University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Smart Contract Application

Florian Schüpfer
Lucern, Switzerland
Student ID: 10-718-328

Supervisor: Sina Rafati Niya, Thomas Bocek
Date of Submission: August 15, 2017

University of Zurich
Department of Informatics (IFI)
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Einleitung: In den letzten Jahren wurden Platformen wie Ebay, Uber oder Airbnb, auf denen Privatpersonen direkt Güter oder Dienstleistungen handeln können, zunehmend populär. Dieses Phänomen wird häufig auch als Peer-to-Peer (P2P) Economy bezeichnet. Im Gegensatz zu echten P2P Computernetzwerken ist die Infrastruktur dieser Platformen nicht dezentralisiert, sondern ist vom Plattformbetreiber und von diversen Zahlungsplatformen abhängig.
Die dezentralisierte Kryptowährung Ethereum wurde im July 2015 eingeführt. Im Gegensatz zu anderen Kryptowährungen, wie zum Beispiel Bitcoin, unterstützt Ethereum nicht nur monetäre Transaktionen zwischen Teilnehmern sondern wurde explizit für die verteilte Ausführung von Code entwickelt. Ein Smart Contract ist Stück Code, dass zusammen mit den dazugehörigen Daten auf einer verteilten Datenbank, der Blockchain, gespeichert wird. Der Zustand solcher Smart Contracts kann von externen Akteuren oder von andere Smart Contracts über Transaktionen verändert werden.

Ziele: Das Hauptziel dieser Arbeit ist der Entwurf und die Implementierung einer Android Anwendung, die das Konzept von Smart Contracts in Ethereum verwendet um das Abschliessen von elektronischen Kauf- und Mietverträgen zwischen zwei Parteien zu ermöglichen, ohne dass eine Drittpartei in den Einigungs- oder Zahlungsprozess involviert ist. Die Anwendung erlaubt Anwendern das flexible Erstellen von Kauf- und Mietverträgen, welche rechtlich gültig sind und ermöglicht den beiden Parteien den Austausch von persönlichen Daten zur Identifizierung über das Scannen von QR-Codes oder über eine Drahtlostechnologie wie Bluetooth, Wi-Fi oder einen anderen Kanal.
Die Anwendung erfüllt weiter folgende nicht-funktionalen Anforderungen:

- **Sicherheit and Privacy:** Die Identifizierungsbedingungen können für jeden Vertrag flexibel ausgewählt werden. Die Anwendung hat keine Sicherheitslücken, die den Verlust von Geld oder persönlichen Benutzerdaten ermöglichen.
- **Erweiterbarkeit:** Die Implementierung der Anwendung sollte so allgemein wie möglich sein, damit sie erweitert werden kann wenn Vertragsanforderungen sich ändern.
- **Zuverlässigkeit und Skalierbarkeit:** Die Anwendung is voll funktional und ist in der Lage sich von grundlegenden Fehlern (z.B. Netzwerkausfall) zu erholen.
- **Usability:** Die Anwendung bietet ein benutzerfreundliches User interface.

Die Arbeit beschäftigt sich weiter auch mit der Frage der rechtlichen Gültigkeit und der Rechtssicherheit von Verträgen. Sie klärt die nötigen Bedingungen zur Identifizierung von Personen in elektronischen Verträgen im Kontext des Schweizer Obligationenrechts (OR) ab.

Resultate: Die Anwendung erfüllt die Grundlegenden Anforderungen. Sie ermöglicht dem Benutzer das Definieren und Erstellen von Kauf- und Mietverträgen auf der Ethereum

Blockchain. Erstellte Verträge sowie perönliche Daten können entweder über einen QR-Code oder über die Wi-Fi P2P Schnittstelle von anderen Benutzern eingelesen werden. Die Verträge sind in der Solidity Skriptsprache implementiert und verwenden Depotzahlungen um die Einhaltung der Vertragsbedingungen zu erzwingen. Alle Zahlungen werden in der ethereumeigenen Währung "Ether" abgewickelt.
In der Arbeit wurden verschiedene Drahtlostechnologien für den Austausch von Vertragsdaten und persönlicher Benutzerdaten miteinander Verglichen. Wi-Fi direkt stellte sich als am Besten geeignet heraus, hauptsächlich wegen der hohen Datenübertragungsrate, welche für den Austausch von Mediendateien (Bilder) erforderlich ist.
Die rechtlichen Bedingungen für die Gültigkeit von Kauf- und Mietverträgen wurden geklärt. Es stellte sich heraus, dass die meisten Verträge rechtlich gültig sind, wenn der Preis und das Kaufobjekt bestimmt sind und wenn die Parteien ihr Einverständnis signalisieren. Diese Bedingungen sind beide erfüllt in der Anwendung. Für Verträge, welche die einfache Schriftlichkeit erfordern, liefert nur die qualifizierte elektronische Signatur die nötige Authentizität, welche aus Kostengründen nicht in der Anwendung integriert ist.
Die Anwendung wurde bezüglich Erzeugungs- und Transaktionskosten, Skalierbarkeit, Sicherheit und Privacy ausgewertet. Es wurde gezeigt, dass die Kosten für das Erstellen von Verträgen signifikant reduziert werden können, wenn nur eine Hash-Signatur statt alle Attribute auf der Blockchain gespeichert werden müssen. Weiter wurde experimentell gezeigt dass nur der verfügbare Speicherplatz die Anzahl Verträge limitiert, welche die Anwendung parallel laden kann. Die Anwendung bietet eine hohe Sicherheit gegen den Diebstahl des Private Keys, da alle Transaktionen an den Ethereum Client lokal signiert werden und die Anwendung weder den Private Key noch das Password für das Wallet File speichert. Persönliche Daten werden nur im internen Speicher der Anwendung abgelegt und nie unverschlüsselt im Netzwerk übertragen. Um den Verlust von Geld bei einem Netzwerkausfall zu verhindern, speichert die Anwendung präventiv alle Verträge falls ein Netzausfall entdeckt wird.

Weitere Arbeiten:

- Sobald mobile Ethereum Clients verfübar sind, kann der externe Client ersetzt werden.
- Um die Rechtssicherheit der Verträge zu erhöhen und um das Abschliessen von Verträgen, welche die einfache Schriftlichkeit erfordern, zu ermöglichen, könnte in Zukunft ein Anbieter von qualifizierten Zertifikaten verwendet werden, um den Vertrag durch beide Parteien zu signieren. In der Schweiz erfordert dies, dass beide Parteien eine SuisseId besitzen.
- Um die Erweiterbarkeit der Verträge zu erhöhen könnte ein externer Code Generator verwendet werden, um die Java Klassen direkt aus dem Binary und der ABI Spezifikation des Smart Contract Codes zu generieren.

Abstract

The fully decentralized cryptocurrency Ethereum was introduced in July 2015. In contrast to other cryptocurrencies like Bitcoin, it not only supports the transfer of funds among participants, but was designed to store and execute code in a distributed manner.
This thesis is about the design and implementation of an Android application that uses the concept of smart contracts in Ethereum to conclude electronic purchase and rent contracts directly between two parties without relying on any Trusted Third Party (TTP) for the agreement or payment process.
The identity requirements necessary to identify a person were investigated in the context of Swiss contract law. It was concluded that it is currently not possible to provide perfect legal security in electronic contracts without depending on a provider of certified electronic signatures.
Different device-to-device technologies were compared and evaluated for the use case of exchanging contract data and personal user data between Android devices. Wi-fi direct was deemed the most suitable for the task mostly because of its support for high data rates.
The smart contracts are developed in the Solidity script language and use deposits to increase the compliance between the parties and all payments are conducted in the Ethereum currency ether. The contracts are deployed on a remote machine that hosts an Ethereum client which is connected over its HTTP-RPC interface with the Android device. The Android client uses the Java Web3j integration library to communicate with the smart contracts on the blockchain.
The application was evaluated in terms of deployment- and transaction costs, scalability, security and privacy. Although it still relies on a remote Ethereum client and does not provide perfect legal security, the application allows users to flexibly specify purchase and rent contracts, exchange personal user data and execute the monetary transactions directly on the blockchain.

Die dezentralisierte Kryptowährung Ethereum wurde im July 2015 eingeführt. Im Gegensatz zu anderen Kryptowährungen, wie zum Beispiel Bitcoin, unterstützt Ethereum nicht nur monetäre Transaktionen zwischen Teilnehmern sondern wurde explizit für die verteilte Ausführung von Code entwickelt.
Diese Arbeit beschäftigt sich mit dem Entwurf und der Implementierung einer Android Anwendung, die das Konzept von Smart Contracts verwendet um das Abschliessen von elektronischen Kauf- und Mietverträgen zwischen zwei Parteien zu ermöglichen, ohne dass eine Drittpartei in den Einigungs- oder Zahlungsprozess involviert wird.
Die Bedingungen um eine Person eindeutig zu identifizieren wurden im Kontext des Schweizerischen Obligationenrechts(OR) untersucht. Es stellte sich heraus, dass es momentan nicht möglich ist in einem elektronischen Vertrag perfekte Rechtssicherheit zu gewährleisten ohne von einem Anbieter von zertifizierten Signaturen abhänging zu sein.
Verschiedene direkte Datenübertragungstechnologien wurden zum Zweck der Übertragung von Vertragsdetails und persönlichen Daten miteinander verglichen. Wi-Fi direct stellte sich dabei als am Besten geeignet heraus, hautpsächlich wegen der hohen Datenübertragungsrate.
Die Verträge wurden in der Solidity Skriptsprache entwickelt und verwenden Depotzahlungen um die Einhaltung der Vertragsbedingungen zu gewährleisten. Die Zahlungen werden alle in der Ethereumwährung Ether abgewickelt. Die Verträge werden auf einem Ethereum Client auf einem separaten Server erzeugt welcher über die HTTP-RPC Schnittstelle mit dem Android Client verbunden ist. Der Android client verwendet die Web3j Java Bibliothek um mit den Smart Contracts auf der Blockchain zu interagieren.
Die Anwendung wurde bezüglich Erzeugungs- und Transaktionskosten, Skalierbarkeit, Sicherheit und Privacy ausgewertet.
Obwohl die Anwendung von einem externen Ethereum Client abhänging ist und noch keine perfekte Rechtssicherheit bietet, ermöglicht sie das Spezifizieren von Kauf- und Mietverträgen, der Austausch von persönlichen Daten sowie das Abwickeln von monetären Transaktionen direkt über die Blockchain.

Acknowledgments

I like to thank my supervisors Sina Rafati and Dr.Thomas Bocek at the University of Zurich for their help and support throughout the project and for giving me the opportunity to perform this thesis.

Florian Schüpfer, Lucern 2017-08-12

Contents

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Description of work 2

1.3 Thesis outline 3

2 Related Work **5**

2.1 Decentralized Sharing App 5

2.2 TransActive Grid 5

2.3 Digix 6

3 Blockchains and Smart Contracts **7**

3.1 Bitcoin 7

3.1.1 Addresses 7

3.1.2 Transactions 8

3.1.3 The blockchain 9

3.1.4 Consensus attacks 11

3.1.5 Limitations of Bitcoin 12

3.2 Ethereum 12

3.2.1 Accounts 13

3.2.2 Transactions and messages . . . 13
3.2.3 Blockchain and mining . . . 14
3.2.4 Light clients . . . 14
3.3 Ethereum Smart contracts . . . 16
3.3.1 Solidity . . . 16
3.3.2 Security considerations . . . 21

4 Legal aspects and Identity Management 25
4.1 Definition . . . 25
4.2 Formation of a contract . . . 25
4.2.1 Parties . . . 26
4.2.2 Conclusion of a purchase contract . . . 26
4.3 Formal requirements . . . 27
4.3.1 Electronic signatures . . . 27
4.4 Implications . . . 28
4.4.1 Legal validity . . . 28
4.4.2 Legal security . . . 29
4.5 Signing documents with the SuisseId qualified signature . . . 30
4.6 Identity management in the blockchain . . . 30
4.6.1 Decentralizing Privacy . . . 31
4.6.2 ShoCard . . . 32
4.6.3 Conclusion . . . 32

5 Device-to-device communication 35
5.1 Bluetooth . . . 35
5.1.1 Physical layer . . . 36
5.1.2 Media access control and link Management . . . 36
5.1.3 L2CAP and higher level protocols . . . 37
5.1.4 Security . . . 37

5.2 Bluetooth Low Energy (BLE) . . . 39
5.2.1 Physical layer . . . 39
5.2.2 Link layer . . . 40
5.2.3 L2CAP . . . 40
5.2.4 Attribute Protocol(ATT) . . . 40
5.2.5 GATT . . . 40
5.2.6 Security . . . 40
5.2.7 GAP . . . 41
5.3 WiFi and WiFi-direct . . . 42
5.3.1 Physical layer . . . 42
5.3.2 Network topology . . . 43
5.3.3 Media access . . . 43
5.3.4 Wi-Fi direct . . . 43
5.4 Evaluation . . . 46

6 Design and implementation 49

6.1 The smart contract . . . 50
6.1.1 The purchase contract . . . 51
6.1.2 The rent contract . . . 53
6.2 The Ethereum client . . . 56
6.2.1 Ethereum client interface . . . 56
6.2.2 Test accounts . . . 57
6.3 Java library . . . 58
6.3.1 External libraries . . . 58
6.3.2 Contract integration code . . . 59
6.3.3 Data model and data access layer . . . 67
6.3.4 Service layer . . . 68
6.4 Android client . . . 73

6.4.1 Structural concepts . . . 73
6.4.2 Activities . . . 79
6.4.3 Device-to-device implementation . . . 85
6.4.4 Contract extension . . . 89

7 Evaluation **91**
7.1 Costs . . . 91
7.1.1 Deployment costs . . . 91
7.1.2 Transaction costs . . . 93
7.2 Scalability . . . 93
7.2.1 Application scalability . . . 94
7.2.2 System scalability . . . 95
7.3 Security . . . 95
7.4 Privacy . . . 95
7.4.1 Privacy of personal user data . . . 96
7.4.2 Privacy of contracts . . . 96

8 Summary and Conclusions **97**
8.1 Conclusion . . . 98
8.1.1 Limitations and future work . . . 99

Bibliography **101**

Abbreviations **109**

Glossary **111**

List of Figures **112**

List of Tables **114**

A Diagrams **117**

B Installation Guidelines **125**

B.1 Installation 125

B.1.1 Compilation from source 125

B.2 Testing 126

B.2.1 Run unit tests 126

B.2.2 Run instrumented unit tests 126

B.3 Getting started 127

C Contents of the CD **129**

Chapter 1

Introduction

1.1 Motivation

In the last decade, the Business-to-Consumer (B2C) platforms dominated the e-commerce business. With the introduction of the first Consumer-to-Consumer (C2C) platforms, a new form of market emerged where private persons come together and can exchange goods and services directly. This market has developed from simple sale and resale platforms like Ebay to more advanced short- and long term rental platforms. Examples for such platforms are Airbnb, where private persons can rent out their unused rooms to other persons or RelayRides, where people can share their unused vehicles. The phenomenon of the rapid growth of the C2C market in the last years is often referred to as the "Sharing Economy" by researchers[61].
A common problem that all these platforms today share is that they rely on a Trusted Third Party (TTP), the platform owner, to operate the platform. This has many disadvantages for consumers. They need to sign up for each platform separately and have to give away their private data to the platform owner. This raises privacy concerns. Another problem is that on most platforms, the possible payment methods are limited by the platform owner. Consumers are forced to use a specific payment platform and sometimes have to pay high transaction fees. A TTP also always is a single point of failure for all participants of the platform. It can be attacked by distributed denial of service (DDOS) attacks or it can be shut down by a central authority. So although the Sharing Economy is often also referred to as the Peer-to-Peer (P2P) economy, its infrastructure is often not (yet) fully decentralized as in real P2P computer networks.
A solution to overcome the requirement of a TTP in a contractual agreement between two or more parties is offered by the concept of smart contracts. A smart contract is a digital protocol that facilitates the agreement process between different parties by enforcing certain predefined rules. The concept of a smart contract was first introduced by Nick Szabo [87] in 1997. In his article, he proposes to embed contractual clauses like property rights directly into hard- and software to make it expensive for a party to breach the protocol. He outlines a hypothetical digital security system for cars, where a cryptographic key represents the ownership of the car and is transferred only if the terms implemented in the protocol are fulfilled. In his example, the owner of a car can withdraw the key from

a leaser who does not pay the monthly rent.
At that time, smart contracts were not practically achievable, mainly because there existed no digital infrastructure that allowed the secure execution of the protocols without the need for a TTP at some point. This changed with the introduction of the cryptocurreny Bitcoin.
Bitcoin [75] is a digital payment system proposed by the pseudonym Satoshi Nakamoto in the year 2008. It allows the transfer of funds between participants without relying on a third party, like a bank. The ownership of currency tokens is maintained with the help of public key cryptography, meaning that a token belongs to the user that holds its associated private key. To maintain the integrity of the network, Bitcoin introduced the blockchain (section 3.2.3), a distributed ledger that stores all transactions that were ever committed in a chain of distinct blocks. Participants have to solve a cryptographic puzzle that is used to verify a block. The main incentive for participants to take part in this process is a reward that is distributed when a block is proofed to be valid. In this manner, new Bitcoin tokens are created.
Although the functionality of Bitcoin can be extended beyond the mere transferring of value tokens by writing special protocols, the scripting language used is not powerful enough to express complex contract logic like in the example envisioned by Nick Szabo.
Ethereum [37], a cryptocurrency that was described first in late 2013 by Vitalik Buterin [27] and went live on 30 July 2015, is the first cryptocurrency with full support for smart contracts. It stores the code and related data of a contract on the blockchain and executes code when participants or other contracts issue transactions. The code of a contract is stored in a byte code format and executed concurrently by all participants on the Ethereum Virtual Machine (EVM).
Ethereum supports several scripting languages to write smart contracts and compiles them into byte code format. The most popular among them is Solidity, a contract oriented high level language whose syntax is similar to JavaScript. Various types of applications can be implemented in Solidity, like “Voting”, “Crowd Funding”, “Blind Auctions”, “Multi Signature Wallets” and more [40] [37].

1.2 Description of work

As of today, concluding an electronic purchase contract between 2 private persons requires a platform to mediate the interests of the seller and the buyer. The platform stores the description of a purchase item and its price and the buyer can interact with the platform to buy it. Payment of the item requires another platform like a bank or a credit card institute.
This thesis addresses the question if it is possible to conclude a legally valid electronic purchase or rental contract directly between 2 private parties without the need of a TTP for the agreement or the payment process.
To clarify legal concerns, the thesis tries to answer the question about the properties that the contracts must have to be legally valid and what identity requirements they must satisfy in order to provide legal security.
Further, the thesis tries to answer the question about which technology for device-to-device communication is the most adequate for exchanging sensitive user data between

the two parties for identification purposes.
The goal of this thesis is to design and implement a smart contract application for Android smart phones that enables the safe conclusion of purchase or rent contracts between 2 parties using the Ethereum blockchain to store the contract details and to process the payments.
The application allows a user to flexibly set up a contract that provides enough information to be legally valid. Further, the 2 parties can exchange their identity by using either their phone camera to scan a QR-code or by exchanging the data through Bluetooth, Wi-Fi or any other device-to-device technology.
The application also has to satisfy the following non-functional requirements:

- **Security and privacy:** The identity requirements that are used in a contract can be selected in a flexible manner. The application does not have any security breaches that allow for loss of money or private data of the users.
- **Expandability:** The implementation of the application is as general as possible to integrate new components when the requirements for a contract change.
- **Reliability and scalability:** The application is fully functional and can recover from basic failures (e.g. network errors)
- **Usability:** The application provides a user-friendly interface.

1.3 Thesis outline

Chapter 2: Discusses related work.

Chapter 3: Explains how blockchain technologies and smart contracts work. It explains the cryptocurrencies Bitcoin and Ethereum in more detail and discusses their similarities and differences. It further explains how smart contracts work in Ethereum and how they can be programmed using the Solidity scripting language. It also discusses the vulnerabilities of Solidity and the Ethereum platform in general.

Chapter 4: Discusses and compares different device-to-device technologies and evaluates them to the use case of exchanging user data between the devices of the contractual parties.

Chapter 5: Discusses the legal aspects of the smart contract application in the context of Swiss law. It specifically tries to answers the questions of legal validity and legal security of the contracts generated by the application. It further discusses approaches to implement identity management in the blockchain.

Chapter 6: Contains the design and implementation details of the application. The first part discusses the solidity implementation of the smart contract used in the application. The second part goes into the details of integrating the smart contract on the blockchain with Java code on the Android device. In the last part, the actual Android client implementation that provides the user interface, is shown.

Chapter 7: Evaluates the application according to the functional and non-functional requirements stated in the introduction chapter.

Chapter 8: Summarizes the results and also points out the limitations of the application and future work.

Chapter 2

Related Work

2.1 Decentralized Sharing App

Bogner et all [22] presented a fully decentralized sharing app. The application uses a smart contract on the Ethereum blockchain for the conclusion of rental contracts.
In the application, the creator of a contract can register rental objects that are stored in a dictionary and referenced by their Id's. The contract stores the description of the item together with its rental price and deposit.
When a renter wants to rent an object, she scans its QR code with the camera of her smart phone. The client then invokes the *rentObject* function on the contract and submits the deposit for the object. The renter is then assigned to the object together with a timestamp. When the object is returned, the owner of the contract can then trigger the *reclaimObject* function that calculates the renting fees for the time of usage and sends the renting fee to the address of the owner. The deposit is then returned to the renter.
The front end of the application that provides the user interface is implemented in javascript and HTML5.

2.2 TransActive Grid

TransActive Grid [92] is a P2P energy trading platform that uses Ethereum smart contracts to manage transactions between participants in a local electric power microgrid.
In the system, every owner of a photovoltaic (PV) facility installs a smart meter that keeps track of the surpluses she makes. The smart meter then updates the available surplus for this participant on a smart contract on the Ethereum blockchain. Interested buyers from the same neighbourhood can then interact with the contract to buy energy credits.

The Brooklyn microgrid [23] is the first renewable energy startup that uses the Transactive Grid system to create a local energy market. The first energy transaction took place on April 11, 2016. The system is not yet fully decentralized because it relies on the payment platform Paypal to conduct the financial transactions.

In the future, the developers want to enable members to define their own energy requirements. The user should choose from which sources she wants to buy energy credits and which price she is willing to pay. They also want to make the system fully decentralized.

2.3 Digix

Digix [33] is an asset tokenisation service built on the Ethereum blockchain. Digix is offering physical gold bullions on the blockchain, more specifically, digital tokens linked to real world physical assets.
DGX tokens are created ("minted") through an Ethereum smart contract with each token representing 1 gram of gold. Each DGX token can be linked definitely to a Proof of Asset (PoA) card of a physical gold bar. The PoA card is stored on a smart contract on the blockchain and contains information like the time stamp of the card creation, the gold bar serial number, the purchase receipt and the audit documentation [32].
The PoA asset card is created through the "Asset Registration Process". The registration process accepts a gold contract from a user and creates the PoA card that is linked to the Ethereum address of the user. To convert the PoA asset card to DGX tokens, a user can use the "Minter Smart Contract" that will hold the PoA card and return 1 DGX token per gram of gold. With the "Recaster Smart Contract", a user can exchange its DGX tokens back into PoA cards. To redeem the PoA card back to the physical gold bar, the user can trigger the "Redemption Process" [32].
The main benefit of DGX tokens is that their value is much more stable than the cryptocurrency ether (ETH), because they are directly backed by gold or other physical assets. Another benefit compared to traditional digital gold certificates is that the system does not rely on a centralized database that holds the information stored in a PoA card. Further, the authors claim that the application is less vulnerable to Man in the Middle (MITM) attacks because users use a desktop client to log in instead of a web form [32].

Chapter 3

Blockchains and Smart Contracts

3.1 Bitcoin

In recent years, the cryptocurrency Bitcoin has gained a lot of attention in the public. In contrast to former attempts on digital currencies that relied on a TTP for maintaining the state of the network, Bitcoin uses a fully decentralized protocol for maintaining consent among all nodes. This protocol enables secure, reliable and to a high degree anonymous transactions [27]. As of April 2017, Bitcoin is the most popular cryptocurrency with a market capitalization of over 29 Billion US-Dollar [30].
In the Bitcoin network, every participant of the network has a unique address and can send currency in transactions to other participants. In each transaction, a certain amount of bitcoins will then be transferred from the senders address to the recipients address. Transactions that took place in a certain time frame are grouped together into distinct blocks that constitute a distributed transaction ledger called a blockchain. This distributed data structure is maintained and extended by participants called miners who constantly try to solve a crtyptographic puzzle called a "Proof of Work" (PoW) to validate new blocks. When transactions are broadcasted by participants in the network, miners include them into the blocks they generate. Miners that successfully validate new blocks are rewarded with newly created bitcoins and transaction fees [27] [75].

In the next subsections discuss the most important concepts of the Bitcoin protocol in more detail.

3.1.1 Addresses

Every participant of the Bitcoin network has to create a wallet file that stores the digital keys used to sign transactions to other participants and the Bitcoin address used to receive transactions from other participants. This wallet file is not stored on the blockchain but on the local hard drive of the user [10].

The digital keys used to sign transactions are composed of a public and a private part:

The private key is just a randomly generated 256 bit number:

$$K_{priv} \in \{0,1\}^{256} \tag{3.1}$$

The 512-bit public key is then generated from the private key using a irreversible mapping known as the Elliptic Curve DSA (ECDSA) algorithm [10]:

$$K_{pub} = ECDSA_{512}(K_{priv}) \tag{3.2}$$

When an address owner wants to publish a transaction, she creates a signature using her private key and at the same time reveals her public key. Miners can then verify the authenticity of a transaction with the accounts public key [10].
The public key of an account is only revealed when it signs a transaction. To receive bitcoins, a hash function of the public key is used as the recipient address. This 160-bit hash is generated with the $RIPMED_{160}$ and SHA_{256} hash functions:

$$K_{address} = RIPEMD_{160}(SHA_{256}(K_{pub}))) \tag{3.3}$$

3.1.2 Transactions

Transactions are data structures that are used to transfer value between participants. Every transaction contains a number of inputs and outputs. The inputs reference outputs of transactions in the past called "unspent transaction outputs" or UTXO. UTXO are chunks of bitcoins that belong to a specific owner and are recorded on the blockchain. The UTXO that belong to a specific owner are scattered amongst hundreds of transactions and blocks. Therefore, the actual balance of a Bitcoin address is not stored in the network and must be calculated by the wallet software [11].
Since an owners balance consists of a number of UTXO with different values, its often not possible to send a specific amount of bitcoins to a recipient address. Instead, multiple UTXO are consumed as an input of a transaction and the output consists of a new UTXO representing the actual amount sent to the desired recipient and another UTXO representing the change that is sent back to the wallet of the sender [11]. This is illustrated in figure 3.1.

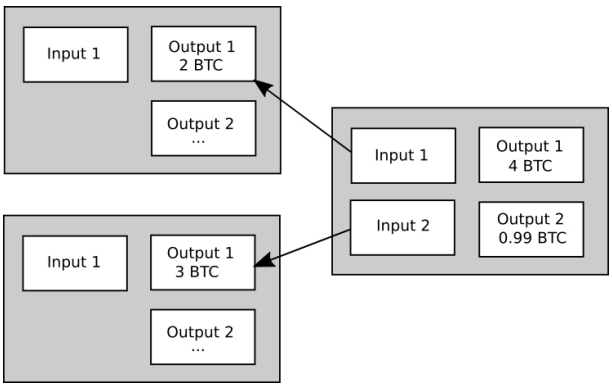

Figure 3.1: Outputs of transactions A and B as inputs of transaction C.

