



**Budapest University of Technology and Economics**

Faculty of Electrical Engineering and Informatics

Department of Measurement and Information Systems

# **Blockchain-based control of device access in cyber-physical systems**

BACHELOR'S THESIS

*Author*

Péter Garamvölgyi

*Advisor*

Imre Kocsis

December 7, 2017

# Contents

<b>Kivonat</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Blockchain and Cyber-Physical Systems</b>	<b>2</b>
2.1 Blockchain Platforms and Applications . . . . .	2
2.1.1 The Blockchain Data Structure . . . . .	2
2.1.2 Decentralized Consensus . . . . .	4
2.1.3 Cryptocurrencies . . . . .	6
2.1.4 Bitcoin . . . . .	7
2.1.5 Ethereum . . . . .	9
2.2 Cyber-Physical Systems . . . . .	11
2.3 Blockchain IoT/CPS Applications . . . . .	13
2.4 Standardization . . . . .	15
2.5 Main Contributions of This Thesis . . . . .	16
<b>3 Design of a Blockchain-Based Device Rental Platform</b>	<b>17</b>
3.1 Development Process . . . . .	17
3.2 Use Cases for a Device Rental Platform . . . . .	17
3.3 Glossary of Terms . . . . .	19
3.4 Functional Analysis . . . . .	19
3.5 Access Control and Smart Contract Overview . . . . .	20
3.6 Device Descriptors . . . . .	21
3.7 Device and Client Overview . . . . .	23

3.7.1	Device Control Flow . . . . .	24
3.7.2	Client Control Flow . . . . .	25
3.7.3	Combined Control Flow . . . . .	26
3.7.4	Client-Device Communication . . . . .	26
3.7.5	Challenges and Library Implementation Highlights . . . . .	29
3.8	Smart Contract-Based Key Exchange over Public Blockchain . . . . .	31
3.8.1	Key Exchange Overview . . . . .	31
3.8.2	Diffie-Hellman Key Exchange on the Blockchain . . . . .	33
3.8.3	Key Exchange Implementation . . . . .	34
<b>4</b>	<b>Smart Contracts: Towards Model-based Techniques</b>	<b>36</b>
4.1	Challenges and Pitfalls of Smart Contract Programming . . . . .	36
4.2	State Modeling for Smart Contract Programming . . . . .	37
4.3	Example: State Modeling of the Device Rental Platform . . . . .	38
4.4	Towards Automatic Code Generation . . . . .	41
4.5	Discussion of Code Generation . . . . .	45
4.6	Additional Smart Contract Functionalities . . . . .	47
4.7	Limitations of Model-Based Smart Contract Programming . . . . .	48
<b>5</b>	<b>Demonstration and Evaluation</b>	<b>50</b>
5.1	Local Testing . . . . .	50
5.1.1	Test Setup . . . . .	50
5.1.2	Results . . . . .	51
5.2	Cost Benchmarks . . . . .	51
5.3	A Step-By-Step Guide for Testing on the Public Testnet . . . . .	52
5.4	Evaluation . . . . .	58
<b>6</b>	<b>Summary and Future Work</b>	<b>59</b>
	<b>Acknowledgements</b>	<b>60</b>
	<b>Bibliography</b>	<b>64</b>
	<b>Appendix</b>	<b>65</b>
A.1	Complete Code of the Device Rental Platform Smart Contract . . . . .	65

## HALLGATÓI NYILATKOZAT

Alulírott *Garamvölgyi Péter*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2017. december 7.

---

*Garamvölgyi Péter*  
hallgató

# Kivonat

A Bitcoin és egyéb kriptovaluták sikere következtében egyre több új felhasználási területre talál a Bitcoin alapjául szolgáló blockchain technológia. Sokak szerint a blockchain a kiberfizikai rendszerek (CPS) és Internet of Things (IoT) sok kihívására megoldást nyújthat. Ebben a dolgozatban áttekintem ezen két terület főbb fejleményeit és kihívásait, majd végigvezetem az olvasót a Device Rental System tervezésén és megvalósításán.

A Device Rental System egy, a blockchain és kiberfizikai rendszerek technológiákat ötvöző platform, melynek segítségével a felhasználók időlimitált, exkluzív hozzáférést árulhatnak különféle CPS eszközökhöz az Ethereum platformon keresztül. A hozzáférés-szabályozás részeként egy blockchain-alapú Diffie-Hellman kulcscsere algoritmus tervezésére és implementációjára is kitérek. Emellett egy új, UML állapotgép diagramokon alapuló smart contract fejlesztési módszert is bemutatok, melynek segítségével a smart contract programozás számos tipikus hibája elkerülhető.

# Abstract

As a result of the recent success of Bitcoin and other cryptocurrencies, the underlying blockchain technology has been applied to numerous new application domains in various industries. Many expect this technology to be a key enabler of Cyber-Physical Systems (CPS) applications, with a special emphasis on the Internet of Things (IoT). This thesis reviews the state of the art of both blockchain and CPS technologies and discusses the design and implementation of the Device Rental System.

The Device Rental System is a platform combining blockchain and CPS technologies that enables users to sell access to their CPS devices on the Ethereum platform, with a special emphasis on the time-limited, exclusive access pattern. As part of the access control mechanism, a Diffie-Hellman-based on-chain key exchange protocol is presented. Furthermore, a novel, UML-based method of developing Ethereum smart contracts on the blockchain is discussed, with which developers can avoid many of the pitfalls and challenges of smart contract programming.

# Chapter 1

## Introduction

With the recent success of Bitcoin and other cryptocurrencies and distributed applications (*dapps*), the underlying blockchain technology has been gaining significant attention from the industry. Numerous new application areas have been investigated [1] [2], with participants including industry leaders such as Microsoft or IBM.

At the same time, the area of the Internet of Things (IoT) has also been evolving fast. Some projections suggest as many as 30 billion IoT devices by 2020, each automatically cooperating with each other to fulfill their tasks. The long list of application areas for such IoT technologies includes supply chain applications, smart home systems, and urban planning among others.

Both of the aforementioned technologies have been described as *tipping points* by the World Economic Forum [3], potentially giving rise to deep shifts in society. While it has been suggested that the blockchain technology can be a key enabler of IoT technologies, progress in the combination of these two areas has been relatively slow. Challenges posed by IoT and potentially solved by blockchains include security, device-to-device communication, and trustless cooperation using codified smart contracts.

In this thesis, one possible combination of these two areas is investigated: a platform for time-limited rental of CPS/IoT devices. The idea is that anyone can provide any service (potentially using a CPS device's sensors) relying on the blockchain for secure payments and transaction logging. I will describe the design of this system with some implementation details and give an evaluation of it. I will also reflect on some issues of blockchain-based software development and propose some solutions.

This thesis is organized as follows. In Chapter 2, an overview of blockchain and CPS technologies is given, and recent combinations of the two are discussed. Chapter 3 is a detailed account of the main components of the device rental system. Chapter 4 concentrates on smart contract programming, describing some of its pitfalls and challenges and proposing a solution based on formal modeling. In Chapter 5, the system is evaluated using results acquired from manual testing. And finally, Chapter 6 gives a summary and discusses promising directions for future research.

## Chapter 2

# Blockchain and Cyber-Physical Systems

This section gives an overview of current blockchain and CPS technologies and introduces some of the proposed combinations of the two. The open issues that this thesis addresses are also discussed.

### 2.1 Blockchain Platforms and Applications

A blockchain is a distributed record of data that is based on the consensus of a specific group and is under normal circumstances immutable. The blockchain technology has been steadily gaining popularity since its introduction in 2008 by Satoshi Nakamoto in his paper describing Bitcoin [4]. Since then, encouraged by the success of Bitcoin, numerous *altcoins* have appeared: cryptocurrencies and other blockchain-projects inspired by and based on the technology behind Bitcoin. One of the most notable alternatives to Bitcoin is Ethereum, which aims to be a platform for developing a whole range of blockchain-based applications, while also addressing many of Bitcoin's issues. The next few sections give a general overview of the blockchain and also discuss two specific platforms, Bitcoin and Ethereum.

#### 2.1.1 The Blockchain Data Structure

In its essence, a blockchain is an immutable list of data records organized in blocks. Appending a new block happens as follows.

1. Collect some records  $\{r_{n,1}, r_{n,2}, \dots, r_{n,m}\}$  to be included in the next block  $B_n$ .
2. Append the hash<sup>1</sup> value of the previous block ( $prevhash_n = hash_{n-1}$ ) and the block-specific metadata ( $metadata_n$ ), to the group of records.

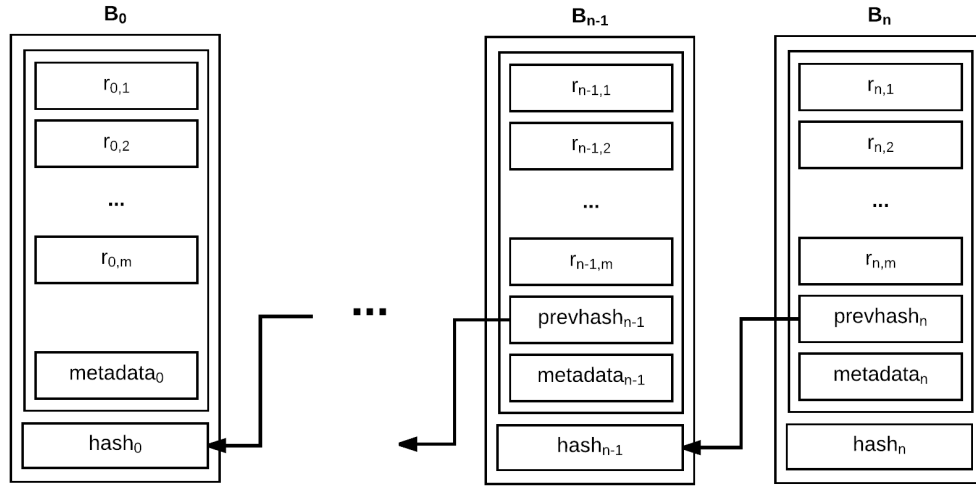
---

<sup>1</sup>In the case of Bitcoin, the SHA256 cryptographic hash function is used.



3. Compute  $hash_n$ , the hash of the group  $\{r_{n,1}, r_{n,2}, \dots, r_{n,m}, prevhash_n, metadata_n\}$ .
4. Block  $B_n$  consists of the group  $\{r_{n,1}, r_{n,2}, \dots, r_{n,m}, prevhash_n, metadata_n, hash_n\}$ .

By requiring each block to reference the previous block by its hash, this structure represents a chain of blocks. The first block ( $B_0$ ), called *genesis block*, is chosen arbitrarily and is usually hard-coded into the blockchain validation algorithm. The block-specific metadata field usually contains a timestamp and an integer value called *nonce* used for the Proof-of-Work algorithm discussed in the next section. Note that the number of records in blocks can differ, and a block can in some cases contain zero records.



**Figure 2.1:** The Blockchain Data Structure

Cryptographic hash functions have multiple properties that make them appropriate for use cases in cryptographic applications:

- First, they are practically irreversible. This means that, given a hash value  $H = h(x)$ , the only way to find the source value  $x$  is to try all the possibilities in a brute-force manner. For example, if the source value is represented by 256 bits, one would potentially need to try  $2^{256}$  different values to find  $x$ , making this reversal computationally infeasible.
- Second, cryptographic hash functions generate high entropy output. This means that no information about the input is leaked in the output. A consequence of this is that even a small change in the input, such as changing a single bit, should result in a completely different hash value.

The blockchain ensures immutability of data on multiple levels.

- First, a block is sealed by its hash. By recomputing the hash of the data in a single block and comparing it with the given hash, one can easily detect any changes to the data.

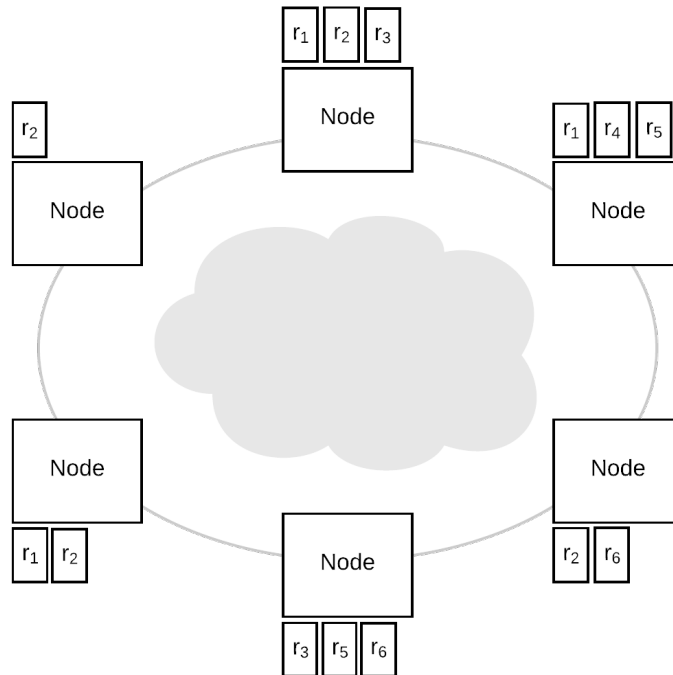
- Furthermore, new blocks contribute to the integrity of all previous blocks. As block  $B_n$  references the previous block  $B_{n-1}$  by its hash, it also indirectly references all preceding blocks  $B_{n-2}, B_{n-3}, \dots, B_0$ . Any change to any of these blocks would result in a new hash for the block and so the hash values of all subsequent blocks would also change.

For a full validation of the blockchain, we have to start from the genesis block. For each block, we first recompute the hash value using the block data and the hash of the previous block. Then, we compare the result with the hash value stored in the block. Any deviation would show that data integrity has been compromised.

### 2.1.2 Decentralized Consensus

Reaching consensus in groups is a problem in Computer Science with a large body of literature and numerous solutions and application domains. The Proof-of-Work algorithm described in this section is the solution to the subproblem of reaching group consensus in a decentralized network of nodes without the need for trust between them.

Let us consider a group of nodes in a decentralized peer-to-peer network. The group would like to maintain a common data store, of which each node has a local copy. Due to some inherent properties of data propagation and network latency, each node's local store has a potentially different subset of all records introduced into the network. Thus, the nodes need some way to reach consensus about the global state, as well as the validity of individual records. See Figure 2.2 for a visual representation of this problem.



**Figure 2.2:** The Challenge of Decentralized Consensus

In a private network where all nodes are subject to some common mechanism of authentication and authorization (A&A), and group membership is subject to explicit management by some common membership service, one can use algorithms like *Practical Byzantine Fault Tolerance* (PBFT) [5]. Previously, in public networks, one would have needed a central trusted authority for validating records and maintaining the state of the data store. Nakamoto's most important contribution is a blockchain-based algorithm called *Proof-of-Work* (PoW) for reaching decentralized consensus in a public network without the need for trust [4].

The Proof-of-Work algorithm works as follows.

1. Nodes issue records (also called *transactions*) and receive records from other nodes in the network.
2. Nodes, having collected a certain number of valid records, attempt to create a block with these records. To create a new block, nodes are required to find a hash of the records with certain properties by adjusting an extra integer value in the block called *nonce*. The difficulty of this *cryptographic puzzle* is adjusted dynamically so that the average amount of time required to solve it is constant.
3. Once a node has managed to seal a new block by finding an appropriate hash, it broadcasts the new block to the network.
4. Other nodes, upon receiving the new block, validate the records in the block and the block hash. If the block is valid, it is stored and the nodes start working on the next block.
5. For each valid block, the node that created it receives a financial reward, for example in the form of a certain amount of Bitcoin. This serves as an incentive to participate in the network and create new blocks.

This algorithm is called Proof-of-Work because a certain amount of work (i.e. time, power, CPU cycles) is needed to solve the cryptographic puzzle and produce a valid block. To compensate nodes for their work, Nakamoto introduced financial incentives. Thus, some people argue that Nakamoto's biggest contribution is not technological but game theoretical instead: By adjusting the economic incentives of all the participants in the Bitcoin network using financial rewards, it simply became more rational to play by the rules.

In financial applications such as cryptocurrencies, an important type of attack is the *double-spending attack*. In this scenario, the malicious attacker issues a transaction for buying some goods or services. Once the attacker has received the service, he issues another transaction that would send the same funds to himself instead of the service provider. Then the attacker would try to make the network accept the second transaction instead of the first one. Nakamoto's analysis has shown that, in order to succeed against the Proof-of-Work consensus, the attacker would need more than 50% of the network's full computational capacity. This kind of attack is often referred to as the *51% attack*. Thus, with the assumption that the majority of the participants would not cooperate to cheat,

and that the network is large enough to make it financially infeasible to assume control of a large number of nodes, the Proof-of-Work algorithm is considered to be secure.

The Proof-of-Work algorithm was the first example of decentralized consensus without the need for trust. However, PoW's energy requirements tend to increase over time. In [6], the authors' analysis suggests that the power used for Bitcoin transaction validation in 2014 was comparable to Ireland's energy consumption. As a consequence, since Nakamoto's original paper in 2008, there has been active research on alternative consensus methods, including *Proof-of-Stake* (PoS) [7] and *Proof-of-Elapsed-Time* (PoET) [8].

### 2.1.3 Cryptocurrencies

The need for fully digital, programmable currencies has been around for decades. Digital versions of fiat currencies all require a trusted central authority to maintain the accepted ledger of all financial transactions, i.e. banks are the single source of truth. While this solution is widely accepted and used today, it has several drawbacks, including the need for a trusted third party, scaling problems, the centralization of power, limitations in doing programmatic payments, the inability to make direct online payments across borders, and the inability to make micropayments.

Satoshi Nakamoto's original paper suggests that the need for relatively large transaction fees is a direct consequence of the reversibility of transactions [4]. As financial institutions must be willing to negotiate disputes and in some cases revert transactions, these institutions need the transaction fee to cover the cost of mediation. Moreover, the lack of non-reversible payments force merchants offering non-reversible services to collect more information about clients to avoid fraud.

