Eine Blockchain / Ethereum Anwendung und ihre Auswirkungen auf Smart Cities

BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

University of Applied Sciences
HOCHSCHULE
EMDEN·LEER

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M.Sc.)
Medieninformatik

Björn Lakeberg

01.11.2015 - 12.06.2016

Betreuer

Prof. Dr. Stefan Edlich (Berlin)
Prof. Dr. Martin Schiemann-Lillie (Emden)

Erklärung

Soweit meine Rechte berührt sind, erkläre ich mich einverstanden, dass die vorliegende Masterarbeit Angehörigen der Beuth Hochschule für Technik Berlin und der Hochschule Emden/Leer für Studium, Lehre und Forschung uneingeschränkt zugänglich gemacht werden darf.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Masterarbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben sind.

Datum, Unterschrift

Inhaltsverzeichnis

1 Einleitung 6

1.1 Motivation für Ethereum 7

1.2 Ziel und Aufbau dieser Arbeit 8

2 Einführung in Smart Cities 9

2.1 Motivation und Hintergrund 9

2.2 Verwandte Konzepte 10

2.3 Smart City Definitionen 12

2.4 Modell nach Giffinger und Manville 15

2.5 Erfolgreiche Beispiele in Europa 17

3 Einführung in die Blockchain 20

3.1 Motivation und Hintergrund 20

3.2 Kryptografische Grundlagen 22

3.2.1 Symmetrische Verschlüsselung 22

3.2.2 Asymmetrische Verschlüsselung 23

3.2.3 Hashfunktionen 24

3.3 Aufbau der Blockchain 26

3.4 P2P-Netzwerk 29

3.5 Transaktionen 32

3.6 Anwendungsszenarien 36

3.6.1 Blockchain 1.0 - Währungen und Zahlungssysteme 36

3.6.2 Blockchain 2.0 - Smart Properties und Contracts 39

3.6.3 Blockchain 3.0 - Anwendungen und App-Coins 42

4 Ethereum als Plattform 47

4.1 Historie und Entwicklung 47

4.2 Ethereum Protokoll 49

4.2.1 State Transition Funktion 49

4.2.2 Ethereum Blockchain 52

4.2.3 Ethereum Virtual Machine 57

4.3 Dapps und Contracts 59

4.4 Ethereum Clients ... 64

5 Implementierung einer Dapp ... 68

5.1 Use Case „Parking Places“ ... 68
5.2 Tools und Frameworks ... 70
5.3 Solidity Contract ... 74
5.3.1 Modifier und Events ... 75
5.3.2 Funktionen ... 77
5.3.3 Deployment ... 78
5.4 JavaScript (D)App ... 80
5.4.1 Meteor Client und Pakete ... 81
5.4.2 Meteor Templates ... 83
5.4.3 Google Maps ... 86
5.4.4 Meteor Build Client ... 88

6 Test und Anwendung ... 90

6.1 Evaluation des Contract ... 90
6.2 Evaluation der Dapp ... 93
6.3 Einrichtung und Ausführung ... 98

7 Auswirkungen auf Smart Cities ... 102

7.1 Dapp „Parking Places” ... 102
7.2 Blockchain Technologie ... 107

8 Schlussbetrachtung ... 112

8.1 Zusammenfassung ... 112
8.2 Ausblick und Fazit ... 116

9 Verzeichnisse ... 118

9.1 Abbildungen ... 118
9.2 Tabellen ... 119
9.3 Listings ... 120
9.4 Formeln ... 120
9.5 Literatur ... 121

A. Anhang 129

A.1 Docker Container „geth-node” 129

A.2 Docker Container „meteor-nodejs” 129

B. Anhang 131

B.1 Solidity Contract „ParkingPlaces“ 131

B.2 NatSpec Benutzerdokumentation 135

B.3 NatSpec Entwicklerdokumentation 135

B.4 Contract ABI von „ParkingPlaces” 137

C. Anhang 139

C.1 „ParkingPlaces” View aus HTML und CSS 139

C.2 „ParkingPlaces” JavaScript 141

C.3 Testcases für Contract und Dapp 148

1 Einleitung

Im Kontext der schnell wachsenden globalen Urbanisierung, Innovation und Technologie wurde der Begriff Smart Cities und verwandte Konzepte bereits in den frühen 90er verwendet parallel zur Entstehung des Internets [NK13], [Sc11], [Ho15]. Herausforderungen des demografischen Wandel erfordern smarte Lösungen basierend auf dem Fortschritt der Informations- und Kommunikationstechnologie (IKT). Definitionen von Smart Cities und verwandten Konzepten stellen IKT in den Vordergrund und verdeutlichen so die Relevanz von Themen wie z.B. Big Data, Cloud Computing, Mobile Computing, Soziale Netzwerke oder Internet of Things (IoT). Laut einer Prognose von Gartner [Ga14] sind 2015 insgesamt 1,1 Billionen verbundene Geräte in Smart Cities weltweit installiert. Im Rahmen dieser Arbeit werden unterschiedliche Smart Cities Definitionen, verwandte Konzepte und ein Modell basierend auf europäischen Studien behandelt. Smart Cities sind eine Anwendungsdomäne für Technologie und Innovation wie der Blockchain.

Die Blockchain ist nach Swan [Sw15] die größte technologische Innovation von Bitcoin und das fünfte revolutionäre Computerparadigma nach Sozialen Netzwerken und Mobile Computing. Der IBM Business report „Device Democracy“ mit dem Untertitel „Saving the future of the Internet of Things“ [BP14] nutzt die Blockchain als technologisches Prinzip.

Unter dem Pseudonym Satoshi Nakamoto [Na08], [Na09] wurde im November 2008 der Artikel „Bitcoin: A Peer-to-Peer Electronic Cash System“ veröffentlicht, welches digitale Währung und Zahlungssysteme wesentlich verändert. Die Blockchain basiert auf kryptografischen Grundlagen und Sicherheit über ein dezentrales P2P-Netzwerk (Peer-to-Peer) ohne Vertrauen in dritte Parteien als „Single Point of Failure“. Mit der Blockchain wurden hierbei mit dem „Double Spending“ Problem nach Bonadonna [Bo13] und „Byzantine Generals“ Problem nach Lamport et al. [LSP82] zwei Lösungen für grundlegende Probleme aus 40 Jahren Forschung in der Kryptografie sowie aus 20 Jahren Forschung mit kryptografischen Währungen gelöst. Das Konzept der Blockchain geht über finanzielle Anwendungen im Kontext der Blockchain 1.0 hinaus. Bitcoins Design unterstützt durch seine integrierte Skriptsprache unterschiedliche Transaktionstypen und aktuelle wie zukünftige Anwendungsbereiche [Na10]. Die Blockchain 2.0 umfasst anwendungsspezifische MetaCoins und Protokolle wie Ethereum, Smart Properties, Smart Contracts aus dem Konzept von Nick Szabo [Sz97], dezentrale Anwendungen und autonome Agenten. Aufbauend hierauf umfasst die Blockchain 3.0 weiterführend alle Anwendungen und App-Coins (eigene Tokens oder Währungen) in nahezu allen Bereiche und Eigenschaften einer Smart City. Im Rahmen dieser Arbeit wird das Konzept von Bitcoin und der Blockchain betrachtet und ein Überblick der möglichen Anwendungsgebiete gegeben.

Das Blockchain 2.0 Protokoll Ethereum wurde als dezentrale Plattform für die Ausführung von Smart Contracts und dezentralen Anwendungen ausgewählt. Im Rahmen dieser Arbeit wird Ethereum mit eigener Blockchain und Protokoll mit Bitcoin verglichen. Darüber hinaus wird ein Proof-of-Concept (PoC) erstellt bestehend aus einer dezentralen Anwendung

(Dapp). Diese basiert auf einem Smart Contract und einer grafischen Benutzeroberfläche innerhalb der Blockchain, des Protokolls und P2P-Netzwerk von Ethereum. Dabei werden mehrere Prototypen implementiert und evaluiert. Der Use Case für die Anwendung wurde im Bereich Smart Mobility aus dem Kontext der Smart City des Projekt „The Smart Parking Network Barcelona“ abgeleitet. Der PoC soll zudem als Beispiel für die Entwicklung einer Dapp mit Ethereum bzw. mit der Blockchain dienen. Die Motivation für Ethereum wird im nächsten Abschnitt beschrieben. Daraufhin folgt eine Übersicht der aus dem Titel abgeleiteten Zielstellung und Aufbau dieser Arbeit.

1.1 Motivation für Ethereum

Ethereum ist eine dezentrale Open Source Plattform zur Ausführung von Smart Contracts mit eigener Blockchain und der kryptografischen Währung Ether (ETH). Diese ist hinter der Währung Bitcoin (BTC) nach dem Marktkapital vom 30.05.2016 von 1.012.664.130 USD und einem Tagesvolumen von 22.677.00 USD die zweitgrößte kryptografische Währung [Co16d]. Vision und Entwicklung von Ethereum gehen Ende 2013 zurück auf den Mitbegründer und Chefautor des Bitcoin Magazins Vitalik Buterin. Anfang 2014 veröffentlichte Buterin ein Whitepaper [Bu14b] und Ethereum Mitbegründer Dr. Gavin Wood ein Yellowpaper [Wo14]. Die Plattform wurde im Juni 2014 über das größte Crowdfunding in BTC und fünftgrößte Crowdfunding insgesamt mit 18.439.086 USD binnen 42 Tagen finanziert [Wi16a]. Zu diesem Zeitpunkt gab es das erste P2P-Testnetz mit Implementierung des Clients nach formaler Beschreibung des Yellowpaper in mehreren Programmiersprachen. Im Juli 2015 wurde die erste Version und im Verlauf dieser Arbeit im März 2016 die zweite Version live gestellt.

In Bitcoin wird alle zehn Minuten ein Block kryptografisch über den Hashalgorithmus SHA-256 erstellt. Ethereum hingegen verwendet die aktuellen SHA-3 Standards und erstellt alle 14 Sekunden einen neuen Block. Die integrierte Skriptsprache in Bitcoin ist wie auch das Speichern von Daten eingeschränkt. Ethereum verwendet eine Turing-vollständige Maschine, auf der höhere Programmiersprache wie Solidity für Smart Contracts aufbauen können. Durch Prinzipien des Ethereum Protokoll können alle Anwendungsbereiche abgebildet werden ohne Einschränkungen wie z.B. einer begrenzten Datenspeicherung. Ethereum verwendet mit Accounts ein simpleres Konzept als Bitcoin. Hierdurch sind u.a. eine feingranulare Kontrolle des Kontostands und mehrstufige Zustände möglich. Unterschiede wie Accounts, Zustände und Zustandsübergänge, Datenstrukturen sowie die Skriptsprache bzw. Turing Maschine zwischen Bitcoin und Ethereum werden im Rahmen dieser Arbeit herausgestellt.

Große IT-Unternehmen wie IBM und Microsoft verwenden seit 2015 die Ethereum Blockchain. Die Cloud Computing Plattform Microsoft Azure bietet die Blockchain als Dienst an (Blockchain as a Service) [Gr15]. Außerdem hat Microsoft Solidity in seine Visual Studio Entwicklungsumgebung integriert. IBM und Samsung verwenden die Ethereum Blockchain in ihrem Projekt ADEPT (Autonomous Decentralized Peer-to-Peer Telemetry) [Pa15].

Im Mai 2016 hat mit „The DAO“ das größte Crowdfunding [Wi16a] insgesamt einen Zeitraum von 28 Tagen mit ca. 160 Millionen USD stattgefunden. „The DAO“ ist eine dezentrale autonome Organisation, die nur aus Smart Contracts in Ethereum besteht. Insgesamt zeigt dies die breite Unterstützung und Akzeptanz der Plattform Ethereum und der Blockchain.

1.2 Ziel und Aufbau dieser Arbeit

Aus dem Titel *„Eine Blockchain / Ethereum Anwendung und ihre Auswirkungen auf Smart Cities“* lässt sich die Aufgabenstellung und Struktur dieser Arbeit ableiten.

- *auf Smart Cities* — Die Definition und Eigenschaften von Smart Cities sollen analysiert werden, damit später Auswirkungen der implementierten Prototypen und der Blockchain Technologie auf Smart Cities diskutiert werden können.
- *Blockchain* — Die Technologie hinter Bitcoin und dessen Anwendungsbereiche sollen von der kryptografischen Währung über Smart Contracts bis App-Coins detailliert analysiert werden. Hierbei werden u.a. kryptografische Grundlagen und die Historie von Bitcoin beschrieben.
- *Ethereum* — Die dezentrale Plattform Ethereum zur Ausführung von Smart Contracts basiert auf der Blockchain Technologie und soll im Vergleich zu Bitcoin analysiert werden. Anwendungen und Anwendungsbereiche sollen exemplarisch beschrieben werden.
- *Eine Anwendung und* — Es soll ein Use Case im Kontext von Smart Cities konzipiert werden. Für diesen soll innerhalb eines PoC für eine dezentrale Anwendung mit Ethereum Prototypen implementiert, bereitgestellt und evaluiert werden. Hierfür wurde Docker als containerbasierte Virtualisierung und das JavaScript Framework Meteor eingesetzt. Die Auswahl von Tools und Frameworks wird später erläutert.
- *ihre Auswirkungen* — Auf Basis der Eigenschaften von Smart Cities, der erstellten Prototypen, der Blockchain Technologie und der Ethereum Plattform sollen Auswirkungen auf Smart Cities diskutiert werden.

Die Reihenfolge dieser Punkte entspricht den folgenden Kapiteln bis einschließlich Kapitel 7. Die Anwendung wurde in Kapitel 5 mit der Implementierung und Kapitel 6 mit Test und Ausführung unterteilt. Zuletzt folgt mit Kapitel 8 eine Schlussbetrachtung als erweiterte Zusammenfassung sowie Ausblick über Entwicklungen dezentraler Anwendungen wie auch des erstellten Prototyps und der Ethereum Blockchain.

2 Einführung in Smart Cities

In diesem Kapitel werden Smart Cities und verwandte Konzepte vorgestellt sowie deren Hintergrund und Zusammenhang beschrieben. Anschließend werden Definitionen von Smart Cities chronologisch eingeordnet. Hierbei werden auch die Begriffe Smart und Smart Computing erklärt. Danach werden Eigenschaften und Dimensionen für eine Klassifizierung von Smart Cities beschrieben. Basis hierfür ist das Modell von Giffinger et al. [Gi07]. In der Studie „Mapping Smart Cities in the EU“ von Manville et al. [Ma14] wird dieses Modell erweitert und Auswirkungen von Smart City Projekten und Initiativen innerhalb der Europäischen Union analysiert. Später wird ein Use Case einer dezentralisierten Anwendung im Kontext von Smart Cities mit der Blockchain Technologie und Ethereum Plattform konzipiert, als Prototyp umgesetzt und diskutiert.

2.1 Motivation und Hintergrund

Der Begriff Smart Cities wurde bereits in den frühen 90er im Kontext von Urbanisierung, Globalisierung, Innovation und Technologie verwendet [NK13], [Sc11]. Aktuell leben ca. 54% von ca. 7,3 Billionen Menschen weltweit in urbanen Regionen. Die Kontinentalverteilung in Abb. 1 zeigt, dass außer Asien und Afrika knapp dreiviertel der Bevölkerung in Städten leben. Die UN prognostiziert für Asien und Afrika das größte Wachstum, so dass der Abstand zu anderen Kontinenten wie Europa oder Nordamerika sinkt. In 2050 leben ca. 66,4% von ca. 9,5 Billionen Menschen weltweit in urbanen Regionen. [De14]

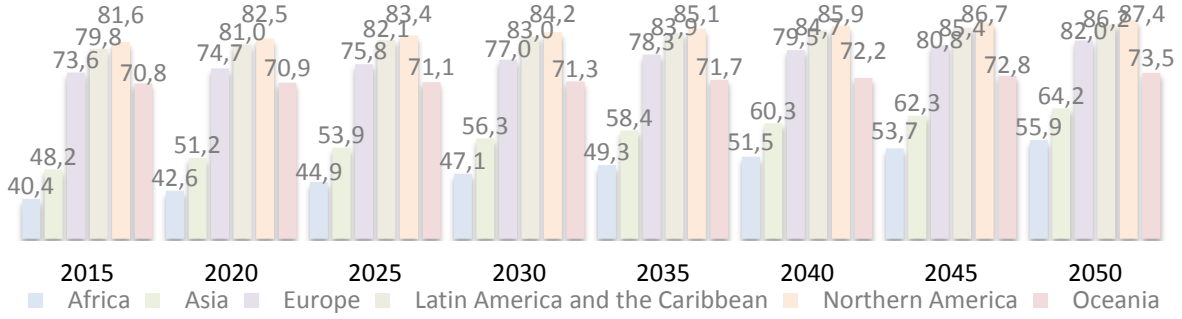

ABB. 1: KONTINENTALE URBANISIERUNG VON 2015 BIS 2050

Dieses Wachstum erfordert ein effizienteres Energie- und Ressourcenmanagement. Dabei sollen Länder und Städte sich an internationalen Leitzielen wie Europa 2020 oder den G7 Klimazielen orientieren. Europa 2020 definiert fünf Kernziele in den Bereichen, Forschung und Entwicklung, Beschäftigung, Klimawandel und nachhaltige Energiewirtschaft, Bildung und Bekämpfung von Armut und sozialer Ausgrenzung [Eu15]. In der Studie „Mapping smart cities in the EU“ von Manville et al. [Ma14] wird der Beitrag von Smart Cities auf diese Ziele analysiert. Washburn et al [Wa09] beschreiben weitere Herausforderungen:

- Ressourcenmangel
- Unzureichende und verschlechternde Infrastruktur

- Energieknappheit und Preisinstabilität
- Globale Umwelt und Gesundheitsbedenken
- Bessere wirtschaftliche Möglichkeiten und soziale Leistungen

Dieser demografische Wandel und seine Herausforderungen erfordern smarte Lösungen basierend auf dem Fortschritt von IKT. Howard [Ho15] beschreibt eine Zeitachse mit insgesamt 134 Ereignissen um das Thema vernetzte Geräte vom ersten tragbaren Computer (wearable) in 1961 bis zur Vorhersage von IBM, dass alle Fahrzeuge mit dem Internet bis 2025 verbunden sind. Der Begriff Internet of Things (IoT) wurde dabei erstmalig von Kevon Ashton 1999 im Rahmen seiner Arbeit mit Sensoren-Technologien wie RFID (Radio Frequency IDentification) verwendet. Einige Jahre später bekam IoT zunehmend breite Aufmerksamkeit. 2005 kam der erste ITU-Bericht (International Telecommunication Union) zu IoT heraus. 2007 wurde mit dem European Research Cluster on the Internet of Things (IERC) eine eigene Forschungsorganisation gegründet. 2008 wurde die erste jährliche Konferenz zu IoT in Europa für Forschung und Industrie abgehalten.

McKinsey [Mc15] zufolge haben IoT-Anwendungen in 2025 ein geschätztes wirtschaftliches Potential von 6,2 Trillionen USD. Damit ist IoT die drittgrößte disruptive Technologie hinter mobilem Internet und automatisierter Wissensverarbeitung. IoT wird von McKinsey definiert als Netzwerk mit preiswerten Sensoren und Aktoren zur Datensammlung, Monitoring, Entscheidungsfindung und Prozessoptimierung. Im IBM Businessreport „Device Democracy“ [BP14] wird IoT mit der Blockchain-Technologie als „Internet of Decentralized, Autonomous Things“ und als Demokratisierung der digitalen Welt beschrieben. Die Blockchain Technologie dient hierbei als Form der Überprüfung und Übereinstimmung von Billionen von Geräten, die nicht alle vertrauenswürdig sind. Hierdurch lassen sich eine Vielzahl an Beispielen für IoT-Anwendungen in Smart Cities konstruieren. IoT-Anwendungen sind z.B. in der Energie- und Wasserversorgung oder Verkehrssicherheit und -Überwachung möglich. Laut einer Statistik von Gartner [Ga14] sind 2015 weltweit 1,1 Billionen verbundene Geräte in Smart Cities installiert. In 2016 soll sich die Anzahl mehr als verdoppelt haben. Zudem sagt Gartner [Ga13] für 2020 weltweit 26 Billionen Geräte (ohne PCs, Tablets und Smartphones) voraus, die sich seit IPv6 adressieren lassen. Im nächsten Abschnitt werden mit Smart Cities verwandte Konzepte und Begriffe beschrieben und klassifiziert wobei Technologie (IoT, Blockchain usw.) eine eigene Dimension darstellt.

2.2 Verwandte Konzepte

Es gibt viele unterschiedliche Begriffe die neben und teilweise auch synonym für Smart Cities verwendet werden. Hollands [Ho08] beschreibt dieses Problem in dem Artikel „Will the real smart city please stand up?“ von 2008. Der Begriff Smart City wird in Forschung, Politik, Industrie und Wirtschaft unterschiedlich verwendet und es gibt keine einheitliche Definition.

Nam und Pardo [NP11] unterteilen mit Smart Cities Konzepte in drei Dimensionen Human, Technologie und Institution mit Verweis auf insgesamt 46 Studien in denen die jeweiligen Konzepte thematisiert werden. Tab. 1 gibt eine kurze Beschreibung der einzelnen Konzepte.

Konzept	Dimension	Beschreibung
	HUMAN	
Creative City		basiert auf humaner Infrastruktur wie kreativen Berufen und Arbeitskräften, Wissensnetzwerken, Vereinen, Freiwilligen-organisationen, verbrechensfreien Umgebungen und abendlicher Unterhaltungswirtschaft
Learning City		ist aktiv beteiligt an der Erstellung von qualifizierten Arbeits-kräften in der Informationswirtschaft
Knowledge City		ist analog zur Learning City und zielgerichtet konzipiert um Wissen zu fördern.
Human City		besitzt Möglichkeiten zur Ausschöpfung humaner Potentiale und kreativen Lebens.
	TECHNOLOGIE	
Digital City		ist eine breitbandig (IKT) angebundene Community mit innovativen Diensten angepasst auf die Bedürfnisse der Stadt, deren Mitarbeiter, Einwohner und Unternehmen.
Intelligent City		verwendet die aktuellste IKT für Infra- und Infostruktur innerhalb einer Wissensgesellschaft mit der Unterstützung zum Lernen, für technologische Entwicklung und innovative Verfahren.
Ubiquitous City		erweitert die Digital City um allgegenwärtige Verfügbarkeit und Erreichbarkeit der Infrastruktur und Dienste.
Hybrid City		besteht aus einer physischen Stadt mit einer virtuellen Abbildung im Cyberspace (Gebäuden, Menschen etc.).
Information City		bezieht sich auf digitale Umgebungen zur Sammlung von Informationen lokaler Communities, die der Öffentlichkeit über Webportale zur Verfügung stehen.
Wired City		ist eine verkabelte Stadt siehe Hollands [Ho08].
	INSTITUTION	
Smart Community		besteht aus Unternehmen, Regierungen und Bevölkerung mit dem Verständnis für die Potentiale von IKT und dessen Nutzen zur Verbesserung von Arbeit und Leben.

TAB. 1: ÜBERSICHT VON CITY KONZEPTEN GRUPPIERT NACH DIMENSIONEN

Für eine detaillierte Beschreibung und Vergleich der jeweiligen Konzepte siehe [NP11]. In der Literatur werden darüber hinaus weitere Begriffe und Konzepte erwähnt wie Cyber City, Mobile City, Sustainable City, Connected City, Entrepreneurial City, Pioneer City, Livable City, Talented City oder Eco-City. Diese lassen sich ebenfalls einer Dimension zuweisen, werden

hier aber nicht weiter betrachtet. Alle Konzepte stehen in Beziehung zueinander und die Smart City geht als Schnittmenge der Dimensionen hervor wie in Abb. 2 dargestellt.

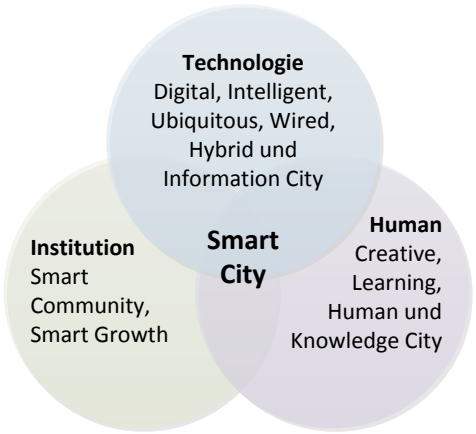

ABB. 2: CITY KONZEPTE UND DIMENSIONEN [NP11]

Diese Dimensionen einer ganzheitlichen Betrachtung von Konzepten um die Smart City wird häufig verwendet und zitiert z.B. von Manville et al. [Ma14]. Im folgenden Kapitel werden verschiedene Smart City Definitionen beschrieben, die sich auf diese Dimensionen beziehen.

2.3 Smart City Definitionen

Dieses Kapitel beschreibt die Begriffe Smart und Smart Computing. Darüber hinaus wird das breite Spektrum einer Smart City gezeigt über seine Definitionen von 2000 - 2014 mit unterschiedlichen Schwerpunkten.

Laut Oxford Dictionary [Ox15] ist etwas smart, wenn es sauber, ordentlich, modisch, intelligent, schnell und computergesteuert ist. Nam und Pardo [NP11] beschreiben „smart“ aus unterschiedlichen Perspektiven. Aus Markt- und Markensicht wird der Begriff für Marketingstrategien verwendet. In der Städteplanung bezieht sich „smart“ als ideologische und strategische Ausrichtung. In der IKT werden mit „smart“ intelligente Produkte und Dienste, Künstliche Intelligenz und selbst denkende Maschinen impliziert. In der Literatur der Begriff „intelligent“ häufig synonym verwendet und die Bedeutung von IKT in vielen Definitionen von Smart Cities hervorgehoben. Die Definition von Washburn et al. [Wa09] für Smart Cities basiert auf Smart Computing welches sie zunächst wie folgt definieren:

> *„Eine neue Generation integrierter Hardware-, Software- und Netzwerktechnologien die IKT Systeme im Bewusstsein von Echtzeit der realen Welt und moderner Analytik anbieten, um die Menschen bei intelligenten Entscheidungen über Alternativen und Handlungen, die Ergebnisse von Geschäftsprozessen und Bilanzen optimieren, zu unterstützen.“*

Die anschließende Definition für Smart Cities von Washburn et al. [Wa09] aufgeführt in Tab. 2 hat somit ihren Schwerpunkt in der Dimension Technologie. Nach Walravens und Ballon

[WB13] gibt es unterschiedliche Ansätze Smart Cities zu definieren. Einige haben eine ganzheitliche Perspektive über IKT, Wettbewerb, Sozial- und Humankapital, Natürlichen Ressourcen bis zur Lebensqualität und Partizipation. Andere basieren auf Nachhaltigkeit, Innovation und Wettbewerbsfähigkeit. Sie orientieren sich an Leitzielen z.B. Europa 2020. Unterschieden wird auch nach dem „Top-Down"-Prinzip (Initiativen der Stadt) und dem „Bottom-Up"-Prinzip (Bürgerinitiativen). Manville et al. [Ma14] unterteilen Smart Cities Definitionen in solche die ihren Fokus auf IKT legen und solche die Smart Cities im weitesten Sinne betrachten. Dabei weisen sie darauf hin, dass eine Smart City viel mehr von der Smartness seiner Bürger und Communities sowie deren Wohlbefinden und Lebensqualität bestimmt wird. Ein Bericht der ISO [IS14] zeigt, dass sich mehrere Institutionen mit der Standardisierung und Normung von Smart Cities beschäftigen. Tab. 2 gibt einen Überblick einiger Smart City Definitionen unter Angabe von Quelle und Jahr.

Quelle	Definition
Hall et al. [Ha00]	*„Eine Stadt die Zustände aller kritischen Infrastrukturen wie Straßen, Brücken, Tunneln, Schienen, U-Bahnen, Flughäfen, Seehäfen, Kommunikation, Wasser, Strom, größere Gebäude überwacht und integriert, kann besser ihre Ressourcen optimieren, Aktivitäten zur vorbeugende Instandhaltung planen und überwachen sowie gleichzeitig Dienste für die Bürger maximieren."*
Komninos [Ko02]	*„Ein Gebiet mit hoher Kapazität für Lernen und Innovation, dass auf der Kreativität seiner Gemeinde, seinen Institutionen zur Generierung von Wissen und seiner digitalen Infrastruktur für Kommunikation und Wissensmanagement basiert."*
Partridge [Pa04]	*„Eine Stadt wo IKT die Meinungsfreiheit und Verfügbarkeit öffentlicher Informationen und Dienste stärken."*
Giffinger et al. [Gi07]	*„Eine gut funktionierende und zukunftsweisende Stadt für Wirtschaft, Menschen, Regierung, Mobilität, Umwelt und Wohnen, errichtet in smarter Kombination von Talent und Tätigkeiten von selbst entscheidenden, unabhängigen und bewussten Bürgern."*
Caragliu, Del Bo und Nijkamp [CDN09]	*„Eine Stadt ist smart wenn Investitionen in Human- und Sozialkapital, traditionelle und moderne Kommunikationsinfrastruktur ein nachhaltiges Wirtschaftswachstum und hohe Lebensqualität, vernünftiges Management natürlicher Ressourcen über eine teilnehmende Regierung ermöglichen."*
Washburn et al. [Wa09]	*„Die Anwendung von Smart Computing Technologien, um die kritischen Infrastruktur Komponenten und Dienste einer Stadt - welche Stadtverwaltung, Bildung, Gesundheitswesen, öffentliche Sicherheit, Liegenschaften, Verkehrswesen und Versorgung beinhalten - intelligenter, vernetzter und effizienter zu machen."*
Toppeta [To10]	*„Eine Stadt die IKT und Web 2.0 mit Organisation, Design und Planung kombiniert, um bürokratische Prozesse durch Entmaterialisierung zu beschleunigen und dabei zu helfen, neue und innovative Lösungen für die Komplexität des Stadtmanagement zu identifizieren, um Nachhaltigkeit und Wohnlichkeit zu verbessern."*

Harrison et al. [Ha10]	*„Eine Stadt verbindet die physikalische-, die IKT-, die soziale- und die Business-Infrastruktur, um die kollektive Intelligenz der Stadt wirksam einzusetzen.“*
Nam und Pardo [NP11]	*„In Smart Cities geht es um die wirksame Zusammenarbeit innerhalb und zwischen grundlegenden Bereiche der Stadt (z.B. Transport, öffentliche Sicherheit, Energie, Bildung, Gesundheitswesen und Entwicklung). Smart City Strategien erfordern innovative Interaktionsformen mit Stakeholdern, die Ressourcenverwaltung und das Angebot von Diensten.“*
Rios [RI12]	*„Eine Stadt, die Inspiration gib sowie Kultur, Wissen und das Leben miteinander teilt. Eine Stadt, die ihre Bewohner motiviert, das eigene Leben zu kreieren und aufblühen zu lassen.“*
Batty et al. [Ba12]	*„Eine Stadt in der IKT mit traditioneller Infrastruktur verschmilzt sowie neue digitale Technologien koordiniert und integriert werden.“*
Haque [Ha12]	*„Jedes angemessene Modell einer Smart City muss sich ebenfalls auf die Smartness ihrer Bürger und Gemeinden sowie auf Wohlbefinden und Lebensqualität konzentrieren, ebenso wie auf die in einer Stadt für die Menschen wichtigen Prozesse und Aktivitäten, die sehr unterschiedlich und teilweise auch widersprüchlich sein können.“*
Smart City Groups [Sm13a]	*„Städte sollten als Systeme von Systemen gesehen werden, so dass sich Möglichkeiten für ein digitales Nervensystem, intelligente Reaktionsfähigkeit und Optimierung auf allen Ebenen der Systemintegration eröffnen.“*
Smart Cities and Communities [Sm13b]	*„Smart Cities kombinieren diverse Technologien, um ihre Auswirkungen auf die Umwelt zu reduzieren und den Bürgern ein besseres Leben zu bieten. Dieses bedarf neben technischer Herausforderung auch organisatorische Veränderung in Regierungen und der gesamten Gesellschaft. Die Erstellung einer Smart City als multidisziplinäre Herausforderung erfordert die Zusammenarbeit von Beamten, innovativen Anbietern, Politikern, Akademikern und der Gesellschaft.“*
Manville et al. [Ma14]	*„Eine Stadt adressiert öffentliche Themen über IKT-basierte Lösungen auf Basis mehrerer Stakeholder in einer kommunalen Partnerschaft.“*

TAB. 2: SMART CITY DEFINITIONEN VON 2000 BIS 2014

Die betrachteten Definitionen für Smart Cities gehen bis auf das Jahr 2000 zurück und haben unterschiedliche Schwerpunkte auf wirtschaftliche, soziale oder technologische Faktoren.

Manville et al. [Ma14] sehen als Kern von Smart Cities die Bildung und Verbindung von Humankapital, Sozialkapital und IKT Infrastruktur für eine größere, nachhaltigere und wirtschaftliche Entwicklung mit besserer Lebensqualität. Dabei beziehen sie sich auf die sechs Eigenschaften (Wirtschaft, Menschen, Regierung, Mobilität, Umwelt und Wohnen) der Definition von Giffinger et al. [Gi07]. Im nächsten Kapitel wird das Modell einer Smart City von Giffinger und hierauf aufbauend das Modell von Manville beschrieben, welches die Dimensionen von Nam und Pardo und somit auch verwandte Konzepte von Smart Cities berücksichtigt.

2.4 Modell nach Giffinger und Manville

In 2007 erstellten Giffinger et al. [Gi07] in ihrer Studie „Smart cities - Ranking of European medium-sized cities“ ein Modell für Smart Cities mit sechs Eigenschaften über die sie auch Smart Cities definieren. Jede Eigenschaft besitzt eine strategische Ausrichtung und Faktoren wie Abb. 3 zeigt. Alle Faktoren basieren auf Indikatoren verschiedener Datenquellen wie Eurostat [Eu15].

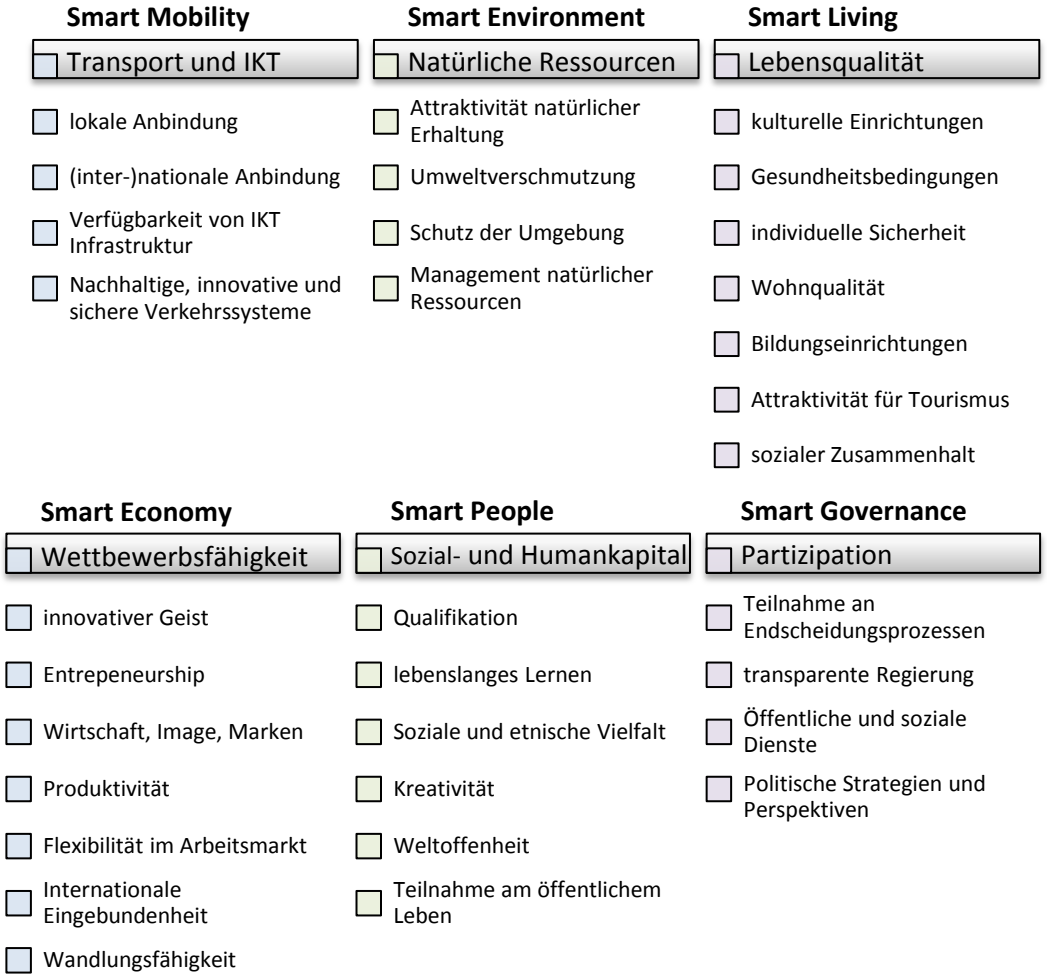

ABB. 3: EIGENSCHAFTEN UND FAKTOREN EINER SMART CITY [GI07]

Dieses Modell wird in vielen Studien und Arbeiten ([CDN09], [GG10], [SBM12], [Ba12]) wie auch von Manville et al. [Ma14] verwendet. Im Rahmen des europäischen Smart City Project der TU Wien wird das Modell eingesetzt für das Ranking und den Vergleich europäischer Smart Cities. In 2013 wurden mit Version 2.0, in 2014 mit Version 3.0 die Faktoren und die Indikatoren bzw. die Datenbasis aktualisiert. Daneben gibt es seit 2015 eine Version 4.0 für größere Städte (300.000 - 1.000.000 Einwohner) während vorherige Versionen für mittlere Städte (100.000 - 500.000 Einwohner) gelten. [Gi15]

Nach Manville et al. [Ma14] muss eine Smart City Initiative oder ein Projekt mindestens eine dieser Eigenschaften besitzen. Eine Smart City besteht aus einer oder mehreren Initiativen, die aus einem oder mehreren Projekten besteht. In der Studie „Mapping Smart Cities in the EU“ wurden von 468 Städten mit mehr als 100.000 Einwohnern aus den 28 Mitgliedsstaaten

240 Smart Cities mit Initiativen und Projekten untersucht. Dabei erweitern Manville das Modell von Giffinger um die Dimensionen von Nam und Pardo siehe Abb. 4.

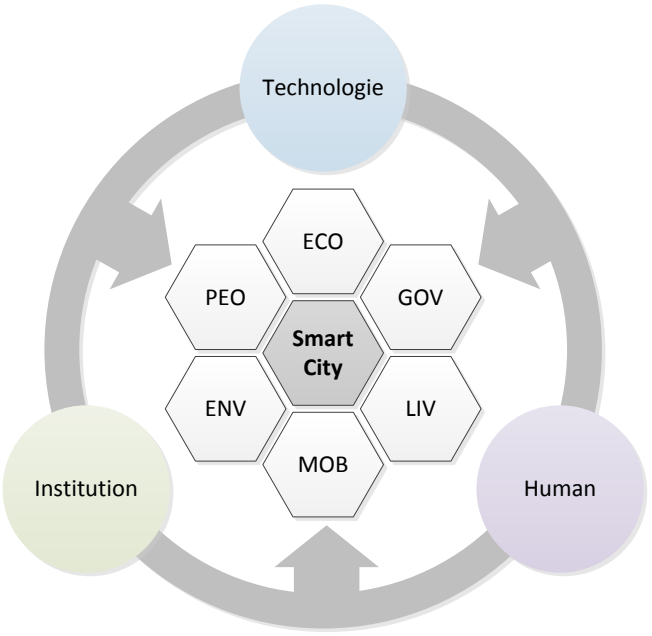

ABB. 4: SMART CITY MODELL [MA14]

Manville et al. [Ma14] bezeichnen diese Dimensionen als „building blocks“ oder auch als Komponenten einer Smart City Initiative, die neben der Entwicklung und Anwendung von Technologien auch humane und soziale Aspekte um die Rolle der Stakeholder beinhalten. Eine Komponente kann sich auf eine oder mehrere Eigenschaften beziehen. Nam und Pardo [NP11] beschreiben für jede Dimension eine strategische Ausrichtung (untere Zeile in Tab. 3). Zum Beispiel hat eine Initiative mit dem Ziel zur Reduzierung der Umweltbelastung durch industrielle Heizöfen die Eigenschaft Smart Environment und besitzt als Komponenten verschiedene Umwelttechnologien sowie Komponenten für deren Wissen und Steuerung.

Technologie	**Human**	**Institution**
Physikalische Infrastruktur	Human Infrastruktur	Steuerung
Smarte Technologien	Sozialkapital	Verfahrensweisen
Mobile Technologien		Vorschriften und Richtlinien
Virtuelle Technologien		
Digitale Netzwerke		
INTEGRATION	LERNEN UND WISSEN	STEUERUNG

TAB. 3: DIMENSIONEN UND IHRE STRATEGISCHE AUSRICHTUNG [NP11]

Im nächsten Kapitel wird beschrieben wie Smart Cities, Initiativen und Projekte basierend auf den Eigenschaften und Dimensionen aus Abb. 3 und Abb. 4 sowie Tab. 3 hinsichtlich ihres Erfolgs bewertet werden.

2.5 Erfolgreiche Beispiele in Europa

In der Analyse europäischer Städte untersuchten Manville Städte in Europa, die mindestens eine der im vorherigen Kapitel beschrieben Eigenschaften erfüllen. Dieses Kapitel beschreibt Kriterien nach denen erfolgreiche Smart Cities bemessen werden.

Manville et al. [Ma14] haben bei der Betrachtung nur Städte betrachtet die mindestens den ersten Reifegrad erreicht haben. Dieser ist in vier Stufen gegliedert:

1. Strategie oder Richtlinien für eine Smart City
2. Projektplan oder Vision ohne Implementierung oder Pilot
3. Pilot zum Testen von Initiativen
4. Initiative vollständig eingeführt

Dabei ist festzustellen, dass die Anzahl von Smart Cities (Verhältnis zu allen 468 Städten), dessen Reifegrad und Eigenschaften jeweils mit Größe (Anzahl Einwohner) und geografischer Verteilung korrelieren. Größere Städte sind häufiger Smart Cities mit höherem Reifegrad und gleichmäßiger Verteilung aller Eigenschaften.

