version 1.0[46] soon. The links to the documentation of both versions can be found under "References".
The version of the web3 library can be checked using:

```
> web3.version
```

The output of the synchronization could hamper the ability to properly read the output of the JavaScript console. The verbosity can be set via the following command:

```
> debug.verbosity(x)
```

Replace x with 0 for silent, 1 for error, 2 for warn, 3 for info, 4 for debug and 5 for detail. The verbosity defaults to info[42].

```
> eth.blockNumber
```

returns the block number of the latest synchronized block, which is the amount of all previous blocks. The first block, the also called the genesis block, starts with the block number 0. The current block number can be checked through so-called block explorers, e.g., rinkeby.etherscan.io.

```
> eth.syncing
```

returns the current block number and the highest block number. As soon as the client is synchronized it returns "false"[45].

As previously mentioned, the node was set up to accept and handle api calls in the form of http requests, which follow the JSON-RPC 2.0 specification[47]. The implementation heavily relies on the API, as it is used to not only fetch data from the blockchain, but also to send transactions and make smart contract calls[43][44].

6.2 Smart Contract

The following subsections will describe every step of deploying a smart contract to the Rinkeby Ethereum testnet.

6.2.1 Wallet

The first thing needed to start programming smart contracts is an Ethereum wallet. MetaMask is a browser extension for Chrome, Firefox and Opera, that not only allows to manage multiple accounts on multiple test chains, it also injects the web3.js library into websites allowing to interact with the Ethereum blockchain and smart contracts on web pages. Visit the MetaMask[48] website and download the browser extension. A new account will be generated using a mnemonic phrase, defined in the BIP39 (Bitcoin improvement proposal)[49]. It usually consists of 12 words which represent a private key, essentially creating an easy way to remember / write down private keys. An example for a mnemonic phrase is:

```
short heavy hidden anger nephew tragic fade dad renew finger among tiny
```

This phrase translates to the following seed:

```
b7b36d9ca1e105045344ecb7ca7b9449bfc0889139c9719876d03cf7b5814861
37e905b9e94e50c03ca22871937ae3c754dea1427eede8198c6774d90fc1a1f4
```

Using the BIP44[50] standard an unlimited amount of private keys can be derived from the seed. For example the first private key derived from this seed would be:

```
0xfb8502c03ea336344dc44b66b1a3c01e2917138e92bfa93c54725166394cd46b
```

with the corresponding address

```
0x4d43c1E254a9333fB0D8A50BD3f01b6787ee8895
```

The second derived private key is:

```
0x64b45c024041178aff2f9ed7b7026fff6890c871818c39c1c7bd826e6aa33773
```

with the corresponding address

```
0xd38F7dc2d9B6F6D9d5CB6C8813e213D5DC541458
```

and so on. This means that the single mnemonic phrase will act as a backup phrase for all accounts that will be created inside the MetaMask wallet. After MetaMask was set up create a second account and set the network to Rinkeby. Ethers are required to send transactions on the testnet and can be received via the website faucet.rinkeby.io.

6.2.2 IDE

The smart contract was developed using Remix. It's an online IDE including a compiler for the languages Solidity and Vyper, various debugging and testing tools and can be found under remix.ethereum.org.
After choosing *Solidity* as an environment a compiler has to be set. Because the programming language was designed especially for Ethereum, it is still under very heavy development with frequent updates coming out. The documentation for each specific version can be found under
`https://solidity.readthedocs.io/en/v0.5.8/`
while replacing "0.5.8" (the latest stable version at the time of writing) with the desired compiler version.

Inside the "Run" tab the connection to the blockchain can be chosen under "Environment". The most important options are:

- *JavaScript VM*: A personal Ethereum blockchain implemented in JavaScript that runs locally. It comes with 5 accounts which are preloaded with 100 ETH. It's suited for early development stages, unit testing, and all in all quick tests, as there are no transaction times.
- *Injected Web3*: As mentioned before, MetaMask injects Web3 into websites. When choosing this option, the currently active account and the chosen network in MetaMask are used for development.