In this example, four bitcoins are sent to another participant. Since the sender does not have an UTXO with the exact value available, she references two previous transaction outputs with values of two and three bitcoins as inputs of transaction C. She then sends four bitcoins to the recipient in output 1 of transaction C. Output 2 is the change directed back to herself. The difference of the input and output values is a transaction fee that is collected by the miners which include this transaction in their blocks. Transaction fees give an incentive for miners to prioritize transactions over others or to include them at all. Transaction fees also prevent abusive transactions to destabilize the network [11].
The UTXO created by a transaction includes a *locking script* that "locks" the UTXO by specifying the conditions needed to spend it a new transaction. The script itself is implemented in a simple stack-based, non-touring complete script language called "Script" [18]. When a transaction is constructed, the sender can decide whether she wants to use the *pay-to-pub-key-hash* (p2pkh) or a *pay-to-script-hash* (p2sh) script to be used. In the first case, the public key hash of the recipient of an UTXO is included in this field and the holder of the associated private key is then considered the owner of the UTXO. This is by far the most common use case [11]. By using the p2sh script, more complex conditions can be specified by combining the operations of the Script language. A common use case for the p2sh is the multisignature scheme, in which multiple key holders have to sign an UTXO to spend it in a transaction [18].
Transaction inputs are pointers to UTXO that reference the transaction hash of the transaction that created them and the index of the UTXO in that transaction. A transaction input also includes the unlocking-script that satisfies the conditions of the locking script. In most cases, this is a signature that proves the ownership of the address in the locking script [18].

3.1.3 The blockchain

The heart of a decentralized cryptocurrency network is the blockchain, an append-only data structure that stores every transaction ever executed in the network. Every block in the blockchain contains the hash of its predecessor. This creates a chain of blocks reaching back to the first block, the genesis block. Because subsequent blocks depend on each other, it is not feasible to change a block retrospectively when it was present for a longer time in the chain. This makes it very hard to conduct double-spending attacks (Section 3.1.4) [16].

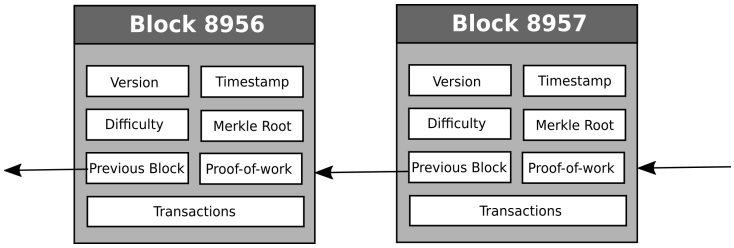

Figure 3.2: The structure of a block in the blockchain.

Figure 3.2 shows the structure of a single block in the blockchain. Besides the transactions that took place in a time frame of about 10 minutes, it also contains a header

with metadata. The header contains the proof-of-work (section 3.1.3) that validates the integrity of the block and the root of the Merkle Tree to improve the scalability of the system (section 3.1.3). it further contains the timestamp of its creation, the version of the Bitcoin protocol and the hash of the previous block [16].

Merkle trees

The Merkle Tree is a data structure that stores a digital signature for the whole list of transactions in a block. It is used to verify the integrity of a transaction in an efficient way with the help of a Binary Hash Tree [12].

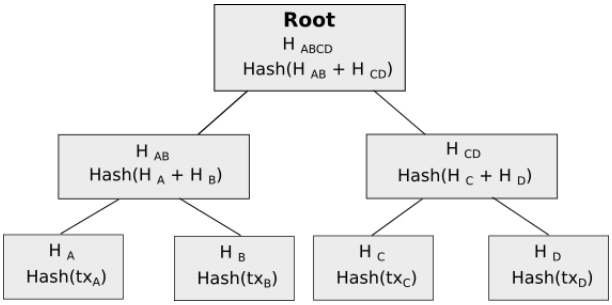

Figure 3.3: The structure of a Merkle Tree.

Figure 3.3 illustrates the structure of the Merkle Tree. Its leaf nodes contain the hash value of the individual transactions of a block. Every non-leaf nodes hash value is the hash of its two children. The resulting root node of this tree is then included in the block that contains the associated transactions [12].
The Merkle Tree allows a node only to download the header of a block and a small number of nodes from the Merkle Tree to validate a transaction. In the example illustrated in figure 3.4, only the nodes H_D, H_{AB} and H_{EFGH} are required to proof the inclusion of the transaction H_C in the block. This verification takes on average $\mathcal{O}(\log n)$ time, and at most $\mathcal{O}(n)$ time because the structure is the same as in a binary search tree [12].
If an attacker smuggles in an invalid transaction somewhere at the bottom of the tree, the hash of that transaction propagates upward to the root node and makes the hash of the whole block invalid [27].

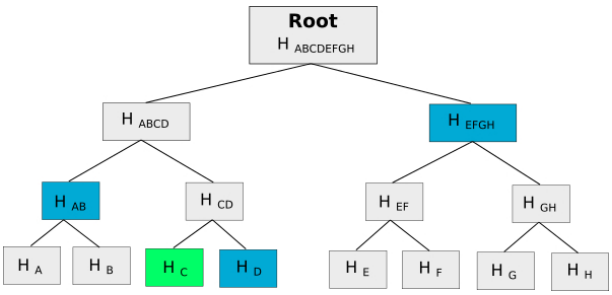

Figure 3.4: Validation of a transaction in a Merkle Tree.

The ability to download and validate only a part of a block to validate individual transactions is important for the sustainability of the network. A full node storing and processing all transactions of every block consumes 15GB of disk space as of April 2014 and grows

by 1GB per month. It is not feasible for a mobile device to store such amounts of data [27].
A protocol known as Simplified Payment Verification (SPV) enables the so-called "light nodes" or SPV nodes. SPV nodes only download a blocks header and the part of the Merkle Tree relevant to proof the inclusion of a transaction. In a similar way, SPV nodes proof the inclusion of a block in the blockchain by validating the blocks header. The combination of theses validation processes allows an SPV node to proof that a transaction belongs to the blockchain by downloading less than 1 kilobyte of data [12].

Proof of work

Every mining node in the network accepts all transactions that took place since the creation of the last block. A mining node then includes the transactions in a new block and tries to validate it by repeatedly generating a random integer (nonce) until the hash of this integer together with the block header yields a value that is smaller than a predefined target. The target value is adjusted dynamically and depends on the total hashing power of the network. It is adjusted such that a block is created every 10 minutes on average. The mining process makes it hard for a single node to manipulate the state of the network unless it controls the majority of the mining power [20] [27].
The first node that manages to compute the nonce is rewarded with the transaction fees and the coinbase reward. At the genesis node, this coinbase reward was 50 bitcoins, but every 210'000 blocks the reward is halfed. The coinbase reward is the only source of new currency and thus, the supply of bitcoins is limited to 21 million [13].
When a mining node finds a nonce, it adds it to the block header and broadcasts the block to all nodes in the network. Since the broadcasting process takes some time, it is possible that two or more miners find a solution for the same block and propagate it in the network. This creates a fork in the blockchain and for a short time, multiple chains can coexist. When this happens, new blocks are added to the longest chain by honest miners [16].

3.1.4 Consensus attacks

As mentioned in section 3.2.3, it is almost impossible to change a block in the blockchain after a certain time ,because all subsequent blocks would also have to be mined again. However, it is still possible for an attacker with a high hashing power to insert invalid transactions in blocks in the near future or to deny service to specific participants of the network. Such attacks are referred as 51 percent attacks, because an attacker needs to have at least 51 percent of the hashing power in order for an attack to almost certainly succeed [13].
The first case is called a double spending attack. In such an attack, the attacker tries to insert transactions containing UTXO of her address multiple times. In this scenario, the attacker spends an UTXO in a transaction and waits until the transaction is included in a block. The attacker then forks the blockchain below this block by mining a new block in which she spends the same UTXO again in another transaction to an address that is owned by her. She then tries to create more blocks than the rest of the miners to create a

chain that is longer than the chain containing the original transaction. If she manages to do that, the original transaction is considered invalid because the miners will always work on the longest chain. To protect against such an attack, the recipient of a transaction should wait for at least 6 confirmations until she accepts the payment. The longer she waits, the more unlikey it is that a 51 percent attack succeeds [13] [27].
In the second case the attacker can ignore the transactions to a specific Bitcoin address by not including them in her blocks. When another miner includes transactions for the address, she can fork the blockchain and invalidate these blocks as long as she has the majority of the mining power in the network. [13]
A majority attack is also possible with less than 51 percent of the hashing power, but the probability that an attack succeeds is reduced. Although it is not possible for a single miner to control even 1 percent of the network, centralized mining pools offer a high risk for for-profit attacks by a mining pool operator [13].

3.1.5 Limitations of Bitcoin

In addition to simple monetary transactions, cryptocurrency networks can in theory be used to store arbitrary data and perform arbitrary state transitions and thus enable more complex and powerful applications such as Smart Contracts or Distributed Anonymous Organizations (DAOs) [27].
There have been several attempts to retrofit the simple scripting language of Bitcoin to implement various smart contracts. However, since this scripting system has a low expressiveness and is not touring complete, many tasks are computationally very expensive to perform or not even possible. It is, therefore, preferable to use a general purpose smart contract scripting language. The Ethereum Virtual Machine (EVM), which is used in this application, is the first system that incorporates such capabilities [31].

3.2 Ethereum

Ethereum is a distributed computing platform that uses a blockchain to store not only the state of user accounts, but also program code and its associated state. It allows for the distributed execution of arbitrary code and provides a suitable platform for the development of smart contracts discussed further in section 3.3. Ethereum was specifically designed to allow anyone to write smart contracts and decentralized applications. It supports several scripting languages that can be compiled to byte code that is executed on the Ethereum Virtual Machine (EVM). [27][37].
As of May 2017, Ethereum is the second largest cryptocurrency with a market capitalization of over 8 billion US-Dollars [30].

The following subsections discuss the most important aspects of Ethereum.

3.2.1 Accounts

An account is an object that stores the state of a users account balance or of a contract. The first account is called an externally owned account (EOA), because it belongs to an external entity and is controlled by a private key. It can send messages by creating and signing transactions with its private key [27][37].
The second account is called a contract account and its state is controlled by the contract code. When a contract account receives a message, its code is executed. It can also send messages to other contracts or create new contracts [27][37].
The EVM does not differentiate between the 2 account types. Every account has a key-value store called a storage and a balance in wei that can be changed by sending transactions that include ether [37].

3.2.2 Transactions and messages

Transactions A transaction is a data package that is signed by an externally owned account. It contains the sender of the transaction, the recipient of the transaction, the amount of ether sent, an optional data field, a gas limit and a gas price field [27].

Messages A message is like a transaction that is sent from one contract to another contract. It behaves similar to an ordinary function call in other programming languages. Every time the running code of a contract account executes the "call" opcode, a message is generated and sent to the recipient contract or externally owned account. Both, transactions and calls, can be used to create new contracts, to invoke functions of a contract or to transfer ether to a contract or to an externally owned account [27] [37].

Ether Ether is the currency token used by Ethereum to pay for transaction fees. Developers have to provide ether when they deploy contract code to the blockchain and users have to spend or burn ether when they invoke transactions on a contract. Ether can be exchanged against other fiat currencies via various cryptocurrency exchanges [27][37].
Ether can be divided further into smaller units, the smallest among them is called wei. One ether is equal to 10^{18} wei [27][37].

Gas Gas is used to pay transaction fees in Ethereum. Gas is not a currency, but an internal pricing unit that decouples the market price of the ether from the cost of transactions. 1 unit of gas represents the price for the most simple operation executed on the EVM. The price for one unit of gas, the gas price, can be dynamically adjusted to adapt to fluctuating values of the ether currency. The gas price can be defined by the originator of a transaction and miners decide whether they want to include the transactions in their blocks or not. [37].
The gas limit or startgas value of a transaction determines how many computational steps a transaction is allowed to execute. The more lines of code a contract executes and the more memory and storage it uses, the higher the gas limit of the initial transaction needs

to be [37].
Transaction fees and the gas limit help to prevent the execution of faulty code, like infinite loops and they help to save computational resources on the network. The gas limit also disincentivises potential denial-of-service attacks on the network [27][37].

3.2.3 Blockchain and mining

The Ethereum blockchain is very similar to Bitcoin. The most important difference is the fact, that a block not only contains a list of transactions but also the whole state of the network. The state is stored in a data structure called "Patricia Tree" [27].
The Patricia Tree is a modified Merkle Tree that is optimized for the insertion and deletion of nodes. It stores the state of all contract and externally owned accounts. Every block stores a reference to the root of the tree and updates only the parts that changed because of the effects of the transactions in that block. This allows new nodes to only download the Patricia Tree instead of all blocks to retrieve the state of all accounts and therefore saves a considerable amount of disk space. It is estimated that if this concept would be applied to Bitcoin, it would require a node to store between 5 and 20 times less data [27].
Ethereum also uses a different Proof-of-work algorithm, called Ethash, that produces a block every 12 seconds in average compared to 10 minutes in Bitcoin. This has the advantage, that transactions can be processed faster and a recipient of a transaction does not have to wait long until she can consider a transaction to be safe. It also increases the interactivity of applications that interact with contracts on the blockchain. Further, Ethash is memory hard and therefore ASIC resistant [37].
A negative effect of the fast block time is that the stale rate, the rate at which blocks that are not part of the main chain are produced, is increased. This is a security risk that can lead to centralization to mining pools since many miners will not be rewarded for their effort in mining new blocks. While in Bitcoin, such blocks are considered "orphan" and are no longer used, the GHOST protocol of Ethereum also allows for the inclusion of such "uncle" blocks and rewards the miners of them [27].
In contrast to Bitcoin, the mining reward for a block is static and exactly 5.0 ether. Successful miners also collect all gas that is used in the transactions of a block. Miners of "uncle" blocks receive 7/8 of the static block reward [37].
The total amount of ether issued in a year is statically bounded to 1/3 of the pre-sale, which is approximately 18 million ether. It is estimated that approximately 1% of the total monetary base is lost every year due to the death of key owners, lost of private keys or transactions to empty addresses. Therefore the supply of ether grows at a disinflationary rate until the rate of annual loss and destruction of ether will balance the rate of issuance and the currency no longer grows [27].

3.2.4 Light clients

A protocol for fully or partially light clients in the Ethereum network is still under development As of April 2017. However, fully and partially light clients will play an important role in the future.

A partially light client is a client that verifies every block but stores almost nothing on its hard disk. Instead it uses DHT GET requests to access blocks from other nodes and validates them locally. The goal is to use almost only partially light clients in the future that store nothing but the last few thousand blocks [35].
A fully light client does not process most transactions. It can access transactions and states by recursively downloading nodes from the Merkle Tree of a block or from the Patricia Tree of the network. For example, if a light client wants to know the state of an account, it requests the root of the Patricia Tree from a source and then downloads nodes recursively until it arrives at the desired value. If it wants to check for a specific transaction, it can request the block number and index for it and recursively search for the transaction in the Merkle Tree of that block. Fully light clients will also be able to collectively validate blocks and watch for specific events that took place on the blockchain. The latter functionality is useful to observe the state of a contract on the blockchain. All operations except the event watching have a complexity of $\mathcal{O}(\log n)$, which makes them highly efficient and suitable for smart phones and other hardware constraint devices [35].

3.3 Ethereum Smart contracts

A smart contract is a special account that stores executable code together with its associated data and an account balance on the blockchain. Smart contracts have an address (a public key) and are created by transactions. Transactions are also used to interact with a contract on the blockchain by sending money to its account balance or by executing code. To execute the code of a contract, a function call containing the functions name and its parameters is binary encoded and sent to the contract in the data field of the transaction [31] [37].

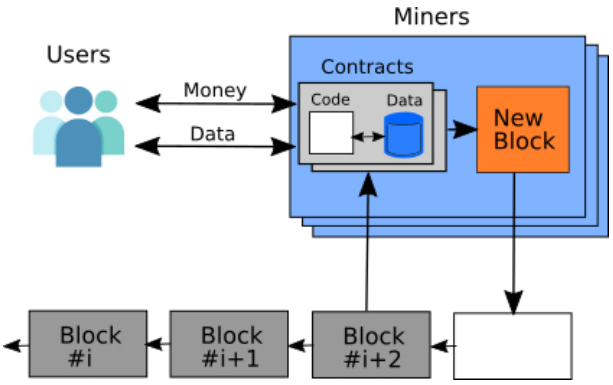

Figure 3.5: Execution of a smart contract on the blockchain.

Figure 3.5 illustrates the interaction of externally owned accounts (users) with a contract. Every time a contract receives a message from another contract or a transaction from a user, it can receive ether or execute a function that is specified in the data field. In the same way, the contract can send money from its balance to other accounts or execute functions on other contracts through broadcasting of messages [31].
Execution of the code takes place on all mining nodes in the network concurrently which reach consensus over the new state of the contract using a proof-of-work algorithm. The persistent variables of a contract are stored on the storage, a key-value store associated with the contract that is persisted on the blockchain. Access to the storage is very expensive (20000 units of gas per 256 bit word) because it has to be stored on every full node in the network. Intermediate results of computations are stored in the memory, a non-persistent byte-array. The state as well as the code of a contract are public and the code of a contract cannot change retrospectively [27] [37].

3.3.1 Solidity

Ethereum supports different script languages for writing smart contract code. The most popular among them is Solidity, a contract oriented high-level language with a static type system that has a syntax very similar to javascript [40].
The main construct in Solidity is the contract. Similar to a class, a contract can contain fields, functions, function modifiers, struct types and enum types and it is also inheritable.[40].
The following subsections briefly describe the most important concepts of a contract in

Solidity and also discuss some of the pitfalls and security issues that can arise when writing smart contracts in Solidity.

Variables and data types

State variables hold values permanently stored on the contract storage. Since Solidity is a statically typed language, the data type of a variable must be specified. Solidity provides the following elementary types that can be combined to form more complex types [50]:

- **Booleans:** stores a boolean value (true, false)
- **Integers:** Includes different subtypes of signed (int) and unsigned(uint) integers with different byte lengths.
- **Address:** represents a 20 byte Ethereum address.
- **fixed-size byte arrays:** includes byte arrays from length 1 to 32.
- **dynamically-sized arrays:** this includes the "bytes" type, a dynamically-sized byte array and the string type, a dynamically sized UTF8-encoded string.

Solidity provides some special functions and variables that always exist in the global namespace. They are used to provide information about the state of the blockchain or the contract. The most important variable used in a contract is the "msg" object that provides information about the message sent to a contract in a function call such as the sender of the message, the remaining gas for a message call or the value sent with the message [45].

Function calls

Functions are the executable units of a contract and can either be called internally or externally. An internal function call is a call to a function of the currently executing contract. An external function call is a call to a function in an external contract instance [44].
An external function call uses the message call directive of the EVM that expects the name of the function and its arguments in a hashed format. When a contract wants to call a function of another contract, it can specify the amount of ether it wants to send and the amount of gas it wants to spend in the *value* and *gas* functions of the function object [44].
The following example illustrates how the contract *Consumer* invokes functions or sends ether to another contract *InfoFeed* using 4 different methods in its *callFeed* function:

Listing 3.1: An external function call

```
contract InfoFeed {
    function info() payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(address addr) { feed = InfoFeed(addr); }
    function callFeed() {
   feed.info.value(10).gas(800)();
        feed.call.value(10).gas(800)(bytes4(sha3("info()")));
        feed.delegatecall(bytes4(sha3("info()")));
   feed.send(10);
    }
}
```

The first call is a direct call of the function and the second call uses the *call* directive to invoke it. In the second case, the function of the callee must be specified by its hash signature. The *delegatecall* is similar to the *call* but the invocation of the called function is executed in the callers environment. This means that the variable *this* refers to the calling contract. This is also the reason why no value is transferred and no gas usage is specified for the *delegatecall*[15]. The *send* function does not invoke the *info* function of the *InfoFeed* contract but will just send 10 wei to the balance of the contract. However, the *send* function invokes the fallback function of the called contract what can lead to unexpected behavior as explained in section 3.3.1. The 3 latter methods also differ in their exception behavior further described in section 3.3.1.
In this example, the calling contract explicitly uses the *InfoFeed* interface to "cast" the address *addr* to an *InfoFeed* contract. However, it is not guaranteed that the contract at the provided address actually belongs to an *InfoFeed* contract or to any contract at all [44] . The behavior of a call in case of type mismatch can vary:

- if the address of the callee is not a contract address, an exception is thrown (since v0.4.0)
- if the address belongs to a contract that has a function with the same signature, then the function is executed.
- if the address belongs to a contract that has no matching function, its fallback function is executed.

Interaction with other contracts can be dangerous since their implementations are not specified by their interface. The called contract can execute arbitrary instructions and can therefore cause serious security risks [44].

The fallback function

When a Solidity contract is compiled into EVM bytecode, its functions are dispatched at the beginning of the compiled contract and identified by a signature derived from the functions name and its arguments. When a function is invoked, this signature is provided in the message call. If no matching function can be found or no signature is provided at all, the fallback function of a contract is invoked instead [15].
The fallback function has no arguments and cannot return anything. If a contract receives an empty signature together with ether but does not define a fallback function, it will throw an exception (since Solidity v0.4.0). The fallback function should not contain instructions that use a lot of gas because this can also cause an exception when the *send* function is used to transmit ether. In this case only 2300 gas is available and the contract will not be able to receive ether when more gas is used by the fallback function [43][15].
The fact that the fallback function can be executed unexpectedly exposes a security vulnerability that can be exploited by attackers to execute a non-recursive function in a loop. This is referred to as "reentrancy" and illustrated in the following example.

Listing 3.2: Contract Bob

```
contract Bob {

bool sent = false;
function ping(address c)
{
  if (!sent) {
    c.call.value(2)();
    sent = true;
  }
}

}
```

Listing 3.3: Contract Mallory using contract Bob

```
contract Bob { function ping(); }

contract Mallory {

function() {
  Bob(msg.sender).ping(this);
}

}
```

First Bob sends 2 wei to Mallory in its *ping* function by using the empty call directive. This will then invoke the fallback function of Mallory which executes the *ping* function of Bob again. Since the call to the fallback function of Bob has not returned, the condition "if(!sent)" is still true and Bob receives another 2 wei. This loop ends when either the execution runs out of gas or the executions stack limit is reached or when Bob has no

more ether in its balance [15].
To prevent security vulnerabilities that arise from re-entrant code, a contract should always first check the conditions necessary to perform an action. Then the local state variables that have changed due to the action should be assigned before making any calls that transfer ether. This paradigm is also called "Checks-Effects-Interactions". In the example above, the instruction "sent = true" should be before the instruction "c.call.value(2)()" [49].

Exception handling

If an account does not contain code or a function throws an exception or the called contract runs out of gas, a function call will cause an exception. An exception will stop the execution of the current function and will revert all changes to the state and balance of the contract caused by it. The exception will then recursively "bubble up" the call chain [42].
At the moment, catching exceptions is not yet possible and the only side effect noticed by a caller is that the transaction runs out of gas. Another problem is that when using the *send* function of a contract or one of the low-level functions *call* or *delegatecall*, the state of a contract is not reverted but instead these functions will return the boolean value "false" [42]. These irregularities in the way exceptions are handled in Solidity can affect the security of contracts. For example when a contract does not check the return value of the *send* function, it could wrongly assume that the transaction was successful because no exception was thrown [15].

Visibility of functions and state variables

There are four different types of visibilities for functions and variables in Solidity:

- **public:** public functions can be called either internally by this class or by derived classes and externally by a message call. For public variables, an automatic getter function is created [47].
- **private:** private functions and state variables are only visible internally by the declaring contract and not by derived contracts [47].
- **external:** external functions behave like public function with the difference that they cannot be called inside the contract by invoking "f()", but by invoking "this.f()" [47].
- **internal:** internal functions and variables behave like private ones with the difference that they cannot be called with the "this" parameter inside the contract [47].

It is important to note, that even if a state variable or function is marked "private", that does not mean, that it cannot be seen from outside the blockchain. It only means that

a variable or function cannot be accessed by another contract. Since all transactions in the blockchain are visible for everyone, the state and the functionality of every contract is also visible for everyone [47][15].

Function modifiers

Function modifiers can be used to change the behavior of a function. They check that a function can only be executed in a specific context. For example a function can have a modifier that checks that the caller must have a specific address. A special modifier is the "payable" modifier that specifies that a function can receive ether. If ether is sent to a function that lacks this modifier, the function will throw an exception [46].

Events

Another concept of Solidity are events. Events are interfaces that allow the usage of the EVM logging facilities. These interfaces are used by applications to subscribe to events of a contract and to monitor the state of a contract without interacting with it directly [41].
When an event is called, its arguments are stored in the transaction log data structure of the blockchain, which is not accessible from within a contract directly. The hash of the signature of an event is used for identification and is called a topic. To allow the search for specific argument values of a topic, up to three parameters of an event can receive the "indexed" attribute [41].

3.3.2 Security considerations

This section discusses general Ethereum specific security vulnerabilities that can arise when working with smart contracts and possible ways a programmer can prevent them or limit their impact.

Call-stack limit

Whenever a contract invokes a function of another contract or of itself, the associated call stack of the transaction grows. The maximum depth the call stack can reach is 1024 frames. Until October 18th 2016 it was possible to exploit this fact by generating an almost-full call stack and then invoking a victims function. This would then cause unexpected behaviour that has been exploited together with other vulnerabilities [15].
This problem was then addressed by a hard-fork of the Ethereum blockchain that changed the way gas consumption for the "call" and "delegatecall" is computed. After the fork, the caller can not use more than 63/64 of its gas and can therefore not reach a call stack depth near 1024 [15].

Keeping secrets in a contract

There are a lot of contracts that need to keep its state secret for a while. For example, multi player games can often not reveal a decision of a player before all other players made their decision because it would give them an advantage. In such cases, the players must use a cryptographic "commitment" to bind its choice without revealing it to other players [31].
In the simplest case, such a commitment can be a hash function of a random number (nonce) and the actual choice of the player. After all players sent their commitment, they open it by invoking a function that accepts the nonce and the choice in plain text [31].

Immutability of contracts

Once a contract is deployed, its code cannot be altered any more. This means if a contract exposes any security vulnerabilities, there is no way to directly patch it [15].
A way to limit the impact of the immutability is to keep the architecture of a system as modular as possible and to build in some kind of "maintenance" function that can be used by the owner of a contract to replace faulty implementations of its dependencies [49].

Unpredictable state

Since the state of a contract can be updated in every block and transactions in a block can be grouped arbitrarily by a miner, a caller can not know for sure in which state the contract is when sending a transaction. The actual state of a contract can also be unclear when the blockchain forks. It is possible that a contract is updated on a branch of the blockchain that turns out to be the shortest one at a later stage. The changes made to the contract would then be reverted [15].
The fact that the state of a contract is not always known to a caller can be exploited by adversaries. Atzei [15] describes a case in which a library provided by an adversary was used to steal all money from contracts that used that library in their code.

Generating randomness

To generate random numbers contracts often use the hash or the timestamp of a block that appears at a given time in the future, since this value will be the same for everyone in the network. Since miners can control the content and orders of blocks, a malicious miner controlling only a minority of the network could bias the outcome of the random process [15].
A proposed solution for this attack is the incorporation of timed commitment protocols. The process is similar to the commitment process in an online game. All parties send their commitment together with a deposit. Later, participants open their commitments and reveal their secrets. The random number is then computed from the revealed secrets. If a participant doesn't reveal her secret, she loses her deposit [15].

Time constraints

When a contract relies on block timestamps to implement time constraints, a malicious miner interested in manipulating the behaviour of a contract could try to manipulate the timestamp of her blocks to a certain degree [15].

Chapter 4

Legal aspects and Identity Management

Since the contracts created by the application implemented in this thesis should be valid and accepted by law, it is necessary to have a closer look into contract law and define the requirements that a purchase contract has to fulfil to be legally accepted. Further a purchase contract should no only be legal, but also enforceable by law. This poses the question about the required properties a purchase contract must have to offer legal security.
The first part of this chapter discusses and finally answers these questions in the context of Swiss contract law. Section 4.5 describes how electronic contract can be signed in Switzerland using the SuisseId and section 4.6 goes into the details of how digital identities could be managed in the blockchain in future legislations.

4.1 Definition

The law doesn't give a definition of a purchase contract but describes the duties of the involved parties that result out of the conclusion of purchase contract: It obligates the seller to hand out the purchase item and to transfer the property of it to the buyer and it obligates the buyer to pay the purchase price to the seller [26].

4.2 Formation of a contract

The following subsections define the party and contractual elements that have to be considered for the formation of a purchase contract.

4.2.1 Parties

The parties of a purchase contract are the buyer and the seller. The seller has two main duties to fulfil: He is obligated to hand out the purchase item to the buyer and he is obligated to transfer the property right over the purchase item to the buyer [26].
The buyer has the duty to pay the purchase price to the seller. He also must accept the item that is offered to him in the purchase contract. If nothing else is negotiated or commonly accepted, the reception of the item must take place immediately. Further the buyer has the duty to review the purchase item after the reception and notify the seller about possible defects [93].