Für die Bewertung von Smart Cities haben Manville et al. [Ma14] erfolgreiche Smart Cities und Initiativen definiert:

> *Eine **Smart City Initiative ist erfolgreich wenn diese** breite Unterstützung findet, eindeutige Vorgaben angelehnt an politischen Zielen und aktuellen Problemen hat, konkrete Ergebnisse und Auswirkungen hervorbringt sowie reproduzierbar und skalierbar ist. Der Erfolg muss dabei durch feststellbare Kennzeichen messbar sein.*
>
> *Eine **Smart City ist erfolgreich wenn diese** bedeutsame Vorgaben (angelehnt an Europa 2020 und aktuellen Auswirkungen), bestehend aus einer Mischung von politischen Zielen und Merkmalen, und ein ausgeglichenes Portfolio an Initiativen besitzt, Reife erlangt sowie aktiv an Netzwerken teilnimmt.*

Von insgesamt 240 Smart Cities haben Manville et al. [Ma14] 37 mit insgesamt 50 Initiativen ausgewählt. Die Auswahl erfolgte aufgrund eines Reifegrads von mindestens Stufe 2, der Summe an verfügbaren Informationen, einer repräsentativen geografischen Verteilung und Population sowie dem Projekterfolg (basierend auf den zuvor beschriebenen Definitionen). Dabei wurden Projekte hinsichtlich ihrer Ziele, Stakeholder, Steuerung, Finanzierung, Erfolg und Nutzen sowie der Möglichkeiten für Skalierung und Verbreitung miteinander verglichen und fünf Projekttypen ermittelt:

1.	Nachbarschaftseinheiten	*z.B. Hafencity in Hamburg*
2.	Testumgebung für Mikro-Infrastrukturen	*z.B. Klima-Straße in Köln*
3.	Intelligente Verkehrssysteme	*z.B. Vehicle Inductive Profile in Enschede*
4.	Managementsysteme für Ressourcen	*z.B. E-Energie in Mannheim*
5.	Plattformen für Partizipation	*z.B. AmsterdamOpent.nl*

In einer quantitativen Analyse haben Manville et al. [Ma14] eine Auswahl von 20 Smart Cities und deren Eigenschaften hinsichtlich ihrer Ausrichtung auf die Europa 2020 Ziele in den Bereichen Beschäftigung, Forschung, Entwicklung, Innovation, Klima, Energie, Bildung, Armut und soziale Ausgrenzung bewertet. Eine Initiative mit der Eigenschaft Smart Mobility kann z.B. Beschäftigung und Bildung fördern, in dem die Verkehrssituation zur jeweiligen Bildungseinrichtung oder zum Arbeitsplatz verbessert wird. Das reduziert auch das Risiko von Armut und sozialer Ausgrenzung. Die Beziehung einer Eigenschaft und einem der Ziele aus Europa 2020 ist dabei gewichtet über die Stärke ihrer Verbindung. Diese kann direkt, indirekt (z.B. durch Wissenstransfer) oder kollektiv (z.B. in der Erstellung einer Interessengemeinschaft) bestehen. Für die Erfolgsmessung wurden folgende drei Scores eingeführt:

- Eigenschaften-Score gibt an wie weit das Portfolio an Initiativen vom Idealwert (Erfüllung aller Eigenschaften einer Smart City) entfernt ist.
- Reichweiten-Score gibt an ob jede Eigenschaft in mindestens einer Initiative vorhanden ist.
- Performanz-Score ist analog zum Eigenschaften-Score nur mit der Gewichtung einzelner Eigenschaften hinsichtlich ihrer Relevanz bezogen auf die Ziele von Europa 2020.

Zuletzt wurden von Manville et al. [Ma14] Fallstudien mit den erfolgreichsten Smart Cities (Manchester, Barcelona, Kopenhagen, Amsterdam, Helsinki und Wien) des Reichweiten-Scores durchgeführt. Dabei wurden 12 Lösungen identifiziert und drei Erfolgsfaktoren Vision, Personen und Prozess definiert. Eine Vision beschreibt die Bedürfnisse und Wünsche eine Stadt in einen besseren Lebensraum umzuwandeln. Eine Stadt besteht neben Komponenten aus Personen, die sie gestalten und formen. Diese Transformation ist dabei der Prozess, dessen Erfolg durch effektives Projekt- und Wissensmanagement sowie Evaluation bestimmt wird. Auswirkungen der betrachteten Lösungen verteilen sich auf den unterschiedlichen Nutzlevel siehe Abb. 5.

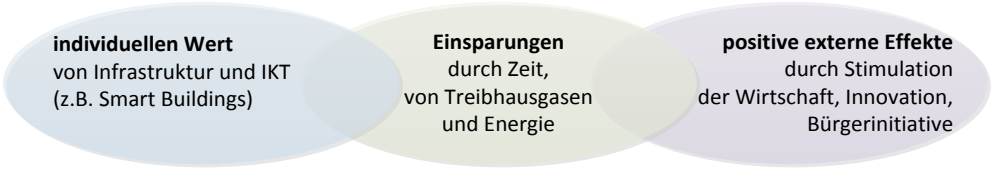

ABB. 5: NUTZLEVEL VON SMART CITY LÖSUNGEN [MA14]

Als Ergebnis der Fallstudien wurden von Manville et al. [Ma14] acht Smart City Lösungen in den Bereichen Transport und Mobilität, Energie- und Gebäudetechnik sowie Smart Governance mit Potential bezogen auf Europa 2020 Ziele identifiziert. Diese Lösungen können in den meisten Städten angewandt werden, da sie sind über ihre Kosteneffektivität gut skalierbar sind. Dies ist erforderlich um den Grad an Auswirkungen zu erzielen der z.B. für Europa 2020 Ziele erforderlich ist. In Kopenhagen, Paris und London werden z.B. im Bereich Transport und Verkehr mit smarter Fahrradplanung die CO_2 Emission reduziert und positive Auswirkungen auf die Gesundheit erzielt. In Barcelona und Milano gibt es im Bereich der Energie- und Gebäudetechnik eine Smart City Beleuchtung zur Reduzierung des

Energieverbrauchs und der CO_2 Emission sowie mit positiven Auswirkungen auf die öffentliche Sicherheit. Im Bereich Smart Governance haben Barcelona und Manchester einzelne Zugangspunkte für Dienstleistungen eingerichtet, so dass der Verkehr zur Stadtverwaltung und die CO_2 Emission reduziert wird. Weiter konnten Manville et al. [Ma14] im Rahmen ihrer Fallstudien die in Tab. 4 aufgeführten „Best-Practices“ basierend auf den Erfolgsfaktoren Vision, Personen und Prozess für die Entwicklung und Umsetzung von Smart City Lösungen erkennen.

Erfolgsfaktor	„Best Practises“
1. Vision	○ „Quick-Wins“ d.h. Ziele niedrig ansetzen und dafür schnelle Ergebnisse erzielen ○ Inklusion und Beteiligung aller Einwohner unabhängig ihres Alters
2. Personen	○ Nutzerzentriert nach den „Bottom-Up“-Prinzip (z.B. Crowdfunding) ○ Inspiration durch „City Champions“ ○ Kollaboration und Kooperation in Erstellung und Entwicklung ○ Zentrales und eigenes Smart City Büro welches die Vision kommuniziert
3. Prozess	○ Daten und Informationen veröffentlichen ○ Lokale Koordination und Orientierung ○ Über „Living Labs“ und Netzwerke lernen und Wissen verteilen

TAB. 4: ERFOLGSFAKTOREN UND "BEST PRACTISES" [MA14]

Diese „Best Practises“ sind zum Teil Gegenstand weiterer Arbeiten. Zum Beispiel werden Smart Cities in dem Artikel „Smart Cities and the Future Internet“ von Schaffers et al. [Sc11] als Umgebungen für offene, nutzerzentrierte Innovationen zum Experimentieren und Validieren („Living Labs“) von internetgestützten und zukunftsfähigen Diensten untersucht. Der nutzerzentrierte Ansatz wird im Bereich Crowdsourcing und „Living Labs“ auch in der Arbeit „Smart Ideas for Smart Cities“ von Schuurmann et al. [SBM12] behandelt.

Eine Vision kann ebenso zum Image einer Stadt, eines Projektes oder einer Initiative werden. Daher können auch die von Nijkamp und Kourtit [NK13] beschriebenen Images (Connected-, Entrepreneurial-, Pioneer- und Liveable City 2050) herangezogen werden. Innovation und Wohlstand beschreiben die Entrepreneurial City. Smart Logistik und nachhaltige Mobilität beschreiben die Connected City. Ökologische Nachhaltigkeit beschreibt die Liveable City und soziale Teilnahme wie auch Kapital beschreiben die Pioneer City.

Mit Smart Cities wurde in diesem und vorangegangenen Kapiteln eine Anwendungsdomäne für Technologie und Innovation vorgestellt. Dabei wurde der Hintergrund, verwandte Konzepte, Definitionen, ein Modell und erfolgreiche Beispielen in Europa beschrieben. Im nächsten Kapitel wird mit der Blockchain eine Technologie vorgestellt, die ein neues Computing Paradigma einführt. Hierbei werden Historie und Hintergrund sowie Grundlagen und Funktionsweise beschrieben. Weiter werden Anwendungen und Anwendungsbereiche beschrieben, die mit Wirtschaft, Menschen, Regierung, Mobilität, Umwelt und Wohnen alle Smart City Eigenschaften betreffen.

3 Einführung in die Blockchain

Dieses Kapitel beschreibt die Blockchain Technologie als größte technologische Innovation von Bitcoin und gleichzeitig revolutionäre Entwicklung vergleichbar mit Paradigmen wie dem Internet, sozialen Netzwerken und Mobile Computing. An dieser Stelle werden zunächst Motivation und Hintergrund von Bitcoin betrachtet. Danach wird mit den kryptografischen Grundlagen die Basis beschrieben. Im Anschluss folgt eine Beschreibung der Struktur der Blockchain, des P2P-Netzwerkes und seinen Transaktionen. Nach dieser technischen Betrachtung werden unterschiedliche Anwendungsbereiche und Beispiele gezeigt. Auf der Blockchain Technologie basieren Alternative (Alt)-, App- und Meta-Coins sowie dezentrale Plattformen wie Ethereum, welche in Kapitel 4 analysiert und in Kapitel 5 für die Implementierung einer Dapp innerhalb eines PoC verwendet wird.

3.1 Motivation und Hintergrund

Bitcoin wurde unter dem Pseudonym Satoshi Nakamoto in seinem Artikel „Bitcoin: A Peer-to-Peer Electronic Cash System“ im November 2008 und in der ersten Version v0.1 im Januar 2009 veröffentlicht. Bitcoin besteht aus der digitalen Währung, einem P2P-Protokoll und einem Client jeweils als Open Source Software. Die Blockchain dient hierbei als verteilte, öffentliche und dezentrale Datenbank. Bitcoin wurde entwickelt um Schwächen von elektronischen Zahlungssystemen im E-Commerce zu beseitigen. Diese sind das Vertrauen in dritte Parteien als „Single Point of Failure“, hohe Transaktionskosten und limitierte Transaktionsgrößen sowie reversible Transaktionen für eine potentielle Betrugs- und Konfliktbehandlung wie z.B. in PayPal. [Na08], [Na09]

Bitcoin basiert auf 40 Jahren Forschung in der Kryptografie sowie auf 20 Jahren Forschung mit kryptografischen Währungen und erreichte hier nach Swan [Sw15] durch die Blockchain einen fundamentalen Durchbruch mit der Lösung zwei grundlegender Probleme:

1. „Double Spending“ Problem siehe Bonadonna [Bo13]
2. „Byzantine Generals“ Problem siehe Lamport et al. [LSP82]

„Double Spending“ beschreibt den Transfer einer Währungseinheit an mehr als ein Ziel. Mit einer dritten Partei als zentrale Autorität wie z.B. in PayPal werden Transaktionen auf „Double Spending“ geprüft und verhindert bzw. in einer Konfliktbehandlung aufgelöst. Nur wenn Währungseinheiten von einer Quelle stammen, die alle Transaktionen verwaltet, kann darauf vertraut werden, dass sie einmalig verwendet wurden. In Bitcoin sind alle getätigten Transaktionen in Blöcken innerhalb der Blockchain enthalten. Die Blockchain ist eine chronologische Verkettung von kryptografisch verifizierten Blöcken mit allen Transaktionen über ein verteiltes, dezentrales und öffentliches P2P-Netzwerk. Bitcoin besitzt keine zentrale Autorität und basiert daher nicht auf dessen Vertrauen sondern auf der kryptografischen Verifikation und Berechnung innerhalb der Blockchain. [Na08]

Das „Byzantine Generals“ Problem beschreibt das Problem wenn sich mehrere örtlich voneinander getrennte Generäle über die Strategie einer Schlacht einigen müssen. Die Schwierigkeit hierbei sind fehlerhafte oder manipulierte Nachrichten und Verräter oder illoyale Generäle. Zuverlässige Computersysteme müssen widersprüchliche Informationen fehlerhafter oder kompromittierter Komponenten behandeln können. [LSP82] Dieses Problem besteht auch wenn ein System mit mehreren Clients über einen unsicheren und unzuverlässigen Übertragungsweg wie dem Internet kommuniziert. Bitcoin nutzt hierfür die kryptografische Berechnung und Verifikation in der Blockchain als Konsens über die Mehrheit der beteiligten Miner. Als Mining wird der Prozess bezeichnet, der neue Blöcke durch kryptografische Hashberechnung der Blockchain hinzufügt. Der Miner erhält für jede Transaktion in dem hinzugefügten Block die Transaktionsgebühren und pro Block 25 BTC (Stand November 2014, in 2009 gestartet mit 50 BTC). Das System ist so konzipiert, dass ein Block ca. alle zehn Minuten hinzugefügt wird und somit maximal 21 Millionen BTC im Mai 2140 erzeugt sein werden. [An15], [Bi16h]

Bitcoin versteckt die Identität der Benutzer analog zu einer E-Mail-Adresse hinter einer öffentlichen Adresse als Pseudonym. Diese befindet sich mit den zugehörigen privaten Schlüsseln in sogenannten Wallets (Teil des Bitcoin Client). Über private Schlüssel wird der Besitz von BTC in Transaktionseingängen nachgewiesen, um diese für neue Transaktionen zu verwenden. Alle Transaktionen sind in der Blockchain enthalten und damit transparent. Die Validierung von Transaktionen basiert auf einer stapelverarbeitenden, nicht Turing-vollständigen Skriptsprache, mit der es möglich ist BTC zu sperren und zu entsperren. Bitcoin kann mit der Blockchain für jede Form digitaler Assets genutzt werden. [Fr15] Mit der Skriptsprache und unterschiedlichen Transaktionstypen sind Smart Properties, Smart Contracts und autonome Agenten möglich.

Swan [Sw15] beschreibt Bitcoin mit der Blockchain (2010 - 2020) in einer verbundenen Welt (Wearables, IoT-Sensoren, Smartphones, Tablets, Laptops, Smart Homes, Smart Cars, Smart Cities) als fünftes revolutionäre Computerparadigma nach Sozialen Netzwerken und Mobile Computing (2000 - 2010). Dabei beschreibt sie die Blockchain als Protokoll der Anwendungsschicht innerhalb des ISO/OSI-Referenzmodells für Transaktionen digitaler Währung und Assets sowie komplexen Verträgen. Transaktionen können von Mensch zu Mensch, Menschen zu Maschine sowie Maschine zu Maschine (M2M) erfolgen, wobei Bitcoin keine maximale Transaktionsgröße besitzt und somit Mikrozahlungen unterstützt. Diese sind insbesondere für IoT- und M2M-Kommunikation erforderlich.

Bitcoin ist die erste und basierend auf dem Marktkapital größte kryptografische Währung. Daneben gibt es mit Alt-Coins weitere wie z.B. Litecoin oder Peercoin, die auf dem Open Source Projekt Bitcoin basieren. Im Gegensatz dazu speichern Meta-Coins wie Ripple, Litecoin oder Ethereum anwendungsspezifische Metadaten in einer eigenen oder in der Blockchain von Bitcoin. [Fr15] Im nächsten Kapitel werden die kryptografischen Grundlagen beschrieben auf die Bitcoin, die Blockchain sowie Alt- und Meta-Coins aufbauen.

3.2 Kryptografische Grundlagen

Die Währung Bitcoin wird auch als digitale und kryptografische Währung bezeichnet. Die Funktionsweise von Bitcoin basiert dabei im Kern auf kryptografischen Algorithmen wie symmetrischen und asymmetrischen Verschlüsselungsverfahren, digitalen Signaturen und Hashfunktionen. Dieses Kapitel beschreibt die von Bitcoin verwendeten kryptografischen Verfahren.

3.2.1 Symmetrische Verschlüsselung

Bitcoin verwendet die symmetrische Verschlüsselung um private Schlüssel, mit denen Transaktionen signiert werden in Wallets zu schützen. Antonopoulos [An15] und Franco [Fr15] vergleichen den privaten Schlüssel mit der geheimen PIN (Personal Identification Number) einer EC-Karte und den öffentlichen Schlüssel mit der zugehörigen Kontonummer.

Schon Julius Caesar (100 v. Chr.) benutzte die symmetrische Verschlüsselung im gallischen Krieg. Bei der symmetrischen Verschlüsselung verwenden mehrere Parteien einen Schlüssel als gemeinsames Geheimnis zum Ver- und Entschlüsseln von Nachrichten. Der Schlüssel muss zuvor über einen sicheren Übertragungskanal geteilt worden sein. Symmetrische Verschlüsselung besitzt keine Nachweisbarkeit. Einer dritten Partei kann nicht nachgewiesen werden, von wem eine Nachricht verschlüsselt wurde. Die symmetrische Verschlüsselung unterscheidet zwischen Kanal- und Blockverschlüsselung. Bei der Kanalverschlüsselung wird jedes `Bit` eines Datenstroms einzeln verschlüsselt, während bei der Blockverschlüsselung eine Nachricht fester Länger in einzelnen Blöcken verschlüsselt wird. [PP10]

AES (Advanced Encryption Standard) ist eine Blockverschlüsselung mit einer Schlüssellänge von `128 Bit` bis `256 Bit`. AES wurde vom National Institute of Standards and Technology (NIST [NI01]) spezifiziert und ist in den USA als erster öffentlicher Verschlüsselungsalgorithmus mit `192 Bit` und `256 Bit` Schlüssellänge für „TOP SECRET“ Dokumente zertifiziert. Die Blocklänge ist in AES mit `128 Bit` festgelegt und wird durch eine zweidimensionales Arrays mit (`4 * 4 * 8 Bits = 128 Bits`) repräsentiert. In AES-256 ist ein Algorithmus mit 14 Durchläufen bestehend aus mathematischen Transformationen auf dem Array definiert. AES-256 hat eine Schlüssellänge von `256 Bit`.

Bitcoin verwendet als Nachricht den privaten Schlüssel der asymmetrischen „Public Key“ Verschlüsselung. Da dieser für die Signatur von Transaktionen und als Zugriff auf BTC verwendet wird, muss er in der Wallet geschützt werden. Die Referenzimplementierung „Bitcoin Core Wallet“ verwendet hierfür AES-256. [Fr15] Private Schlüssel in Bitcoin sind `256 Bit` lang und werden in zwei Blöcken je `128 Bit` im CBC-Modus (Cypher Block Chaining) codiert. Hier wird vor dem Verschlüsseln des zweiten Blocks dieser im Klartext mit dem verschlüsselten ersten Block per `XOR` verknüpft. [PP10] Die asymmetrische Verschlüsselung und ihre Verwendung in Bitcoin werden im nächsten Kapitel beschrieben.

3.2.2 Asymmetrische Verschlüsselung

Bitcoin nutzt die asymmetrische Verschlüsselung für die Erstellung von Bitcoin Adressen und innerhalb der Blockchain für das Signieren und Verifizieren von Transaktionen. Bitcoin verwendet als asymmetrischen Algorithmus Elliptic Curves (EC) zusammen mit dem Digital Signature Algorithm (DSA) als Schema für digitale Signaturen.

Asymmetrische Verschlüsselung oder auch „Public Key“ Verschlüsselung wurde von Diffie, Hellmann und Merkle in den 70er für den sicheren Schlüsselaustausch über einen unsicheren Kanal entwickelt. Dies ist mit der symmetrische Verschlüsselung nicht möglich. [Fr15] Im Gegensatz zur symmetrischen Verschlüsselung gibt es einen öffentlichen (public) und einen geheimen (private) Schlüssel zwischen denen eine mathematische Asymmetrie besteht. Mit dem privaten Schlüssel kann der öffentliche Schlüssel berechnet werden aber nicht umgekehrt. Eine mit dem öffentlichen Schlüssel verschlüsselte Nachricht kann nur mit dem zugehörigen privaten Schlüssel entschlüsselt werden.

Eine weitere Anwendung für die asymmetrische Verschlüsselung sind digitale Signaturen, die nachweisbar den Urheber einer Nachricht bestätigen. Bei digitalen Signaturen wird eine Nachricht mit dem private Schlüssel signiert. Die Signatur wird mit der Nachricht zusammen verschickt, so dass der Empfänger mit dem zugehörigen öffentlichen Schlüssel die Signatur der Nachricht verifizieren kann. Wegen der mathematischen Asymmetrie zwischen privatem und öffentlichem Schlüssel ist die Integrität und Authentizität der Nachricht sichergestellt. Statt der Nachricht wird der Hashwert der Nachricht signiert, da dieser unabhängig zur Größe der Nachricht eine feste Länge besitzt. [PP10]

EC basiert auf dem diskreten logarithmischen Problem über die Addition und Multiplikation von Punkten auf einer elliptischen Kurve aus Formel 1. Der öffentliche Schlüssel ist ein Punkt (`x, y`) auf dieser Kurve, der sich aus der Multiplikation des privaten Schlüssels mit einem konstanten Startpunkt `A` auf der Kurve berechnet. [An15] Certicom [Ce10] definiert u.a. Startpunkt `A` und die Parameter der elliptischen Kurve aus Formel 1 in verschiedenen Sicherheitslevel und Standards. Bitcoin verwendet das Parameterset „secp256k1“ mit `256 Bit` Schlüssellänge und `128 Bit` Sicherheitslevel d.h. es sind 2^{128} Operationen notwendig, um den privaten Schlüssel mit `256 Bit` zu finden.

$$y^2 = x^3 + a * x + b \; mod \; p$$

FORMEL 1: ELLIPTISCHE KURVE

ES gibt mit DSA oder RSA Verfahren die für denselben Sicherheitslevel eine Schlüssellänge von `3072 Bit` besitzen. Daher wurde ECDSA von Satoshi Nakamoto in Bitcoin verwendet [Fr15]. DSA ist ein NIST [NI15] Standard und wurde 1991 erstmalig spezifiziert. ECDSA ist eine Variante von DSA basierend auf elliptischen Kurven aus Formel 1, die vom American National Standards Institute (ANSI) 1998 standardisiert wurde. Hierin sind ebenfalls analog zu Certicom [Ce10] alle Parameter für die Generierung des privaten Schlüssel und für die elliptische Kurve aus Formel 1 zur Berechnung des öffentlichen Schlüssels enthalten. In

ECDSA wird die Signatur ebenfalls durch einen Punkt (r, s) auf der elliptischen Kurve aus Formel 1 repräsentiert. Berechnet wird zunächst r als x-Koordinate über die Multiplikation des Startpunktes A mit einer zufällig erzeugten und einmalig zu verwendenden Nummer k. Dann wird s als y-Koordinate berechnet unter Berücksichtigung von r, der erzeugten Nummer k, dem privaten Schlüssel und dem Hashwert der Nachricht. [Fr15] Mit der Nachricht und der Signatur sowie in ECDSA definierter Parameter kann der öffentliche Schlüssel berechnet werden. Damit kann der Empfänger die Integrität und Authentizität der Nachricht bzw. des Sender verifizieren.

Bitcoin verwendet innerhalb von ECDSA 2048 Bit und 256 Bit Primzahlen als DSA Parameter. Öffentliche Schlüssel und Signaturen sind mit je 256 Bit für die Koordinaten (x, y) insgesamt 512 Bit lang. Da sich y aus x berechnen lässt, kann der öffentliche Schlüssel durch das Weglassen von y komprimiert werden. Hierzu werden 8 Bit als Präfix vorangestellt, der angibt ob ein komprimierter Schlüssel vorliegt. Dies reduziert die Schlüssellänge (512 Bit + 8 Bit – 256 Bit = 264 Bit) und die Größe von Transaktionen. [An15] In Transaktionen gibt es mehrere Eingangswerte, die jeweils mit einem privaten Schlüssel signiert wurden. Bitcoin verwendet pro Transaktion ein „Public Key“ Schlüsselpaar als zusätzliche Sicherheit um die Privatsphäre zu erhöhen [Na08]. Im nächsten Kapitel werden Hashfunktionen beschrieben, die in Bitcoin für Signaturen verwendet werden.

3.2.3 Hashfunktionen

Bitcoins Sicherheit und Mining basiert auf Hashfunktionen. Die Hashfunktion SHA-512 wird von der Referenzimplementierung „Bitcoin Core Wallet“ verwendet. Die Hashfunktionen SHA-256 und RIPEMD160 werden zur Generierung der öffentlichen Bitcoin Adresse benutzt. Darüber hinaus basieren die kryptografische Verifikation in der Blockchain und digitale Signaturen mit ECDSA aus dem vorherigen Kapitel auf SHA-256.

Hashfunktionen erzeugen einen digitalen Fingerabdruck für eine Nachricht beliebiger Länge. Der durch die Funktion berechnete und eindeutige Hashwert mit einer festen Länge stellt zusammen mit der Nachricht dessen Integrität sicher. Es gibt schlüsselbasierte Hashfunktionen wie MAC (Message Authentication Code) und schlüssellose Hashfunktionen wie SHA-256. Wichtigste Anforderungen dabei sind nach Paar und Pelzl [PP10]:

1. Einwegfunktion d.h. es nicht möglich aus dem Hashwert die Nachricht zu ermitteln
2. Keine identischen Hashwerte für unterschiedliche Nachrichten und umgekehrt

SHA-512 und SHA-256 bekannt unter dem Namen SHA-2 sind definiert im Secure Hash Standard (SHS) vom NIST [NI02] als Nachfolger von SHA-1 aus den 90er mit 160 Bit. Der Wert hinter SHA steht für die Länge des Hashwertes. Die Nachricht wird bei SHA-256 auf 512 Bit Blöcke geparst bzw. aufgefüllt. Bei SHA-512 müssen die Blöcke 1024 Bit groß sein. Beide Algorithmen besitzen acht initiale Hashwerte sowie 64 (SHA-256) bzw. 80 (SHA-512) Konstanten. In SHA-256 werden 32 Bit Wörter bzw. 64 Bit Wörter in SHA-512

verwendet. Hierauf findet die Hashberechnung in 64 bzw. 80 Durchläufen über sechs logische Funktionen statt. Jeder Durchlauf ermittelt acht Hashwerte (8 * 32 Bit = 256 Bit, 8 * 64 Bit = 512 Bit) für die Berechnung der nächsten Iteration. Zuletzt werden die Werte verkettet und bilden den Hashwert von SHA-256 bzw. SHA-512.

RIPEMD160 (RACE Integrity Primitives Evaluation Message Digest) wurde in Europa von 1988 bis 1992 entwickelt, 1996 veröffentlicht und ist wie SHA-256, SHA-512 und andere in ISO/IEC standardisiert. Der Wert 160 steht für die Länge des Hashwertes. Die Nachricht wird wie bei SHA-256 auf 512 Bit Blöcke geparst. Der Algorithmus besitzt zehn Konstanten und fünf initiale Hashwerte. Die Hashberechnung findet in 80 Durchläufen über fünf logische Funktionen auf Basis von 32 Bit Wörtern statt. Jeder Durchlauf ermittelt fünf Hashwerte (5 * 32 Bit = 160 Bit) für die Berechnung der nächsten Iteration. Zuletzt werden die Werte verkettet und ergeben den Hashwert von RIPEMD160. [DBP96], [DBP12]

Die „Bitcoin Core Wallet“ verwendet innerhalb der symmetrischen Verschlüsselung des privaten Schlüssels 25.000 Durchläufe von SHA-512 auf ein vom Benutzer eingegebenes Passwort zusammen mit einem zufälligen Wert (Salt). Hashwert des Passworts und Salt werden zusammen gespeichert. Damit wird das Passwort zusätzlich verstärkt und „Brute-Force“ Angriffe verlangsamt. [Fr15] Bitcoin Adressen werden über den öffentlichen 512 Bit langen Schlüssel (256 Bit für komprimierte Schlüssel) der asymmetrischen Verschlüsselung oder einer Skriptadresse mit den Hashfunktionen SHA-256 und RIPEMD160 generiert. Zuerst wird der Hashwert mit SHA-256 berechnet. Anschließend wird hieraus der Hashwert mit RIPEMD160 berechnet und die Adresse von 256 Bit auf 160 Bit verkürzt. Der Adresse werden 8 Bit für den Adresstyp als Präfix vorangestellt. Hieraus wird per SHA-256^2 (zweimalige Anwendung von SHA-256) ein Hashwert gebildet, wobei die ersten 32 Bit als Prüfsumme der Adresse als Suffix angehängt werden. Diese Adresse wird Base58 kodiert, um Verwechslungen durch ähnliche Zeichen zu vermeiden. [An15], [Fr15]

Bitcoin Mining basiert auf der Lösung eines kryptografisches Puzzles „Proof-of-Work“ (PoW) um neue Blöcke der Blockchain hinzuzufügen. Hierfür wird solange ein Wort für den Block generiert bis hierzu ein SHA-256^2 Hashwert berechnet wurde, der mit einer definierten Anzahl an Nullen beginnt. Diese Anzahl beschreibt die Schwierigkeit und wird alle 2.016 Blöcke vom Bitcoin Client neu berechnet. [An15], [Fr15] Die Hashrate wird gemessen in Hashes per second (H/s) und hat sich von Mega hashes in 2009/2010, zu Giga hashes in 2011, zu Tera hashes in 2012/2013 bis zu Peta hashes in 2014/2015 potenziert. [Bl16b]

Ralph Merkle [Me79] hat 1979 im Rahmen von digitalen Signaturen einen binären Hashbaum, auch Merkle Baum genannt, erstmalig beschrieben und patentiert [Eu82]. Mit dem Merkle Baum werden große Datenmengen per Hashverfahren reduziert und deren Integrität analog zu digitalen Signaturen sichergestellt. Hierfür sind $2^{n+1}-1$ Hashberechnungen für $N=2^n$ Nachrichten erforderlich. Bei der Verifizierung sind $\log_2(N)$ Schritte erforderlich. In Bitcoin werden über den Merkle Baum SHA-256^2 Hashwerte

einzelner Transaktionen abgebildet, wobei die Blätter den Hash der Transaktionen wie in Abb. 6 darstellen.

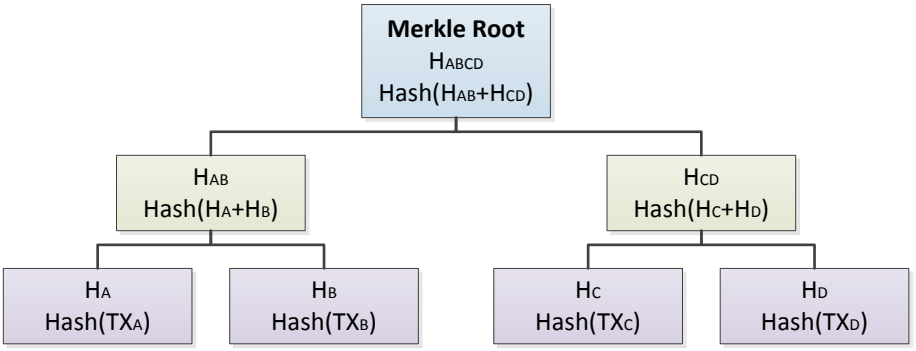

ABB. 6: MERKLE BAUM MIT VIER TRANSAKTIONEN [AN15]

Abb. 6 zeigt die String Verkettung der Hashwerte zweier Knoten, woraus ein SHA-256^2 Hash für den übergeordneten Knoten bis zur Wurzel (`Root`) gebildet wird. Der Merkle Baum wird vom Miner bei der Blockerstellung erzeugt. Wenn `N` ungerade ist wird der binäre Baum durch Duplizierung des letzten Hash ausbalanciert. Möchte ein Client prüfen ob Transaktion `D` aus Abb. 6 in einem Block vorhanden, muss dieser nur einen Teil des Merkle Baum nämlich H_C und H_{AB} da `Root` und Hash der Transaktion `D` bekannt sind. Es sind $\log_2(4)=2$ Hashberechnungen notwendig. [An15]

SHA-256^2 statt SHA-256 wurden laut Franco [Fr15] vermutlich von Satoshi Nakamoto für Bitcoin als Sicherheit vor sogenannten „length extension“ Angriffen sowohl für die Generierung von Adressen als auch für das Bitcoin Mining ausgewählt. RIPEMD wurde von Satoshi Nakamoto zur Reduzierung der Länge von Adressen verwendet und Reduktion der Transaktionsgröße. Da jede Transaktion mehrere Adressen enthält, ein Block mehrere Transaktionen enthält und jeder Block öffentlich in der Blockchain gespeichert wird, ist die Datengröße ein wichtiger Aspekt. Der Aufbau der Blockchain mit seinen Blöcken wird im nächsten Kapitel beschrieben.

3.3 Aufbau der Blockchain

Jeder Block in der Blockchain wird durch seinen Hashwert eindeutig identifiziert und ist mit dem Hashwert seines Vorgängers verknüpft. Diese chronologische Verkettung von Blöcken sind verknüpfte Zeitstempel mit denen die Existenz eines Blocks zu einer bestimmten Zeit nachgewiesen werden kann. Hierfür wird analog zu den digitalen Signaturen der Hash eines Blocks als digitaler Zeitstempel verwendet.

Bei verknüpften Zeitstempeln wird, um die Sicherheit der Kette zu erhöhen, der Hashwert aus dem Hash des Blocks und des vorherigen Blocks gebildet. Abb. 7 zeigt die Hashkette gebildet aus den Blockhashes als „Timestamp Server“.

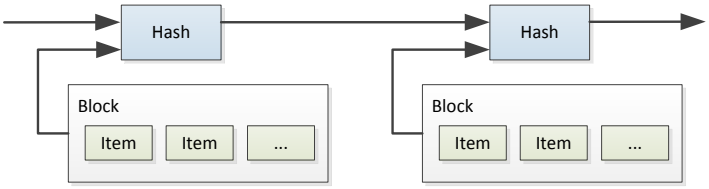

ABB. 7: BLOCKHASHES UND VERKNÜPFTE ZEITSTEMPEL [NA08]

Im Gegensatz zu einer zentralen Autorität z.B. einer TSA (Time Stamping Authority) basiert die Blockchain auf einem öffentlich verbreiteten Hash als Lösung des PoW. Das Problem muss schwer zu lösen und seine Lösung leicht zu prüfen sein. Es gibt zwei unterschiedliche Protokolle für PoW Systeme, die jeweils Schutz vor DoS (Denial of Service) Angriffe bieten. Beim Challenge-Response Protokoll verlangt der Server für die Response eines Client Request ein PoW als Challenge wie z.B. ein CAPTCHA (Completely Automated Public Turing test to tell Computers Humans Apart). Die Blockchain nutzt das Solution-Verification Protokoll. Dem Client wird eine Aufgabe auf Basis eines Algorithmus und sich ändernder Schwierigkeit gestellt, dessen Lösung zum Server geschickt und dort verifiziert wird. [Fr15]

Im vorherigen Kapitel wurde die partielle Hash Inversion als PoW innerhalb der Blockchain beschrieben. Ein erstellter Block ist nur dann reversibel, wenn sein PoW und der von allen nachfolgenden Blöcken erneut ausgeführt werden. Die längste Kette stellt den größten PoW, die meiste Sicherheit und den Konsens der im Netzwerk beteiligter Knoten dar. [Na08]

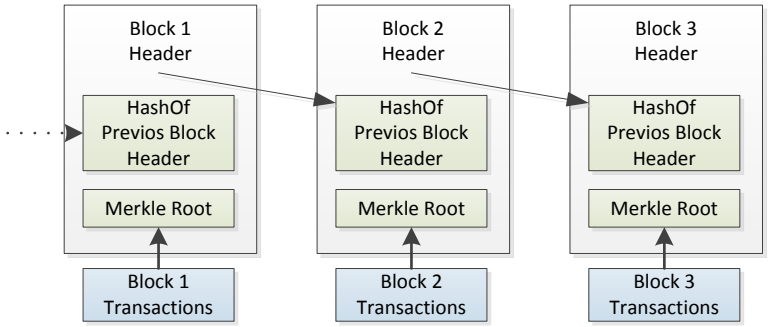

ABB. 8: BLÖCKE IN DER BITCOIN BLOCKCHAIN [BI16H]

Ein Block wird eindeutig über den SHA^2 Hash seines Blockheader identifiziert und ist jeweils mit dem Hash seines Vorgänger verknüpft. Abb. 8 zeigt die Verknüpfung von Blöcken über den Hash und den Merkle Baum als binärer Hash Baum über Transaktionen im Blockheader. Neben diesem enthält ein Block mit den Transaktionen siehe Abb. 8 das wichtigste Element. Weiter besitzt ein Block die Größe und Anzahl seiner Transaktionen. Aktuelle Statistiken wie die durchschnittliche Blockgröße, Anzahl Transaktionen pro Block, und Größe der Blockchain sind im Bitcoin Block Explorer [Bl16b] verfügbar. Am 18.01.2016 war die Blockchain `52.352 Mbyte` und ein Block mit durchschnittlich 1.386 Transaktionen ca. `728,37 KByte` groß. Nicht jeder Knoten im Bitcoin Netzwerk speichert die komplette Blockchain. Tab. 5 zeigt die Struktur eines Blocks und den Metadaten seines Blockheader, die im Bitcoin Core Client in einer LevelDB gehalten werden. LevelDB ist eine Open Source Database von Google für Key-Value Paare. [GD16]

Größe	Datentyp	Feld	Beschreibung
4 Byte	uint32_t	Blockgröße	Größe des Blocks in Bytes
80 Byte		**Block Header**	**Metadaten des Block**
4 Byte	uint32_t	Version	Versionsnummer des Bitcoin Client
32 Byte	char[32]	Hash des vorherigen Block	Referenz auf den Hash des vorherigen Blocks in der Kette
32 Byte	char[32]	Merkle Root	Hash vom Root des Merkle Baums mit den Transaktionen
4 Byte	uint32_t	Zeitstempel	Erstellzeitpunkt des Blocks (Unix Zeit in Sekunden)
4 Byte	uint32_t	Schwierigkeit	Schwierigkeit des PoW für den Block
4 Byte	uint32_t	Nonce	Generiertes Wort des PoW für den Block
1-9 Byte	VarInt	Transaktionscounter	Zähler für die Transaktionen in dem Block
x Byte	**raw**	**Transaktionen**	**Transaktionen in dem Block**

TAB. 5: STRUKTUR EINES BITCOIN BLOCKS [AN15]

Blockheader bestehen aus insgesamt 80 Byte wobei 32 Byte Felder jeweils SHA-256^2 Hashwerte repräsentieren. Die Versionsnummer gibt an welche Validierungsregeln für den Block gelten. Zum Beispiel muss das Erstellungsdatum des Blocks größer oder gleich dem Mittelwert der letzten 11 Blöcke sein. Liegt das Datum mehr als zwei Stunden in der Zukunft eines vollständigen Bitcoin Client wird es von diesem nicht akzeptiert. Außerdem darf ein Block nicht größer als 1 MByte sein. [Bi16h] Transaktionen sind im Rohdatenformat in einem Block enthalten und werden im Kapitel 3.5 beschrieben. Einige Knoten im Netzwerk speichern lediglich den Blockheader. Im vorherigen Kapitel wurde der Merkle Baum und die partielle Hash Inversion als PoW in der Blockchain beschrieben, die mit dem Root des Merkle Baum, der Schwierigkeit und Nonce des PoW im Blockheader gespeichert werden.

Mit BIP-34 (Bitcoin Improvement Proposal) wurde Version 2 eingeführt, die als weiteren Parameter die Blockhöhe erfordert. Diese beginnt beim Genesisblock 0 und wird mit jedem Block erhöht. Sie identifiziert die Position eines Blocks in der Blockchain und ist aufgrund von Blockchain Forks (Abzweigungen) nicht eindeutig. Mit Version 2 und BIP-34 wurde im September 2012 erstmalig ein Soft Fork durchgeführt d.h. eine Übergangsphase mit zweiter Blockchain. Änderungen am Protokoll oder Client werden über BIPs kommuniziert und meist stufenweise eingeführt. BIPs müssen von über 55% der Miner akzeptiert werden. [Da16b]

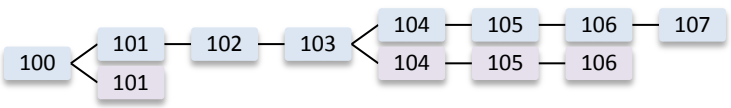

ABB. 9: BITCOIN BLOCKCHAIN FORKS [FR15]

Ein Fork mit einem Block gibt es nach Franco [Fr15] durchschnittlich alle 50 Blöcke während ein Fork mit zwei oder mehr Blöcken sehr selten vorkommt siehe auch Anzahl verwaister

Blöcke im Bitcoin Block Explorer [Bl16b]. Ein Fork tritt auf wenn zwei Miner einen neuen Block zur selben Zeit hinzufügen siehe 101 in Abb. 9. In diesem Fall gewinnt der Miner mit dem Block auf dem der nächste Block gebildet wird. Die längste Blockchain wird in Abb. 9 blau dargestellt mit den Blöcken der Höhe 100 bis 107 ist immer die richtige, so dass die Blöcke mit derselben Blockhöhe (lila) der kürzeren Kette verwiesen werden. Transaktionen von Blöcken eines Blockchain Fork gehen in einen Cache mit unbestätigten Transaktionen, sofern diese nicht bereits in einem Block der längsten Blockchain enthalten sind. Nach Antonopoulos [An15] ist das Blockintervall von zehn Minuten ein Kompromiss zwischen Zeit für die Transaktionsbestätigung und der Wahrscheinlichkeit des Auftreten eines Fork.

Im nächsten Kapitel wird das P2P-Netzwerk mit unterschiedlichen Knoten beschrieben. Diese haben verschiedene Rollen und Funktionen. Ein großer Unterschied ist die Speicherung und Aktualisierung der Blockchain sowie die Verifikation von Transaktionen.

3.4 P2P-Netzwerk

Bitcoin wurde von Nakamoto [Na08] als öffentliches P2P-Netzwerk entworfen, welches Transaktionen in der Blockchain transparent über einen hashbasierten PoW mit einem Zeitstempel versieht. Jeder Knoten arbeitet im Konsens dieses Netzwerks auf der längsten Blockchain wie im vorherigen Kapitel beschrieben.

Wenn zwei Blöcke mit derselben Höhe über das Netzwerk verteilt werden, kommen diese bei den Knoten in unterschiedlicher Reihenfolge an. Der erste Block der beim Knoten ankommt wird verwendet. Der zweite Block derselben Höhe wird gespeichert falls diese Blockchain länger wird als die auf dem der Knoten aktuell arbeitet. In dem Fall wechselt der Knoten zur längeren Blockchain. Nakamoto [Na08] beschreibt folgende Schritte als Prinzipien innerhalb des Netzwerks:

1. Neue Transaktionen werden allen Knoten gesendet.
2. Jeder Knoten sammelt neue Transaktionen in einem Block.
3. Jeder Knoten arbeitet an einem PoW für seinen Block.
4. Wenn ein Knoten ein PoW gefunden hat, wird der Block an alle Knoten gesendet.
5. Knoten akzeptieren Blöcke nur wenn alle seine Transaktionen valide und nicht bereits verwendet wurden.
6. Knoten akzeptieren den Block wenn sie den nächsten Block in der Kette auf dem Hash des akzeptierten Blocks aufbauen.