Underneath, the smart contract can be chosen and deployed to the blockchain. Next to the deploy button, constructor arguments can be passed. Optionally an existing smart contract can be loaded from an address.

After a contract has been deployed or loaded, all functions and public variables will be

listed under the "Deployed Contracts" section. This will be the main way to interact with the smart contract.

6.2.3 Solidity

This subsection will explain the basics and key features of the Solidity programming language. A solidity source file has the ".sol" file extension. The language has a C++ style syntax and works very similarly to object-oriented programming and is also called contract-oriented[51]. Contracts can be viewed as classes and also work with interfaces and inheritance.
See Listing 1 for an example of a minimalistic smart contract, which demonstrates the general syntax as well as some of the features listed below.

General notice
There are a few important aspects in how certain things behave during smart contract programming and execution:

- Solidity does not implement floats at the time of writing. This is especially important for calculations using Ether. Therefore it is important to remember that Ether is always assumed as Wei by smart contract functions.
- When a function throws, all changes to the state that were made up to this point are reverted and the transaction is marked as failed.
- *undefined* and *null* does not exist in Solidity, variables rather have a default type, e.g., 0 for integers[52].

Types
The most important data types in Solidity are[52]:

- *bool*: The possible values are "true" and "false".
- *integer*: There are signed (*int*) and unsigned (*uint*) integers in Solidity. *uint* is an alias for *uint256*. The smallest size for an integer is 8 bit (e.g., *uint8*), the sizes grow by 8 up to 256 bit. The same applies to signed integers as well.
- *address*: The address variable stores 20 bytes.
- *address payable*: The address payable variable stores 20 bytes as well. Additionally it holds the members *transfer* and *send* to send Ether to that address.
- *bytes32*: Holds 32 bytes of data.

- *mapping(keyType => valueType)*: Mappings work similarly to hash tables. The key can be of any elementary type, the value can be of any type, even another mapping. An example for a frequently used mapping would be the balance variable: *mapping(address => uint256) balances;*
  Every address points to a uint256 which represents a balance.
- *bytes[]*: Dynamically-sized byte array.
- *string*: Dynamically-sized UTF-8 encoded string.

Pragma

The first line of a solidity file defines the compiler version[53]:

```
pragma solidity ^0.5.4;
```

The ^ symbol means that the code can be compiled by a compiler with the versions 0.5.4 and above, but below 0.6.0. Without the ^ symbol only the compiler version 0.5.4 can compile the source code.

Important variables and functions

Solidity features some global units, variables and functions which can be very useful, if not necessary for smart contract programming:

- *ether*: The ether unit multiplies the current value by 10^{18}, e.g., 3 ether equals 3,000,000,000,000,000,000.
- *time units*: The following time units are available: "seconds", "minutes", "hours", "days", "weeks". 1 seconds equals 1, 1 minutes equals 60 seconds or 60, 1 hours equals 60 minutes or 3600 and so on.
- *now*: An alias for *block.timestamp*, a *uint256* variable as seconds since unix epoch. The variable is set by the miner during validation so it should not be used for random number generation as the number can be varied by up to ± 15 seconds, because the timestamp of a block has to be higher than the one of the previous block.
- *msg.sender*: Of type *address payable* and contains the sender of the current message. If a function is called directly by an EOA and not by a smart contract, *msg.sender* will contain the sender of the transaction.
- *msg.value*: Of type *uint256* and contains the number of Wei sent with the current message.
- *<address payable>.transfer(uint256 value)/<address payable>.send(uint256 value)*: Both functions send "value" in Wei to the payable address. Transfer throws, send returns false on failure.