According to [9], early solutions to this problem, like Wei Dai's B-money [10], achieved independent money-issuance using cryptographic puzzles, but these solutions all relied on a central trusted authority to maintain records of ownership. There have also been attempts to solve this issue by voting, but early attempts at this could not prevent *Sybil attacks*, where the outcome of the vote could be influenced by a malicious attacker. In 2008, however, Satoshi Nakamoto presented Bitcoin, whose Proof-of-Work algorithm is a sophisticated combination of earlier methods to solve the problem of decentralized consensus, while also preventing *double spending* and *Sybil attacks*, thus making Bitcoin the first truly decentralized digital currency, that is, a cryptocurrency.

The success of Bitcoin gave rise to numerous other cryptocurrencies (*altcoins*), including Bitcoin Cash (BCH), Bitcoin Gold (BTG), Dash (DASH), Ethereum (ETH), IOTA (MIOTA), Litecoin (LTC), Nxt (NXT), Ripple (XRP), and Zcash (ZEC). Most of these cryptocurrencies share Bitcoin's blockchain-based architecture but diverge from it in many important aspects. For example, Bitcoin Cash is the result of some Bitcoin miners' decision to raise the block size in order to increase Bitcoin's transaction throughput, thus creating an alternative chain, a mechanism often referred to as a *hard fork*.

While many of these cryptocurrencies have soared in value and their market caps can reach hundreds of millions of USD, there are still many controversies and issues to be solved. In the long run, there is a clear conflict of interest between cryptocurrencies and banks (or even governments), as currencies like Bitcoin practically cannot be controlled and regulated by single parties, only by group consensus. This resulted in cases like the government of China banning many cryptocurrencies.<sup>2</sup> It is also unclear whether existing regulations apply to the new market of cryptocurrencies, an issue gaining attention due to the recent boom in the number of ICOs (*Initial Coin Offering*), a new way for companies to raise money through cryptocurrencies. Many of these ICOs collected record amounts of money but turned out to be scams or failed for various reasons. Another issue is the volatility of the price of these cryptocurrencies. A recent example is Bitcoin's price soaring from around \$4000 to over \$11,000 in the last few months, but there are also examples of prices falling just as rapidly.

Cryptocurrencies also inspire numerous philosophical debates. Most notably, it is yet unclear whether Bitcoin is a *currency* (like USD or EUR) or rather a *commodity* (like gold), i.e. whether it is a *medium of exchange* or a *store of value*. Such debates gave rise to Bitcoin Cash that attempts to address Bitcoin's slow transaction times and thus aims to become a cryptocurrency that could potentially be used in everyday transactions. Meanwhile, a recent announcement by the head of South Korea's central bank stated that Bitcoin is a form of commodity,<sup>3</sup> being one of the first financial leaders to take a stance on this issue.

While there are still many issues that need to be addressed, the success of Bitcoin and its alternatives is so far unquestionable. Whether we will see a rise or fall of cryptocurrencies, the underlying blockchain technology is here to stay, finding more and more new applications.

#### 2.1.4 Bitcoin

As the title of Nakamoto's paper [4] suggests, Bitcoin aims to be a decentralized electronic cash system, a concept known as cryptocurrency today. In the decades preceding Nakamoto's work, various other papers have suggested ways to achieve this goal, but they all struggled with creating secure systems without the need for central trusted authorities. Nakamoto's introduction of the Proof-of-Work algorithm was a breakthrough that made it possible to maintain a secure, decentralized ledger of all financial transaction in a trustless public network.

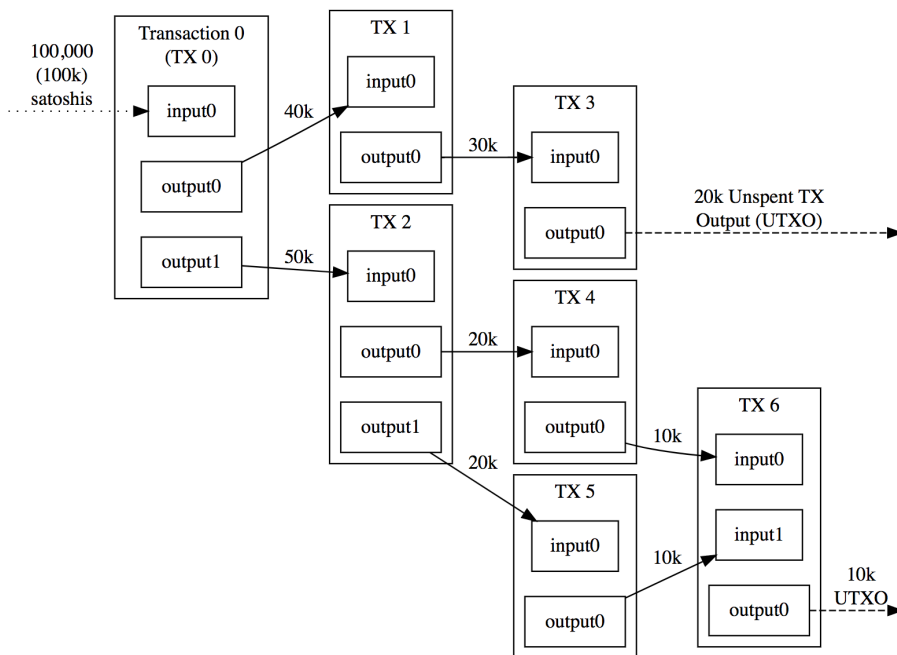
In Bitcoin's nomenclature, records are called *transactions*. The process of validating transactions by creating new blocks and solving cryptographic puzzles is called *mining*, while the nodes performing it are referred to as *miners*. The blockchain storing the transactions is often referred to as the *shared ledger*.

---

<sup>2</sup>[www.economist.com/blogs/graphicdetail/2017/10/daily-chart-0](http://www.economist.com/blogs/graphicdetail/2017/10/daily-chart-0)

<sup>3</sup>[www.coindesk.com/bitcoin-commodity-not-currency-says-south-korean-central-bank-chief/](http://www.coindesk.com/bitcoin-commodity-not-currency-says-south-korean-central-bank-chief/)

In the Bitcoin network there are no entities directly corresponding to individual *coins*; Bitcoin is represented by transactions instead. Each transaction references the outputs of one or more previous transactions and has one or more outputs. Transaction outputs correspond to certain (possibly non-integer) amounts of Bitcoin. Transaction outputs that have not been referred to by other valid transactions are called UTXOs (*Unspent Transaction Outputs*). See figure 2.3 for a visual representation of Bitcoin transactions.



**Figure 2.3:** Bitcoin Transactions (Source: bitcoin.org)

In order to use a UTXO, the sender has to prove that she is entitled to spend the associated amount of Bitcoin by solving a cryptographic puzzle. This puzzle usually consists of providing a cryptographic signature proving that the sender has the private key corresponding to the public key of the UTXO, although Bitcoin also has a low-level scripting language for defining more complex conditions. The part of the inputs not covered by the outputs is regarded as a transaction fee, the amount of which is chosen by the sender. Once the sender has composed a transaction, she broadcasts it to the network and waits for miners to validate it and include it in a block.

The cryptographic puzzle used for Bitcoin mining is finding a hash that is smaller than a certain number, i.e. its binary representation has a certain number of zeroes at the front. To solve this puzzle, miners have to try numerous possibilities by adjusting the nonce value. The Bitcoin network is configured so that the average time needed for finding a new block is 10 minutes.

Having found a new block, the miner receives a certain amount of *new* Bitcoin, as well as all the transaction fees associated with the transactions included in the new block. As miners receive the transaction fees, they are incentivized to prioritize transactions with higher fees. Moreover, mining is the only way of creating Bitcoin. It is also worth noting

that Bitcoin has a limited supply, so the amount of Bitcoin issued for each new block is dynamically decreased over time so that the goal amount is reached by the year 2140.

Note that, contrary to popular belief, Bitcoin by itself provides no guarantee of privacy and anonymity, as investigated in [11] and [12] among others.

### 2.1.5 Ethereum

Bitcoin, although it has basic scripting support, is designed to be a cryptocurrency, i.e. an instrument of value transfer. In 2013, realizing that there are many more financial and other applications to this technology, Vitalik Buterin proposed an alternative to Bitcoin called Ethereum [13]. Ethereum aims to be a platform for cryptocurrencies and decentralized applications. While conceptually very similar to Bitcoin, Ethereum diverges from its predecessor in many important aspects.

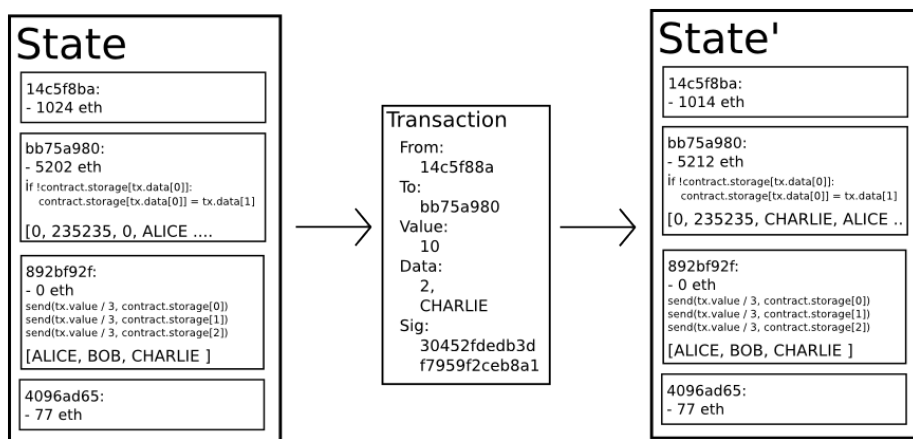
The most important changes in Ethereum are the following:

- accounts instead of UTXOs,
- Turing complete programming support instead of lightweight scripting,
- memory-heavy cryptographic puzzle for Proof-of-Work,
- 12 second block time instead of 10 minutes,
- transactions can be included in multiple blocks (*uncles*),
- uncapped currency issuance,
- gas instead of transaction fees.

In Ethereum, the main building blocks are *accounts* with given balances expressed in *ether*, Ethereum’s main currency. There are two types of accounts: *Externally Owned Accounts* (EOA) and *Smart Contracts*. Externally Owned Accounts are controlled by human users and can only be used for sending and receiving transactions. On the other hand, Smart Contracts can have code associated with them and can execute arbitrary functionality upon receiving transactions. This makes it possible to develop new cryptocurrencies in a couple of lines of code, while also facilitating the implementation of complex use cases such as financial derivatives, crowdfunding scenarios, or decentralized organizations.

To make the implementation of arbitrary Smart Contracts possible, Ethereum has a virtual machine called the *Ethereum Virtual Machine* (EVM). The Ethereum Virtual Machine works with a collection of low-level operation codes constituting a Turing complete programming environment. Ethereum also offers various high-level programming languages compiling to EVM bytecode. Execution in the Ethereum network is *fully deterministic* and *redundantly parallel*: while validating a transaction to a contract, each node executes the same code and must get exactly the same result.

The state of the Ethereum network consists of accounts and their balances, as well as the deployed contract code and the storage given to each contract to maintain its own state between transactions (see figure 2.4). Each node in the network keeps track of this state, storing it in a special data structure called *Merkle-Patricia trie*, that facilitates efficient lookup and update. Transactions can create a change in the platform’s state, e.g. a change in an account’s balance or a contract’s storage. Note that the current state of the network can be maintained incrementally or computed by replaying all the transactions from the genesis block. Similarly to Bitcoin, transactions are grouped into blocks, that are sealed using a variation of the Proof-of-Work algorithm.



**Figure 2.4:** Ethereum State Transition (Source: Ethereum White Paper [13])

Ethereum’s blockchain structure and mining process have some significant divergence from Bitcoin’s. Due to the wide use of specialized ASIC hardware, Bitcoin mining has a tendency of centralization. As a result, a handful of mining pools control the vast majority of the Bitcoin network’s computing power, which can lead to undesirable consequences [14]. Ethereum discourages the use of ASIC circuits for mining by using a different, memory-heavy cryptographic puzzle. Furthermore, Ethereum’s block time was reduced to 12 seconds instead of 10 minutes, aiming to enable everyday payment scenarios, a scaling problem Bitcoin has yet been unable to solve. Short block time reduces security and increases inequalities arising from network latency. In order to compensate for this, blocks containing previously confirmed transactions can also be referenced by new blocks as *uncles*, thus increasing the security of the whole network. Many of these ideas are adopted from the GHOST protocol described in [15].

Ethereum introduced a *gas system* to replace Bitcoin’s transaction fees. Each opcode in the EVM is assigned a gas cost that is proportional to the resources it requires. For instance, storage operations and complex cryptographic functions require more work to execute than simple arithmetic, and this is also mirrored in their corresponding gas costs. When sending a transaction to a contract, the sender has to specify the maximum amount of gas she is willing to spend, as well as the gas/ether exchange rate. Execution of contract



code requires a certain amount of gas, directly arising from the gas cost of the individual opcodes executed. If the provided gas is insufficient, the transaction is regarded as valid, but all the state changes are reverted. In this case, the ether corresponding to the gas limit provided will be transferred to the miner to compensate her for the cost of execution. Otherwise, the ether amount corresponding to the gas amount used is transferred to the miner, while the remainder is transferred back to the sender of the transaction. This system discourages wasting the Ethereum network’s resources, while also preventing attacks arising from Ethereum’s Turing completeness (forcing miners into infinite loops).

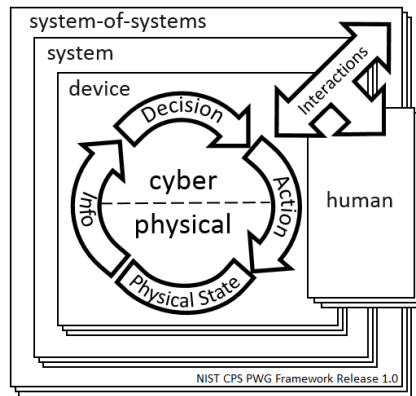
A more detailed description of Ethereum can be found in the Ethereum Yellow Paper [16].

## 2.2 Cyber-Physical Systems

The US National Science Foundation defines cyber-physical systems (CPS) as follows:

*Cyber-physical systems (CPS) are engineered systems that are built from, and depend upon, the seamless integration of computational algorithms and physical components. [...] CPS technology will transform the way people interact with engineered systems – just as the Internet has transformed the way people interact with information [17].*

There is no generally accepted consensus about the relationship of the concepts of CPS and the *Internet of Things* (IoT). Some claim that IoT is a set of interconnected CPSs. In this thesis, I am going to use these two terms interchangeably, favoring the term CPS.<sup>4</sup>



**Figure 2.5:** CPS Conceptual Model (Source: NIST CPS PWG [2])

With the increasingly ubiquitous use of computers and microcontrollers in many industries, we are experiencing an explosion in the number of devices. McKinsey’s analysts predict that the number of such devices will grow to 30 billion by 2020 from today’s 10 billion.<sup>5</sup>

<sup>4</sup>See Guihan Chen’s presentation for a more detailed explanation of the relationship of these two terms at [www.microsoft.com/en-us/research/wp-content/uploads/2010/07/Guihai-Chen\\_Oct19.pdf](http://www.microsoft.com/en-us/research/wp-content/uploads/2010/07/Guihai-Chen_Oct19.pdf)

<sup>5</sup>[www.mckinsey.com/industries/high-tech/our-insights/the-internet-of-things-sizing-up-the-opportunity](http://www.mckinsey.com/industries/high-tech/our-insights/the-internet-of-things-sizing-up-the-opportunity)

As these devices collect data (*sensors*) and interact with their environment (*actuators*), the need for efficient data handling and device management has become more urging than ever.

Along with the growing number of such devices and systems, another trend is the growing need for direct device-to-device communication. The traditional model is that devices communicate with a central computing unit (e.g. in the cloud) that does the actual data processing and device orchestration. The coordination needed for this setup require large bandwidths and can quickly deplete the energy resources of the devices [18]. With the exponential growth of the number of devices and the need for complex, local use cases, this model has been becoming unsustainable. Thus, more and more manufacturers are experimenting with direct device-to-device workflows, eliminating the need for central device management.

Typical use cases for device-to-device workflows include:

- **Home applications:** A refrigerator automatically reordering the necessary food items based on the owner’s preferences. An energy management device detecting unnecessary power consumption and ordering various home appliances (e.g. television) to turn off.
- **Industrial applications:** Possible applications include smart metering, decentralized device management, and automatic device cooperation between various industrial entities.
- **Supply chain applications:** With smart devices following the route of products from the manufacturer to the consumer, it is becoming possible to track the whole shipping process. This has numerous advantages, including fraud detection and transparency.

As of today, the area of cyber-physical systems is full of challenges. Below are some of the issues most relevant to our discussion:

- **Scalability:** Due to the steep growth of the number of connected devices, traditional computing models have been becoming less and less profitable due to high maintenance and communication costs. A decentralized approach is needed.
- **Security:** As our infrastructures depend more and more on CPS devices, there is also a growing number of ways for malicious attackers to exploit them. Progress in the security of CPS systems, however, has not kept up with the spread of such systems, making many exploits possible. A recent example receiving a lot of publicity is hackers remotely controlling components of a moving vehicle.<sup>6</sup> As a consequence, many leaders in the industry have started to emphasize the importance of this aspect of CPS systems, making it a top priority.

---

<sup>6</sup>[www.theguardian.com/technology/2016/sep/20/tesla-model-s-chinese-hack-remote-control-brakes](http://www.theguardian.com/technology/2016/sep/20/tesla-model-s-chinese-hack-remote-control-brakes)

- **Connectivity:** Today’s centralized infrastructures require connectivity between the devices and the cloud (e.g. using the AMQP or MQTT protocols in Microsoft’s Azure IoT Hub solution). However, as many of the CPS devices are on-field devices with little or no network connectivity, data collection and device management has become a serious challenge. A common solution is building mesh networks of devices with designated centers serving as gateways to the Internet and thus to the central management unit in the cloud.
- **Cost-efficiency:** Even if devices can connect to the Internet directly, it might not be worth it. Establishing and maintaining connection requires resources (e.g. computing, battery power). CPS devices, often being simple microcontrollers, might not be able to afford to be online all the time. Emerging network technologies, such as 5G, might address this issue [19].
- **Inter-device cooperation:** As direct device-to-device communication is becoming a more and more serious expectation, we need a way to securely and reliably exchange information and value between devices. Ideally, devices should not need to trust each other. For example, when a washing machine is reordering detergent, it should not need to rely on trust, but instead needs a mechanism that can make sure that if it transfers money, it will actually receive the ordered goods.
- **Micropayments:** To make independent devices able to cooperate, we need to provide a way of low-overhead value transfer between these devices. Today’s methods either lack the possibility of programmatic payments or have significant overhead (i.e. transaction fees) that make them unsuitable for CPS applications.