Antonopoulos [An15] beschreibt vier Funktionen die ein Knoten im Netzwerk besitzt. Die erste Funktion ist das Routing von Nachrichten. Jeder Knoten im Netzwerk muss diese Funktion unterstützen. Die zweite Funktion ist das Speichern und Aktualisieren einer vollständigen Kopie der Blockchain. Diese Knoten werden auch vollständige Knoten genannt. Die dritte Funktion ist das Mining. Diese Knoten besitzen meist spezielle Hardware für die

Hashberechnung. Die letzte Funktion besitzen Wallets primär mit der Verwaltung von privaten und öffentlichen Schlüsseln. Der Knoten der Referenzimplementierung „Bitcoin Core“ bestehend aus Client und Wallet umfasst alle vier Funktionen [Na09]. Im Weiteren werden nur die Knoten mit vollständiger Blockchain und leichtgewichtige SPV (Simplified Payment Verification) Wallets Knoten betrachtet, da sie sich in der Verwendung der Blockchain grundlegend unterscheiden. Auf Mining und erweiterte Netzwerke durch Mining Pools und weitere Protokolle wie z.B. Stratum oder dem Payment Protokoll in BIP-70 wird nicht weiter eingegangen, da der Fokus auf der Funktion und Kommunikation von SPV und vollständigen Blockchain Knoten liegt.

Innerhalb des P2P-Netzwerks findet die Kommunikation über TCP (Transmission Control Protocol) statt. Es gibt zwei Testnetze mit den Portnummern `18333` und `18444` sowie das Hauptnetz mit der Portnummer `8333`. Die Protokollversion ist abhängig vom Client mit den enthaltenen BIPs [Da16b]. In einem Netzwerk gibt es Daten- und Kontrollnachrichten jeweils mit `24 Byte` Header mit den Feldern aus Tab. 6.

Größe	Datentyp	Feld	Beschreibung
`4 Byte`	`char[4]`	Startstring	Identifikation für das Netzwerk
`12 Byte`	`char[12]`	Befehlsname	Zähler für die Transaktionen in dem Block
`4 Byte`	`uint32_t`	Payloadgröße	Größe der Nutzdaten max. 32 `MByte`
`4 Byte`	`char[4]`	Prüfsumme	ersten `4 Byte` des SHA-256^2 der Payload

TAB. 6: MESSAGE HEADER IM BITCOIN P2P-NETZWERK [BI16H]

Ein Client muss für alle Nachrichten mindestens eine Verbindung zu einem anderen Knoten besitzen. Die erfolgreichsten Adressen von langlaufenden, stabilen Knoten werden vom Client gespeichert und eine Verbindung zu diesen bei Neustart wieder aufgenommen. Über Datennachrichten können Blöcke und Transaktionen angefragt und übertragen werden. Kontrollnachrichten verwalten Verbindungen zwischen den Knoten.

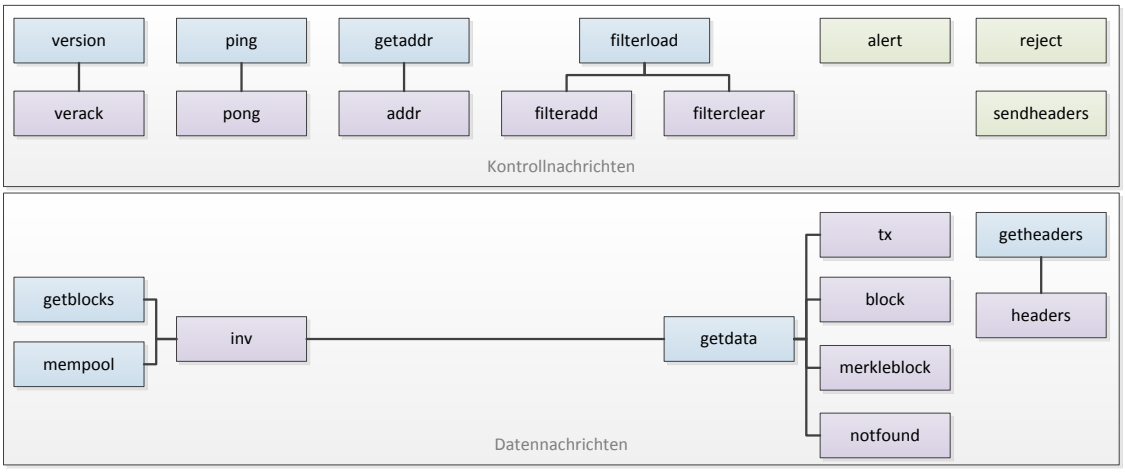

ABB. 10: KONTROLL- UND DATENNACHRICHTEN IM BITCOIN P2P-NETZWERK [BI16H]

Abb. 10 zeigt alle Befehle beider Nachrichtentypen. Synchrone Kommunikation wird mit Requests in blau und Responses in lila, asynchrone Nachrichten werden grün dargestellt. Beispielsweise wird mit `version` und `verack` ein Handshake durchgeführt. Über `addr` und `getaddr` werden Netzwerkadressen angefragt aber auch unaufgefordert übermittelt. `alert` warnt vor Problemen innerhalb des Netzwerks und ist die einzige über eine Signatur authentifizierte Nachricht die nur von Bitcoin Core Entwicklern stammt. Über `getblocks` werden Hashes von Blockheader und mit `mempool` unbestätigten Transaktionen angefragt, die mit `inv` (Inventory) beantwortet werden. `inv` kann auch unaufgefordert für die Benachrichtigung neuer Transaktionen oder Blöcke gesendet werden. Mit `getdata` werden Transaktionen, Blöcke oder Teile des Merkle Baum angefragt, die mit `tx`, `block`, `merkleblock` oder `notfound` beantwortet werden. [Bi16h]

Ein neuer Knoten kennt anfangs nur Genesis Block 0 und muss sich mit Knoten im Netzwerk verbinden, um mit `getblocks` das Inventar `inv` der verbundenen Knoten abzufragen. Von diesen werden `block` bis zur vollständigen Blockchain über `getdata` abgerufen. Bei mehr als `50 GByte` Größe der Blockchain dauert der Vorgang einige Tage. Ein vollständiger Knoten synchronisiert sich fortlaufend mit dem Netzwerk über den Vergleich von Blockhöhe und Hashes seiner Blockchain. Er validiert eine Transaktion über die komplette Blockchain und kann hierüber verifizieren, dass es sich nicht um ein „Double Spending“ handelt. [An15]

Im Gegensatz zu vollständigen Knoten sind SPV Knoten als leichtgewichtige Clients in Speicherplatz und Rechenkapazität eingeschränkt wie z.B. Smartphones oder Tablets. Am 20.01.2016 war die Blockchain laut Bitcoin Block Explorer [Bl16b] mit 394.193 Blöcken `52.977 Mbyte` groß. Die Summe der Blockheader zu diesem Zeitpunkt betrug bei `80 Byte` pro Header ca. `30 Mbyte`. Eine vollständige Blockchain ist hiernach um den Faktor 1.761 größer als die Summe seiner Blockheader. SPV Knoten sind nach Nakamoto [Na08] keine vollständigen Knoten, da sie nur eine Kopie der längsten Blockchain bestehend aus den Blockheadern besitzen. Diese werden analog zu `getdata` mit `getheaders` ermittelt. Für die Verifikation von Transaktionen wird bei SPV über einen Teil des Merkle Baums geprüft ob eine Transaktion in dem Block mit dem Merkle Baum enthalten ist. Der Merkle Baum zu dem Block ermittelt der Knoten über `getdata` und bekommt als Antwort einen `merkleblock` und wenn vorhanden die Transaktion `tx`. [Bi16h] Wenn die nächsten sechs Blöcke auf dem Block mit der zu validierenden Transaktion in der Blockchain angehängt werden ist diese Transaktion für den SPV Knoten validiert. Antonopoulos [An15] beschreibt bei SPV die Möglichkeit von DoS-Angriffen, da diese Knoten im Gegensatz zu vollständigen Knoten bei der Validierung einer Transaktion auf die Knoten im Netzwerk angewiesen sind. Außerdem können sie unbeabsichtigt Adressen ihrer Wallet preisgeben. Hierfür wurden in BIP-37 Kontrollnachrichten `filteradd`, `filterclear` und `filterload` für Bloom Filter eingeführt. Mit Bloom Filtern können Suchmuster in beliebiger Präzision für öffentliche und P2SH (Pay to Script Hash siehe Kapitel 3.5) Adressen spezifiziert werden, wobei die Präzision das Datenvolumen im Netzwerk versus Privatsphäre steuert. Für eine detaillierte

Beschreibung von Bloom Filtern und BIPs sowie Protokoll- und Clientversionen siehe BIPs [Da16b] und Bitcoin [Bi16h].

Im folgenden Kapitel wird die Struktur einer Transaktion, ihren Typen und der stapelbasierte Skriptsprache zum Sperren und Entsperren von Transaktionen bzw. deren Ein- und Ausgänge beschrieben.

3.5 Transaktionen

In den vorherigen beiden Kapiteln wurde der Aufbau der Blockchain mit seinen Blöcken als Transporteinheit für Transaktionen und dessen Verteilung und Verifikation von vollständigen und leichtgewichtigen Knoten im Netzwerk beschrieben. Bitcoin wurde konzipiert um Transaktionen in einem Netzwerk dezentral über die Blockchain zu erstellen, zu speichern, zu verteilen und zu validieren. Transaktionen sind das wichtigste Element und werden in diesem Kapitel ausführlich betrachtet.

In Kapitel 3.3 wurde der Aufbau eines Blocks beschrieben. Das letzte Feld in einem Block besitzt eine variable Anzahl an `Bytes` im Rohformat welches in Tab. 7 dargestellt auf „top-level“ Ebene durch eine Transaktion repräsentiert wird. Diese beschreibt den Übergang von einem oder mehreren Eingängen zu einem oder mehreren Ausgängen.

Größe	Datentyp	Feld	Beschreibung
4 Byte	uint32_t	Version	Versionsnummer
1-9 Byte	VarInt	Eingangszähler	Zähler für die Eingangswerte
x Byte	txIn	Eingänge	Eine oder mehrere Eingangswerte
1-9 Byte	VarInt	Ausgangszähler	Zähler für die Ausgangswerte
x Byte	txOut	Ausgänge	Eine oder mehrere Ausgangswerte
4 Byte	uint32_t	Sperrzeit	Unix Zeit in Sekunden oder Blocknummer

TAB. 7: STRUKTUR EINER BITCOIN TRANSAKTION [AN15]

Die Transaktionsgröße ist abhängig von der Anzahl der Ein- und Ausgänge. Die Reihenfolge der Transaktionen im Rohformat innerhalb des Blocks muss der Reihenfolge des Merkle Baums entsprechen. Die Versionsnummer beschreibt welche Validierungsregeln für die Transaktion gelten. Die Sperrzeit gibt an wann die Transaktion über einen Block in die Blockchain frühestens aufgenommen werden darf und kann entweder als Unix Zeit in Sekunden (größer als 500 Millionen) oder Blocknummer bzw. Blockhöhe (kleiner oder gleich 500 Millionen) angegeben werden. Per Default wird 0 gesetzt, so dass die Transaktion unmittelbar ausgeführt werden soll. [Bi16h]

Jeder Transaktionsausgang enthält den zu spendenden Betrag (1 Satoshi = 10^{-8} BTC als kleinste Einheit) und einer Adresse bzw. einem Skript basierend auf dem öffentlichen Schlüssel des Empfängers. Nur dieser kann mit seinem privaten Schlüssel den Betrag in

weiteren Transaktionen wiederverwenden. Ein Eingang besitzt eine Referenz auf den Ausgang der vorherigen Transaktion. Der Bitcoin Client besitzt einen UTXO Cache (Unspent TransactionX Outputs) mit nicht gespendeten Ausgängen, der mit jedem neuen Block aktualisiert wird. Über diesen Cache werden Transaktionen validiert indem geprüft wird ob der referenzierte Ausgang vorhanden ist. Eine Transaktion muss immer den vollen Betrag aus dem UTXO Cache verwenden. Soll nur ein Teilbetrag transferiert werden muss eine Transaktion mit zwei Ausgängen erstellt werden. Eine mit dem Teilbetrag und eine mit dem Differenzbetrag an die eigene Adresse. Bei der Validierung der Transaktion wird weiter geprüft, ob die Summe der Eingänge größer als die Summe der Ausgänge ist. Der Differenzbetrag zwischen Ein- und Ausgängen einer Transaktion gilt als Transaktionsgebühr für das Aufnehmen der Transaktion in einen Block durch den Miner. Diese Gebühr ist nach Nakamoto [Na08] neben den BTC der Coinbase Transaktion Ansporn und Vergütung für den Support des Netzwerks durch Rechenleistung. [Fr15]

Die erste Transaktion in einem Block ist eine Coinbase Transaktion die im Gegensatz zu anderen Transaktionen nicht den Ausgang einer vorherigen Transaktion referenziert. Coinbase Transaktionen werden durch den Miner des Blocks erstellt, der hierfür 25 BTC erhält. [An15] Coinbase Transaktionen werden nicht weiter betrachtet. Das Format für den Eingang `txIn` einer Transaktion ist in Tab. 8 aufgeführt.

Größe	Datentyp	Feld	Beschreibung
1-9 Byte	VarInt	Skriptgröße	Größe des Skriptes in Bytes
x Byte	char[]	Signaturskript	Skript zum Entsperren des Ausgangs der vorherigen Transaktion
4 Byte	uint32_t	Sequenz	Sequenznummer
36 Byte	**outpoint**	**vorheriger Ausgang**	**Referenz auf den Ausgang einer vorherigen Transaktion**
32 Byte	char[32]	Hash	TXID als SHA-256^2 Hash der Transaktion
4 Byte	uint32_t	Index	Index auf den Ausgang innerhalb der Transaktion (0 für den ersten)

TAB. 8: STRUKTUR EINES BITCOIN TRANSAKTIONSEINGANGS [BI16H]

Das Signaturskript `scriptSig` entsperrt im Transaktionseingang den von dem Skript `scriptPubKey` referenzierten und gesperrtem Ausgang der vorherigen Transaktion. Es gibt fünf Transaktionstypen, die durch unterschiedliche Skripte verarbeitet werden können. [An15] Das Format für den Ausgang `txOut` einer Transaktion ist in Tab. 9 aufgeführt und enthält ein Skript `scriptPubKey` basierend auf dem öffentlichen Schlüssel des Empfängers welches einen Betrag bzw. den Ausgang einer Transaktion sperrt.

Größe	Datentyp	Feld	Beschreibung
8 Byte	uint64_t	Betrag	Menge an Satoshis (10^{-8} BTC)
1-9 Byte	VarInt	Skriptgröße	Größe des Skriptes in Bytes

x Byte	char[]	Public Key Skript	Skript zum Sperren des Ausgangs

TAB. 9: STRUKTUR EINES BITCOIN TRANSAKTIONSAUSGANGS [BI16H]

Das Guthaben einer Bitcoin Adresse ist die Summe der Beträge von Transaktionsausgängen die in keinen der folgenden Transaktionen als Eingang referenziert werden (UTXO). Das Guthaben einer Wallet ist die Summe der Guthaben aller Bitcoin Adressen. [An15]

Bitcoin verwendet für das Sperren und Entsperren von Transaktionen eine imperative, stapelbasierte Skriptsprache angelehnt an Forth. Die Skriptsprache ist zustandslos und ohne Schleifen nicht Turing-vollständig um potentielle Angriffe auf das Netzwerk zu vermeiden. Der Stapel wird nach dem LIFO (Last In / First Out) Prinzip mit Daten und Befehlen aus dem Skript abgearbeitet und gibt `TRUE` oder `FALSE` zurück. Hierüber wird die Transaktion autorisiert oder verworfen. [Fr15] In Abb. 11 wird der Ablauf einer P2PKH (Pay to Public Key Hash) Transaktion gezeigt bestehend aus den Skripten `scriptPubKey` und `scriptSig`.

STAPEL ↓	***SKRIPT →***	
	scriptSig	*scriptPubKey*
	<sig> <pubKey>	DUP HASH160 <pubKeyHash> EQUALVERIFY CHECKSIG
<sig>		
	scriptSig	*scriptPubKey*
	<sig> **<pubKey>**	DUP HASH160 <pubKeyHash> EQUALVERIFY CHECKSIG
<pubKey>		
<sig>		
	scriptSig	*scriptPubKey*
	<sig> <pubKey>	**DUP** HASH160 <pubKeyHash> EQUALVERIFY CHECKSIG
<pubKey>		
<pubKey>		
<sig>		
	scriptSig	*scriptPubKey*
	<sig> <pubKey>	DUP **HASH160** <pubKeyHash> EQUALVERIFY CHECKSIG
<pubKeyHash>		
<pubKey>		
<sig>		
	scriptSig	*scriptPubKey*
	<sig> <pubKey>	DUP HASH160 **<pubKeyHash>** EQUALVERIFY CHECKSIG
<pubKeyHash>		
<pubKeyHash>		
<pubKey>		
<sig>		
	scriptSig	*scriptPubKey*
	<sig> <pubKey>	DUP HASH160 <pubKeyHash> **EQUALVERIFY** CHECKSIG
<pubKey>		
<sig>		
	scriptSig	*scriptPubKey*
	<sig> <pubKey>	DUP HASH160 <pubKeyHash> EQUALVERIFY **CHECKSIG**
TRUE		

ABB. 11: P2PKH SKRIPT IN BITCOIN [AN15]

Abb. 11 zeigt mit den fett markierten Befehlen und Elementen im Stapel das Ergebnis des aktuellen Bearbeitungsschrittes innerhalb des Skripts beginnend mit der Ablage der Signatur `<sig>` der neuen Transaktion auf den Stapel gefolgt vom öffentlichen Schlüssel `<pubKey>` welcher oben auf den Stapel gelegt wird. Ab hier beginnt das Skript `scriptPubKey` mit der Duplizierung `DUP` des `<pubKey>` dessen Hash `HASH160` (siehe Kapitel 3.2.3)

anschließend berechnet und mit <pubKeyHash> oben auf den Stapel gelegt wird. Dabei wird ein <pubKey> vom Stapel entfernt. Hiernach wird der <pubKeyHash> aus dem scriptPubKey auf den Stapel gelegt und im nächsten Schritt mit EQUALVERIFY verglichen. Damit findet die erste Prüfung statt. Wenn beide Hashes nicht übereinstimmen wird die Transaktion invalidiert. Andernfalls werden beide Hashes vom Stapel entfernt. Zuletzt wird mit CHECKSIG über den öffentlichen Schlüssel <pubKey> geprüft ob die Signatur <sig> der Transaktion korrekt ist. Bei erfolgreicher zweiter Prüfung werden beide Elemente vom Stapel genommen und TRUE für eine valide Transaktion auf den Stapel gelegt. Ein P2PK Skript ist ähnlich im Aufbau aber deutlich einfacher und kürzer, da hier kein Hash berechnet und geprüft wird. In scriptSig ist nur <sig> enthalten und in scriptPubKey mit <pubKey> der öffentliche Schlüssel und CHECKSIG als Befehl zum Prüfen der Signatur. [Fr15]

Befehle oder Operationen in Skripten werden durch OpCodes mit dem Präfix OP_ dargestellt. Im Beispiel aus Abb. 11 dargestellt ohne Präfix wurden OP_DUP, OP_HASH160 (verwendet OP_HASH256), OP_EQUALVERIFY (Sequenz aus OP_EQUAL und OP_VERIFY) und OP_CHECKSIG verwendet. Für eine vollständige Liste und Beschreibung aller OpCodes und Signatur Hashtypen siehe Bitcoin [Bi16h], [Bi16e].

P2PKH Transaktion sind mit 84,1 % die häufigsten Transaktionen in Bitcoin. Daneben gibt es P2PK (Pay to Public Key) Transaktionen mit 12,9%, P2SH Transaktionen mit 2,5%, Multi-Signatur und Datentransaktionen OP_RETURN jeweils mit 0,2% als Standard Transaktionen. [We16] Da durch P2PKH und P2PK mit 97% aller Transaktionen in Bitcoin abgedeckt sind, werden die anderen Transaktionstypen nicht detailliert betrachtet. P2PK Transaktionen verwenden statt dem Hash bzw. der Bitcoin Adressen öffentliche Schlüssel. In Multi-Signatur Transaktionen (BIP-11) werden M öffentliche Schlüssel verwendet, die durch N (max. 3) Signaturen entsperrt werden können. P2SH Transaktionen (BIP-16) verwenden in scriptSig N Signaturen und M öffentliche Schlüssel wobei diese in scriptPubKey als Hash enthalten sind. Datentransaktionen besitzen kein scriptSig und werden verwendet um maximal 40 Byte Daten in der Blockchain zu speichern. [Bi16h], [Da16b]

Da Transaktionen signiert sind und keine vertraulichen Informationen enthalten, können sie in transportbasierten Netzwerken beliebig verteilt werden. Die Übertragung kann über unsichere Übertragungskanäle wie z.B. über Wi-Fi, Bluetooth oder NFC (Near Field Communication) erfolgen. Darüber hinaus ist es möglich Transaktionen kodiert als Emoticons, QR-Codes (Quick Response) oder als Textnachricht in Chats und Foren zu verteilen. Mit Bitcoin wird aus Geld eine Datenstruktur, die über seine Skriptsprache programmierbar ist. [An15]

Im nächsten Kapitel werden Anwendungsszenarien und Beispiele in nahezu alle Bereichen aufgezeigt, die mit der Blockchain möglich sind. Eine entscheidende Rolle hierbei besitzt die Skriptsprache sowie scriptSig und scriptPubKey mit ihren Transaktionstypen.

3.6 Anwendungsszenarien

In den vorherigen Kapiteln wurde der Hintergrund und die Motivation von Bitcoin, dessen kryptografischen Grundlagen, die Struktur der Blockchain sowie das P2P-Netzwerk und Transaktionen behandelt. Dieses Kapitel beschreibt die Anwendung der Blockchain anhand von Beispielen. Dabei werden Anwendungsszenarien nach Swan [Sw15] in drei Kategorien oder Versionen unterteilt. Anwendungen in Version 1.0 beziehen sich auf die Zahlung mit der Währung wie in Nakamoto [Na08] ursprünglich beschrieben. Anwendungen in Version 2.0 besitzen umfangreichere Transaktionen. Hierin werden Smart Properties und Contracts abgebildet. Anwendungen in Version 3.0 gehen über Transaktionen in den Bereichen Währung, Finanzen und Wirtschaft hinaus. Sie finden sich vor allem in den Bereichen Regierung, Gesundheit, Wissenschaft, Bildung, Kultur und Kunst. Damit werden alle Eigenschaften einer Smart City aus Kapitel 2.4 abgedeckt.

3.6.1 Blockchain 1.0 - Währungen und Zahlungssysteme

Die Blockchain 1.0 umfasst die digitale Währung und das Zahlungssystem. Bitcoin wurde von Nakamoto [Na08] als Zahlungssystem mit der ersten kryptografischen und gleichnamigen Währung Bitcoin (BTC) konzipiert. Mittlerweile gibt eine große Anzahl Alt-Coins basierend auf der Blockchain wobei Bitcoin mit über sechs Milliarden Marktkapital bei über 15 Millionen BTC mit 399,99 USD pro BTC (Stand 24.01.2016) die größte darstellt [Co16d].

Bitcoin ist auf maximal 21 Millionen Einheiten limitiert die im Mai 2140 durch Mining erreicht sein werden. Das Wachstum neuer BTC über die Zeit ist definiert. Bitcoin ist ein Tauschmedium basierend auf der Nutzung intensiver Rechenleistung der Miner für die Erstellung und dem Hinzufügen von Blöcken und Transaktionen zur Blockchain. Bitcoin unterliegt teilweise großen Schwankungen im Marktpreis. Abb. 12 zeigt einen Wertverlust in USD von ca. 20% innerhalb von 28 Tagen.

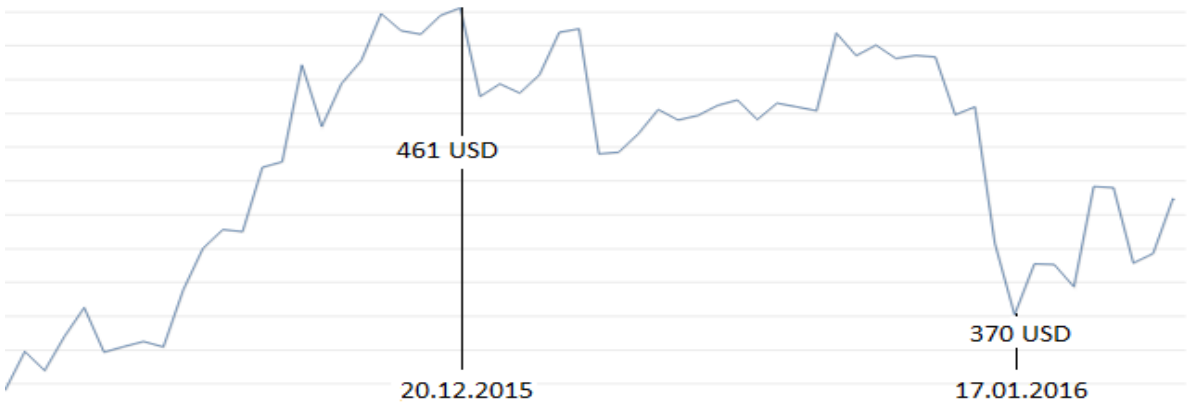

ABB. 12: SCHWANKUNGEN IM BITCOIN HANDELSPREIS [BL16B]

Die Preisinstabilität aus Abb. 12 ist ein Nachteil für die Nutzung von BTC als Wertanlage oder als Rechnungsgröße im E-Commerce z.B. in Onlineshops. Vor- und Nachteile für die Nutzung der Bitcoin Währung als Tauschmedium und Wertanlage sind in Tab. 10 aufgeführt.

	Pro	Contra
Bitcoin als Tauschmedium	○ birgt keine Gefahr von Sicherheitsverletzungen wenn private Schlüssel geheim bleiben ○ erhöht Anonymität durch pseudonyme Adressen ○ besitzt geringe Transaktionsgebühren ○ hat keine Grundgebühr ○ unterstützt Mikrozahlungen ○ unterstützt Übertragung und Speichern digitaler Assets ○ schützt vor Betrug in Rücklastschriften ○ überweist umgehend ○ ist ein Push Zahlungssystem gesteuert vom Benutzer	○ ist keine „echte“ Währung basierend auf dem Umsatz eines Devisenmarkt ○ besitzt keine „Netzwerk“-Externalitäten im Wettbewerb mit anderen Währungen ○ bietet keine Kreditoption ○ besitzt als Sachanlage hohe Kosten für Berichtswesen und Regelbefolgung ○ hat eine Transaktionsbestätigung von mehreren Minuten ○ hat einen Skalierungsengpass in der Speicherung der Blockchain und den UTXO im RAM ○ kann in seiner Technologie von Unternehmen und Regierungen übernommen werden ○ wirkt abschreckend auf Benutzer so dass allein durch Nutzung in Unternehmen keine signifikante Kosteneinsparung stattfindet
Bitcoin als Wertanlage	○ vermeidet Beschlagnahmung, Kapitalkontrolle und unangemessene Versteuerung ○ hat keine Lagerungskosten ○ ist einfach im Transport ○ hat eine maximale Anzahl an existierenden BTC fixiert per Algorithmus ○ ist kryptografisch gesichert ○ bietet automatische Buchführung ○ ist deflationär	○ basiert auf Open Source d.h. einfach und legal zu replizieren und zu ersetzen ○ ist schwankungsanfällig ○ bietet keine Kontrolle über die Geldmenge und -Umlauf ○ ist versteuerungspflichtig ○ ist kein gesetzliches Zahlungsmittel ○ kann von Regierungen verboten werden ○ hat keinen physischen Rückhalt und dadurch keinen intrinsischen Wert ○ hat hohe Produktionskosten ○ hat keine Einlagensicherung

TAB. 10: VOR- UND NACHTEILE VON BITCOIN ALS WÄHRUNG [FR15]

Vorteile und Nachteile von Bitcoin als Währung aus Tab. 10 sind abhängig vom Kontext, der jeweiligen Sichtweise und in Relation zu betrachten. Es ist einerseits ein Vorteil, dass die maximale Anzahl an BTC limitiert und die Währung deflationär ist, andererseits ein Nachteil dass keine Kontrollmöglichkeit über Geldmenge und Umlauf besteht. Das Problem der

Preisinstabilität kann zum Beispiel über die Sicherung von BTC zu einem bestimmten Wechselkurs gelöst werden. Uphold [Up16] ist ein Anbieter der dieses Verfahren unterstützt und darüber hinaus im Gegensatz zu einer klassischen Bank alle Guthaben vorhält. Nachteil hier wiederum ist das Vertrauen in eine dritte Partei als Vermittler entgegen der Grundidee von Bitcoin. Ein Grund für das Problem der Preisinstabilität liegt darin dass es in den Ländern keine, eindeutige oder einheitliche Rechtsprechung gibt. Das Konzept lässt sich nicht in vorhandene Strukturen eingliedern. BTC und andere kryptografische Währungen sind illegal, als Sachanlage versteuerungspflichtig oder als Währung anerkannt. [Th15] Innerhalb der im Folgenden aufgeführten Anwendungsbereiche und Geschäftsfelder nach Franco [Fr15] und Swan [Sw15] für die Blockchain 1.0 und darüber hinaus sollte die Rechtsprechung des jeweiligen Landes bzw. der Länder berücksichtigt werden.

- Geldtransfer
- Währungsumtausch und ATM
- Web Wallets
- Zahlungs- und Treuhanddienste
- Mining
- Softwareentwicklung

Eine BTC Überweisung hat im Vergleich zu den durchschnittlichen Gebühren weltweit von 5,91% - 7,52% im dritten Quartal 2015 [Th16] sehr niedrige Transaktionskosten. Im Bitcoin Core Client sind minimale Transaktionskosten implementiert. Transaktionen unter `10.000 Bytes` mit mindestens 0,01 BTC als Summe aller Ausgänge und mit einer hohen Priorität können ohne Gebühren versendet werden. Die Priorität berechnet sich aus dem summierten Wert der Eingänge, Alter der Transaktion und Größe [Bi16h]. Die Gebühr lag am 26.01.2016 bei 0,1 USD Cent (378 Satoshis) für die Übernahme einer Transaktion in den nächsten Block [Bi16f]. Diese Gebühren gehen an die Miner und beinhalten keine Steuern oder sonstige Gebühren z.B. für die Umlage von Kosten für Regulation und Einhaltung wie bei Banken und Kreditinstituten getrieben durch Rechtsprechungen der Länder.

BitcoinAverage [Bi16g] gibt eine Übersicht zu Währungskursen, Markt- und Tauschvolumen basierend auf einem Preisindex über die größten Anbieter für Bitcoin Währungsumtausch im E-Commerce. Es wird ein API (Application Programming Interface) bereitgestellt welche z.B. in Web Wallets genutzt wird. Am 21.05.2016 gab es weltweit 667 bei Coin ATM Radar [Co16a] installierte ATMs (Automated Teller Machine) für den Tausch von BTC und anderer kryptografischer Währung. Analog bietet CoinMap [Co16c] eine historisierte Karte mit 7789 Händlern. Web Wallets werden angeboten um eine nutzerfreundliche Plattform für die Nutzung von Bitcoin analog zum Onlinebanking zu geben. Eine Wallet Software sichert die öffentlichen Schlüssel bzw. die Bitcoin Adressen und die geheimen privaten Schlüssel. Web Wallets bieten weitere Funktionen wie z.B. Betrugserkennung oder Treuhanddienste. Sie verwenden entweder eine einfache Validierung über SPV oder eine vollständige Validierung über die komplette Blockchain. Bitcoin [Bi16d] beschreibt in seiner Anleitung „Wie man Bitcoin verwendet“ eine Übersicht unterschiedlicher Wallets mit Screenshots und

Funktionen. Es gibt neben Web Wallets auch Hardware Wallets, Desktop und Mobile Wallets oder Paper Wallets, die oftmals von ATMs verwendet werden.

Zahlungs- und Treuhanddienste sind teilweise in Web Wallets als Funktion integriert. Einer der größten Zahlungsanbieter ist BitPay [Bi16a] mit über 40 E-Commerce Integrationen und APIs sowie über 150 Währungen. Hier wird ein Payment Protokoll nach BIP70-72 und URI (Unified Resource Identifier) nach BIP-21 unterstützt sofern die Wallet Software diese auch unterstützt. BitPay berechnet keine Grundgebühr und besitzt für eine unlimitierte Transaktionsanzahl eine Gebühr von 1% pro Transaktion. PayPal [Pa16] liegt im Vergleich bei 1,5% + 0,35€ für eine inländische Transaktion bei einem monatlichem Umsatz über 25.000€ und 10% + 0,10€ für Mikrozahlungen. Treuhanddienste und Konfliktbehandlung können über Multi-Signatur Wallets abgebildet werden über den gleichnamigen Transaktionstyp. BitPay hat mit CoPay eine eigene Multi-Signatur Wallet entwickelt.

Mining ist gemessen am Umsatz der größte Geschäftsbereich in Bitcoin bzw. der Blockchain 1.0. Am 27.01.2016 betrug der Tagesumsatz aus 3.850 erstellten BTC und den erhaltenen Transaktionsgebühren bezogen auf den Marktpreis von 394,12 USD pro BTC insgesamt 1.537.171,65 USD [Bl16b]. Die Mining Industrie beinhaltet Hersteller für spezielle ASIC (Application Specific Integrated Circuit) Prozessoren, Datencentern, Miner, Operatoren für Mining Pools und dem Hosted Mining oder auch Cloud Mining mit MaaS (Mining as a Service). [Bi16b], [Fr15]

Im nächsten Kapitel wird mit der Version 2.0 die Dezentralisierung von Märkten als nächste Stufe der Blockchain Entwicklung beschrieben, die Version 1.0 - Dezentralisierung von Geld und Zahlung - erweitert und über einfache Zahlungen mit der Währung hinausgeht.

3.6.2 Blockchain 2.0 - Smart Properties und Contracts

Mit Bitcoin wurde in Nakamoto [Na08] die Basis für alle Blockchain Versionen geschaffen und hier als kryptografische Währung und Dezentralisierung des Zahlungsverkehr im E-Commerce die Blockchain 1.0 beschrieben. Bitcoins Design unterstützt eine Vielfalt an möglichen Transaktionstypen und zukünftigen Anwendungsbereichen [Na10]. Die Blockchain 2.0 wird von Swan [Sw15] als Dezentralisierung von Märkten durch den Transfer digitaler Assets über die Blockchain beschrieben. Blockchain 2.0 umfasst weitere Protokolle, Smart Properties, Smart Contracts, dezentrale Anwendungen und autonome Agenten.

In der Version 2.0 ist die Blockchain neben der Währung Bitcoin ein dezentralisiertes Register für Transfer und Besitznachweis digitaler Assets. Ein Asset wird über den privaten Schlüssel des pseudonymen Eigentümers kontrolliert und in der Blockchain gespeichert. Eine Transaktion für den Verkauf eines Assets wird von Käufer und Verkäufer signiert und hat zwei Ein- und Ausgänge. Das Asset des Verkäufers und der Verkaufspreis des Käufers gehen als Eingang in die Transaktion, die als Ausgang den Verkaufspreis an den Verkäufer und das Asset an den Käufer transferiert. Ein Smart Property ist ein Objekt dessen Eigentum über die Blockchain kontrolliert wird und direkten Zugang zur Blockchain besitzt. Ändert sich der

Eigentümer kann das entsprechend enkodierte Smart Property als digitales Asset dies über die Blockchain verifizieren und den öffentlichen Schlüssel des neuen Eigentümers übernehmen. So können sowohl physische Objekte wie z.B. Automobile als auch nicht-physische Objekte wie z.B. eine Wählerstimme als Smart Property enkodiert und als digitales Asset in der Blockchain repräsentiert werden. [Fr15] Die Ledra Capital LLC [Le14] führt in ihrem Blog eine Liste möglicher Anwendungsbereiche für Smart Properties in der Blockchain und umfassen alle Eigenschaften einer Smart City aus Kapitel 2.4 durch die Bereiche Wirtschaft, Menschen, Regierung, Mobilität, Umwelt und Wohnen.

Durch IoT werden Billionen Geräte mit dem Internet verbunden, die als Smart Properties in der Blockchain enkodiert werden können. Durch eingebettete Technologien wie Sensoren, QR Codes, NFC Tags, iBeacons und Wi-Fi kann echtzeitnah zugegriffen werden. Die Blockchain bietet eine neuartige Möglichkeit der anonymisierten Identitätsprüfung und sicheren Zugriff. Smart Properties und Contracts basieren darauf, dass sich zwei oder mehr Parteien nicht kennen oder vertrauen müssen. Smart Contracts werden über Transaktionen mit umfangreichen Befehlen im Kontext von Smart Properties abgebildet. Sie unterscheiden sich von herkömmlichen Verträgen in der autonomen Ausführung eines definierten Codes in einem dezentralisierten Netzwerk. [Sw15] Das Konzept stammt aus 1997 bzw. dem Artikel „Formalizing and Securing Relationships on Public Networks“ von Nick Szabo [Sz97]. Ein Beispiel ist der Zutritt von Eigentümer und Kreditinstitut eines finanzierten Automobils als enkodiertes Smart Property und besteht aus folgenden Schritten und Eigenschaften.

1. Eine Sperre die der Eigentümer aufheben kann aber dritte ausschließt.
2. Eine „Hintertür“ für das Kreditinstitut ist per Default deaktiviert.
3. Die „Hintertür“ für das Kreditinstitut wird aktiviert falls eine Zahlung für eine bestimmte Zeit nicht erfolgt ist.
4. Die „Hintertür“ für das Kreditinstitut wird gelöscht wenn die finale Zahlung erfolgt ist.

Bitcoin [Bi16h], [Bi16e] beschreibt weitere Beispiele für Smart Contracts und skizziert deren Ablauf. Mit Smart Contracts können Kautionen, Treuhanddienste und Streitschlichtung, Wettbüros, Auktionen, Tauschbörsen oder Crowdfunding abgebildet werden. Das größte Crowdfunding in Bitcoin und das fünftgrößte Crowdfunding insgesamt hat durch das Projekt Ethereum mit 31.591 BTC zum damaligen Wert von 18.439.086 USD für 42 Tage stattgefunden. Hierbei wurden von 60.102.216 ETH als eigene kryptografische Währung bzw. Tokens zur Nutzung der Plattform verkauft. [Wi16a] Ein Smart Contract für Crowdfunding hat folgenden Ablauf [Bi16e]:

1. Ein Entrepreneur erstellt eine öffentliche Adresse und publiziert diese für das Crowdfunding unter Angabe der Gesamtsumme z.B. 1.000 BTC
2. Teilnehmer erstellen invalide Transaktionen mit gespendeten Beträgen als Eingang und als Ausgang die Gesamtsumme signiert durch `SIGHASH_ALL|SIGHASH_ANYONECANPAY` an die öffentliche Adresse des Entrepreneur

3. Die Transaktion wird auf einen Server hochgeladen, wo sie gespeichert und die Höhe der gespendeten Bitcoins aktualisiert wird
4. Wenn 1.000 BTC erreicht wurden erstellt der Server eine Transaktion mit den Eingängen der gespeicherten Transaktionen und einem Ausgang mit dieser Gesamtsumme an die öffentliche Adresse des Entrepreneur.
5. Diese Transaktion wird dann über das Netzwerk verteilt und validiert

Mit dem Signatur-Hashtyp `SIGHASH_ALL` werden alle Ein- und Ausgänge signiert und somit vor Änderungen geschützt. Über die Modifikation `SIGHASH_ANYONECANPAY` wird nur dieser eine Eingang signiert, so dass weitere hinzugefügt oder entfernt werden können. Ein weiteres Beispiel für Smart Contracts sind Mikrozahlungen z.B. für die Moderation von Forenbeiträgen oder für die Datennutzung eines Wi-Fi Hotspots. Hier werden „off-chain" Transaktionen genutzt, die analog zum Beispiel Crowdfunding erst im letzten Schritt in der Blockchain veröffentlicht werden. [Bi16h]

Smart Properties und Contracts erfordern das Speichern weiterer Daten in der Blockchain. Hierfür gibt es in Bitcoin mehrere Möglichkeiten. In Kapitel 3.5 wurden Datentransaktionen und Multi-Signatur Transaktionen beschrieben. Datentransaktionen mit dem Befehl `OP_RETURN` können `40 Byte` Daten speichern. Dadurch dass der Ausgang dieser Transaktion nicht als Eingang wieder genutzt werden kann, können sie von den Knoten aus dem UTXO Cache entfernt werden. Eine weitere Möglichkeit sind Fake Adressen über die aufgrund des Hashverfahren eine beliebe Datenmenge gespeichert werden kann. Da es keinen privaten Schlüssel zur Fake Adresse gibt sind die im Ausgang transferierten Beträge (546 Satoshis ist der Mindestausgang einer Transaktion) verloren und verbleiben im UTXO Cache. Mit Multi-Signatur Transaktionen können `N-1` Fake Adressen verwendet werden und eine reale Adresse die den Ausgang entsperren kann. Neben diesen Möglichkeiten wurden mit Meta-Coins Blockchain 2.0 Protokolle und Netzwerke entwickelt, die digitale Assets in der Bitcoin oder einer eigenen Blockchain speichern können und neue Anwendungen wie autonome Agenten ermöglichen. Ethereum, Litecoin und Ripple besitzen mit ihrer kryptografischen Währung als Meta-Coins nach Crypto-Currency Market Capitalizations [Co16d] das größte Marktkapital direkt hinter Bitcoin. [Fr15]

Piasecki [Pi16] führt eine umfangreiche Liste und Vergleich von Blockchain 2.0 Protokollen teilweise verifiziert durch Projektbeteiligte. Hier werden wichtige Informationen wie eigene Blockchain und Wallet, Netzwerkgeschwindigkeit, Transaktionsgebühren, Blockerstellung und Mining gegenüber gestellt. Ethereum wird ausführlich in Kapitel 4 beschrieben. Die Entwicklung von Wallet Software und Blockchain APIs wie im Bitcoin Block Explorer [Bl16b] und BitcoinAverage [Bi16g] ist eine wichtige Anwendungskategorie in der Blockchain 2.0. Neben Speicher-, Nachrichten- und Dateidiensten, Mobilen Zahlungen, Wallet Interaktionen, Identitätsprüfung und Reputation beschreibt Swan [Sw15] die Verknüpfung zu IoT- und M2M-Kommunikation. Zum Beispiel kann eine Smartwatch mit Smart City Verkehrssensoren kommunizieren und über einen Smart Contract eine Fahrspur reservieren und bezahlen. Autonome Agenten laufen ohne menschliche Unterstützung als Computerprogramm mit

eigener Existenz und werden häufig auch als DAC (Decentralized Autonomous Corporation, als DAO (Decentralized Autonomous Organization), als DAS (Decentralized Autonomous Society) oder einfach als Dapp (Decentralized application) bezeichnet. Autonome Agenten können Smart Contracts eingehen, Assets und Tokens empfangen und senden. Sie führen Aufgaben aus, die den Menschen unterstützen. [Fr15] In der Definition von Johnston et al. [Jo15] muss eine Dapp folgende Kriterien erfüllen:

1. *Die Anwendung muss Open Source sein und autonom laufen ohne dass jemand die Mehrheit der Tokens besitzt. Änderungen im Protokoll aus Verbesserungsvorschlägen und Marktfeedback kann nur durch die Mehrheit der Nutzer entschieden werden.*

2. *Die Anwendungsdaten und Befehlseingaben müssen kryptografisch in einer öffentlichen dezentralen Blockchain gespeichert werden um einen „Single Point of Failure" zu vermeiden.*

3. *Die Anwendung muss kryptografische Token (BTC oder native Token) benutzen, die notwendig für den Zugriff auf die Anwendung sind. Jeder Beitrag von Miner soll mit Tokens der Anwendung belohnt werden.*

4. *Die Anwendung muss Tokens gemäß standardisierter, kryptografischer Algorithmen als Beweis für die Teilnahme wertschaffender Knoten (Bitcoin nutzt PoW).*

Bitcoin ist nach Johnston et al. [Jo15] selbst eine Dapp. Weitere Beispiele für Dapps werden von Swan [Sw15] als Äquivalent zur entsprechenden zentralisierten Lösung dargestellt. Die Dapp Storj ist beispielsweise die dezentrale Variante der Dropbox. Daneben gibt es u.a. Dapps für UBER, Twitter oder Facebook. Im nächsten Kapitel wird die Blockchain mit vielen Beispielen über Währungen, Finanzen und Wirtschaft hinaus betrachtet. Grundlage hierfür sind Smart Properties und Contracts.