4.2.2 Conclusion of a purchase contract

The following subsections discuss the most important rules that have to be considered for a purchase contract to be valid.

General rules

For the conclusion of a purchase contract, the general rules on contract conclusions defined in OR 1 ff have to be applied. The most important rules are briefly described here.

Conformable will It is crucial that both parties express their will to conclude the contract. The Obligationenrecht (OR) says that a declaration of intent of both parties is necessary for the conclusion of a contract. It is also mentioned that this declaration can be explicit or by implication. This means that the relationship between the parties may be created orally, in writing or by conduct. For example, when a party accepts a good or a service against payment, this is considered a declaration of intent by conduct [25].
In the online business, the declaration of intent is sent electronically by a click on a button or by sending an e-mail to the seller. However, an offer in an online-shop is usually not considered an offer in legal terms, but an offer to the customer to send a bid to the seller. This means that the offer is made when a customer orders a good or a service. The seller has to accept this offer for a contract to be formed. He can accept this offer either explicitly or by implication by sending the goods to the customer [80].
Alternatively, Art.7 says that the applicant of an offer is not legally bound to it if he adds a disclaimer of warranty to the offer or if such a caveat lies in nature or in the circumstances of an offer [86].

Requirements for validity The purchase contract has to fulfil specific formal requirements for special kinds of contracts (for example when buying real estate). This is discussed in more detail in section 4.3. Further the purchase contract has to conform to the barriers of contractual freedom. This means that the content of the contract can be chosen freely unless it violates the law or it is an agreement against public policy [93].

Articles of agreement For the conclusion of a contract the parties need to agree at least on the essential articles of agreement (Essentialia Negotii). For the purchase contract, both parties must agree on the price and the purchase item for the contract to be valid [26].

4.3 Formal requirements

Most purchase contracts don't require any special formal requirements for their validity. Most purchase contracts that have movable goods as their subject fall into this category. For theses contracts the 2 contractual parties are free to choose whatever form they want. If they define no form it is assumed that parties don't want to be obliged to any. This principle is also referred to as "freedom of form" (Formfreiheit) [93].

The Swiss contract law defines 3 different kinds of formal requirements that a contract can have:

- **Handwritten signature** (Einfach Schriftlichkeit): This form requires a contract that is signed with the handwritten signature of all parties involved in the contract. It is considered the weakest formal requirement that a contract can fulfil. This form is required for purchase contracts involving patents, trade markets or designs [93].
- **Qualified handwritten signature** (Qualifizierte Schriftlichkeit): In addition to the handwritten signature, this form requires additional elements that must be fulfilled, e.g. that a contract must be handwritten from start to end [93].
- **Notarial act** (Öffentliche Beurkundung): A notarial act requires that the declarations of intent of both parties as well as the purchase object itself is written down a special form by a solicitor. This form is required in all purchase contracts involving the purchase of real estate [93].

Formal requirements can fulfil 2 purposes. They fulfil a warning function that protects the parties from precipitation and they fulfil a security function that give the parties law security meaning that the signature of a party can be used to identify a party in front of a court. If no specific form is mentioned in a written contract, it is assumed that the formal requirement of the handwritten signature is used [93].

4.3.1 Electronic signatures

If the law doesn't require special formal requirements for a contract, it can also be concluded electronically. if there are prescriptions by law it can be difficult to fulfil them in electronic form. In some cases, for example when a notarial act or qualified handwritten signature is required, a purchase contract cannot be concluded electronically. However, when only the handwritten signature is required, an electronic signature can be used under certain conditions [80].

The Swiss contract law defines 3 different kinds of electronic signatures [93]:

- **Electronic signature:** Data in an electronic form that is added or logically connected to other data and can be used to authenticate this data.
- **Advanced electronic signature:** An electronic signature that fulfils the following requirements:
  - It is exclusively connected to the owner
  - It allows the identification of the owner
  - It is created with means that are under full control of the owner
  - It is connected with the associated data in a way that prevents the retroactive manipulation
- **Qualified electronic signature:** An advanced electronic signature that is based on a secure signature authority and on a qualified and valid certificate at the time of the creation.

Only the qualified electronic signature is considered a secure proof in front of a court and has the same authenticity like the handwritten signature under the condition that it is issued by an officially recognized provider of certification services as defined in the federal law for electronic signatures (ZertES) [80] [53].

4.4 Implications

In this section the questions posed at the beginning of this chapter are discussed in the context of the smart contract application described in Chapter 6. In the following subsections the term "smart contract" refers not only to the smart contract code executed on the blockchain but also to its representation in the Android application.

4.4.1 Legal validity

To be legally accepted, the smart contract must be conform with the general rules described in section 4.2.2.
The seller declares its intent by creating a new purchase contract and deploying it on the blockchain. It is mentioned in section 4.2.2 that an offer is not binding if the seller adds a disclaimer of warranty to the offer or if such a caveat lies in the nature or in the circumstances of an offer. The nature of the offer in the smart contract is very different from an offer in an online shop. In a conventional online shop, the seller does not have to accept an offer from a customer and the buyer only has to pay the purchase price if the seller confirmed the purchase.
In contrast, the smart contract code allows only one buyer per offer and the account of the buyer is immediately charged after clicking on the "buy" button. Further, the smart contract is locked after the payment of the buyer and neither party can withdraw its money any more. In the opinion of the author, one could argue that such an offer is binding by

nature because the smart contract code does not allow the withdrawal of payment after an offer is accepted by the buyer.
The articles of agreement are fulfilled if the price and the purchase item are defined properly. Since the smart contract requires the seller to specify the price in a common currency the first condition is satisfied. Regarding the definition of the purchase item, the law does not specify in which form and how detailed this description must be. The author, therefore, assumes that a written description of the purchase item and its properties together with one or more pictures of it is sufficient.
Regarding the formal requirements it can be stated that every smart contract is valid as long as its content does not require the contract to fulfil special formal properties. This means that for example contracts involving the purchase of real estate, patents, designs or trade markets cannot be concluded with the smart contract.

The author concludes that smart contracts generated by the implemented application are legally valid under the following conditions:

- The content of the smart contract does not violate the law and is not an agreement against public policy
- The content of the smart contract does not have special formal requirements by law

4.4.2 Legal security

As mentioned in section 4.3.1, only the qualified electronic signature offers the same authenticity like a ahndwritten signature and can be used as definite proof of the identity of a party. The federal law about the electronic signature (ZertES) states that the qualified electronic signatures can only be issued by providers with a certified public key infrastructure [53]. Section 4.5 discusses how a Document Signing Service (DSS) of such a provider could be integrated in the application.
Because of the high costs and implementation efforts, SuisseId is not integrated in the application at the moment and therefore, legal security can not be provided. However, in future legislations, blockchain technology could be used to manage identities in a more decentralized manner. Section 4.6 discusses this issue in more detail.

4.5 Signing documents with the SuisseId qualified signature

The only solution to provide the same authenticity as in a handwritten signature in the current legal framework in Switzerland is to use the SuisseId qualified signature of a certified public key infrastructure provider. According to a list published by the Swiss government, there are currently 4 different providers that can issue and store electronic certificates that satisfy the law for electronic signatures (ZertES) [76]. All of them require a user to buy a SuisseId USB stick or chip card that stores the Qualified Signature (QC) that can be used to sign documents and an Authentication (IAC) that can be used to authenticate a person or entity in an application [84].
Because a mobile device usually has no USB port to read in the QC, an online Document Signature Service (DSS) can be used to sign documents. The SwissSign AG offers a mobile signing service that allows owners of a SuisseId to authenticate themselves with a password on registered service providers [83].
According to SwissSign it is only possible to sign documents over a registered service provider when using the mobile signing service. SwissSign itself provides the Multi Signing platform for online signing services, which is still in the beta phase as of August 2017 [85]. With the Multi Signing platform, the owner of a SuisseId can authenticate herself, upload a PDF document and sign it. She can also add the e-mail address of other business partners and send them the document for signing.
Integrating the SuisseId qualified electronic signature in a smart contract would involve the following steps:

- Creating a PDF document from a contract either before or after the deployment.
- Signing the PDF document by both parties using SwissSign Multi Signing or a similar platform.
- Calculating the hash of the signed document and adding it in the constructor of the contract or attaching it after the deployment in a separate transaction.

Using SuisseId to sign a contract has the benefit that both parties of a contract can be clearly identified. This gives additional legal security but also enables the conclusion of contracts that require a handwritten signature.
The drawback of this method is that both parties need a SuisseId and that it can only be used to conclude contracts with Swiss citizens. It also relies on a centralized infrastructure which should not be necessary in an application that uses a distributed ledger.

4.6 Identity management in the blockchain

Today private and public entities store massive amount private user data. This data is used to personalize services, optimize the corporate decision-making process, predict future trends and more. Numerous hacks and security breaches in the past raised privacy

issues and question today's centralized data collection. The general terms of organizations that process customer data are often non-transparent to an individual and it is not clear who has access to the data and for which purpose the data is used. It is also difficult for an individual to revoke access to personal data because it no longer has control of the data [101].
Blockchain technology could be used in the future for individuals to manage their private data. The public ledger provides information that no distinct entity owns and the decentralized network of peers that take part in the mining process guarantee a high validity of the data. To securely store the private data of an individual, the data is encrypted and either stored on the individuals device or in a storage layer on top of the blockchain. With the help of public key cryptography, the blockchain can then be used by the individual to grant or revoke access to processors and TTPs can use the blockchain to certify the individuals data by signing it with their private keys [62].
The following subsections discuss a theoretical and a practical approach of Identity Management in the blockchain.

4.6.1 Decentralizing Privacy

A possible solution for managing private data in the blockchain is proposed by Nathan et all [101]. In their paper they describe a system that allows a user to give fine grained access rights for specific personal data to service providers.
They propose to introduce 2 new kinds of transactions on the blockchain: the T_{Access} transaction for access control management and the T_{Data} transaction that is used for data storage and retrieval. When the user wants to give access to a service, a new shared identity (user, service) is created and sent to the blockchain in the T_{Access} transaction. Private data from the users phone is then encrypted using a shared encryption key and sent to the blockchain in a T_{Data} transaction.
The actual data is not stored in the blockchain but in a decentralized key-value store based on the Kademilia distributed hashtable (DHT) routing protocol. Each storage node also has a database to persist encrypted data and an interface to the blockchain.
The shared identity that is created when a user gives access to a service is composed of the signing key - pairs of the owner (the user) and the guest (the service) and a symmetric encryption key. These keys are exchanged through a secure channel between the user and the service. A user can then add or revoke policies together with the signing keys on the blockchain in a T_{Access} transaction. When the service wants to access data in a T_{Data} transaction, the storage node verifies the service by its signature and checks that the policy is defined by the user.
The authors of the paper conclude that their platform enables trust-less access control that gives the user full control over her data and that companies can focus utilizing data without being overly concerned about security issues. Further it is argued that regulatory decisions about collecting, storing and sharing sensitive data would be simpler in a decentralized platform and that law and regulations could be programmed into the blockchain itself in the future.

4.6.2 ShoCard

A practical example of digital Identity Management in the blockchain is the ShoCard app [81] that is currently in a demo phase.
With the ShoCard app, users can scan in their ID cards together with a selfie of their face. This personal data is then encrypted and stored on the users phone and on a ShoStore Server. The blockchain only stores the hash signature of the personal data that is signed with the private key of the user. The ShoCard app can be used in the public sector and in different industries to verify a users identity without storing personal data.
One use case of the ShoCard app is the SITA Traveler App [82] used to identify travellers across different airports and hotels.
When using the Traveller app, the user goes first through a registration process in which an Airline Agent first verifies the ID card of the user and takes an image of her. If all information matches, the agent makes a secure request through the agent's ShoCard App to the airline server requesting a certification of the traveller and the generation of a new travel token. The certification information is then stored on the blockchain and the traveler app receives the travel token and creates an encrypted envelope that contains a pointer to the traveller's ShoCardID and seal record, a copy of the selfie image, the travel token and a pointer to the certification records. This envelope is then stored on the ShoStore Server for later retrieval.
To identify the user without network connectivity, The app generates a QR code that contains the Envelope Id and the symmetric pass-code used for decryption. Another agent can then scan the QR code with the agent app and retrieve the envelope. She then validates the information in the envelope and also makes a picture of the user that is than compared to the selfie stored in the envelope to check that the person holding the token is the same person who was granted the token. If everything matches, the user can proceed through the checkpoint.
The authors claim to the conclusion that the Digital identity has many benefits. It gives authorities the ability to certify and revoke certifications in real-time, which is not possible with physical passports. It further gives the user an unified travel experience since the traveller identification can be extended to other industries outside the airport. The traveller identity enhances the security but does not compromise the users privacy, because the user controls her private data. The authors further claim that the traveller identity creates a web of trust for the user that is strengthened over time giving them preference treatment and freedom of travel. It is argued that a users credentials strengthen when she travels with more airlines and that other credentials, such as banking, employment and other certifications can further increase the web of trust over an individual.

4.6.3 Conclusion

In April 2016, the European Union (EU) parliament approved the General Data Protection Regulation (GDPR). The GDPR replaces the older Data Protection Directive 95/46/EC and regulates the way in which the personal data of EU citizens can be managed and processed by third parties. The main goal of the regulation is the improvement of the security of personal data in the EU. It also wants to return the control and management of

personal data and identities to the individual [62]. The regulation talks about how every processor that holds personal data must provide access, consent management, erasure and portability to the user. Each organization managing data for the individual bears the responsibility for upholding those rights [62].
So it remains to be seen, whether blockchain technology finds its way into future data protection legislations but the examples discussed above show that blockchains have the potential to return the ownership of personal data to the user and to provide a more secure, flexible and expandable interface for accessing user data for data processors.

Chapter 5

Device-to-device communication

The following sections describe briefly the most widely distributed wireless technologies that are used by mobile devices today with a focus on data throughput and security capabilities. In the evaluation section (5.4), the technologies are compared to the use case of exchanging user- and contract data between two devices.

5.1 Bluetooth

Bluetooth is a wireless radio system defined in the IEEE 802.15.1 standard. It was designed for short-range communication and is often used to replace cables in peripheral devices like mice, keyboard and printer. Networks in this application context are also known as Wireless Personal Area Networks (WPAN) [71].
The following subsections give a brief overview over the classic Bluetooth protocol stack up to version 3.0 + HS.

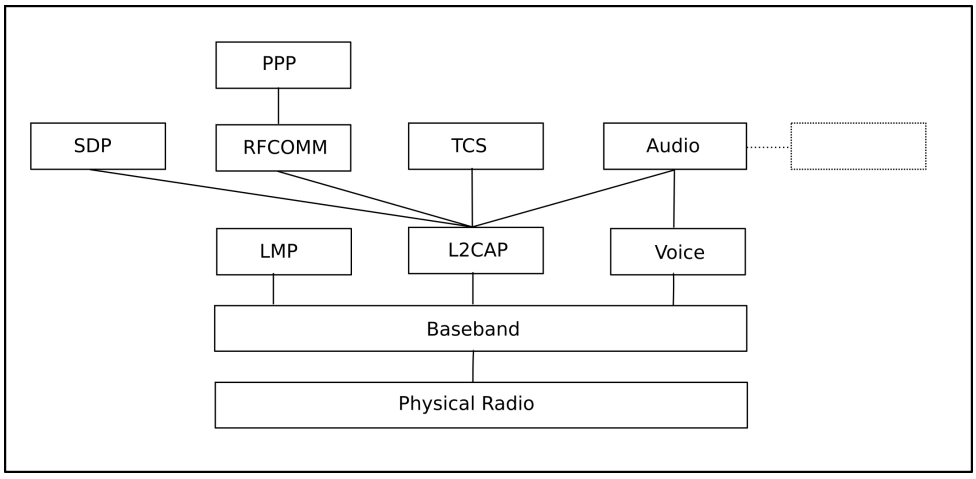

Figure 5.1: Bluetooth protocol stack.

5.1.1 Physical layer

Bluetooth operates in the 2.4 GHz ISM Band. In most countries, 79 channels with a bandwidth of 1 MHz are allocated. In original Bluetooth version 1, only Gaussian shaped Frequency (GFSK) modulation was used with a signal rate of 1 Mb/s. However, with the need for higher data rates, the optional Enhanced Data Rate (EDR) mode was introduced in Bluetooth 2.0. By using a combination of GFSK and Phase Shift Keying modulation (PSK) in the two variants DQPSK and DPSK, EDR supports data rates up to 3 Mb/s. In Bluetooth 3.0, an optional High Speed (HS) mode was introduced that allows for data rates up to 24 Mb/s. In this configuration, the Bluetooth link is only used for negotiation and connection establishment, and the actual payload is transferred over an 802.11 link [21].

5.1.2 Media access control and link Management

A simple Bluetooth piconet is formed by a master and up to 7 slave devices. The master can connect to multiple slave devices but a slave can only communicate with the master in a point-to-point fashion. The devices are synchronized over a frequency hopping scheme that is defined by the master. To reduce power consumption, a slave device can also be in a standby, or parked, mode [54]. A simple network formed this way is illustrated in figure 5.2.

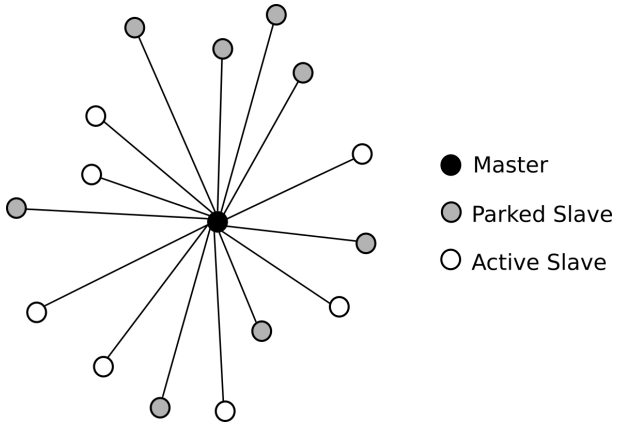

Figure 5.2: Bluetooth piconet structure.

A scatternet is formed when multiple piconets overlap. In this configuration nodes belonging to different networks can communicate when there is at least one node that belongs to two or more piconets. A device can be a slave in multiple piconets at the same time but it can only be a master in at most one [71] [54].
The medium is accessed using the Frequency Hopping Spread Spectrum (FHSS) technique. When a slave joins a piconet, it first sends an *Inquiry* message to the master and learns its address and clock phase. The slave can then compute the hopping sequence that normally changes the channel 1600 times per second. Messages are exchanged during the resulting time slots between master and slave devices. A message can take up to 5 consecutive slots and during multislot messages the channel does not change [54].
The Link Manager is responsible for connection establishment, authentication, security,

quality of service (QoS), power consumption and transmission scheduling [54].
The Link Layer specifies two different types of links: *Asynchronous ConnectionLess links* (ACL) and *Synchronous Connection-Oriented* links (SCO). SCO links offer a connection that guarantees a fixed bit rate of 64 Kb/s and are used for streaming applications (e.g. voice and music streaming) and ACL links are appropriate for non-real-time (datagram) traffic. In contrast to SCO, ACL links provide error protection by means of a 16-bit CRC applied to the payload and optional packet retransmission on error. The Link Manager provides an interface to negotiate quality of service (QoS) parameters like the transfer rate, latency and other parameters for an ACL link between master and slave devices [54].

5.1.3 L2CAP and higher level protocols

The Logical Link Control Adaptation Protocol(L2CAP) protocol provides services to the upper layer protocols. It is responsible for protocol multiplexing from and to the upper level protocols and it segments and reassembles data packets from the upper protocol layers [54].
Although it is possible to directly implement the IP protocol on top of L2CAP, IP is typically implemented using the Point-to-Point Protocol (PPP) on top of the Radio Frequency Communications (RFCOMM) protocol that emulates a serial port [54].

5.1.4 Security

During the evolution of Bluetooth versions, 4 different security modes have been defined for the authentication and encryption of messages:

- **Mode 1**: No security measures
- **Mode 2**: Service-level enforced security
- **Mode 3**: Link-level enforced security
- **Mode 4**: Service-level enforced security mode

Security modes 1-3 are considered legacy and are only used when communicating with devices that implement version 2.0 or lower of the standard. Devices that implement Bluetooth 2.1 or higher only use security mode 4 [59]. Prior to Bluetooth 2.1, a PIN was used as the source of entropy to establish a shared secret between the devices. This procedure did not offer enough protection against passive eavesdropping and brute force attacks on the PIN [59].
Bluetooth 2.1 introduced the SSP pairing procedure which uses Elliptic Curve Diffie-Hellman (ECDH) public key cryptography to calculate the 128-bit link key with the Secure And Fast Encryption Routine + (SAFER+) that is later used to encrypt and authenticate the payload [59].

The ECDH procedure makes passive eavesdropping infeasible. To authenticate the channel and provide additional protection against man-in-the-middle (MITM) attacks, SSP provides four *association models*:

- **Out of band (OOB)**: A channel that cannot be easily intercepted (e.g. Near Field Communication (NFC)) is used to exchange the keys
- **Numeric comparison**: A key inserted by the user on both devices
- **Passkey entry**: like numeric comparison, but used when none or only one device has output capabilities
- **Just works**: The public keys are exchanged without any additional authentication method

When using the "Just works" method, man-in-the-middle (MITM) attacks are still possible [59].

5.2 Bluetooth Low Energy (BLE)

Bluetooth Low Energy (BLE) is an emerging wireless standard designed specifically for low-power consumption and is mainly used in small devices that are highly constraint regarding battery life and processing power and do not require high bandwidth communication (e.g. small sensors and other IoT devices). It was introduced in 2010 as part of the new Bluetooth 4.0 standard and further refined in Bluetooth 4.1 and 4.2 [88].
BLE is not directly compatible with the standard Bluetooth protocol stack but many devices implement the Bluetooth Smart Ready Mode (or Dual-Mode) and are compatible to both classic Bluetooth and BLE [88]. In the following subsections, the most important aspects of the BLE protocol stack are explained in more detail.

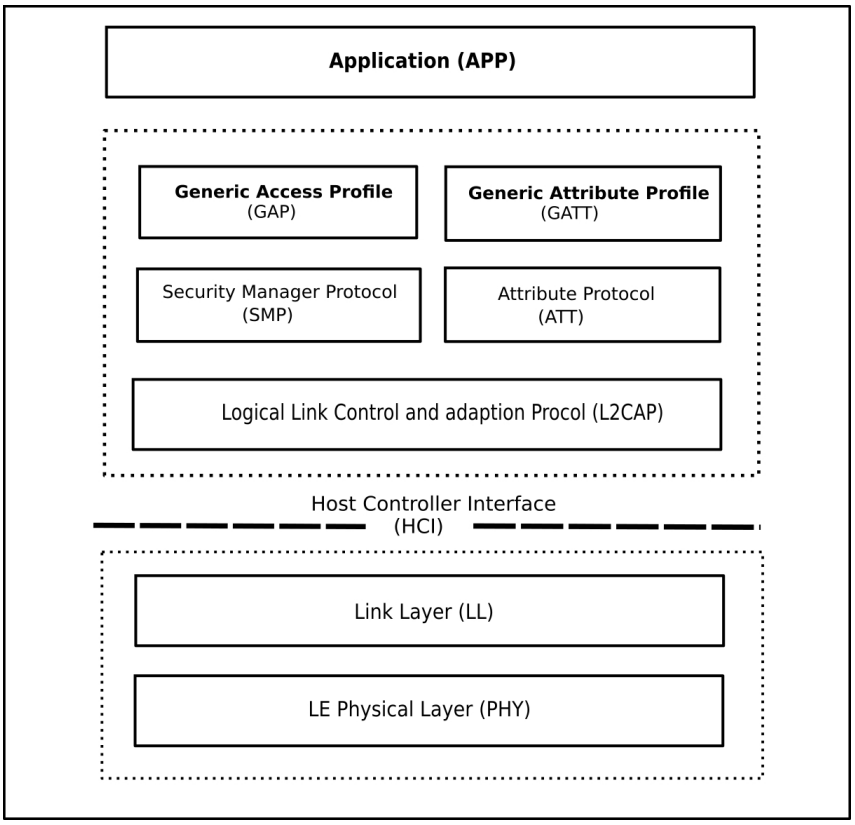

Figure 5.3: Overview of the BLE protocol stack.

5.2.1 Physical layer

Like classic Bluetooth, BLE operates in the 2.4 GHz ISM band and uses 40 radio channels with a bandwidth of 2 MHz. 3 of the 40 channels are used as advertising channels for broadcasting and service discovery. The remaining channels are used for the bidirectional communication between connected devices. To minimize interferences in the 2.4 GHz band, the radio channel is changed periodically using a technique called *frequency hopping spread spectrum*. The modulation used to encode the binary signals is Gaussian Frequency Shift Keying (GFSK) with a modulation rate the is fixed at 1 Mbit/s which is therefore also the upper throughput limit for the BLE standard [89].

5.2.2 Link layer

In a non-connected state, a BLE device can assume the role of an *advertiser* or a *scanner*. The advertiser broadcasts advertising packets through an advertising channel. A scanner is a device that receives advertising packets.To establish a bidirectional communication channel, two devices have to connect to each other. In this scenario, the advertiser uses advertising packets to signalise that it is a connectible device. A device listening to such advertising packets is called an *initiator*. The initiator can create a point-to-point connection by transmitting a connection request message [89].
When a connection is established, the initiator assumes the role of the master device and the advertiser becomes the slave device. A master can be connected to multiple slaves but a slave can only be connected to one master simultaneously. The master controls the frequency hopping scheme that is used and also coordinates the medium access with a Time Division Multiple Access (TDMA) scheme [89].

5.2.3 L2CAP

The Logical Link Control and Adaptation Protocol (L2CAP) is responsible for multiplexing the upper protocol layers and encapsulate them into the standard BLE protocol format. It takes large packets from the upper protocol layers and fragments them into smaller chunks of 27-byte packets. On the other side, it recombines multiple small packets to a large packet and sends it to the appropriate upper layer of the host [89].

5.2.4 Attribute Protocol(ATT)

The Attribute Protocol (ATT) defines how a client and a server communicate with each other. In BLE a device can be a client or a server independently of its Link Layer role. Every server has a set of attributes that contain information that is managed by the GATT. A client can send requests to access the server's attributes or commands to write its attribute values. Requests/Commands trigger a response by the server. A server can also send notifications to the client directly [89].

5.2.5 GATT

The Generic Attribute Profile (GATT) builds on top of the ATT and adds an additional data abstraction layer. It groups attributes into characteristics, pieces of user data with associated metadata. Multiple characteristics are further grouped into services [89] [91].

5.2.6 Security

BLE defines multiple security services at the link layer and at the ATT layer that are group into 2 modes with multiple levels. Mode 1 comprises of 3 levels. In level 1, whether

encryption nor authentication is used. In level 2, the data packets on the link layer are encrypted using a 128-bit AES cypher. ON level 3, a 4-byte Message Integrity Check (MIC) is added to the payload to verify the integrity of a message and encryption is then applied to the payload and the MIC [89] [90].
In security mode 2, BLE can provide data integrity without data encryption. It uses a 12-byte signature to authenticate messages at the ATT layer [90].
When encryption is required, a *pairing* procedure is used to exchange the encryption keys. First the devices generate a short-term key (STK) by one of three methods:

- **Just works**: The keys are exchanged without further authentication mechanisms.
- **Out of band**: Another wireless technology like NFC is used to exchange the keys.
- **Passkey entry**:The user has to enter a key on both devices.

Only the last 2 methods provide protection against man-in-the-middle (MITM) attacks [89].
After the exchange of the STK, the devices exchange up to three 128-bit keys. The Long-term Key (LTK) used for Link-Layer encryption and authentication, the Connection Signature resolving Key(CSRK) to sign data at the ATT level and the Identity Resolving Key (IRK) that can be used to generate private addresses for a public address. Private addresses are used to prevent that a device can be tracked based on its public address [89].

5.2.7 GAP

The Generic Access Profile (GAP) is the highest level protocol of the BLE standard and it defines roles, modes and procedures for the discovery of services and devices, for the connection establishment and for security modes and levels. It defines four different roles that a device can assume at a given time in a BLE network. A device in the *Broadcaster* role only sends out advertising data packets and doesn't take part in any connections. The *Observer* is the counterpart of the Broadcaster and only receives advertising packets. A device in the role of the *Central* can establish multiple connections at a time and is always in the link layer role of the master. It listens to advertising packets and then initiates a connection. A device in the *Peripheral* role is in the slave role of the link layer and waits for connection requests by the central [90].