## 2.3 Blockchain IoT/CPS Applications

There have been various attempts to combine blockchain and IoT/CPS technologies. Some view the blockchain as a key enabler of IoT applications for the following reasons:

- **Digital payments:** Cryptocurrencies are the first means of truly digital payment. As such, they have first-class support for programmatic payments, which is necessary for the vision of IoT devices directly communicating and trading with each other.
- **Security:** IoT security has been an issue often discussed but rarely addressed appropriately. With recent examples of exploits, it has become clear that for the vision of billions of devices communicating directly with each other to become reality, security is of paramount importance. Many believe that the security and immutability of blockchains would form a secure basis for large-scale IoT projects.
- **Decentralization:** As the execution of a deployed smart contract and the validation of transactions do not depend on any central authority but only on the network itself, blockchain technologies might provide a more robust foundation for IoT solutions. In theory, the devices could keep functioning even after the maintainer has discontinued

support for various reasons. This means that blockchain technologies could help IoT developers eliminate *Single Points of Failure* (SPoF).

- **Smart Contracts:** The deterministic execution of smart contracts would allow devices to communicate with each other and offer services without the need to trust one another. Devices could provide various services such as data or computational capacity, with well-defined conditions enforced by the deterministic, immutable contract code.

In the following few paragraphs, I present some of the attempts that have been made to address these current issues of IoT using blockchain-based solutions.

Filament is an attempt to provide a whole technology stack to make secure, autonomous, decentralized IoT applications possible. Filament utilizes Telehash to create proximity-based mesh networks of devices with multi-channel communication between network nodes. A technology called Blockname is used for device discovery, smart contracts for well-defined interactions between devices, and the Blocklet protocol for microtransactions. Filament has also released a standalone device called Filament Tap, an inexpensive, surface-mount device, to facilitate building IoT systems. Use cases for Filament’s architecture include remote monitoring, smart metering, and fleet management. [20]

ADEPT, IBM’s enterprise solution for blockchain IoT, organizes devices into three categories and assigns responsibilities based on device capabilities. For example, only stronger devices would act as full nodes in the blockchain network. IBM has built a Proof-of-Concept (PoC) project based on Telehash for building device networks, BitTorrent for file sharing between devices, and Ethereum for blockchain. In this PoC they implemented a smart washing machine scenario, where the washing machine detects when it starts to run out of detergent and automatically orders more without any human intervention. [21]

IOTA is an alternative for current blockchain technologies addressing the problem of micropayments by creating a *blockchain without the chain*. IOTA claims that the requirement that miners solve computationally hard cryptographic puzzles as part of Bitcoin’s Proof-of-Work algorithm might be impractical for small IoT devices, and results in growing transaction fees. Instead, IOTA proposes a DAG of transactions they call the *tangle*. Each participant wishing to submit a transaction into the network must reference and validate two previous transactions. As every device participates in transaction validation, there is no need for transaction fees and thus micropayments become possible. The creators of IOTA claim this setup is secure against various types of double-spending attacks. However, the empirical evidence has not yet been as strong as for blockchain systems that gain confidence from Bitcoin’s almost decade-long success. [22]

Device identity is a crucial requirement for security and dynamic device discovery. Chroni-  
cled has been building an open registry and corresponding helper tools and SDKs to maintain strong device identities for NFC/BLE chips. This, they claim, makes it easy to discover counterfeits, while also enabling end users to retrieve rich information about the product they purchased. They store unique, immutable device identities on the Ethereum

blockchain, while product meta-information is stored on cloud platforms maintained by producers. Their proof-of-concept is a sneaker authentication system, where custom-made sneakers each have a unique chip inside them and a corresponding unique identity on the blockchain. [23]

In [24] the authors propose a peer-to-peer, blockchain-based platform for on-demand access to manufacturing resources called *Blockchain Platform for Industrial Internet of Things* (BPIIoT). In their system, manufacturing is enabled by distributed applications running on the blockchain, while cloud access is also integrated. Uses cases for BPIIoT include on-demand manufacturing, smart diagnostics and maintenance, supplier identity tracking, and inventory.

In [25] the authors propose to use a blockchain-based infrastructure for secure machine-to-machine (M2M) communication. Their design uses a public blockchain for immutable device identities and a private blockchain for tamper-proof device messaging. The paper also discusses a case study of using their system for cotton spinning production.

Apart from the examples above, various companies have been integrating blockchain into specific IoT applications. Microsoft's Project Manifest aims to implement blockchain-based product tracking using IoT devices.<sup>7</sup> Chain of Things has been contributing to the IoT revolution by analyzing case studies and developing hardware IoT solutions.<sup>8</sup> Slock.it is working on connecting blockchain to the physical world by making anything rentable; their Proof-of-Concept is a smart lock which makes it easy to rent out apartments without any interaction needed by the owner.<sup>9</sup> Deloitte has been investigating use cases for blockchain technologies in the telecommunications sector [1].

For more information, [26] is an excellent overview of the state of the art of blockchain and IoT systems.

## 2.4 Standardization

Multiple organizations have been working on supporting the progress of CPS and blockchain technologies by working on the corresponding standards.

- The US National Institute of Standards and Technology's (NIST) CPS Public Working Group has been working on a framework, including reference architectures and a common taxonomy, aiming to facilitate interoperability between CPS systems [2]. The CPS Public Working Group has identified numerous future use cases, including *smart transportation*, *robots for manufacturing and health care*, and *smart grid solutions for energy delivery*.

---

<sup>7</sup>[azure.microsoft.com/en-us/blog/can-blockchain-secure-supply-chains-improve-operations-and-solve-humanitarian-issues](https://azure.microsoft.com/en-us/blog/can-blockchain-secure-supply-chains-improve-operations-and-solve-humanitarian-issues)

<sup>8</sup>[www.chainofthings.com](http://www.chainofthings.com)

<sup>9</sup>[slock.it](http://slock.it)

- Standards Australia has defined a *Roadmap for Blockchain Standards*, identifying technical issues, use cases, and standards development activities to be undertaken by the Technical Committee ISO/TC 307 [27]. Their surveys suggest that many government services expect to see the use of blockchain technologies in a wide range of areas, including *land transfers and property title registrations, personal identification and passport documentation, management of health records, and vehicle registrations*.
- The main goals of the International Telecommunication Union’s (ITU) Focus Group on Application of Distributed Ledger Technology (FG DLT) are to identify distributed ledger-based applications and services, to provide support for the implementation of such services, and to assist further standardization work [28]. Another group, ITU’s Focus Group on Data Processing and Management (FG DPM), has been working on supporting data management in smart cities using blockchain and IoT [29].
- IEEE has also been working on blockchain-related standards, including *2418 - Standard for the Framework of Blockchain Use in Internet of Things (IoT)* [30].

The main goals of these standards are to provide clear definitions of capabilities and requirements and to enable interoperability between existing solutions. Such standards will provide a framework for fast, secure, and transparent development of blockchain and CPS applications.

## 2.5 Main Contributions of This Thesis

As the previous sections show, in the past few years, and especially in the last year, significant work has been done in the area of applying blockchain technologies to CPS/IoT scenarios. However, there still are many open issues that need to be addressed, including:

1. a standard, secure method for performing micropayments,
2. standards for device-to-device and human-to-device communication,
3. standard ways to offer services through smart contracts,
4. provably secure and bug-free implementation of such contracts.

To help bridge some of these gaps, the work presented in this thesis addresses the third issue by implementing an example project for renting out CPS devices, where services and device access are offered through smart contracts. Having experienced the various pitfalls of the new paradigm of smart contract programming, I also address the fourth issue by proposing a model-driven development approach where smart contract behavior is defined using standard UML state diagrams and contract code is generated directly from these models.

## Chapter 3

# Design of a Blockchain-Based Device Rental Platform

This chapter describes the design of a platform for renting out services and access to CPS devices using the blockchain ("Device Rental Platform"). Descriptions of the main components, their interactions, the design decisions made, and some implementation details are given. All the components presented here can be found in the project's repository.<sup>1</sup>

### 3.1 Development Process

The development process consisted of four parts: defining requirements, design, implementation, and verification, with multiple iterations on the design and implementation steps. Figure 3.1 is a graphical representation of the development process, with the corresponding sections of this document for each step.

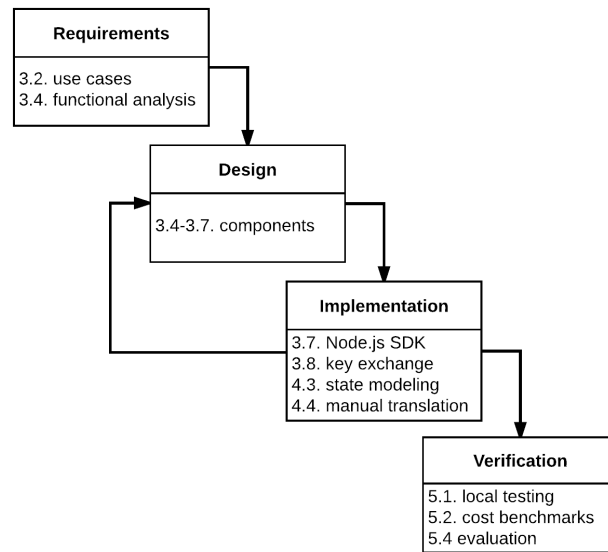
### 3.2 Use Cases for a Device Rental Platform

First, let us regard some typical use cases for the Device Rental Platform:

1. Alice has a Raspberry Pi microcontroller equipped with a temperature sensor in her home in Berlin. Realizing her device has unused capacity and her data can be valuable to others, Alice decides to offer time-limited access to her device's real-time temperature readings. Bob would like to collect various temperature readings all across Central Europe for his research project. To achieve this, Bob executes a geolocation-query filtering for all devices equipped with temperature sensors in the area of Central Europe. Bob finds Alice's device (and many others), buys access to it for a certain period of time, and collects the data he needs. Alice receives an amount of money for the services offered by her device.

---

<sup>1</sup>[github.com/Thegaram/device-rental-platform](https://github.com/Thegaram/device-rental-platform)



**Figure 3.1:** Development Process

2. Carol would like to offer time-limited, exclusive, real-time access to a footage of the bird's nest next to her window recorded by her web camera. Dave, a bird enthusiast, buys 10 minutes of access to the video stream every day.
3. Frank is throwing a party. He would like to democratize choosing music but realizes this could become chaotic rather quickly. Instead, he offers a service where anyone can play a song for a small amount of money. Song requests are queued and played one after another.
4. Energy Corp. has been collecting various metrics of its infrastructure using a large collection of CPS devices. Realizing this could be of huge value to others, Energy Corp. decides to start selling access to this data in a real-time fashion.

As the above examples suggest, the aim of this project is to implement a flexible platform capable of accommodating a wide range of use cases. This means there are no constraints on the device types, the services offered, or the actual access method to these services. Thus, the platform's goal is to take care of most of the book-keeping (transaction tracking, device discovery) and access control workflows (require access, receive payment, grant access), while leaving the details of the actual services to the people offering them. However, to make implementation easier, a simple SDK is also offered, supporting some common use cases.

Furthermore, note that the clients purchasing the services in the above examples need not be humans at all. Having a comprehensive framework for offering, discovering, and purchasing a wide array of services offered by CPS devices will contribute to the vision of direct device-to-device interaction.



### 3.3 Glossary of Terms

To prevent confusion, a brief glossary of the terms used in this chapter is given.

- **Device:** can refer to the CPS device to be rented out, or the code running on it.
- **Owner:** the entity responsible for registering and configuring the device.
- **Client:** can refer to the entity who wants to rent a given device, responsible for initiating a rental request and following the system’s protocol to gain access to the device, or the code controlling this process.
- **Service:** the service offered by the device through rental; can be direct device access, access to specific data, execution of certain actions, etc.

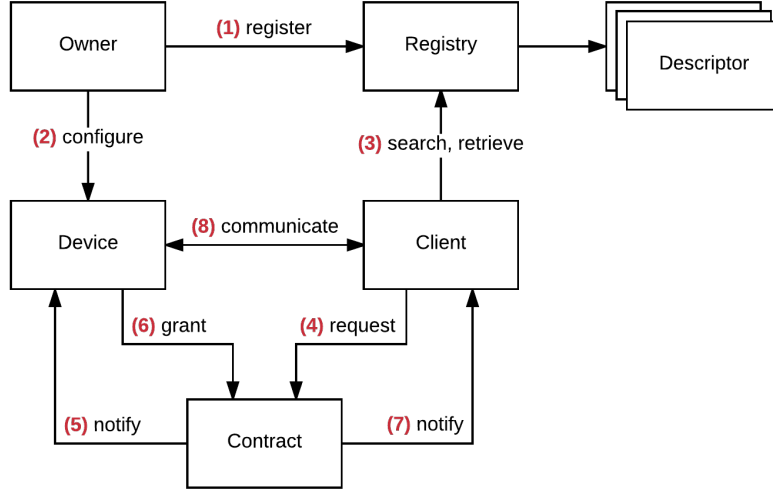
### 3.4 Functional Analysis

The Device Rental System has the following high-level requirements:

1. **Ease-of-use:** Renting out a device should be as easy and flexible as possible, requiring only as much configuration and coding as is absolutely necessary.
2. **Device discovery:** People who want to use such services should be able to search and query the available services on the platform according to various criteria, e.g. device capabilities, geological location, etc.
3. **Flexibility:** The system should have no dependency on specific device types or service patterns.
4. **Security:** The system’s design should make it practically impossible to abuse either the platform or certain devices through the platform.
5. **Robustness:** The system should be robust enough to handle failures and prevent any loss of value.

Figure 3.2 shows the high-level architectural model of the system. This model has six components:

1. The **Owner** is the person offering a service by renting out her CPS device. She is responsible for configuring the *Device* and registering it in the *Device Registry* (see below) to enable device discovery and capability queries.
2. The **Device Registry** is responsible for storing *Device Descriptors* describing device capabilities and services. It supports CRUD operations for 1) registering, updating, and deleting devices (*Owner*), and 2) querying devices and services according to various criteria (*Client*).



**Figure 3.2:** Device Rental Platform Components

3. **Device Descriptors** are data packets with a standardized format that describe various aspects of devices and their services offered, e.g. sensor types and description, geological location, service description, the price of the service, access method, etc. *Device Descriptors* enable programmatic device discovery and queries.
4. The **Device** is the actual CPS device rented out, offering the given service. It is configured by the *Owner* and communicates with both the *Client* and the *Contract* (see below).
5. The **Client** is the component requiring access to and using the device or service. It can find devices by querying the *Device Registry*. It communicates with both the *Device* and the *Contract* (see below).
6. The **Contract** is a smart contract controlling access to the device, receiving payments and keeping track of every transaction for transparency. It is accessed by both the *Device* and the *Client*.

Note that the crucial part of this architecture is the smart contract, orchestrating device access in a secure and predictable manner.

The next sections give an overview of most these components.

### 3.5 Access Control and Smart Contract Overview

This section gives a brief overview of the system's smart contract component, highlighting its most important aspects. A detailed account is given in Chapter 4.

Use cases for the Device Rental Platform vary along three dimensions:

1. **Access type:** one can offer *data* (e.g. the temperature sensor use case), *access* (e.g. renting out a server), or *action* (e.g. the music player use case).

2. **Exclusivity:** one can offer *exclusive* access, i.e. limit access to one client at a time (e.g. exclusive access to camera footage), or *shared* access, i.e. let many clients access the service simultaneously (e.g. letting many people access temperature readings at the same time). Note that shared access might require some additional access control, e.g. queueing.
3. **Time:** one can offer *single* access (e.g. for reading a single piece of data), or *time-limited* access, i.e. for accessing a service for a certain period of time.

Categorized according to these dimensions, renting out one’s web camera footage would be of type (*exclusive, time-limited, data*); renting out a server would be of type (*exclusive, time-limited, access*), while the music player example would be of type (*shared, single*,<sup>2</sup> *action*).

The focus of this thesis is the exclusive, time-limited access pattern, as this is common for most use cases, and is also relatively complex. For such an access pattern, the contract must support the following operations:

- **Request Access:** The client must be able to request access to the device by sending a certain amount of money to the contract. The amount sent is proportional to the length of the time period the client intends to buy access for.
- **Access Started:** The device must be able to signal once it has allowed access for a particular client.
- **Access Finished:** The device must also be able to signal once it has stopped allowing access so that others can buy access again.
- **Access Failed:** The device must be able to signal execution failure so that the client can reclaim her money.
- **Cancel:** The client must be able to cancel requests and reclaim her money before execution starts or after an error has occurred.

These operations are implemented as Ethereum transactions to the smart contract with the names **request**, **access\_started**, **access\_finished**, **access\_failed**, and **cancel**, correspondingly.

In addition, the contract must track all active, waiting, and failed requests, and make money available as a refund for the latter. The owner of the contract should be able to withdraw money for any number of successful requests.

## 3.6 Device Descriptors

In order to let users search and query devices, a standardized way of describing device capabilities and services is needed. In the Device Rental System, a simple JSON-based

---

<sup>2</sup> *single* because sending a song request is a single, one-time request

device descriptor scheme is proposed, that can easily be read and composed by humans, but also enables efficient processing by code.