- *function ()*: A function declared without any name is also called the fallback function. This function is called, when no matching function identifier is provided. Usually this function is triggered, when a simple transaction is sent to the smart contract, that's why the fallback function can often be seen with the *payable* modifier.
- *require(bool condition, string memory message)*: The require function throws if the condition is not met and provides the custom error message.
- *keccak256(bytes memory) returns (bytes32)*: Computes the Keccak-256 hash of an input.
- *ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)*: For a given message and r, s, v values of a ECDSA signature the function returns the address associated with the public key from the signature.

Constructor
A constructor function is optional and can be used to execute code directly when the smart contract is stored on the blockchain[54].

Modifiers
A function can have multiple different modifiers, that make it behave in different ways[55]. First, there is the visibility modifier, which manages the access to functions and variables:

- *public*: The public modifier makes a function visible from inside and outside the smart contract. Adding the modifier to a variable automatically generates a getter function with the same name as the variable.
- *external*: The external modifier is only applicable for functions. It makes them only visible from outside the smart contract. To call the function from inside the smart contract it has to be called via "this.func()" instead of "func()".
- *internal*: The internal modifier makes a function or variable only visible internally. This means they can only be accessed from the contract and all derived contracts.
- *private*: The private modifier makes a function or variable only visible from the contract itself. It's important to notice that although a variable might be private, it still can be read, since all data stored on the blockchain is public.

Reading from the blockchain does not require mining or involve transaction fees. Therefore functions that do not write to storage can be marked as such via a modifier:

- *view*: If a function has a view modifier, it cannot write to storage, only read from it.

- *pure*: If a function has a pure modifier, it cannot modify storage. Additionally it cannot read state, e.g., read from variables. It's primarily used for computations.

The *payable* modifier allows a function to receive Ether. If Ether is included in a function call that doesn't have the *payable* modifier, it throws.

It's also possible to write custom modifiers. Listing 1 shows the function of a custom modifier through a commonly used *onlyOwner* modifier, which only allows the owner of a smart contract to call the function.

Events

Events are an important part of smart contract programming for two key reasons:

1. With a complex smart contract handling thousands of transactions, it can get confusing to keep track of all changes made to the state of the smart contract. Events are a good way to sort these changes into different categories and make them searchable by specific filters.

2. Some time passes until a transaction was mined, therefore the return value of a function is not returned to the sender of the transaction. Events can be used to act as a return value. A computer system or a frontend can then scan for these events and act accordingly, e.g., show updates to the user.

See Listing 1 for an example of an event.

```
1  pragma solidity 0.5.8;
2
3  contract ModifierExample{
4      // variable to store the owner of the smart contract
5      address owner;
6      // number to demonstrate the view function
7      uint256 public num;
8
9      // definition of an event
10     // the indexed keyword allows to use the parameter as a filter
11     event ownerChanged(address indexed oldOwner, address indexed newOwner);
12
13     // the constructor gets called after contract creation
14     // arguments can be passed to the constructor
15     constructor(uint256 _num) public {
16         // set owner to the sender of the transaction
17         owner = msg.sender;
18         // set the number to the passed argument
19         num = _num;
20     }
```

```
21
22      // custom modifier
23      modifier onlyOwner() {
24          // throws if sender of call is unequal to value stored in owner variable
25          require(owner == msg.sender, "sender is not owner");
26          // _; defines where the code of the function, the modifier is used on, runs
27          _;
28      }
29
30      // function to change the address of the smart contract
31      function changeOwner(address _owner) external onlyOwner {
32          // emits the ownerChanged event
33          emit ownerChanged(owner, _owner);
34          // sets the owner variable to the passed argument
35          owner = _owner;
36      }
37
38      // the function will execute the code and return the calculated number
39      // because of the view modifier no state is modified and no transaction has to be
            sent
40      function calculatedNum() public view returns(uint256) {
41          return num * 2;
42      }
43
44  }
```

Listing 1: Basic structure of a smart contract

Libraries

Libraries can be used to reduce code redundancy and save transaction fees. For the implementation of this smart contract, a so-called SafeMath library was used to detect integer over- and underflows. An example on how to use a library for a certain data type can be found in line 70 of the smart contract source code:

```
70  using SafeMath for uint256;
```