5.3 WiFi and WiFi-direct

WiFi (Wireless-Fidelity) is a term coined by the Wi-Fi Alliance in stands for the various versions of the IEEE 802.11 wireless protocols. Its aim is to provide high bandwidth wireless connectivity to portable devices in a Wireless Local Area Network (WLAN). The standard defines the physical radio link and the Media Access Control (MAC) layer [54] [63].

5.3.1 Physical layer

All 802.11 standards operate in the ISM frequency bands and use different frequencies and different modulation schemes. The 802.11b standard introduced in July 1999 offered data rates up to 11 Mbps and was using a modulation scheme known as Complementary Code Keying (CCK) as well as supporting Direct-Sequence Spread Spectrum (DSSS). At the same time the 802.11a standard was developed that used Orthogonal Frequency Division Multiplexing (OFDM) and operated in the 5 GHz ISM band. The 802.11g standard was introduced in June 2003 and used the more popular 2.4 GHz band together with OFDM to achieve data rates up to 54 Mbps. It is backward compatible to 802.11b. The latest standard, 802.11n, was released in 2009 and introduced many new features that allow data rates up to 600 Mbps. The most important improvements are Multiple Input Multiple Output (MIMO) Antennas that allow for multiple parallel channel paths between sender and receiver, the support for 40 MHz channel bandwidth and an improved antenna technology. It is backward compatible to 802.11g and 802.11b devices [64] [63].

	802.11A	**802.11B**	**802.11G**	**802.11N**
Approval Date	July 1999	July 1999	June 2003	Oct 2009
Max. data rate	54	11	54	600
Modulation	OFDM	CCK or DSSS	CCK, DSSS, or OFDM	CCK, DSSS, or OFDM
RF Band	5	2.4	2.4	2.4 or 5
Number of spatial streams	1	1	1	1,2,3 or 4
Channel width (MHz)	20	20	20	20 or 40

Table 5.1: Overview of different IEEE 802.11 standards

5.3.2 Network topology

A Wi-Fi WLAN comprises of cells called *Basic Service Set* (BSS) that are composed of portable or fixed stations. The most simple BSS is a BSS that connects at least two devices in an ad-hoc topology, called *Independent Basic Service Set* (IBSS). BSS that are connected in infrastructure mode via an *Access Point* (AP) form an *Extended Service Set* (ESS). Access Points are connected with each other and form a *Distribution System* which can use any communication protocol (e.g. an Ethernet network) [54] [71].

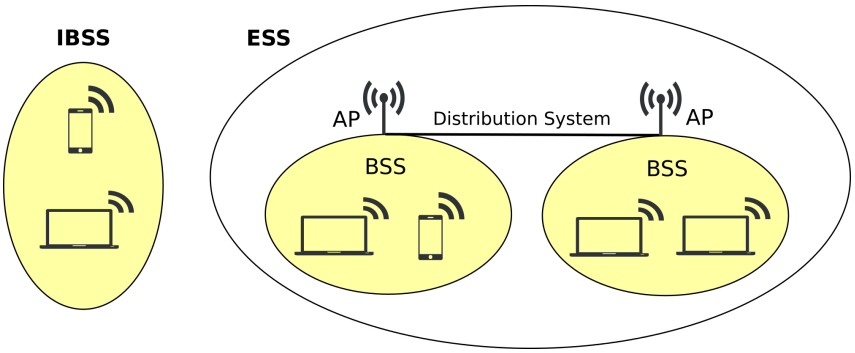

Figure 5.4: Wi-Fi network topologies.

5.3.3 Media access

The fundamental Wi-Fi MAC protocol used to access to access the shared radio channel is called *Distributed Coordination Function* (DCF). The channel is accessed with a CSMA/CA (Carrier Sense Multiple Access / Collision Avoidance) scheme. In its basic version, a station that wants to send data waits for a short time, called Inter Frame Space(IFS) and transmits one frame. If the sender does not get an acknowledgement (ack) of the receiver, it retransmit the frame up to a maximum number of time frames before giving up. A variation to this algorithm is the RTS/CTS (request to send/clear to send) algorithm, in which the sending station first transmits an RTS message and the receiving station responds with a CTS message when the channel is free. This reduces collisions since a station that is not in the range of the sender but the receiver will not send when it notices the CTS from the receiver [54].
Wi-Fi also defines an optional Access Mode called *Point Coordination Function* (PCF) that can only be used in infrastructure networks. In PCF, the *Point Coordinator* (PC), usually the AP, polls every station in A BSS in a round robin scheme and can therefore prevent any contention. The standard requires that both methods must coexist, meaning that a contention-free period follows on a contention period [54].

5.3.4 Wi-Fi direct

Wi-Fi direct is a software extension of the 802.11 protocol and was developed for direct communication of two or more devices in the absence of a distribution network. Instead of building on top of the 802.11 ad-hoc mode, which was never widely adopted and lacks

efficient power saving support and extended QoS capabilities, it extends the infrastructure mode and lets devices negotiate the role of the AP. It inherits all the enhanced QoS, power saving, and security mechanisms of the infrastructure mode [28].
The next subsections discusses the most important new features introduced by Wi-Fi direct.

Architecture

In Wi-Fi direct devices are called *P2P devices* and form *P2P groups* which are functionally equivalent to a BSS in the infrastructure mode. The device that implements the functions of the AP is called the *Group Owner* (GO) and clients are called *P2P clients*. Theses roles are dynamic and negotiated in the discovery phase. Therefore all devices implementing the standard have to implement both the client and the GO functionality [28].
The GO runs a *Dynamic Host Configuration Protocol* (DHCP) server to provide the IP addresses to the clients and only the GO can cross-connect clients to external networks (e.g. an infrastructure WLAN). When the GO disconnects, the P2P group is also disconnected and must be set up again [28].

Group formation

There are several ways in which devices can establish a P2P Group. In the *standard* case, the devices first run a discovery algorithm and then negotiate which device will act as the GO and on which channel in the 2.4 or 5 GHz band the GO will operate. After the GO agreement, a secure connection is established using the *Wi-fi Protected Setup* (WPF) procedure discussed further in section 5.3.4. Finally the GO runs the DHCP server and the P2P client can obtain its IP address. In the *autonomous* group formation, a device can autonomously create a group and become the GO. P2P clients can then directly discover the GO by scanning the medium and do not need the negotiating phase. In the *persistent* case, devices have declared a group as *persistent* in the past and have stored the respective roles and network credentials. After the discovery phase, when a peer recognizes that a persistent group has been formed with the other peer, it can initiate an *Invitation Procedure* (a two-way handshake) to quickly re-instantiate the group [28].

Service discovery

A new feature of Wi-Fi Direct is a Layer 2 discovery protocol called *Generic Advertisement Protocol* (GAS). Using the service discovery protocol, P2P devices can exchange service information before the actual P2P Group formation [28].

Security

Wi-Fi direct uses the *Wi-Fi Protected Setup* (WPS) procedure to secure the connection with minimal user intervention [28].

WPS is based on the *Wi-Fi Protected Access 2* (WPA2) protocol. WPA2 implements the IEEE 802.11i standard and provides data confidentiality and integrity by using the *Advanced Encryption Standard* (AES)-CCMP cypher. When used in Personal mode, a Pre-Shared-Key (PSK) must be present both at the AP and the client for the mutual authentication. The 256 bit PSK is usually generated from a plain text pass phrase that must be entered on both devices. After the authentication, a set of temporary keys is exchanged between the AP and the client which are regularly updated [14].
WPA2 is considered secure against most attacks like man-in-the-middle (MITM), authentication forging, replay, key collision, weak keys, packet forging and brute–force attacks [14].
Using WPS as authentication mechanism, the PSK can be exchanged with less user intervention than usually required by using one of the following methods:

- **Push-Button-Connect(PBC)**: The user has to press a button on both the AP and the client device [94].
- **PIN**: The user has to enter the PIN of the Wi-Fi adapter into the web interface of the AP (Internal registrar) or the user has to enter the PIN of the AP in a UI form on the client device (External registrar) [94].

It has been shown that the External Registrar authentication is potentially vulnerable to brute-force attacks, because the authentication is entirely based on the PIN [94]. Wi-Fi direct uses the PBC authentication method, which is vulnerable against MITM attacks during the authentication phase, which is only active when the buttons are pressed on both devices [60] [94].

5.4 Evaluation

The following table summarizes the most important properties of the different wireless technologies regarding their operating frequency range, typical data rate and security standards used.

	Bluetooth	**Bluetooth 4.0 Low Energy (BLE)**	**Wi-Fi direct**
IEEE Standard	802.15.1	802.15.1	802.11 (a, b, g, n)
Frequency (GHz)	2.4	2.4	2.4 and 5.0
Max. data rate (Mbps)	1-3 (24 with HS)	1	11 (b), 54 (g), 600 (n)
Security	128 bit SAFER+	128 bit AES, user defined on application layer	256 bits AES-CCMP

Table 5.2: Overview of different wireless technologies

Data throughput

Since both the user profile and the contract can contain high resolution images, the payload can easily exceed 1 megabyte. Both classic Bluetooth and Bluetooth Low Energy do not provide the bandwidth to transmit larger files in reasonable time, especially since the achievable data rates will be below the theoretical limit. Bluetooth 3.0+HS could provide up to 24 Mbps, but is not very widely used. Therefore only Wi-Fi direct can provide the required throughput.

Security

The user profile contains personal data and therefore the transmission should be encrypted appropriately. Wi-Fi direct has a stronger encryption by using 256 bit key length but it is also vulnerable MITM attacks during the WPS authentication procedure. Both Bluetooth and BLE offer OOB and PIN authentication to exchange keys and can be considered safer when these mechanisms are used properly.
Although the exchanged data is personal and allows a potential adversary to identify the parties, it does not provide her with information that could be used directly to steal money out of a contract or to use the identity of a party to sign a contract. Even if the parties could sign a contract with a certified signature in the future, the private key would never be transmitted to the other device. Therefore, in the opinion of the author, security is certainly important but does not have the same priority than data throughput in this use case.

Conclusion

The author concludes that Wi-Fi direct is the most suitable technology for the exchange of contract and user data since it provides the highest throughput and offers reasonable security without much effort from the users side.
Section 6.4.3 describes the Android implementation that is used in the application.

Chapter 6

Design and implementation

This chapter describes the design and implementation of the Smart Contract, the Java integration code and the Android client application.

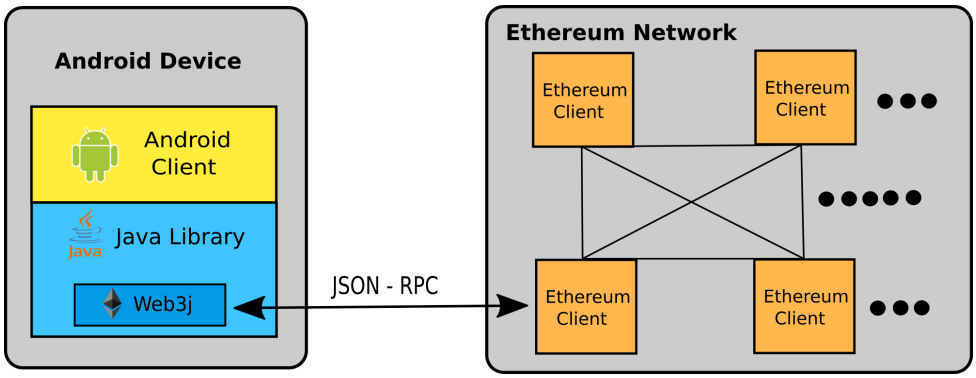

Figure 6.1: Overview of the architecture.

Figure 6.1 show an overview of the overall architecture. The application uses a smart contract deployed on the Ethereum blockchain that stores the description of the purchase or rent item and manages the transaction between the 2 parties. The smart contract and its implementation is discussed in section 6.1. Section 6.3 discusses how the smart contract code executed on the blockchain can be integrated into a Java application using the Web3j framework and the implementation of the actual Android UI components is discussed in section 6.4.

6.1 The smart contract

The following section discusses the Solidity code for the purchase- and rent contract that is used in the application. Listing 6.1 shows the base class used for the purchase and rent contract.

Listing 6.1: The TradeContract base class

```
contract TradeContract {

string public title;
string public description;
uint public price;
uint public deposit;
address public seller;
address public buyer;
bool public verifyIdentity;
bytes32[] private imageSignatures;

enum State { Created, Locked, Inactive, AwaitPayment }
State public state;

function TradeContract(string _title, string _description, bool _verify, uint
    _deposit, uint _price, bytes32[] _imageSignatures)
{
 ...
}

modifier require(bool _condition) {
if (!_condition) throw;
_;
}

modifier onlyBuyer() {
if (msg.sender != buyer) throw;
_;
}

modifier onlySeller() {
if (msg.sender != seller) throw;
_;
}

modifier inState(State _state) {
if (state != _state) throw;
_;
}

event aborted();
```

```
function abort();

}
```

The description of the item is composed of a title, a textual description and an array of image signatures. An image signature is calculated by using the SHA_{256} cryptographic hash function of an image. The reason for not storing the whole image in the block chain are the costs associated with the high storage usage. To further reduce the storage usage and the deployment costs, a light contract version that only stores the SHA_{256} hash of the content attributes has been developed:

$$H_{content} = SHA_{256}(title, description, verifyIdentity, imageSignatures) \tag{6.1}$$

Before deploying a contract, the user can choose if she wants to deploy the full or the light version of the contract. The benefits of the light contract are lower deployment costs and higher privacy as discussed in more detail in chapter 7. The benefit of the full contract is that all details can be loaded from the blockchain and the parties do not need to exchange documents through an alternative communication channel. This can be useful when the parties can not meet face-to-face and use Wi-Fi direct (section 6.4.3) to exchange the contract details.
Every contract also stores the address of the 2 contractual parties as well as the price and the deposit for the item. The deposit is used to ensure the contractual compliance of the parties. Finally, the contract stores a boolean value that indicates whether the parties should exchange personal information. This indicator is used by the client application to determine whether personal information like addresses, photos or other personal data should be exchanged between the parties.
Every contract can be aborted under certain circumstances and therefore declares the *aborted* function and an *aborted* event.

6.1.1 The purchase contract

The purchase contract is derived from the TradeContract described above and it is a slightly modified version of the purchase contract presented in the official Solidity documentation [48].

Listing 6.2: The Purchase contract

```
contract Purchase is TradeContract
{

function Purchase(string _title, string _description, bool _verify, bytes32[]
    _imageSignatures) payable
TradeContract(_title, _description, _verify, msg.value / 2, msg.value / 2,
    _imageSignatures)
{
    if (2 * price != msg.value) throw;
```

```
}

event purchaseConfirmed();
event itemReceived();

/// Abort the purchase and reclaim the ether.
/// Can only be called by the seller before
/// the contract is locked.
function abort()
onlySeller
inState(State.Created)
{
    aborted();
    state = State.Inactive;
    if (!seller.send(this.balance))
    throw;
}

/// Confirm the purchase as buyer.
/// Transaction has to include ‘2 * value‘ ether.
/// The ether will be locked until confirmReceived
/// is called.
function confirmPurchase()
inState(State.Created)
require(msg.value == 2 * price)
payable
{
    purchaseConfirmed();
    buyer = msg.sender;
    state = State.Locked;
}

/// The buyer confirms that she received the item.
/// This will release the locked ether.
function confirmReceived()
onlyBuyer
inState(State.Locked)
{
    itemReceived();
    // It is important to change the state first because
    // otherwise, the contracts called using ‘send‘ below
    // can call in again here.
    state = State.Inactive;
    // This actually allows both the buyer and the seller to
    // block the refund.
    if (!buyer.send(deposit) || !seller.send(this.balance))
    throw;
}

}
```

To give the seller an incentive to not betray the buyer, the seller has to transmit ether in the constructor of the contract. As can be seen on lines 17-18, the value provided must be dividable by 2, otherwise the contract throws an exception. Thus the actual price of the item is only half the value provided in the constructor. When the buyer executes the *confirmPurchase* function, she also has to pay a deposit together with the actual purchase price. The state of the contract is then set to *Locked*. From this point on, the funds of the 2 parties are locked and can only be released when the buyer confirms that she received the item. The locking of the deposits makes sure that no party has a monetary advantage in betraying the other since both invested the same amount of money in the contract and neither can get its deposit back when the other party does not comply.
As long as the contract is in the state *Created* meaning that the buyer has not accepted the offer, the seller can execute the *abort* function and the balance stored in the contract is refunded to her.
In the *confirmReceived* function, the buyer confirms that she received the item. The state of the contract is set to *Inactive* meaning that no further interaction with it is possible. The deposit is refunded to the buyer and the rest of the balance stored in the contract is sent back to the seller. Notice that the state is changed before the ether is transmitted with the *send* function. This prevents an attacker from calling the function again from its fallback function (section 3.3.1).
Every time the state of the contract changes, an event is emitted on the event log. This allows the client application to update its UI and inform the buyer or seller that the state has changed.

6.1.2 The rent contract

In the rent contract, the buyer assumes the role of a renter and the seller assumes the role of a hirer. The hirer defines the details of the rent item together with its renting price and the deposit in the constructor of the contract. The renting price is provided in wei per second.
As long as the contract is in the *Created* state, the hirer can execute the *abort* function. When the renter executes the *rent* function, she has to pay the deposit for the item, the state of the contract is set to *Locked* and the start time for the contract is initialized.
When the renter wants to return an item, the hirer executes the *reclaimItem* function which calculates the renting fee based on the renting time and the price and sets the state of the contract to *AwaitPayment*.
In the *AwaitPayment* state, the renter can execute the *returnItem* function which then returns the deposit and the change money to the buyer. The renting price is sent to the renter.
In the *AwaitPayment* state, the hirer can also execute the *reclaimItem* function again in the case the renter does not pay in time.

Listing 6.3: The Rent contract

```
contract Renting is TradeContract
{

uint256 private rentingFee;
```

```
uint256 private start;

function Renting(string _title, string _description, bool _verify, bytes32[]
    _imageSignatures, uint _deposit, uint _rentingPrice)
TradeContract(_title, _description, _verify, _deposit, _rentingPrice,
    _imageSignatures)
{
}

event itemRented();
event itemReturned();
event paymentRequested();

/// Abort the renting contract.
/// Can only be called by the seller before
/// the contract is locked.
function abort()
onlySeller
inState(State.Created)
{
    aborted();
    state = State.Inactive;
}

/// Confirm the renting of an object as buyer.
/// Transaction has to include the specified deposit.
/// The deposit will be locked until "acceptPayment" is called by the seller.
function rentItem()
inState(State.Created)
require(msg.value == deposit)
payable
{
    start = now;
    itemRented();
    buyer = msg.sender;
    state = State.Locked;
}

/// Calculates the renting fee based on the renting time and the renting price
function calculateRentingFee() returns (uint256)
{
    if(state == State.Inactive)
        return rentingFee;

    if(start == 0)
        return 0;

    uint256 rentingTime = (now - start);

    return price * rentingTime;
```

```
}

/// Accepts the renting fee from the buyer
//  returns the deposit and the change back to the buyer
//  returns the renting fee to the seller
function returnItem()
inState(State.AwaitPayment)
onlyBuyer
require(msg.value >= rentingFee)
payable
{
    itemReturned();
    state = State.Inactive;

    uint change = rentingFee - msg.value;
    if (!buyer.send(deposit + change) || !seller.send(this.balance))
    throw;
}

/// Seller reclaims the item.
/// This will calculate the renting fee for the time period.
/// If the buyer does not pay in time, the seller can execute the function
    again
function reclaimItem()
onlySeller
payable
require(state == State.Locked || state == State.AwaitPayment)
{
    rentingFee = calculateRentingFee();
    paymentRequested();
    state = State.AwaitPayment;
}

}
```

6.2 The Ethereum client

As mentioned in section 3.2.4, the light client protocol for Ethereum is still under development as of August 2017. Therefore, to integrate the Android client with the blockchain, a full Ethereum client on a remote machine is used for testing and development purposes. There are several Ethereum client implementations available. Go-Ethereum [55], the Google Go implementation, is the most popular client and is also used in this project.

6.2.1 Ethereum client interface

To communicate with the Go-Ethereum (Geth) client form another process on the same or on another machine, the JSON-RPC protocol is used. JSON-RPC is a stateless, remote procedure call (RPC) protocol. It uses JSON (RFC 4627) as data format to specify function calls. It is a transport agnostic protocol, meaning that it can be used within the same process, between processes on the same machine or between processes on different machines using different mechanisms of inter process communication (IPC) [57].
Geth supports different communication protocols. The most popular are IPC and HTTP. IPC in this context uses a named pipe to communicate with other processes on the same machine. Applications that run on the same machine like the client can use this interface. However, since the Android client must have access from outside of the host, this approach is not suitable for this application. Instead, the HTTP protocol is used.
The Geth client accepts various arguments for setting up development and test environments and to configure the client for remote access. The following parameters were used for development purposes:

```
geth --rpc --rpcaddr "0.0.0.0" --rpcapi eth,net,web3 --dev --datadir
    $home/.ethereum/
```

The HTTP endpoint of the Geth client is activated by providing the *–rpc* argument. This is a bit confusing since IPC also uses the RPC protocol, but not over the HTTP communication protocol. By setting this argument, the HTTP-RPC server listening by default on port 8545 is activated. By setting the *–rpcaddr* to an address different from "localhost" or "127.0.0.1", processes from other machines can access the API. The *–datadir* argument sets the data directory for the blockchain database and the keystore that stores the wallet files of users [55].
The *–rpcapi* argument specifies which parts of the API will be available over HTTP. By default *eth,net* and *web3* are enabled. *Eth* is the most important interface for the interaction with contracts on the blockchain. It provides functions to sign and issue transactions on the blockchain, to query blockchain specific parameters like the gas price and to gather account specific information like the balance or the number of transactions sent from an account. A full list of all functions is available on Github [57].
The following example shows how a transaction can be issued over the RPC interface using the CURL command on a Linux system. It must be noted, that this call will only work, when the address used in the *from* field is unlocked on the Ethereum client, such that it can sign the transaction with the private key of this account. The Android client has

to sign all transactions itself before issuing a transaction using the “eth_rawTransaction” API function. Account management and transaction security are explained in more detail in sections 6.3.4 and 7.3.

Listing 6.4: Example usage of the ”eth_sendTransaction” function

```
params: [{
  "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
  "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
  "gas": "0x76c0", // 30400,
  "gasPrice": "0x9184e72a000", // 10000000000000
  "value": "0x9184e72a", // 2441406250
  "data":
     "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f07244567
}]

// Request
curl -X POST --data
    '{"jsonrpc":"2.0","method":"eth_sendTransaction","params":[{see
    above}],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331"
}
```

As discussed in section 3.2.2, every transaction contains the address of the sender (*from*), the address of the recipient (*to*), the gas price and gas limit parameters (*gas* and *gasPrice*), a value in wei (*value*) and a data field (*data*). If the recipient address belongs to a smart contract, the data field contains an encoded function call.

6.2.2 Test accounts

Creating a new user account in Ethereum involves creating a public/private keypair and storing the private key in an encrypted file. On the server, on which the Ethereum client is running, the javascript console [56] can be used to create new accounts with the “personal.newAccount(passwd)” command [56]. This creates a new wallet file in the directory specified in the *–datadir* argument.
In order to test the application with a test account, the account needs to have a non-zero ether balance. If the test account wallet file is in the specified data directory, this account can be configured as the mining account with the “miner.setEtherbase” javascript command. This account will then earn all mining rewards, because it is the only mining node in the test network. To transfer money to other non-local test accounts, the “eth.sendTransaction” function can be used [56].

6.3 Java library

This section describes how the Solidity contracts (section 6.1) are integrated into the Android application using the Web3j Java library. It further describes how contracts are persisted and managed on the local Android device.

6.3.1 External libraries

The following external libraries were used in the Java library and in the Android client:

Web3j

The Web3j [95] library is an Ethereum integration library for Java 8 and Android that can be used by applications to interact with smart contracts without writing their own integration code. It is in many ways similar to the web3.js library for javascript that is used by web-clients to interact with contracts on the blockchain [100].

JDeffered

JDeferred is a Java Deferred/Promise library that is heavily inspired by JQuery, a deferred object library for Javascript [69]. JDeferred is used to manage asynchronous operations like transactions on a smart contract or the loading of smart contract details from the Android client code.

QRGen

QRgen [79] is a library used for generating QR-code Bitmap images from arbitrary strings. It us used to generate QR-codes for contracts and user profiles (6.4.2).

Ez-vcard

Ez-vcard [52] is a V-Card library written in Java. It can read and write V-Cards in many different formats. It is used in this project to create V-Cards for user profiles (section 6.4.2) and to parse V-Cards from strings.

Qreader

The QReader API [78] is used to scan QR-codes of contract details or user profiles. It wraps the Barcode API [72] of the Google Mobile Vision project to enable the scanning of QR-Codes without writing much boiler plate code.

6.3.2 Contract integration code

As discussed in section 6.2.1, the Android client communicates over the RPC-HTTP interface with the Ethereum client. On a low level, it interacts with the purchase contract by executing transactions similar to the one shown in listing 6.25.
On a higher level, the Android client should treat a purchase contract as an ordinary class instance with methods for the execution of transactions and methods to access contract fields like the price of an item or its description. It should abstract from the fact that the smart contract code is executed on the blockchain. It should abstract from the fact that the smart contract is implemented in another language and uses other data types that have to be marshalled.
It should also abstract from the fact that the execution of a transaction is executed remotely and can take several seconds to minutes to complete. The Android client should be able to handle asynchronous requests to the Ethereum client without blocking the user interface thread and without limiting the user too much in her freedom of interaction with the application. It must also be possible to register the Android client for specific events on a contract and getting notified when theses events occur. Further the Android client should be able cope with connection problems with the Ethereum client and the blockchain without crashing or loosing track of contracts or transactions.
Another challenge is posed by the fact that the Android client must manage the user accounts locally on the smart phone. Theoretically, it is possible to create and manage accounts remotely by exposing the "personal" interface of the Ethereum client on the HTTP server. Using this method, transactions can be sent in plain text like in the example in listing 6.25. However, this is only possible when the Account is unlocked on the Ethereum client by the "personal.unlockAccount" function [56]. Exposing the personal API over HTTP is a serious security breach because every process in the local network can access the HTTP server and use unlocked accounts to issue its own transactions. Therefore, exposing the "personal" API on the Ethereum client is not an option and the Android client must be able create wallet files on the file system of the Android device and sign transactions with the private keys of these accounts before sending them to the Ethereum client.

The functional requirements for the Java integration library can be summarized in the following points:

- It must provide an RPC client that implements the API specification of the Ethereum client.
- It must provide methods for marshalling Java data types to EVM datatypes and vice versa.
- It must provide methods for managing local wallet files and for signing transactions locally.
- It must provide methods for observing contract events that are logged on the blockchain.
- It must provide mechanisms for handling the asynchronous execution of transactions on the smart contract.

- It must cope with network connection problems between the Android client and the host of the Ethereum client.

The contract wrapper code

Most problems described above are already solved when using the Web3j library for Android (section 6.3.1).