Let us regard the descriptor in Listing 3.1.

```
0 {
1   "description": "time-limited access to temperature readings",
2   "weiPerSecond": 1000,
3   "contractAddress": "0xb1df3bd91ab7ff0426943e46bdc9ef52c150b69e",
4
5   "type": "data",
6
7   "data": {
8     "format": "json",
9     "template": {
10      "requestor": "var:string",
11      "tempdata": "var:float",
12      "requestTime": "var:date",
13      "readingTime": "var:date",
14      "format": "var:string"
15    }
16  },
17
18  "access": {
19    "method": "https/rest",
20    "auth": "https basic auth",
21    "parameters": {
22      "accessMode": "const:https/rest",
23      "url": "var:string",
24      "username": "var:string",
25      "accessEndTime": "var:date"
26    }
27  }
28 }
```

**Listing 3.1:** Temperature Service Descriptor (JSON)

The above descriptor contains the following fields:

- **description:** a human-readable description of the service offered, to be read by the end users.
- **weiPerSecond:** the service price expressed in wei/s.<sup>3</sup> Note that the authoritative source of the service price is the contract itself, the value in the descriptor is present only for informative purposes.
- **contractAddress:** the Ethereum address of the contract through which the service can be rented.
- **type:** the access type as described in Section 3.5. Possible values: data, access, action.
- **data:** a description of the data format for access type = data. In this example, the data packets are in JSON formats, containing five fields; temperature data is a float value in the field tempdata.

---

<sup>3</sup>10<sup>18</sup> wei = 1 ether

- **access**: a description of the access method to the device. In this example, temperature readings can be read through a REST endpoint over HTTPS with Basic Auth authentication. The parameters field describes the connection parameters sent over with the request approval; the client is supposed to use these parameters when connecting to the device.

Let us regard another example in Listing 3.2.

```

0 {
1   "description": "play a song",
2   "weiPerSecond": 1000,
3   "contractAddress": "0xb1df3bd91ab7ff0426943e46bdc9ef52c150b69e",
4
5   "type": "action",
6
7   "access": {
8     "method": "https/grpc",
9     "auth": "https basic auth as metadata (username-password)",
10    "parameters": {
11      "accessMode": "const:https/grpc",
12      "url": "var:string",
13      "username": "var:string",
14      "accessEndTime": "var:date"
15    },
16    "requestParameters": {
17      "id": "var:string"
18    }
19  }
20 }
```

**Listing 3.2:** Jukebox Service Descriptor (JSON)

The above descriptor describes a jukebox service, where one can request a song to be played for a given price. The device can be accessed using GRPC over HTTPS with the given connection parameters. The request should contain a string `id` parameter, pointing to a song in a given streaming service.

To describe device capabilities, I propose using a variation of the standardized *Semantic Sensor Network Ontology* (SSN) [31][32]. Capabilities to be described include sensor types and capabilities, availability, and accuracy. Other *geospatial ontologies* are to be used for describing the device’s geolocation.

### 3.7 Device and Client Overview

Once the on-chain access control protocol has been established, what is left is to define how the client and device will interact with the contract and each other. In this section, an overview of the relevant parts of the client and device libraries is presented.

### 3.7.1 Device Control Flow

The control flow for the device code is as follows:

1. Receive new request.
2. Generate private and public keys for key exchange (see Section 3.8).
3. Establish shared secret.
4. Enable access locally for the offered service (e.g. through a REST endpoint).
5. Start access on contract by sending an **access\_started** transaction to the smart contract with the relevant metadata included. This metadata specifies connection method to the device and helps to establish a shared key (see Section 3.8).
6. Wait for the specified period of time.
7. Stop access locally on service.
8. Finish access on the contract by sending an **access\_finished** transaction.

Note that the service is started locally before granting access on the contract. The reason for this is to make sure that the service is enabled by the time the client tries to access it; otherwise, there would be a *race condition*, where it would be possible that the client has already been granted access but still cannot use the service.

Also note that if there has been an error on the device that makes it unable to serve the request, it must send an **access\_failed** transaction to the smart contract. This makes it possible for the sender of the original request to withdraw her funds and also frees up the contract.

Below is a basic Node.js implementation of this logic.

```
0  async function handleRequest(request) {
1    // generate keys for key exchange
2    const dh = new DH(prime);
3    const pubkey = dh.getPublicKey();
4
5    // establish secret
6    const secret = dh.computeSecret(request.pubkey);
7
8    // start access on service and contract
9    await service.startAccess(request.requestId, secret);
10   const keepAliveSeconds = await contract.getAllowedExecutionTimeSeconds(request.requestId);
11
12   const metadata = {
13     username: request.requestId.toString(),
14     accessEndTime: Utils.timeSecondsLater(keepAliveSeconds).toISOString(),
15     accessMode: Utils.AccessMode.REST,
16     url: `https://${serverHost}:${serverPort}/${endpoint}`
17   };
18
19   await contract.accessStarted(request.requestId, pubkey, JSON.stringify(metadata), argv.gas);
20 }
```

```

21 // wait and then stop access
22 await Utils.sleep(keepAliveSeconds * 1000);
23 await service.stopAccess(request.requestId);
24 await contract.accessFinished(request.requestId, argv.gas);
25 };

```

**Listing 3.3:** Device-Side Request Handling (Node.js)

### 3.7.2 Client Control Flow

The control flow for the client code is as follows:

1. Read device descriptor. The descriptor contains the address of the contract corresponding to the target device and the service price.
2. Calculate the price of renting the service for the desired period of time.
3. Generate the private and public keys for the key exchange (see Section 3.8).
4. Request access from contract by sending a **request** transaction containing the necessary funds and some metadata.
5. Wait for approval. The metadata sent along contains the connection method details and the device's public key.
6. Establish shared secret.
7. Connect to the device using the access specifiers sent along with the approval.

Below is a basic Node.js implementation of this logic.

```

0 (async () => {
1   // read device descriptor
2   const descriptorFileContent = fs.readFileSync(argv.descriptorPath);
3   const descriptor = JSON.parse(descriptorFileContent);
4
5   // retrieve price
6   const pricePerSecond = await contract.weiPerSecond();
7   const paymentValue = argv.accessTimeSeconds * pricePerSecond;
8
9   // generate keys for key exchange
10  const dh = new DH(prime);
11  const pubkey = dh.getPublicKey();
12
13  // request access from contract and wait for approval
14  const requestId = await contract.request(pubkey, paymentValue, argv.gas);
15  const approval = await contract.waitForApproval();
16
17  // establish secret
18  const secret = dh.computeSecret(approval.pubkey);
19
20  // perform access
21  await performAccess(approval, certificate, secret);
22 })();
23

```

```

24 async function performAccess(approval, certificate, secret) {
25   // ...
26
27   if (approval.data.accessMode === Utils.AccessMode.REST) {
28     const response = await RESTClient.request(approval.data, certificate, secret);
29     console.log(response);
30   }
31
32   // ...
33 }

```

**Listing 3.4:** Client-Side Access Request (Node.js)

### 3.7.3 Combined Control Flow

Figure 3.3 shows an example execution of the device (left) and client (right) code on a local test environment. Note that the shared secret (marked with red) is derived dynamically and without any direct connection between the two parties. In this example, the temperature reading acquired through a REST endpoint call from the client to the device is marked with green.

<pre> info: checking certificate... info: waiting for request... info: new request: 3! info: generating keys... info: establishing shared secret... debug: 3e08fe6cdfe6b40499c81bf3f46a283b info: starting access on service... info: starting access on contract for 3... info: allowing access for 5 seconds... info: stopping access on service... info: finishing access on contract...  info: waiting for request... info: new request: 4! info: generating keys... info: establishing shared secret... debug: [679a928ed415212979852383d22fbf35] info: starting access on service... info: starting access on contract for 4... info: allowing access for 5 seconds... info: stopping access on service... info: finishing access on contract...  info: waiting for request... </pre>	<pre> info: verifying service price on contract... info: price for 5 seconds of access: 5000 wei info: retrieving server certificate... info: generating keys... info: requesting access from contract... info: request accepted! requestId = 4 info: waiting for approval... debug: requestId=4, pubkey=54fd7cdd58eae43b3e64580aca53ff f11e5cb6a33be4dfd60ef9ad797fa8a896c600f8eb68c904e66e0e2c5f6 a8666e71503d1bb57c216855c3a5e75db9996520f928cae33426b91df10 17b34ddd8c98dfb8b2b3397dbb411b600602951ecd55b0dc376711a58ca 72a540e68afb7aadcc1ad9c3bff2cb36c26cfa28b98e6e980e58abe4c02 8d95c222ebf92764435108dcf98e78a90775e437b8f11640fa4181e60b9 c1aa8b5c00706d4a1a50abfcd51556161467c5943422008d521da9c6b40 e7af2ec1e878300bc1aac6895df8410b80f7a1fa194be79fba82cce011b 5fd6c1c48bd39f05e7e0f0c91990f3eaaadf207989d061ea491412f3ad2b 6fdc0f7e3, username=4, accessEndTime=2017-12-06T11:17:05.18 2Z, accessMode=https/rest, url=https://localhost:8000/tempe rature info: establishing shared secret... debug: [679a928ed415212979852383d22fbf35] info: performing access through https/rest... debug: requestor=4, data=16, requestTime=2017-12-06T11:14: 21.229Z, readingTime=2017-12-06T11:14:18.689Z, format=Celsi us info: access terminated! </pre>
---	---

**Figure 3.3:** Client-Device Execution

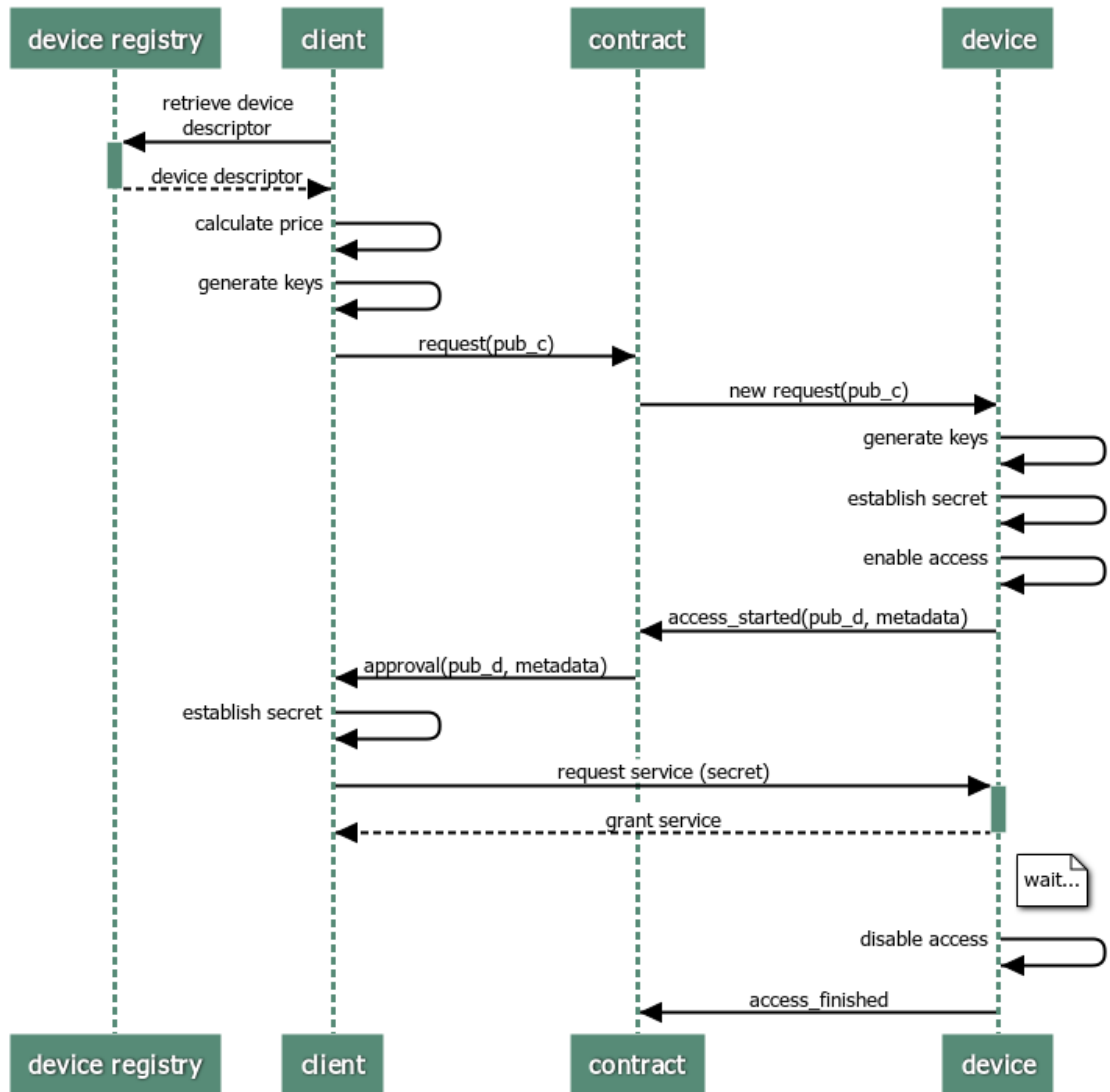
The combined client-device flow is shown on figure 3.4.

### 3.7.4 Client-Device Communication

In the previous section, the main code flow of both client and device code was presented. This section discusses the actual client-device communication process. It is assumed that the client has already been granted access to the device through the smart contract.

The appropriate way of offering the actual service varies widely by use case. For providing direct device access, one could use SSH. For providing data, one can rely on various HTTP-





**Figure 3.4:** Client-Device Code Flow

based communication protocols. For providing actions, one might take parameters through HTTP, or even not have any direct client-device communication at all.

Although it is virtually impossible to accommodate all possible use cases, the Device Rental Platform provides some example implementations:

- **Access through SSH:** If the owner would like to provide direct access to her device (like renting out a virtual server), she can choose the SSH access method. In the example implementation, the device creates a clean Docker container for every request with an SSH server running inside it. Upon approving client access, access mode SSH, the username, hostname, and port are sent along. The client uses this data to connect to the device and open an interactive SSH shell.
- **Data through REST:** In the REST example, the device has an HTTPS server running on it with a given endpoint for temperature readings. This endpoint is

protected by HTTPS basic authentication. Upon receiving a new request, it is added to the list of approved request for the given period of time, and access mode REST, the username, and the connection URL are sent along with the approval. The client uses this data to make a simple HTTPS request to the given endpoint, receiving the data in JSON.

- **Action through GRPC:** GRPC is Google’s RPC protocol based on Google’s data interchange format, protobuf.<sup>4</sup> In the GRPC example, a GRPC command on the device called playSong is provided. This command receives a song name as a parameter and a username-password pair as metadata. Upon receiving a new request, it is added to the list of approved requests for the given period of time, and access mode GRPC, the username, and the connection URL are sent along with the approval. The client uses this data to make a GRPC request to the given endpoint, triggering the song to be played.

To make things clearer, example code for the REST scenario is provided below.

```
0  const credentials = {
1    key : fs.readFileSync('sslcert/localhost.key', 'utf8'),
2    cert: fs.readFileSync('sslcert/localhost.crt', 'utf8')
3  };
4
5  const app = express();
6
7  const approved_requests = {
8    // EMPTY
9  };
10
11  app.use('/temperature', basicAuth({
12    authorizer: createAsyncAuthorizer(approved_requests),
13    authorizeAsync: true,
14    challenge: true
15  }));
16
17  app.get('/temperature', function (req, res) {
18    const temperatureReading = getTemperatureReading();
19    const result = {
20      requestor: req.auth.user,
21      data: temperatureReading.data,
22      requestTime: new Date().toISOString(),
23      readingTime: temperatureReading.time.toISOString(),
24      format: 'Celsius'
25    };
26
27    res.setHeader('Content-Type', 'application/json');
28    res.send(JSON.stringify(result));
29  });
30
31  class Server {
32    constructor(http_port) {
33      this.httpsServer = https.createServer(credentials, app);
34      this.httpsServer.listen(http_port);
35    }
36  }
```

---

<sup>4</sup>[developers.google.com/protocol-buffers](https://developers.google.com/protocol-buffers)

```

37   start_access(requestId, password) {
38       approved_requests[requestId] = password;
39   }
40
41   stop_access(requestId) {
42       delete approved_requests[requestId];
43   }
44 }
45
46 module.exports = Server;

```

**Listing 3.5:** REST Access Utility Class (Node.js)

The above utility class can be used by the device like this:

```

0   const service = new RESTServer(serverPort);
1   // ...
2   service.start_access(request.requestId, secret);
3   // ...
4   service.stop_access(request.requestId);

```

**Listing 3.6:** REST Access on the Device (Node.js)

### 3.7.5 Challenges and Library Implementation Highlights

It has been challenging working with Ethereum smart contracts for the following reasons:

- **Fast evolution:** Ethereum and the associated technologies are changing fast and have not reached a stable version yet. For this reason, it is hard to find up-to-date information with regard to interfaces and SDKs and there are numerous contradictions in the information online.
- **Asynchronicity:** Interacting with smart contracts is inherently asynchronous: requests are sent over HTTP or over other high-latency channels, events are received over WebSockets, etc. Moreover, it is generally accepted practice to wait for a certain number of blocks (i.e. confirmations) before relying on an accepted transaction, which contributes even more to the asynchronous nature of interaction with smart contracts.
- **Privacy:** Ethereum nodes are the gateways to the Ethereum network: they can read data, retrieve old transactions, and validate and send new ones. During development, the goal was to let the device owner depend on a node of their own choice: their own Ethereum node running on their local network or even a third-party service. However, Ethereum client SDKs require *unlocking* sender addresses by providing a passphrase through the node itself. Sharing private data with third-party services would be highly insecure, so the choice has been made to compose and sign transactions locally, sending only already signed transactions to the node. This increased security, but also made the library more complex.