3.6.3 Blockchain 3.0 - Anwendungen und App-Coins

In den beiden vorangegangenen Kapiteln wurde mit der Blockchain 1.0 die Dezentralisierung von Geld und Zahlungen und mit der Blockchain 2.0 die Dezentralisierung von Märkten vorgestellt. Dieses Kapitel beschreibt mit der Blockchain 3.0 weiterführende Anwendungen und ihren App-Coins mit Beispielen in den Bereichen Regierung, Gesundheit, Wissenschaft, Bildung, Kultur und Kunst.

Analog zum Internet im 20. Jahrhundert ist die Blockchain ein neues Paradigma. Die Fähigkeit für das Verständnis seiner Konzepte im 21. Jahrhundert ist die Basis für seine Erweiterbarkeit und seinen Erfolg. Alle Aktivitäten und Quantitäten können in der Blockchain organisiert und administriert werden. Diese können zum Beispiel mit Big Data Technologien die Automatisierung von Prognosen durch Smart Contracts vereinfachen. [Sw15] In Tab. 11 werden einige Anwendungen, Dienste und Plattformen in den Bereichen Regierung, Kultur und Kunst basierend auf der Blockchain aufgelistet und beschrieben.

Anwendung	Beschreibung
gitchain.org	Dezentrales P2P-Netzwerk für Git basierend auf Bitcoin, Namecoin, DHT (Distributed Hash Tables) wurde mangels Crowdfunding bei Kickstarter 2014 eingestellt.
namecoin.info	Alt-Coin für dezentrale Open Source Registrierung und Transfer von Informationen sowie DNS (Domain Name System) mit Top-Level-Domain .bit erhältlich unter dotbit.me.
openlibernet.org	Dezentrales, globales, sicheres, anonymes, robustes und neutrales Kommunikationsnetzwerk inspiriert von Bitcoin.
onename.com	Dienst für die Registrierung, Verifizierung und Authentifizierung digitaler Identität mit einer Blockchain ID. Ist von Namecoin zu Bitcoin gewechselt.
coinapult.com	Plattform für das Kaufen, Verkaufen und Empfangen von BTC auch über einfache und kostengünstige SMS und E-Mail als Zugang für Orte ohne oder geringe Anzahl an Banken und eingeschränkter Technologie.
seansoutpost.com	Beispiel für eine Spendenaktion „Satoshis Forest“ mit BTC.
proofofexistence.com	Dienst für das Hashen einer Datei und Übernahme in die Blockchain als Existenz- und Besitznachweis mit Zeitstempel und Integritätsprüfung.
virtual-notary.org	Virtueller Notar der neben Dateien vieles mehr wie z.B. Grundbesitz oder Wechselkurse in die Bitcoin Blockchain übernimmt und dafür Zertifikate ausstellt und prüft.
monegraph.com	Dezentrale Plattform für die Lizenzierung, Copyright, Veröffentlichung und Vertrieb digitaler Inhalte (aktuell nur Bildformate mit max. `100 MByte`) mit Smart Contracts.
ascribe.io	Wie Monegraph nur für alle Formate bis max. `25 GByte`.
factom.org	System für Datenmanagement mit Batch Transaktionen in der Bitcoin Blockchain für große Datenmengen in Unternehmen und Regierungen.
bitcongress.org	Dezentrale Abstimmungsplattform für die unmittelbare Abstimmung und Gesetzgebung sowie Debatten, Quorum, Kommentare mit Wählern und Zuordnung von Abgaben, Kapital und Budget für Projekte mit BTC.
bitcoinhivemind.com	P2P Orakelprotokoll für Prognosemärkte nimmt Informationen in Sidechain auf und nutzt CashCoins und VoteCoins in Smart Contracts.

TAB. 11: BLOCKCHAIN 3.0 FÜR REGIERUNG, KULTUR UND KUNST [SW15]

Die Anwendung $gitchain$ aus Tab. 11 ist ein Beispiel für das Tracking sämtlicher Aktivitäten der Quellcodeverwaltung Git in der Blockchain. Ein weiteres Beispiel ist die Aufnahme von Trackingdaten des täglichen Lebens (Fitbit, iHealth etc.) in die Blockchain zur Optimierung persönlicher Assistenten und Empfehlungsmaschinen. Die Blockchain ist ein Modell für eine zensurfreie Organisation. $namecoin$ aus Tab. 11 mit dezentralem DNS unterliegt keiner zentralen Kontrolle oder Autorität im Gegensatz zu ICANN (Internet Corporation for Assigned

Names and Numbers), Wikipedia oder Wikileaks. *openlibernet* aus Tab. 11 orientiert sich als Netzwerk an der ursprünglichen Netzneutralität des Internet durch die Blockchain. Die Anwendung $onename$ aus Tab. 11 nutzt ebenfalls DNS Funktionalität für die Identifizierung und Verifizierung von öffentlichen Bitcoin Adressen über eine nutzerfreundliche ID z.B. „blakeberg81“ und kann wie die Anmeldung über Google+ verwendet werden. Die Plattform $coinapult$ aus Tab. 11 zielt auf die von Kultur, Religion, Sprache und Herkunft unabhängige Nutzung von Bitcoin per SMS ohne Internetzugang.

In der Blockchain sind Smart Properties enkodiert um deren Eigentum und Existenz zu bestätigen. Mit den Diensten $proofofexistence$ und $virtual\text{-}notary$ aus Tab. 11 können Dateien als Hash in die Blockchain übernommen werden. Zum Schutz des geistigen Eigentums bieten die Plattformen $monegraph$ und $ascribe$ aus Tab. 11 die Möglichkeit digitale Inhalte basierend auf Smart Contracts mit Copyright über die Blockchain zu lizensieren und zu vertreiben. Durch die Nutzung von Mikrozahlungen über Batch Transaktionen kann die Übernahme von digitalen Inhalten als Hash in die Blockchain zum Schutz geistigen Eigentums während der Veröffentlichung automatisiert werden. Die Plattform $factom$ nutzt diese Form von Mikrozahlungen über Batch Transaktionen für die Übernahme von großen Datenmengen in Unternehmen und Behörden. Mit der Blockchain können günstige, effiziente und personalisierte Lösungen für Behörden und Regierungen entgegen dem Prinzip „one-size-fit-all“ erstellt werden. Swan [Sw15] beschreibt beispielsweise einen RoadCoin in Smart Cities der vorbeifahrende Fahrzeuge für verlorene Lebensqualität durch Verzögerungen von Baustellen entschädigt. Über ein AccidentCoin in Smart Cities entschädigen Unfallbeteiligte vorbeifahrende Fahrzeuge und können dies später mit der Versicherung abrechnen. Weiter können dezentrale Dienste Hochzeiten, Geburten, Todesfälle, Landbesitz oder Ausweise in der Blockchain registrieren.

Die Blockchain basiert auf dem Konsens statt auf konzentrierten Parteien, auf denen Regierungen aufbauen. Sie ist in seiner Natur demokratisch. Ein Beispiel hierfür ist die dezentralisierte Plattform $bitcongress$ aus Tab. 11 für Abstimmungen und Wahlen. Mit der Blockchain können auch andere Systeme wie Delegierungsdemokratie oder die zufällige Auswahl von Wählern in einer Stichprobe umgesetzt werden. Eine weitere Möglichkeit ist die Einbeziehung von Prognosemärkten nach dem Prinzip „vote-on-values-bet-on-believes“ welches von dem Protokoll $bitcoinhivemind$ aus Tab. 11 umgesetzt wird. Ein Verständnis und die Gewöhnung der Konzepte der Blockchain Technologie analog zur Einführung des Paradigma der sozialen Netzwerke ist für den Erfolg in allen Bereichen entscheidend. In Tab. 12 werden weitere Anwendungen, Dienste und Plattformen der Bereiche Gesundheit, Bildung und Wissenschaft basierend aufgelistet und beschrieben.

Anwendung	Beschreibung
gridcoin.us	Kryptografische Währung (GRC) in wissenschaftlichen Projekten der BOINC (Berkeley Open Infrastructure for Network Computing) wie z.B. SETI@home als Vergütung für Rechenleistung genutzt. Projekte können als Non-Profit, Profit oder auch DAC erstellt werden.

foldingcoin.net	Kryptografische Währung (FLDC) wie Gridcoin nur von der Stanford Universität für Folding@home basiert auf Counterparty (eine Plattform auf dem Bitcoin Netzwerk).
primecoin.io	Kryptografische Währung (XPM) mit PoW über Primzahlenermittlung.
zennet.sc	Zennet (XenCoin) ist ein öffentlicher, verteilter und dezentralisierter Supercomputer mit eigener Blockchain und basiert auf Mikrozahlungen. Jeder kann Rechenleistung bereitstellen mieten.
genecoin.me	Dienst für die Sequenzierung von DNA und Übernahme in die Bitcoin Blockchain für die Erstellung einer eigenen kryptografischen Währung (Alt-Coin). Verwendet DBC (Decentralized Blockchain Crawler) um neue Blockchains zu finden und die DNA auch hier zu speichern.
dna-bits.com	Blockchain für den anonymen Austausch genetischer und klinischer Daten unter Berücksichtigung des HIPAA Standards (Health Insurance Portability and Accountability Act) und Big Data.
LearnCoin, SmallChange	Beispielhafte Implementierung und Anleitung für die Erstellung einer eigenen Kryptowährung auf Basis von Litecoin (LTC).

TAB. 12: BLOCKCHAIN 3.0 FÜR GESUNDHEIT, BILDUNG UND WISSENSCHAFT [SW15]

In der Wissenschaft ist ein Anwendungsbereich das Bereitstellen von Rechenleistung in einem P2P-Netzwerk für Projekte mit rechenintensiven Berechnungen wie SETI@home (Search for Extra terrestrial Intelligence) oder Folding@home (protein folding simulation). $gridcoin$ und $foldingcoin$ aus Tab. 12 benutzen das Mining der Blockchain für diese Berechnungen und vergüten mit ihren App-Coins als kryptografischen Währung. Das Prinzip die Arbeit der Miner wissenschaftlich nutzbar zu machen greift $primecoin$ aus Tab. 12 auf durch die Berechnung von Primzahlen statt einer Hashberechnung. Über $zennet$ aus Tab. 12 kann ebenfalls Rechenleistung bereitgestellt werden, die hier analog zu AWS (Amazon Web Services) aber auch angemietet werden kann. Die Vermittlung von VMs (Virtual Machine) dessen Nutzen und Abrechnung findet über die Bitcoin Blockchain mit Mikrozahlungen statt.

Mit $genecoin$ aus Tab. 12 ist es möglich DNA mit ca. $750\ \mathrm{MByte}$ in der Blockchain von Bitcoin und anderen zu speichern und hieraus Alt-Coins zu erstellen. Im Gesundheitsbereich kann eine Datenbank mit personalisierten DNA-Profilen den Kern einer neuartigen Präventivmedizin darstellen und Sachkundigen, Interessierten und Patienten dienen. Die Blockchain ist ein universelles Model für die transnationale Aufzeichnung und Speicherung von Daten aller Art sowie dessen Zugriff gesteuert über App-Coins. Die Speicherung von großen Datenmengen findet dabei sowohl „off-chain“ (Daten) als auch „on-chain“ (Zeiger auf Lokalisation der Daten) statt. Hierüber können z.B. Virus-, Gift- und Samenbanken dezentralisiert werden. Das Projekt $dna\text{-}bits$ aus Tab. 12 nutzt die Blockchain zum anonymen Austausch genetischer und klinischer Daten für Analyse, Diagnostik, Forschung und Abrechnung. Trackingdaten im Bereich Gesundheit z.B. von Wearables wie FitBit oder Smartwatch können korreliert werden. Medizinische Dienste können mit der Blockchain nach dem UBER-Prinzip angeboten und über Healthcoins vergütet werden. Katastrophen

und Krankheitsausbrüche können getrackt werden, so dass Spenden siehe $seansoutpost$ aus Tab. 11 mit Bitcoin direkter und schneller überwiesen werden können.

Im Bildungsbereich können Smart Contracts eingesetzt werden um Lernergebnisse zu verifizieren oder Interaktionen zwischen Lernenden und Lehrenden zu koordinieren. Als Währung für das Erreichen von Lernzielen können LearnCoins wie z.B. in MOOCs (Massiv Open Online Course) verwendet werden. Hierüber können auch ECTS (European Credit Transfer System) Leistungspunkte im Studium abgebildet werden. Ein Smart Contract steuert bspw. die Zulassung zu Aufbaukursen und Prüfungen. Jede Bildungseinrichtung kann ihre eigenen App-Coins erstellen siehe $LearnCoin$ und $SmallChange$ aus Tab. 12. Im akademischen Bereich sind Publikationen und Interaktionen wie Reviews ein weiterer Anwendungsbereich für die Blockchain. Ein App-Coin für wissenschaftliche Beiträge von der Veröffentlichung von Artikeln, Replikation von Versuchsergebnissen bis zu einfachen Kommentaren kann hierbei als Zahlungsmittel analog zu digitalen Bibliotheken wie ACM eingesetzt werden.

Dies sind nur einige Anwendungsbereiche für die Blockchain. In den Bereichen Smart Cities, IoT und M2M ist eine große Herausforderung, dass viele Geräte und Sensoren viele Transaktionen in kurzen Intervallen melden z.B. Mautstellen auf Autobahnen. Generell können verschiedene Anwendungsklassen eigene Blockchains besitzen. Bitcoin mit BTC hat hier mit ca. 80,9% vom 03.06.2016 den größten Marktanteil nach Crypto-Currency Market Capitalizations [Co16d] gefolgt von der Währung ETH von Ethereum. Davon werden ca. 30% von den 500 reichsten Adressen laut BitCoinRichList [Bi16c] gehalten und auch die Verteilung der Miner ist mit ca. 50% auf drei Pools und 71% auf fünf Pools ist ungleichmäßig [Bi16b]. Die Themengebiete Blockchain Performance, Sicherheit und Mining erfordern eine gesonderte Betrachtung und werden im Rahmen dieser Arbeit nicht im Detail behandelt.

Im nächsten Kapitel wird mit Ethereum eine Plattform für die Ausführung dezentralisierter Anwendung im vorgestellt und mit Bitcoin vergleichen. Die vereinfachte Erstellung von Smart Contracts, Datenspeicherung in der eigenen Blockchain und Blockerstellungszeiten von 14 Sekunden sind dabei wichtige Aspekte für sämtliche Anwendungsbereiche. Mit der Ethereum Plattform soll anschließend ein PoC für eine Dapp in der Blockchain umgesetzt werden. Die Blockchain Technologie, die Plattform Ethereum und die erstellt Dapp werden später im Kontext Smart Cities diskutiert.

4 Ethereum als Plattform

In diesem Kapitel wird Ethereum als Plattform für dezentralisierte Anwendungen basierend auf Smart Contracts im Kontext Blockchain 2.0 und Meta-Coins vorgestellt. Im ersten Teil wird Ethereum mit seinen Versionen historisch eingeordnet. Anschließend werden Struktur und Funktion von Ethereum mit eigener Blockchain im Vergleich zu Bitcoin beschrieben. Daraufhin folgt eine Übersicht von Smart Contracts, Dapps und zuletzt eine Beschreibung von Clients für Implementierung und Ausführung in den späteren Kapiteln 5 und 6.

4.1 Historie und Entwicklung

Vitalik Buterin startet mit Bitcoin im Jahr 2011 und wurde im September 2011 Mitbegründer und Chefautor des Bitcoin Magazins. Im November 2013 hatte Buterin mit Ethereum eine Vision eines dezentralisierten Weltcomputers und ein Konzept einer kryptografischen Plattform mit integrierter Programmiersprache für Contracts mit komplexen, arithmetischen Formeln und verschachtelten, bedingten Anweisungen, Schleifen und Verzweigungen. Einen Monat später veröffentlichte Buterin hierzu ein Whitepaper [Bu14b]. Im Februar 2014 veröffentlichte Mitbegründer Dr. Gavin Wood ein Yellowpaper [Wo14] mit der formalen Beschreibung der virtuellen und Turing-vollständigen Maschine. Die Entwicklung war zu diesem Zeitpunkt so weit fortgeschritten, dass Anfang Februar das erste Testnetz basierend auf dem im Yellowpaper beschrieben Protokoll und zwei vollständig kompatiblen Clients (C++ und Go) bereitgestellt wurde. Im Juni 2014 wurde die Ethereum Fundation in der Schweiz als gemeinnützige Stiftung registriert. Ihre Aufgabe ist das Management und die Organisation des Kapitals von 31.391 BTC aus dem 42-tägigen Crowdfunding. Die Stiftung besitzt eine GmbH zum Zweck von Entwicklung, Vermarktung und Lizenzierung der Software. Hieran angebunden ist das DEV Team welches mit 76,5% den größten Anteil des Kapitals erhielt. 13,5% erhielten Miner und 10% die Cryptocurrency Research Group (CCRG). [Bu14a], [Et14]

Nach Crypto-Currency Market Capitalizations [Co16d] ist Ethereum mit ETH die zweitstärkste Kryptowährung. Im Blockchain Explorer EtherScan [Et16i] gibt es aktuelle Statistiken zu Accounts, Transaktionen, Blöcken und Mining. Am 17.02.2016 betrug der Wert von 77.403.816,26 ETH umgerechnet 283.297.736 USD. Davon wurden 5.393.762,56 ETH durch Mining generiert und der Rest stammt aus dem Crowdfunding (60M) und anderer Quellen (12M). Die Anzahl an Contracts betrug 9.529 mit 2.959.333,43 ETH. Die Verteilung entspricht ca. 3,82% ETH gespeichert in diesen Contracts und ca. 96,18% in 61.092 Adressen. [Et16i] Die kleinste Einheit ist Wei mit 10^{-18} ETH. Daneben gibt es Finney (10^{-15}), Szabo (10^{-12}) sowie GWei (10^{-9}), MWei (10^{-6}) und KWei (10^{-3}) basierend auf bekannten Persönlichkeiten im Bereich kryptografischer Währungen.

Die erste Version (1.0) mit dem Namen Frontier wurde am 30.07.2015 veröffentlicht und beinhaltet Clients mit CLI (Command Line Interface) für das Ethereum Netzwerk mit dem

Notwendigsten wie dem Mining und dem Veröffentlichen und Ausführen von Contracts. Ursprünglich sollten Miner mit fünf ETH pro Block 10% der geplanten Vergütung erhalten. Der Grund hierfür liegt darin, dass die Frontier Version von Ethereum selbst als Test und Sicherheitsaudit unter realen Bedingungen (Wettbewerb, Hacker, Miner) verstanden wurde [Bv15]. Auf der Frontier Homepage wurde Ethereum als „A ~~SAFE~~ DECENTRALIZED SOFTWARE PLATFORM“ bezeichnet. Zudem wurde mit den Canary Contracts ein Mechanismus eingeführt, indem jeder Client alle vier Blöcke eine Prüfung über vier Contracts ausführt. Diese geben entweder eine 0 oder eine 1 (z.B. im Falle eines Fork) zurück. Wenn zwei dieser Contracts eine 1 zurückgeben wird das Mining gestoppt und eine Meldung für ein Clientupdate angezeigt. Daneben gab es verschiedene Schutzmaßnahmen wie der täglichen Erstellung eines Checkpoints bei ca. 12 Stunden Versatz zur Blockchain. [Gu15], [Tu15b]

Die derzeit aktuelle Version Homestead ist gerichtet an Dapp-Entwickler und war geplant für Anfang 2016 [Ge15]. Eine Veröffentlichung fand nach positiver Bewertung der vollständigen Stabilität von Frontier durch Kernentwickler und Auditoren und über den Konsens im Netzwerk statt. Die Migration sah zunächst vor das Frontier Netzwerk herunterzufahren, Zustände in Contracts zu löschen, ETH zu transferieren und zum neuen Homestead Netzwerk zu wechseln. Auch sollten Miner für einen Block nun 100% Vergütung erhalten und die Sicherheitswarnung „…~~SAFE~~…“ sowie der Checkpoint-Mechanismus entfernt werden. Da aufgrund einer erfolgreichen Testphase mit Frontier keine Checkpoints eingeführt werden mussten und mit fünf ETH bereits eine 100% Vergütung erfolgt, wurde lediglich die Sicherheitswarnung entfernt. Zudem findet im Hauptnetz bei Blocknummer 1.150.000 (494.000 im Testnetz am 03.03.2016 um 05:42:00 Uhr) ein Fork statt. Zustände in Contracts sowie ETH in Accounts bleiben erhalten. [Gu15], [Tu15b], [Wi16b] Bei einer durchschnittlichen Intervall von 17 Sekunden zwischen zwei Blöcken und der Blocknummer 1.139.113 am 12.03.2016 um 16:38:45 Uhr laut Ethereum Blockchain Explorer [Et16c] kann der Fork auf Homestead für das Hauptnetz per JavaScript siehe Listing 1 berechnet werden.

```
1   var f = 1150000; //fork at block number
2   var b = 1139113; //block number on 12th march 2016 at 16:38:45
3   var t = 17; //avg time in sec between blocks with frontier (homestead ~~ 14 sec)
4   var d = new Date('2016-03-12 16:38:45'); //datetime for b
5   d.setSeconds(d.getSeconds() + (f - b) * t); //datetime with fork to homestead
6   console.log(d); //Mon Mar 14 2016 20:03:24 GMT+0100 (Mitteleuropäische Zeit)
```

LISTING 1: JAVASCRIPT ZUR BERECHNUNG DES ETHEREUM HOMESTEAD FORK

In 2016 soll außerdem noch die Version Metropolis veröffentlich werden mit einer breiten Benutzerschnittstelle für nicht technisch versierte Nutzer. Ein groß angekündigtes Feature ist hierbei der MIST Browser als App-Store für Dapps. Die letzte Version (1.1) mit den Namen Serenity besitzt eine fundamentale Änderung im Mining-Prozess durch den Wechsel des Konsens-Algorithmus von PoW auf Proof-of-Stake. [Et16b], [Gu15]

Im Weiteren wird das Ethereum Protokoll mit Homestead als zum Zeitpunkt dieser Arbeit aktuellste Version betrachtet. Das nächste Kapitel beschreibt im Vergleich zu Bitcoin die State Transition Funktion, Blockchain und virtuelle Maschine des Ethereum Protokolls.

4.2 Ethereum Protokoll

In diesem Kapitel werden mit Accounts, States und der virtuellen Maschine die Prinzipien, Strukturen und Funktionen beschrieben. Dabei werden die grundlegendsten Unterschiede zu Bitcoin aufgezeigt.

Ethereum ist ein Open Source Protokoll basierend auf einer eigenen Blockchain integriert mit einer vollständigen Programmiersprache für die Erstellung von Smart Contracts und Dapps. Mit wenigen Zeilen Code kann z.B. eine Minimalversion von Namecoin siehe Kapitel 3.6.3, ein Reputation- oder ein Währungssystem umgesetzt werden. Darüber hinaus teilen sich Anwendungen unabhängig vom Client die wirtschaftliche Umgebung und Sicherheit der Blockchain. [Bu14b] Das Ethereum Protokoll basiert auf fünf grundlegenden Prinzipien [Et16h]:

1. **„Sandwich“ Komplexität** bedeutet, dass die Komplexität in den mittleren Architekturschichten sitzt und sowohl die obere Schicht mit den Schnittstellen für Entwickler und Anwender als auch in der untersten Schicht der Kern Konsens einfach und verständlich sind.
2. **Freiheit für die Anwender** bedeutet, dass Benutzer nicht in der Verwendung von Ethereum eingeschränkt werden analog zur Netzneutralität und im Gegensatz zu Bitcoins `OP_RETURN` Restriktion auf `40 Bytes`
3. **Generalisierung** bedeutet, dass Funktionen und Operationen in beliebiger Kombination verwendet werden können. So wird z.B. anstatt alle Transaktionen und Nachrichten zu protokollieren die Operation `LOG` angeboten.
4. **Keine Features** bedeutet, dass in Konsequenz der Generalisierung keine Funktionen wie z.B. die Sperrzeit in Bitcoin von Ethereum angeboten werden, da diese in Contracts selbst implementiert werden können.
5. **Risikobereitschaft** bedeutet, die Akzeptanz eines höheren Risikos durch grundlegende Verbesserungen wie z.B. eine 50-fach schnellere Blockerstellung.

Neben der Turing-vollständigen Maschine besitzt Ethereum mit der State Transition Funktion und eigenen Blockchain weitere und grundlegende Unterschiede zu Bitcoin, die in den folgenden Abschnitten beschrieben werden.

4.2.1 State Transition Funktion

Dieses Kapitel beschreibt Accounts, Nachrichten, Transaktionen und die darauf aufbauende State Transition Funktion von Ethereum. Ein Zustand (State) wird über die Ausführung von Transaktionen innerhalb der State Transition Funktion in einen Folgezustand überführt. In Bitcoin besteht der State aus dem UTXO Cache und in Ethereum aus Accounts.

Ein Account besitzt neben Kontostand (Balance) die Adresse, einen Transaktionszähler, den Contract Code und einen internen Speicher. Über den Transaktionszähler wird sichergestellt, dass eine Transaktion nur einmal verarbeitet wird zum Schutz vor Replay Angriffen und „Double Spending“. Der Contract Code wird ausgeführt wenn ein Account eine Transaktion

erhält. Dieser kann den internen Speicher lesen und schreiben oder Nachrichten an andere Accounts senden. Ethereum unterscheidet Accounts durch ihren Besitz und ihre Kontrolle. Es gibt über private Schlüssel extern kontrollierte Accounts und solche die nur von ihrem Contract Code kontrolliert werden und somit autonom laufen. [Bu14b] Die Vorteile von Accounts und UTXO sind in Tab. 13 gegenübergestellt.

UTXO	Accounts
Mehr Möglichkeiten für Skalierung dadurch, dass nur der Besitzer von Coins den Merkle Baum für dessen Besitznachweis verwaltet.	Geringer Speicherverbrauch dadurch, dass ein Account alle UTXO zusammenführt und Transaktionen nur eine Referenz, eine Signatur und einen Ausgang besitzen.
Höherer Level an Privatsphäre dadurch, dass bei jeder Transaktion eine neue Adresse verwendet wird.	Größere Übertragbarkeit dadurch, dass es kein Blockchain Konzept für die Herkunft spezifischer Coins gibt. Accounts sind einfacher zu programmieren und zu verstehen vor allem bei komplexen Skripten. Referenzen bleiben im Client erhalten auch bei neuen Transaktionen. Leichtgewichtige Clients können jederzeit alle Daten eines Accounts beziehen.

TAB. 13: VORTEILE VON UTXO UND ACCOUNTS ALS STATE [ET16H]

Mit Accounts in Ethereum ist im Gegensatz zu Bitcoin eine feingranulare Kontrolle über den Balance möglich. Darüber hinaus können statt des binären State („unspent“) von Bitcoin mit Accounts auch mehrstufige States abgebildet werden. [Et16h]

Der Übergang von einem State zum nächsten wird durch eine Transaktion abgewickelt. Eine Transaktion in Ethereum ist ein signiertes Datenpaket mit einer Nachricht von einem externen Account. Im Unterschied zu Bitcoins Transaktionen kann eine Nachricht sowohl von einem Contract Account als auch von einem externen Account gesendet werden. Daten enthalten und ein Contract Account kann eine Antwort zurückgeben. Eine Transaktion beinhaltet in Ethereum den Empfänger der Nachricht, eine Signatur des Senders, einen Betrag in ETH, die zu sendenden Daten sowie die Werte `STARTGAS` und `GASPRICE` als limitierende Größe für den Miner und die Code-Ausführung zum Schutz vor Endlosschleifen und DOS-Angriffen. Für die Erstellung eines Contract Account gibt es einen separaten Transaktions- und Nachrichtentyp wobei dessen Adresse aus dem Hash der Transaktionsdaten und dem Transaktionszähler gebildet wird. Contract Accounts und externe Accounts haben dieselben Privilegien inklusiv dem Erstellen von Accounts. [Bu14b]

Die State Transition über Transaktionen mit dem State aller UTXO in Bitcoin zeigt das Beispiel aus Abb. 13 mit jeweils `10k` Transaktionsgebühren. Bob sperrt `output0` für Alice und `output1` für Charles. Diese verwenden `input0` und `input1` ihrer Transaktionen `TX1` und `TX2` mit einem Signaturskript zum Entsperren des jeweiligen Outputs von Bob.

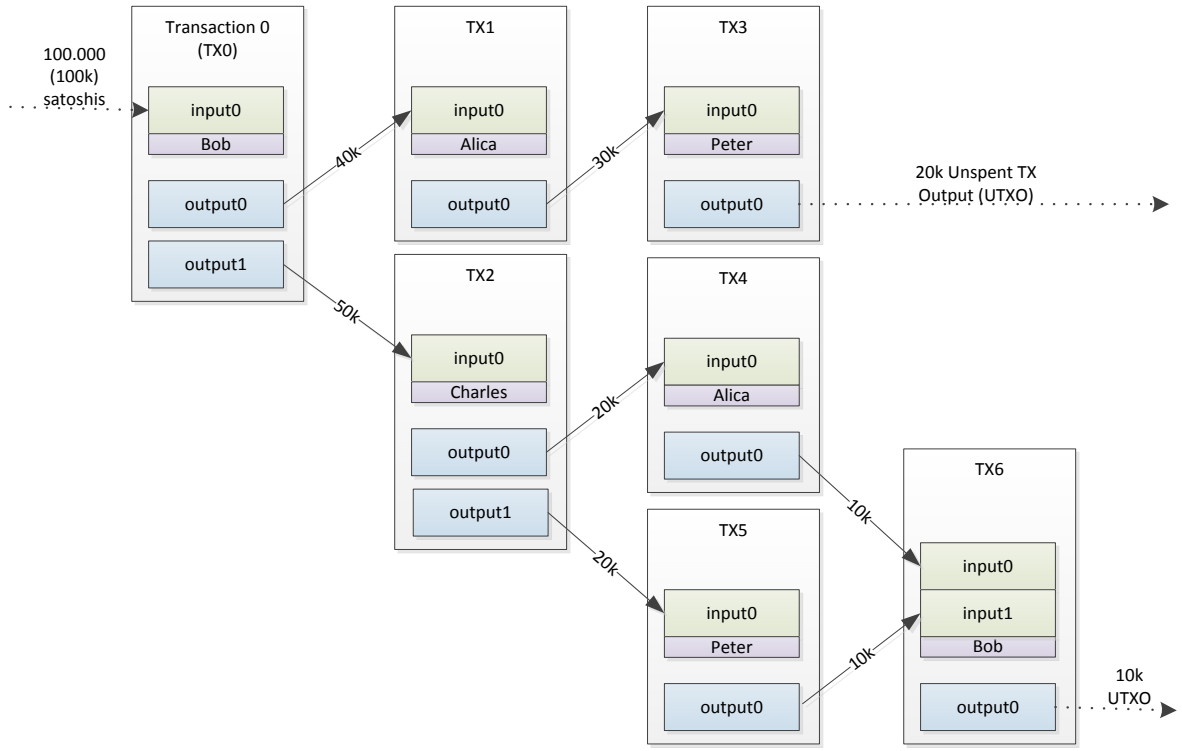

ABB. 13: STATE TRANSITION IN BITCOIN [ET16H]

Analog zum Beispiel der State Transition in Bitcoin aus Abb. 13 besitzen Bob, Alice, Charles und Peter in Ethereum jeweils einen externen Account und der State ist die Summe aller Accounts. Am 20.02.2016 betrug der `GASPRICE` durchschnittlich ca. 55 GWei [Et16i]. Eine einfache Transaktion benötigt `21.000 GAS`. Alle in einer Transaktion ausführbaren Operationen haben abhängig von Speicherbelegung und Rechenzeit einen `GAS` Wert. Eine ausführliche Beschreibung aller Operationen mit `GASPRICE` und Speicherbelegung beschreibt Dr. Gavin Wood in seinem Yellowpaper [Wo14]. Die Gebühren einer Transaktion berechnen sich anfangs durch `GASPRICE*STARTGAS` somit zu 1.155.000 GWei. In Tab. 14 wird das Beispiel aus Abb. 13 mit Accounts in Ethereum in MWei dargestellt, wobei die Transaktionsgebühren (1155 MWei) vom Account abgezogen wurden. Der Account wird hier durch seinen eingefärbten Balance (+/-) repräsentiert und jede Transition erhöht den Transaktionszähler der betreffenden Accounts z.B. Bob_0 auf Bob_1.

i	State i	Transition zu State i+1
0	Bob_0 : 100.000, $Alice_0$: 0, $Charles_0$: 0, $Peter_0$: 0	Bob überträgt Alice 40.000
1	Bob_1 : 58.845, $Alice_0$: 40.000, $Charles_0$: 0, $Peter_0$: 0	Bob überträgt Charles 50.000
2	Bob_2 : 7.690, $Alice_0$: 40.000, $Charles_0$ = 50.000, $Peter_0$: 0	Alice überträgt Peter 30.000
3	Bob_2 : 7.690, $Alice_1$: 8.845, $Charles_0$: 50.000, $Peter_0$: 30.000	Charles überträgt Alice 20.000
4	Bob_2 : 7.690, $Alice_1$: 28.845, $Charles_1$: 28.845, $Peter_0$: 30.000	Charles überträgt Peter 20.000
5	Bob_2 : 7.690, $Alice_1$: 28.845, $Charles_2$: 7.690, $Peter_0$: 50.000	Alice überträgt Bob 10.000

6	Bob_2 : 17.690, $Alice_2$: 17.690, $Charles_2$: 7.690, $Peter_0$: 50.000	Peter überträgt Bob 10.000
7	Bob_2 : 27.690, $Alice_2$: 17.690, Charles : 7.690, $Peter_1$: 38.845	LAST RECENT STATE

TAB. 14: STATE TRANSITION IN ETHEREUM

Die State Transition Funktion von State `i` zu State `i+1` ist in Tab. 14 vereinfacht beschrieben und in Ethereum durch folgende Schritte definiert:

1. Prüfe ob die Transaktion valide ist (insbesondere Signatur und Transaktionszähler).
2. Berechne `STARTGAS*GASPRICE`, ermittle die Senderadresse über die Signatur, ziehe die Gebühren von dessen Account ab und erhöhe den Transaktionszähler um 1.
3. Initialisiere `GAS=STARTGAS` und ziehe hiervon `GAS` per `Byte` basierend auf der Transaktionsgröße in `Byte` ab.
4. Transferiere die Werte einer Transaktion (Betrag und Daten) an den Account des Empfängers. Wenn dieser nicht existiert wird er generiert. Wenn der Account einen Contract Code besitzt wird dieser ausgeführt bis zum Ende oder `OUTOFGAS`.

Der letzte Schritt ist abhängig davon ob die Transaktion erfolgreich ausgeführt werden konnte d.h. `STARTGAS` und Balance des Senders waren ausreichend und im Falle eines Contract konnte dessen Code ohne `OUTOFGAS` terminiert werden. Bei erfolgreicher Transaktion werden die Gebühren dem Mining Account und `GAS` dem Account des Senders hinzugefügt. Im Fehlerfall werden alle Änderungen des States rückgängig gemacht, außer den Gebühren die dem Mining Account hinzugefügt werden. [Bu14b] Das nächste Kapitel zeigt wie State und Transaktion in der Blockchain gespeichert und enkodiert werden.

4.2.2 Ethereum Blockchain

Dieses Kapitel beschreibt den grundlegenden Aufbau der Ethereum Blockchain und seinen Blöcken, die im Gegensatz zu Bitcoin alle Transaktionen und States für die State Transition Funktion enthalten. Darüber hinaus wird mit dem GHOST (Greedy Heaviest Observed SubTree) Protokoll, Merkle Patricia Bäumen (Practical algorithm to retrieve information coded in alphanumeric) und RLP (Recursive Length Prefix) die Datenstruktur vorgestellt.

In der Blockchain von Ethereum sind wie in Bitcoin Blöcke über den Hash des Vorgängers und einem zusätzlichen Timestamp miteinander verknüpft. Dabei wird jeweils ein Merkle Patricia Baum für Transaktionen, dessen Bestätigungen und Empfänger, den State in einem Blockheader sowie für den Speicher eines Account innerhalb des State verwendet. Abb. 14 zeigt diese Struktur mit `STATE_ROOT` und `STORAGE_ROOT` ausschnittsweise. Hier werden exemplarisch ein Teil des Blockheader, die Baumstruktur des States und hierin der Verweis auf einen Account ebenfalls mit einer Baumstruktur für den Speicher dargestellt. Eine vollständige Liste mit allen Feldern des Blockheader zeigt Tab. 15. Das Feld `PREVHASH` aus Abb. 14 lautet hier `parentHash`.

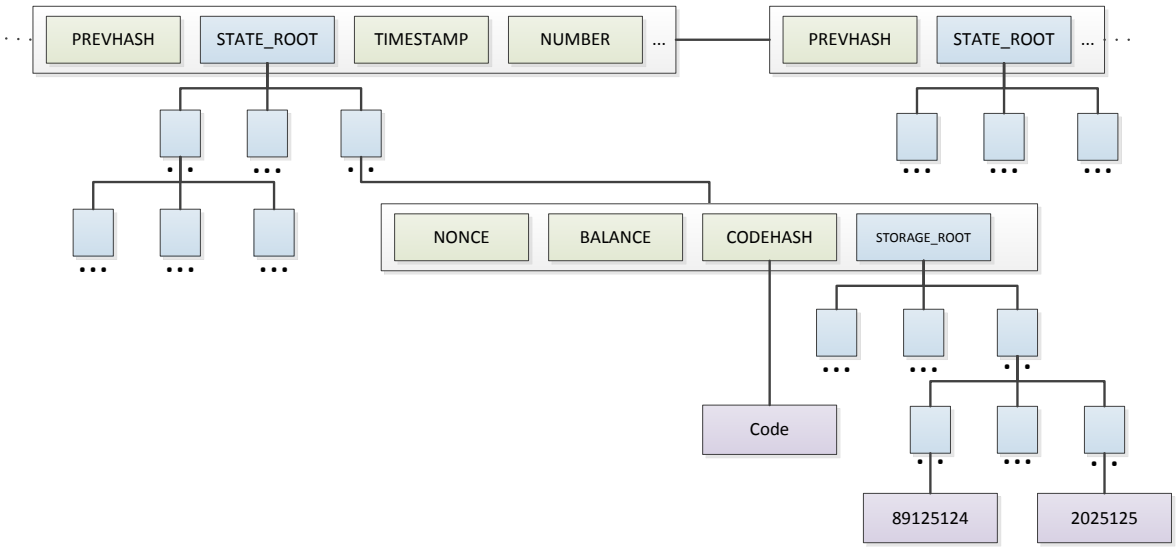

ABB. 14: ETHEREUM BLOCKCHAIN [ET16E]

Angefangen hat die Blockchain mit dem Genesis Block 0 in Frontier (mit dem Hash des Block 1.028.201 vom Testnetz) mit allen Transaktionen des Crowdfunding [Tu15a]. Ein Block besitzt einen Blockheader, eine Liste weiterer Blockheader abgelaufener Blöcke (Forks) gleicher Struktur wie in Tab. 15 und eine Liste mit Transaktionen.

Größe/Typ	Feld	Beschreibung
256 Byte	parentHash	Referenz auf den Hash des vorherigen Blocks in der Kette
256 Byte	ommersHash	Hash der Liste mit weiteren Blockheadern im Block
20 Byte	beneficiary	Adresse des Miner für Transaktionsgebühren
256 Byte	stateRoot	Hash vom Root des Merkle Patricia Baums mit dem State
256 Byte	transactionRoot	Hash vom Root des Merkle Patricia Baums mit den Transaktionen
256 Byte	receiptsRoot	Hash vom Root des Merkle Patricia Baums mit den Transaktionsbestätigungen
x Byte	logsBloom	Bloomfilter zusammengestellt aus indexierten Feldern von Logeinträgen der Transaktionsbestätigungen
Int	difficulty	Schwierigkeit kann über den Timestamp und der Schwierigkeit des vorherigen Blocks in der Kette berechnet werden
Int	number	Blockhöhe gestartet bei 0 (Genesis)
Int	gasLimit	angesetztes GAS Limit für diesen Block
Int	gasUsed	benötigte Menge an GAS für diesen Block
Int	timestamp	Zeitstempel zu Anfang des Blocks (Unix Zeit in Sekunden)
<= 32 Byte	extraData	relevante zusätzliche Daten für den Block (Client und Version)
256 Byte	mixHash	PoW zusammen mit dem Feld „nonce“
64 Byte	nonce	PoW zusammen mit dem Feld „mixHash“

TAB. 15: STRUKTUR DES ETHEREUM BLOCKHEADER [WO14]

In Ethereum wird der aktuellste SHA-3 Standard des NIST [NI15] als Hashfunktion mit `256 Bit` verwendet z.B. für den Root der Merkle Patricia Bäume aus Tab. 15. Die Validierung des Blockheader umfasst folgende Schritte [Wo14]:

1. Prüfe die PoW über `nounce` und `mixHash`.
2. Prüfe ob die `difficulty` entsprechend erhöht wurde wenn der Block weniger als 13 Sekunden nach `timestamp` des `parentHash` erstellt wurde. Bei 13 oder mehr Sekunden muss `difficulty` entsprechend geringer sein.
3. Prüfe ob `gasUsed` kleiner oder gleich dem `gasLimit` ist.
4. Prüfe ob das `gasLimit` nicht mehr als 1/1024 abweicht.
5. Prüfe ob das `gasLimit` 125.000 oder größer ist.
6. Prüfe ob der `timestamp` größer ist als die vom `parentHash`.
7. Prüfe ob `number` um eins höher ist als die vom `parentHash`.
8. Prüfe ob das Feld `extraData 32 Byte` oder kleiner ist.

Leichtgewichtige Clients in Ethereum laden analog zu Bitcoin nur Blockheader und können nur diese verifizieren. Weitere Informationen wie Accounts oder Transaktion werden über das Netzwerk von vollständigen Clients bei Bedarf geladen. Eine Validierung eines Blocks kann nur von vollständigen Clients durchgeführt werden und umfasst neben der Validierung des Blockheaders folgende zusätzliche Schritte [Wo14]:

1. Nimm den State vom `parentHash` und führe hierauf die State Transition Funktion über alle Transaktionen in der indexierten Reihenfolge durch. Hierüber wird der Merkle Patrica Baum des States aufgebaut. Prüfe ob dessen Hash vom Root `stateRoot` entspricht.
2. Erstelle über die Liste der weiteren Blockheader abgelaufener Blöcke (max. 2) eine RLP Sequenz und prüfe ob dessen Hash `ommersHash` entspricht.
3. Erstelle einen Merkle Patrica Baum über die indexierten Transaktionen und prüfe ob der Hash vom Root `transactionRoot` entspricht.
4. Erstelle einen Merkle Patrica Baum über die indexierten Transaktionsbestätigungen und prüfe ob der Hash vom Root `receiptsRoot` entspricht.