The most important features of the Web3j library regarding the functional requirements are:

- The complete implementation of the JSON-RPC client API over HTTP and IPC.
- The support for local Ethereum wallet files and local transaction signing.
- The generation of Java smart contract wrappers for creating, deploying and transacting with smart contracts from native Java code
- Android support

To create a Java class that wraps a smart contract, the solidity compiler can be used to compile the smart contract code and then the wrapper code is generated using the Web3j command line tool:

Listing 6.5: Compilation of the Solidity code and generation of the Java wrapper class

```
$ solc <contract>.sol --bin --abi --optimize -o <output-dir>/

$ web3j solidity generate /path/to/<smart-contract>.bin
    /path/to/<smart-contract>.abi -o /path/to/src/main/java -p
    com.your.organisation.name
```

After the compilation and generation steps a Java class is obtained that provides methods to deploy, load and interact with a contract. The following code snippet shows how the contract wrapper class for the Solidity purchase contract described in section 6.1 looks like:

Listing 6.6: Code snippet of the generated Java wrapper class

```
public final class PurchaseContract extends Contract {

    private static final String BINARY = "60606040525b60018054600160a...";

    private Purchase(String contractAddress, Web3j web3j, TransactionManager
        transactionManager, BigInteger gasPrice, BigInteger gasLimit) {
        super(contractAddress, web3j, transactionManager, gasPrice, gasLimit);
    }
...
```

```
    public static Future<Purchase> deploy(Web3j web3j, Credentials
        credentials, BigInteger gasPrice, BigInteger gasLimit, BigInteger
        initialValue, String title, String description, boolean verifyIdentity)
         {
        return deployAsync(Purchase.class, web3j, credentials, gasPrice,
            gasLimit, BINARY, "", initialValue, title, description,
            verifyIdentity);
    }

...

    public Future<TransactionReceipt> abort() {
        Function function = new Function("abort", Arrays.<Type>asList(),
            Collections.<TypeReference<?>>emptyList());
        return executeTransactionAsync(function);
    }

...

    public Future<Uint256> price() {
        Function function = new Function("price",
                Arrays.<Type>asList(),
                Arrays.<TypeReference<?>>asList(new TypeReference<Uint256>()
                    {}));
        return executeCallSingleValueReturnAsync(function);
    }

...
}
```

The constructor of the contract wrapper class accepts the *web3j* client that is used to execute API calls on the Ethereum client over the RPC-HTTP interface. The *transactionManager* determines how the Android client communicates with the contract, i.e. whether transactions are signed locally or on the Ethereum client [99]. The meaning of the *gasPrice* and *gasLimit* parameters are explained in section 3.2.2.
The *Contract* base class already provides methods for deploying contracts and for executing transactions and calls on a contract. When executing a function that will change the state of a contract, the *executeTransaction* method is used. This method invokes the "eth_rawTransaction" or "eth_transaction" RPC API calls of the Ethereum client depending on the *transactionManager* instance. If the *RawTransactionManager* is used, the transaction is signed locally and when the *ClientTransactionManager* is used, the transaction is signed by the Ethereum client [98]. When executing a simple call that doesn't effect the state of the contract, like the *price*-function that just returns the price of a purchase item, the *executeCallSingleValueReturn* method is used which internally invokes the "eth_call" RPC API call [99].
Every function of the wrapper code returns a Java Future<T> ([66]) object representing the result of an asynchronous operation.

Limitations of the wrapper code The auto-generated contract code already satisfies the first three requirements stated at the beginning of this section. It implements the Ethereum RPC-API, it converts Java data types into EVM datatypes and it can sign transactions locally with the provided Credentials. However it has some limitations:

- The event subscription with filters in Web3j doesn't work out of the box (V2.1). The Android client main thread often shuts down when subscribing to events of more than 1 contract. Section 6.3.2 discusses how this problem was solved by writing some custom code. In version 2.2.1 of Web3j Android library, this problem has been fixed and the custom code is no longer needed.
- The mechanisms for handling asynchronous operations are not sufficient for the use case of this project. Ordinary Future objects don't provide methods for registering handlers from the Android client code. This issue is discussed further in section 6.3.2.
- The generated wrapper code is not flexible enough. Because it does not support inheritance of Solidity contracts, common code is copied in all classes. It is also not possible to specify a Java interface in the code generation process. Because interfaces are necessary for proper unit testing and to abstract from the concrete implementation in the Android Fragments and Activities, the contract wrapper code is only taken as a starting point for the integration code. The wrapper classes can not be re-generated dynamically when the Solidity contracts have to be extended. Section 6.4.4 discusses the necessary steps to extend the Solidity contracts and the Java wrapper code.
- Web3j allows more fine-grained control over parameters used by the integration code than provided in the auto-generated code. Section 6.3.4 describes how services and factories can be used to have more control over the creation process of a contract while providing a leaner interface to the Android client code.

Filters and event subscription

In Web3j, filters can be used to subscribe to events that occur on the blockchain [96]. There exist 3 different types of filters:

- **Block filters:** Provide notification of newly created blocks on the blockchain
- **Transaction filters:** Provide notification of the creation of transactions on the blockchain
- **Topic filters:** Provide notification of custom events that took place on the blockchain

Because the Android client needs to be notified when the state of the contract changes, topic filters can be used. Web3j provides a fully asynchronous, event based API that uses RxJava's Observable API for working with events. In the Observable API, an observer

subscribes itself to an observable object. The observable can emit items that are processed by the observer, who does not need to wait until the next item is emitted [67].
The following code snippet shows how the purchase contract wrapper class can subscribe to events on its associated smart contract on the blockchain:

Listing 6.7: "Subscribing to the 'aborted' event"

```
...

Event event = new Event("aborted", new ArrayList<TypeReference<?>>(), new
    ArrayList<TypeReference<?>>());
String encodedEventSignature = EventEncoder.encode(event);
final EthFilter filter = new EthFilter(DefaultBlockParameterName.EARLIEST,
    DefaultBlockParameterName.LATEST,
    getContractAddress()).addSingleTopic(encodedEventSignature);

        Async.toPromise(new Callable<Subscription>() {

            @Override
            public Subscription call() throws Exception {
                return web3j.ethLogObservable(filter).subscribe(
                        new Action1<Log>() {
                            @Override
                            public void call(Log log) {
                                notifyObservers(event.getName(), null);
                            }
                        });
            }
        }).done(new DoneCallback<Subscription>() {
            @Override
            public void onDone(Subscription result) {
                subscriptions.add(result);
            }
        });

...
```

The *event* object describes the event by its name and its parameters. In case of the "aborted" event, no parameters are needed. Then the event object is hashed such that it can be compared with the signature stored in the event log of the contract. The *filter* object specifies the contract and the time frame for which events should be observed. With the *addSingleTopic*-method, the event to observe is specified [96].
The *filter* is then provided to the *ethLogObservable* method that returns an Observable<Log> object. With the *subscribe* method the observer is registered on the observable. In this case the observer is notified, when the "aborted" event occurred and will notify observers (Android UI objects) for this contract (not shown here). The *subscription* object returned by the *subscribe* method can later be used to unregister the observable.
As mentioned in section 6.3.2 this approach works only for a small amount of subscriptions in version 2.1 of the library. When observing the state of multiple contracts, it could

happen that the main thread of the Android client would no longer respond. The reason for this is was the underlying implementation of the *ethLogObservable* method.
The *filter* object polled the Ethereum client in a regular interval using the "eth_getFilterChanges" function of the RPC-HTTP interface to check for changes for this filter [57].
In the framework version 2.1, the polling task was done in a separate thread and a busy wait approach was used to delay the subsequent requests with the delay being the block time of the blockchain. Because a thread needed to be started for every contract on the blockchain and was only stopped when the subscription for the contract was cancelled, this approach used too many resources on a mobile device when subscribing to multiple contracts at the same time.
A solution for the problem was to use a ScheduledExecutionService [68]. The ScheduledExecutionService can schedule tasks at a fixed rate and returns a ScheduledFuture object that can be used to cancel the task when the subscription is cancelled. With the ScheduledExecutionService, the busy wait was replaced with a call to the "scheduleAtFixedRate" method that starts threads for the actual polling request and stops them after the request is finished.
In version 2.2.1 of the Android Web3j library, this solution has been implemented and therefore, the custom code is no longer needed.

Handling asynchronous operations

In the generated wrapper code, all functions return a Future<T> object to represent the result of a call or a transaction on the smart contract. The future object provides methods to check if the operation has completed, to wait for completion and to obtain the result of the operation. The *get* method will return the result of the operation if it has completed. If it has not yet completed, it will wait for the result and the calling thread is blocked until the result is ready [66].
The behaviour of the Future object does not comply with the goal of having a dynamic application that does not block the UI thread when waiting for the completion of asynchronous operations. An alternative for ordinary Future objects are CompleteableFuture objects that provide methods for registering handler functions upon completion of the operation [65]. However, since CompleteableFuture is only supported in Java 8, it can not yet be used in the Android application.
An alternative for Futures is the Java JDeferred library [69]. JDeferred is a Java Deferred/Promise library that is heavily inspired by JQuery, a deferred object library for Javascript . Similar to a Future object, a promise holds the value of a computation that is resolved in the future. In contrast to a Future that only holds the value of a computation, a promise can explicitly resolve a computation [77]. In the JDeferred library, a promise supports callback functions that are called when an operation succeeds or fails. This callbacks are used by the Android client code to update the UI in response to successful of failed transactions or other long lasting operations.
To create a Promise that will hold the result of an asynchronous operation, the DeferredManager class can be used. It uses an ExecutorService to execute the tasks provided to it in its *when* function in a background thread [69].

To facilitate the creation of Promises the Android application, the *Async* class was im-

plemented. Its *toPromise* function accepts a Callable<T> representing an asynchronous operation and returns a generic SimplePromise<T> object:

Listing 6.8: Snippet of the Async class

```
public class Async {

    private static DeferredManager deferredManager;
    private static ScheduledExecutorService executorService;

    ...

    public static <T> SimplePromise<T> toPromise(Callable<T> callable)
    {
        final Promise<T, Throwable, Void> promise =
            deferredManager.when(callable);

   ...
    }

    ...
}
```

By using the *toPromise* method, the methods in the contract wrapper class can be rewritten to return a Promise object instead of a Future object:

Listing 6.9: Contract class using Promises instead of Future results

```
public class PurchaseContract extends Contract {

...
    public SimplePromise<String> abort()
    {
        return Async.toPromise(new Callable<String>() {
            @Override
            public String call() throws Exception {
                Function function = new Function("abort",
                    Arrays.<Type>asList(),
                    Collections.<TypeReference<?>>emptyList());
                TransactionReceipt result = executeTransaction(function);
                return result.getTransactionHash();
            }
        });
    }
...

}
```

This allows the caller of the method to register handlers to handle the success or error of an operation without blocking the calling thread:

Listing 6.10: Android client code registering callbacks on a Promise

```
...

contract.abort()
        .done(new DoneCallback<String>() {
            @Override
            public void onDone(String result) {
                ...
            }
        })
        .fail(new FailCallback() {
            @Override
            public void onFail(Throwable result) {
                ...
            }
        });
...
```

6.3.3 Data model and data access layer

To store account- and contract information on the local Android device, the data model shown in figure 6.2 was designed.

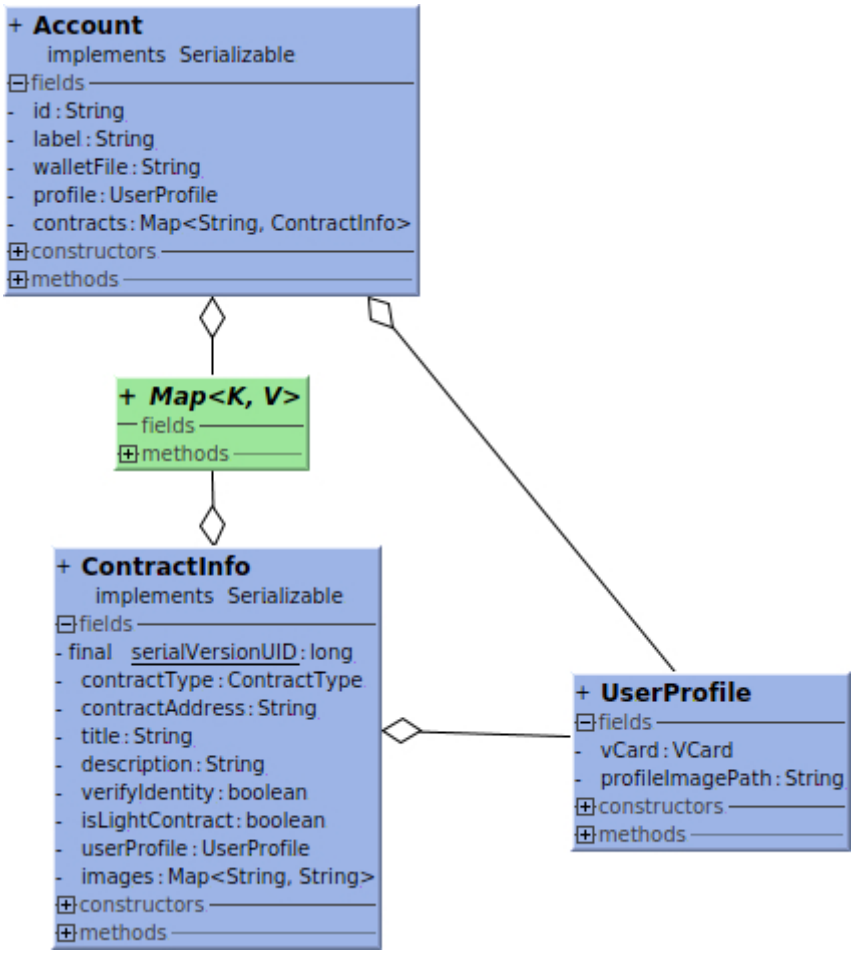

Figure 6.2: Data model for local accounts and contracts.

An Account stores its ID (the public key), a label given by the user and the path to the wallet file on the local file system. Every account also has an associated user profile which stores the personal information of the user in the V-Card format and the file path to the personal photo of the user. The contracts associated with an account are represented by ContractInfo objects that store the address of a contract, the type of the contract (purchase or rent) and all attributes of the contract that are not necessarily stored on the blockchain. This includes the textual *description* and the *title* of the item and a map of *images* that contains the images hash and the absolute path to the image on the local file system. The *isLightContract* field indicates whether the contract wrapper object should load these attributes locally or from the contract on the blockchain. This field is set to "true" if the contract is deployed as a light contract (section 6.1).
Every contract can also store an optional user profile that represents the profile of the other party of this contract. All other information is stored on the blockchain and can be retrieved when the contract is loaded (e.g. the contract price or the contract state).
Account- and contract data is stored on the file system of the Android device in the

JSON format. The *AccountManager* and *ContractManager* interfaces provide methods for creating, deleting and retrieving accounts and contracts and storing user profiles for them. Both interfaces are implemented by the *FileManager* class that uses the Google GSON [58] library to serialize and de-serialize the objects.

Listing 6.11: The Account- and ContractManager interfaces

```
public interface AccountManager
{
    List<Account> getAccounts();
    void addAccount(Account account);
    Account getAccount(String accountId);
    void saveAccountProfile(String accountId, UserProfile profile);
}

public interface ContractManager
{
    void saveContract(ContractInfo contract, String account);
    void deleteContract(String contractAddress, String account);
    List<ContractInfo> getContracts(String account);
    ContractInfo getContract(String address, String account);
}
```

6.3.4 Service layer

The service layer provides functionality that can be used from different Android Activities or Fragments of the Android client (section 6.4). In the following subsections, the services and their functionality are briefly described.

The ContractService

When deploying or loading a contract object, a lot of parameters have to be provided to the factory method (listing 6.6). Since the general parameters, like the Web3j client or the gas price will rarely change, the *ContractService* provides methods for deploying and loading purchase and rent contracts by only specifying the relevant arguments like the address of a contract and its constructor arguments. Further, the *ContractService* provides an interface for saving and removing contracts from the local file system. This functionality is provided by the ContractManager (listing 6.11).

Listing 6.12: The ContractService interface

```
public interface ContractService {

    SimplePromise<ITradeContract> deployPurchaseContract(
            BigInteger price,
```

```
            String title,
            String description,
            Map<String, String> imageSignatures,
            boolean verifyIdentity,
            boolean lightDeployment);

    SimplePromise<ITradeContract> deployRentContract(
            BigInteger price,
            BigInteger deposit,
            TimeUnit timeUnit,
            String title,
            String description,
            Map<String, String> imageSignatures,
            boolean verifyIdentity,
            boolean lightDeployment);

    SimplePromise<ITradeContract> loadContract(ContractType contractType,
        String contractAddress, String account);
    void saveContract(ITradeContract contract, String account);
    void removeContract(ITradeContract contract, String account);
    void saveContract(ContractInfo info, String account);
    void removeContract(String contractAddress, String account);
    SimplePromise<List<ITradeContract>> loadContracts(String account);
    SimplePromise<Boolean> verifyContractCode(String address, String code);
```

The AccountService

The *AccountService* is responsible for managing accounts and their associated user profiles. It provides methods for loading, creating and unlocking of accounts. It uses the Web3j client to retrieve general account information from the blockchain, like the balance of an account.
There are two different implementations of the *AccountService*:

- The *ParityAccountService* uses the Parity client of Web3j to unlock accounts on the Ethereum client. Transactions are then signed by the Ethereum client. This implementation should only be used for testing purposes since it involves sending the password for the wallet file in plain text over the network [99].
- The *WalletAccountService* uses the *WalletUtil* class of Web3j to create and unlock local wallet files on the Android device. The credentials (private/public key pairs) are then used to sign Transactions locally using the *RawTransactionManager* of web3j. The signed transactions are then sent to the Ethereum client [99]. This implementation is secure since the password for the wallet file is not sent over the network but it is also much slower because decrypting the wallet file on the Android device can take much time. It takes approximately 30 seconds with weak encryption and 2 minutes with strong encryption to unlock the wallet file on a Samsung Galaxy A5 device.

The AccountManager (listing 6.11) is used in both implementations for managing accounts on the local file system.

Listing 6.13: The AccountService interface

```
public interface AccountService {

    SimplePromise<List<Account>> getAccounts();
    SimplePromise<Account> createAccount(String alias, String password);
    SimplePromise<Account> importAccount(final String alias, final String
        password, final String walletFile);
    SimplePromise<Boolean> unlockAccount(Account account, String password);
    void lockAccount();
    SimplePromise<BigInteger> getAccountBalance(String account);
    UserProfile getAccountProfile(String account);
    void saveAccountProfile(String accountId, UserProfile profile);
}
```

The ConnectionService

The *ConnectionService* is responsible for periodically checking the connection to the Ethereum client and for notifying subscribers when the state of the connection changes. It provides methods to start and stop polling and to query the connection state to the Ethereum client.

Listing 6.14: The ConnectionService interface

```
public interface EthConnectionService
{
    void startPolling();
    void stopPolling();
    boolean hasConnection();
}
```

The EthConvertService

The *EthConvertService* provides methods to gather real-time information about ether exchange rates for different currencies and for converting currencies from/to ether. Its implementation uses the RESTful web API of cryptocompare.com [29] to retrieve the exchange rates in a JSON format.

Listing 6.15: The EthExchangeService interface

```
public interface EthConvertService
{
    SimplePromise<BigDecimal> getExchangeRate(final Currency exchangeCurrency);
    SimplePromise<BigDecimal> convertToCurrency(final BigDecimal amountEther,
        final Currency exchangeCurrency);
    SimplePromise<BigDecimal> convertToEther(final BigDecimal amountCurrency,
        final Currency exchangeCurrency);
}
```

The EthServiceFactory

All services that need a connection to the Ethereum client and depend on the connection and transaction parameters that can be changed by the user (section 6.4.2) are instantiated by the *EthServiceFactory*. Every time the unlocked account or user defined settings change, the *EthServiceFactory* is used to re-instantiate the service classes.

Listing 6.16: The EthServiceFactory interface

```
public interface EthServiceFactory
{
    AccountService createParityAccountService(String host, int port);

    AccountService createWalletAccountService(
            String host,
            int port,
            boolean useFullEncryption);

    ContractService createWalletContractService(
            String host,
            int port,
            String selectedAccount,
            BigInteger gasPrice,
            BigInteger gasLimit,
            int transactionAttempts,
            int transactionSleepDuration);

    ContractService createClientContractService(
            String host,
            int port,
            String selectedAccount,
            BigInteger gasPrice,
            BigInteger gasLimit,
            int transactionAttempts,
            int transactionSleepDuration);

    EthConvertService createHttpExchangeService();
```

```
    EthConnectionService createConnectionService(String host, int port);
}
```

6.4 Android client

The Android client provides the user interface for the application. It uses the contract wrapper classes (section 6.3.2) to interact with contracts on the blockchain and it uses the service classes (section 6.3.4) to manage contracts and accounts.
The Promise-API (section 6.3.2) is used to dynamically update the UI when transactions complete.
The client code is composed of multiple Android Activity classes that share a common base class. Most UI logic is not implemented in the Activities themselves, but in Fragment classes that can be reused in different contexts.

6.4.1 Structural concepts

The following subsections discuss the most important structural concepts used in the architecture of the Android client.

The ActivityBase class

The ActivityBase class is the base class for all major Activities in the Android client and contains code used in every Activity:

- It initializes the Android Toolbar and contains the Toolbar interaction logic
- It handles permission request results (e.g. for accessing the device camera or the external storage)
- It contains and initializes the BroadcastReceiver (section 6.4.1) and handles completed transactions and other global messages
- It implements the ApplicationContextProvider interface to provide the ApplicationContext (section 6.4.1) to Fragment components.

ApplicationContext

All Activities and Fragments should have access to certain functionality to interact with accounts and contracts or to access global application settings. The life-cycle of these objects need to be managed centrally.
Usually, data or functionality between different Activities of an Android application is shared by passing objects in the Activities Intents. This requires however, that the object is serializable, which is not always possible when using external libraries. Another problem is that the life-cycle of an object passed through an Intent cannot be managed properly any-more. This is especially a problem when only one instance of a class can exist like in the case of the Ethereum service objects (section 6.3.4).
A possible solution for those problems would be to implement a Singleton pattern and

provide these objects statically. However by doing this, the dependencies of classes that depend on theses objects are no longer visible. Functionality can be accessed from anywhere in the code. Static objects can also not be mocked easily for testing purposes and are therefore considered bad practice in object-oriented software engineering.
A better solution is to manage all instances in a custom Application object and to provide them through an interface that can be used by Activities and Fragments.
The *ApplicationContext* interface provides access to global functionality including access to the Ethereum service classes through the ServiceProvider and many other Android specific services that are discussed in more details in the following subsections. The *ApplicationContext* is implemented by a custom Android Application class [1], the AppContext. Upon creation of the application, it instantiates and initializes all service classes. The *ApplicationContext* can be accessed through the *ApplicationContextProvider* interface which is implemented by the ActivityBase class.

Listing 6.17: The ApplicationContext interface

```
public interface ApplicationContext extends ActivityChangedListener
{
    ServiceProvider getServiceProvider();
    SettingProvider getSettingProvider();
    TransactionHandler getTransactionManager();
    P2PSellerService getP2PSellerService();
    P2PBuyerService getP2PBuyerService();
    PermissionProvider getPermissionProvider();
    BroadCastService getBroadCastService();
    MessageService getMessageService();
    Context getContext();
}
```

Listing 6.18: The ApplicationContextProvider interface

```
public interface ApplicationContextProvider {
    ApplicationContext getAppContext();
}
```

The ServiceProvider

All services are created by the EthServiceFactory (section 6.3.4) and centrally managed through the *ServiceProvider* class. The *ServiceProvider* acts as the bridge between the service layer and the Android Activities and Fragments. It provides accessors for all service objects and is responsible for re-instantiating them when global settings (section 6.4.2) changed or when the selected account changed (section 6.4.2).

Listing 6.19: The ServiceProvider interface

```
public interface ServiceProvider
```

```
{
    ContractService getContractService();
    AccountService getAccountService();
    EthConvertService getExchangeService();
    EthConnectionService getConnectionService();
}
```

SettingProvider

The *SettingProvider* provides accessors for global settings that can be configured in the SettingActivity (section 6.4.2). It implements the OnSharedPreferenceChangedListener interface and updates its fields when changes in the SharedPreferences [6] of the application occur. It also listens to global account messages and keeps track of the currently selected user account.

Listing 6.20: The SettingProvider interface

```
public interface SettingProvider
{
    String getWalletFileDirectory();
    String getWalletFileEncryptionStrength();
    String getHost();
    int getTransactionSleepDuration();
    int getTransactionAttempts();
    BigInteger getGasLimit();
    BigInteger getGasPrice();
    String getSelectedAccount();
    int getPort();
    String getAccountDirectory();
    String getImageDirectory();
}
```

ActivityChangedListener

Certain services like the MessageService or the PermissionProvider need to have access to the currently active Activity to access permissions or to access the FragmentManager. The *ActivityChangedListener* is implemented by the ApplicationContext instance and all ActivityBase instances call the respective interface method in their "onResume" and "onPaused" methods. The ApplicationContext object then calls the respective interface methods on all services that implement the *ActivityChangedListener* interface.

Listing 6.21: The ActivityChangedListener interface

```
public interface ActivityChangedListener {
```

```
    void onActivityResumed(ActivityBase activity);
    void onActivityStopped(ActivityBase activity);
}
```

MessageService

The *MessageService* interface is used to display error or info messages to the user from every component of the application. It has access to the current active Activity by implementing the ActivityChangedListener interface.

Listing 6.22: The MessageService interface

```
public interface MessageService
{
    void showErrorMessage(String message);
    void showMessage(String message);
    void showSnackBarMessage(String message, int length);
}
```

BroadcastService

By using the JDeferred API to handle transactions on a contract (section 6.3.2), Android Activities and Fragments can attach handlers to a Promise object and update their UI when the transaction is completed. However, by handling the completion of transactions on a specific Activity, the user cannot switch to another Activity because the attempt of updating the user interface of a paused or destroyed Activity will cause an exception. The user will also get no feedback for her interaction in the newly started Activity.
For example when a user creates a new contract or interacts with the contract, that transaction can take 15 seconds (one block time) up to several minutes to complete. It should be possible that the user can go back to the OverviewActivity(section 6.4.2) or any other Activity and get notified there about the success or failure of her transaction.
To enable this behaviour, all main Activities are subscribed to the LocalBroadCastManager in their common base class. The LocalBroadcastManager facilitates sending and receiving broadcast messages within a single Android application [2].
The *BroadcastService* interface wraps access to the LocalBroadcastManager and facilitates the testing of Android components that need to subscribe to it.

Listing 6.23: The BroadcastService interface

```
public interface BroadCastService {

    void registerReceiver(BroadcastReceiver receiver, IntentFilter
        intentFilter);
    void unregisterReceiver(BroadcastReceiver receiver);
    void sendBroadcast(Intent intent);
```

```
}
```

The private BroadcastReceiver ([3]) class of the ActivityBase class handles messages that are emitted when a transaction completed, when the global settings changed or when the unlocked account changed. Activities derived from ActivityBase can override handler methods to react appropriately to specific events. For example the OverviewActivity (section 6.4.2) adds a contract to its list when a new contract is successfully created or reloads its contract list when the unlocked account changes.

TransactionHandler

To broadcast contract transactions with the help of the *BroadcastService*, the *TransactionHandler* was implemented.

Listing 6.24: The TransactionHandler interface

```
public interface TransactionHandler
{
  ...

    <T> void toTransaction(SimplePromise<T> promise, final String
        contractAddress);
    void toDeployTransaction(SimplePromise<ITradeContract> promise,
        ContractInfo contractInfo, String account, ContractService
        contractService);
}
```

The *toTransaction* method waits for an ordinary contract transaction to either complete or fail. In the error case, it displays a message to the user that the transaction could not be completed.
The *toDeployTransaction* method is used by the ContractService (section 6.3.4) to handle contract deployment transaction. If the transaction completes, the *contractInfo* object is saved on the file system. If the transaction fails while the Android device lost connection to the Ethereum client the contract is also saved. This ensures that no contracts can be lost during deployment.

PermissionProvider

The *PermissionProvider* is used to check and request the Android permissions for accessing the camera and the external storage. The result of a permission request is handled by the ActivityBase class which invokes the “onPermissionGranted” or “onPermsissionDenied” method of the Activity that requested the permission.

Listing 6.25: The TransactionHandler interface

```
public interface PermissionProvider {

    String READ_STORAGE = "android.permission.READ_EXTERNAL_STORAGE";
    String CAMERA = "android.permission.CAMERA";

    boolean hasPermission(String permission);
    void requestPermission(String permission);
}
```

6.4.2 Activities

The following subsections describe the individual Android Activities of the application. Their purpose in terms of the use cases they cover is discussed and their structural relationships and most important dependencies are shown in appendix A.