Below is an example code from the Device Rental System helper library for asynchronously waiting for an event with a given number of confirmations.

```
0 // ...
1
2 async waitForNewRequest() {
3   const eventData = await this.waitForEvent('NewRequest');
4   await this.waitForBlock(eventData.blockNumber + this.nconfirmationsdef - 1);
5   return {
6     requestId: eventData.returnValues.requestId,
7     pubkey: eventData.returnValues.pubkey
8   };
9 }
10
11 waitForEvent(eventName) {
12   return new Promise((resolve, reject) => {
13     const subscription = this.contract.events[eventName]((error, result) => {
14       if (error)
15         return reject(error);
16
17       subscription.unsubscribe();
18       resolve(result);
19     });
20   });
21 }
22
23 waitForBlock(blockNumber) {
24   return new Promise((resolve, reject) => {
25     const subscription = this.web3.eth.subscribe('newBlockHeaders')
26       .on('data', data => {
27         if (data.number == blockNumber) {
28           subscription.unsubscribe();
29           resolve();
30         }
31       })
32       .on('error', reject);
33   });
34 }
```

**Listing 3.7:** Waiting for Events (Node.js)

Below is an example of composing and sending specific requests.

```
0 async request(pubkey, value, gas, nconfirmations) {
1   const rawTransaction = await this.getRawTxData('request', [pubkey], value, gas);
2   return await this.sendTransaction(rawTransaction, nconfirmations, gas);
3 }
4
5 async getRawTxData(methodName, args, value, gas) {
6   const cnt = await this.web3.eth.getTransactionCount(this.senderAddress);
7   const gasPrice = await this.web3.eth.getGasPrice();
8   const callData = this.contract.methods[methodName].apply(this, args).encodeABI();
9
10  const txdata = {};
11  txdata.nonce = this.web3.utils.toHex(cnt);
12  txdata.gasPrice = this.web3.utils.toHex(gasPrice);
13  txdata.gasLimit = this.web3.utils.toHex(gas);
14  txdata.to = this.contractAddress;
15  txdata.value = this.web3.utils.toHex(value);
16  txdata.data = callData;
```

```

17
18   const tx = new EthereumTx(txdata);
19   tx.sign(this.senderPrivateKey);
20   const serializedTx = tx.serialize();
21   const rawTransaction = '0x' + serializedTx.toString('hex');
22   return rawTransaction;
23 }
24
25 sendTransaction(rawTransaction, nconfirmations, gas) {
26   nconfirmations = nconfirmations || this.nconfirmationsdef;
27
28   return new Promise((resolve, reject) => {
29     this.web3.eth.sendSignedTransaction(rawTransaction)
30       .on('confirmation', (confirmationNumber, receipt) => {
31         if (confirmationNumber == (nconfirmations - 1)) {
32           // ...
33         }
34       })
35       .on('error', reject);
36   });
37 }

```

**Listing 3.8:** Composing Requests Locally (Node.js)

## 3.8 Smart Contract-Based Key Exchange over Public Blockchain

A crucial part of the architecture of the Device Rental Platform is proper access control. We have to prevent normal or malicious users from accessing devices they are not entitled to. To achieve this, this section presents a simple key-exchange algorithm on the blockchain.

While many claim that the blockchain technology will help cryptography in many ways, as of today there are few applications directly combining the two (apart from the obvious cryptographical components of the blockchain technology itself). In [33], the authors discuss an on-chain key exchange protocol called Diffie-Hellman-over-Bitcoin, that uses certain parts of Bitcoin transactions to derive a common secret between two parties. Inspired by their work, I propose a simple Diffie-Hellman key exchange protocol over the Ethereum blockchain, implemented via messaging through a smart contract.

### 3.8.1 Key Exchange Overview

Access control is orchestrated on the blockchain, independently of the device itself. Thus, we might consider two authentication options:

- **Access token:** Upon granting access, the smart contract provides the client with an access token. When accessing the device, the client must provide this token.

- **Password:** The smart contract provides a platform for the client and device to agree on a common password. When accessing the device, the client must provide this password.

From this, the access token method cannot be used due to the blockchain’s transparency, i.e. lack of privacy. Smart contract execution is both deterministic and fully transparent to network nodes. As a consequence, if the access token is generated by the smart contract, this process can and will be rerun on most of the network nodes, giving them access to this private information. If, however, the token is generated by the device, it still needs a way of sharing it with the client on-chain, which would also need some kind of encryption in order to prevent unauthorized parties from accessing this information. For encryption, the client and device must agree on some common secret, but if we have this secret anyway, why would we need an access token? We can use this secret directly to access the device.

Thus, the access control method in the Device Rental System is the following: During the access request part (client and contract) and the start of execution part (device and contract), the client and device use the blockchain as a channel for securely and transparently exchanging keys and establishing a common secret only known to them. Subsequently, the device grants local access to clients possessing this secret (e.g. by a password check on a REST endpoint). The client provides this secret when accessing the device and is thus granted access.

Note that this key exchange process could also be achieved off-chain, i.e. by direct client-device communication. However, on-chain key exchange has certain advantages:

- **Simplicity:** With on-chain key exchange, we can reuse a communication channel that we already have: the blockchain. For an off-chain solution, we would need a separate, two-way channel for establishing the common secret. While we could reuse the channel for subsequent communication (e.g. REST, WebSocket, GRPC), off-chain key exchange puts a much bigger burden on the device, requiring the implementation of multi-step key exchange protocols (in addition to key checking) for all used channels.
- **Transparency:** When financial transactions are involved, transparency and auditability are crucial. The blockchain provides a transparent, immutable record of all transactions executed. Thus, if key exchange is done on-chain, the device owner can point to the corresponding transaction logs in case of a dispute, e.g. a client claiming that the owner has stolen their money without providing the actual service. (Note, however, that while the use of the exchanged common key is enforced by our SDKs, the device code can easily bypass this and thus these records cannot serve as full proof of the service actually being offered.)

The following chapters discuss the relevant implementation details of this on-chain key exchange protocol.

### 3.8.2 Diffie-Hellman Key Exchange on the Blockchain

The key exchange algorithm presented here is based on the Diffie-Hellman (DH) key exchange algorithm [34]. This algorithm enables secure exchange of keys over a public channel. The DH algorithm works as follows:

1. Alice and Bob would like to establish a common secret. First, they agree on a large prime  $p$  and a base (also called generator)  $g$ . Note that these values are public.
2. Alice and Bob both generate a random private key,  $priv_a$  and  $priv_b$  correspondingly.
3. Alice and Bob both derive a public key from their private keys:

$$pub_a = g^{priv_a} \bmod p$$

$$pub_b = g^{priv_b} \bmod p$$

4. Alice sends her public key  $pub_a$  to Bob, Bob computes:

$$secret_b = (pub_a)^{priv_b} \bmod p$$

5. Bob sends his public key  $pub_b$  to Alice, Alice computes:

$$secret_a = (pub_b)^{priv_a} \bmod p$$

6. Alice and Bob now share a common secret:

$$secret_b = (pub_a)^{priv_b} \bmod p = g^{priv_a priv_b} \bmod p = (pub_b)^{priv_a} \bmod p = secret_a$$

A DH-based key exchange protocol was chosen for the following reasons:

- **Security:** The Diffie-Hellman key exchange algorithm has been used widely for decades and has also been given a thorough examination by numerous experts of cryptography. That is, the DH algorithm has been proven to be secure if used properly.
- **Availability:** Being a ubiquitous key exchange algorithm, DH has thorough support in many languages through libraries and SDKs. It is generally discouraged to reimplement cryptographic algorithms, due to the numerous pitfalls and errors that could undermine the algorithm's security. With Diffie-Hellman, we can rely on well-tested libraries written by experts, thus minimizing the costly error of incorrect implementation of key exchange.

For the on-chain key exchange protocol presented here, the following steps are required:

1. The client and device agree on the DH parameters (prime and generator). This can either be hardcoded into the device and client SDKs or stored on the smart-contract.

2. The client and device both generate their own private keys and derive the corresponding public keys.
3. While requesting access through the contract, the client sends her public key, thus making it available to the device.
4. The device, upon receiving the client's public key, derives the shared secret.
5. While granting access through the contract, the device sends its public key, thus making it available to the client. The device also enables access using the given secret locally for the requested time interval.
6. The client, upon receiving the device's public key, derives the shared secret.
7. The client accesses the device on the given communication channel and provides her secret. The device checks whether the provided secret matches its own. If it does not, it rejects the request. Otherwise, it grants access to the client.

Note that, for accessing the device, the client must use a secure channel (e.g. HTTPS), otherwise the secret will leak and be available for others to use. The use of secure channels thus must be enforced on the device side for security reasons.

Note also that one is supposed to use different keys for every exchange, i.e. the device must generate new keys for every access request it serves.

### 3.8.3 Key Exchange Implementation

As discussed in the previous section, two transactions are needed to implement DH: one to let the client share her public key, and another to let the device share its own. For this, however, we do not actually need to introduce new transactions. As we have two-way on-chain communication for access control anyway, we can piggyback these pieces of data as metadata on the existing requests:

1. the client sends her public key as metadata along with her **request** transaction,
2. the device sends its public key as metadata along with its **access\_started** transaction.

With this architecture, the only overhead associated with key exchange is the slightly increased transaction cost due to the larger transaction payloads. There is no significant time overhead.



Conceptually, the **request** transaction's metadata is structured as follows:

```
0 {  
1   pubkey: ...  
2 }
```

**Listing 3.9: request** metadata (JSON)

The only attribute needed is the client's public key. Note that some other information, including the payment amount, must be included in the transaction, but they are not considered to be part of the transaction's metadata.

The **access\_started** transaction's metadata is structured as follows:

```
0 {  
1   // key-exchange parameters  
2   pubkey: ...,  
3  
4   // common access parameters  
5   username: ...,  
6   accessEndTime: ...,  
7   accessMode = ...,  
8  
9   // access mode specific parameters  
10  host = ...,  
11  port = ...  
12 }
```

**Listing 3.10: access\_started** metadata (JSON)

The public key is sent along so that the client can immediately derive the common secret and use it as a password together with the corresponding username to access the device. Other parameters include various access parameters. See the previous sections for more details.

## Chapter 4

# Smart Contracts: Towards Model-based Techniques

### 4.1 Challenges and Pitfalls of Smart Contract Programming

A paradigm previously unknown to many programmers, developing on the Ethereum Platform involves numerous pitfalls and security issues [35]. This chapter presents some of these problems, while also proposing some ways to alleviate them.

One of the most common issues of smart contract programming is privacy. The EVM's most commonly used high-level programming language, Solidity, provides the usual OOP primitives for encapsulation and information hiding. It is easy to forget that anything stored or processed on the blockchain is public and visible to everyone; thus, the value of private variables can also be read from the global state of the Ethereum blockchain. This makes it challenging to implement many privacy-critical applications (e.g. gambling).

In [36] the authors present the implementation of a simple rock-paper-scissors game on the blockchain as part of the course they teach. The students' typical first solution leaks the first player's choice, thus making it trivial for her opponent to choose something that beats it. Another pitfall in this game is that once the opponent knows he is going to lose for sure, he is not incentivized anymore to send any money and participate. These problems all have solutions (send money ahead, commit to one's choice but not disclose it), but they are not straightforward for people coming from other programming paradigms.

Another notable example of the exploits of the Ethereum Platform is the infamous DAO attack. A DAO is a *Decentralized Autonomous Organization*: an organization that is governed by the code of a smart contract running on the blockchain. Depending on the actual implementation, money transfers and changes to the contract's code might only be effective once a certain majority of stakeholders have accepted them through voting. The name *The DAO* is used to refer to one specific DAO created by the Slock.it team and started in the Spring of 2016. Not long after raising \$150m in a crowdfunding campaign, a recursive call bug was found in the contract's code and soon exploited by some unknown

hacker before the community could react, thus making the organization unable to use most of its funds.<sup>1</sup>

Recently, in the Fall of 2017, \$150m's worth of ether was frozen and made permanently unavailable due to a bug found in a popular Ethereum wallet's smart contract implementation, soon to be inadvertently exploited by some user after its discovery. The bug made it possible for anyone to destroy the contract, due to improper use of a library contract that resulted in incomplete initialization.<sup>2</sup>

According to [36], a common mistake observed while teaching a lab course on smart contract programming is the incorrect encoding of state machines. Contract programming typically includes encoding state machines, either explicitly or implicitly. Mistakes in encoding state machines will often lead to leaking money in corner cases, either making it possible for attackers to claim money they are not entitled to, or making money unavailable, possibly forever.

The above examples make it clear that smart contract programming involves numerous pitfalls with often millions of dollars at stake. There have been several attempts to alleviate these issues and make smart programming safer and less prone to errors. Some people have been working on formal verification methods to prove the correctness of smart contract code mathematically [37]. Others have proposed alternative programming languages to Solidity (or even EVM bytecode), potentially making formal verification an inherent feature of the language [38]. This thesis proposes an alternative approach: using formal models and code generation to avoid many of the above pitfalls.

## 4.2 State Modeling for Smart Contract Programming

Formal modeling has been an integral part of many areas of software development. The use of such methods is characteristic of high-risk domains, such as medical applications or aviation controllers. Modeling a problem using standardized diagrams lets programmers abstract out and concentrate on the main structure and logic of their solution. This makes it easier to reason about this logic and find errors, even automatically, using various analysis tools.

Another advantage of formal models is the ability to generate code: Once a software has a detailed specification in the form of formal models, the relevant code can often be generated automatically, preventing many errors that would happen during *manual translation* and making the development process faster.

Agreeing with the authors' aforementioned concerns in [36] about the implementation of complex state machines in smart contract code, I propose a model-driven approach to design smart contract logic using standard UML statecharts, and potentially even automatically generate smart contract Solidity code from such diagrams.

---

<sup>1</sup>[www.coindesk.com/understanding-dao-hack-journalists](http://www.coindesk.com/understanding-dao-hack-journalists)

<sup>2</sup>[www.cryptocoinsnews.com/i-accidentally-killed-it-parity-wallet-bug-locks-150-million-in-ether](http://www.cryptocoinsnews.com/i-accidentally-killed-it-parity-wallet-bug-locks-150-million-in-ether)

Even though smart contract programming is generally accepted to be tricky, very few model-based approaches have been used or investigated so far. In [39], the authors build on the *ADICO* method borrowed from the area of institutional analysis to specify functionalities of smart contracts using easy-to-read rule-based statements. Urs Zeidler’s `uml2solidity` project<sup>3</sup> can be used to generate Solidity code from UML class diagrams, thus offering a way to describe the structure of smart contracts using formal diagrams. This thesis proposes an alternative approach: Instead of describing the structure of smart contracts, let us use UML state diagrams to capture the crucial parts of their logic.

### 4.3 Example: State Modeling of the Device Rental Platform

The advantages of state modeling in smart contract programming are best demonstrated by showing its use during the implementation of the Device Rental Platform. The choice of tool for this was the YAKINDU framework<sup>4</sup> for state machine modeling. This Eclipse-based project lets us model complex state machines graphically, test them manually, extend them, and even do some formal verification using external tools.

As described in Section 3.5, the smart contract for orchestrating exclusive, time-limited device access in a safe manner would require five types of transactions: **request**, **access\_started**, **access\_finished**, **access\_failed**, and **cancel**. The latter makes users able to cancel their request before execution starts or after execution fails, and be able to withdraw their money.

The proposed model has three states: **Device Free**, **Waiting for Device ACK**, and **Device Occupied**.

- In the **Device Free** state, the CPS device is idle (i.e. not connected to any client). The contract is unoccupied, waiting for incoming requests. Failed requests can still be canceled in this state.
- In the **Waiting for Device ACK** state, a new request has been received and accepted, but execution on the CPS device has not started yet. The new request or previously failed requests can be canceled in this state. To prevent new requests from *blocking* the contract, a timeout mechanism is used: if execution does not start for a given time interval, other requests can replace the one currently occupying the contract.
- In the **Device Occupied** state, a request has been received and accepted, and execution on the device has started as signaled by the device itself. Failed requests can still be canceled in this state. This state can only be left upon a signal from the device itself, as only the device can know the details of the execution (i.e. whether execution has finished or failed).

---

<sup>3</sup>[github.com/UrsZeidler/uml2solidity](https://github.com/UrsZeidler/uml2solidity)

<sup>4</sup>[www.itemis.com/en/yakindu/state-machine](http://www.itemis.com/en/yakindu/state-machine)

Having defined the semantics of the three main states, let us now take another look at the five request types:

- **request**: a new request for device access. Can come from any address. Can be accepted if the device is free (*Device Free* state), or timeout has been passed in the *Waiting for Device ACK* state.
- **access\_started**: execution has started on the device. Can come only from the device itself (i.e. its corresponding Ethereum address). Can be accepted in the *Waiting for Device ACK* state. After this, the contract is in the state *Device Occupied*.
- **access\_finished**: execution has finished on the device. Can come only from the device itself (i.e. its corresponding Ethereum address). Can be accepted in the *Device Occupied* state. After this, the contract is in the state *Device Free*.
- **access\_failed**: execution has failed on the device. Can come only from the device itself (i.e. its corresponding Ethereum address). Can be accepted in the *Device Occupied* state. After this, the contract is in the state *Device Free*. The money associated with failed requests can be withdrawn at any time by the owner of the funds.
- **cancel**: cancel request and withdraw funds. Can come only from the issuer of the request to be canceled. Withdrawable requests are either failed requests or accepted requests that have not been executed yet. Can appear in any state.

Even though there are only three states in this model, it is already quite complex and has several corner cases. The above English description is prone to discrepancies and ambiguities in interpretation, and translating it into actual contract code is a nontrivial task. Thus, a standard UML state model is used instead.

Figure 4.1 shows the state model implementation in the YAKINDU statechart tool. As it can be seen from the model, not only is the semantics of the program much clearer in graphical form, we can also encode various implementation details in the statechart. For example, the developer might realize that every request needs a unique identifier so that the client and device can reference them. Thus, a simple nonce-based ID-generation method is also encoded in the statechart: upon receiving a **request** transition from the state **Device Free**, a new identifier is generated by calling the **generateRequestId** method of the contract (to be implemented manually), the nonce is incremented and the new request is registered. State transition conditions have also been encoded in the diagram, e.g. a **cancel** request received in the state **Waiting for Device ACK** can result in a state transition if the canceled request is the current request, or could result in remaining in the same state otherwise.

A big advantage of tools like YAKINDU is the ability to actually run the state model manually. This way of manual testing not only enables the developer to test corner cases and discover bugs but is also a nice way to better understand the problem and the execution logic.