Listing 2: Using the SafeMath library

Here are some examples on how to add, subtract or multiply integers using the library.

```
1  uint256 a = 5;
2  uint256 b = 7;
3
4  // add two integers
5  // c = a + b
6  uint256 c = a.add(b)
7
8  // subtract two integers
9  // d = b - a
```

```
// would result in an integer underflow, therefore the safemath library throws
uint256 d = b.sub(a)

// multiply two integers
// e = a * b
uint256 e = a.mul(b)
```

Listing 3: Examples for SafeMath calculations

6.2.4 Source Code

```
/**
 * @title SafeMath
 * @dev Unsigned math operations with safety checks that revert on error
 * @notice https://github.com/OpenZeppelin/openzeppelin-solidity
 */
library SafeMath {
    /**
     * @dev Multiplies two unsigned integers, reverts on overflow.
     */
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
        // benefit is lost if 'b' is also tested.
        // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b);

        return c;
    }

    /**
     * @dev Integer division of two unsigned integers truncating the quotient, reverts on
          division by zero.
     */
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // Solidity only automatically asserts when dividing by 0
        require(b > 0);
        uint256 c = a / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold

        return c;
    }

    /**
     * @dev Subtracts two unsigned integers, reverts on overflow (i.e. if subtrahend is
          greater than minuend).
     */
```

```
41      function sub(uint256 a, uint256 b) internal pure returns (uint256) {
42          require(b <= a);
43          uint256 c = a - b;
44
45          return c;
46      }
47
48      /**
49       * @dev Adds two unsigned integers, reverts on overflow.
50       */
51      function add(uint256 a, uint256 b) internal pure returns (uint256) {
52          uint256 c = a + b;
53          require(c >= a);
54
55          return c;
56      }
57
58      /**
59       * @dev Divides two unsigned integers and returns the remainder (unsigned integer
             modulo),
60       * reverts when dividing by zero.
61       */
62      function mod(uint256 a, uint256 b) internal pure returns (uint256) {
63          require(b != 0);
64          return a % b;
65      }
66  }
```