Transaktionen und Transaktionsbestätigungen sind über einen Index der ihre Reihenfolge bestimmt miteinander verknüpft. Transaktionsbestätigungen beinhaltet den Knoten im State Merkle Patricia Baum nach Durchführung der Transaktion, die benötigte Menge `GAS`, eine Liste mit Logeinträgen und einem hierauf basierenden Bloomfilter. `logsBloom` verknüpft diese per `OR` im Blockheader. Diese Struktur basiert wie Bitcoins SPV auf leichtgewichtigen Clients. [Et16h] Mit Patrica Merkle Bäumen soll Ethereum für alle Geräteklassen unabhängig von Rechenleistung, Netzwerk- und Speicherkapazität genutzt werden können. Dies umfasst Computer, Notebooks, Tablets, Smartphones sowie generell Smart Properties und das IoT-Umfeld. Mit Patrica Merkle Bäumen sind z.B. folgende Anfragen möglich [Bu15a]:

- Ist die Transaktion in einem Block enthalten?
- Wie lautet die aktuelle Balance eines Accounts?
- Existiert ein Account?
- Welche Instanzen zu einem Event eines bestimmten Typs (z.B. Zielerreichung eines Crowdfunding Contract) wurden in einem bestimmten Zeitraum von einer bestimmten Adresse ausgegeben?

Die erste Anfrage wird über den Baum mit den Transaktionen, die zweite und dritte Anfrage über den Baum mit dem State und die die vierte Anfrage über den Baum mit den Transaktionsbestätigungen beantwortet. [Bu15a]

In Ethereum werden Merkle Patrica Bäume basierend auf dem Hex Code Alphabet mit 16 + 1 terminierenden Kind Knoten für Key-Value (KV) Paare zu je `32 Byte` aufgebaut. Für Suche, Einfügen und Löschen haben diese Bäume die Komplexität von $\log_2(N)$ analog zur Verifikation in Merkle Bäumen. Bei Werten größer `32 Byte` wird ein SHA-3 mit `256 Bit` gebildet, so dass alle Werte immer `32 Byte` groß sind. [Wo14] Der Key ist gleichzeitig der Pfad zu einem Blatt in dem Baum, welches den zugehörigen Wert enthält. Abb. 15 zeigt den Merkle Patrica Baum aus einem Beispiel von Ethereum [Et16f] für eine Liste mit KV-Paaren `('dog', 'puppy'), ('horse', 'stallion'), ('do', 'verb'), ('doge', 'coin')`. Keys werden in Hexcode_{16} dargestellt und Knoten sind von `A-G` nummeriert.

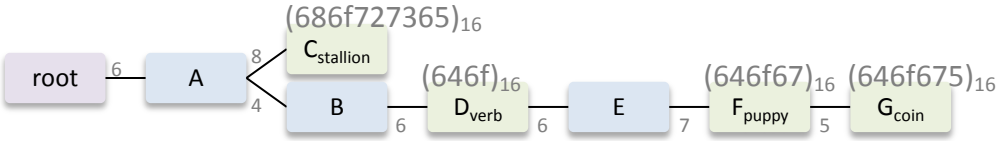

ABB. 15: MERKLE PATRICA BAUM IN ETHEREUM [ET16F]

Die Knoten `root, B, E` und `G` aus Abb. 15 sind KV-Knoten, die aus einer Liste mit zwei Items bestehen z.B. für Knoten `B` mit `['00 6f', D]`. Dem Key wird dabei jeweils ein Hex Präfix (HP) vorangestellt, der für eine gerade Anzahl an Zeichen sorgt und das Ende wie bei `C` und `G` markiert. Die restlichen Knoten sind sogenannte divergierende Knoten, die aus 17 Items bestehen. Der Knoten `A` bspw. mit `['', '', '', '', B, '', '', '', C, '', '', '', '', '', '', '', '']` zeigt an vierter Stelle auf Knoten `B` und an achter Stelle auf Knoten `C`. Die erste Stelle mit dem Index 0 und die anderen haben keinen Verweis. Der letzte Eintrag bei KV-Knoten kann entweder wie bei `B` ein Verweis auf einen anderen Knoten oder wie bei `C` mit `'stallion'` den Wert enthalten. Der letzte Eintrag bei divergierenden Knoten ist entweder leer wie bei `A` oder besitzt den Wert wie bei `D`. Da in dem Beispiel aus Abb. 15 keine Knoten größer `32 Byte` sind werden die Daten lediglich RLP enkodiert und müssen nicht in eine lokalen KV-Datenbank (Ethereum nutzt wie Bitcoin Googles LevelDB [GD16]) gespeichert werden. Ab `32 Byte` wird ein Hash gebildet und als Key zusammen mit dem Value in der Datenbank gespeichert. In dem Merkle Patrica Baum für den State sind die Keys die Ethereum Adressen und in dem Merkle Patrica Baum für

Transaktionen und dessen Bestätigungen ein Integer als Index für die Reihenfolge bzw. Transaktionszähler. [Et16f], [Wo14] Mit der RLP Enkodierung werden in Ethereum `Byte` Arrays als Objekte serialisiert um einfache Datentypen an höhere Schichten zu geben. Integer werden in Big-Endian (höherwertige `Bits` auf kleinerer Speicheradresse) ohne führende Nullen interpretiert wobei Gleitkommawerte und Vorzeichen nicht vorgesehen sind. [Et16j] Die Regeln für die RLP Enkodierung sind in Tab. 16 beschrieben.

Typ	Größe	Beschreibung
`string`	`1 Byte`	Wert bleibt unverändert
`string`	`0-55 Byte`	`0x80` + Länge des Strings z.B. `'dog'=[0x83, 'd', 'o', 'g']`
`string`	`> 55 Byte`	`0xb7` + Länge des String
`list`	`0-55 Byte`	`0xc0` + Länge der Liste z.B. `['dog','puppy']=[0xca, 0x83, 'd','o','g', 0x85, 'p','u','p','p','y']`
`list`	`> 55 Byte`	`0xf7` + Länge der Liste

TAB. 16: RLP ENKODIERUNG IN ETHEREUM [ET16J]

Im Gegensatz zu Bitcoin besteht die Blockchain in Ethereum auf dem GHOST Protokoll von Sompolinsky und Zohar [SZ15] aus Forks und ist somit baumartig aufgebaut. Das GHOST Protokoll adressiert das Problem aus Bitcoin der verminderten Sicherheit durch kürzere Bestätigungszeiten. Diese resultiert aus der erhöhten Anzahl abgelaufener Blöcke durch die Zeit für die Verbreitung eines Blocks innerhalb des Netzwerks. Der Ansatz von GHOST ist es die abgelaufenen Blöcke in die Berechnung der längsten Kette aufzunehmen. Ethereum adaptiert dieses Modell mit der Besonderheit, dass Miner für abgelaufenen Blöcke und die, die sie integrieren, eine anteilige Vergütung bekommen. Dies reduziert den Effekt der Zentralisierung von Mining Pools im Netzwerk mit großer Hashrate, reduziert die Rate abgelaufener Blöcke und geringere Zeitdifferenz. [Bu14b], [Bu14c] Am 25.02.2016 betrug laut Blockchain Explorer [Et16i] die durchschnittliche Zeitdifferenz zwischen zwei Blöcken 17 Sekunden (seit Version Homestead 14 Sekunden). Der Miner erhält für einen erstellten Block fünf ETH Vergütung. Darüber hinaus kann dieser maximal zwei abgelaufene Blöcke enthalten wofür der Miner nochmals jeweils einen Anteil von 1/32 erhält. In diesem Fall erhält der Miner des abgelaufenen Blocks erhält einen Anteil, der sich wie Formel 2 zeigt abhängig von der Blocknummer U_i des inkludierenden Blocks und seiner Blocknummer B_{H_i} berechnet.

$$\sigma[U_c]_b = \sigma'[U_c]_b + \left(1 + \frac{1}{8}\left(U_i - B_{H_i}\right)\right) R_b$$

FORMEL 2: VERGÜTUNG ABGELAUFENER BLÖCKE IN ETHEREUM [WO14]

Der neue Balance b des State σ des Accounts c des abgelaufenen Blocks U (Uncle) ist gleich dem vorherigen State σ' addiert mit dem Anteil der Vergütung ($R_b = 5$ ETH Revenue) für das Erstellen eines Blocks. Der Anteil berechnet sich über die Differenz der Blocknummern i (Inkrement) und liegt durch die Beschränkung auf die 7. Generation zwischen 1/8 und 7/8. Ein abgelaufener Block muss einen valide Blockheader besitzen und darf nicht in vorangegangenen Blöcken übernommen worden sein. [Et16h]

Der Mining Algorithmus Ethash in Ethereum basiert auf speicherintensiver Berechnung statt auf spezifischer Hardware wie mit ASIC in Bitcoin um das Mining mehr zu dezentralisieren. Hierfür wird `1 GByte` Datensatz aus einem `16 MByte` pseudozufälligem Cache aus den Blockheadern (ohne `mixHash` und `nounce` siehe Tab. 15) generiert und alle 30.000 Blöcke aktualisiert. Das Mining besteht aus der SHA-3 Hashberechnung zufällig ausgewählter `64 Byte` Abschnitte aus dem Datensatz unter Berücksichtigung der `difficulty` siehe Tab. 15. Leichtgewichtige Clients können die PoW des Mining über denselben Cache aus den Blockheadern mit geringen Ressourcen effizient verifizieren. [Et16a]

Im nächsten Kapitel wird die Turing-vollständige Maschine vorgestellt, die den Contract Code ausführt und basierend auf Patricia Bäumen und RLP mit `32 Byte` Wörtern arbeitet. Dabei wird neben der Ausführungsumgebung insbesondere gezeigt wie mit jeder Operation das `GAS` berechnet wird.

4.2.3 Ethereum Virtual Machine

Der Kern von Ethereum basiert auf einer eigenen Turing-vollständigen, virtuellen Maschine (EVM) zur Ausführung von Transaktionen mit der zuvor beschrieben State Transition Funktion in der Blockchain. In diesem Kapitel wird die Umgebung und Code-Ausführung der EVM beschrieben, auf die Clients und höhere Programmiersprachen wie Solidity für Dapps und Contracts wie später gezeigt wird aufsetzen.

Die EVM arbeitet analog zur Skriptausführung in Bitcoin mit einem LIFO Stapel von maximal 1.024 `32 Byte` Einträgen und mit einem `32 Byte`-Array als unbegrenzten Speicher passend zu SHA-3 mit `256 Bit`, ECDSA und den `32 Byte` KV-Paaren der Merkle Patricia Bäume. Speicher und Stapel der EVM sind volatil im Gegensatz zum Account Speicher als Teil des States. Der Contract Code ist unveränderbar. Während der Ausführung besteht Zugriff auf Transaktion, Nachricht und Daten aus dem Blockheader. Die Umgebung I umfasst während der Code-Ausführung laut Wood [Wo14]:

- Adresse des Account mit dem Code
- Adresse des Senders der Transaktion als Urheber
- `GASPRICE` der Transaktion
- `Byte`-Array mit den Eingabedaten für die Ausführung
- Adresse des Account der die Ausführung initiiert
- Betrag in Wei
- `Byte`-Array mit dem Code zur Ausführung
- Blockheader des vorliegenden Blocks
- Anzahl an `CALL` und `CREATE` bis zur aktuellen Ausführung

Das Ergebnis der Code-Ausführung ist ein Tupel aus dem resultierenden State σ', den Ausgabedaten, dem verbleibenden `GAS` und dem Substate A bestehend aus einer Liste mit Logeinträgen, zu löschenden Accounts und `GAS` zur Rückerstattung. Die Code-Ausführung der EVM ist von Wood [Wo14] definiert durch Formel 3.

$$X\big((\sigma,\mu,A,I)\big) \equiv \begin{cases} (\emptyset,\mu,A^0,I,()) & wenn\ Z(\sigma,\mu,I) \\ O(\sigma,\mu,A,I)\cdot H(\mu,I) & wenn\ H(\mu,I)\neq\emptyset \\ X\big(O(\sigma,\mu,A,I)\big) & andernfalls \end{cases}$$

FORMEL 3: CODE-AUSFÜHRUNG IN ETHEREUM [WO14]

Die Funktion Z gibt `true` zurück wenn eine Ausnahme aufgetreten ist wie z.B. bei invalider Adresse oder Operation, Überlauf des Stapel bzw. Speichers oder bei `OUTOFGAS`. In diesen Fällen werden alle Änderungen rückgängig gemacht. Die Funktion H gibt an ob die Code-Ausführung regulär mit `STOP`, `SUICIDE` oder `RETURN` beendet wurde oder noch aktiv ist. Der Maschinen State μ besteht aus einem Tupel von verfügbarem `GAS`, einem Programmzähler, Speicherinhalt, Anzahl der Wörter im Speicher und den Inhalt des Stapels. Die Funktion X selbst ist rekursiv und ruft die Funktion O auf, die eine Operation auf dem Stapel ausführt. Dabei wird der Maschinen State μ verändert durch die Reduzierung des verfügbaren `GAS`, der Erhöhung des Programmzählers und der Veränderung des Stapels. Jede Operation definiert die Anzahl an Elementen, die sie vom Stapel nimmt und dem Stapel hinzufügt. In der Regel ändern Operationen nur den Stapel und nicht den Speicherinhalt, Anzahl der Wörter im Speicher, den State σ oder den Substate A. Jede Ausführung einer Operation wie z.B. `ADD`, `MUL` oder `MOD` benötigt eine definierte Menge an `GAS`. Diese ist abhängig von der Komplexität und zusätzlich benötigtem Speicher. Formel 4 zeigt die Kostenfunktion C in Abhängigkeit zur Anzahl der Wörter im Speicher (μ_i) und der Operation.

$$C(\sigma,\mu,I) \equiv C_{memory}(\mu_i') - C_{memory}(\mu_i) + \begin{cases} G_{zero} & wenn\ w \in \{STOP, SUICIDE, RETURN\} \\ \dots & \dots \\ G_{mid} & wenn\ w \in \{ADDMOD, MULMOD, JUMP\} \end{cases}$$

$$C_{memory}(a) \equiv G_{memory} * a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

FORMEL 4: KOSTENFUNKTION FÜR OPERATIONEN IN ETHEREUM [WO14]

Das Wort w aus Formel 4 wird dabei über den Programmzähler aus dem `Byte`-Array des ausgeführten Codes ausgelesen und repräsentiert eine Operation hier als Beispiel für G_{zero} mit `0 GAS` und G_{mid} mit `8 GAS`. Es gibt einige Operationen wie z.B. `MSTORE`, die Daten in den Speicher kopieren so dass μ_i höher ist als sein vorheriger Wert μ_i'. Die Kosten für den Speicher berechnet die Funktion C_{memory} als Summe der Multiplikation von $G_{memory} = 3$ `GAS` mit der Anzahl der Wörter im Speicher $a = \mu_i \vee \mu_i'$ und der quadratischen Funktion mit $a^2/512$. Über die quadratische Funktion bleiben die Kosten bis ca. `704 Byte` (22 Wörter) linear und steigen danach kontinuierlich an. Mit der Kostenfunktion in Formel 4 werden Endlosschleifen, die mit Rekursion und den Operationen `JUMP` und `CALL` möglich sind, verhindert. Eine vollständige Aufstellung aller Operationen und ihren Kosten gibt Wood im Anhang G und H seines Yellowpaper [Wo14].

Im Blockchain Explorer EtherScan [Et16i] können zu Contract Accounts der Contract Code als Bytecode und Quellcode mit Operationen angezeigt werden. Zwei weitere Blockchain Explorer EtherCamp [Et16d] und EtherChain [Et16c] zeigen darüber hinaus für Contracts den

Solidity Quellcode mit ABI (Application Binary Interface) an. EtherCamp [Et16d] zeigt zudem auch den Speicher von Accounts und Contracts mit an und kann Transaktionen in der EVM über Programmzähler, Speicher, Stack und Operation nachverfolgen. Ein Beispiel für ein Contract und seine Code-Ausführung ist schrittweise mit den Änderungen von Speicher und Stapel basiert auf der Operation beschrieben in einem Tutorial von Ethereum [Et16e]. Der Contract stellt ein Namensregister zur Verfügung und empfängt Nachrichten mit `64 Byte` (`32 Byte` als Key und `32 Byte` als Value). Bei jeder eingehender Nachricht wird geprüft ob der Key im Account Speicher des Contract vorhanden ist. Wenn nicht wird hier das KV-Paar eingetragen.

Im nächsten Kapitel werden Anwendungsbereiche für Dapps mit Contracts sowie einige Beispiele beschrieben und aufgezeigt. Dabei werden die Vorteile mit der EVM als Turing-vollständige Maschine verdeutlicht.

4.3 Dapps und Contracts

Im Kapitel 3.6 wurden Anwendungsszenarien mit Bitcoin betrachtet, die auf der Blockchain aufbauen. Diese lassen sich basierend auf der Blockchain auch auf Ethereum übertragen. In diesem Kapitel werden Contracts und dezentralisierte Anwendungen (Dapps) mit Ethereum beschrieben sowie einige Beispiele aufgeführt.

In Kapitel 3.6.2 wurde der Begriff Dapp definiert. Ethereum [Et16e] beschreibt eine Dapp selbst als eine Kombination zwischen einem oder einer Gruppe von Contracts mit einem GUI (Graphical User Interface) um diese nutzen zu können. Eine vollständige Dapp besteht somit auf unterster Ebene aus der Businesslogik basierend auf den Komponenten wie Contracts, der Blockchain und weiteren Systemen sowie auf oberster Ebene aus einem GUI.

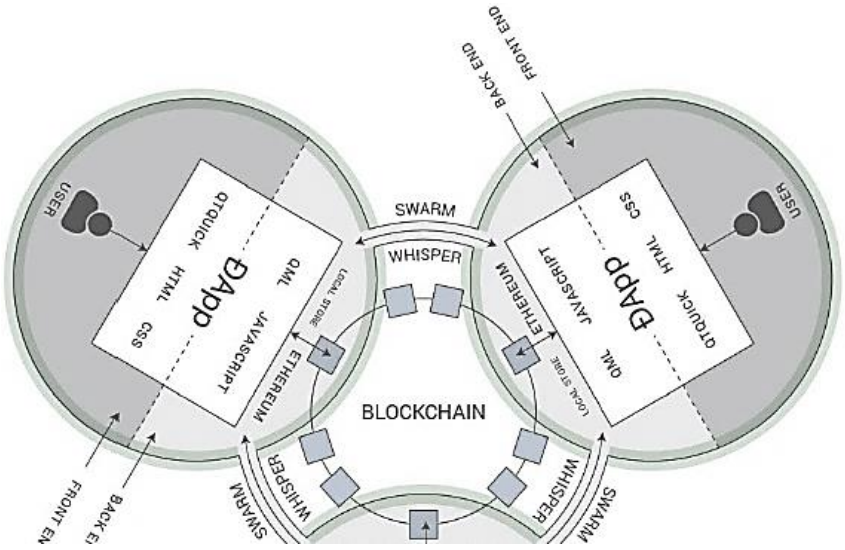

ABB. 16: DAPPS IN ETHEREUM [BU14A]

Abb. 16 zeigt das Frontend mit den Sprachen HTML (HyperText Markup Language), CSS (Cascading Style Sheets) und QTQuick als UI und das Backend mit den Sprachen JavaScript

und QML (QT Metaobject Language). Das Backend hat Zugriff auf den Ethereum Client bzw. auf den lokalen Speicher und auf die Blockchain (Contracts und Account States). Die Verbindung zwischen den Benutzer neben der Blockchain zeigt weitere P2P basierte Systeme Whisper und Swarm als Kommunikations- und Datenschicht, die sich zum Zeitpunkt der Arbeit noch in der Entwicklungsphase befanden. Whisper und Swarm gehören neben Contracts zur Vision von Ethereum und werden vermutlich mit dem Dapp-Browser MIST in der nächsten Version Metropolis erscheinen. Die weiteren Komponenten aus Abb. 16 werden basierend auf den Clients der aktuellen Version Homestead im nächsten Kapitel beschrieben. Dapps werden von Buterin [Bu14b] analog zu Kapitel 3.6 in drei Kategorien (finanzielle, semi-finanzielle und nicht-finanzielle Anwendungen) unterteilt und besitzen mit Ethereum vier wesentliche Vorteile gegenüber Bitcoin:

1. **Turing-vollständig.** Im Gegensatz dazu werden ohne Schleifen mehr Zeilen im Code benötigt. Ein alternativer ECDSA in Bitcoin erfordert z.B. 256 Multiplikationsrunden jeweils einzeln im Code enthalten.
2. **Wertbewusstsein.** Im Gegensatz dazu verfährt UTXO nach dem Prinzip „alles-oder-nichts“. Daher haben Skripte in Bitcoin keine feingranulare Kontrolle über den abzuhebenden Betrag, der in vielen Apps wie z.B. im Hedgegeschäft benötigt wird.
3. **Mehrstufiger Zustand.** Im Gegensatz dazu haben UTXO einen binären Zustand. Daher unterstützt Bitcoin keine komplexeren zustandsorientierten Contracts wie z.B. solche für dezentralisierte Organisationen.
4. **Bewusstsein der Blockchain.** Im Gegensatz haben UTXO keine Sicht auf Daten der Blockchain wie `nounce` oder `parentHash`. Dadurch haben Skripte in Bitcoin keine Quelle für zufällige Daten, wie sie in vielen Anwendungen wie z.B. im Glückspiel benötigt werden.

Das ETHDEV Team um Gavin Wood und Vitalik Buterin stellt für die Dapp Entwicklung eine Suite an Core Dapps zur Verfügung, die Nutzern benötigte Funktionen wie Wallets und ein Nachrichtensystem zur Verfügung stellen. Darüber hinaus sollen Meta-Dapps andere Dapps unterstützen wie z.B. ein Dapp-Store, ein Reputationssystem, ein Namensregister, ein Anti-Sybil-System oder dezentralisierte Steuerungsprogramme wie P2P Mining Pools. Neben diesen Core- und Meta-Dapps unterstützt das ETHDEV Team auch periphere Dapps [Bu14a]:

- Dezentralisierter Austausch (ETH basierte Assets und andere Tokens / Coins)
- Dezentralisierte Marktplätze
- Finanzanwendungen wie Hedgegeschäfte und Versicherungen
- Crowdfunding
- Treuhand, Verbraucherschutz und -schlichtung
- Veröffentlichung von Inhalten (mit Bonussystem)
- Dezentralisiertes Protokoll für Werbung
- Dezentralisiertes Organisationsmanagement

Ethereums State kann Balances, Reputationen, Vertrauensarrangements und generell mit Smart Properties alle Daten der realen Welt beinhalten, die ein Computer repräsentatieren

kann. Wood [Wo14] beschreibt mit Datenfeeds ein Pattern mit dem Daten der „externen Welt“ z.B. Wetterdaten oder Währungskurse Contracts und Dapps zur Verfügung gestellt werden können. Ein Datenfeed ist ein einzelner Service, dessen Genauigkeit und Zeitbezug über einen zusätzlichen Contract verwaltet wird. Dieser Contract dient als Bibliothek und stellt anderen Contracts Funktionen zur Verfügung (in Abb. 17 grün). In einem Tutorial von Ethereum [Et16e] werden drei weitere Typen von Contracts beschrieben. Ein weiterleitender Contract (Forwarding in Abb. 17 blau) kann externe Accounts abbilden mit komplexen Zugangsrichtlinien z.B. eine Multi-Signatur Wallet. Weitere Contracts verwalten Daten (in Abb. 17 rot), die nützlich für Andere oder für die „externe Welt“ sind z.B. eine eigene Währung oder Mitgliedschaften in Organisationen. Der letzte Typ verwaltet die Beziehung zwischen Benutzern (in Abb. 17 lila) wie bei Wettbüros oder Treuhandkontos. Abb. 17 zeigt ein Beispiel für die Interaktion zwischen diesen Contracts mit der Wette über 100 GavCoin zwischen Alice und Bob, dass zu keiner Zeit die Temperatur in San Francisco über 35°C steigt.

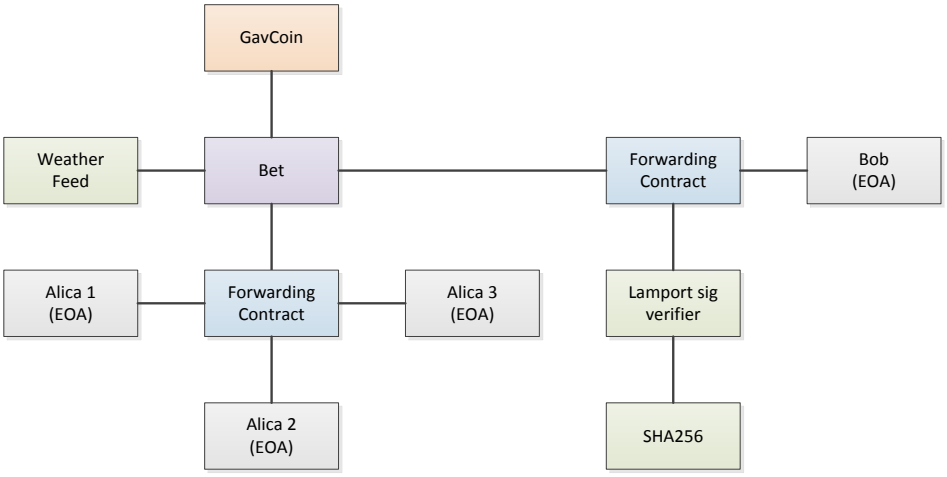

ABB. 17: INTERAKTION UNTERSCHIEDLICHER CONTRACTS [ET16E]

Alice besitzt drei EOA (External Owned Account) und verwendet ein Forwarding welches Nachrichten nur bei Genehmigung über zwei von drei privaten Schlüsseln sendet. Bob besitzt ein EOA und verwendet ein Forwarding, der nur Lamport signierte Nachrichten mit SHA-256 sendet. „Bet“ benötigt Wetterdaten aus dem „Weather Feed“ und überträgt den Betrag GavCoin an den Forwarding Contract des Gewinners. „GavCoin“ speichert den Betrag und Adresse. Übertragung meint die gleichwertige Änderung des Betrags der Forwarding Adressen von Alice und Bob. Der Solidity Quellcode für den Contract GavCoin zusammen mit der ABI Spezifikation und Dokumentation wird im Ethereum Wiki [Et16g] beschrieben. Beispiele weiterer Contracts wie DAOs, Auktionen, Abstimmungen oder Crowdfundings sind u.a. in den Ethereum Dokumentationen [Et16b], [Et16k] beschrieben. Für Solidity gibt es von Reitwiessner [Re16] ein Online-Compiler und Laufzeitumgebung in Echtzeit mit weiteren Funktionen wie `GAS` Berechnung, Bytecode und JavaScript für ein Deployment.

Wie anfangs beschrieben besitzen Dapps neben den Contracts ein darüber liegendes GUI. Mit dem Solidity Compiler sowie den Blockchain Browsern EtherCamp [Et16d] und EtherScan [Et16i] wurden bereits drei Dapps genannt, die in Ethereum als Live-Version bereitgestellt

wurden und über Webbrowser eine Schnittstelle zum Benutzer besitzen. In Tab. 17 werden weitere Dapps mit verfügbarer Live-Version (Stand 06.03.2016) aus dem größten Dapp Verzeichnis EtherCasts [Et16l] beschrieben.

Dapp	Beschreibung
etheroll.com	Glückspiel mit 100 seitigem Würfel, bei man innerhalb eines Wurfs auf einen von zwei Bereichen (1-50, 51-100) wettet und seinen Einsatz um den Faktor 1,98 (2% Provision) vermehren kann.
cryptorps.com	Dapp als Glückspiel „Schere-Stein-Papier“ basiert nicht auf dem Zufallsprinzip sondern auf den Wetten der Spieler wobei 90% (10% Provision) des verlorenen Einsatzes in den Jackpot gehen.
app.etherdoubler.com	Ein Depot zur Verdopplung einer Anlage, die pyramidenförmig aufgebaut wird. Sobald unterhalb einer Anlage zwei weitere Anlagen existieren, wird der Wert der Anlage um den Faktor 1,9 (10% Provision) vermehrt.
ethereumpyramid.com	Wie Dapp EtherDoubler nur mit der Möglichkeit auch in Bitcoin einzuzahlen für maximal 1 ETH.
groupgnosis.com	Dapp als Prognosemarkt wo zukünftige Events eingestellt werden können. Der Fund beträgt dabei 50 ETH (2% Provision). Anteile können gekauft, verkauft oder nach Eintritt des Events ausgezahlt werden.
etherwall.com	Desktop Wallet zur Verwaltung von Accounts und Senden von Transaktionen.
ethereumwall.com	Nachrichtenboard mit Nachrichten bis zu `300 KByte` unterstützt Markdown-Syntax und besitzt eine „Like“- und „Spenden“-Funktion.
etherid.org	Namensregister mit IPFS (InterPlanetary File System) Gateway für die Verwaltung von Domain mit Registrierung, Kauf und Verkauf. Der Besitz einer Domain dauert max. ein Jahr (ca. 2.000.000 Blöcke).
notareth.meteor.com	Dapp zur Existenzprüfung einer Datei basierend auf ihrem Hash.
oraclize.it	Service stellt Data Feeds per API zur Verfügung und verwendet eine Existenz Prüfung der Datenquellen analog zur Dapp NotarEth.
ethereum-alarm-clock.com	Dapp die Funktionen eines Contract in der Zukunft zu einer bestimmten Blocknummer ausführt.
etheria.world	Dapp als Spiel mit einer Virtuelle Welt basierend auf einem Register für virtuellen Landbesitz (1 ETH) welches über bewirtschaftet und bebaut (für 20 Blöcke je 12 Stunden ca. 2.500 Blöcke) werden kann.
etherlisten.com	Dapp zur Visualisierung und musikalischen Unterstreichung von Blöcken und Transaktionen basierend auf ihrer Größe.

TAB. 17: LIVE DAPPS IN ETHEREUM [ET16L]

Neben den beschriebenen Dapps aus Tab. 17 gibt es viele weitere Dapps die von EtherCasts [Et16l] in Staging-Phasen wie Prototyp, Demo oder Konzept kategorisiert werden. Daneben gibt es mit DappCentral [Da16c] und DappsList [Da16a] zwei weitere Verzeichnisse für Dapps und Dapp Browser. MIST ist die offizielle Ethereum Implementierung einer Wallet Dapp (Beta) mit Dapp Browser (Alpha), welche in der Version Metropolis von Ethereum erscheinen soll. DappsList bietet dabei ähnliche Informationen wie EtherCasts während DappCentral darüber hinaus auch Informationen zum Account und Contract bietet. Am 13.03.2016 hatte EtherCasts 168, DappsList 58 und DappCentral 11 Dapps gelistet. Zwar ist EtherCasts das größte Verzeichnis für Dapps, dennoch sind einige Dapps hier nicht gelistet und in Tab. 18 aufgeführt. *augur.net* wird von Vitalin Buterin und *slock.it* von fünf Mitgliedern des Ethereum ETHDEV Team u.a. Gavin Wood begleitet und mit entwickelt.

Dapp	Beschreibung
safemarket.github.io	Dapp als vollständiges P2P Shopsystem mit Produkten, Bestell-Versand und Zahlungsabwicklung mit Forum, Submärkten und einer Treuhandfunktion. Derzeit ist nur eine Alphaversion verfügbar.
freemyvunk.com	Dapp als Bewegung gegen die Spieleindustrie. Diese soll virtuelle Güter in der Blockchain als Smart Properties behandeln. VNK ist dabei die Währung die man für die Verbreitung der Nachricht über soziale Medien erhält. Dapp bietet den Marktplatz für virtuelles Eigentum in Onlinespielen wie World of Warcraft.
makerdao.com	DAO mit eigener Währung Dai dessen Preisstabilität über Contracts sichergestellt werden soll. Dazu wird eine zweite spekulative und volatile Währung benutzt.
slock.it	Dapp zum Leihen, Verkaufen und Teilen von ungenutzten Gegenständen wie z.B. Fahrrädern, Parktickets oder Unterkünften nach dem Prinzip von AirBnb gesteuert über Schlösser als Smart Properties.
augur.net	Dapp als Prognosemarkt analog zu groupgnosis.com aus Tab. 17 nur mit umfangreicherem GUI und größerem Funktionsumfang.
govevue.com	DAO zum Teilen von 30 Sekunden Videos. Man erhält BTC und Vevue Tokens für das Erstellen und Teilen eines Videos zu angefragten Orten der Umgebung

TAB. 18: WEITERE DAPPS IN ETHEREUM [DA16A], [DA16C]

Häufig wurden Dapps mit Live-Version in Ethereum Frontier als Alpha- oder Betaversionen beschrieben, da erst mit Homestead eine sichere und stabile Umgebung für einen produktiven Livebetrieb zur Verfügung stand. Dies zeigt sich dadurch, dass die Warnhinweise entfernt wurden siehe Kapitel 4.1. Mit Homestead hat die Entwicklung von Dapps und Akzeptanz zugenommen wie „The DAO“ als das größte Crowdfunding [Wi16a] beweist. Im nächsten Kapitel werden unterschiedliche Clients von Ethereum vorgestellt, mit denen Dapps entwickelt, bereitgestellt und in Ethereum betrieben werden können.

4.4 Ethereum Clients

In diesem Kapitel werden mit Ethereum Clients die Implementierungen des Ethereum Protokolls beschrieben, auf dessen Grundlage später eine Auswahl für den PoC getroffen wird. Die ersten vollständigen Clients nach formalisierten Beschreibung des Yellowpaper von Wood [Wo14] waren im Februar 2014 fertiggestellt. Weiter waren Implementierungen in Java und Python zu diesem Zeitpunkt nahezu fertiggestellt [Bu14b].

Mittlerweile gibt es Clients in Haskell, Rust und JavaScript. Eine Übersicht aller Clients zeigt Tab. 19 unter Angabe der Programmiersprache, Version, Plattform und Herkunft [Et16t].

Client	Sprache	Version	Beschreibung
MIST	JavaScript	0.5.2 (H)	MIST ist eine Multi Signatur Wallet und ein Dapp Browser für Mac OSX, Windows und Linux/Unix basiert auf NodeJS NPM, Meteor und Electron. Der Client enthält eine MIST-API und mit eth und geth vollständige Ethereum Knoten.
go-ethereum	Go	1.3.5 (H)	geth ist ein vollständiger Ethereum Knoten für Mac OSX, Windows und Linux/Unix sowie Raspberry Pi (ARM). Der Client dient als zukünftige Basis für Benutzer und den MIST Browser. Er besitzt neben der Dapp-API ein CLI, eine JavaScript Konsole und eine Management API. Aktuell wird nur CPU Mining unterstützt.
cpp-ethereum	C++	1.2.2 (H)	eth ist ein vollständiger Ethereum Knoten für Mac OSX, Windows und Linux/Unix. Der Client besitzt ein CLI mit Solidity Compiler. Darüber hinaus wird eine Qt-basierte GUI und mit MIX eine Entwicklungsumgebung für Contracts und UI geliefert. Es werden GPU und CPU Mining im Single- und Pool-Modus unterstützt.
pyethapp	Python	1.1.1	pyethapp ist ein vollständiger Ethereum Knoten für Mac OSX und Linux/Unix. Diese Implementierung ist die am lesbarsten und für Forschung und akademische Zwecke geeignet jedoch nicht performant für eine High-End Nutzung. Mit Version 1.2.0 soll eine Homestead kompatible Version kommen.
ethereumjs-lib	JavaScript	3.0.0 (H)	JavaScript Bibliothek mit den Kernfunktionen von Ethereum aufgeteilt in Module wie VM, Blockchain, Block, Trie oder Ethash. Bibliothek kann mit Ausnahme der Netzwerkmodule mit browserify genutzt werden.
EthereumJ	Java	1.2.0-rc1 (H)	Java Bibliothek die in sämtlichen Java/Scala Projekten integriert werden kann. Unterstützt CPU Mining für das Testnetz und private Netze jedoch nicht wirtschaftlich im Ethereum Hauptnetz. Mit Version 1.2.0-rc1 ist ein Release Candidate (RC) veröffentlicht worden.

ethereumH	Haskell	0.0.4	ethereumH ist ein Ethereum Client basierend auf der Haskell Plattform. Aktuellste Version ist vom 05.01.2015.
Parity	Rust	1.0.0-rc1 (H)	Parity ist ein vollständiger Ethereum Knoten für Linux/Unix und Mac OSX (später auch für Windows) mit Fokus auf Performance und kleinem Footprint. Mit Version 1.0.0-rc1 ist ein Release Candidate (RC) veröffentlicht worden.

TAB. 19: ETHEREUM CLIENTS - STAND 17.03.2016 [ET16T]

Clients in Tab. 19 die kompatibel mit der Homestead Version des Ethereum Protokolls sind werden hier mit einem (H) neben der Version gekennzeichnet. Alle Clients bis auf Parity, ethereumH und Ethereum(J) werden von der Ethereum Fundation entwickelt. Ethereum(J) und ethereumH werden von ConsenSys [Co16b] entwickelt, die zusammen mit Canonical und BlockApps mit uPort und Nimbus an einer Dapp für Ubuntu Geräte zur Identifikation über den biometrischen Fingerabdruck arbeiten. BlockApps [Bl16a] verwendet selbst als Client mit ethereumH aus Tab. 19 die Haskell Implementierung und bietet mit STRATO eine kommerzielle Lösung der Ethereum Blockchain. Darüber hinaus bietet Microsoft (Azure) in Zusammenarbeit im November 2015 mit ConsenSys die Blockchain as a Service (BaaS) und integriert Solidity in seine Entwicklungsumgebung Visual Studio. IBM verwendet Ethereum in seinem IoT-Konzept ADEPT. [Gr15], [Pa15]

Eine mobiler Client ist für C++ und Java (EthereumJ aus Tab. 19) adaptiert für Android in Entwicklung [Su16]. Der Client Parity aus Tab. 19 wird von ETHCORE entwickelt. In den beiden Unternehmen ConsenSys und ETHCORE sind Mitglieder des ETHDEV Teams wie Gavin Wood oder Vinay Gupta u.a. als Gründer beteiligt. Im Februar 2016 wurde von Gavin Wood ein Benchmark [Et16n] zur Messung und Vergleich der Performance unterschiedlicher Clients durchgeführt. Jeder Client musste die ersten 1.000.000 Blöcke des Hauptnetzes Frontier verarbeiten. Gemessen wurden der Speicherverbrauch und die Zeit für die Blockverarbeitung sowie die durchschnittliche CPU-Auslastung. Ergebnisse sind in Abb. 18 dargestellt wobei geringere Werte eine bessere Performanz anzeigen.

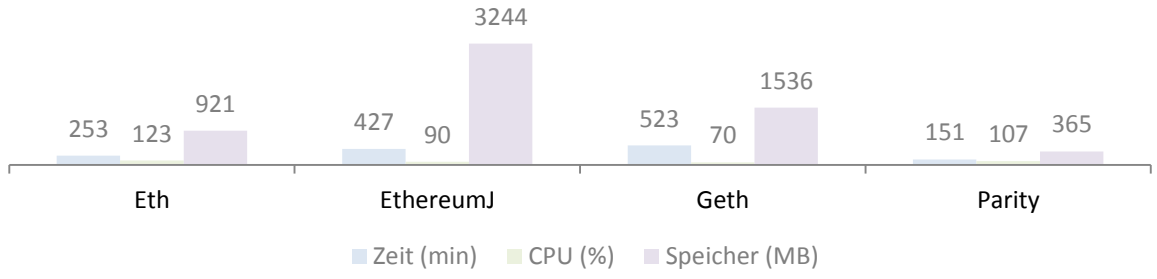

ABB. 18: BENCHMARK FÜR ETHEREUM CLIENTS [ET16N]

Anhand der Messergebnisse aus Abb. 18 lässt sich ableiten, dass der Parity Client zwar die zweitgrößte durchschnittlichste CPU-Auslastung besitzt, aber mit 151 Minuten und 365 Mbyte Speicher der schnellste Client mit dem geringsten Speicherverbrauch ist. Außerdem ist zu erkennen, dass die CPU-Auslastung mit der Blockverarbeitung korreliert. Darüber

hinaus untersuchte Wood mit einem weiteren Benchmarks langlebige (State und Speicher) und kurzlebige Merkle Patricia Bäume (Transaktionen und Bestätigungen). Dieser wurden mit den Clients bzw. Sprachen C++, CPython, PyPy (alternative Python Implementierung), Go, Pure JS (JavaScript) und Parity durchgeführt. Abb. 19 zeigt die Messergebnisse.

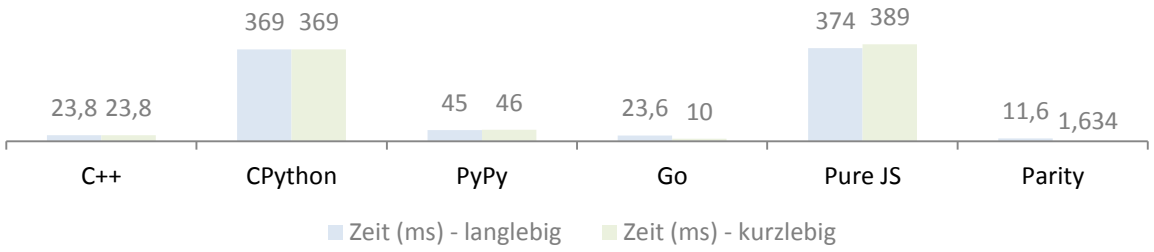

ABB. 19: BENCHMARK FÜR MERKLE PATRICA BÄUME [ET16N]

Zum einen wurde ein langlebiger Patrica Baum mit 1.000 `32 Byte` KV-Paaren aufgebaut, wo nach neun eingefügten Paaren der SHA3 Root Hash neu berechnet wird. Zum anderen wurde ein kurzlebiger Merkle Patrica Baum mit 1.000 `32 Byte` KV-Paaren aufgebaut, wo bei jedem eingefügten Paar der Root ermittelt wird. [Et16n] Anhand der Messergebnisse aus Abb. 19 lässt sich ableiten, dass Pure JS und CPython bzgl. der Verarbeitung von lang- und kurzlebigen Bäumen die längste Zeit benötigen. Die Zeiten von C++, CPython, PyPy und Pure JS unterscheiden sich zwischen lang- und kurzlebigen Bäumen kaum voneinander, während Go mit 10ms und Parity mit 1,634ms hier für kurzlebige Merkle Patricia Bäumen deutlich weniger Zeit benötigen.

Mit der Homestead Version des Ethereum Protokolls wurden die Clients aus Tab. 19 über EIP-2, EIP-7 und EIP-8 (Ethereum Improvement Proposal) angepasst und veröffentlicht.