Application Core

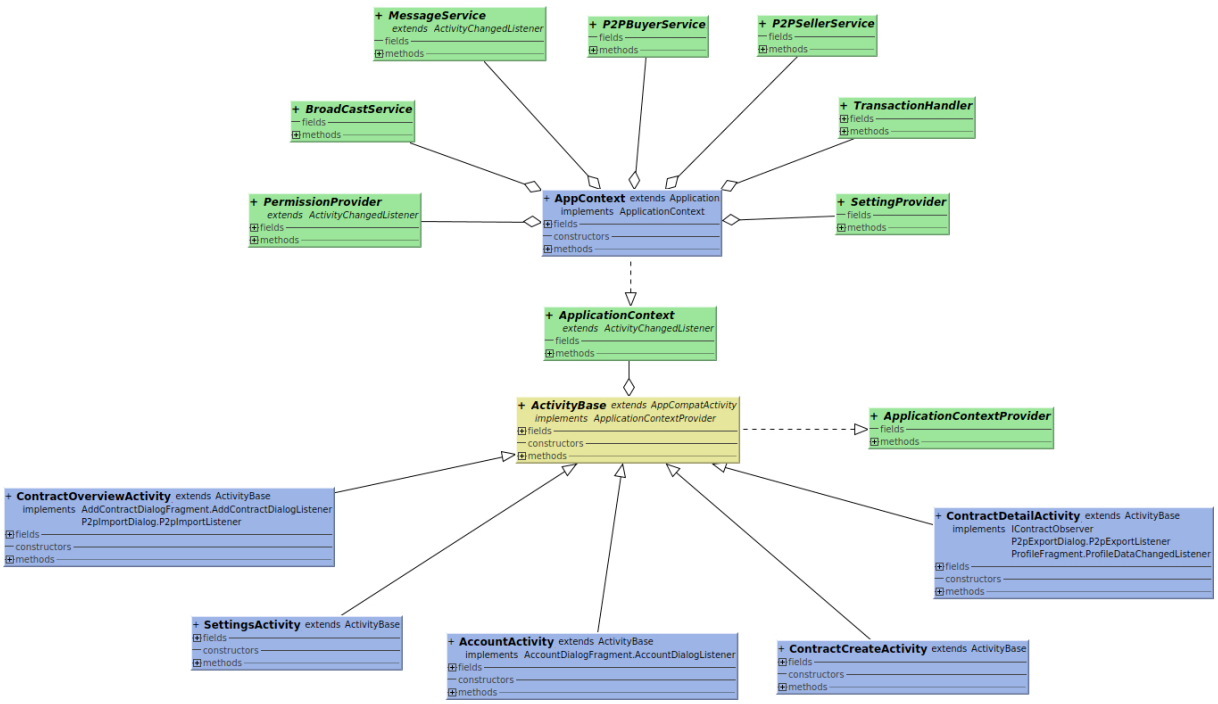

Figure 6.3: Application core structure.

The UML class diagram in figure 6.3 shows the structure of the core of the Android client. The *AppContext* manages the service- and provider implementations discussed in section 6.4.1 and the *ActivityBase* provides the *ApplicationContext* through the *ApplicationContextProvider* interface to the derived Activities and their Fragments.

The AccountActivity

The *AccountActivity* provides the user interface to access and manage accounts. Its main interaction logic is contained in the *AccountFragment*, which displays a list of available Ethereum accounts which the user can lock or unlock (1). The *AccountActivity* also provides the *AccountDialogFragment* (2) to create new accounts or to import existing accounts from a wallet file on the file system. After the user has unlocked an account, the balance for this account is displayed at the top left corner(3).

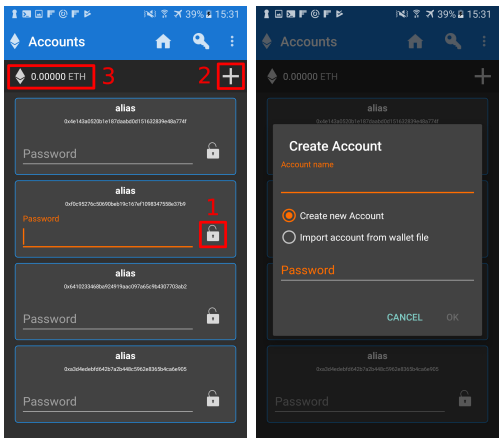

Figure 6.4: Account Activity

Figure 6.5: Account Import Dialog

The OverviewActivity

The *OverviewActivity* is the main Activity and the entry point of the application. Its main component is the *TradeContractFragment* that displays a list of stored contracts for the unlocked account. A list item displays some properties of a contract like its title and price and navigates the user to the *ContractDetailActivity* for the selected contract when she clicks on it.
The top interaction bar contains buttons to import contracts using the *P2PImportDialog* (1) (section 6.4.3), to import contracts by scanning of QR-codes with the camera (2) using the *QrScanningActivity* and to create new contracts (3) in the *AddContractDialogFragment*. The *OverviewActivity* also provides a full text search (4) to filter the contract list based on keywords.

The CreateActivity

In the *CreateActivity*, the user can create new purchase or rent contracts. The Activitie's interaction logic is contained in the concrete implementation of the *ContractDeployFragment* and depends on the contract type. Both deployment Fragments contain the common attributes of a contract like its title, description and images that can be defined by the

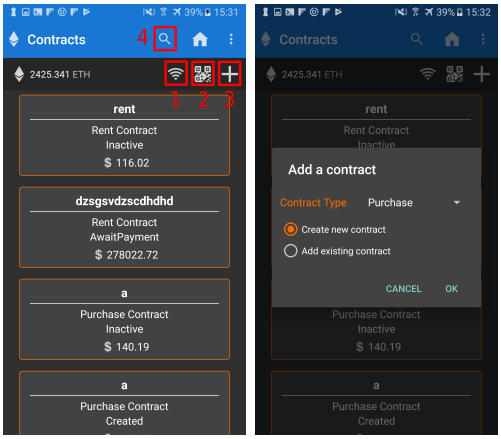

Figure 6.6: Overview Activity

Figure 6.7: Contract Create Dialog

user. In the *PurchaseContractDeployFragment*, the user can set the price for the purchase item and in the *RentContractDeployFragment*, she can set the renting price and the deposit for the item. For both Fragments, the user can change the currency that is used to calculate the price in ether.
The user can decide whether she wants to set up an anonymous contract without the exchange of any user data with the other party or if personal data should be exchanged. Before the deployment of the contract, the user can also decide whether all attributes should be stored on the blockchain (full deployment) or if only the hash of the content (title, description, images) should be stored (light deployment).
Once all required fields are filled out correctly the user can deploy the contract by clicking the “deploy” button. She is then redirected to the OverviewActivity and is later notified by a dialog when the contract was created or when an error occurred.

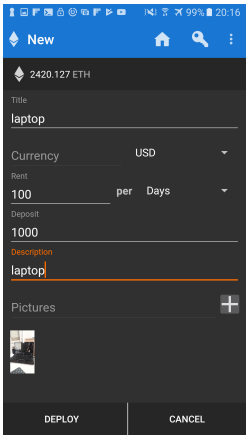

Figure 6.8: Create Activity

The DetailActivity

The *DetailActivity* displays the properties of a contract and it provides controls for executing transactions on it. The *DetailActivity* can be reached by clicking on a contract list item in the OverviewActivity. The concrete *ContractDetailFragment* implementation depends on the contract type. It contains text fields to display the properties of a contract like its price, description or its current state. The buttons at the bottom of the *ContractDetailFragment* are used to execute transactions on the contract. Depending on the current state and the role of the user (buyer or seller) in this contract, actions are enabled or disabled. It also contains a bitmap image of a QR-code that contains the contract details (3). The image can be maximized and scanned by another party to import this contract.
If the contract requires the exchange of personal data, the interaction buttons will be invisible and the user is notified by a Dialog that she needs to scan the profile of the other party to interact with the contract. This can be done by either scanning the QR-code of a profile of another user (2) or by exporting the contract to another user through the Wi-Fi direct connection (1).
After the profile has been scanned or imported, it is displayed in a separate tab and the interaction buttons become visible.

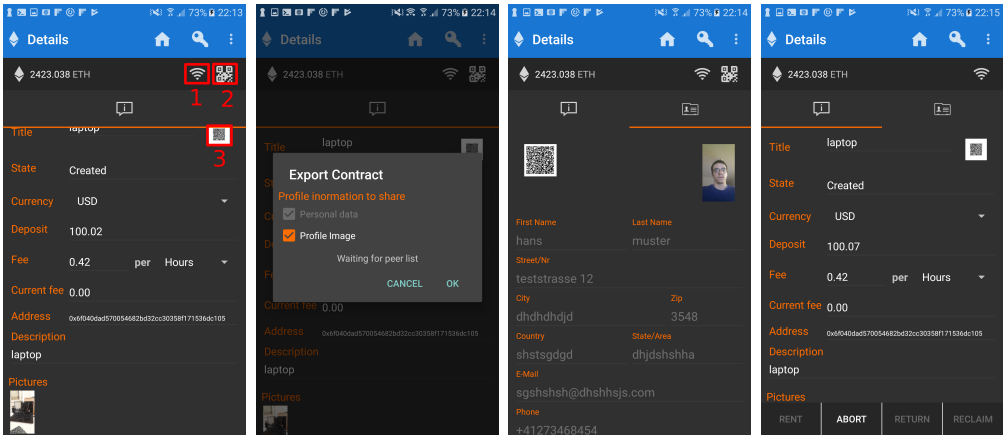

Figure 6.9: Detail Fragment before identity verification

Figure 6.10: Contract Export Dialog

Figure 6.11: Imported Profile

Figure 6.12: Detail Fragment with contract interactions

The ProfileActivity

In the *ProfileActivity*, the user can edit and save her personal information. The whole UI logic for editing and saving user information is contained in the *ProfileFragment* that can be used either in edit- or read-only mode. It is also used to display user profiles in the DetailActivity.

The *ProfileFragment* contains text fields for the users name, address, e-mail address and phone number. Once the user filled out all required fields, she can save the profile and a QR code for this profile is generated and displayed in a bitmap image(1). This image can be maximized and scanned by other users to import this profile. The user can also change her personal image(2) by either importing an image from the file system or by making a new image with the camera of the device.

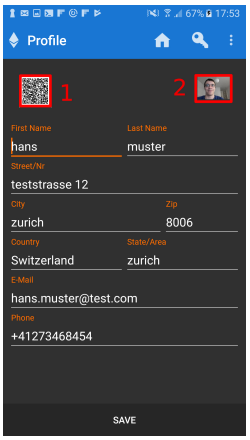

Figure 6.13: The Profile Activity

The SettingActivity

The *SettingsActivity* is used to display and edit the SharedPreferences [6] for the application. It contains a PreferenceFragment [4] that shows a hierarchy of Preference objects defined in an xml resource. When the user changes the value of an object, it is stored automatically in the SharedPreferences of the application. When the SharedPreferences change, the SettingProvider instance is notified and updates itself accordingly.

The applications settings include:

- **Ethereum client settings:** Include the host and port of the Ethereum client on the remote machine and the polling interval, in which the connection to the client is checked.
- **Transaction settings:** Include parameters used for the transactions by the TransactionManager of the Web3j framework such as the gas price, the gas limit, the number of transaction attempts and the sleep time between transaction attempts.
- **Account settings:** Include the encryption strength used for the encryption of local wallet files and the account management mode that should be used. When local account management is selected, accounts will be created on the device and transactions will be signed locally. When using remote transaction management, accounts will be unlocked on the Ethereum client and transactions will be signed remotely. Do only use remote management for testing purposes and never when making transactions on the live network.

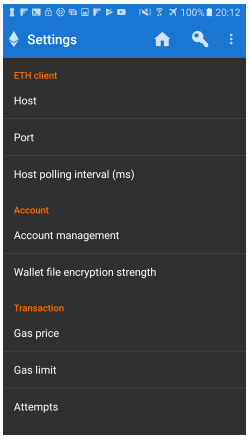

Figure 6.14: The Setting Activity

The QrScanningActivity

The *QrScanningActivity* class is the only Activity that is not derived from the *ActivityBase* class. It is used by other Activities to scan QR-codes and extract serialized contract details or user profiles with help of the QReader API (section 6.3.1).
It is composed of a single SurfaceView [8], to which a CameraSource [74] is attached. The CameraSource uses the camera of the device to retrieve subsequent images which it then analyses with the BarcodeDetector [73]. Whenever the BarcodeDetector detects a barcode, the detection method of its Processor is called and checks whether the decoded string is a valid user profile or contract object. Depending on the action supplied in the Intent that started the Activity, the scanning process continues until either a contract or user profile is detected or the Activity is interrupted by the user.

6.4.3 Device-to-device implementation

As discussed in chapter 5, the Wifi-Direct communication protocol is the most suitable for the use case of exchanging contract and user information, mainly because of its high data throughput required to exchange larger media files.
The Android framework provides the Wifi-P2P interface that allows devices to connect directly to each other via Wi-Fi without an intermediate access point [63].

Connection management

The *WifiP2PManager* class of the Android Wi-Fi direct framework is used to discover other peers in the surrounding and to connect to them. The *WifiP2PManager* also allows a client to register callbacks that are invoked when a method succeeds or fails. Further it can receive intents that notify the client when specific events occur on the framework, e.g. when a new device is detected, when a connection is established or lost [63].
The *WifiConnectionManager* class implements the *P2PConnectionManager* interface and uses an instance of the *WifiP2PManager* to discover new peers and to connect and disconnect to them. It further accepts a *P2PConnectionListener* callback that is invoked when a connection is established or lost, when the available devices in the surrounding change or when an error occurs during a connection attempt.
The purpose of these interfaces is to decouple the P2P connection interface from the underlying protocol implementation.

Listing 6.26: The P2P connection interfaces

```
public interface P2PConnectionManager
{
    void startListening(P2PConnectionListener connectionListener);
    void stopListening();
    void connect(String deviceName);
    void disconnect();
}

public interface P2PConnectionListener {

    void onConnectionLost();
    void onPeersChanged(List<String> deviceList);
    void onConnectionEstablished(ConnectionInfo connectionInfo);
    void onConnectionError(String message);
}
```

P2P service layer

To facilitate the interaction of the Android UI Dialogs with the connection management code and with the data exchange code, two interfaces were implemented that accept a

buyer- or seller specific callback interface in their *requestConnection* method.

Listing 6.27: The P2PSellerService and P2PBuyerService interfaces

```
public interface P2PService<T extends P2pCallback> {
    void disconnect();
    void requestConnection(T callback);
}

public interface P2PSellerService extends P2PService<P2pSellerCallback>
{
    void connect(String deviceName);
}

public interface P2PBuyerService extends P2PService<P2pBuyerCallback> {
}
```

Both services implement the *P2PConnectionListener* interface and register themselves on the *P2PConnectionManager*. Both services also use a buyer or seller specific *Peer* implementation to exchange data with the other device.
The *P2PSellerCallback* and *P2PBuyerCallback* interfaces are implemented by the corresponding Android Dialogs and their callback methods are invoked by the service instance in case of connection specific updates or errors or directly by the peer instance in case of data related updates (when profile or contract related data is requested or has been received or when communication errors occur).

Listing 6.28: The P2PSellerCallback and P2PBuyerCallback interfaces

```
public interface P2pCallback {
    void onP2pInfoMessage(String message);
    void onP2pErrorMessage(String message);
    void onTransmissionComplete();
}

public interface P2pBuyerCallback extends P2pCallback
{
    void onContractInfoReceived(ContractInfo contractInfo);
    void onUserProfileRequested(UserProfileListener listener);
}

public interface P2pSellerCallback extends P2pCallback
{
    void onContractInfoRequested(ContractInfoListener listener);
    void onUserProfileReceived(UserProfile data);
    void onPeersChanged(List<String> deviceNames);
}
```

Data transfer layer

The actual data exchange protocol used to exchange contract and profile data between the two devices is implemented by the buyer- and seller specific *Peer* implementations. The instances are created and started by the service instances after a connection between the two peers has been established.

Listing 6.29: The Peer interface

```
public interface Peer {
    void start();
    void stop();
}
```

The peer implementations use Java sockets to send and receive serialized JSON objects in case of profile and contract details or binary data-streams in case of image files.
in Wi-Fi direct, only one peer assumes the role of the *Group Owner* (GO) (section 5.3.4). After the connection has been established, only the IP of the GO is known to both peers. Because the assignment of group ownership is not deterministic, both peer implementations can assume the role of the server or the client. The role of the server is always assumed by the GO. After a connection has been established, the GO searches a free local TCP port and waits a moment before opening the server socket such that the other peer can also discover the same free port. After the TCP connection has been established, the peers assume their respective roles and use the opened sockets for the rest of the data exchange.
The actual data exchange protocol is illustrated in the following figure:

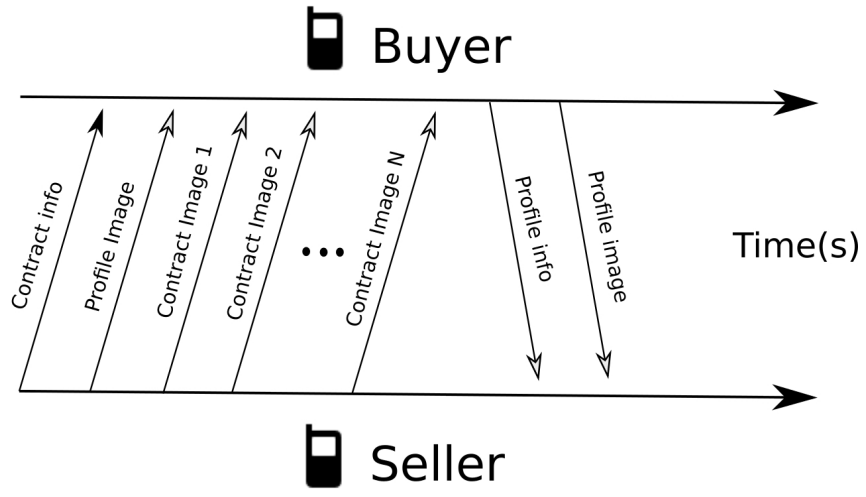

Figure 6.15: P2P data exchange protocol.

The seller peer first sends the serialized contract info object, which also contains the optional profile information of the seller, to the buyer peer. The profile information can contain an optional profile image and the contract itself can also contain optional images. Since the paths of the images are contained in the info objects, the buyer peer knows how

many image files to expect. if the contract info contains the user profile of the seller, the buyer peer also returns its profile and associated image back to the seller peer.

Android user interface

The user interface for the contract import- and export functions consist of 2 Android Dialogs. The *ContractExportDialog* implements the *P2pSellerCallback* interface and registers itself on the *P2PSellerService*. It can be opened in the DetailActivity of a contract. It first displays a list of nearby devices to the user. When the user selects a device, the buyer's device displays a system Dialog in which the buyer can accept or refuse the connection request.
If the buyer accepts the request, a connection is established and the contract information together with its images are sent from the sellers device to the buyer. If the contract requires the exchange of user information, the seller can first decide what information she wants to share. At the moment, the V-card of the user is mandatory and the profile image is optional. If user information is sent, the buyer will have to return her profile before the transmission is complete.
Both users can cancel the transmission every time by clicking on the "cancel" button which will also destroy the Wi-Fi direct group.

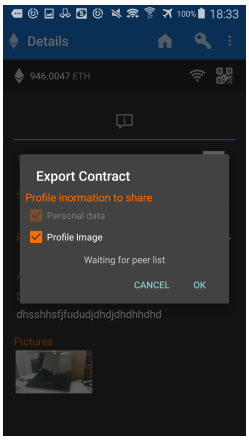

Figure 6.16: The P2pExportDialog

6.4.4 Contract extension

To add additional attributes or transaction functions to an existing Solidity contract or to create a new contract that is derived from the TradeContract (section 6.1), the Solidity code needs to be recompiled and the Java wrapper class and certain Android Fragments need to be extended manually.
The steps involved in extending an existing contract include:

1. Add a new public variable or public function to the rent or purchase contract. This needs to be done in the full and light versions of the contract.
2. Compile the full and light versions of the contract with a Solidity compiler (e.g. the browser solidity compiler [24]) and copy the binary string into the corresponding Java wrapper class.
3. In case a new variable needs to be provided to the constructor of the Solidity contract, the "deploy" method for the TradeContract base class and the "deploy" method of the ContractService also need to be changed. The variable also needs to be added as a field of the ContractInfo class if it is a content attribute that is not available in the light version of the Solidity contract.
4. Add public accessor methods or transaction methods similar to the existing ones both to the contract wrapper class and to the interface of the contract. For accessor methods that return content attributes that can be stored locally, check if the instance is a local contract and return the local value in this case.
5. Use the new attributes or functions in the existing Detail Deploy Activities.

In case of a new contract derived from the TradeContract, the following steps should be followed:

1. Create a new Solidity contract and derive it from TradeContract. Do this for the full and light versions of the contract.
2. Create a new Java Wrapper class derived from the TradeContract base class and create also a new interface derived from ITradeContract.
3. Add a new ContractType value to the "ContractType" enum
4. Compile the full and light versions of the contract with a Solidity compiler (e.g. the browser solidity compiler [24]) and copy the binary strings into the corresponding Java wrapper class.
5. Follow steps 3+4 for the extension of an existing contract to add the fields and transaction methods
6. Create a new Detail- and DeployFrament for the new contract type
7. Adapt the ContractDetailActivity and the ContractCreateActivity such that the correct fragment is loaded for the new contract type.

Chapter 7

Evaluation

7.1 Costs

As already mentioned in section 3.2.2, the creation of Ethereum contracts and the interaction with these contracts consume gas and have to be paid by the sender of a transaction. This section discusses the deployment and interaction costs associated with the purchase and rent contracts described in section 3.3.

7.1.1 Deployment costs

According to the gas costs spreadsheet published in the official Solidity documentation, the main costs for deploying a contract are the costs associated with storing the contract code ("CREATEDATA") with costs of 200 gas per byte and the costs for storing additional data on the storage of the contract ("STORAGEADD") with 20000 gas per 256 bit word. Further, in addition to the 21000 gas for a normal contract transaction, 32000 gas have to be paid for a transaction that creates a contract [39].
The absolute deployment costs for a contract dependent heavily on the size of the contract code and the amount of bytes that it assigns to the storage in its constructor. Since both the purchase and the rent contract don't perform any calculation intensive work in their constructors, their deployment costs in gas can be estimated using the following formula:

$$C_{gas} = (53000 + 200 * N_{bytes} + 20000 * N_{words}) \tag{7.1}$$

Where C_{gas} are the total transaction costs in gas, N_{bytes} is the contract size in bytes and N_{words} is the number of 256 bit words that are initialized in the constructor. To calculate the deployment price in US-Dollar, the gas usage has to be multiplied with the the gas price P_{gas} and the US-Dollar exchange rate for the ether $P_{exchange}$:

$$C_{dollar} = C_{gas} * P_{gas} * P_{exchange} * 10^{-18} \tag{7.2}$$

Empirical results

The actual deployment costs in gas for the rent contract and purchase contract were evaluated using the browser Solidity online compiler ([24]) and the local Ethereum test client described in section 6.2. For the median gas price, a value of 21 gwei (gigawei) was assumed [51]. For the US-Dollar exchange rate, the average value of 210 dollar per ether since May 2017 was taken as a reference value [38].
Table 7.1 summarizes the measured costs for deploying the rent and purchase contracts in full and light deployment mode using different amounts of additional data to store. The values represent the costs in gas and the values in braces represent the costs in US-Dollar. The costs for the light contracts are significantly lower because their source code uses less space. Further they use less storage by storing all content attributes (text and images) in a single 32 byte hash. Therefore, the deployment costs are constant and independent of the amount of additional data sent in the constructor.
The measured values are close to the values estimated by the simplified equations 7.1 and 7.2 and are shown in table 7.2. The Mean Absolute Percent Error (MAPE) is 5.8 percent.

	Empty Contract	**With 1 image (32 bytes)**	**With 2 images (64 bytes)**	**With 3 images (96 bytes)**
Purchase contract	691023 (3.05)	697617 (3.08)	719104 (3.17)	740591 (3.27)
Rent contract	760550 (3.35)	797144 (3.52)	818631 (3.61)	840118 (3.70)
Purchase contract light	389702 (1.72)	389702 (1.72)	389702 (1.72)	389702 (1.72)
Rent contract light	486946 (2.15)	486946 (2.15)	486946 (2.15)	486946 (2.15)

Table 7.1: Measured deployment costs for different deployment configurations

	Empty Contract	**1 image (32 bytes)**	**2 images (64 bytes)**	**3 images (96 bytes)**
Purchase contract	644800 (2.84)	664800 (2.93)	684800 (3.02)	704800 (3.11)
Rent contract	737000 (3.25)	757000 (3.33)	777000 (3.43)	797000 (3.52)
Purchase contract light	363600 (1.60)	363600 (1.60)	363600 (1.60)	363600 (1.60)
Rent contract light	454000 (2.00)	454000 (2.00)	454000 (2.00)	454000 (2.00)

Table 7.2: Estimated deployment costs for different deployment configurations

7.1.2 Transaction costs

The transaction costs for the purchase and rent contracts are dominated by the fix transaction costs ("GTX") of 21000 gas, the costs for adding a 256 bit word to the storage ("STORAGEADD") of 20000 gas, the costs for modifying a word on the storage ("STORAGEMOD") of 5000 gas and the costs for making a call from the contract that contains ether ("GCALLVALUETRANSFER") of 9000 gas. Other costs are not significant since no mayor computation is done in any of the transaction functions. Therefore, to estimate the transaction costs, the following formula can be used:

$$C_{gas} = 21000 + 20000 * N_{words\_add} + 5000 * N_{words\_mod} + 9000 * N_{tx} \tag{7.3}$$

Where C_{gas} are the total costs in gas, N_{words_add} are the number of 256 bit words added to the storage in the transaction, N_{words_mod} are the number of word that are modified in the transaction and N_{tx} are the number of messages with ether that are sent in the transaction function (e.g. for refunding ether to the buyer or seller). The total costs in US-Dollar can be estimated by equation 7.2.
Table 7.3 compares the measured transaction costs for all transactions of the rent and purchase contracts to the estimated costs with equation 7.3. It was found that a variable is only initialized on the storage when a value different from 0 is assigned to it. For example, the *state* variable of the contract is only added in the *abort* or *confirmPurchase* function, when the value changes from the initial state. This has to be taken into account when using the formula.
The model yields very good results with an MAPE of only 3.8 percent.

	abort	**confirm purchase**	**confirm received**	**rent item**	**reclaim item**	**return item**
Measured costs	49584 (0.22)	62890 (0.28)	41657 (0.18)	82721 (0.36)	41913 (0.18)	48146 (0.21)
Estimated costs	50000 (0.22)	61000 (0.27)	44000 (0.19)	81000 (0.36)	44000 (0.19)	45000 (0.20)

Table 7.3: Real and estimated transaction costs

7.2 Scalability

This section discusses the scalability of the Android application in terms of the number of contracts it can handle concurrently and the scalability of the whole system including the external server that hosts the Ethereum client.

7.2.1 Application scalability

There are two factors that limit the number of contracts that the application can handle concurrently:

- **Internal storage limit:** Since every contract is saved on the device, the available free space on the internal storage limits the number of contracts that can be stored.
- **Synchronization overhead:** Since every loaded contract is synchronized with the blockchain through the Ethereum client on the server, the polling requests needed to keep track of the contract states could lead eventually to a very high CPU and network load on the Android device.

Internal storage limit

The internal storage limit available for the application depends on the total size of the internal storage and the internal storage space that is occupied by other applications. The size of one contract depends on the number of characters and on the number of images used for the description of a contract item. A minimum JSON serialized contract that stores only 1 character as title, one character as description plus the metadata needed to properly load the contract from the blockchain (address, contract type) has a size of 179 bytes. Therefore the upper limit for the number of contracts that can be stored on the internal storage can be expressed with the formula:

$$N_{contracts} \leq (S_{internal} - S_{occupied}) \div 179 \tag{7.4}$$

Where $N_{contracts}$ is the number of contracts, $S_{internal}$ is the total size of the internal storage in bytes and $S_{occupied}$ is the internal storage in bytes that is occupied by other applications.

Synchronization overhead

The synchronization overhead is not a limiting factor in reality. The only Activity that loads and displays more than one contract, is the OverviewActivity (section 6.4.2). It uses an Android RecyclerView that can reuses already instantiated views again when the user scrolls down in the list. Every time a view is bound the a contract at another position, the view is unregistered from the old contract and registered on the new contract. A contract is registered for all events that can happen on the smart contract on the blockchain and every event subscription involves 1 HTTP request to the Ethereum client to check if the event has been emitted (see section 6.3.2).
Debugging the application revealed that only four different views are used to display the contracts and therefore only the states for four contracts have to be tracked on the Ethereum client concurrently. In an empirical test that loaded 1000 contracts into the RecyclerView, no performance issues were detected. When scrolling through the list, contracts are loaded continuously.