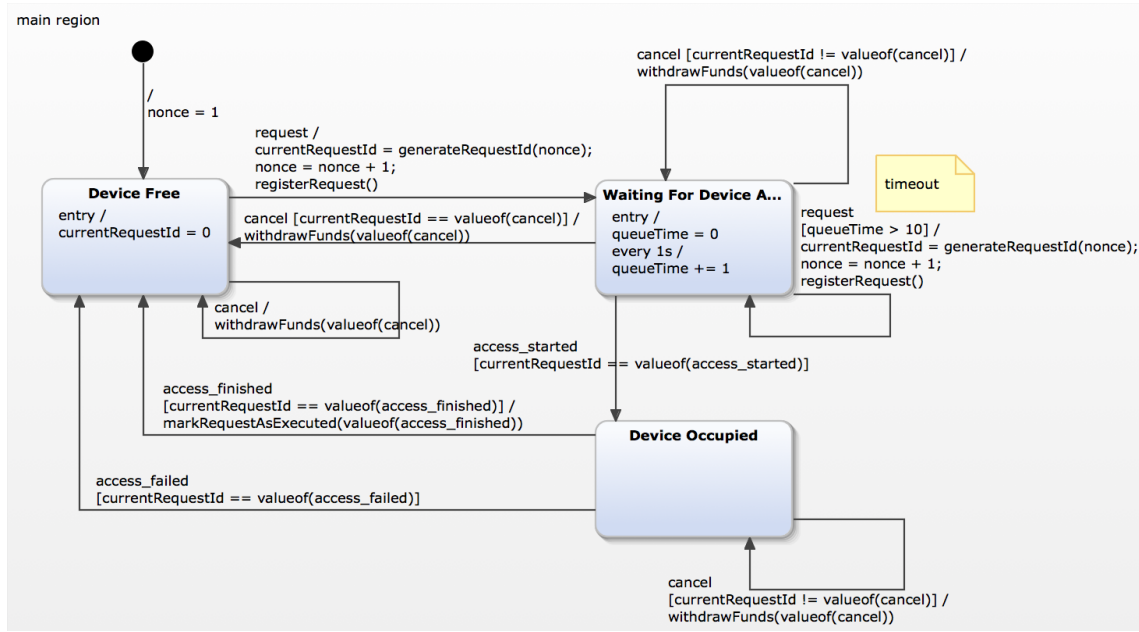
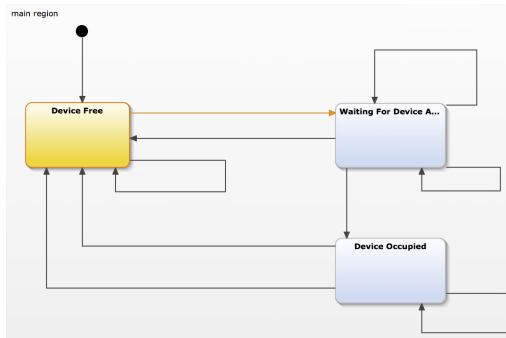
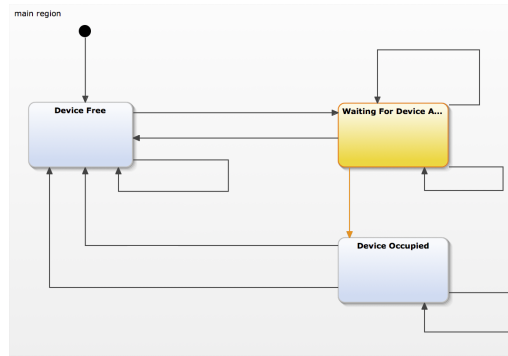


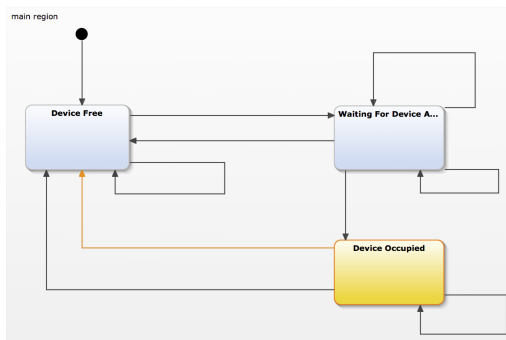
Figure 4.1: Device Rental Platform UML State Model



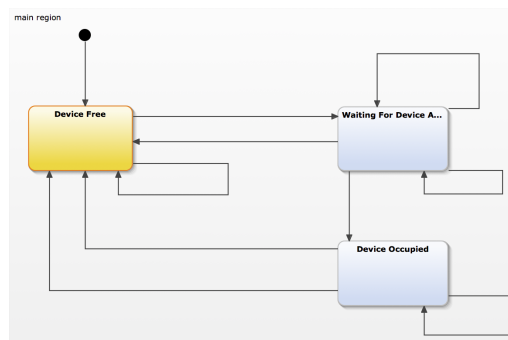
(1)



(2)



(3)



(4)

Figure 4.2: Normal Execution Flow

An example execution of the model can be seen on figure 4.2. The state changes from **Device Free** to **Waiting for Device ACK** to **Device Occupied** to **Device Free**. The state transitions are triggered using the following sequence of transactions: **request**, **access\_started**, **access\_finished**.

Other flows and contracts can be modeled similarly.

## 4.4 Towards Automatic Code Generation

As mentioned earlier, a big advantage of formal modeling is the ability to automatically generate code from the model. This way we can prevent many human errors that would arise while translating the model into code.

The implementation of a fully functional code generation framework for Solidity code is beyond the scope of this thesis. Instead, a step-by-step method for manual translation is presented, thus showing the feasibility of this kind of smart contract development. The author hopes that the reasoning outlined here will encourage the blockchain community to investigate the use of such models in smart contract development and start implementing the relevant tools. The full code of the manually translated contract can be found in Appendix A.1.

The state execution system implemented in Solidity code needs to be able to encode states and transitions first. To do this, it is best to leverage Solidity's enum type which is the recommended type to represent state machines according to the official language documentation.<sup>5</sup>

```
0 enum State {  
1     DeviceFree,  
2     WaitingForDeviceAck,  
3     DeviceOccupied,  
4     Invalid,  
5     Choice  
6 }  
7  
8 enum Transition {  
9     Request,  
10    Cancel,  
11    AccessStarted,  
12    AccessFinished,  
13    AccessFailed  
14 }
```

**Listing 4.1:** Contract States and Transitions (Solidity)

Note the introduction of two new states: `Invalid` and `Choice`. In the diagram above only valid transitions are included, i.e. transitions not included are implicitly assumed to be invalid. In Solidity code, however, a full specification of states and transitions is needed. The state `Invalid` was introduced to mark invalid state transitions. For example, an `access_finished` transition from the state `Device Free` is not allowed and thus results in the state `Invalid`.

The introduction of the new state `Choice` is necessary due to the fact that the source state and transition cannot always clearly specify the resulting state. This is due to some

---

<sup>5</sup><https://solidity.readthedocs.io/en/develop/common-patterns.html#state-machine>

state transitions depending on potentially complex conditions, e.g. if the source state is **Waiting for Device Access** and the transition is **cancel**, we need to evaluate a specific condition to see the resulting state. Thus, I included the transient state **Choice**, the role of which is to enable us to evaluate such conditions and choose the appropriate end state.

Instead of encoding all transitions in a long list of if-else statements, the state execution engine can leverage Solidity's mapping data structure instead. A mapping consists of key-value pairs and is stored in the contract's storage.

```

0 // State -> Transition -> State
1 mapping(uint => mapping(uint => uint)) public transitions;
2
3 function registerTransition(State from, Transition transition, State to) internal {
4     transitions[uint(from)][uint(transition)] = uint(to);
5 }
6
7 function initializeExecutionEngine() internal {
8     registerTransition(State.DeviceFree, Transition.Request      , State.WaitingForDeviceAck);
9     registerTransition(State.DeviceFree, Transition.Cancel      , State.DeviceFree);
10    registerTransition(State.DeviceFree, Transition.AccessStarted , State.Invalid);
11    registerTransition(State.DeviceFree, Transition.AccessFinished, State.Invalid);
12    registerTransition(State.DeviceFree, Transition.AccessFailed  , State.Invalid);
13
14    registerTransition(State.WaitingForDeviceAck, Transition.Request      ,State.WaitingForDeviceAck);
15    registerTransition(State.WaitingForDeviceAck, Transition.Cancel      ,State.Choice);
16    registerTransition(State.WaitingForDeviceAck, Transition.AccessStarted ,State.DeviceOccupied);
17    registerTransition(State.WaitingForDeviceAck, Transition.AccessFinished, State.Invalid);
18    registerTransition(State.WaitingForDeviceAck, Transition.AccessFailed  , State.Invalid);
19
20    registerTransition(State.DeviceOccupied, Transition.Request      , State.Invalid);
21    registerTransition(State.DeviceOccupied, Transition.Cancel      , State.DeviceOccupied);
22    registerTransition(State.DeviceOccupied, Transition.AccessStarted , State.Invalid);
23    registerTransition(State.DeviceOccupied, Transition.AccessFinished, State.DeviceFree);
24    registerTransition(State.DeviceOccupied, Transition.AccessFailed  , State.DeviceFree);
25 }

```

**Listing 4.2:** Initializing the State Execution Engine (Solidity)

Note that as code execution and the use of storage both require gas, the choice of using contract storage entails a trade-off: the choice to have a one-time initialization cost for storage usage (as `initializeExecutionEngine` is only called once), and thus avoid the cost of evaluating lots of if-else statements that would be required for every single request during state lookup. Using a mapping also greatly simplified the resulting code. The cost benefits of this trade-off could be further investigated using various benchmarks or using a detailed calculation of gas costs associated with code execution.

The access modifier `internal` means that these methods cannot be called externally, only from other methods of the same contract.

Having defined all the state transitions, it is time to create the transaction endpoints:

```

0 function request(string pubkey) external payable transitionNext(Transition.Request, 0) {
1     // EMPTY
2 }
3

```



```

4 function cancel(int id) external transitionNext(Transition.Cancel, id) {
5     // EMPTY
6 }
7 function access_started(int id, string pubkey, string data)
8     external transitionNext(Transition.AccessStarted, id) onlyOwnerOrDevice {
9     // EMPTY
10 }
11 function access_finished(int id)
12     external transitionNext(Transition.AccessFinished, id) onlyOwnerOrDevice {
13     // EMPTY
14 }
15 function access_failed(int id)
16     external transitionNext(Transition.AccessFailed, id) onlyOwnerOrDevice {
17     // EMPTY
18 }

```

**Listing 4.3:** Transaction Endpoints (Solidity)

The access modifier `external` means that these methods can only be invoked by sending transactions to the contract. This means that these transactions can cause changes in the contract’s state, and will leave a mark on Ethereum’s blockchain, making them transparent and traceable.

Solidity makes it possible to define so-called *modifiers*. Modifiers can be added to any other method, extending its functionality by executing some extra code either before or after the method body (or both). In Listing 4.3, the `transitionNext` modifier is used to connect these transaction endpoints to the state machine execution engine.

```

0 function getNextState(State state, Transition transition) internal view returns (State) {
1     return State(transitions[uint(state)][uint(transition)]);
2 }
3
4 function clarifyNextState(State state, Transition transition, int arg) internal view returns(State) {
5     if (state == State.WaitingForDeviceAck && transition == Transition.Cancel)
6         return (arg == currentRequestId) ? State.DeviceFree : State.WaitingForDeviceAck;
7     require(false); // should not reach here...
8 }
9
10 modifier transitionNext(Transition transition, int arg) {
11     State nextState = getNextState(currentState, transition);
12
13     if (nextState == State.Choice)
14         nextState = clarifyNextState(currentState, transition, arg);
15
16     require(nextState != State.Invalid);
17
18     checkGuard(currentState, transition, arg);
19     performExit(currentState);
20     performAction(currentState, transition, arg);
21     performCustomTransitionLogic(currentState, transition, arg);
22
23     _; // note: supposed to be empty
24
25     performEntry(nextState);
26     currentState = nextState;
27 }

```

**Listing 4.4:** State Transition (Solidity)

The `transitionNext` modifier in Listing 4.4 works like this: First, the next state from the state mapping is retrieved using the current state (`currentState`) and the current transition (`transition`). If the resulting state is a choice state (`State.Choice`), some custom condition is evaluated (`clarifyNextState`) to get the actual state. As a consequence of the following `require` call, if the resulting state does not exist (`State.Invalid`), an exception is thrown. An exception means that all preceding changes are reverted, and all the provided gas is used up. The sender can check whether her transaction failed through the corresponding transaction receipt.

Once the next state (`nextState`) is determined, the engine continues to check any guard conditions on the transaction, perform the exit condition of the old state, perform the action associated with the current transition, perform the entry condition of the new state, and finally update the current state.

```

0 function checkGuard(State state, Transition transition, int arg) internal view {
1     if (state == State.WaitingForDeviceAck && transition == Transition.Request) {
2         require(now - queueTime > timeout);
3     }
4
5     else if (state == State.WaitingForDeviceAck && transition == Transition.AccessStarted) {
6         require(currentRequestId == arg); // arg ~ id
7     }
8
9     // ...
10 }
11
12 function performEntry(State state) internal {
13     if (state == State.DeviceFree) {
14         currentRequestId = 0;
15     }
16
17     else if (state == State.WaitingForDeviceAck) {
18         queueTime = now;
19     }
20 }
21
22 function performAction(State state, Transition transition, int arg) internal {
23     if (state == State.DeviceFree && transition == Transition.Request) {
24         currentRequestId = generateRequestId(nonce);
25         nonce = nonce + 1;
26         registerRequest();
27     }
28
29     else if (state == State.DeviceFree && transition == Transition.Cancel) {
30         withdrawFunds(arg);
31     }
32
33     // ...
34 }
35
36 function performExit(State state) internal {
37     // ...
38 }

```

**Listing 4.5:** State Transition Components (Solidity)

Note that the code observed so far could almost entirely be generated automatically from the statechart. The developer is also allowed to extend the generated code at certain places. For example, methods specified in the statechart are generated, but their bodies have to be implemented manually:

```

0 function generateRequestId(int nonce) internal pure returns (int) {
1     // <USER_CODE>
2     return nonce;
3     // </USER_CODE>
4 }
5
6 function withdrawFunds(int id) internal {
7     // <USER_CODE>
8     var r = incompleteRequests[id];
9     // note: throws on nonexistent entry
10
11     require(r.addr == msg.sender);
12
13     var amount = r.value;
14     delete incompleteRequests[id];
15     msg.sender.transfer(amount);
16     // </USER_CODE>
17 }

```

**Listing 4.6:** Extension Points (Solidity)

Another extension point is the `performCustomTransitionLogic` method, in which the developer can execute arbitrary code before the state transition takes place.

## 4.5 Discussion of Code Generation

The method for generating Solidity code from statecharts as described above, although it was executed *manually* this time, could be executed automatically. Having generated code with well-defined extension points for the developers, many of the pitfalls of smart contract development could be prevented.

However, there still are some issues to solve. UML state diagrams are not powerful enough by themselves to express some concepts of smart contract logic. One of the missing pieces is the ability to specify modifiers for transaction endpoints. In Listing 4.3 two such modifiers are used: `payable` and `onlyOwnerOrDevice`. The former is a built-in modifier that enables transaction endpoints to receive funds, a functionality that is needed for almost all smart contract applications. The latter is a custom modifier used to control the addresses allowed to send a specific transaction. Here, some state transitions are only allowed if the sender is the device (or the owner). `onlyOwnerOrDevice` is implemented in the following way:

```

0 address public owner;
1 address public device;
2
3 modifier onlyOwner {
4     require(msg.sender == owner);
5     _;
6 }

```

```

7 function setDeviceAddress(address newDevice) onlyOwner external {
8     device = newDevice;
9 }
10
11 modifier onlyOwnerOrDevice {
12     require(msg.sender == owner || msg.sender == device);
13     _;
14 }

```

**Listing 4.7:** Custom Modifiers (Solidity)

This means that before executing the method to which this modifier is attached (e.g. `access_started`), the sender’s Ethereum address is examined first. If the address matches neither the owner’s nor the device’s, the transaction is rejected by throwing an exception and thus reverting all state changes.

One way to alleviate the limitations of UML state diagrams with respect to Solidity’s modifiers is to include this check as a standard transition condition instead of using modifiers. This works with `onlyOwnerOrDevice`. With `payable`, however, Solidity offers no alternative other than using modifiers. Another solution would be to provide extension points in method headers:

```

0 function request(string pubkey) external transitionNext(Transition.Request, 0)
1     // <USER_CODE>
2     payable
3     // </USER_CODE>
4 {
5     // EMPTY
6 }
7
8 // ...

```

**Listing 4.8:** Extending Modifiers (Solidity)

A third option would be to extend state model functionalities through a metamodel. This could let the developers append specific annotations to transitions in their statecharts, which would be generated as modifiers. This, however, would probably raise compatibility issues with existing tools.

Another unsolved problem is the way of defining state transition arguments and passing them to the execution engine. For example, as visible in Listing 4.3, the `cancel` transaction needs to provide the request identifier corresponding to the request to be canceled. As these transaction endpoints are only connected to the state execution engine through the `transitionNext` modifier, there seems to be no elegant way to make these arguments available to the engine. A possible solution is to directly access and decode the message payload at later points in the contract’s code.

## 4.6 Additional Smart Contract Functionalities

Through well-defined extension points, the developer is able to add arbitrary code to the generated smart contract skeleton. This section presents some of the added functionality during the implementation of the Device Rental Platform.

In order to be able to use HTTPS for client-device communication, the client needs to be able to access the device's SSL certificate. The best place to store this is the contract itself: by letting the device store its certificate on the blockchain, we get a tamper-proof, safe, transparent store readable to anyone.

```
0 string public certificate;
1
2 function setCertificate(string cert) external onlyOwnerOrDevice {
3     certificate = cert;
4 }
```

**Listing 4.9:** Certificate Handling (Solidity)

With these few lines, the SSL certificate can be safely stored on the blockchain. We can be sure that no unauthorized parties can change it to an arbitrary certificate, as long as the owner's or device's private keys are not exposed to the public. Note that `certificate` is defined as a public member variable. As a consequence, the Solidity compiler will automatically generate the corresponding getter method. As calling a contract method locally is a low-cost operation (i.e. no transaction needs to be sent), it is relatively simple and cheap to check whether the certificate on the contract matches the one used by the device and only update in case of deviation.