Listing 4: SafeMath library

```
1   pragma solidity ^0.5.0;

    library SafeMath {
        // insert SafeMath library code from above
    }

68  contract SocketPaymentChannel {
69      // use the SafeMath library for calculations with uint256 to prevent integer over-
            and underflows
70      using SafeMath for uint256;
71
72      address public owner;
73      uint256 public pricePerSecond;
74
75      // stores the balances of all customers and the owner
76      mapping(address => uint256) public balances;
77
78      // duration after a payment channel expires in seconds
79      uint256 public expirationDuration;
80      // minimum required deposit for payment channel
81      uint256 public minDeposit;
82
```

```
83      // global payment channel variables
84      // boolean whether the payment channel is currently active
85      bool public channelActive;
86      // timestamp when payment channel was initialized
87      uint256 public creationTimeStamp;
88      // timestamp when payment channel will expire
89      uint256 public expirationDate;
90      // address of current customer
91      address public channelCustomer;
92      // value deposited into the smart contract
93      uint256 public maxValue;
94
95      // nonces to prevent replay attacks
96      mapping(address => uint256) public customerNonces;
97
98      // events
99      event InitializedPaymentChannel(address indexed customer, uint256 indexed start, uint256 indexed
            maxValue, uint256 end);
100     event ClosedPaymentChannel(address indexed sender, uint256 indexed value, bool indexed expired,
            uint256 duration);
101     event PriceChanged(uint256 indexed oldPrice, uint256 indexed newPrice);
102     event Withdrawal(address indexed sender, uint256 indexed amount);
103
104     // modifier that only allows the owner to execute a function
105     modifier onlyOwner() {
106         require(msg.sender == owner, "sender is not owner");
107         _;
108     }
109
110     constructor(uint256 _pricePerSecond, uint256 _expirationDuration, uint256 _minDeposit) public {
111         owner = msg.sender;
112         pricePerSecond = _pricePerSecond;
113         expirationDuration = _expirationDuration;
114         minDeposit = _minDeposit;
115         channelActive = false;
116     }
117
118     /// @notice function to initialize a payment channel
119     /// @return true on success, false on failure
120     function initializePaymentChannel() public payable returns (bool) {
121         // payment channel has to be inactive
122         require(!channelActive, "payment channel already active");
123         // value sent with the transaction has to be at least as much as the minimum
                required deposit
124         require(msg.value >= minDeposit, "minimum deposit value not reached");
125
126         // set global payment channel information
127         channelActive = true;
128         // set channel customer to the address of the caller of the transaction
129         channelCustomer = msg.sender;
130         // set the maximum transaction value to the deposited value
```

```
131         maxValue = msg.value;
132         // set the timestamp of the payment channel intialization
133         creationTimeStamp = now;
134         // calculate and set the expiration timestamp
135         expirationDate = now.add(expirationDuration);
136 
137         // emit the initialization event
138         // It's cheaper in gas to use msg.sender instead of loading the channelCustomer
                variable, msg.value instead of maxValue, etc.
139         emit InitializedPaymentChannel(msg.sender, now, now.add(expirationDuration), msg.value);
140         return true;
141     }
142 
143     /// @notice function to close a payment channel and settle the transaction
144     /// @dev can only be called by owner
145     /// @param _value the total value of the payment channel
146     /// @param _signature the signature of the last off-chain transaction containing the
          total value
147     /// @return true on success, false on failure
148     function closeChannel(uint256 _value, bytes memory _signature) public onlyOwner returns (bool) {
149         // save value to a temporary variable, as it is reassigned later
150         uint256 value = _value;
151         // payment channel has to be active
152         require(channelActive, "payment channel not active");
153         // call verify signature function, if it returns false, the signature is invalid
              and the function throws
154         require(verifySignature(value, _signature), "signature not valid");
155 
156         // increase nonce after payment channel is closed
157         customerNonces[channelCustomer] = customerNonces[channelCustomer].add(1);
158 
159         // if maxValue was exceeded, set value to maxValue
160         if (value > maxValue) {
161             value = maxValue;
162             // value of payment channel equals the exact deposited amount
163             // credit owner the total value
164             balances[owner] = balances[owner].add(value);
165         } else {
166             // credit owner the value from the payment channel
167             balances[owner] = balances[owner].add(value);
168             // refund the remaining value to the customer
169             balances[channelCustomer] = balances[channelCustomer].add(maxValue.sub(value));
170         }
171 
172         // emit payment channel closure event
173         emit ClosedPaymentChannel(msg.sender, value, false, now - creationTimeStamp);
174 
175         // reset channel information
176         channelActive = false;
177         channelCustomer = address(0);
178         maxValue = 0;
```

```
179         expirationDate = 0;
180         creationTimeStamp = 0;
181
182         return true;
183     }
184
185     /// @notice function to timeout a payment channel, refunds entire deposited amount
        to customer
186     /// @return true on success, false on failure
187     function timeOutChannel() public returns (bool) {
188         // payment channel has to be active
189         require(channelActive, "payment channel not active");
190         // payment channel has to be expired
191         require(now > expirationDate, "payment channel not expired yet");
192
193         // increase nonce after payment channel is closed
194         customerNonces[channelCustomer] = customerNonces[channelCustomer].add(1);
195
196         // return funds to customer if channel was not closed before channel expiration
            date
197         balances[channelCustomer] = balances[channelCustomer].add(maxValue);
198
199         // emit payment channel closure event
200         emit ClosedPaymentChannel(msg.sender, 0, true, now - creationTimeStamp);
201
202         // reset channel information
203         channelActive = false;
204         channelCustomer = address(0);
205         maxValue = 0;
206         expirationDate = 0;
207         creationTimeStamp = 0;
208
209         return true;
210     }
211
212     /// @notice helper function to validate off-chain transactions
213     /// @param _value value of the off-chain transaction
214     /// @param _signature signature / off-chain transaction
215     /// @return true if the sender of the off-chain transaction is equal to the current
        customer, else false
216     function verifySignature(uint256 _value, bytes memory _signature) public view returns (bool) {
217
218         // split signature into r,s,v values (https://programtheblockchain.com/posts
            /2018/02/17/signing-and-verifying-messages-in-ethereum/)
219         require(_signature.length == 65, "signature length is not 65 bytes");
220
221         bytes32 r;
222         bytes32 s;
223         uint8 v;
224
225         // split using inline assembly
```

```
226         assembly {
227             // first 32 bytes of message
228             r := mload(add(_signature, 32))
229             // second 32 bytes of message
230             s := mload(add(_signature, 64))
231             // first byte of the next 32 bytes
232             v := byte(0, mload(add(_signature, 96)))
233         }
234 
235         // to recover the address of the sender, the signed data has to be recreated
236         // variables that are included in the message: value, address of contract, nonce
                of customer
237         address contractAddress = address(this);
238         // hash variables
239         bytes32 message = keccak256(abi.encodePacked(_value, contractAddress, customerNonces[
                channelCustomer]));
240         // prefix message with ethereum specific prefix
241         bytes32 prefixedMessage = keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32
                ", message));
242         // ecrecover recovers the address from a signature and the signed data
243         // returns true if recovered address is equal to customer address
244         return ecrecover(prefixedMessage, v, r, s) == channelCustomer;
245     }
246 
247     /// @notice function to withdraw funds from the smart contract
248     /// @return true on success, false on failure
249     function withdraw() public returns (bool) {
250         // save balance of sender to a variable
251         uint256 balance = balances[msg.sender];
252         // best practice: set balance of sender to zero before sending the transaction,
                see reentrancy attack
253         balances[msg.sender] = 0;
254         // send entire balance to sender
255         msg.sender.transfer(balance);
256         // emit withdrawal event
257         emit Withdrawal(msg.sender, balance);
258         return true;
259     }
260 
261     /// @notice function to change the electricity price, only callable by the owner
262     function changePrice(uint256 _newPrice) public onlyOwner {
263         // emit price changed event
264         emit PriceChanged(pricePerSecond, _newPrice);
265         // update price per second
266         pricePerSecond = _newPrice;
267     }
268 }
```