1. **EIP-2.** Hard Fork von Frontier auf Homestead bei Blocknummer 1.150.000. Eine Transaktion wurde von 21.000 auf 53.000 `GAS` erhöht. Eine Sicherheitslücke bei Transaktionssignaturen wurde behoben. Bei `OUTOFGAS` während der Erstellung eines Contract wird kein leerer Contract mehr hinterlassen. Die Berechnung der `difficulty` wurde verändert, so dass die Zeit zwischen zwei aufeinanderfolgenden Blöcken von 17 auf 13 Sekunden im Durchschnitt reduziert wurde und die Wahrscheinlichkeit für Soft Forks verringert. [Bu15b]
2. **EIP-7.** MIT `DELEGATECALL` wurde eine neue Operation eingeführt analog zum bestehenden `CALLCODE` nur mit der Weitergabe von Sender und Wert des aufrufenden Codes. [Bu15c]
3. **EIP-8.** Die Vorwärtskompatibilität der Subprotokolle devp2p, RLP und RLPx TCP wurde nach dem Robustheitsgrundsatz [Po80] verbessert. [La15]

Am 17.03.2016 waren mit MIST, go-ethereum, cpp-ethereum und ethereumjs-lib vier Clients vorhanden, die mit Homestead kompatibel sind und über ein RC hinaus veröffentlich wurden [Et16t]. Angekündigt wurden die beiden vollständigen Clients C++ (eth) und Go (geth) mit der Version aus Tab. 19, wobei der MIST Client beide beinhaltet. Für den Go Client soll mit Version 1.4 ein größeres Feature Release u.a. mit Key Management erscheinen. Dieses

Release wurde vom Homestead Update getrennt, um Änderungen auf ein Minimum zu reduzieren und Debugging zu erleichtern. Die neue Operation `DELEGATECALL` soll in Solidity nutzbar sein, ist aber ebenfalls noch nicht veröffentlich worden. Der Go Client kann sich mit dem Command `geth attach` mit einem laufenden C++ Client verbinden. [Wi16b] Der Ethereum Node Explorer [Et16p] gibt eine aktuelle Übersicht verwendeter Clients, Versionen, Betriebssysteme in einer geografischen Verteilung mit Historie. Der Go Client wird hiernach mit über 90% im Ethereum Hauptnetz am Meisten genutzt.

Die Auswahl des Ethereum Clients wird innerhalb des nächsten Kapitels im Kontext der Implementierung einer Dapp zusammen mit dem Use Case, dem Solidity Contract und der JavaScript App beschrieben. Unter dem Aspekt Tools und Frameworks wird ein Docker Container mit dem Client und einer vollständigen Ethereum Umgebung erstellt. Zuvor wird der Use Case für die Dapp in dem Bereich Smart Mobility konzipiert, die über den Ethereum Client deployt und ausgeführt werden soll.

5 Implementierung einer Dapp

Nachdem in dem vorangegangenen Kapitel Ethereum als Plattform für die Ausführung von Smart Contracts und Dapps beschrieben wurde, soll eine Dapp mit Ethereum als Prototyp konzipiert und implementiert werden. Dabei wird zunächst ein Use Case im Kontext von Smart Mobility im Modell nach Manville erstellt. Anschließend wird eine Auswahl eines Ethereum Clients sowie weiteren Tools, APIs und Frameworks getroffen. Mit diesen wird erst der Contract und danach die JavaScript Anwendung implementiert und deployt. Ausführung und Test der Dapp wird später in Kapitel 6 beschrieben.

5.1 Use Case „Parking Places“

An dieser Stelle wird ein Use Case für den Dapp Prototypen konzipiert. Dieser soll zum einen als Beispiel für die Erstellung von Smart Contracts und Dapps mit Ethereum dienen. Zum anderen soll der Kontext einer Smart City nach dem Modell von Manville aus Kapitel 2.4 und 2.5 berücksichtigt werden.

Die meisten und erfolgreichsten Smart City Lösungen gibt es nach Manville et al. [Ma14] im Bereich nachhaltige, intelligente, innovative und sichere Verkehrssysteme bzw. innerhalb der Eigenschaft Smart Mobility. Ein Referenzprojekt ist das Projekt „The Smart Parking Network Barcelona“. Analog hierzu wurde die Situation von öffentlichen Parkplätzen mit Parkautomaten betrachtet, die für eingeworfenes Münzgeld ohne Ausgabe von Wechselgeld einen Parkschein mit entsprechender Parkdauer auswerfen siehe Abb. 20 aus [Wi15]. Weitere Automaten und Aspekte rund um das Thema Parkraumbewirtschaftung wurden nicht näher betrachtet.

ABB. 20: PARKSCHEINAUTOMAT UND PARKSCHEIN [WI15]

Der Parkschein aus Abb. 20 wird hinter die Windschutzscheibe gelegt, damit nachgewiesen werden kann, dass der Parker zu gegebener Zeit berechtigt ist dort zu parken. Für solche Parkplätze gibt es keine Auskunft, ob und wie viele Slots belegt oder frei sind. Oft werden mehrere Parkplätze angefahren, bis ein freier Slot gefunden wurde. Eine Verlängerung des Parkscheins ist nicht möglich, so dass vor Ort ein neuer Parkschein gelöst werden muss. Eine Dapp soll Parktickets in der Blockchain speichern, öffentliche Parkplätze verwalten und den Anwendern die Möglichkeit geben ein Parkticket zu lösen und zu verlängern unabhängig

vom Ort. Hierfür wurde mit „Parking Places“ eine Dapp konzipiert, die in Abb. 21 als Use Case mit drei Akteuren dargestellt wird.

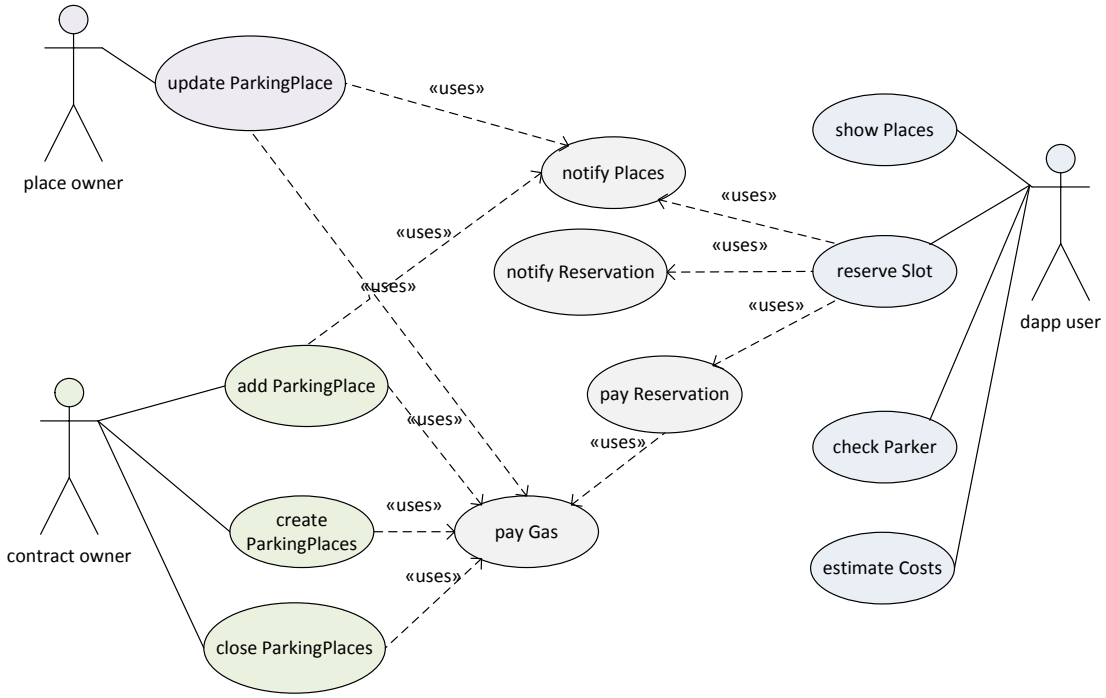

ABB. 21: DAPP USE CASE DIAGRAMM "PARKING PLACES"

Neben der Verwaltung, Reservierung und Bezahlung von Parkplätzen und ihren Slots erhält der Anwender Information über Geoposition und Anzahl freier Slots. Beim Hinzufügen, Reservieren oder Aktualisierung eines Parkplatzes wird ein Event getriggert, da sich die Anzahl freier Slots eines Parkplatzes geändert haben könnte. Das Anzeigen von Parkplätzen und ihren Slots, die Prüfung eines Parkenden und die Kostenberechnung erfordern für ihre Ausführung keine Transaktion, da nur Informationen abgerufen werden. Dagegen benötigen die restlichen Anwendungsfälle wie das Reservieren eines Slots eine Transaktion und somit das Bezahlen von `GAS`, da hier der State verändert wird.

Der in Abb. 21 dargestellte Use Case dient als Vorlage und Entwurf für den Contract bzw. den gesamten Dapp Prototypen. Als Referenz wurde die Stadt Oldenburg gewählt, die einen Stadtplan [St16] im Internet anbietet der u.a. für den Innenstadtbereich 18 Parkplätze unter Angabe von Stellplätzen (Slots), Kosten und Öffnungszeiten enthält. Der Dapp Prototyp „Parking Places“ soll jeden Parkplatz über seine Geoposition als Markierung auf einer Karte abbilden, zusätzliche Informationen und die Möglichkeit der Reservierung bieten. Darüber hinaus sollen Aktualisierungen der Parkplätze über die Blockchain visualisiert werden.

Auf eine konkrete UI Beschreibung über Mock-up’s oder Wireframes wird an dieser Stelle verzichtet, da es sich bei der Dapp um eine prototypische Umsetzung mit Beispielcharakter handelt. Daher fanden auch Aspekte wie Usability (DIN EN ISO 9241-11), User Experience (DIN EN ISO 9241-210) oder der mobile Kontext keine Berücksichtigung. Im folgenden Kapitel werden Tools und Frameworks für die Implementierung des Dapp-Prototyp „Parking Places“ beschrieben und ausgewählt.

5.2 Tools und Frameworks

Der Use Case „Parking Places“ für die Dapp wurde im vorherigen Kapitel beschrieben. Hier wird nun eine Auswahl des Clients aus Kapitel 4.4 getroffen sowie Tools und Frameworks beschrieben, mit denen die Dapp in den folgenden Kapiteln implementiert, deployt, evaluiert und ausgeführt werden soll. Hierfür werden Docker Container mit JavaScript Umgebung und Ethereum erstellt.

Die Blockchain basiert auf einer verteilten, dezentralen und kryptografisch verketteten Datenbank. Docker ist eine Open Source Plattform mit containerbasierter Virtualisierung für über das Entwickler und Systemadministratoren verteilte Anwendungen als Container automatisiert bauen, ausliefern und betreiben können. [Tu15c] Docker soll im Rahmen des Prototyps für eine effiziente und leichtgewichtige Entwicklung in Build und Ausführung von Dapps unter verschiedenen Umgebungen sowie als Sandbox-Umgebung für Entwicklung, Tests und Lernszenarien eingesetzt werden. Dabei sollen jeweils über mit Automatic Build verbundene, öffentliche Repositories GitHub (Quellcodeverwaltung) und DockerHub (Verwaltung von Containerimages) die Entwicklung, die Bereitstellung und die Ausführung von Dapps (Beispiele und Prototyp) im Sinne der Reproduzierbarkeit, Nachvollziehbarkeit und Dokumentation beschreiben. Vorteile dieses Vorgehens sind nach Boettinger [Bo15]:

1. **Ein Docker Image beinhaltet alle erforderlichen Abhängigkeiten.** Identifikation von Abhängigkeiten, die erforderlich sind um den Code auszuführen.
2. **Ein Docker Image und/oder Container kann 1:1 exportiert und importiert werden.** Aktualisierungen von einzelnen Abhängigkeiten führen dazu, dass der Code nicht mehr in der Originalumgebung ausgeführt wird, was möglicherweise das Ergebnis verfälscht. Stichpunkte hierbei sind Versionsupdate und Bug fixes.
3. **Ein Dockerfile beschreibt präzise die Installation eines gesamten Images.** Unpräzise Anleitung erschwert die Einrichtung der Umgebung und die Ausführung des Codes bzw. führt ggf. auch zu falschen Ergebnissen.
4. **Docker ist einfach zu lernen und passt nahezu nahtlos in bestehende Workflows.** Komplexes Setup und Einrichtung erfordern ggf. eine zeitaufwendige Einarbeitung in verschiedene Tools und Frameworks.

Eine Dapp benötigt eine Verbindung zum P2P-Netzwerk über einen Ethereum, um Contracts ausführen zu können. Für Erstellung und Ausführung von Contracts werden ETH benötigt, welche über Mining generiert werden. Die prototypische Umsetzung der Dapp erfolgt innerhalb des Testnetzwerks „morden“, welches mit geringerer Schwierigkeit und Hashrate das Mining auch für normale Desktop-PCs in kurzer Zeit ermöglicht. Der erste Schritt für den Prototyp ist die Erstellung eines Docker Containers mit Ethereum Client. Hierüber sollen Accounts erstellt, ETH per Mining generiert sowie Contracts kompiliert, deployt und ausgeführt werden.

Als Client wurde Geth aus folgenden Gründen ausgewählt:

- Neben Windows, Unix und MacOS X wird auch der Raspberry Pi unterstützt.
- Vollständiger Client als Homestead Version verfügbar.
- CPU-Mining wird unterstützt (für Generierung von ETH im Testnetz).
- Mit über 90% am meist genutzter Client.
- Integriert in Multi Signatur Wallet und Dapp Browser MIST.
- Per geth attach auch möglich den C++ Client zu nutzen.
- Enthält JavaScript Runtime Environment (JSRE) interaktiv und skriptbasiert.
- JSRE ist interaktiv über REPL (Read, Evaluate & Print Loop) Konsole.
- web3 JavaScript Dapp API und admin API (Management) über JSRE verfügbar.
- Stellt JSON-RPC (Remote Procedure Call) als Endpoint zur Verfügung.
- Standardisierte JS Object Notation Crockford [Cr06] in RFC4627 wird verwendet.
- Sehr gute CPU-Auslastung siehe Abb. 18 im Standardbetrieb (Nicht Mining).
- Sehr gute Verarbeitungszeit für Merkle Patrica Bäume siehe Abb. 19.
- Angekündigtes großes Feature Release 1.4.0.

Der Solidity Compiler `solc` muss neben `geth` installiert werden. Abb. 22 zeigt den Container „geth-node“ (grau) mit Abhängigkeiten, Ethereum (grün) und Endpoints (blau).

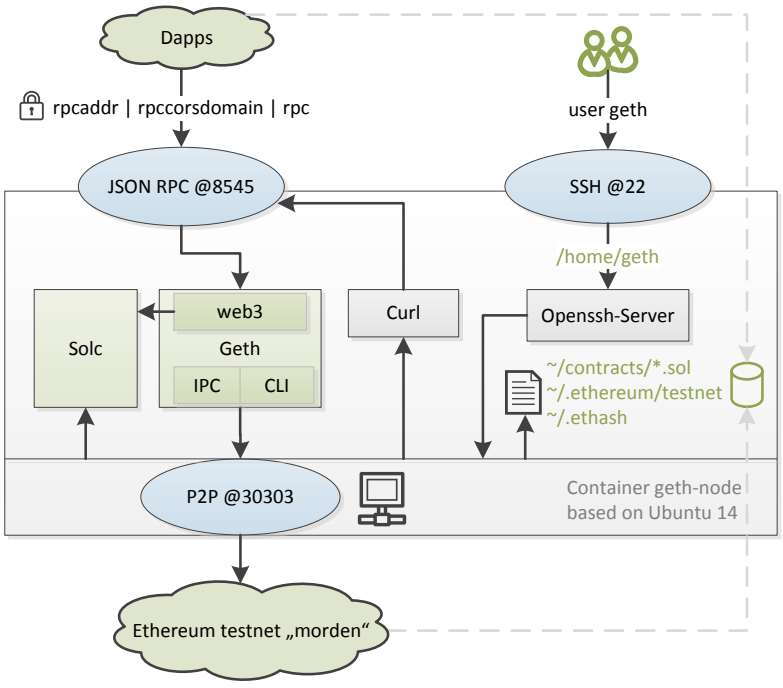

ABB. 22: DOCKER-CONTAINER "GETH-NODE" ALS ETHEREUM CLIENT

Der Container aus Abb. 22 basiert auf Ubuntu und ist von außerhalb mit dem Benutzer geth über einen Openssh-Server per SSH (Secure SHell) erreichbar, der auf alle Komponenten Zugriff hat. Der Ethereum Client `geth` stellt mit seiner JSON RPC API `web3` einen weiteren Endpoint zur Verfügung mit dem sich Dapps außerhalb des Containers verbinden können. Beim Starten des Clients können per `--rpc` Parameter Domain und Host sowie Module eingeschränkt werden. Mit `curl -X POST --data '{…}' localhost:8545` kann die JSON RPC API ebenfalls angesprochen werden. Solidity Contracts können über `solc`

entweder direkt oder über die JSON RPC API per `eth.compile.solidity` kompiliert werden. Neben der RPC API bietet geth zusätzlich eine dateibasierte IPC API (Inter Process Communication) (`~/.ethereum/geth.ipc`) die ebenfalls JSON verwendet und dieselben Funktionen unterstützt. Der Client verbindet sich mit per TCP über den Port 30303 zum Testnetzwerk „morden“ durch den Parameter `--testnet`. Im Homeverzeichnis (`~`) des Benutzers geth befinden sich Solidity Contracts mit der Dateiendung `*.sol` und alle Daten von Ethereum des Testnetzwerks unter `.ethereum/testnet` wie z.B. die Schlüssel der eigenen Accounts unter `keystore` oder die Blockchain unter `chaindata`. Daten die im Rahmen des Mining Algorithmus Ethash generiert werden liegen neben `.ethereum` unter `.ethash`. Das Ethereum Netzwerk besitzt ebenso wie Dapps mit ihren Contracts dieselben Daten der Blockchain daher führen in Abb. 22 zwei gestrichelte Pfeile in den Container um dies zu verdeutlichen. Der Container wird im Anhang A.1 über sein Dockerfile beschrieben.

Eine Dapp besteht neben seinen Contracts aus einem GUI welches sich über die JavaScript Dapp API `web3` per JSON RPC mit dem Ethereum Client verbindet und Contracts über Nachrichten und Transaktionen ansprechen kann. Die Befehle sind dieselben, die per `curl` oder in der `geth console` ausgeführt werden [Et16q]. Im Ethereum Wiki [Et16m] wird die Integration der `web3.js` mit NPM, Bower, Vanilla JS und Bower beschrieben. NPM ist ein Paketmanager für JavaScript im Bundle mit der JavaScript Laufzeitumgebung NodeJS. Bower ist ein Paketmanager für webbasierte Browserpakete (Frontend) und benötigt NPM und NodeJS zur Installation und Ausführung. Vanilla JS ist ein JavaScript App Framework für unterschiedliche Plattformen. Meteor ist eine JavaScript App Plattform, welche von Ethereum [Et16o] aus folgenden Gründen empfohlen wird:

- Vollständige JavaScript Implementierung mit Tools zur Unterstützung von Single Page Apps (SPA) wie z.B. Templating und Bundling.
- Entwicklungsumgebung mit Live-Reload, CSS Injection und Precompiled Support wie z.B. mit SASS (Syntactical Awesome Style Sheets)
- Frontend-Code kann in eine einzelne HTML-Datei, JS- und CSS-Datei zusammen mit Assets zusammengefasst (Bundle) werden.
- Unterstützt reaktive Programmierung ähnlich AngularJS und React.
- Besitzt mit Minimongo eine In-Memory Datenbank als MongoDB Emulator für den Webbrowser (mit indexDB, webSQL - Structured Query Language - und local storage)

Das Prinzip der Dezentralisierung von Dapps wird durch SPA unterstützt. Diese müssen nicht zentral gehostet werden und können direkt über den Webbrowser im `file://` Protokoll oder den Dapp-Browser im `eth://` Protokoll aufgerufen werden. Meteor unterstützt dieses Prinzip und bietet sechs Pakete für Ethereum, die einige Funktionen und Designs für Dapps beinhalten [Pe16]. Der zweite Schritt für die Implementierung einer Dapp bzw. eines Dapp Prototyps ist die Erstellung eines Container mit einer JavaScript Laufzeitumgebung, Paketmanagement sowie der Verlinkung des Containers „geth-node“ mit dem Ethereum

Client über die JSON RPC API. Hierfür wurde wie Abb. 23 zeigt Meteor und NodeJS mit NPM als JavaScript Umgebung (grün) in dem Container „meteor-nodejs“ (grau) integriert, der mit seinen Abhängigkeiten über seine Endpoints (blau) erreichbar ist.

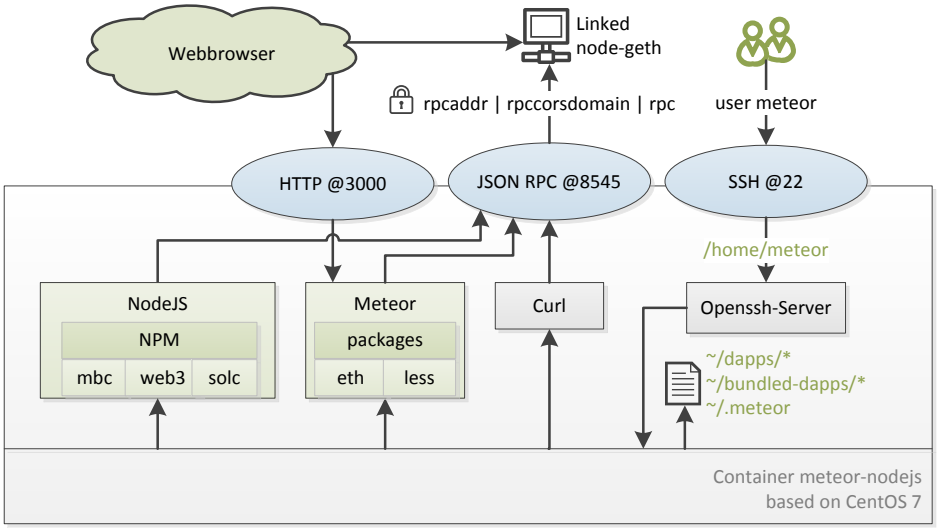

ABB. 23: DOCKER-CONTAINER "METEOR-NODEJS" ALS JAVASCRIPT-UMGEBUNG FÜR DAPPS

Der Container aus Abb. 23 basiert auf CentOS und ist von außerhalb mit dem Benutzer meteor über einen Openssh-Server per SSH erreichbar ist, der auf alle Komponenten Zugriff hat. Über die Container Verlinkung zu „geth-node“ ist die JSON RPC API verfügbar. Ein Webbrowser kann per `http://` die Meteor Dapp aufrufen und benötigt selbst Zugriff auf den Container „geth-node“ da die Dapp den Ethereum Node clientseitig anspricht. Pakete wie less und die beschriebenen sechs Ethereum Pakete können über Meteor installiert werden. Meteor und seine Pakete liegen im Homeverzeichnis (~) des Benutzers meteor unter `.meteor`. Parallel dazu liegen die Dapps unter `dapps/*` und zur SPA gebundenen Dapps unter `bundled-dapps/*`. Eine SPA kann per SSH heruntergeladen werden und über den Webbrowser im `file://` Protokoll aufgerufen werden. Der Container „geth-node“ mit dem laufenden Ethereum Client ist aber erforderlich. Neben Meteor ist in dem Container mit NodeJS eine weitere JavaScript Umgebung vorhanden, da nur über dessen Paketmanager NPM das Paket `mbc` (meteor-build-client) installiert werden kann, um Dapps als SPA zusammenzufassen. Darüber hinaus bietet NodeJS eine interaktive JavaScript Konsole, die um die Pakete `web3` und `solc` erweitert wurde, um auch hier den Zugriff auf Ethereum und Entwicklung von Dapps zu ermöglichen. NodeJS und Meteor sind beide integriert in WebStorm der JavaScript IDE (Integrated Development Environment) von JetBrains. Der Container wird im Anhang A.2 über sein Dockerfile beschrieben.

Für die in Kapitel 5.1 beschriebene Kartendarstellung wird die Google Maps JavaScript API integriert, für die mehrere Pakete in Meteor verfügbar sind. Im nächsten Kapitel wird die Implementierung des Smart Contract vorgestellt und seine Funktionsweise im Kontext der Ethereum Blockchain beschrieben. Einrichtung und Ausführung der Dapp über die Docker Container werden später nach der Evaluation in Kapitel 6.3 beschrieben.

5.3 Solidity Contract

Kapitel 4.3 wurde Solidity als Programmiersprache und der Online-Compiler und seine Laufzeitumgebung in Echtzeit vorgestellt. An dieser Stelle wird der Contract „ParkingPlaces“ implementiert, der über den unteren Link aus dem GitHub Repository geladen werden kann. Dieser enthält die verwendete Solidity Version und Einstellungen zur Optimierung des Codes. Im letzten Kapitel wurde aus Ethereum Client und Solidity Compiler ein Docker Container erstellt. Dieser wurde um Git erweitert, um Contracts aus Repositories laden zu können.

*http://chriseth.github.io/browser-solidity/#**gist**=99a7e5b15d564d77fd61e4301afcac45&**version**=soljson-v0.3.1-2016-04-12-3ad5e82.js&**optimize**=true*

Contract Code und NatSpec (Natural Specific Format) Dokumentation befinden sich im Anhang B. Unter der Adresse `0xad3d7d21862dfa1f9d91569240a9ed06ac276b4d` ist der Contract Code bei den Blockchain Explorern EtherChain [Et16c] und EtherCamp [Et16d] im Testnetz „morden“ verifiziert und besteht aus vier State Variablen:

1. Adresse des „contract owner“ als Controller
2. Kosten pro Block (in Wei)
3. Dynamisches Array mit Parkplätzen (eigener Typ)
4. Mapping zwischen Adressen und Parkplätzen

Die ersten drei Variablen sind `public` und erhalten automatisch Getter-Methoden, die nicht implementiert werden müssen. Ein Platz hat eine eigene Struktur bestehend aus der Adresse „place owner“, einem Namen, seinen Koordinaten (Längen- und Breitengrad) sowie einem weiteren dynamischen Array an Slots. Ein Slot besitzt eine eigene Struktur und enthält die Adresse des Parkenden und Blocknummer als Parkende. Controller und Kosten werden bei der Erstellung des Contract initial gesetzt. Der Controller ist der Sender der Transaktion und die Blockkosten werden dem Konstruktor als Argument übergeben. Das Mapping wird intern genutzt um über die „place owner“ Adresse (Key) den zugehörigen Platz (Value) zu ermitteln. Auch der Typ String ist ein dynamisches Array aus UTF-8 enkodierten Zeichen. Eine Übersicht der Datenstruktur des Contract zeigt Tab. 20.

Größe	Datentyp	Variable	Beschreibung
`4 Byte`	`address`	controller	Adresse des Besitzer des Contract
`32 Byte`	`uint256`	blockCosts	Kosten pro Block für eine Reservierung
`x Byte`	`[]`	places	Dynamisches Array mit Parkplätzen
`x Byte`	**`struct`**	**Place**	**eigene Struktur eines Parkplatzes**
`20 Byte`	`address`	owner	Adresse des Besitzer des Parkplatzes
`x Byte`	`string`	name	Name des Platzes

x Byte	string	latitude	Breitengrad der Geo Koordinate
x Byte	string	longitude	Längengrad der Geo Koordinate
x Byte	[]	slots	Dynamisches Array mit Slots
52 Byte	**struct**	**Slot**	**eigene Struktur eines Parkslots**
20 Byte	address	parker	Adresse des Parkenden
32 Byte	uint256	reservedBlock	Blocknummer des Parkende

TAB. 20: STATE VARIABLEN IN CONTRACT "PARKINGPLACES"

Die Struktur aus Tab. 20 bezeichnet den Speicher des Accounts und kann in EtherCamp eingesehen werden kann. Die Größe des Speichers und die GAS Kosten hängen von der Anzahl der Parkplätze, der Größe der `string` Felder eines Platzes und seiner Anzahl von Slots ab. Mit festen Array Größen und `byte` Arrays statt `string` könnte die Speichergröße reduziert und GAS Kosten gesenkt werden. In Tab. 20 sind die `struct` Typen `Place` und `Slot` fett markiert. `struct` ist eine Gruppierung von mehreren Variablen und wird von den dynamischen Arrays `places` und `slots` verwendet. Da die Variablen `controller`, `blockCosts` und `places` als `public` deklariert wurden, sind sie Teil des ABI (siehe Anhang B.4) und können über Nachrichten abgerufen werden.

In den folgenden Abschnitten wird der Contract mit den hier beschriebenen Variablen über Modifier, Events und Funktionen vervollständigt. Zuletzt wird gezeigt wie der Contract in der Ethereum Blockchain deployt wird.

5.3.1 Modifier und Events

Innerhalb des Contracts werden nach den State Variablen Modifier und Events definiert, die von Funktionen verwendet werden und Variablen prüfen oder als Log Event ausgeben.

Modifier werden durch das Schlüsselwort `modifier` eingeleitet und können Parameter entgegen nehmen. Mit Modifier können bestimmte Bedingungen vor der Ausführung einer Funktion geprüft werden. Je nachdem ob eine Bedingung erfüllt wurde oder nicht, kann die Semantik der Funktion verändert werden. Eine Funktion kann mehrere Modifier besitzen, die nach den Parametern aufgeführt werden. Derselbe Modifier kann in unterschiedlichen Funktionen verwendet werden.

Alle im Contract definierten Modifier erzeugen eine Ausnahme wenn ihre Bedingung nicht erfüllt ist. Wird die Bedingung erfüllt wird durch ein _ der Body der entsprechenden Funktion ausgeführt. Mit diesen Modifier werden u.a. die Einschränkungen der Akteure „contract owner“ und „place owner“ Anwendungsfälle auszuführen aus Abb. 21 abgebildet. Nur der „contract owner“ darf Parkplätze hinzufügen oder den Contract schließen und nur der „place owner“ darf seinen Platz aktualisieren. Beide dürfen jedoch keine Reservierung durchführen. Ein weiterer Modifier prüft ob eine Transaktion Value (Wei) besitzt, da dies nur innerhalb einer Reservierung erforderlich ist. In Tab. 21 werden alle Modifier des Contract unter Angabe ihrer Parameter und Bedingung beschrieben.

Name	Parameter	Bedingung
isController		prüft ob der Sender der Transaktion der Controller ist
isOwner	`address`	prüft ob der Sender der Transaktion die Adresse besitzt
hasValue		prüft ob die Transaktion einen Value (Wei) > 0 besitzt
allowReservation	`address`	prüft ob der Sender der Transaktion die Adresse besitzt oder der Controller ist

TAB. 21: MODIFIER IN CONTRACT "PARKINGPLACES"

Mit Events wird das Logging in Ethereum durch Bloom Filter und Transaktionsbestätigungen abgebildet. Nur erfolgreiche Transaktionen besitzen Eventlogs die z.B. im Blockchain Explorer EtherChain [Et16c] zu jeder Transaktion im gleichnamigen Reiter angezeigt werden. Der Blockchain Explorer EtherCamp [Et16d] zeigt die vollständige Transaktionsbestätigung. Events werden durch das Schlüsselwort `event` eingeleitet und können Parameter besitzen die in der Bestätigung der Transaktion gespeichert werden. Events werden innerhalb des Contract verwendet um über neue oder aktualisierte Parkplätze sowie über Reservierungen und dessen Transaktionen zu benachrichtigen. Sie bilden die „notify“-Anwendungsfälle aus Abb. 21 ab und werden in Tab. 22 beschrieben.

Name	Parameter
PlaceAdded	`address` place, `string` name, `string` latitude, `string` longitude
SlotsAdded	`address` place, `uint` amount
Reservation	`address` place, `address` parker, `uint` reservedBlock
Transaction	`address` from, `address` to, `uint` transferred, `uint` refund, `uint` block

TAB. 22: EVENTS IN CONTRACT "PARKINGPLACES"

Der Contract benachrichtigt über einen neuen Parkplatz mit allen Variablen des Typ `Place` als Parameter innerhalb des Events `PlaceAdded`. Ein neuer Parkplatz hat initial keine Slots. Das Hinzufügen von Slots wird vom Contract über das Event `SlotsAdded` unter Angabe der Adresse des Parkplatzes und der Anzahl der hinzugefügten Slots benachrichtigt. Das Event `Transaction` wird vom Contract bei einer Transaction mit Value (Wei) und der Funktion `reserveSlot` kurz vor dem Event `Reservation` getriggert. Beide Events besitzen als Parameter die Adresse des Parkenden (`parker` und `from`) und die des Parkplatzes (`place` und `to`). Die weiteren Parameter beziehen sich auf das Parkende (`reservedBlock`), das transferierte Value (`transferred`) und das zurückerstattete Value (`refund`) jeweils in Wei sowie die Anzahl der reservierten Blocks (`block`). Durch Angabe des Schlüsselworts `indexed` können bis zu drei Parameter indexiert werden, so dass danach als „Topic“ gesucht und gefiltert werden kann. Events können mit dem Schlüsselwort `anonymous` gekennzeichnet werden, so dass über den Event Namen nicht gefiltert werden kann. Alle Events sind im ABI (siehe Anhang B.4) enthalten.

Funktionen die Modifier verwenden und Events aufrufen werden im Folgenden beschrieben.

5.3.2 Funktionen

In diesem Abschnitt werden alle Funktionen des Contracts beschrieben. Diese werden unterschieden in solche die über Nachrichten angesprochen werden können und solche die eine Transaktion erfordern, da sie Variablen innerhalb des States über die EVM verändern.

Der Contract besitzt mit dem Konstruktor, einem Fallback und `close` drei besondere Funktionen. Der Konstruktor `ParkingPlaces` ist `public` und wird mit dem Parameter `blockCosts` initial und einmalig bei Erstellung des Contract aufgerufen. Die Fallback Funktion ist `public` und besitzt keinen Namen oder Parameter. Sie hat den Modifier `hasValue`, so dass jede nicht definierte Transaktion an den Contract eine Ausnahme erzeugt. Bei einer Ausnahme (`throw`) wird die Transaktion zurückgerollt und kein Value transferiert. Die Funktion `close` hat den Modifier `isController` und führt ein `selfdestruct(controller)` aus, der den Contract über die `SUICIDE` Operation schließt. Hiernach ist der Contract Account Speicher und Code gelöscht und Value wird an den `controller` zurückerstattet. Weitere Funktionen, die eine Transaktion erfordern, sind in Tab. 23 mit ihren Parametern und Modifier aufgeführt.

Name	Parameter	Modifier
addPlace	`address` owner, `string` name, `string` lat, `string` long	isController, hasValue
addSlots	`address` owner, `uint` amount	isOwner, hasValue
reserveSlot	`address` owner, `uint` untilBlock	allowReservation
payReservation	`address` owner, `address` parker, `uint` time	
getNextFreeSlot	`address` owner	

TAB. 23: "TRANSACTION"-FUNKTIONEN IN CONTRACT "PARKINGPLACES"

Die beiden Funktionen `payReservation` und `getNextFreeSlot` aus Tab. 23 sind `internal` d.h. sie können nur vom Contract selbst oder abgeleiteten Contracts angesprochen werden und sind nicht im ABI (siehe Anhang B.4) enthalten. Beide werden innerhalb der Funktion `reserveSlot` aufgerufen und besitzen keine Modifier. Alle anderen Funktionen aus Tab. 23 sind `public`. Die Funktion `addPlace` triggert `PlaceAdded`, die Funktion `addSlots` triggert `SlotsAdded` und die Funktion `reserveSlots` triggert die beiden Events `Transaction` und `Reservation`. Neben den „Transaction“-Funktionen aus Tab. 23 hat der Contract „Message Call“-Funktionen, die über das Schlüsselwort `constant` definiert sind und über Nachrichten aufgerufen werden. Die verwendeten Funktionen im Contract sind `public` (Default) und in Tab. 24 aufgeführt.

Name	Parameter	Returns
existsPlace	`address` owner	`bool` exists
getSlotCount	`address` owner	`uint` count
getFreeSlotCount	`address` owner, `uint` atBlock	`uint` count

getNextFreeBlock	`address` owner	`uint` block
calculateEstimatedCosts	`uint` atBlock, `uint` toBlock	`uint` costs
getReservedBlock	`address` owner, `address` parker	`uint` block, `uint` index

TAB. 24: "MESSAGE CALL"-FUNKTIONEN IN CONTRACT "PARKINGPLACES"

Da Aufrufe der „Message Call“-Funktionen aus Tab. 24 keine Transaktion besitzen und nicht in der EVM ausgeführt werden, gibt es keinen Eintrag im Blockchain Explorer im Gegensatz zu Aufrufen der „Transaction“-Funktionen aus Tab. 23. Der Blockchain Explorer EtherCamp [Et16d] zeigt den Funktionsnamen und die geänderten Werte der Variablen des Account Speicher neben einem ausführlichen VM-Trace und Eventlogs. Alle öffentlichen Funktionen sind in der NatSpec Dokumentation für Benutzer und Entwickler (Anhang B.2 und B.3) sowie mit Events und Variablen im ABI (Anhang B.4) beschrieben.

Im Folgenden wird das Deployment des Contract, die Initialisierung mit Daten über die Funktionen `addPlace` und `addSlots` und die Erstellung von Ethereum Accounts mit JavaScript beschrieben.

5.3.3 Deployment

Für die weitere Entwicklung der JavaScript App im nächsten Kapitel muss der Contract im Ethereum Testnetz „morden“ deployt werden. Außerdem müssen Parkplätze und Slots über Transaktionen erstellt werden. Für jeden Parkplatz ist ein eigener Account erforderlich, der wie der Account zur Erstellung des Contract Ether besitzen muss. Der Blockchain Explorer EtherCamp [Et16d] bietet die Möglichkeit Ether im Testnetz abzurufen. Für ein besseres Verständnis von Ethereum wurden ETH per Mining selbst generiert. Dies wird auch im GitHub und DockerHub Repository siehe Anhang A.1 beschrieben.

Der vollständige Contract im Anhang B.1 wird durch den Solidity Online Compiler in Echtzeit syntaktisch geprüft. Auch das Interface, die `GAS` Kosten, Größe in `Bytes`, der Bytecode und eine JavaScript für das Deployment werden hier in Echtzeit erstellt. Der Solidity Contract `ParkingPlaces.sol` und JavaScript für Deployment sowie die Initialisierung der Dapp sind im GitHub Repository unter *https://github.com/blakeberg/parking-dapp* im Verzeichnis `contracts` vorhanden. Dem Docker Container „geth-node“ wurde das Package `git` hinzugefügt, um dieses Repository zu klonen. Für die Erstellung von 18 weiteren Accounts für die Parkplätze wurde das JavaScript in Listing 2 erstellt.

```
1  for (i = 1; i < 19; i++) {
2      personal.newAccount("place" + i);
3      eth.sendTransaction({from:eth.accounts[0], to:eth.accounts[i], value: web3.toWei(0.5,
   "ether")});
4      personal.unlockAccount(eth.accounts[i], "place" + i);
5  }
```

LISTING 2: JAVASCRIPT "CREATEPARKINGPLACESACCOUNTS.JS"

Hier und in den folgenden Listings wird die JavaScript API von Ethereum verwendet. Über die JavaScript Konsole des Ethereum Go Client `geth` werden JavaScript Dateien per

`loadScript` geladen und ausgeführt. Das JavaScript aus Listing 2 transferiert jeweils 0,5 ETH vom Account 0 („Main Account“) an den erstellten Account und entsperrt diesen für weitere Transaktionen. Mit dem JavaScript aus Listing 3 wird der Balance aller Accounts angezeigt.

```
1  var i = 0;
2  eth.accounts.forEach( function(e){
3      console.log("eth.accounts["+i+"]: " +  e + " \tbalance: " +
   web3.fromWei(eth.getBalance(e), "ether") + " ether");
4      i++;
5  })
```

LISTING 3: JAVASCRIPT "SHOWBALANCES.JS"

Ist jede Transaktion innerhalb eines Blocks verarbeitet und die Blöcke synchronisiert zeigt das JavaScript aus Listing 3 für Accounts 1-18 ein Balance von 0,5 ETH an und für Account 0 ein Balance um 9 ETH reduziert. Daher sollten über den Main Account mindestens zwei Blöcke per Mining erstellt worden sein bei 5 ETH Vergütung pro Block. Das JavaScript aus Listing 4 für das Deployment des Contract enthält das `ABI` in JSON Format (siehe Anhang B.4) und den `BYTECODE`, die aufgrund ihrer Größe mit einem Platzhalter versehen sind. Die Transaktion wird vom Main Account ausgeführt und per Callback das Ergebnis verifiziert. Der Hash der Transaktion wird direkt ausgeloggt und die Adresse des Contract sobald ein Block die Transaktion verarbeitet hat und synchronisiert wurde, falls kein Fehler aufgetreten ist.

```
1  var _blockCosts = web3.toWei(1, "finney");
2  var parkingplacesContract = web3.eth.contract(ABI);
3  var parkingplaces = parkingplacesContract.new(
4     _blockCosts,
5     {
6       from: web3.eth.accounts[0],
7       data: BYTECODE,
8       gas: 3000000
9     }, function(e, contract){
10     if(!e) {
11       if(!contract.address) {
12         console.log("Contract transaction send: TransactionHash: " +
   contract.transactionHash + " waiting to be mined...");
13       } else {
14         console.log("Contract mined! Address: " + contract.address);
15       }
16     } else {
17         console.log("Error in contract creation: " + e);
18     }
19 })
```

LISTING 4: JAVASCRIPT "PARKINGPLACES.JS"

Da im Container “geth-node” der Solidity Compiler `solc` als Paket geladen und installiert ist, können das `ABI` per `solc --abi ParkingPlaces.sol` und der `BYTECODE` per `solc --optimize --bin ParkingPlaces.sol` aus der Bash-Konsole generiert werden. Weiter kann mit `solc` eine Benutzer- und Entwicklerdokumentation (basierend auf NatSpec siehe Anhang B.2 und B.3) erstellt oder die `GAS` Kosten berechnet werden. Aus den Daten der Stadt Oldenburg [St16] wurden 18 öffentlich kostenpflichtige Parkplätze unter Angabe des Namen, der Lokalisierung und der Anzahl an Stellplätzen ermittelt. Die Geokoordinaten (Längen- und Breitengrad) wurden über Google Maps ermittelt, welche von der Dapp später auch verwendet wird. Für das Erstellen von Parkplätzen innerhalb des

Contract Account Speicher wurde ein JavaScript erstellt, welches für jeden Parkplatz eine Transaktion von dem zugehörigen Account ausführt. Exemplarisch werden zwei dieser Transaktionen aus dem JavaScript `CreateParkingPlaces.js` in Listing 5 dargestellt.

```
1  parkingplaces.addPlace(eth.accounts[14], "Bahnhof Nord, Willy-Brandt-Platz", "53.145114",
   "8.222248", {from:eth.accounts[0], gas: 300000});
2  parkingplaces.addPlace(eth.accounts[15], "Am Pferdemarkt", "53.146410", "8.212653",
   {from:eth.accounts[0], gas: 300000});
```

LISTING 5: AUSCHNITT AUS JAVASCRIPT "CREATEPARKINGPLACES.JS"

Da über die Funktion `addPlace` keine Slots angelegt werden, wurde hierfür ein JavaScript erstellt. Teilweise müssen für einen Parkplätz mehrere Transaktionen zum Anlegen von Slots ausgeführt werden. Dies ist erforderlich da ansonsten das `GAS` Limit der Transaktion bzw. eines Blocks überschritten wird. Listing 6 zeigt einen Ausschnitt des JavaScript `CreateParkingPlacesSlots.js` mit sechs Transaktionen für den Parkplatz „Am Pferdemarkt“ (Account 15) mit insgesamt 401 Slots.

```
1  parkingplaces.addSlots(eth.accounts[15], 80, {from:eth.accounts[15], gas: 4000000});
2  parkingplaces.addSlots(eth.accounts[15], 80, {from:eth.accounts[15], gas: 4000000});
3  parkingplaces.addSlots(eth.accounts[15], 80, {from:eth.accounts[15], gas: 4000000});
4  parkingplaces.addSlots(eth.accounts[15], 80, {from:eth.accounts[15], gas: 4000000});
5  parkingplaces.addSlots(eth.accounts[15], 80, {from:eth.accounts[15], gas: 4000000});
6  parkingplaces.addSlots(eth.accounts[15],  1, {from:eth.accounts[15], gas: 1000000});
```

LISTING 6: AUSCHNITT AUS JAVASCRIPT "CREATEPARKINGPLACESSLOTS.JS"

Da nicht verwendetes `GAS` zurück erstattet wird, kann nicht überbezahlt werden. Das JavaScript aus Listing 5 und Listing 6 kann nur geladen und ausgeführt werden, wenn die Variable `parkingplaces` vom Typ `contract` definiert also in derselben Sitzung, in der auch der Contract erstellt wurde. Die Variable kann innerhalb einer neuen Sitzung mit dem Befehl `web3.eth.contract(ABI).at(ADDRESS)` instanziiert werden, wobei `ABI` und `ADDRESS` (Adresse des Contract) Platzhalter sind. So wird der Contract auch bei der Implementierung der JavaScript App im nächsten Kapitel instanziiert und seine Funktionen aufgerufen.