7.2.2 System scalability

At the moment the Android application relies on an external server that hosts an Ethereum client to interact with contracts on the blockchain (section 6.2). This configuration has scalability issues when multiple android clients connect to the same server. Eventually the network interface will be a bottleneck, especially when many contracts have to be synchronized concurrently.
Since the current system is only a prototypical implementation and not intended to be used in a production environment, these scalability issues are not further addressed. As explained in section 3.2.4, light and mobile Ethereum clients are still under development and eventually the goal is to run the Ethereum client on the Android device itself. In this configuration, the system would be 100 percent scalable and standalone.

7.3 Security

This section discusses the most important security aspects including the security of Ethereum accounts and the security of transactions.
As explained in sections 6.2.2 and 6.4.2, accounts can either be managed locally on the Android device or remotely on the server running the Ethereum client. The second method should only be used for testing purposes since it involves unlocking the account on the Ethereum client and sending the wallet password over the network in plain text.
When using the WalletAccountService (section 6.4.2) the local wallet file is decrypted using the password provided by the user and the credentials are used by the RawTransactionManager to sign every transaction with the private key belonging to the account.
Using the WalletAccountService provides protection against eavesdropping because the wallet password is never sent to the Ethereum client. It also provides protection against man-in-the-middle (MITM) attacks because a transaction cannot be altered any more after it has been signed with the private key. The password for a wallet file is never persisted on the file system and the private key of an unlocked wallet file is only stored in volatile memory.
There are only two scenarios left in which an attacker could obtain or use the private key of an account:

- The attacker has root access on the operating system and can intercept the password from the user by using a key logger.
- The attacker has physical access to the phone and the account is still unlocked. In this case the attacker could use the account to sign rent or purchase contracts and transmit money to her own account.

7.4 Privacy

This section discusses how privacy issues that arise from storing and exchanging personal user data are addressed in the application. It further discusses possible privacy issues that

can arise when storing contract details on the blockchain and how they can be avoided.

7.4.1 Privacy of personal user data

All profile information including the profile images are stored on the internal storage of the Android application. This will prevent that other applications on the device can access this data. However, it does not provide protection against an attacker that has physical access to a non-encrypted file system or against an attacker that has root access on the operating system [7].
When user data is exchanged over WiFi-direct it is always encrypted using WiFi Protected Setup (WPS). As discussed in section 5.3.4, WPS uses WPA2 to encrypt and authenticate data and offers good protection against eavesdropping and brute-force attacks but it is vulnerable against MITM attacks during the short time frame in which the encryption keys are exchanged.

7.4.2 Privacy of contracts

Storing contract details in plain text on a smart contract can cause privacy issues because the addresses of the seller and the buyer are also stored on the contract and are therefore publicly accessible (section 3.3). A party that knows a person that belongs to a particular address (e.g. when it exchanged contract or user data with that person in the past) could look up details of other contracts that were signed by that person. To prevent that, the light deployment option (section 3.3) can be used. When using light deployment, only the hash of the contract details that do not have to be stored on the blockchain is stored on the smart contract.

Chapter 8

Summary and Conclusions

The introduction chapter posed multiple questions that were addressed in different chapters of the thesis.
Chapter 4 addressed legal aspects of smart contracts and answered the questions of legal validity and legal security of a contract. It was found that a purchase or rent contract is legally valid according to Swiss law when the purchase or rent item together with its price is defined and the two parties declare their declaration of will. The declaration of will can also be a handshake or a click on a digital button. A specific form is only required by law for special types of contracts. A specific form can also be used to identify the parties and to have more legal security. It was found that only the qualified electronic signature provides the same legal security as a handwritten signature on paper according to Swiss contract law. Chapter 4 also discussed how personal data could be managed on the blockchain directly by the individual in the future. Personal data could be certified by an authority using public key cryptography but only the individual would be responsible for storing and managing its data.
Chapter 5 discussed and compared different device-to-device technologies for the use case of exchanging contract details and personal data between two Android devices. The compared technologies included classic Bluetooth, Bluetooth Low Energy (BLE) and Wi-Fi direct. The chapter was concluded with the recommendation for Wi-Fi direct because it is the only technology that provides the required data rate to exchange multiple larger images in reasonable time and that also provides high protection of the transmitted data in terms of encryption key length.
Chapter 6 discussed the design and implementation of an Android application to conclude purchase and rent contracts that uses the Ethereum blockchain to store the contractual details, to enforce compliance and to process the payments.
Section 6.1 showed how the purchase and rent contracts are implemented in the Solidity script language. Both contracts use deposits to give compliance incentives to the parties and for both contracts, a light version that stores only the hash of the content attributes, was implemented to lower the deployment costs.
Section 6.3 discussed how the smart contracts are integrated into the Android client by using the Web3j Java library to generate the Java wrapper code. It was found that the generated code is not flexible enough regarding asynchronous operation handling and smart contract event subscription and because the wrapper code does not support inheri-

tance in the Solidity contracts. Therefore the wrapper code was modified and other third party libraries were used to solve the mentioned problems.
Section 6.3 also discussed how contracts and accounts are stored on the local files system and how accounts and contracts can be managed from the Android client code by using different service instances.
In section 6.4 the design and implementation of the Android client code was discussed.
The first part showed how the Android application instance and a context interface can be used to manage globally accessible objects like the services to manage accounts and contracts, and to provide them to Android Activities and Fragments. The first part also showed how smaller service classes can be used to provide specific functionality that can be accessed over the context interface and be mocked easily for testing purposes.
The second part discussed the design and implementation of the different Android Activities, Fragments and Dialogs used to manage accounts, create and interact with contracts, manage application settings and exchange data with other Android devices.
In chapter 7, the Android application was evaluated in terms of costs, scalability, security and privacy.
The specific deployment and transaction costs were measured and a simple equation was derived that can account for most of the costs for the rent and purchase contracts. It was shown that the light contract versions costs are significantly lower because less code and attributes are stored on the blockchain.
Regarding scalability It was experimentally confirmed that the the number of contracts that the application can handle in parallel is only limited by the available space on the internal storage of the Android device. It was further argued that the external Ethereum client used in this prototypical implementation will no longer be needed when light clients are available in the future.
It was concluded that the application offers a high degree of security and privacy because it signs transactions on the local Android device, stores contract and personal information only on the internal storage of the device and never sends this information unencrypted over the network.

8.1 Conclusion

The implemented Android application satisfies the functional and non-functional requirements stated at the beginning of this thesis.
It allows two parties to conclude legally valid electronic purchase and rent contracts. It is completely independent of any TTP and uses the Ethereum blockchain to store the contract details, enforce the contractual clauses and to process payments in ether. The seller can specify the contract details including the price, deposit, textual description and images and deploy the contract on the blockchain. The contract details can be exchanged either by scanning the QR-code of a contract or by using Wi-Fi direct. In the latter case, the parties can flexibly decide which personal information they want to share.
The application signs contracts on the local Android device and protects personal user data by storing it only on the internal storage and by encrypting it when it is sent over the network. The application is highly scalable and can handle a large number of contracts in parallel. It detects network errors and prevents that the user can lose money by stor-

ing contracts pre-emptively when the network fails during contract creation transactions. The contracts of the application can be extended with new attributes or functionality with moderate effort.
The application is similar to the decentralized sharing app of Bogner et all [22]. Both are mobile applications and rely on an external Ethereum client to interact with contracts on the blockchain and both applications support the conclusion of rent contracts between two parties. The most important difference is that the decentralized sharing app uses a central smart contract to manage existing rent items and is more focused on the commercial renting of items from a store. The application implemented in this thesis focuses more on direct unique transactions between two parties and also supports purchase contracts. Another difference is that the application of Bogner et all does not support the exchange of personal information between participants.

8.1.1 Limitations and future work

- A limitation of the application is the reliance on a server that hosts the Ethereum client to interact with the smart contracts on the blockchain. An external sever can limit the scalability when multiple Android clients are connected to it. It also limits the reliability of the system because it becomes an additional point of failure.
  As of August 2017, both the light client protocol [35] and the implementation [36] are still under development. As soon as the light client implements the Ethereum RPC API, the external client can potentially be replaced with a light client on the Android device and then the application would become completely independent of external infrastructure.

- Another limitation in the current implementation is the lack of legal security of the contracts. At the moment, a contract is signed with the public key of both parties. As explained in chapter 4, the personal information exchanged between the parties does not provide the necessary authenticity to tie a person to a specific public key. This can become an issue when contracts with a high value are concluded or when the contract requires handwritten signatures to be valid. As outlined in section 4.5, the SwissSign certification infrastructure could be used to sign a pdf document and then the hash of this document could be deployed together with the smart contract. A decentralized solution that would allow the individual to manage its personal data on the blockchain has been discussed in section 4.6. However, this solution would require a globally accepted certification process for personal data that does not exist at the moment.

- To improve the expandability of the application, it would be beneficial if the attributes and transaction functions for the Java wrapper class could be generated directly from the Solidity contract in the future. For various reasons explained in section 6.3.2, the auto generated wrapper classes of the Web3j library are not flexible enough and do not satisfy all requirements of the application. However, the auto generated code could be taken as a starting point and modified by a custom code generator to fit the required structure. This would involve:

- identifying common code in the ABI definition of the smart contracts and generating a common base class.
- deleting common functions and attributes from the generated classes
- Replacing asynchronous calls used in Web3j with JDeferred Promise API calls
- Adding code for event subscription and unsubscription
- generating the interfaces for all classes
- Generating service methods for deploying and loading the contracts

Bibliography

[1] Android, Application. URL:https://developer.android.com/reference/android/app/Application.html, Last visited May 9, 2017

[2] Android BroadCastManager. URL:https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html, Last visited May 9, 2017

[3] Android, BroadCastReceiver. URL:https://developer.android.com/reference/android/content/BroadcastReceiver.html, Last visited May 9, 2017

[4] Android, PreferenceFragment. URL:https://developer.android.com/reference/android/preference/PreferenceFragment.html, Last visited May 9, 2017

[5] Android, RecyclerView. URL:https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html, Last visited July 30, 2017

[6] Android, SharedPreferences. URL:https://developer.android.com/reference/android/content/SharedPreferences.html, Last visited May 9, 2017

[7] Android, Security Tipps. URL:https://developer.android.com/training/articles/securitytips.html, Last visited July 31, 2017

[8] Android, SurfaceView. URL:https://developer.android.com/reference/android/view/SurfaceView.html, Last visited May 9, 2017

[9] Android, Wifi-P2P. URL:https://developer.android.com/guide/topics/connectivity/wifip2p.html, Last visited July 31, 2017

[10] A.M. Antonopoulos: *Mastering Bitcoin, "Keys, Addresses, Wallets";* O'Reilly Media , Sebastopol, CA, USA, April 2014, ISBN: 978-1-449-37404-4

[11] A.M. Antonopoulos: *Mastering Bitcoin, "Transactions";* O'Reilly Media , Sebastopol, CA, USA, April 2014, ISBN: 978-1-449-37404-4

[12] A.M. Antonopoulos: *Mastering Bitcoin, "The Blockchain";* O'Reilly Media , Sebastopol, CA, USA, April 2014, ISBN: 978-1-449-37404-4

[13] A.M. Antonopoulos: *Mastering Bitcoin, "Mining and Consensus";* O'Reilly Media , Sebastopol, CA, USA, April 2014, ISBN: 978-1-449-37404-4

[14] P. Arana: *Benefits and Vulnerabilities of Wi-Fi Protected Access 2 (WPA2).* URL:http://cs.gmu.edu/~yhwang1/INFS612/Sample_Projects/Fall_06_GPN_6_Final_Report.pdf, Last visited July 30, 2017

[15] N.Atzi, M.Bartoletti, T.Cimoli: *A survay of attacks on Ethereum smart contrats.* URL:https://eprint.iacr.org/2016/1007.pdf, Last visited August 12, 2017

[16] Bitcoin, Block chain. URL:https://en.bitcoin.it/wiki/Blockchain", Last visited May 9, 2017

[17] Bitcoin, Double-spending. URL:https://en.bitcoin.it/wiki/Double-spending,Last visited May 9, 2017

[18] Bitcoin, Script. URL:https://en.bitcoin.it/wiki/Script, Last visited May 9, 2017

[19] Bitcoin, Smart property. URL:https://en.bitcoin.it/wiki/SmartProperty, Last visited May 9, 2017

[20] Bitcoin, Proof of work. URL:https://en.bitcoin.it/wiki/Proof_of_work, Last visited May 9, 2017

[21] Bluetooth radio interface, modulation and channels. URL:http://www.radio-electronics.com/info/wireless/bluetooth/radio-interface-modulation.php, Last visited July 21, 2017

[22] A.Bogner, M.Chanson, A.Meeuw: *A Decentralised Sharing App running a Smart Contract on the Ethereum Blockchain;* 6th International Conference on the Internet of Things(IoT16), Stuttgart, Germany, November 2016

[23] Brooklyn Microgrid. URL:http://microgridmedia.com/brooklyn-startup-broadens-solar-power-access-with-p2p-energy-exchange/, Last visited May 9, 2017

[24] Browser Solidity online compiler. URL:https://ethereum.github.io/browser-solidity/, Last visited July 30, 2017

[25] E.Bucher: *Zustandekommen des Vertrages.* URL:http://www.eugenbucher.ch/pdf_files/Bucher_ORAT_10.pdf, Last visited May 9, 2017

[26] E.Bucher: *Kaufvertrag im Allgemeinen.* URL:http://www.eugenbucher.ch/pdf_files/Bucher_ORBT_03.pdf, Last visited May 9, 2017

[27] V.Buterin: *A next-generation smart contract and decentralized application platform.* White paper, 2014

[28] D.Camps-Murr, A.Garcia-Saavedra, P.Serrano: *Device to device communications with WiFi Direct: overview and experimentation.* URL:http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=81D8D0CEA0130A332E2EB14EC8563A1E?doi=10.1.1.725.7590&rep=rep1&type=pdf, Last visited July 20, 2017

[29] CryptoCompare.com. URL:https://www.cryptocompare.com/, Last visited May 9, 2017

[30] Cryptocurrency market capitalization. URL:https://coinmarketcap.com/, Last visited May 9, 2017

[31] K. Delmolino, M. Arnett, A.E. Kosba, A. Miller, E. Shi: *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab.* IACR Cryptology ePrint Archive, 2015 (2015), 460, URL:http://fc16.ifca.ai/bitcoin/papers/DAKMS16.pdf, Last visited August 12, 2017

[32] Digix White paper. URL:https://dgx.io/whitepaper.pdf, Last visited May 9, 2017

[33] Digix.io. URL:https://digix.io/, Last visited May 9, 2017

[34] Ebay, general terms of agreement. URL:http://pages.ebay.ch/help/policies/user-agreement.html, Last visited May 9, 2017

[35] Ethereum, Light Client Protocol. URL:https://github.com/ethereum/wiki/wiki/Light-client-protocol, Last visited May 9, 2017

[36] Ethereum, Light Client implementation. URL:https://github.com/paritytech/parity/wiki/Light-Client, Last visited August 4, 2017

[37] Ethereum, Documentation. URL:http://www.ethdocs.org/en/latest/, Last visited May 9, 2017

[38] Ethereum, Average exchange rate since May 1, 2017. URL:https://poloniex.com/public?command=returnChartData&currencyPair=USDT_ETH&start=1493596800&end=1501286400&period=86400", Last visited July 28, 2017

[39] Ethereum, Gas costs. URL:https://docs.google.com/spreadsheets/d/1m89CVujrQe5LAFJ8-YAUCcNK950dUzMQPMJBxRtGCqs/edit#gid=0, Last visited July 27, 2017

[40] Ethereum, Solidity. URL:https://solidity.readthedocs.io/en/develop/, Last visited May 9, 2017

[41] Ethereum, Solidity, Events. URL:https://solidity.readthedocs.io/en/latest/contracts.html#events, Last visited May 9, 2017

[42] Ethereum, Solidity, Exceptions. URL:https://solidity.readthedocs.io/en/latest/control-structures.html#exceptions, Last visited May 9, 2017

[43] Ethereum, Solidity, Fallback function. URL:https://solidity.readthedocs.io/en/latest/contracts.html#fallback-function, Last visited May 9, 2017

[44] Ethereum, Solidity, Functions. URL:https://solidity.readthedocs.io/en/latest/control-structures.html#function-calls, Last visited May 9, 2017

[45] Ethereum, Solidity, Global variables and functions. URL:https://solidity.readthedocs.io/en/latest/units-and-global-variables.html, Last visited May 9, 2017

[46] Ethereum, Solidity, Modifiers. URL:https://solidity.readthedocs.io/en/latest/contracts.html#modifiers, Last visited May 9, 2017

[47] Ethereum, Solidity, Visibility. URL:https://solidity.readthedocs.io/en/latest/contracts.html#visibility-and-getters, Last visited May 9, 2017

[48] Ethereum, Solidity, Remote purchase contract. URL:https://solidity.readthedocs.io/en/develop/solidity-by-example.html#safe-remote-purchase, Last visited May 9, 2017

[49] Ethereum, Solidity, Security. URL:https://solidity.readthedocs.io/en/latest/security-considerations.html, Last visited May 9, 2017

[50] Ethereum, Solidity, Types. URL:https://solidity.readthedocs.io/en/latest/types.html#types, Last visited May 9, 2017

[51] Ethgasstation. URL:https://ethgasstation.info, Last visited July, 16, 2017

[52] Ez-VCard. URL:https://github.com/mangstadt/ez-vcard#mavengradle, Last visited May 9, 2017

[53] Federal law about the electronic signature (ZertES). URL:https://www.admin.ch/opc/de/classified-compilation/20011277/index.html, Last visited May 9, 2017

[54] E.Ferro, F.Potorti: Bluetooth and Wi-Fi Wireless Protocols: A survey and a comparison [2004]. URL:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.1709&rep=rep1&type=pdf, Last visited July 21, 2017

[55] Go-Ethereum. URL:https://github.com/ethereum/go-ethereum, Last visited May 9, 2017

[56] Go-Ethereum, javascript console. URL:https://github.com/ethereum/go-ethereum/wiki/JavaScript-Console, Last visited May 9, 2017

[57] Go-Ethereum, JSON-RPC. URL:https://github.com/ethereum/wiki/wiki/JSON-RPC, Last visited May 9, 2017

[58] GSON. URL:https://github.com/google/gson, Last visited May 9, 2017

[59] K.Haataja, K.Hyppönen, S.Pasanen, P.Toivanen: *Bluetooth Security Attacks, "Overview of Bluetooth Security";* SpringerBriefs in Computer Science, 2013. URL:http://www.springer.com/cda/content/document/cda_downloaddocument/9783642406454-c2.pdf?SGWID=0-0-45-1434420-p175453762, Last visited August 12, 2017

[60] M.Hardesty: *Simple security for wireless,* URL:http://news.mit.edu/2011/secure-wifi-0822, Last visited July 24, 2017

[61] F.Hawlitschek, T.Teubner, G.Henner: *Understanding the Sharing Economy - Drivers and Impediments for Participation in Peer-to-Peer Rental;* 49th Hawaii International Conference on System Sciences(HICSS), Koloa, HI, USA, Januar 2016, ISBN: 978-0-7695-5670-3.

[62] M.Hay: *How GDPR plus blockchain leads to the future of self-sovereign identity.* URL:http://www.janrain.com/blog/how-gdpr-plus-blockchain-leads-to-the-future-of-self-sovereign-identity/, Last visited April 9, 2016

[63] IEEE 802.11 Wi-Fi Standards. URL:http://www.radio-electronics.com/info/wireless/wi-fi/ieee-802-11-standards-tutorial.php, Last visited July 21, 2017

[64] IEEE 802.11n Standard. URL:http://www.radio-electronics.com/info/wireless/wi-fi/ieee-802-11n.php, Last visited July 21, 2017

[65] Java, CompleteableFuture. URL:https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html, Last visited May 9, 2017

[66] Java, Future. URL:https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html, Last visited May 9, 2017

[67] Java, Observable. URL:http://reactivex.io/documentation/observable.html, Last visited May 9, 2017

[68] Java, ScheduledExecutionService. URL:https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledExecutorService.html, Last visited May 9, 2017

[69] JDeffered. URL:https://github.com/jdeferred/jdeferred, Last visited May 9, 2017

[70] JQuery. URL:https://github.com/jquery/jquery, Last visited May 9, 2017

[71] L.Jin-Shyan, S.Yu-Wei, S.Chung-Chou: *A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi;* 33rd Annual Conference of the IEEE Industrial Electronics Society (IECON), 2007, Taipei, Taiwan. URL:http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.477.1670&rep=rep1&type=pdf, Last visited July 20, 2017

[72] Mobile Vision, Barcode API. URL:https://developers.google.com/vision/android/barcodes-overview, Last visited May 9, 2017

[73] Mobile Vision, BarcodeDetector. URL:https://developers.google.com/android/reference/com/google/android/gms/vision/barcode/BarcodeDetector, Last visited May 9, 2017

[74] Mobile Vision, CameraSource. URL:https://developers.google.com/android/reference/com/google/android/gms/vision/CameraSource, Last visited May 9, 2017

[75] S.Nakamoto: Bitcoin: A peer-to-peer electronic cash system [2008]. URL:https://bitcoin.org/bitcoin.pdf, Last visited May 9, 2017

[76] Public Key Infrastructure (PKI) providers in Switzerland. URL:https://www.sas.admin.ch/sas/de/home/akkreditiertestellen/akkrstellensuchesas/pki.html, Last visited August 4, 2017

[77] Promises and Futures. URL:https://en.wikipedia.org/wiki/Futures_and_promises, Last visited May 9, 2017

[78] Qreader API. https://github.com/nisrulz/qreader, Last visited August 12, 2017

[79] QrGen. URL:https://github.com/kenglxn/QRGen, Last visited May 9, 2017

[80] A.Schmid, T.Pachmann: Der Vertragsabschluss beim Onlinehandel. URL:http://www.pachmannlaw.ch/publikationen/pdf/Der_Vertragsabschluss_beim_Onlinehandel.pdf, Last visited May 9, 2017

[81] Shocard App. URL:https://shocard.com/, Last visited May 9, 2017

[82] Shocard Whitepaper. URL:https://shocard.com/wp-content/uploads/2016/11/travel-identity-of-the-future.pdf, Last visited May 9, 2017

[83] SuisseId mobile service. URL:https://www.post.ch/en/business/a-z-of-subjects/suisseid/products-and-prices/suisseid-mobile-service, Last visited August 4, 2017

[84] SuisseId specification. URL:https://www.suisseid.ch/sites/default/files/uploads/suisseid_specification_13.pdf, Last visited August 4, 2017

[85] SuisseId Multi Signing Platform. URL:https://www.multisigning.ch/, Last visited August 4, 2017

[86] Swiss Contract Law (Obligationenrecht). URL:https://www.admin.ch/opc/de/classified-compilation/19110009/index.html#a1, Last visited May 28, 2017

[87] N.Szabo: The idea of smart contracts [1997]. URL:http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_idea.html, Last visited May 9, 2017

[88] K. Townsend , R. Davidson , C.Cufi: *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking, "Introduction";* O'Reilly, Sebastopol, CA, USA, May 2014, ISBN: 1491949511

[89] K. Townsend , R. Davidson , C.Cufi: *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking, "Protocol Basics";* O'Reilly, Sebastopol, CA, USA, May 2014, ISBN: 1491949511

[90] K. Townsend , R. Davidson , C.Cufi: *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking, "GAP (Advertising and Connections)";* O'Reilly, Sebastopol, CA, USA, May 2014, ISBN: 1491949511

[91] K. Townsend , R. Davidson , C.Cufi: *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking, "GATT (Services and Characteristics)";* O'Reilly, Sebastopol, CA, USA, May 2014, ISBN: 1491949511

[92] Transactive Grid. URL:https://www.slideshare.net/JohnLilic/transactive-grid, Last visited May 9, 2017

[93] Vertragsrecht.ch . URL:https://www.vertragsrecht.ch, Last visited May 9, 2017

[94] Viehböck S. Brute forcing Wi-Fi Protected Setup. URL:https://sviehb.files.wordpress.com/2011/12/viehboeck_wps.pdf, Last visited July 11, 2017

[95] Web3j. URL:https://github.com/web3j/web3j, Last visited May 9, 2017

[96] Web3j. Documentation. URL:https://docs.web3j.io/, Last visited May 9, 2017

[97] Web3j. Filters. URL:https://docs.web3j.io/filters.html, Last visited May 9, 2017

[98] Web3j, Smart Contracts. URL:https://docs.web3j.io/smart_contracts.html, Last visited August 2, 2017

[99] Web3j, Transactions. URL:https://web3j.readthedocs.io/en/latest/transactions.html, Last visited July 30, 2017

[100] Web3js. URL:https://github.com/ethereum/wiki/wiki/JavaScript-API, Last visited May 9, 2017

[101] G.Zyskind , O.Nathan, A.Pentland: *Decentralizing Privacy: Using Blockchain to Protect Personal Data;* 36th IEEE Symposium on Security and Privacy Workshops (SPW), Fairmont, San Jose, CA, USA, May 2015. URL: http://inpluslab.sysu.edu.cn/files/Paper/Security/Decentralizing_Privacy__Using_Blockchain_To_Protect_Personal_Data.pdf, Last visited August 12, 2017

Abbreviations

B2C	Business-to-Consumer
BLE	Bluetooth low energy
C2C	Consumer-to-Consumer
DAPP	Distributed application
EVM	Ethereum Virtual Machine
MITM	Man-in-the-middle-attack
NFC	Near field communication
OR	Obligationenrecht (Swiss contract law)
P2P	Peer-to-Peer
PoA	Proof of Asset
TTP	Trusted third party
UTXO	Unspent transaction output
Wi-Fi	Wireless fidelity

Glossary

Address Adresses in a cryptocurrency are used to send and receive transactions on the network. An address is a fixed length string that is often derived from a public key.

ASIC ASIC is an acronym for "Application Specific Integrated Circuit". ASICs are silicon chips specifically designed to do a single task. In the case of bitcoin, they are designed to process SHA-256 hashing problems to mine new bitcoins.

Bitcoin The well known cryptocurrency, based on the proof-of-work blockchain.

Blockchain A blockchain is a type of distributed ledger, comprised of unchangable, digitally recorded data in packages called blocks (rather like collating them on to a single sheet of paper). Each block is then 'chained' to the next block, using a cryptographic signature. This allows block chains to be used like a ledger, which can be shared and accessed by anyone with the appropriate permissions

Block Reward The reward given to a miner which has successfully hashed a transaction block. Block rewards can be a mixture of coins and transaction fees, depending on the policy used by the cryptocurrency in question, and whether all of the coins have already been successfully mined. The current block reward for the Bitcoin network is 25 bitcoins for each block.

Cryptocurreny A form of digital currency based on mathematics, where encryption techniques are used to regulate the generation of units of currency and verify the transfer of funds. Furthermore, cryptocurrencies operate independently of a central bank.

Distributed Ledger Distributed ledgers are a type of database that are spread across multiple sites, countries or institutions. Records are stored one after the other in a continuous ledger. Distributed ledger data can be either "permissioned" or "unpermissioned" to control who can view it.

Genesis Block The first block in a block chain.

Hashrate The number of hashes that can be calculated by a miner in a period of time.

Mining The process by which transactions are verified and added to a blockchain. This process of solving cryptographic problems using computing hardware also triggers the release of cryptocurrencies.

P2P Peer-to-peer (P2P) refers to the decentralized interactions that happen between at least two parties in a highly interconnected network. P2P participants deal directly with each other through a single mediation point.

Private Key A private key is a string of data that shows you have access to bitcoins in a specific wallet. Private keys can be thought of as a password; private keys must never be revealed to anyone but you, as they allow you to spend the bitcoins from your bitcoin wallet through a cryptographic signature.

Proof-of-Work A system that ties mining capability to computational power. Blocks must be hashed, which is in itself an easy computational process, but an additional variable is added to the hashing process to make it more difficult. When a block is successfully hashed, the hashing must have taken some time and computational effort. Thus, a hashed block is considered proof of work.

Smart Contracts Smart contracts are contracts whose terms are recorded in a computer language instead of legal language. Smart contracts can be automatically executed by a computing system, such as a suitable distributed ledger system.

Transaction Fee A small fee imposed on some transactions sent across the bitcoin network. The transaction fee is awarded to the miner that successfully hashes the block containing the relevant transaction.