Listing 5: Payment channel smart contract

List of Figures

List of Tables

List of Listings

List of Abbreviations

References

[1] ETH Gas Station. Fees. https://ethgasstation.info/, 2019. [Online; accessed 2019-05-06].

[2] blockchain.com. 7 day average chart. https://www.blockchain.com/de/charts/transactions-per-second?daysAverageString=7. [Online; accessed: 2019-05-06].

[3] Visa. Annual Report 2018. https://s1.q4cdn.com/050606653/files/doc_financials/annual/2018/Visa-2018-Annual-Report-FINAL.pdf. [Online; accessed: 2019-01-05].

[4] Shrimpton T. Rogaway P. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. *Roy B., Meier W. (eds) Fast Software Encryption. FSE 2004. Lecture Notes in Computer Science, vol 3017. Springer, Berlin, Heidelberg*, (2004).

[5] Satoshi Nakamoto. Bitcoin white paper. https://bitcoin.org/bitcoin.pdf, 2008. [Online; accessed 2019-05-06].

[6] Dr. Gavin Wood. Ethereum yellow paper. https://ethereum.github.io/yellowpaper/paper.pdf, 2019. [Online; accessed 2019-05-06].

[7] Scott Vansto Don Johnson, Alfred Menezes. The Elliptic Curve Digital Signature Algorithm (ECDSA). https://web.archive.org/web/20170921160141/http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf. [Online; accessed 2019-05-06].