The actual request handling is implemented as follows:

```
0 struct Request {
1     address addr;
2     uint time;
3     uint256 value;
4 }
5
6 mapping(int => Request) public incompleteRequests;
7 int private currentRequestId;
8 uint256 public weiPerSecond;
9
10 function validateRequest(Request r) internal view {
11     require(r.value > 0);
12     require(r.value % weiPerSecond == 0);
13 }
14
15 function registerRequest() internal {
16     var r = Request(msg.sender, now, msg.value);
17     validateRequest(r);
18
19     incompleteRequests[currentRequestId] = r;
20 }
21
22 function withdrawFunds(int id) internal {
23     var r = incompleteRequests[id]; // note: throws on nonexistent entry
```

```

24     require(r.addr == msg.sender);
25
26     var amount = r.value;
27     delete incompleteRequests[id];
28     msg.sender.transfer(amount);
29 }
30
31 function markRequestAsExecuted(int id) internal {
32     var r = incompleteRequests[id];
33     profit += r.value;
34     delete incompleteRequests[id];
35 }
36
37 function getAllowedExecutionTimeSeconds(int id) public view returns (uint) {
38     var r = incompleteRequests[id]; // note: throws on nonexistent entry
39     return r.value / weiPerSecond;
40 }

```

**Listing 4.10:** Request Handling (Solidity)

The member variable `weiPerSecond` represents the service cost per second ( $10^{18}$  wei = 1 ether); this can be specified in the contract’s constructor. The member variable `profit` keeps track of the profit from successful transactions so that the owner can withdraw these funds.

## 4.7 Limitations of Model-Based Smart Contract Programming

In the preceding sections, we have seen a workflow for modeling smart contracts using standard UML diagrams and generate code from these diagrams. It has been observed that the code generation part could potentially be automated. Using this method, smart contract programming could potentially become much safer. The overhead for safety is relatively small, consisting of slightly increased execution and deployment costs.

However, as mentioned before, there are certain limitations to this method:

- **Complexity:** Supporting forks and other constructs where a given source state and state transition can lead to multiple end states based on some complex criteria is a non-trivial task that increases code complexity.
- **Incompleteness:** There is no direct way to represent Solidity’s modifiers in UML diagrams. One solution would be to extend the UML schema to include special modifier attributes on state transitions. However, this only works if this attribute is specified for all occurrences of the transition.
- **Parameter passing:** It is problematic to pass transaction argument to the state execution engine. One solution would be to, instead of passing variables, directly read the message body when needed.

However, these obstacles can certainly be tackled. I hope that the analysis presented in this thesis will inspire the blockchain community to further investigate this topic.

## Chapter 5

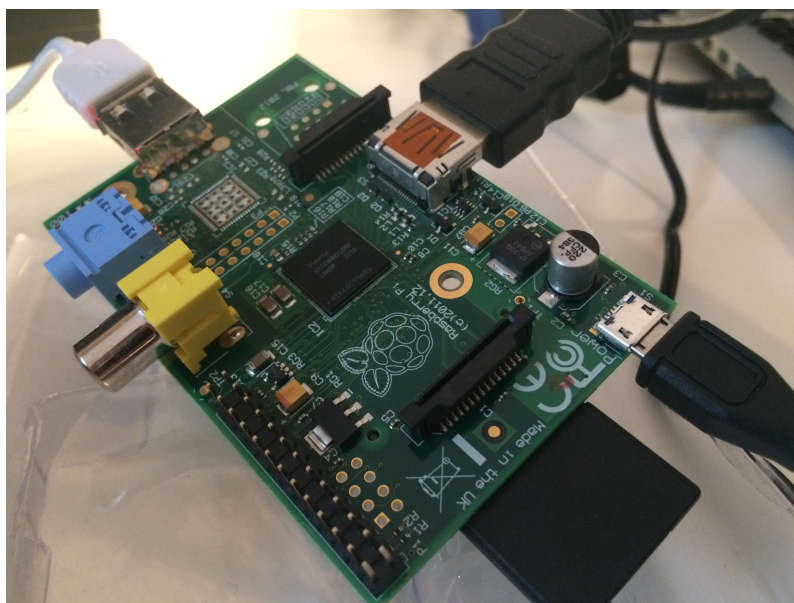
# Demonstration and Evaluation

This chapter presents a test evaluation of the Device Rental Platform and a discussion of the results. The tests presented include a local test with an actual CPS device and a step-by-step guide for trying the contract on the Ethereum test network.

### 5.1 Local Testing

#### 5.1.1 Test Setup

For the local test evaluations, a Raspberry Pi model A (2013) was chosen, equipped with an 8GB SD card and the Raspbian operating system. The device was connected to the local LAN network using an Ethernet cable and a USB-Ethernet adaptor. The device was configured through SSH.



**Figure 5.1:** Local Test Setup



A local Ethereum test network was set up using `geth`, the official Go Ethereum client. The node was initialized with some test accounts with certain initial ether balances. The difficulty was set so that the block time would be around 1 second. The node was set to mine continuously. RPC functionality to the node was opened through WebSocket on a specific address available from the local network.

To summarize, the local network had the following three components:

- **Node (contract):** a `geth` node running on a MacBook Pro. A Firewall exception has been added so that the node is accessible inside the local network.
- **Client:** a Node.js application running on the same MacBook Pro.
- **Device:** a Node.js application running on the Raspberry Pi device.

### 5.1.2 Results

In the test environment above, the system functioned as expected. The client initiated rental by requesting access from the contract, the device granted access upon receiving the request notification, and then the two sides entered direct communication.

Some shortcomings observed:

- It takes a long time to set up the Node.js environment on a small device like the Raspberry Pi model A. This includes updating to the latest Node.js version, downloading libraries, etc. This time could be reduced by using pre-bundled releases.
- Issuing self-signed certificates for IP addresses turned out to be challenging. The address has to be specified in the certificate's *SubjectAltName* field, or otherwise Node.js will not accept it.

## 5.2 Cost Benchmarks

According to `geth`'s gas cost estimation mechanism, deploying the contract would need about 2,238,061 gas which, at the time of writing, would require about \$5's worth of ether at 5 Gwei gas price, and would require about 3 minutes to deploy.<sup>1</sup>

Table 5.1 presents some benchmarks from contract execution in the Remix online Solidity IDE, containing transaction and execution costs in gas, and the corresponding dollar estimates (using the current exchange rate).

Using these numbers, the contract execution overhead for renting out a device consists of a one-time deployment fee of \$5 and a \$0.3 execution fee per rental. There is also a \$0.1 cost associated with withdrawing any profits, which does not need to happen after each device rental. For the client, there is only a one-time device rental fee and the associated execution cost of about \$0.7.

---

<sup>1</sup>source: <https://ethgasstation.info>

Transaction	$C_T$	$C_E$	$C_\$$
request	153,992	132,016	\$0.671
access_started	35,055	12,119	\$0.111
access_finished	36,041	50,617	\$0.203
withdrawProfit	19,643	13,371	\$0.077

**Table 5.1:** Simple Cost Benchmarks (Remix)

Note that the above numbers can vary with each execution, especially taking into account the payload carried by the **access\_started** and **access\_finished** transactions (public key, approval) and the potentially rapid changes in gas costs and exchange rates. However, these benchmarks show the magnitude of the cost overhead of the contract execution.

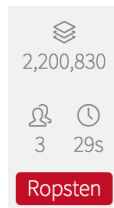
Seeing that the average gas price results in a few minutes delay, one future improvement for the Device Rental System would be to take into account such delays and send the **access\_finished** transaction a bit earlier accordingly. This also entails a trade-off by the owner: she can choose higher transaction fees for potentially better device utilization.

As a conclusion, as long as the rental price of the device covers at least the contract execution (which is likely, especially when clients rent for longer periods of time), the cost overhead imposed by using a smart contract is acceptable. However, the growing transaction fees on the Ethereum blockchain might mean that other platforms could provide a more stable basis for the Device Rental Platform.

### 5.3 A Step-By-Step Guide for Testing on the Public Testnet

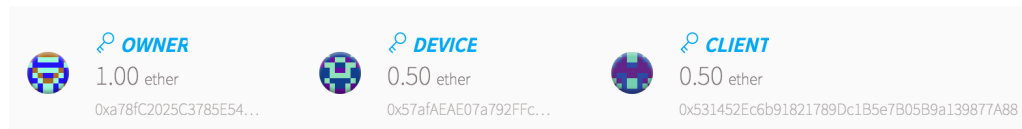
There are multiple ways to interact with the Ethereum Network, including the geth CLI, the Remix online IDE, and the official Ethereum Wallet application (also known as Mist). In this section, a step-by-step guide for deploying and using the Device Rental contract is given.

We are going to use Ethereum’s Ropsten test network to avoid spending real ether. Please note, however, that the steps presented here would be the same for the real Ethereum network. To switch to the Ropsten public test network on the Ethereum Wallet application, choose Develop/Network/Ropsten from the menu. Once the client has synced the network’s blockchain data, something similar should be visible in the lower left corner:



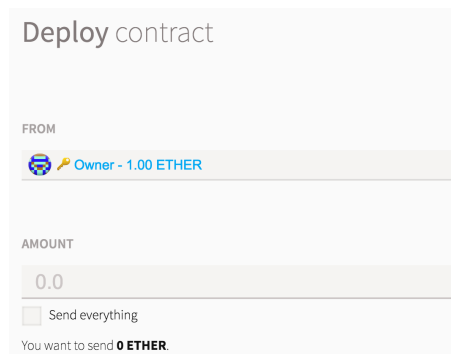
**Figure 5.2:** Network Information (Mist)

First, we have to make sure we have accounts with sufficient balances. New accounts can be created in the Ethereum Wallet application by clicking on *Add Account* under the *Wallets* page. To acquire funds on the test network, there are multiple *ether faucets*, where one can claim ether for free.<sup>2</sup> It is also fairly easy to transfer ether between accounts using the desktop application. Once the accounts have been created, it should look like this:



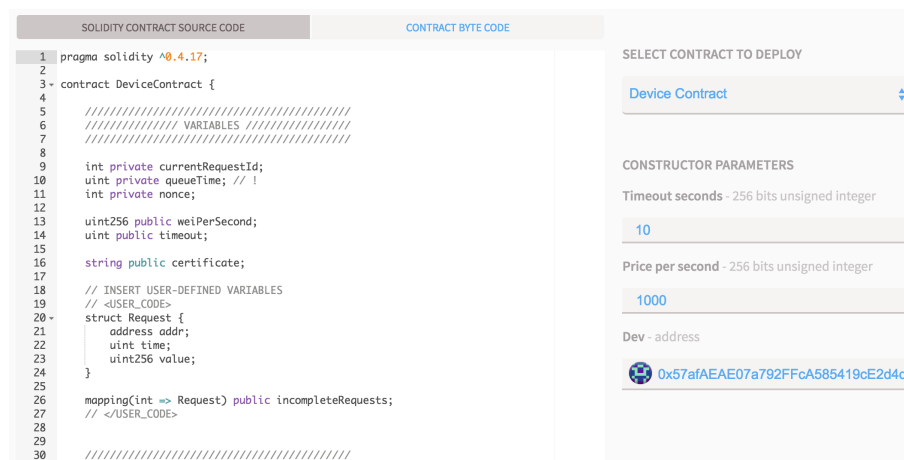
**Figure 5.3:** Available Accounts (Mist)

The next step is to deploy the contract. For this, simply click on *Deploy New Contract* under the *Contracts* page. First, we have to choose the account with which to create the contract:



**Figure 5.4:** Deploy Contract 1. (Mist)

And then simply copy the contract code from the repository into the source code field.<sup>3</sup> Once we have set the constructor parameters, it should look something like this:



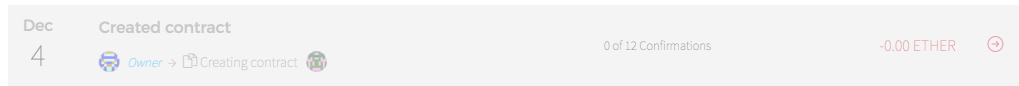
**Figure 5.5:** Deploy Contract 2. (Mist)

<sup>2</sup>e.g. faucet.metamask.io

<sup>3</sup>Note: the contract might not compile as the desktop application treats warnings as errors; in this case, simply proceed to fix the warnings and then continue with the next steps.

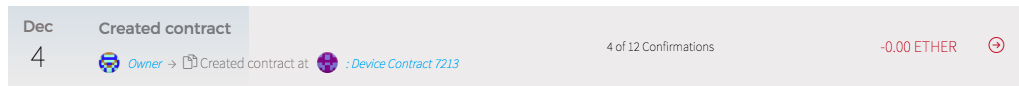
After clicking on *Deploy*, we see an overview of all the relevant information. Enter the password and click *Send Transaction*.

Back on the *Wallet* page, at the bottom, we can see all transactions. First, we see the contract deployment as an unconfirmed transaction:



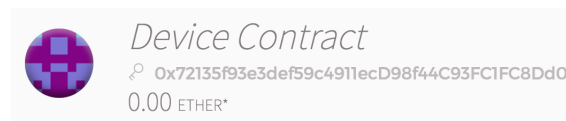
**Figure 5.6:** Unconfirmed Transaction (Mist)

After a few minutes, we should receive some confirmations, which means that the contract was created successfully:



**Figure 5.7:** Confirmed Transaction (Mist)

After this, the contract appears under the *Contracts* page:



**Figure 5.8:** Contract (Mist)

Clicking on it, we can view the values of all the public members of the contract:



**Figure 5.9:** Contract Values (Mist)

On the right, we can send transactions to the contract. Let us first send a **request** transaction with a dummy public key parameter and 10,000 wei's worth of ether to the contract.

Select function

Request

Pubkey - string

client\_pubkey

Execute from

Client - 0.50 ETH

Send ether\*

0.0000000000000001

EXECUTE

**Figure 5.10: request transaction (Mist)**

After sending the transaction and waiting for some time, a *New Request* event should appear on the bottom:

Watch contract events

Filter events

New Request

request id: 1

pubkey: client\_pubkey

0 of 12 Confirmations

**Figure 5.11: New Request Event (Mist)**

This is the event that notifies the device about the new request, including the request identifier and the public key. Using this request id, we can extract some interesting information from the contract:

Get allowed execution time seconds

Id - 256 bits signed integer

1

10

**Figure 5.12: Request Information 1. (Mist)**

Incomplete requests

256 bits signed integer

1

Addr

Client

Time

1512410667 (2 minutes ago)

Value

10000

**Figure 5.13:** Request Information 2. (Mist)

Next, let us send an *access\_started* transaction using the device’s account. We will use dummy values for the public key and metadata fields for now.

Select function

Access\_started

Id - 256 bits signed integer

1

Pubkey - string

device\_pubkey

Data - string

{ approval: true }

Execute from

Device - 0.50 ETH

EXECUTE

**Figure 5.14:** *access\_started* transaction (Mist)

After a period of time, we should see the *Request Approved* event at the bottom. This is the event that notifies the client that she can access the device now. Sent by an actual CPS device, the data field would contain all the necessary information for connecting to the device.

Request Approved

request id: 1

pubkey: device\_pubkey

data: { approval: true }

0 of 12 Confirmations

**Figure 5.15:** Request Approved Event (Mist)

And finally, we need to send an *access\_finished* transaction:

Select function

Access\_finished

Id - 256 bits signed integer

1

Execute from

Device - 0.50 ETHER

EXECUTE

**Figure 5.16:** `access_finished` transaction (Mist)

After this transaction, the contract is back in the original state, and the request is gone:

Incomplete requests

256 bits signed integer

1

Addr

0x00

Time

0

Value

0

**Figure 5.17:** Request Information 3. (Mist)

We have just completed the basic execution flow of the contract. Other flows worth trying include canceling requests before execution starts, signaling execution failure and canceling failed requests, and sending unauthorized transactions. Below is an example of the latter: let us send an `access_started` transaction to the contract from the Client's address:

Select function

Access\_started

Id - 256 bits signed integer

2



Pubkey - string

...

Data - string

...

Execute from

  Client - 0.50 ETH

EXECUTE

**Figure 5.18:** Invalid Transaction 1. (Mist)

After clicking *Execute*, the overview window immediately tells us that contract execution will not succeed and we cannot even send this transaction.

[illegible]

**Figure 5.19:** Invalid Transaction 2. (Mist)

This is a mechanism of the Ethereum Wallet to avoid wasting ether unnecessarily. If we did send the transaction, it would fail and all the gas provided would be used up.

## 5.4 Evaluation

Initial evaluations suggest that the architecture of the Device Rental System works as intended and is able to fulfill its role. The next steps include a more thorough investigation of the robustness and security of the system. Another important step is improving usability, by providing features such as a form for generating device descriptors and automatically deploying contracts.



## Chapter 6

# Summary and Future Work

This thesis was an investigation of the topic of blockchain and smart contract programming applied to CPS systems. To present and address the challenges of blockchain and IoT discussed earlier, the implementation of a smart contract based device rental system was presented. With a special emphasis on model-driven smart contract development, this work contributes to the solution of the aforementioned issues by serving as an example of how these technologies can work together and how their problems might be tackled. The next few paragraphs suggest some directions for further work.

First, the proposed model-driven smart contract programming method can be further elaborated. An important future step would be implementing (semi-)automatic code generation using tools like YAKINDU. Another important direction is the static analysis of such models to catch bugs early in the development process. The combination of the UML-based modeling approach outlined in this thesis with other methods is also a promising future direction.

Second, the proposed on-chain key exchange protocol could also benefit from further research. Finding and implementing other use cases is one possible direction. One could investigate the safety of the approach outlined in this thesis, and experiment with other approaches to find one that works the best.

And last, while the Device Rental System works as a Proof-of-Concept prototype, it is far from being a stable, ready-to-be-used product. Thus, I would like to continue its development in the future, possibly together with members of the open-source community.

# Acknowledgements

I am grateful to my advisor, Imre Kocsis; without his help, encouragement, and insights, this thesis would not exist. I would also like to thank all my family and friends for supporting me all along.