5.4 JavaScript (D)App

Im vorherigen Kapitel wurde mit dem Contract die erste Komponente der Dapp beschrieben. An dieser Stelle wird nun die Implementierung der Dapp fortgeführt, in dem ein GUI erstellt wird. Dieses stellt dem Anwender die Funktionen des Contracts gemäß ABI Spezifikation zur Verfügung. Das GitHub Repository der Dapp unter *https://github.com/blakeberg/parking-dapp* beinhaltet Contract und UI. Dieses Kapitel beschreibt die Implementierung der Dapp über das JavaScript Framework Meteor. Für die JavaScript Umgebung wurde in Kapitel 5.2 ein Docker Container erstellt. Dieser wurde um Git erweitert, um App Code aus Repositories laden zu können.

Als Umgebung wurde die JavaScript App Plattform Meteor, die JavaScript IDE WebStorm und die Container aus Anhang A.1 und A.2 verwendet. Abb. 24 zeigt die Struktur und Abhängigkeiten der Dapp für die Entwicklung der Dapp. Für die Ausführung der Dapp sind nicht zwingend Container notwendig wie in Kapitel 6.3 später gezeigt wird. Über WebStorm kann die Dapp gestartet, angehalten und debugged werden, da Meteor in der IDE integriert ist. Die Dapp kann auch über Meteor gestartet werden. Darüber hinaus kann die Dapp zu einer SPA zusammengefasst und im Webbrowser ausgeführt werden. Die externen Abhängigkeiten in Abb. 24 blau dargestellt und ein laufender Ethereum Client sind zwingend erforderlich. Die interne Struktur des GUI in Abb. 24 als Verzeichnisbaum dargestellt zeigt die drei wesentlichen Komponenten JavaScript, HTML und CSS bzw. LESS und den Images wie z.B. den Icons für die Marker der Google Map.

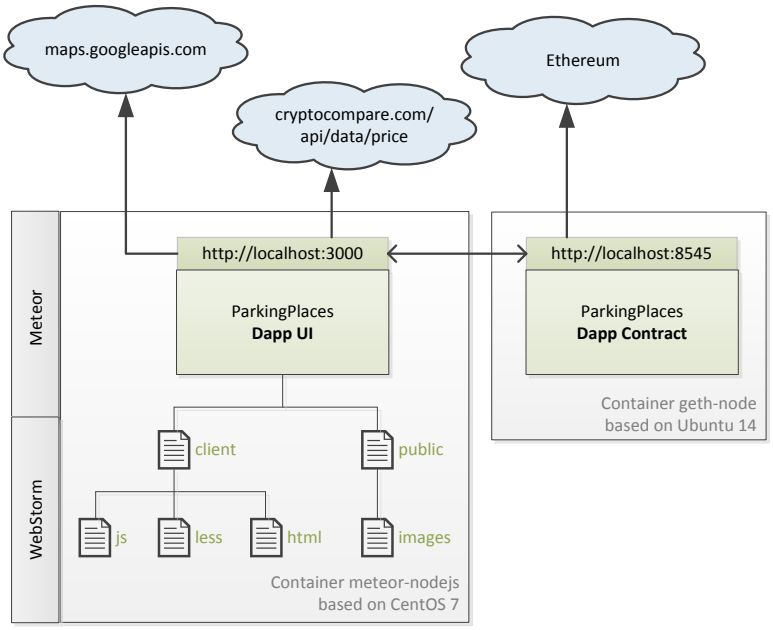

ABB. 24: ENTWICKLUNG UND AUFBAU DER DAPP

Im Folgenden werden die Implementierung mit dem JavaScript Framework Meteor und Integration von Google Maps und Ethereum beschrieben.

5.4.1 Meteor Client und Pakete

Mit Meteor steht ein umfangreiches reaktives JavaScript Framework zur Verfügung, welches aufgrund direkter Empfehlung von Ethereum und Unterstützung durch Ethereum Pakete für die Implementierung des Dapp Prototypen verwendet wurde [Et16o]. Die Dapp läuft als SPA vollständig im Client (Webbrowser) ohne Server und muss nicht deployt werden. An dieser Stelle werden Client und verwendete Pakete vorgestellt. Für eine vollständige Beschreibung des Meteor Frameworks wird auf die Entwicklerdokumentation [Me16] verwiesen.

In dem Verzeichnis `ui` des Dapp Repository gib es ein `.meteor` Verzeichnis welches alle genutzten Pakete, Plattformen und Versionen beinhaltet. Eine Übersicht der hinzugefügten Pakete zeigt Tab. 25.

Paket	Beschreibung
ethereum:web3	web3 JavaScript Dapp API von Ethereum zur Kommunikation via RPC mit dem Ethereum Client wird verwendet um den Contract zu laden.
ethereum:dapp-styles	LESS/CSS Sammlung eines einheitlichen Designs angelehnt an den Dapp-Browser MIST mit eigenen Fonts und Icons.
less	Build Plug-In welches LESS Dateien in CSS kompiliert und wird unter anderem von dapp-styles verwendet bzw. um das Design zu importieren.
ethereum:blocks	Zur Nutzung einer reaktiven Kollektion der letzten 50 Blocks wird verwendet um Informationen des aktuellen Blocks anzuzeigen, für die Aktualisierung von Markern und Aufruf einiger Funktionen des Contract.
ethereum:accounts	Zur Nutzung einer reaktiven Kollektion eigener Accounts wird genutzt um den Auswahldialog eigener Account für eine Reservierung zu füllen.
dburles:google-maps	Google Maps API wird zur Darstellung der Parkplätze über animierte Marker mit eigenen Icons (abhängig von der Anzahl der freien Slots) auf der Google Map mit zusätzlichem Click Event zur Anzeige der Informationen zum Parkplatz mit Adresse, Name, Koordinaten, freien und belegten Slots sowie der nächsten freien Blocknummer verwendet.
ethereum:elements	Sammlung von Templates von denen modale Dialogen, Eingabe von Account Adressen und Auswahl eines Accounts verwendet werden.

TAB. 25: ZUSÄTZLICHE METEOR PACKAGES IN "PARKINGPLACES"

Einige dieser Pakete besitzen wiederum Abhängigkeiten zueinander oder anderen Paketen. Daher ist die Liste unter `.meteor/versions` länger als die in Tab. 25. Jedes Paket kann über den Meteor Paketmanager Atmosphere [Pe16] mit Readme, Abhängigkeiten und Versionshistorie eingesehen werden. Mit dem Befehl `meteor create --release 1.2.1 <appname>` wird eine neue Anwendung erstellt, die mit der Version 1.2.1 kompatibel ist. Per Default sind schon einige Basis-Pakete wie `jquery` oder `blaze-html-templates` integriert. Sollten diese nicht verwendet werden können sie über den Befehl `meteor remove <package>` entfernt werden. Analog hierzu können neue Pakete per `meteor add <package>` hinzugefügt werden. Hier gilt es zu evaluieren ob alle Pakete und Abhängigkeiten verwendet werden und ob ein Update auf neuere Version möglich ist. Meteor wurde im Rahmen dieser Arbeit auf Version 1.3 aktualisiert und ist verfügbar für Mac OS X, Linux und Windows.

In dem JavaScript `parking-dapp.js` (siehe Anhang C.2) wird in der ersten Zeile mit `if (Meteor.isClient) {…}` die clientseitige Implementierung eingeleitet. Da die Dapp vollständig im Client läuft, befinden sich alle weiteren Anweisungen innerhalb dieser geschweiften Klammern. Anschließend werden Konstanten wie `ETH_RPC_ADDRESS` und Variablen wie `web3` und `parkingplaces` deklariert und über `web3 = new Web3(new Web3.providers.HttpProvider(ETH_RPC_ADDRESS))` eine Verbindung zum Ethereum Client aufgebaut. Der Contract „ParkingPlaces“ wird wie in Kapitel 5.3.3

beschrieben über die Funktion `loadContract()` dann in Variable `parkingplaces` instanziiert. Wenn der Client startet wird über `Meteor.startup(function () {…}` die Kollektionen `EthBlocks` und `EthAccounts` initialisiert und `GoogleMaps` geladen.

Die weiteren Funktionen und das Rendering der Views sind über Templates umgesetzt und werden im folgenden Abschnitt beschrieben.

5.4.2 Meteor Templates

Innerhalb der reaktiven Templating und Rendering Engine Blaze von Meteor werden HTML Dateien im Bereich `<head>`, `<body>` und `<template>` geparst, kompiliert und zum Client geschickt. An dieser Stelle werden Blaze Templates der JavaScript App beschrieben. Statt Blaze kann mit Meteor auch Angular oder React verwendet werden.

Insgesamt werden acht Templates aus eingebundenen Paketen verwendet siehe Tab. 26.

Name	Beschreibung
dapp_modal_question	Modales Fenster mit einer Frage (ok, cancel).
dapp_addressInput	Eingabefeld für Ethereum Adressen in Hex mit Icon und Validierung.
dapp_modalPlaceholder	Platzhalter für Modale Fenster.
dapp_selectAccount	Auswahl eins Account durch Name, Balance, Account Typ und Icon.
dapp_formatBalance	Formatierung von ETH verwendet API von cryptocompare (Price Feed)
dapp	Haupt-Template enthält alle anderen Templates.
modal_info	Template zur Anzeige von modalen Dialogen ohne Aktionen.
googleMap	Template zur Anzeige der Google Map

TAB. 26: METEOR TEMPLATES IN "PARKINGPLACES"

Das vollständige HTML mit allen Templates aus Tab. 26 und Designs befindet sich im Anhang C.1. Modale Dialoge werden zur Anzeige von Informationen verwendet. Die Google Map wird zur Anzeige der Parkplätze mit Markern genutzt. Die Adresseingabe und -Auswahl ist für die Reservierung eines Parkplatzslots notwendig. Die Formatierung von ETH ist für Balance aber auch Kosten einer Reservierung und Kosten für einen Block hilfreich, da mit der Umrechnung in EUR und BTC der aktuelle Wechselkurs mit angezeigt wird.

Templates besitzen JavaScript mit Datenkontext und Hilfsmethoden. Im HTML wird Spacebars als Template-Sprache verwendet, wo jeder Aufruf in doppelt geschweifte Klammern gesetzt wird. Hierin sind Bedingungen und Schleifen möglich, wie sie für die Anzeige der Contract Events verwendet wird. Das Template `dapp` besitzt im JavaScript `parking-dapp.js` (siehe Anhang C.2) Events-, `helpers` und eine `onCreate` Funktion.

In Tab. 27 werden die `helpers`-Funktionen mit Aufrufen innerhalb des HTML beschrieben.

Name	Beschreibung
currentBlockNumber	Aktualisiert alle Marker auf der Google Map falls der Modulo der aktuellen Blocknummer und der Konstante `REFRESH_INTERVALL` 0 ergibt. Gibt die aktuelle Blocknummer zurück.
currentBlockTime	Gibt eine formatierte Uhrzeit des Zeitstempels des aktuellen Blocks zurück. `<h3>last block {{currentBlockNumber}} at {{currentBlockTime}}</h3>`
accounts	Gibt eine Liste von Accounts des verbundenen Ethereum Client zurück. `<div class="from">{{> dapp_selectAccount accounts=accounts showAccountTypes=true}}</div>`
contractLogs	Ruft die aktuelle Blocknummer ab und gibt ein Array mit Eventlog. `<div class="payment">{{#each contractLogs}}<li>{{this}}</li>{{each}}</div>`
contractController	Gibt vom Contract die `public` State Variable `controller` zurück. `{{> dapp_addressInput placeholder=contractController disabled="true"}}`
contractParkingCosts	Gibt vom Contract die `public` State Variable `blockCosts` zurück. `{{dapp_formatBalance contractParkingCosts "0,0.00[000] UNIT"}}`
estimatedParkingCosts	Ruft die Funktion `calculateEstimatedCosts` des Contract mit der aktuellen Blocknummer und der globalen Variable `block` auf und gibt das Ergebnis zurück. `<small> {{dapp_formatBalance estimatedParkingCosts "0,0.00[000] UNIT"}}</small>`
mapOptions	Wenn die Google Map geladen ist werden die Konstanten `MAP_ZOOM` und `CENTER` in einem Objekt zurückgegeben. `<main class="dapp-content">{{> googleMap name="map" options=mapOptions}}</main>`

TAB. 27: HELPER FUNKTIONEN FÜR TEMPLATE "DAPP" IN "PARKING-PLACES"

Die `helpers`-Funktionen sind wie Tab. 27 in Ausschnitten zeigt in der HTML View eingebunden. Diese ist über ihre Templates reaktiv d.h. sobald ein neuer Block vom Ethereum Client synchronisiert wurde, wird das Template innerhalb der View dort wo der aktuelle Block eingebunden ist partiell neu gerendert. Dies erfolgt über die `helpers`-Funktionen `currentBlockNumber`, `contractLogs`, `currentBlockTime` und `estimatedParkingCosts`.

Die View wird für jeden Block aktualisiert und zusätzlich alle `REFRESH_INTERVALL` Blöcke die Marker auf der Google Map mit den Parkplätzen und Verfügbarkeiten. Weiter wird die View bei auftretenden Events aktualisiert. Die in Kapitel 5.3.1 beschriebenen Contract Events werden bei der Erstellung des Template `dapp` registriert werden. Hier werden initial für alle Parkplätze des Contracts Marker mit `click` Event auf der Google Map erstellt. Neben diesen Events werden für das Template `dapp` eigene `click` und `change` Events definiert für die Interaktion mit `button` und `input` Feldern siehe Abb. 25 unter `dapp-aside` (Sidebar).

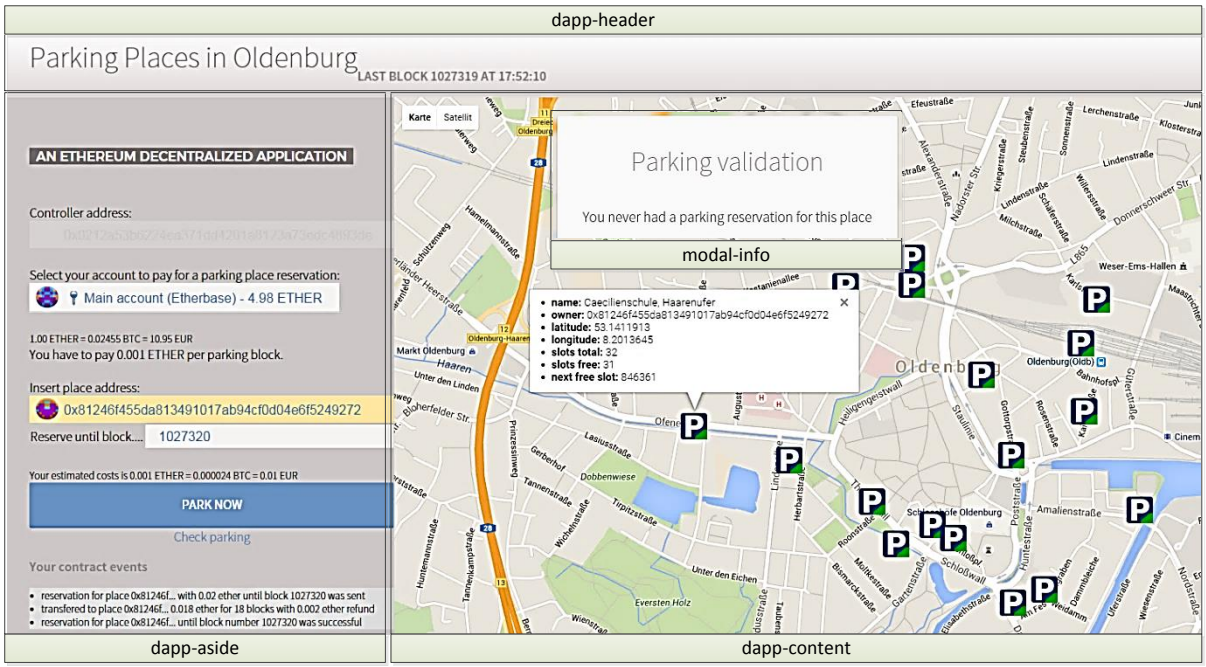

ABB. 25: HTML-VIEW MIT BEREICHEN NACH "DAPP-STYLES"

Beim Klick auf „Check Parking“ wird die Adresse des Parkplatzes unterhalb von „Insert place address“ und die Adresse des gewählten Account über `TemplateVar.getFrom(…)` aus den Templates `dapp_addressInput` und `dapp_selectAccount` ermittelt. Hiermit wird über die Contract Funktion `existsPlace` der Parkplatz validiert und anschließend über die Contract Funktion `getReservedBlock` die reservierte Blocknummer ermittelt. Abhängig von der Blocknummer wird über den Aufruf eines modalen Dialogs eine entsprechende Information innerhalb des Templates `modal_info` ausgegeben wie Abb. 25 mit der Meldung „Parking validation“ zeigt. Anders als in Abb. 25 sind die Bereiche `dapp-header`, `dapp-content` und `dapp-aside` bei modalen Dialogen eingefroren. Das Reservieren eines Parkplatzes besitzt im Vergleich zur Validierung des Parkenden folgende Unterschiede:

- Blocknummer als Parkende muss eingegeben werden.
- Es wird geprüft ob die Blocknummer in der Zukunft ist.
- Es wird geprüft ob der Parkplatz einen freien Slot hat.
- Es werden die Kosten berechnet und geprüft ob der Balance des Accounts ausreicht.
- Der ausgewählte Account muss entsperrt sein, da eine Transaktion ausgeführt wird.
- Ein modaler Dialog wird vor Ausführung zur Bestätigung dieser angezeigt.
- Es wird ein Eintrag in das Template `contractLogs` geschrieben.

Beim Erstellen des Template `dapp` werden alle Events des Contracts registriert, so dass auf Reservierungen und Transaktionen reagiert wird. Für beide Events wird über die Methode `isOwnAccount` geprüft ob ein Account aus dem Template `dapp_selectAccount` betroffen ist. In Listing 7 wird gezeigt wie das Event `Transaction` des Contract behandelt

wird und dabei die Argumente `args` aus dem Parameter `result` gelesen werden. Bei eigenem Account wird ein Eintrag in das Template `contractLogs` geschrieben.

```
parkingplaces.Transaction({}, '', function (error, result) {
  if (!error) {
    if (isOwnAccount(result.args.from)) {
      eventlogs.push("send to " + result.args.to + " " + result.args.transfered + " wei" +
      " for " + result.args.block + " blocks with " + result.args.refund + " wei refund");
    }
  }
});
```

LISTING 7: HANDLING CONTRACT EVENT "TRANSACTION"

Bei der Reservierung erfolgt eine Meldung über einen modalen Dialog über das Template `modal_info` wie in Abb. 25 dargestellt. Außerdem werden alle Marker der Google Map aktualisiert, dessen Implementierung im folgenden Abschnitt beschrieben wird.

5.4.3 Google Maps

Google Maps wurde für die Kartendarstellung der Parkplätze aus dem Contract über Marker und InfoWindows verwendet. An dieser Stelle wird die Integration und Interaktion mit dem Contract beschrieben wie die Aktualisierung über Contract Events.

Die Darstellung der Google Map über das Paket `google-maps` und Template `googleMap` erstreckt sich über den vollen Bereich `dapp-content`. Ist die Map geladen wird über das Array `places` des Contract iteriert und jedes Mal ein Marker mit einem definierten Timeout `TIMEOUT_ANIMATION` per Animation `DROP` (fallend) der Map hinzugefügt wie Listing 8 zeigt. Dabei sind `places` und `markers` assoziative Arrays und haben als Key die Adresse `owner` des Parkplatzes, der selbst innerhalb des Contract als Array definiert ist.

```
var marker = new google.maps.Marker({
  title: places[owner][1],
  position: {lat: Number(places[owner][2]), lng: Number(places[owner][3])},
  map: GoogleMaps.maps.map.instance,
  animation: google.maps.Animation.DROP,
  icon: getIcon(owner)
});
markers[owner] = marker;
```

LISTING 8: HINZUFÜGEN EINES GOOGLE MARKERS

Das Icon des Markers wird über die Methode `getIcon` ermittelt anhand freier und belegter Slots, welche über die Funktionen `getFreeSlotCount` und `getSlotCount` abgefragt werden. Bei weniger als `RED_THRESHOLD` % freien Slots wird ein rotes, bei weniger als `YELLOW_THRESHOLD` % ein gelbes und sonst ein grünes Icon angezeigt siehe Abb. 26.

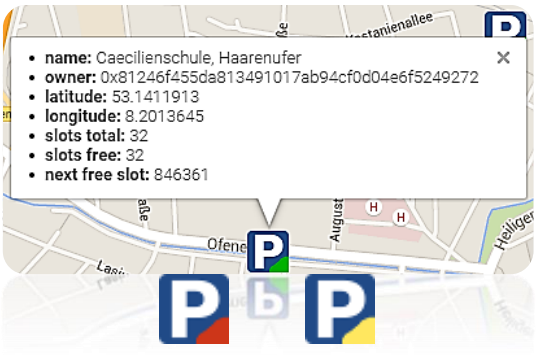

ABB. 26: ICONS UND INFOWINDOW FÜR GOOGLE MARKER

Das Original Icon unter *https://commons.wikimedia.org/wiki/File:Parking_icon.svg* wurde gemäß Bestimmung des Copyrights („public domain“) überarbeitet siehe Abb. 26. Jedem Marker wird neben dem Icon über die Methode `addMarkerInfo` ein InfoWindow und ein `click` Event hinzugefügt. Das InfoWindow besteht aus einer HTML kodierten Liste mit allen Informationen des Parkplatzes mit Anzahl seiner Slots, der freien Slots und nächsten freien Blocknummer wie Abb. 26 zeigt. Hierfür wird neben den Funktionen `getFreeSlotCount` und `getSlotCount` auch `getNextFreeBlock` des Contracts aufgerufen. Das `click` Event wird dem Marker hinzugefügt über `addListener('click', function() {…})` zusammen mit einem Handler, der auf dieses Event reagiert. Hierüber werden Animationen des Marker beendet, ein vorhandenes InfoWindow und Icon aktualisiert sowie das InfoWindow geöffnet. Alle `REFRESH_INTERVALL` Blöcke werden die Marker auf der Google Map ohne Animation oder Hinweis aktualisiert. Sollte ein Parkplatz dem Contract hinzugefügt werden wird das Event `PlaceAdded` getriggert, welches wie in Listing 8 beschrieben einen Marker erstellt wie beim Laden der Google Map. Sollte ein Parkplatz neue Slots erhalten, wird das Event `SlotsAdded` getriggert. Hierbei wird der entsprechende Marker aktualisiert, wobei die Karte auf die Position des Markers zentriert wird und der Marker mit der Animation `BOUNCE` (auf der Karte springend) seine Aktualisierung visualisiert. Diese Visualisierung erfolgt auch wenn das Event `Reservation` getriggert wurde. Alle Events wurden bei der Erstellung des Template `dapp` registriert.

Bei der Integration der Google Maps API wurde kein Account bei Google angelegt, was auch in Browserkonsole als Warnung angezeigt wird. Google sieht eine kostenlose Nutzung vor „[…] bis zur Überschreitung eines Kontingent von 25.000 geladenen Karten pro Tag für 90 aufeinanderfolgende Tage…“ [Go16]. Später ist zu prüfen, ob dieses Kontingent ausreicht bzw. da es sich bei der Reservierung von Parkplätzen um eine kostenpflichtige Transaktion handelt ein anderes Nutzungsmodell von Google gewählt werden muss. Alternativ kann die API von OpenStreetMap genutzt werden.

Das Meteor Pakete `google-maps` und alle weiteren Pakete und Abhängigkeiten können über den Meteor Build Client zu einer SPA zusammengefasst werden, wie im folgenden Abschnitt beschrieben wird.

5.4.4 Meteor Build Client

In einer SPA läuft die gesamte Anwendung über eine einzige Webseite, wobei die Präsentationsschicht nicht mehr wie bei traditionellen Anwendungen im Server liegt sondern im Client. Die clientseitige Implementierung wurde in Kapitel 5.4.1 beschrieben. Dieses Kapitel beschreibt den Meteor Build Client, der die erstellte Meteor Dapp in eine SPA bindet, die lokal im Browser ausgeführt werden kann.

Der Meteor Build Client kann über den Paketmanager NPM von NodeJS geladen und installiert werden. Da bis vor kurzem dies nur für Apps in der Meteor Version 1.2.1 vom 26.10.2015 möglich war, wurde der Prototyp mit dieser Version umgesetzt. Dies gilt es mit neueren Meteor Versionen wie 1.3.0 vom 28.03.2016 nochmals zu evaluieren, da auch die Version des „meteor-build-client“ aktualisiert wurde und es einen Fix für die Meteor Version 1.3.0 gibt [Vo16]. Mit dem Befehl `meteor-build-client ../bundled-parking-dapp -p ""` wurde innerhalb des Container die Dapp in eine SPA zusammengefasst und in das Dapp-Repository unter `bundled-parking-dapp` hochgeladen. In dem Verzeichnis sind die drei Icons, ein JavaScript, ein HTML und ein CSS enthalten. Zudem wurden die Ressourcen (Icons und Fonts) der Pakete `dapp-style` und `elements` in dem Verzeichnis `packages` abgelegt. Diese Struktur zeigt Abb. 27 ebenso wie die Abhängigkeiten zu einem lokalen Ethereum Client sowie der Google Maps API und Price Feed von cryptocompare.

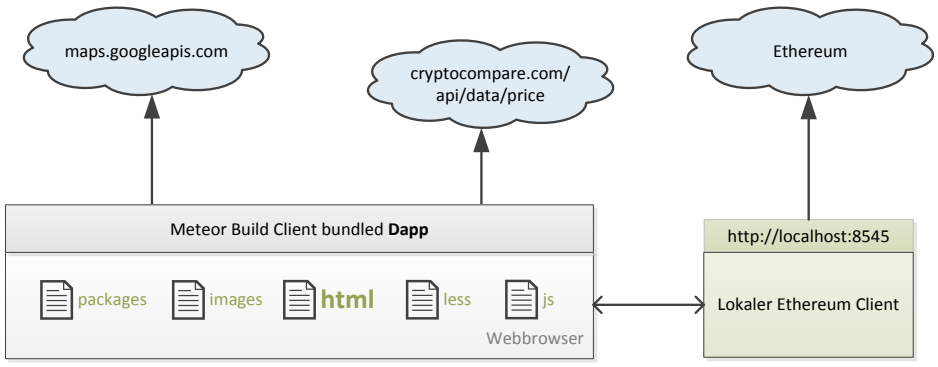

ABB. 27: METEOR BUILD CLIENT BUNDLED DAPP

Die SPA kann aus dem Repository heruntergeladen und die `index.html` lokal ausgeführt werden. Dabei muss der Webbrowser Zugriff per RPC auf einen verbundenen Ethereum Client besitzen und das Internet besitzen. Im HTML wird das CSS und JavaScript geladen, wobei hier JavaScript für Meteor hinzugefügt wurde siehe Listing 9.

```
1  <link rel="stylesheet" type="text/css" class="__meteor-css__"
   href="20b6ce15a86bd2f79344a33afd4d91ef4d850db6.css?meteor_css_resource=true">
2  <script type="text/javascript">__meteor_runtime_config__ =
   JSON.parse(decodeURIComponent("%7B%22meteorRelease%22%3A%22METEOR%401.2.1%22%2C%22ROOT_URL
   _PATH_PREFIX%22%3A%22%22%2C%22ROOT_URL%22%3A%22%22%7D"));</script>
3  <script type="text/javascript" src="888c89abebb2f230638e8967839a1c529f91e780.js"></script>
4  <script type="text/javascript">Meteor.disconnect();</script>
```

LISTING 9: METEOR BUILD CLIENT SKRIPTE IN GENERIERTEM HTML

In Zeile 3 von Listing 9 wird das über alle Pakete generierte und obfuskierte JavaScript geladen, welches die Pfadkonfiguration im JavaScript aus Zeile 2 benötigt. Hier wird ein URL enkodierter String dekodiert und in die Variable `__meteor_runtime_config__` geschrieben. Der URL-enkodierte String ist im JSON-Format und lautet dekodiert `{"meteorRelease":"METEOR@1.2.1","ROOT_URL_PATH_PREFIX":"","ROOT_URL":""}`. Der Pfad wurde mit dem Parameter `-p ""` beim Build gesetzt. Meteor muss nicht installiert sein, auch wenn es Zeile 2 und 4 aus Listing 9 vermuten lassen. Alles ist im generierten JavaScript wie auch das der Dapp siehe Anhang C.2. Der `disconnect` Befehl aus Zeile 4 trennt den Client vom Server. Das generierte CSS ist auf eine Zeile komprimiert und minimiert enthält das CSS aller Pakete und das der Dapp siehe Anhang C.1.

Die generierten Dateien in der Meteor bundled Dapp in Abb. 27 können lokal im Webbrowser ausgeführt werden. Eine Bereitstellung über einen externen Webserver wäre zu prüfen, da sich die Dapp aktuell über `http://localhost:8545` per RPC mit dem Ethereum Client verbindet und dessen Accounts verwendet. Eine andere Möglichkeit ist die Verteilung der Dapp und ihre Ressourcen über Ethereum Subprotokoll Swarm siehe Abb. 16 in Kapitel 4.3.

Im nächsten Kapitel werden Contract und Dapp iterativ getestet sowie die Einrichtung und Ausführung der Dapp über Docker Container beschrieben. Bei der Evaluation von Contract und Dapp wird die iterative Entwicklung und Evaluation mehrerer Prototypen gezeigt, die jeweils aufeinander aufbauen.

6 Test und Anwendung

Im letzten Kapitel wurde die Implementierung der Dapp mit Solidity Contract und JavaScript (D)App vorgestellt. Hier wird nun zunächst das Vorgehen der iterativen Entwicklung und Verifikation des Smart Contract beschreiben. Anschließend wird die Integration des Contract in die JavaScript App über mehrere Prototypen (POC) evaluiert. Zuletzt wird die Einrichtung und Ausführung der Dapp über Docker Container vorgestellt.

6.1 Evaluation des Contract

Als Zielerreichung für die Evaluation gilt die Umsetzung des Use Case aus Kapitel 5.1 als Dapp der Ethereum Blockchain. An dieser Stelle wird gezeigt, wie der Contract iterativ entwickelt und evaluiert wurde, bis ein geeigneter Testkandidat gefunden war. Im folgenden Kapitel wird dann gezeigt wie dieser in POC4 und POC5 in die JavaScript App eingebunden wurde.

Während der Implementierung des Contracts wurde ein iteratives Vorgehen gewählt d.h. jedes Element des Contracts von der Datenstruktur, dem Konstruktor, Events, Modifier bis zu den Funktionen wurde separat umgesetzt und geprüft. Bei erforderlichen Korrekturen wurde im Anschluss ein neuer Contract Testkandidat deployt und geprüft. Die `close()` Funktion, die State Variable `controller` und der Modifier `isController` mussten zuerst implementiert werden, um den Account Speicher und Code der vergangenen Testkandidaten zu löschen und Value zurückzuerstatten. Innerhalb dieses Prozess wurden vom Account `0x0212a53b6224ea371dd4201a8123a73edc4893de` in 40 Tagen insgesamt 93 Contract Testkandidaten erstellt, von denen 9 (ca. 9,7%) wegen `OUTOFGAS` mit Fehler abgebrochen sind und 83 nach den Tests wieder geschlossen wurden.

Im Solidity Online Compiler wird der Contract in Echtzeit syntaktisch geprüft und ein JavaScript für das Deployment des Contract generiert. Dieses wurde über den Container „geth-node“ ausgeführt ebenso wie Transaktionen und Message Calls zum Testen der Funktionen, Variablen, Events und Modifier des Contract. Innerhalb des Blockchain Explorer EtherCamp [Et16d] wurde der VM Trace, Contract Speicher und Stack der EVM Ausführung bei der Verifikation des Testkandidaten analysiert. Hierfür wurden basierend auf dem Use Case aus Kapitel 5.1 acht Testcases mit Testfällen erstellt, die in Tab. 28 beschrieben sind.

Testcase	Testfälle	Beschreibung
close ParkingPlaces	4	Testet die Variable controller, den Konstruktor, den Modifier isController und die Funktion close.
add ParkingPlace	15	Testet die Strukturen Place und Slot, die Variable places, die Funktionen addPlace und existsPlace sowie das Event PlaceAdded.
update ParkingPlace	6	Testet Mapping placeOf, die Funktion addSlots, den Modifier isOwner und das Event SlotsAdded.

show Places	18	Testet die Funktionen getSlotCount, getFreeSlotCount und getNextFreeBlock.
estimate Costs	5	Testet die Variable blockCosts, den Konstruktor mit Argument und die Funktion calculateEstimatedCosts.
reserve Slot / check Parker	17	Testet die Events Reservation und Transaction, die Funktionen reserveSlot und getReservedBlock sowie die internen Funktionen payReservation und getNextFreeSlot.
allowReservation	7	Test den Modifier allowReservation innerhalb der Funktion reserveSlot.
hasValue	5	Test den Modifier hasValue innerhalb der Funktionen addPlace und addSlots

TAB. 28: ÜBERSICHT DER TESTCASES FÜR EVALUATION DES CONTRACT

Eine ausführliche Beschreibung der Testfälle der in Tab. 28 beschriebenen Testcases befindet sich in Tab. 32 im Anhang C.3. Jede Änderung des Contract erforderte eine erneute Ausführung der vom betroffenen Bereich (Funktion, Modifier, Variablen, Events) abhängigen Testcases. Durch die permanente Wiederholung von Ausführungen wurden neue Fehler direkt behoben. Die Testausführungen wurden manuell über JavaScripts durchgeführt und sollten aufgrund des eingesetzten Zeitinvest in einer weiterführenden Betrachtung automatisiert werden. Ebenso war für die Prüfung des VM Trace der Transaktionen im Blockchain Explorer EtherCamp [Et16d] während der Tests mit erhöhtem Zeitinvest verbunden. Der Blockchain Explorer EtherScan [Et16i] konnte zunächst keine Contracts mit Argumenten im Konstruktor verifizieren und bei EtherCamp haben VM Trace und Anzeige des Account Storage nicht funktioniert. Diese Probleme konnten mit dem DEV Team behoben werden. Hier sind weiter Frameworks und Tools zu prüfen, die ein lokales Debugging von Ethereum Transaktionen und Contracts unterstützen analog zum Debugging von JavaScript innerhalb der IDE WebStorm.

Unabhängig von den erwarteten Ergebnissen der Testfälle in den in Tab. 28 beschriebenen ersten sechs Testcases wurden Fehler bzw. Möglichkeiten zur Optimierung festgestellt, die in Tab. 29 beschrieben und teilweise nachträglich umgesetzt wurden.

Beschreibung	Lösung
1. Value innerhalb einer Transaktion wird an den Contract Account transferiert und ist hier gesperrt.	Über einen Modifier der prüft ob eine Transaktion Value besitzt `msg.value > 0` und in dem Fall per `throw` eine Exception auslöst, wird die Transaktion zurückgerollt und kein Value transferiert.
2. Der `controller` und `owner` sollte nicht reservieren dürfen.	Über einen Modifier der prüft ob eine Transaktion von dem `controller` oder vom `owner` stammt und in dem Fall per `throw` eine Exception auslöst, wird die Transaktion zurückgerollt und kein Value transferiert.
3. Der Contract sollte vollständig autonom agieren. Aktuell gibt es	Für die Verwaltung von Places und deren Slots ist ein Konzept für eine DAO zu erstellen, welches im Kontext

	einen `controller` der den Contract schließen (gesperrtes Value wird an ihn in dem Fall transferiert) und einen Place hinzufügen darf.	Smart City unter die Eigenschaft Smart Governance fällt. Hier wird u.a. wie von Ethereum [Et16s] beschrieben eine Funktion zum Ändern des controller verwendet. Eine unmittelbare Maßnahme wäre das Entfernen der `close` Funktion und das Hinzufügen einer Funktion zum Ändern des `controller`.
4.	Der controller sollte keinen Place mit seiner Adresse erstellen können, da jeder Place ebenfalls autonom agieren soll.	Über einen Modifier der prüft ob owner des Place identisch mit dem `controller` ist und in dem Fall per `throw` eine Exception auslöst, wird die Transaktion zurückgerollt und kein Value transferiert.
5.	Bei Ausführung einer Transaktion an den Contract mit unbekannter Funktion und Value wird dieses an den Contract transferiert und in diesem gesperrt.	Über eine Fallback Funktion ohne Namen und Parametern die per `throw` eine Exception auslöst, wird die Transaktion zurückgerollt und kein Value transferiert.
6.	Nach dem Hinzufügen eines Place mit einem Slot ist dieser zwar im Contract Account Speicher aber kann nicht abgerufen werden.	Slots werden per Funktion `addSlots` hinzugefügt, so dass bei der Funktion `addPlace` das Array `slots` nicht gesetzt werden muss.
7.	Besitzt ein Parker für einen Place noch eine gültige Reservierung soll er diese verlängern können statt einen neuen Slot zu belegen und die komplette Parkdauer zu bezahlen.	Innerhalb der Funktion `reserveSlot` muss geprüft werden, ob der Parker noch eine gültige Reservierung besitzt. Wenn ja dann muss entsprechend der Parkdauer `reservedBlock` diese nicht mehr bezahlt werden sondern nur die Verlängerung.
8.	Typdefinition von Variablen innerhalb des Contract optimieren.	Hier wurde in Kapitel 5.3 bereits eine Lösung für Arrays mit festen Größen genannt. Hierfür wären die `string` Variablen für Namen und Koordinaten auf ein `byte` Array umzustellen und die Anzahl an Slots und Places innerhalb der Arrays zu definieren. Zudem kann `uint` (256) auf `uint`(8, 16, 32, 64, 128) umgestellt werden.
9.	Indexierte Events zum Suchen und Filtern nach Topics.	Dies wurde in Kapitel 5.3.1 bereits beschrieben.

TAB. 29: FEHLER UND MÖGLICHKEITEN ZUR OPTIMIERUNG DES CONTRACT

Die Lösungen der Punkte 1, 2 und 5 aus Tab. 29 wurden umgesetzt und sind innerhalb des Contract über die Modifier `hasValue` und `allowReservation` sowie der Fallback-Funktion auch bereits in Kapitel 5.3.1 und 5.3.2 beschrieben. Die Lösungen 4, 6, 8 und 9 aus Tab. 29 wurden zwar erfolgreich implementiert und getestet aber noch nicht in den Prototypen oder die Testcases übernommen. Die letzten beiden Testcases in Tab. 29 wurden nachträglich abhängig von der Umsetzung der Punkt 1, 2, 5 und 7 erstellt. Für die Lösung zu Punkt 7 aus Tab. 29 wurde die Funktion `reserveSlot` umgeschrieben und der Testcase „reserve Slot / check Parker" angepasst. Dieser Punkt ist ein wesentlicher Aspekt des Use Case aus Kapitel 5.1, denn das Verlängern eines Parkticket bietet einen erheblichen Mehrwert, welcher später im Kontext von Smart Cities diskutiert wird.

Der zuletzt evaluierte Contract (Testkandidat) wurde in die Dapp übernommen und wird im folgenden Kapitel evaluiert. Dabei wird der Contract ab POC4 in die JavaScript App integriert.

6.2 Evaluation der Dapp

Im vorherigen Kapitel wurde der Contract iterativ entwickelt und Zwischenstände über die in Tab. 32 im Anhang C.3 definierten Testcases evaluiert. Der Contract der letzten Iteration wurde in die Meteor JavaScript App eingebunden, die mit GUI und Contract die dezentralisierte Anwendung (Dapp) bilden. Dieses Kapitel beschreibt die Entwicklung und Verifikation der Dapp bis zur vollständigen Integration des Contract. Als Zielerreichung gilt auch hier die Umsetzung des Use Case aus Kapitel 5.1 als Dapp der Ethereum Blockchain.

Bei der Entwicklung der Dapp wurden fünf aufeinander aufbauende Prototypen erstellt, die im Repository unter `prototypes` abgelegt wurden. Jeder POC ist ein eigenes Meteor Projekt der JavaScript IDE WebStorm, über die Meteor gestartet, gestoppt, angehalten, debugged sowie Pakete hinzugefügt oder entfernt werden können. Mit der Debugging Funktion konnte jeder POC hinsichtlich seines JavaScript verifiziert werden:

1. Meteor JavaScript Anwendung mit Ethereum Dapp Styles verwendet Pakete `dapp-styles` und `less`.
2. Anbindung eines Ethereum Client und Darstellung von Blöcken und Accounts, Währungskursen und modalen Dialogen verwendet Pakete `blocks`, `accounts` und `elements` (referenzieren u.a. `tools` und `web3`)
3. Integration von Google Maps mit Marker und InfoWindow verwendet Paket `google-maps`
4. Integration des Contract und Darstellung auf Google Map (keine neuen Pakete)
5. Interaktion mit Contract durch Ausführung von Reservierungen (keine neuen Pakete)

POC1 aus Abb. 28 (links) beinhaltet lediglich das Design und Gerüst einer Meteor JavaScript Anwendung mit einem Template. Innerhalb des Template Helpers werden hier Strings zurückgegeben und die Template Events schreiben einen Logeintrag in die Browserkonsole.

In POC2 aus Abb. 28 (rechts) wurde der Ethereum Client angebunden und innerhalb des Template Helpers Block- und Account-Daten verwendet. Ein weiteres Template wurde hier zum Anzeigen von modalen Dialogen hinzugefügt welches Logeinträge der Browserkonsole ersetzt und die Inhalte der beiden Inputfelder ausliest. Zudem wird in POC2 der Währungskurs in EUR und BTC von 1 ETH über einen Price Feed geladen und angezeigt.