[8] BitInfoCharts. Bitcoin Avg. Transaction Fee historical chart. https://bitinfocharts.com/comparison/bitcoin-transactionfees.html. [Online; accessed: 2019-06-01].

[9] Vitalik Buterin. Toward a 12-second Block Time. https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/, 2014. [Online; accessed 2019-05-06].

[10] etherscan.io. Ethereum Average Block Time Chart. https://etherscan.io/chart/blocktime, 2019. [Online; accessed 2019-05-06].

[11] Paypal. Fees for micro transactions. https://www.paypal.com/de/webapps/mpp/paypal-fees, 2019. [Online; accessed 2019-05-06].

[12] Coincap. Cryptocurrency List. https://coincap.io/. [Online; accessed: 2019-06-01].

[13] blockchain.com. Average Confirmation Time. https://www.blockchain.com/en/charts/avg-confirmation-time?timespan=all&daysAverageString=7. [Online; accessed: 2019-05-12].

[14] Serguei Popov. The Tangle. https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf, 2018. [Online; accessed 2019-05-06].

[15] The Tangle Monitor. Average Confirmation Time. https://tanglemonitor.com/, 2019. [Online; accessed 2019-05-06].

[16] Brian Pugh. Stress Testing The RaiBlocks Network: Part II. https://medium.com/@bnp117/stress-testing-the-raiblocks-network-part-ii-def83653b21f, 2018. [Online; accessed 2019-05-06].

[17] Repnode.org. Block Propagation and Confirmation Times. https://repnode.org/network/propagation-confirmation, 2019. [Online; accessed: 2019-02-22, 14:54].

[18] Colin LeMahieu. Nano: A Feeless Distributed Cryptocurrency Network. https://nano.org/en/whitepaper. [Online; accessed 2019-05-06].

[19] Etherchain. Account of DAO Hacker. https://www.etherchain.org/account/0x304a554a310c7e546dfe434669c62820b7d83490#history. [Online; accessed: 2019-06-05].

[20] accessec GmbH. Car Wallet. https://accessec.com/car-wallet/?fbclid=IwAR0bbgAra62V6FNWt_Ani5zi6v3xK4CXyNqefIPNH0ukJYZ-k3qdgUMffGQ. [Online; accessed: 2019-05-23].

[21] Bosch Presse. DLT restores trust in the internet. https://www.bosch-presse.de/pressportal/de/en/dlt-restores-trust-in-the-internet-189824.html. [Online; accessed: 2019-05-23].

[22] Innogy. Share&Charge. https://blog.slock.it/blockchain-energy-p2p-sharing-project-share-charge-going-into-live-beta-ad4e069e79d, https://shareandcharge.com/. [Online; accessed: 2019-05-23].

[23] Innogy. BlockCharge. https://www.youtube.com/watch?v=0A0LqJ9oYNg. [Online; accessed: 2019-05-23].

[24] Innogy. Share&Charge: the future of charging. https://innovationhub.innogy.com/news-event/1AStXhfTXmG6EKSSQUm6mQ/share-charge-the-future-of-charging. [Online; accessed: 2019-05-23].

[25] Share&Charge. Blogpost regarding closure. http://snc-wordpress.hidora.com/information-for-users/. [Online; accessed: 2019-05-23].

[26] Ethereum Foundation. State Channel Research. https://ethresear.ch/c/state-channels. [Online; accessed: 2019-05-23].

[27] Ethereum Foundation. Contract ABI Specification. https://solidity.readthedocs.io/en/v0.5.8/abi-spec.html. [Online; accessed: 2019-06-05].

[28] Ethereum Foundation. Design Rationale. https://github.com/ethereum/wiki/wiki/Design-Rationale. [Online; accessed: 2019-05-19].

[29] Ethereum Foundation. EIP 155. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md. [Online; accessed: 2019-05-19].