# Bibliography

- [1] Deloitte. Blockchain @ Telco. Technical report, 2016.  
[https://www2.deloitte.com/content/dam/Deloitte/za/Documents/technology-media-telecommunications/za\\_TMT\\_Blockchain\\_TelCo.pdf](https://www2.deloitte.com/content/dam/Deloitte/za/Documents/technology-media-telecommunications/za_TMT_Blockchain_TelCo.pdf)  
(last accessed on 2017-12-03).
- [2] NIST. Cyber-Physical Systems Public Working Group, 2017.  
<https://pages.nist.gov/cpspwg> (last accessed on 2017-12-03).
- [3] World Economic Forum. Deep Shift: Technology Tipping Points and Societal Impact. Technical report, 2015.  
[http://www3.weforum.org/docs/WEF\\_GAC15\\_Technological\\_Tipping\\_Points\\_report\\_2015.pdf](http://www3.weforum.org/docs/WEF_GAC15_Technological_Tipping_Points_report_2015.pdf) (last accessed on 2017-12-03).
- [4] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.  
<https://bitcoin.org/bitcoin.pdf> (last accessed on 2017-12-03).
- [5] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. *OSDI*, 99:173–186, 1999.
- [6] D. Malone and K.J. O’Dwyer. Bitcoin Mining and its Energy Footprint. *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communities Technologies (ISSC 2014/CICT 2014)*, pages 280–285, 2014.
- [7] Sunny King and Scott Nadal. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. *self-published paper*, 2012. <http://peerco.in/assets/paper/peercoin-paper.pdf>  
(last accessed on 2017-12-03).
- [8] Rick Echevarria. 2017: A Summer of Consensus, 2017. <https://software.intel.com/en-us/blogs/2017/10/05/2017-a-summer-of-consensus> (last accessed on 2017-12-03).
- [9] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys and Tutorials*, 18(3):2084–2123, 2016.
- [10] Wei Dai. B-money. <http://www.weidai.com/bmoney.txt> (last accessed on 2017-12-03).
- [11] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. *Security and Privacy in Social Networks*, pages 197–223, 2013.

- [12] D. Ron Shamir and A. Quantitative Analysis of Anonymity in the Bitcoin Transaction Graph. *Security and Privacy in Social Networks*, pages 6–24, 2013.
- [13] Vitalik Buterin. Ethereum White Paper, 2013.  
<https://github.com/ethereum/wiki/wiki/White-Paper> (last accessed on 2017-12-03).
- [14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8437:436–454, 2014.
- [15] Yonatan Sompolsky and A Zohar. Accelerating Bitcoin’s Transaction Processing. Fast Money Grows on Trees, Not Chains. *IACR Cryptology ePrint Archive*, 881, 2013.
- [16] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper*, pages 1–32, 2014. <http://gavwood.com/paper.pdf> (last accessed on 2017-12-03).
- [17] National Science Foundation. Cyber-Physical Systems (CPS) (NSF17529). Technical report, 2017. <https://www.nsf.gov/pubs/2017/nsf17529/nsf17529.htm> (last accessed on 2017-12-03).
- [18] Paulo Tabuada. Cyber-Physical Systems: Position Paper. *NSF Workshop on Cyber-Physical Systems*, 2006.
- [19] Maria Rita Palattella, Mischa Dohler, Alfredo Grieco, Gianluca Rizzo, Johan Torsner, Thomas Engel, and Latif Ladid. Internet of Things in the 5G Era: Enabling Technologies and Business Models. *IEEE Journal on Selected Areas in Communications*, 34(3):510–527, 2016.
- [20] Filament Project. Fondation for the next Economic Revolution. 2016.  
<https://filament.com/assets/downloads/Filament Foundations.pdf> (last accessed on 2017-12-03).
- [21] Sanjay Panikkar, Sumabala Nair, Paul Brody, and Veena Pureswaran. ADEPT: An IoT Practitioner Perspective. 2015.  
<http://ibm.biz/devicedemocracy> (last accessed on 2017-12-03).
- [22] Serguei Popov. The Tangle. 2016.  
[https://iota.org/IOTA\\_Whitepaper.pdf](https://iota.org/IOTA_Whitepaper.pdf) (last accessed on 2017-12-03).
- [23] Chronicled. Open Registry for IoT. 2016.  
<http://www.chronicled.org/whitepaper.pdf> (last accessed on 2017-12-03).
- [24] Arshdeep Bahga and Vijay K. Madisetti. Blockchain Platform for Industrial Internet of Things. *Journal of Software Engineering and Applications*, 09(10):533–546, 2016.
- [25] Shiyong Yin, Jinsong Bao, Yiming Zhang, and Xiaodi Huang. M2M Security Technology of CPS Based on Blockchains. *Symmetry*, 9(9), 2017.

- [26] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access*, 4:2292–2303, 2016.
- [27] Standards Australia. Roadmap For Blockchain Standards. Technical report, 2017. [http://www.standards.org.au/OurOrganisation/News/Documents/Roadmap\\_for\\_Blockchain\\_Standards\\_report.pdf](http://www.standards.org.au/OurOrganisation/News/Documents/Roadmap_for_Blockchain_Standards_report.pdf) (last accessed on 2017-12-03).
- [28] ITU-T FG-DFS. Distributed Ledger Technologies and Financial Inclusion. Technical report, 2017. [https://www.itu.int/en/ITU-T/focusgroups/dfs/Documents/201703/ITU\\_FGDFS\\_Report-on-DLT-and-Financial-Inclusion.pdf](https://www.itu.int/en/ITU-T/focusgroups/dfs/Documents/201703/ITU_FGDFS_Report-on-DLT-and-Financial-Inclusion.pdf) (last accessed on 2017-12-03).
- [29] IUT-T FG-DPM. Focus Group on Data Processing and Management to support IoT and Smart Cities & Communities, 2017. <https://www.itu.int/en/ITU-T/focusgroups/dpm/Pages/default.aspx> (last accessed on 2017-12-03).
- [30] IEEE-SA. 2418 - Standard for the Framework of Blockchain Use in Internet of Things (IoT), 2017. <https://standards.ieee.org/develop/project/2418.html> (last accessed on 2017-12-03).
- [31] Michael Compton, et al. The SSN Ontology of the W3C Semantic Sensor Network Incubator Group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012.
- [32] Xiaoming Zhang, Yunping Zhao, and Wanming Liu. A Method for Mapping Sensor Data to SSN Ontology. *International Journal Science and Technology*, 8(6):303–316, 2015.
- [33] P McCorry, Sf Shahandashti, D Clarke, and F Hao. Authenticated Key Exchange over Bitcoin. In *International Conference on Research in Security Standardisation*, number December, pages 3–20, 2015.
- [34] Whitfield Diffie and Martin E Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
- [35] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts. In *International Conference on Principles of Security and Trust*, number July, pages 164–186, 2015.
- [36] Kevin Delmolino, Mitchell Arnett, Ahmed E Kosba, Andrew Miller, and Elaine Shi. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *IACR Cryptology ePrint Archive*, 2015:460, 2015.
- [37] Karthikeyan Bhargavan, Nikhil Swamy, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, and Thomas Sibut-Pinote. Formal Verification of Smart Contracts. *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS’16*, pages 91–96, 2016.

- [38] Russell O Connor. Simplicity: A New Language for Blockchains. *arXiv:1711.03028*, 2017.
- [39] Christopher K. Frantz and Mariusz Nowostawski. From Institutions to Code: Towards Automated Generation of Smart Contracts. *Proceedings - IEEE 1st International Workshops on Foundations and Applications of Self-Systems, FAS-W 2016*, pages 210–215, 2016.

# Appendix

## A.1 Complete Code of the Device Rental Platform Smart Contract

```
0  pragma solidity ^0.4.17;
1
2  contract DeviceContract {
3
4      //////////////////////////////////
5      //////////////////// VARIABLES ////////////////////
6      //////////////////////////////////
7
8      int private currentRequestId;
9      uint private queueTime;
10     int private nonce;
11
12     uint256 public weiPerSecond;
13     uint public timeout;
14
15     // INSERT USER-DEFINED VARIABLES
16     // <USER_CODE>
17     string public certificate;
18     uint256 private profit;
19
20     struct Request {
21         address addr;
22         uint time;
23         uint256 value;
24     }
25
26     mapping(int => Request) public incompleteRequests;
27     // </USER_CODE>
28
29
30     //////////////////////////////////
31     //////////////////// CONSTANTS ////////////////////
32     //////////////////////////////////
33
34     // INSERT USER-DEFINED CONSTANTS
35     // <USER_CODE>
36     address public owner;
37     address public device;
38
39     modifier onlyOwnerOrDevice {
40         require(msg.sender == owner || msg.sender == device);
41         _;
```

```

42     }
43
44     modifier onlyOwner {
45         require(msg.sender == owner);
46         _;
47     }
48
49     function setDeviceAddress(address newDevice) onlyOwner public {
50         device = newDevice;
51     }
52     // </USER_CODE>
53
54
55     //////////////////////////////////////
56     /////////////////// EVENTS ///////////////////
57     //////////////////////////////////////
58
59     // INSERT USER-DEFINED EVENTS
60     // <USER_CODE>
61     event NewRequest(int indexed requestId, string pubkey);
62     event RequestApproved(int indexed requestId, string pubkey, string data);
63     // </USER_CODE>
64
65
66     //////////////////////////////////////
67     ////////// STATE EXECUTION ENGINE //////////
68     //////////////////////////////////////
69
70     enum State {
71         DeviceFree,
72         WaitingForDeviceAck,
73         DeviceOccupied,
74         Invalid,
75         Choice
76     }
77
78     enum Transition {
79         Request,
80         Cancel,
81         AccessStarted,
82         AccessFinished,
83         AccessFailed
84     }
85
86     // State -> Transition -> State
87     mapping(uint => mapping(uint => uint)) public transitions;
88
89     State public currentState;
90
91     function registerTransition(State from, Transition transition, State to) internal {
92         transitions[uint(from)][uint(transition)] = uint(to);
93     }
94
95     function getNextState(State state, Transition transition) internal view returns (State) {
96         return State(transitions[uint(state)][uint(transition)]);
97     }
98
99     function clarifyNextState(State state, Transition transition, int arg) internal view returns (
100         State) {
101         if (state == State.WaitingForDeviceAck && transition == Transition.Cancel)

```



```

101         return (arg == currentRequestId) ? State.DeviceFree : State.WaitingForDeviceAck;
102
103         require(false); // should not reach here...
104     }
105
106     function initializeExecutionEngine() internal {
107         registerTransition(State.DeviceFree, Transition.Request, State.WaitingForDeviceAck);
108         registerTransition(State.DeviceFree, Transition.Cancel, State.DeviceFree);
109         registerTransition(State.DeviceFree, Transition.AccessStarted, State.Invalid);
110         registerTransition(State.DeviceFree, Transition.AccessFinished, State.Invalid);
111         registerTransition(State.DeviceFree, Transition.AccessFailed, State.Invalid);
112
113         registerTransition(State.WaitingForDeviceAck, Transition.Request, State.WaitingForDeviceAck);
114         registerTransition(State.WaitingForDeviceAck, Transition.Cancel, State.Choice);
115         registerTransition(State.WaitingForDeviceAck, Transition.AccessStarted, State.DeviceOccupied);
116         registerTransition(State.WaitingForDeviceAck, Transition.AccessFinished, State.Invalid);
117         registerTransition(State.WaitingForDeviceAck, Transition.AccessFailed, State.Invalid);
118
119         registerTransition(State.DeviceOccupied, Transition.Request, State.Invalid);
120         registerTransition(State.DeviceOccupied, Transition.Cancel, State.DeviceOccupied);
121         registerTransition(State.DeviceOccupied, Transition.AccessStarted, State.Invalid);
122         registerTransition(State.DeviceOccupied, Transition.AccessFinished, State.DeviceFree);
123         registerTransition(State.DeviceOccupied, Transition.AccessFailed, State.DeviceFree);
124     }
125
126     function performEntry(State state) internal {
127         if (state == State.DeviceFree) {
128             currentRequestId = 0;
129         }
130
131         else if (state == State.WaitingForDeviceAck) {
132             queueTime = now;
133         }
134     }
135
136     function checkGuard(State state, Transition transition, int arg) internal view {
137         if (state == State.WaitingForDeviceAck && transition == Transition.Request) {
138             require(now - queueTime > timeout);
139         }
140
141         else if (state == State.WaitingForDeviceAck && transition == Transition.AccessStarted) {
142             require(currentRequestId == arg); // arg ~ id
143         }
144
145         else if (state == State.DeviceOccupied && transition == Transition.Cancel) {
146             require(currentRequestId != arg); // arg ~ id
147         }
148
149         else if (state == State.DeviceOccupied && transition == Transition.AccessFinished) {
150             require(currentRequestId == arg); // arg ~ id
151         }
152
153         else if (state == State.DeviceOccupied && transition == Transition.AccessFailed) {
154             require(currentRequestId == arg); // arg ~ id
155         }
156     }
157
158     function performAction(State state, Transition transition, int arg) internal {
159         if (state == State.DeviceFree && transition == Transition.Request) {
160             currentRequestId = generateRequestId(nonce);

```

```

161         nonce = nonce + 1;
162         registerRequest();
163     }
164
165     else if (state == State.WaitingForDeviceAck && transition == Transition.Request) {
166         currentRequestId = generateRequestId(nonce);
167         nonce = nonce + 1;
168         registerRequest();
169     }
170
171     else if (state == State.DeviceFree && transition == Transition.Cancel) {
172         withdrawFunds(arg);
173     }
174
175     else if (state == State.WaitingForDeviceAck && transition == Transition.Cancel) {
176         withdrawFunds(arg);
177     }
178
179     else if (state == State.DeviceOccupied && transition == Transition.Cancel) {
180         withdrawFunds(arg);
181     }
182
183     else if (state == State.DeviceOccupied && transition == Transition.AccessFinished) {
184         markRequestAsExecuted(arg);
185     }
186 }
187
188 function performExit(State state) internal {
189     // EMPTY
190 }
191
192 modifier transitionNext(Transition transition, int arg) {
193     // 1. retrieve and validate next state
194     // 2. check guard conditions
195     // 3. perform exit actions of old state
196     // 4. perform generated code associated with transition
197     // 5. perform user-defined code associated with transition
198     // 6. perform entry actions of new state
199     // 7. update current state
200
201     State nextState = getNextState(currentState, transition);
202
203     if (nextState == State.Choice)
204         nextState = clarifyNextState(currentState, transition, arg);
205
206     require(nextState != State.Invalid);
207
208     checkGuard(currentState, transition, arg);
209     performExit(currentState);
210     performAction(currentState, transition, arg);
211     performCustomTransitionLogic(currentState, transition, arg);
212
213     _; // note: supposed to be empty
214
215     performEntry(nextState);
216     currentState = nextState;
217 }
218
219
220 ///////////////////////////////////////////////////

```

```

221 ////////////////////////////////////////////////// INIT ///////////////////////////////////
222 //////////////////////////////////////////////////
223
224 function DeviceContract(uint timeoutSeconds, uint256 pricePerSecond, address dev) public {
225     initializeExecutionEngine();
226
227     nonce = 1;
228     timeout = timeoutSeconds;
229     weiPerSecond = pricePerSecond;
230
231     // INSERT ADDITIONAL INIT CODE
232     // <USER_CODE>
233     owner = msg.sender;
234     device = dev;
235     profit = 0;
236     // </USER_CODE>
237 }
238
239
240 //////////////////////////////////////////////////
241 //////////////// OPERATIONS //////////////////////////
242 //////////////////////////////////////////////////
243
244 function generateRequestId(int nonce) internal pure returns (int) {
245     // INSERT IMPLEMENTATION
246     // <USER_CODE>
247     return nonce;
248     // </USER_CODE>
249 }
250
251
252 //////////////////////////////////////////////////
253 //////////////// TRANSITIONS //////////////////////////
254 //////////////////////////////////////////////////
255
256 function request(string pubkey) external payable transitionNext(Transition.Request, 0) {
257     // EMPTY
258     NewRequest(currentRequestId, pubkey);
259 }
260
261 function cancel(int id) external transitionNext(Transition.Cancel, id) {
262     // EMPTY
263 }
264
265 function access_started(int id, string pubkey, string data) external transitionNext(Transition.
AccessStarted, id) onlyOwnerOrDevice {
266     // EMPTY
267     RequestApproved(currentRequestId, pubkey, data);
268 }
269
270 function access_finished(int id) external transitionNext(Transition.AccessFinished, id)
onlyOwnerOrDevice {
271     // EMPTY
272 }
273
274 function access_failed(int id) external transitionNext(Transition.AccessFailed, id)
onlyOwnerOrDevice {
275     // EMPTY
276 }
277

```

```

278     function performCustomTransitionLogic(State oldState, Transition transition, int arg) internal {
279         // INSERT ADDITIONAL USER-DEFINED TRANSITION LOGIC
280         // NOTE: EXECUTED AFTER GENERATED GUARDS AND TRANSITION CODE!
281         // <USER_CODE>
282         // </USER_CODE>
283     }
284
285     // USER-DEFINED FUNCTIONS
286     // <USER_CODE>
287     function validateRequest(Request r) internal view {
288         require(r.value > 0);
289         require(r.value % weiPerSecond == 0);
290     }
291
292     function registerRequest() internal {
293         var r = Request(msg.sender, now, msg.value);
294         validateRequest(r);
295
296         incompleteRequests[currentRequestId] = r;
297     }
298
299     function withdrawFunds(int id) internal {
300         var r = incompleteRequests[id];
301         // note: throws on nonexistent entry
302
303         require(r.addr == msg.sender);
304
305         var amount = r.value;
306         delete incompleteRequests[id];
307         msg.sender.transfer(amount);
308     }
309
310     function markRequestAsExecuted(int id) internal {
311         var r = incompleteRequests[id];
312         var amount = r.value;
313
314         profit += amount;
315         delete incompleteRequests[id];
316     }
317
318     function getAllowedExecutionTimeSeconds(int id) public view returns (uint) {
319         var r = incompleteRequests[id];
320         // note: throws on nonexistent entry
321         return r.value / weiPerSecond;
322     }
323
324     function setCertificate(string cert) external onlyOwnerOrDevice {
325         certificate = cert;
326     }
327
328     function withdrawProfit() external onlyOwner {
329         var amount = profit;
330         profit = 0;
331         msg.sender.transfer(amount);
332     }
333     // </USER_CODE>
334 }

```

**Listing A.1.1:** Complete Smart Contract Code