List of Figures

3.1 Outputs of transactions A and B as inputs of transaction C. 8
3.2 The structure of a block in the blockchain. 9
3.3 The structure of a Merkle Tree. 10
3.4 Validation of a transaction in a Merkle Tree. 10
3.5 Execution of a smart contract on the blockchain. 16

5.1 Bluetooth protocol stack. 35
5.2 Bluetooth piconet structure. 36
5.3 Overview of the BLE protocol stack. 39
5.4 Wi-Fi network topologies. 43

6.1 Overview of the architecture. 49
6.2 Data model for local accounts and contracts. 67
6.3 Application core structure. 79
6.4 Account Activity 80
6.5 Account Import Dialog 80
6.6 Overview Activity 81
6.7 Contract Create Dialog 81
6.8 Create Activity 81
6.9 Detail Fragment before identity verification 82
6.10 Contract Export Dialog 82
6.11 Imported Profile 82
6.12 Detail Fragment with contract interactions 82
6.13 The Profile Activity 83
6.14 The Setting Activity 84
6.15 P2P data exchange protocol. 87
6.16 The P2pExportDialog 88

A.1 Structure of the Account Activity 118

A.2 Structure of the Overview Activity 119
A.3 Structure of the Create Activity 120
A.4 Structure of the Detail Activity 121
A.5 Structure of the Profile Activity 122
A.6 Structure of the Setting Activity 123
A.7 Structure of the Wi-Fi direct components 124

List of Tables

5.1 Overview of different IEEE 802.11 standards 42
5.2 Overview of different wireless technologies 46

7.1 Measured deployment costs for different deployment configurations . . 92
7.2 Estimated deployment costs for different deployment configurations . 92
7.3 Real and estimated transaction costs 93

Appendix A

Diagrams

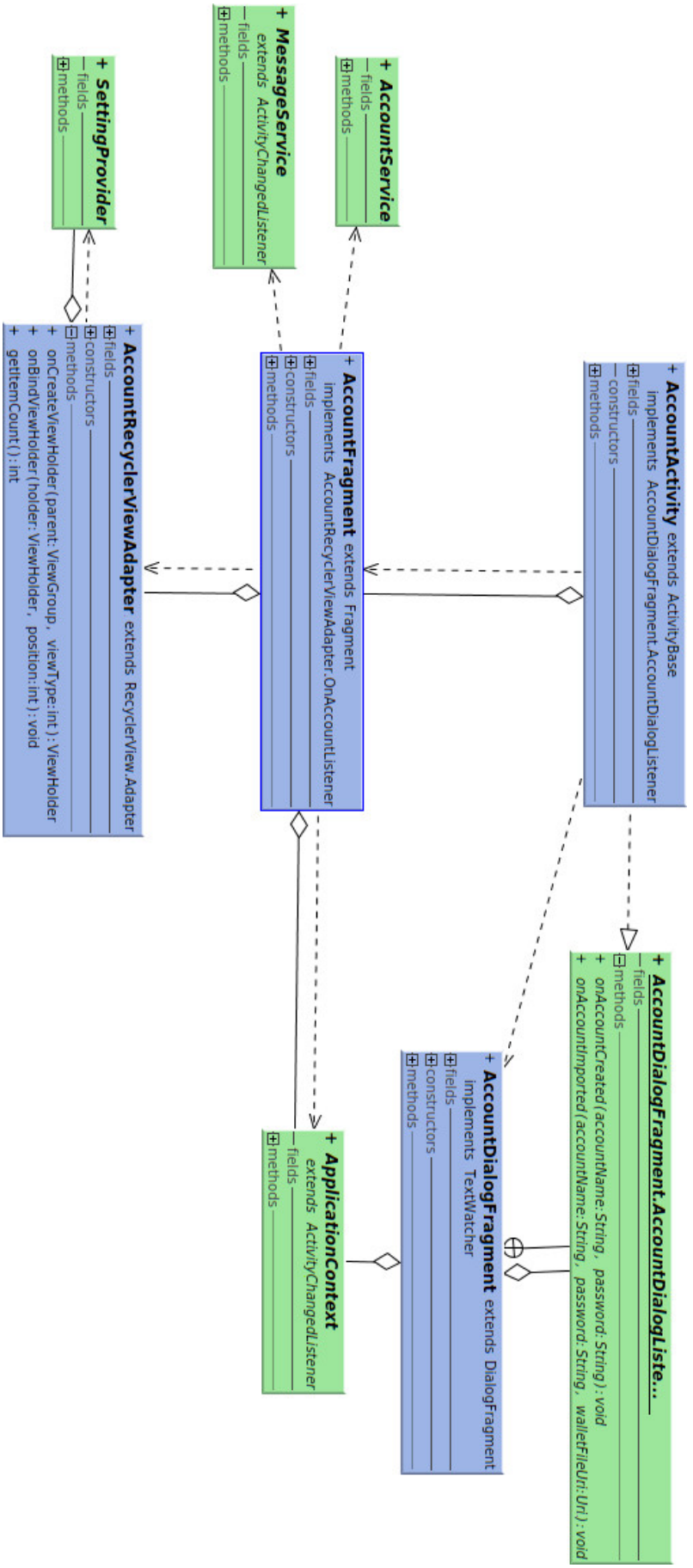

Figure A.1: Structure of the Account Activity

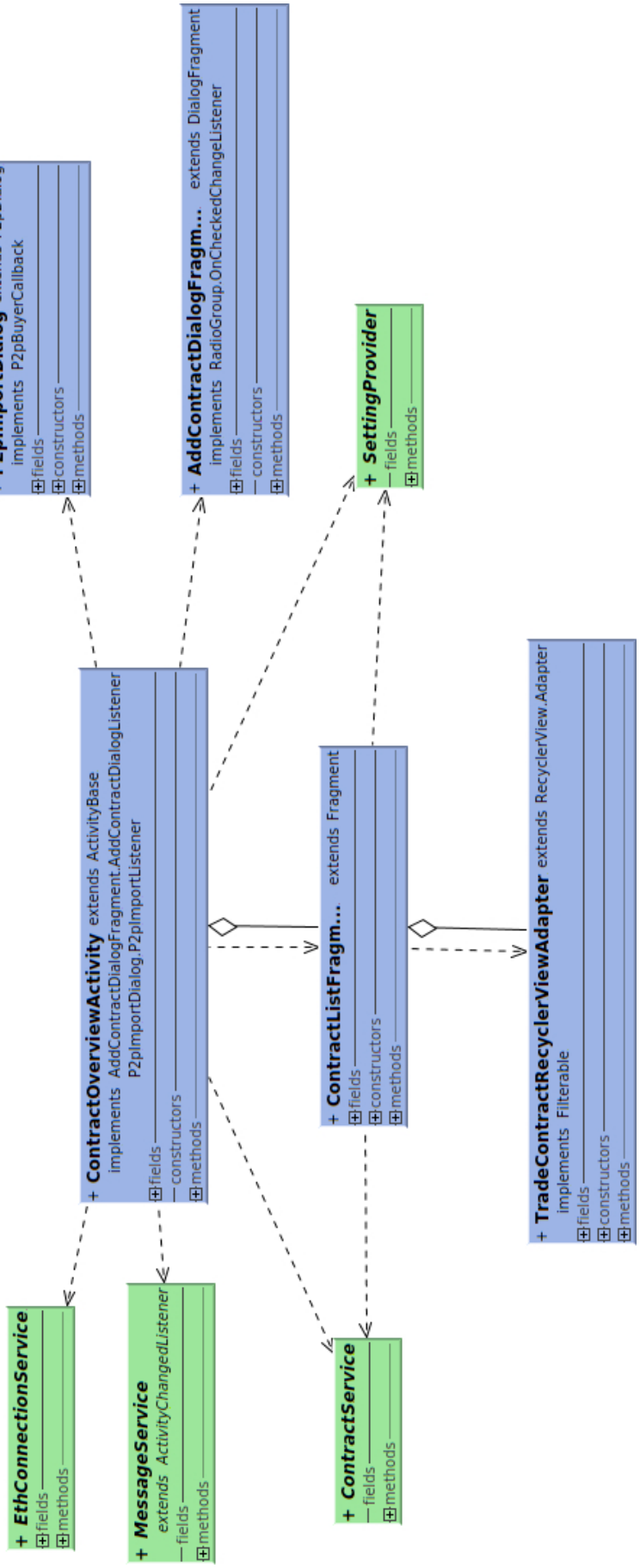

Figure A.2: Structure of the Overview Activity

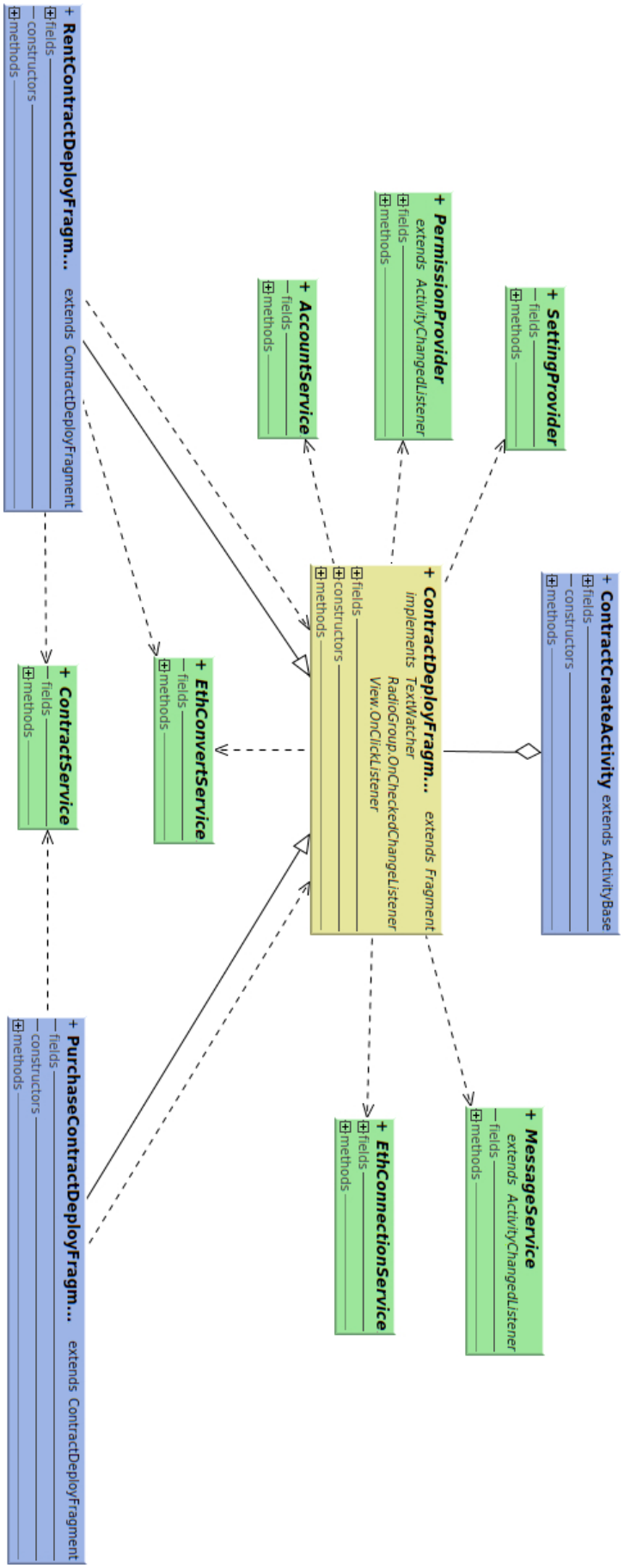

Figure A.3: Structure of the Create Activity

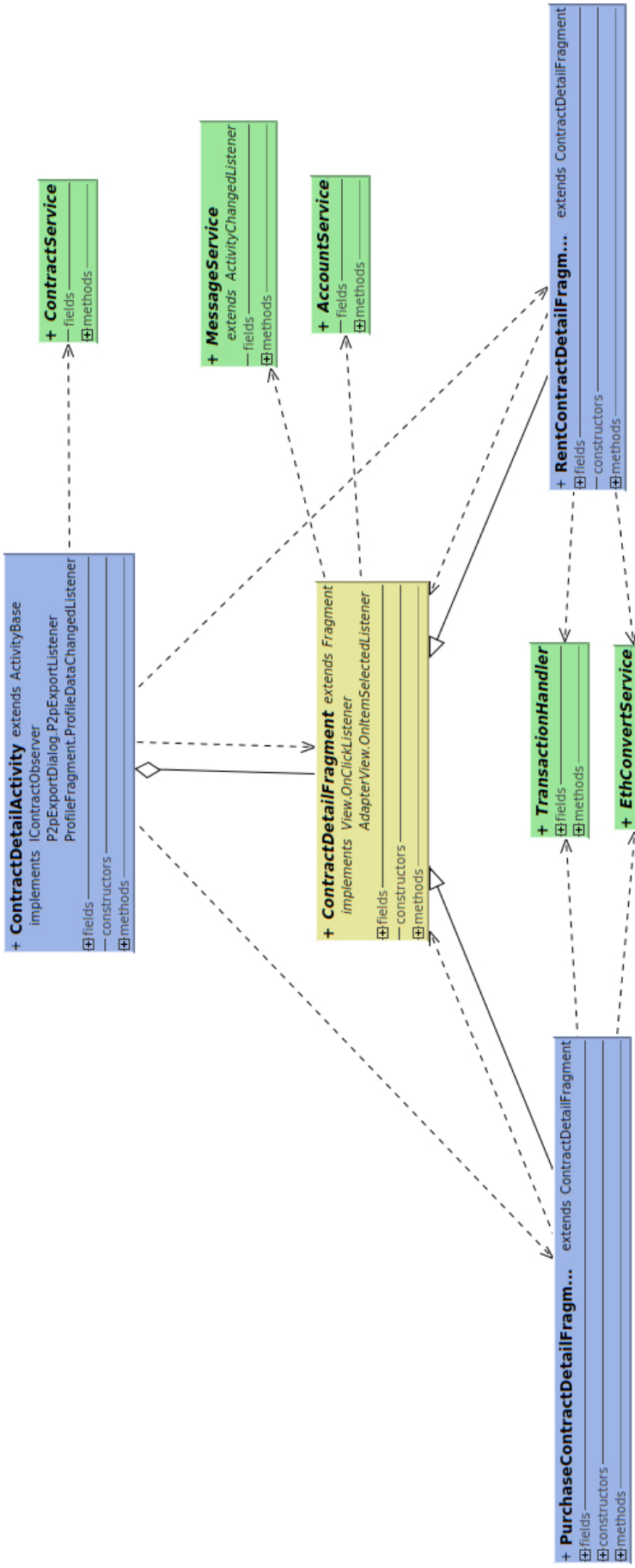

Figure A.4: Structure of the Detail Activity

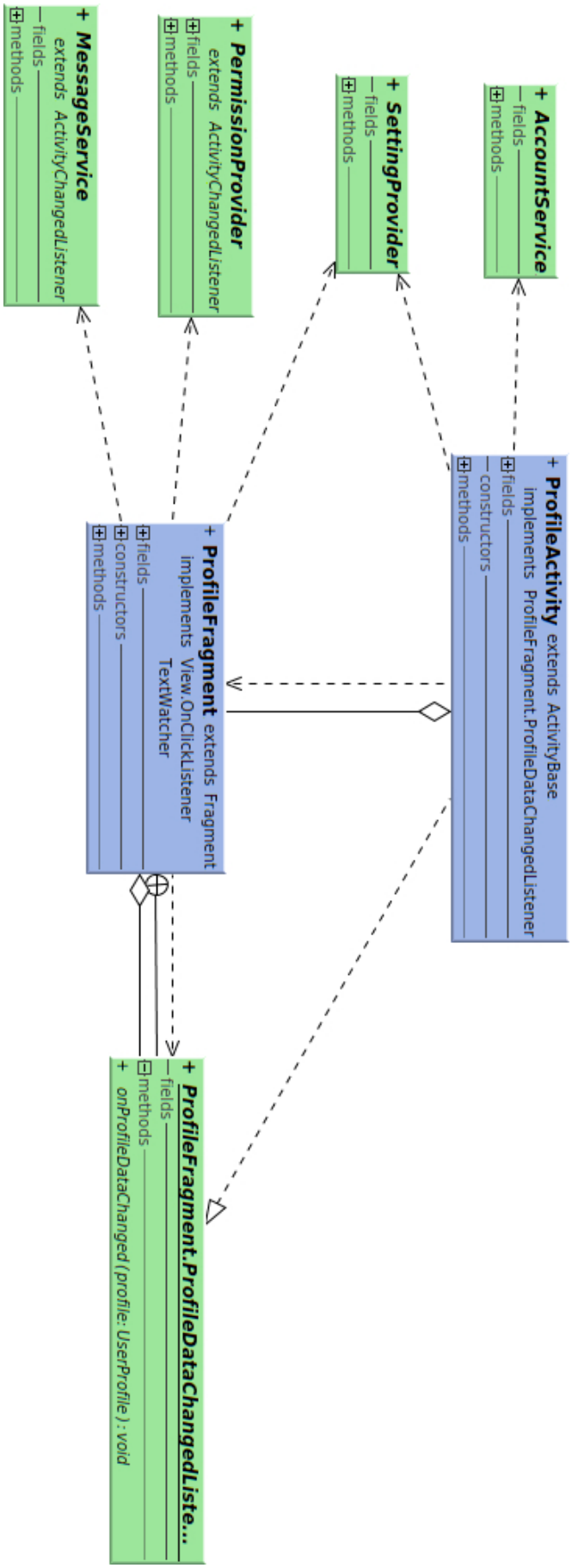

Figure A.5: Structure of the Profile Activity

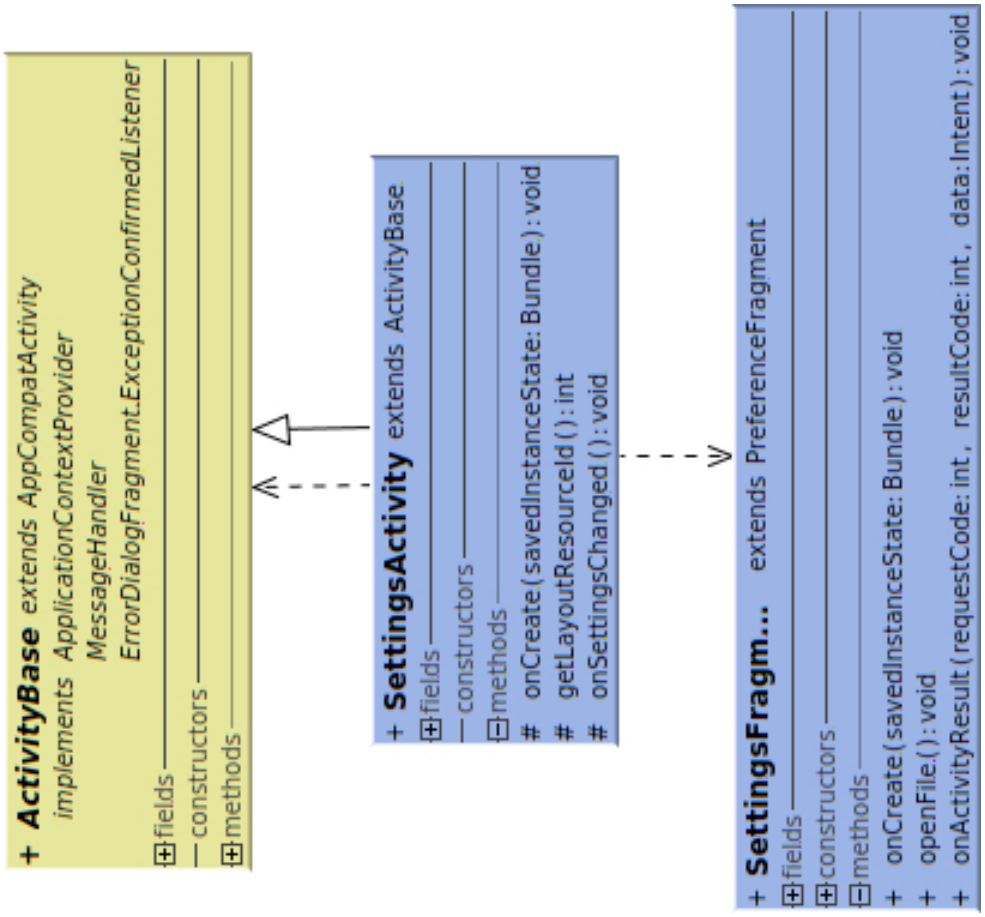

Figure A.6: Structure of the Setting Activity

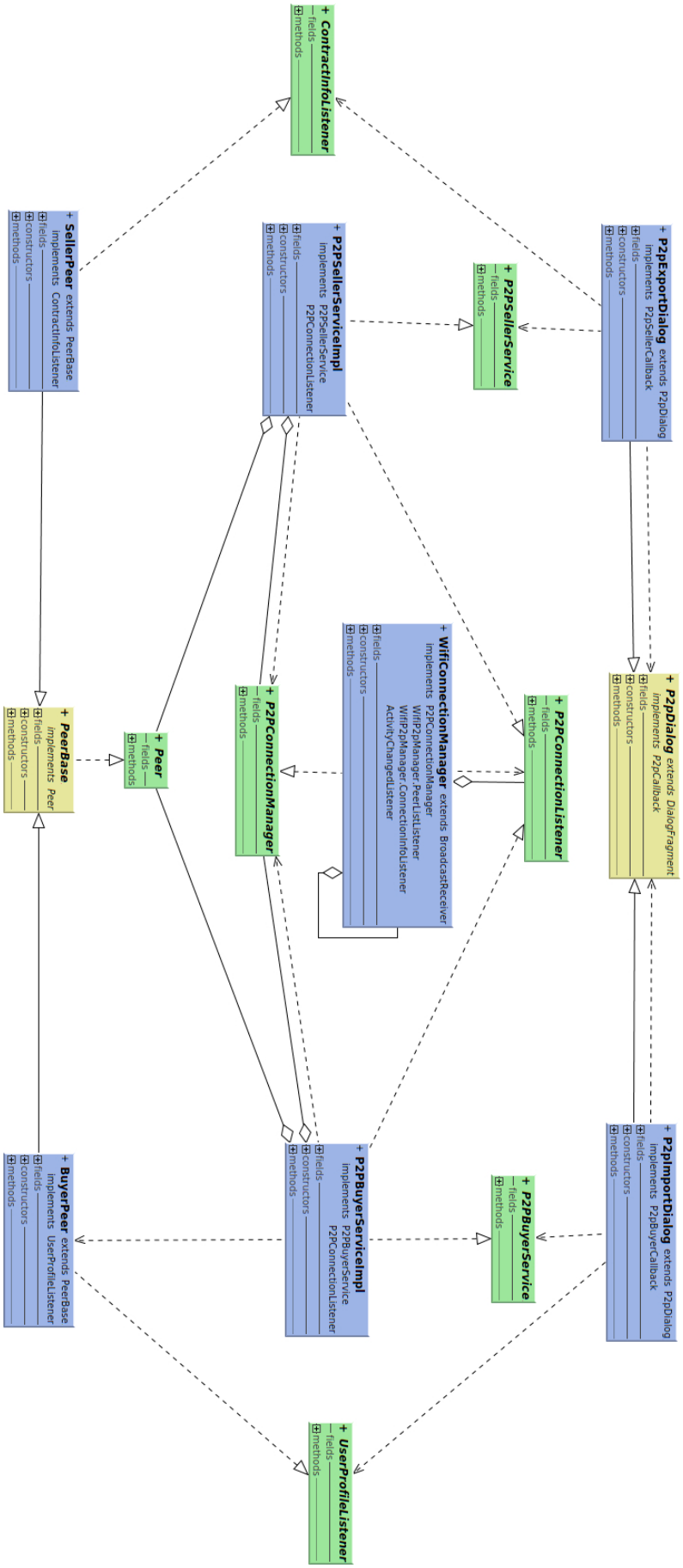

Figure A.7: Structure of the Wi-Fi direct components

Appendix B

Installation Guidelines

B.1 Installation

The application can be installed directly from the .apk on the CD. In order to install the application you have to enable the setting that allows the installation from unknown sources. On Samsung devices, this option is usually in the *Settings* -> *Security* menu.

1. Copy the .apk file from the "application" folder to your phones external storage (e.g. into the Download folder)
2. Browse to the .apk file with the internal file browser of your phone and click on it to install the application

B.1.1 Compilation from source

To compile the application from source, the Android SDK and the Android Support Library v7:23.4.0 or higher must be installed on the machine.
The path the the Android SDK on your system must be specified. For linux systems, this can be done with the following command assuming that the SDK location is in the directory "Android/Sdk" of your home directory:

```
export ANDROID_HOME=$HOME/Android/Sdk
```

1. Copy the "SmartContract" folder in the "application" directory to a directory on your file system
2. Build the application .apk file with either of those two options:
   - Command line (recommended):
     (a) Step into the root folder of the checked out project
     (b) Execute the command: *./gradlew build* to build all the resources (*./gradlew clean* in order to clean all the build files)
   - Android Studio:
     (a) Import the checked out project into Android Studio

   (b) Go to Build –> Rebuild Project
3. The SmartContract application file is in the following path: "/app/build/outputs/apk".

B.2 Testing

The application contains unit-tests to test the most important service classes (section 6.3.4) and it contains instrumented unit-tests to test the logic of the UI components (Activities, Fragments) of the application.

B.2.1 Run unit tests

1. Copy the "SmartContract" folder in the "application" directory to a directory on your file system
2. run the tests with either of those options:
   - Console:
     (a) Step into the root folder of the project
     (b) Execute the command: *./gradlew test*
     (c) The test results are in the directory: "app/build/test-results"
   - Android Studio:
     (a) Import the checked out project into Android Studio
     (b) Right click on the test package -> run tests

B.2.2 Run instrumented unit tests

To run the instrumented tests, an Android device with at least SDK version 21 must be connected via USB to the computer from which the tests are executed.
On the connected device, the developer options must be enabled and "USB debugging" must be enabled.
Unfortunately, the tests hang when they are executed all in series. It is recommended to execute every test class manually in the Android Studio.

1. Copy the "SmartContract" folder in the "application" directory to a directory on your file system
2. run the tests with either of those options:
   - Console:
     (a) Step into the root folder of the project
     (b) Execute the command: *./gradlew connectedAndroidTest*
     (c) The test results are displayed on the console
   - Android Studio (recommended):
     (a) Import the checked out project into Android Studio
     (b) Right click on the Android test package -> run tests

B.3 Getting started

In order to create smart contracts and to interact with them, an Ethereum client must be installed on a remote machine and it must be accessible from the local network.

To install the Go-Ethereum client, follow the instructions on `https://github.com/ethereum/go-ethereum`.
On an Ubuntu system, you can run the following commands on the console:

```
sudo apt-get install software-properties-common
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install ethereum
```

To run the geth client on a test network, you can execute the following command on the console:

```
geth --rpc --rpcaddr "0.0.0.0" --rpcapi personal,db,eth,net,web3
    --rpccorsdomain="*" --dev
```

The *–rpc*, *–rpcaddr* and rpccorsdomain parameters are necessary to enable the RPC interface over HTTP and to enable access to processes outside the local host.

Make sure that the host of the client and the Android device are both connected to the same WLAN network and start the Android application.

1. Create a new account on the Ethereum host with the “personal.newAccount” function of the javascript console (section 6.2.2).
2. In the application, go to *menu* -> *settings* -> *Host* and enter the IP address of the Ethereum host.
3. If you want to use existing accounts that were created on the Ethereum host, you can set the *setting* -> *Account management* to ”REMOTE”. Note that this option should only be used in test networks, because it will send the wallet password in plain text over the network and is therefore inherently unsafe!
4. Go back to *menu* -> *accounts* and either unlock an account on the host, create a new local account or import an existing wallet file.
5. Ensure that the unlocked account has enough funds. If you use a remote account, you can set it as the mining account of the test network by using the “miner.setEtherbase” command on the javascript console. You can also send money to a local account by using the “eth.sendTransaction” function (section 6.2.2).
6. Click on the *home* button and start creating or importing contracts!

For more information on how to use the single Activities, have a look at section 6.4.2

Appendix C

Contents of the CD

- **application:** contains the application .apk file.
  - **SmartContract:** Contains the gradle build files and settings to build the application and the source files of the application, the unit tests and the instrumented unit tests.
- **related work:** Contains the PDF and HTML documents of the related work material.
- **solidity contracts:** Contains the Solidity source files of the full and light versions of the smart contracts that are used in the application.
- **thesis:** Contains the PDF and PS documents of the written report.
  - **sources:** Contains all the latex source files used to generate the written report
  - **figures:** Contains the figures that are used in the report and their source files.