[30] Takahiro Okada. RLP C++ implementation. https://github.com/kopanitsa/web3-arduino/blob/master/src/Util.cpp. [Online; accessed: 2019-05-21].

[31] Ethereum Foundation. Packed Encoding. https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#abi-packed-mode. [Online; accessed: 2019-05-21].

[32] Ethereum Foundation. Signature prefix. https://github.com/ethereum/wiki/wiki/JSON-RPC#eth_sign. [Online; accessed: 2019-05-23].

[33] Kenneth MacKay. micro-ecc. https://github.com/kmackay/micro-ecc. [Online; accessed: 2019-01-15].

[34] Consensys. Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/recommendations/. [Online; accessed: 2019-06-05].

[35] Ethereum Foundation. Ethereum 2.0. https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/eth-2.0-phases/. [Online; accessed: 2019-05-23].

[36] Allegro MicroSystems Inc. Fully Integrated, Hall Effect-Based Linear Current Sensor. https://www.sparkfun.com/datasheets/BreakoutBoards/0712.pdf. [Online; accessed: 2019-05-11].

[37] Silicon Labs. CP210x USB to UART Bridge VCP Drivers. https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers. [Online; accessed: 2019-05-12].

[38] Links2004. WebSocket Server and Client for Arduino. https://github.com/Links2004/arduinoWebSockets. [Online; accessed: 2019-05-12].

[39] Aleksey Kravchenko. Keccak-256 C implementation. https://github.com/firefly/wallet. [Online; accessed: 2019-05-21].

[40] Ethereum Foundation. Go Ethereum. https://github.com/ethereum/go-ethereum. [Online; accessed: 2019-05-11].

[41] Ethereum Foundation. Installing Geth. https://github.com/ethereum/go-ethereum/wiki/Installing-Geth. [Online; accessed: 2019-05-12].

[42] Ethereum Foundation. Command Line Options. https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options. [Online; accessed: 2019-05-12].

[43] Ethereum Foundation. JSON RPC API. https://github.com/ethereum/wiki/wiki/JSON-RPC. [Online; accessed: 2019-05-12].

[44] Ethereum Foundation. Management APIs. https://github.com/ethereum/go-ethereum/wiki/Management-APIs. [Online; accessed: 2019-05-13].

[45] Ethereum Foundation. JavaScript Ethereum API 0.2x.x. https://github.com/ethereum/wiki/wiki/JavaScript-API. [Online; accessed: 2019-05-13].

[46] Ethereum Foundation. JavaScript Ethereum API 1.0. https://web3js.readthedocs.io/en/1.0/. [Online; accessed: 2019-05-13].

[47] JSON RPC Google Group. JSON RPC 2.0. https://www.jsonrpc.org/specification. [Online; accessed: 2019-05-19].

[48] Metamask. Wallet browser extension. https://metamask.io/. [Online; accessed: 2019-05-13].

[49] Bitcoin. BIP39 Mnemonic phrases. https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki. [Online; accessed: 2019-05-13].

[50] Bitcoin. Private key derivation. https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki. [Online; accessed: 2019-05-13].

[51] Ethereum Foundation. The Solidity Contract-Oriented Programming Language. https://solidity.readthedocs.io/en/v0.1.5/README.html. [Online; accessed: 2019-05-14].

[52] Ethereum Foundation. Types. https://solidity.readthedocs.io/en/v0.5.3/types.html. [Online; accessed: 2019-05-16].

[53] Ethereum Foundation. Layout of a Solidity Source File. https://solidity.readthedocs.io/en/v0.5.8/layout-of-source-files.html. [Online; accessed: 2019-05-14].

[54] Ethereum Foundation. Constructor. https://solidity.readthedocs.io/en/v0.5.8/contracts.html. [Online; accessed: 2019-05-15].

[55] Ethereum Foundation. Modifiers. https://solidity.readthedocs.io/en/v0.5.8/miscellaneous.html#function-visibility-specifiers. [Online; accessed: 2019-05-15].