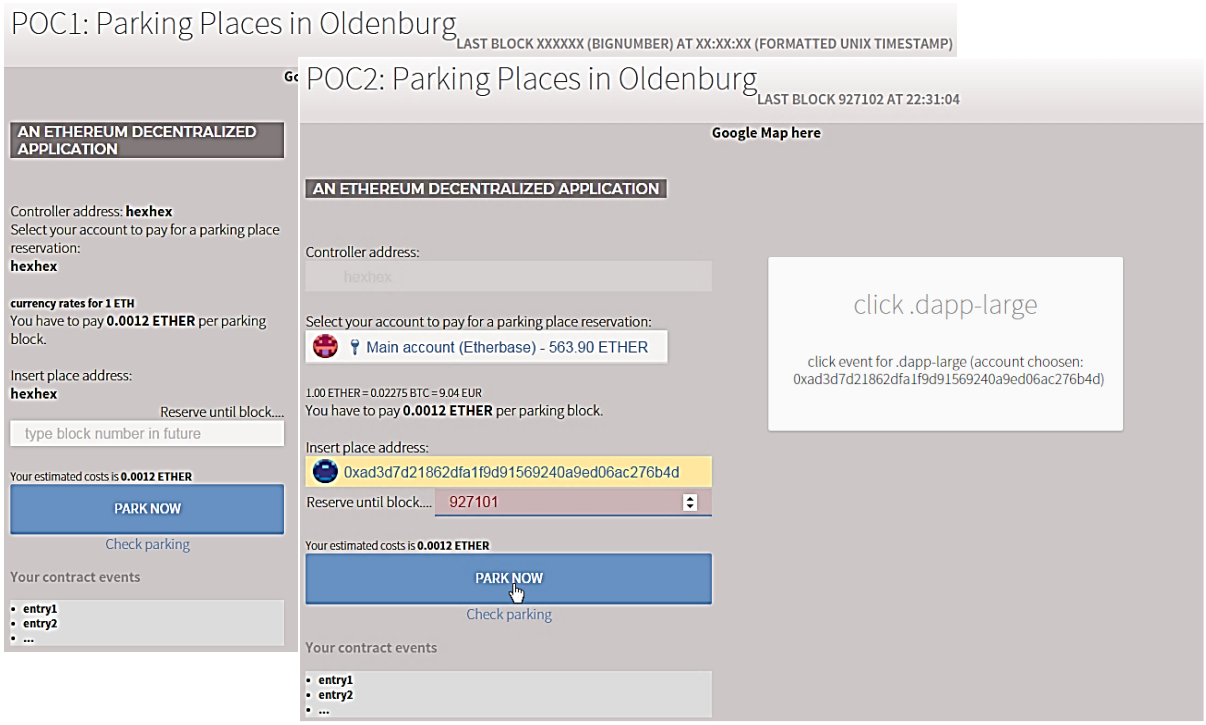

ABB. 28: POC1 MIT GUI UND POC2 MIT ETHEREUM CLIENT

In POC3 siehe Abb. 29 wurde die Google Map in das vorhandene Template integriert und drei exemplarische Marker mit InfoWindow hinzugefügt. Die Koordinaten und Daten für das InfoWindow entsprechen dabei den Daten der späteren Initialisierung des Contract bzw. der Dapp. Weiter ist eine Aktualisierung der Marker bzw. ihres InfoWindow anhand des aktuellen Block umgesetzt worden.

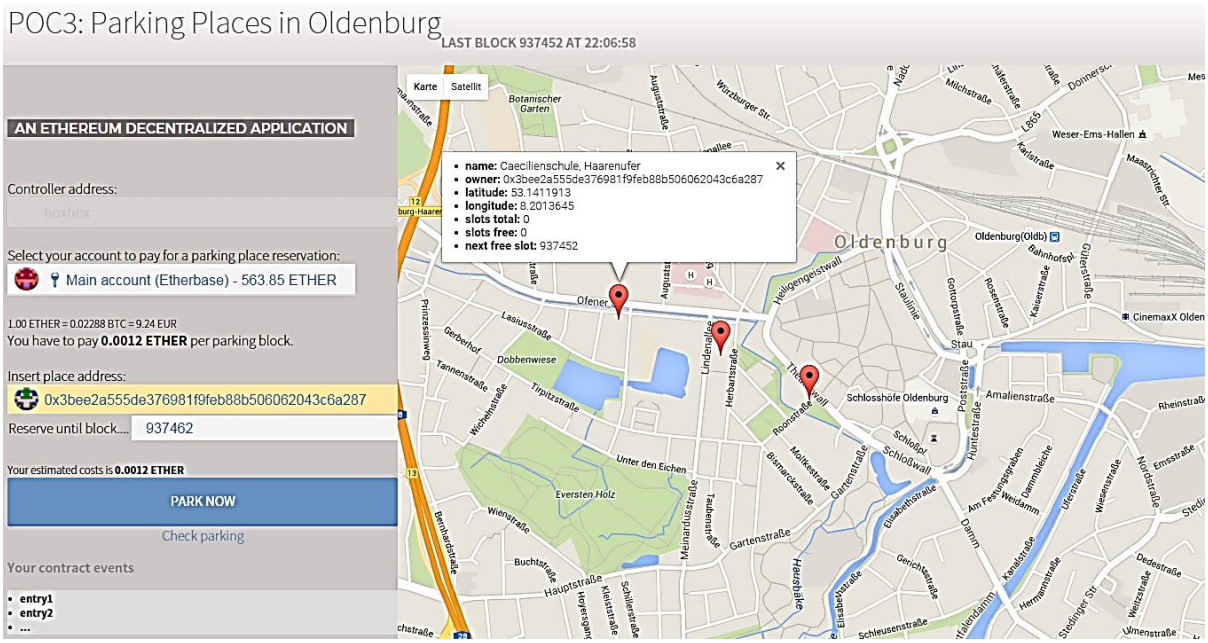

ABB. 29: POC3 MIT GOOGLE MAP

In POC4 aus Abb. 30 wurde der Contract integriert, so dass der erste Dapp POC erstellt wurde. Auch wenn POC2 und POC3 bereits einen Ethereum Client angebunden hatten, definiert sich eine Dapp wie in Kapitel 4.3 beschrieben über seinen Contract und sein GUI.

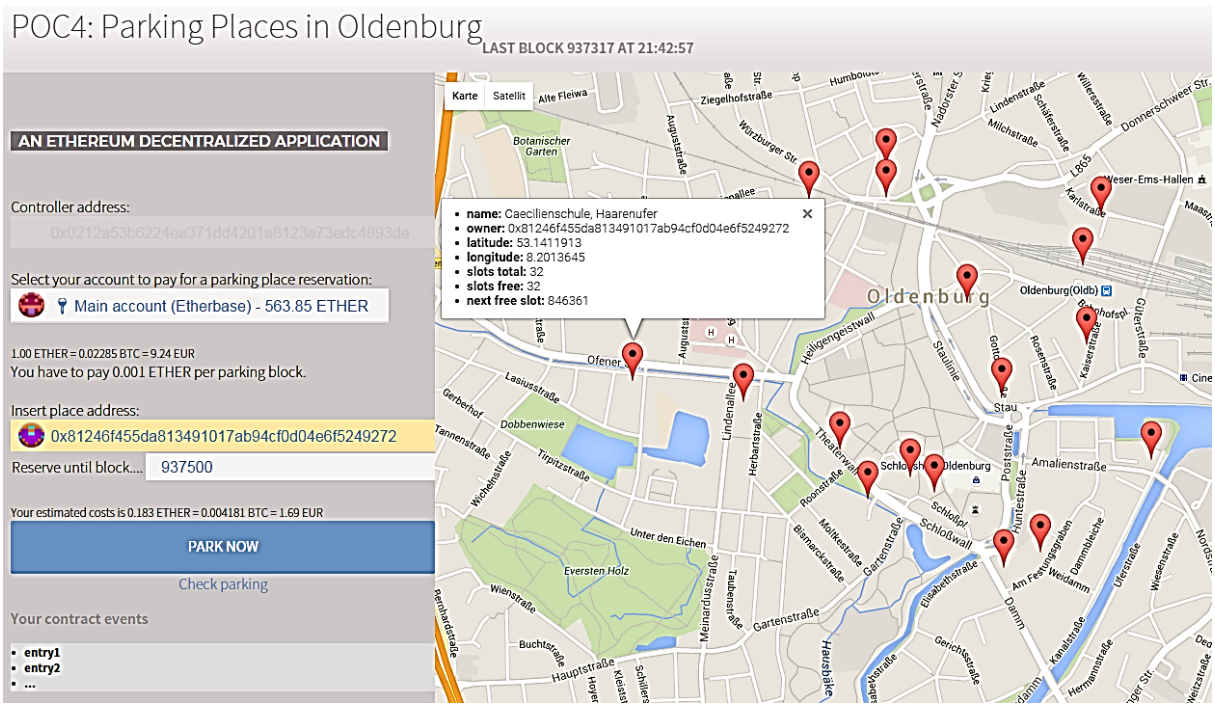

ABB. 30: POC4 MIT CONTRACT (MESSAGE CALLS)

Der Contract wurde mit 18 Parkplätzen und Slots der Dapp wie in Kapitel 5.3.3 beschrieben neu deployt und initialisiert. Hierbei konnte die Anbindung der Events `PlaceAdded` und `SlotsAdded` innerhalb der Dapp verifiziert werden, da für jeden Parkplatz ein Marker direkt auf der Google Map erscheinen bzw. aktualisiert werden muss. Bei der Integration des Contract wurden alle Message Call Funktionen außer `getReservedBlock` angebunden, da die Transaktion über die `reserveSlot` Funktion in POC5 integriert wurde. Daher wurden nur die genannten Events integriert und das Eventlog wie auch die „Park Now“ und „Check parking“ Buttons aus POC2 blieben als Mock. Wie Abb. 30 gezeigt werden Informationen über Slots, Controller Adresse und Kosten pro Block „estimated Costs“ aus dem Contract geladen. Hiermit konnte die Kostenberechnung (Kosten pro Block multipliziert mit der Differenz zwischen eingegebener Blocknummer und aktueller Blocknummer) verifiziert werden. Mit manuellen Transaktionen der Funktion `reserveSlot` über die Geth Konsole des Ethereum Client wurde in POC4 die regelmäßige Aktualisierung der Marker und InfoWindows hinsichtlich der Verfügbarkeit („slots free“ und „next free slot“) geprüft.

In POC5 wurden die Events `Reservation` und `Transaction` integriert, in dem ein Eintrag in das Eventlog geschrieben wird bzw. ein modaler Dialog wie in Abb. 31 mit den Informationen angezeigt wird. Weiter wurde die Funktion `reserveSlot` integriert und mit dem „Park Now“ Button ein modaler Dialog verknüpft. Durch diesen Use Case musste nun auch die Message Call Funktion `getReservedSlot` angebunden werden und ist mit dem Button „Check parking“ und einem modalen Dialog verknüpft.

In Abb. 31 wird eine Reservierung und ihre Prüfung über modale Dialoge und einem Ausschnitt des Eventlogs dargestellt, da sich in der Darstellung im Vergleich zu POC4 aus Abb. 30 nichts geändert hat.

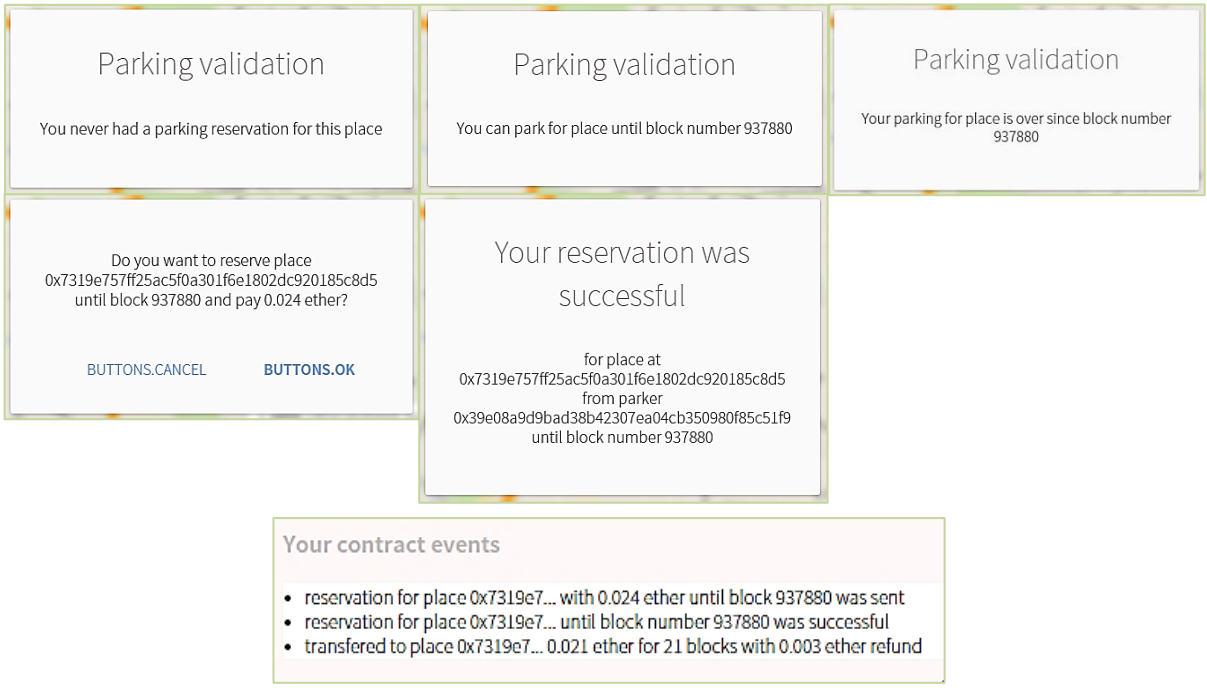

ABB. 31: POC5 MIT CONTRACT (TRANSACTIONS)

Abb. 31 zeigt die modalen Dialoge des „Check parking“ Button in den drei Zuständen, den Dialog zur Abfrage einer Reservierung nach `click` Event auf den Button „Park Now“ und den Dialog der über das Contract Event `Reservation` angezeigt wird. Außerdem wird das Eventlog für die ausgeführte Reservierung mit drei Einträgen angezeigt. Der erste Eintrag wird nach Abschicken der Transaktion über die Funktion `reserveSlot` geschrieben. Der zweite und dritte Eintrag wird beim Contract Event `Reservation` und `Transaction` geschrieben. Über das Eventlog, die Browserkonsole und die modalen Dialoge konnten die POCs neben JavaScript Debugging und den in Kapitel 6.1 beschriebenen Möglichkeiten zur Evaluation des Contract verifiziert werden. Darüber hinaus wurden die Testcases in Anhang C.3 ausgeführt um Änderungen im Contract Account Speicher über die Dapp zu verifizieren zumal diese keine Möglichkeit besitzt Parkplätze oder Slots hinzuzufügen. Analog zur Evaluation des Contract im vorherigen Kapitel wurde für die Dapp mit Tab. 30 eine Liste mit Fehlern bzw. Möglichkeiten zur Optimierung der Dapp, die bei der Verifikation von POC1-5 aufgefallen waren.

Beschreibung	Lösung
1. Anzeige der Marker mit eigenen Icons zur direkten Visualisierung der Verfügbarkeit freier Slots eines Parkplatzes.	Erstellung eigener Icons mit [P] Symbol und farblicher Kennzeichnung (rot, grün, gelb) der Verfügbarkeit. Das Icon wird abhängig von definierten Thresholds und dem Verhältnis von freien zur Gesamtanzahl an Slots eines Parkplatzes für den Marker gesetzt. Bei Aktualisierung des Marker sollte das Icon mit aktualisiert werden.

2. Eventlog nur für eigene Accounts, da sonst alle Reservierungen und Transaktionen angezeigt werden.	Über einen Abgleich der Adressen aus den Argumenten der Contract Events wird nur dann ein Eintrag in das Eventlog geschrieben, wenn es sich um eine Adresse der eigenen Accounts handelt.
3. Prüfung ob die Felder für die Eingabe von Blocknummer und Adresse ausgefüllt wurden, da die Transaktion sonst fehlschlägt.	Funktion prüft ob Feld für Adresse einen Wert besitzt sonst wird ein modaler Dialog angezeigt. Die Funktion soll über beide Buttons aufgerufen werden.
4. Prüfung ob zur eingegebenen Adresse der Parkplatz existiert, da die Transaktion sonst fehlschlägt.	Funktion prüft über einen Message Call der Funktion existsPlace des Contract ob der Platz existiert und zeigt sonst einen modalen Dialog an.
5. Prüfung ob der eingegebene Block in der Zukunft ist, da die Transaktion sonst fehlschlägt.	Funktion prüft ob der eingegebene Block größer als der aktuelle Block über den Ethereum Client ist. Sonst wird ein modaler Dialog angezeigt.
6. Prüfung ob der Parkplatz der eingegebenen Adresse noch freie Slots besitzt, da die Transaktion sonst fehlschlägt.	Funktion prüft über einen Message Call der Funktion getFreeSlotCount des Contract ob mindestens ein Slot noch frei ist. Sonst wird ein modaler Dialog angezeigt.
7. Prüfung ob der Balance des gewählten Account ausreichend für die Reservierung ist, da die Transaktion sonst fehlschlägt.	Funktion prüft ob der Balance des gewählten Account größer als die geschätzten Kosten für die Reservierung sind. Sonst wird ein modaler Dialog angezeigt.
8. Prüfung dass der gewählte Account nicht der Adresse des Place owner oder controller entspricht, da die Transaktion sonst fehlschlägt.	Funktion prüft über Message Calls der Funktionen controller und existsPlace des Contract ob die Account Adresse dem controller oder einem place owner entspricht. Hier wird ein modaler Dialog angezeigt.

TAB. 30: FEHLER UND MÖGLICHKEITEN ZUR OPTIMIERUNG DER DAPP

Punkt 1 aus Tab. 30 ist aufgefallen aufgrund der Tatsache, dass keine Übersicht der Verfügbarkeit aller Parkplätze vorhanden war. Hierfür musste man die InfoWindows aller Marker öffnen die sich überlagern. Außerdem haben die Standard Icons von Google für Marker nicht den Symbolcharakter wie das aus dem Verkehr bekannte [P]. Die drei Ampelfarben signalisieren die Verfügbarkeit freier Plätze. Punkt 2 aus Tab. 30 ist aufgefallen, weil parallel manuelle Reservierungen und Transaktionen anderer Accounts ebenfalls in das Eventlog geschrieben wurden. Somit wurde das Eventlog schnell groß und unübersichtlich. Der Dapp User muss seine Reservierungen und zugehörigen Transaktionen suchen und ist irritiert über die der anderen. Das Eventlog dient neben modalen Dialogen der Verifikation erfolgreicher Reservierungen und Transaktionen. Abb. 31 zeigt eine Reservierung mit 0,024 ETH wobei 0,003 ETH zurückerstattet wurden, da statt für 24 Blöcke nur für 21 Blöcke reserviert werden musste. Drei Blöcke liegen als Differenz zwischen dem Block zum Zeitpunkt der Reservierung über die Dapp und der tatsächlichen Verarbeitung der Transaktion innerhalb eines Blocks. Weiter ist aufgefallen, dass viele Transaktionen nicht erfolgreich verarbeitet werden konnten aufgrund der Punkte 3-8 aus Tab. 30. Der Dapp User

bekommt hierbei keine Rückmeldung einer erfolgreichen Reservierung, Einträge im Eventlog oder Fehlermeldung. Fehlerursachen sind durch den Contract bekannt und können vorab durch Prüfung und direkte Rückmeldung über modale Dialoge vermieden werden.

Alle in Tab. 30 beschriebenen Punkte außer Punkt 8 wurden in einem letzten Prototyp umgesetzt. Punkt 8 wurde zwar erfolgreich implementiert und getestet aber noch nicht in den Prototypen integriert. Dieser sechste in Kapitel 5 beschriebene Prototyp befindet sich parallel zu `prototypes` im Repository unter `ui`. Hier wurde die Verzeichnisstruktur basierend auf den Empfehlungen von Ethereum [Et16o], Vogelsteller [Vo16] und Percolate Studio [Pe16] umgestellt, die nicht genutzten Pakete (siehe Kapitel 5.4.1) entfernt und die Dapp (siehe Kapitel 5.4.4) in eine SPA gebunden. Die Einrichtung und Ausführung dieses letzten Dapp-Prototyps über Docker Container wird im nächsten Kapitel gezeigt.

6.3 Einrichtung und Ausführung

Dieses Kapitel beschreibt die Installation einer Docker Umgebung mit den beiden Docker Containern „geth-node“ und „meteor-nodejs“ beschrieben in Kapitel 5.2 sowie im Anhang A. Zudem wird die Ausführung und Einrichtung des Ethereum Clients mit der Dapp „Parking Places“ und seinen Anwendungsfällen für den Dapp User aus Kapitel 5.1 beschrieben.

Für die Ausführung der Docker Container muss zunächst eine Docker Engine als Umgebung installiert werden. Diese ist für Mac OS X, Windows, diverse Linux Distributionen und Cloud Provider verfügbar. Die Docker Engine basiert auf einer leichtgewichtigen Open Source Container Technologie mit Workflows für Build und Ausführung von Containern. Die Installation und Funktionsweise der Docker Engine bzw. der Docker Toolbox kann der Docker Dokumentation [Do16] entnommen werden und wird vorausgesetzt. Die Container „geth-node“ und „meteor-nodejs“ aus Kapitel 5.2 und Anhang A sind im DockerHub einer SaaS (Software as a Service) Plattform verfügbar. Der Container „geth-node“ enthält den Ethereum Client geth und benötigt ohne Mining und der Blockchain vom Testnetz ca. `1,2 GByte`. Der Container „meteor-nodejs“ enthält die JavaScript Umgebung und benötigt ca. `2,5 GByte`, so dass mind. `5 GByte` zur Verfügung stehen sollten.

Innerhalb des CLI der Docker Engine wird wie Listing 10 zeigt das Image für den Ethereum Client aus dem DockerHub geladen, hieraus ein Container erstellt und ausgeführt. Der Container läuft als Daemon unter dem Host und Namen `geth` mit SSH Port 10022 und RPC Port 8545. Das Passwort für die User beider Container lautet jeweils `newpass`. In Listing 10 wird über den letzten Befehl der Ethereum Client `geth` mit RPC gestartet, der sich mit dem Testnetz verbindet und die Blockchain synchronisiert.

```
1   docker pull blakeberg/geth-node:v0.3
2   docker run -d -h geth --name geth -p 10022:22 -p 8545:8545 blakeberg/geth-node:v0.3
3   ssh -p 10022 geth@localhost
4   nohup geth --testnet --fast --cache=256 --rpc --rpcaddr "geth" --rpccorsdomain "*" &
```

LISTING 10: ETHEREUM CLIENT ÜBER CONTAINER AUSFÜHREN

Bei der Synchronisation der Blockchain in Listing 10 wird der Datenbank Cache vom Default 16 auf 256 `MByte` erhöht und über den Parameter `fast` nur Transaktionsbestätigungen und Blockheader geladen. Dies ist nur bei der ersten Synchronisation möglich, da hier nur der State geladen wird und keine vergangenen Blöcke bzw. Transaktionen verarbeitet werden. Nachdem die Blockchain initial synchronisiert wurde, deaktiviert der Client automatisch den Parameter `fast`. Die Logausgabe kann mit `STRG+C` beendet werden da der Client Prozess über `nohup` weiterläuft. Mit `tail -f nohup.out` kann die Logausgabe fortgesetzt werden. Die Synchronisation der Blockchain dauert für 1.026.002 Blöcke 72 Minuten und belegt ca. `870 MByte` Speicherplatz. Diese Laufzeit wurde am 28.05.2016 gemessen auf einem Windows 7 (Docker Toolbox Version 1.11.1b) Host in einer VM mit `1 GByte` Speicher und einem Kern (AMD Phenom X4 965 CPU) sowie `20 GByte` in einer VMDK (Virtual Machine DisK) auf einer SSD (Solid State Disk). Die VM wurde über die Docker Toolbox mit den oben genannten Einstellungen und dem Client `geth` in der Version 1.3.6 erzeugt.

Für die spätere Verwendung der Dapp „ParkingPlaces“ wird ein Account mit ETH benötigt der in Listing 11 erstellt wird. Hier werden über die JSRE CLI des `geth` Clients jeweils drei Accounts mit der jeweiligen Passphrase erstellt.

```
1   geth attach
2   personal.newAccount("newAccount1")
3   personal.newAccount("newAccount1")
4   personal.newAccount("newAccount1")
5   exit
```

LISTING 11: ETHEREUM ACCOUNTS ERSTELLEN

Bei der Anlage der Accounts aus Listing 11 wird die erstellte Adresse jeweils zurückgegeben. Vor Ausführung muss die Blockchain synchronisiert sein. Über den Blockchain Explorer EtherCamp [Et16d] soll dann für jede dieser Adressen über „Get Free Ether“ 5 ETH transferiert werden, damit später in der Dapp über den entsprechenden Account ein Parkplatz reserviert bzw. ein Parkticket verlängert werden kann. Die Dapp kann wie in Kapitel 5.4.4 beschrieben als SPA ausgeführt werden oder über das Meteor Framework, welches im Container „meteor-nodejs“ bereitgestellt wird. Listing 12 zeigt das Laden des Images aus dem DockerHub, das Erstellen und Ausführen des Containers anhand des Image sowie die Ausführung der Meteor Anwendung „Parking Places“. Hierbei wird der Container mit dem gestarteten Container aus Listing 10 verlinkt und das Repository „parking-dapp“ aus GitHub geladen. Da der Ethereum Client über den verlinkten Container unter dem Host `geth` und nicht `localhost` ausgeführt wird, muss dies im JavaScript per `sed` ersetzt werden.

```
1   docker pull blakeberg/meteor-nodejs:v0.4
2   docker run -d -h meteor --name meteor -p 20022:22 -p 3000:3000 --link=geth:geth
    blakeberg/meteor-nodejs:v0.4
3   ssh -p 20022 meteor@localhost
4   git clone https://github.com/blakeberg/parking-dapp.git
5   cd parking-dapp/ui
6   sed -i.bak 's/localhost/geth/' client/parking-dapp.js
7   meteor
```

LISTING 12: METEOR DAPP ÜBER CONTAINER AUSFÜHREN

Der letzte Befehl aus Listing 12 startet die Dapp unter Port 3000, wobei Meteor erstmalig installiert wird und alle abhängigen Pakete lädt. Damit die Dapp vom Hostsystem der Docker Engine aufgerufen werden kann, muss ein DNS Eintrag in der `/etc/hosts` mit dem Hostname `meteor` und der entsprechenden IP-Adresse vorgenommen werden. Die Docker Toolbox verwendet die Oracle VirtualBox als HyperVisor für die Verwaltung von VMs. Als Default werden hier IP-Adressen ab 192.168.99.100 per DHCP (Dynamic Host Configuration Protocol) vergeben.

Die Dapp kann nun vom Hostsystem im Browser über `http://meteor:3000` aufgerufen werden. Auf der linken Seite werden die in Listing 11 erstellten Accounts mit je 5 ETH angezeigt. Auf der Google Map werden die in 5.1 beschriebenen Parkplätze als Marker angezeigt. Per Mausklick auf den entsprechenden Marker werden Informationen wie die Ethereum Adresse angezeigt, die per Copy & Paste in das entsprechende Feld auf der linken Seite für eine Reservierung eingetragen wird. Hierüber kann jeder Account über den Button „Check parking“ auf eine Reservierung geprüft werden oder zusammen mit Eingabe einer Blocknummer als Parkende ein Parkplatz reserviert werden. Da für eine Reservierung eine Transaktion erstellt und `GAS` bzw. ETH berechnet wird, muss der Account hierfür wie Listing 13 zeigt werden über die JSRE CLI des `geth` Clients entsperrt werden.

```
1   geth attach
2   personal.unlockAccount(eth.accounts[0], "newAccount1", 900)
3   personal.unlockAccount(eth.accounts[1], "newAccount2", 900)
4   personal.unlockAccount(eth.accounts[2], "newAccount3", 900)
5   exit
```

LISTING 13: ETHEREUM ACCOUNTS ENTSPERREN

Nach dem Entsperren der Accounts können für 900 Sekunden Transaktionen ohne Eingabe der Passphrase getätigt werden wie z.B. eine neue Reservierung oder das Verlängern eines Tickets für eine aktive Reservierung mit nicht abgelaufenem Parkende.

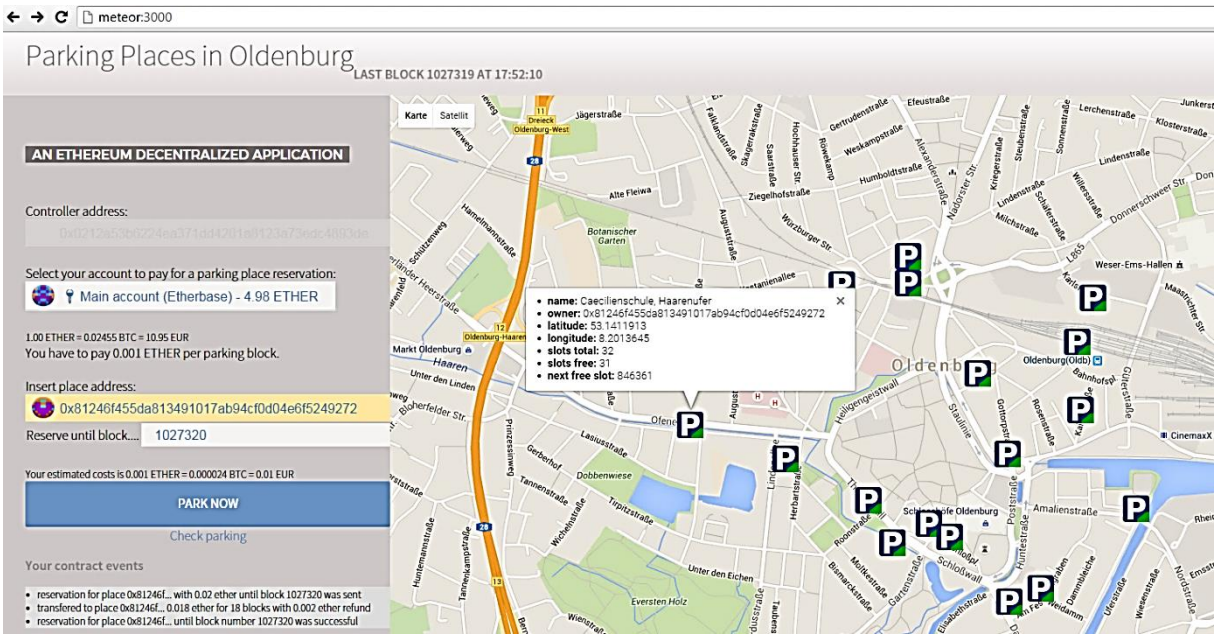

ABB. 32: RESERVIERUNG IN "PARKING-PLACES"

Abb. 32 zeigt die Dapp nach einer getätigten Reservierung über den Button „Park Now“ für die Dauer von 18 Blöcken auf dem Parkplatz Caecilienschule Haarentor, der anschließend noch 31 von 32 Slots frei hat. Hierfür wurden 0,02 ETH gesendet und der Balance des „Main account“ von 5 auf 4,98 ETH reduziert. Bei 18 Blöcken und 0,001 ETH pro Block wurden 0,002 ETH zurück erstattet. Die Prüfung einer Parkreservierung über den Button „Check parking“ würde nun bis einschließlich Block 1.027.320 eine positive Rückmeldung geben. Im vorherigen Kapitel wurden hierfür in Abb. 31 modale Dialoge gezeigt. Als Dapp User können keine Parkplätze oder Slots hinzugefügt werden. Der Use Case aus Kapitel 5.1 beschreibt die Anwendungsfälle zum Anzeigen von Parkplätzen und Slots, eine Kostenfunktion sowie die Reservierung und das Prüfen einer Reservierung.

Neben der Ausführung der Dapp über Docker Container besteht die Möglichkeit den Ethereum Client zusammen mit der Dapp als SPA ohne Meteor Installation direkt über das Hostsystem auszuführen. Die Installation des Ethereum Client `geth` und anderen kann der Homestead Dokumentation [Et16t] entnommen werden. Ist der `geth` Client installiert kann dieser über den Befehl aus Zeile 4 in Listing 10 gestartet werden mit `localhost` als `rpcaddr`. Nach der Synchronisation werden Accounts wie in Listing 11 erstellt und ETH transferiert. Das GitHub Repository wird wie in Zeile 4 aus Listing 12 geladen und die Dapp ausgeführt über die `index.html` in dem Verzeichnis `parking-places/bundled-parking-dapp`.

Im nächsten Kapitel werden mögliche Anwendungsszenarien der implementierten und evaluierten Dapp sowie der Blockchain Technologie und ihre Auswirkungen im Kontext von Smart Cities nach dem Modell von Manville aus Kapitel 2.4 und 2.5 diskutiert.

7 Auswirkungen auf Smart Cities

In diesem Kapitel wird die Dapp „Parking Places“ bestehend aus Contract und GUI der vorherigen beiden Kapitel ganzheitlich betrachtet und ein möglicher Einsatz in Smart Cities diskutiert. Anschließend werden Auswirkungen der Blockchain Technologie aus Kapitel 3 und 4 sowie der Dapp auf Smart Cities diskutiert.

7.1 Dapp „Parking Places”

Dieses Kapitel beschreibt die Dapp im Kontext von Smart Cities und diskutiert mögliche Anwendungsszenarien und Erweiterungen der Dapp in einer Smart City. Der Use Case der Dapp „Parking Places“ in Abb. 21 aus Kapitel 5.1 beschreibt das Parken auf öffentlichen Parkplätzen und besitzt nach dem Smart City Modell von Giffinger siehe Abb. 3 in Kapitel 2.4 die Eigenschaft Smart Mobility.

Strategische Ausrichtung von Smart Mobility sind Transport und IKT mit Faktoren von lokaler und (inter-)nationaler Anbindung, Verfügbarkeit der IKT Infrastruktur sowie nachhaltigen, innovativen und sicheren Verkehrssystemen. Diese können nach Manville et al. [Ma14] Straßenbahnen, Busse, Züge, Metro, PKWs, Motorräder, Fahrräder, Fußgänger usw. in einem oder mehreren Transportmodi z.B. Park & Ride umfassen, wobei saubere und nicht motorisierte Lösungen bevorzugt werden. Hierfür sind relevante und öffentliche Informationen in Echtzeit erforderlich, um Zeit und Kosten einzusparen wie auch den CO_2 Emission und Lärmbelästigung zu reduzieren. Darüber hinaus können diese Daten genutzt werden, um Dienste zu verbessern und Feedback zu geben. Die Dapp „Parking Places“ zeigt echtzeitnah die Auslastung öffentlicher Parkplätze, kann einen Parkplatz reservieren oder ein Parkticket auf Gültigkeit prüfen und unabhängig vom Standort verlängern. Hiermit werden Zeit und Kosten für die Parkplatzsuche und Verlängerung eines Parktickets verringert, wodurch sich CO_2 Emission und Lärmbelästigung zu reduzieren. Durch die dezentralisierte Blockchain sind alle Daten und Transaktionen öffentlich, historisiert und überall zu jederzeit verfügbar. So kann zum Beispiel die Auslastung eines Parkplatzes zu einem bestimmten Zeitpunkt bestimmt werden oder Trends ermittelt werden. Bei einer Reservierung fungiert der Dapp User als „Citizen as a Sensor“ dadurch dass durch die Transaktion der Reservierung ein belegter Slot echtzeitnah (ca. 3 Blöcke je 14 Sekunden) markiert wird. Somit wird jeder Parkplatzslot über die Dapp selbst zu einem Sensor.

Neben der wesentlichen Eigenschaft Smart Mobility kann die Dapp „Parking Places“ auch anderen Eigenschaften einer Smart City zugeordnet werden. Die Eigenschaft Smart Governance lässt sich über die Verwaltung öffentlicher Parkplätze durch die Stadtregierung identifizieren. Diese bietet „Parking Places“ als offenen und transparenten Dienst an unter Partizipation der Dapp User und Administration (`controller` und `owner`) der Parkplätze, Parkkosten, Parkdauer und Öffnungszeiten. Auch die Prüfung gültiger Parktickets und Überwachung von Parkplätzen und seinen Kapazitäten fällt in diesen Bereich.

Durch die dezentralisierte Blockchain sind der Contract, alle Daten und Transaktionen öffentlich und transparent überall zu jederzeit verfügbar. Bezogen auf der Eigenschaft Smart Living soll sich die von der Dapp angestrebte verbesserte Verkehrssituation (vermindertes Staurisiko, reduzierter CO_2 Emission und geringere Lärmbelästigung) sowie der Zeit- und Kostenersparnis positiv auf Wohnqualität und Touristenattraktivität auswirken. Ebenso kann das Prinzip der Blockchain basierend auf seinen kryptografischen Grundlagen die individuelle Sicherheit durch Schutz vor Manipulation fördern. Ein Parkticket kann in der Blockchain nicht verloren gehen und jederzeit nachgewiesen werden. Von einer besseren Verkehrssituation, einem Zeit- und Kostenersparnis sowie steigender Touristenattraktivität kann auch die Wettbewerbsfähigkeit profitieren. Hieran hängen u.a. die An- und Auslieferung von Waren, Besuche in Geschäften bedingt durch die Parkplatzsituation aber auch abhängig von der Lebensqualität der Arbeitsmarkt als Faktoren der Eigenschaft Smart Economy. Durch die Dapp werden Parkplätze selbst zu Sensoren und Parktickets in Papierform obsolet, so dass hierdurch Ressourcen eingespart werden können und innerhalb der Eigenschaft Smart Environment die Faktoren Management natürlicher Ressourcen und Umweltverschmutzung positiv beeinflusst werden. Zuletzt sind die Dapp User in der Eigenschaft Smart People mit einer neuen Technologie verbunden.

Wie in Kapitel 2.4 beschrieben muss eine Smart City Initiative oder Projekt mindestens eine dieser Eigenschaften besitzen, wobei eine Smart City aus mindestens einer Initiative mit mindestens einem Projekt besteht. Die Untersuchung von Manville et al. [Ma14] zeigt, wie Eigenschaften untereinander korrelieren anhand der Häufigkeit ihres Auftretens innerhalb von Initiativen. Smart Mobility und Smart Environment ist die häufigste gemeinsame Eigenschaft, während Smart Environment und Smart Governance am seltensten zusammen auftreten. Smart Mobility ist nach Smart Environment die häufigste Eigenschaft einer Smart City. Alle Eigenschaften stehen in direkter oder indirekter Verbindung zu den Dimensionen Technologie, Institution und Human (siehe Tab. 3 Kapitel 2.4). In den vorangegangenen Kapiteln wurde mit der Blockchain, Ethereum, der Implementierung und Evaluation des Dapp Prototyps vor allem die Dimension Technologie betrachtet. Innerhalb der Dimension Human geht es u.a. um Akzeptanz und Toleranz mit neuen Technologien und Paradigmen wie der Blockchain. Eine fortlaufende IKT Bildung und ein andauernder Lernprozess über die gesamte Bevölkerungsschicht sind nach Nam und Pardo [NP11] Schlüsselfaktoren. Für die Dapp „Parking Places“ sind Akzeptanz, Toleranz und Vertrautheit mit der Ethereum Blockchain von besonderer Bedeutung, da jeder Parkende einen Ethereum Account für das Auslösen und Bezahlen eines Parktickets benötigt. Die Dapp könnte hierbei so angepasst werden, dass alte Verfahrensweisen weiterhin möglich sind und hier z.B. der Contract selbst den Slot reserviert. In der Dimension Institution sind neben Vorschriften und Richtlinien die Partnerschaften zu Stakeholdern (Einwohner, Firmen, Organisationen etc.) enthalten. Dabei ist nach Nam und Pardo [NP11] Transparenz ein entscheidender Faktor für alle Aktivitäten. So muss wie in Kapitel 3.6.1 für die Dapp die gesetzliche Grundlage der Bezahlung von Parktickets mit der Währung ETH geprüft werden. Weiterhin können Partnerschaften zu Parkhausbetreibern neue Möglichkeiten eröffnen z.B. Parkhäuser in die Dapp zu integrieren.

Basierend auf den fünf Projekttypen aus Kapitel 2.5 gehört die Dapp „Parking Places“ zum Typ „intelligente Verkehrssysteme“ mit Fokus auf den Eigenschaften Smart Mobility und Smart Environment wobei der Schwerpunkt auf Smart Mobility liegt. Diese Projekte beinhalten nach Manville et al. [Ma14] oftmals die Verwendung von aktivem GPS und Fahrbahnsensoren zur Verbesserung des Verkehrsflusses, zur Staureduzierung und für das Monitoring von Verkehrsinformationen in Echtzeit. Solche Projekte liegen primär in der Verantwortung und Finanzierung durch den öffentlichen Bereich der lokalen Regierung mit zusätzlichen Stakeholdern im privaten Sektor zur Bereitstellung und Unterstützung von Technologien. Einwohner haben hier als Endnutzer oftmals keine Position. Laut Manville et al. [Ma14] fehlen für die betrachteten Projekte dieses Typs Nachweise für positive Effekte. Hier werden positive (indirekte) Effekte und Externalitäten (siehe Abb. 5 in Kapitel 2.5) vorhergesagt, wenn das Projekt auf Stadtebene ausgerollt wird. Stakeholder für das Smart City Projekt „Parking Places“ sind die lokale Regierung als Betreiber öffentlicher Parkplätze, Parkhausbetreiber, IKT-Unternehmen mit Expertise im Bereich Blockchain, Hersteller von Parkautomaten und letztendlich die Einwohner, welche die Blockchain als neues Paradigma und Technologie akzeptieren und unterstützen müssen.

Ein Referenzprojekt im Bereich Smart Mobility und intelligenten Verkehrssystemen welches von Manville et al. [Ma14] innerhalb einer Fallstudie behandelt wird ist das Projekt „The Smart Parking Network Barcelona“. Über kabellose Sensoren werden belegte Parkplätze identifiziert und an ein zentrales System des Parkhauses weitergeleitet, welches diese wiederum an eine zentrale Stelle weiterleiten. Die Informationen sind über eine mobile App in Echtzeit verfügbar, welche App User zum nächsten freien Parkplatz navigieren kann. Die durchschnittliche Zeit für das Auffinden eines freien Parkplatzes in Barcelona beträgt 15,6 min und kann auf 5 min reduziert werden. Dies verringert Lärmbelästigung und CO_2 Emission und vermeidet Kosten für neue Parkplätze durch effektivere Auslastung. Durch die Ausstattung aller Parkplatzslots mit Sensoren wurde das Projekt mittelfristig eingeplant. Weitere Projekte im Bereich Smart Mobility und intelligenten Verkehrssystemen verwenden ebenfalls Sensoren für die Messung des Lärmpegel, Verkehrsbewegungen und der CO_2 Emission. Diese werden den Einwohnern in Echtzeit über mobile Apps meist mit weiteren Informationen und Funktionen wie einer Navigation zur Verfügung gestellt. Monitoring und Trendanalyse sind wichtige Faktoren im Bereich der Verkehrsführung- und Planung.

Für eine mögliche Anwendung von „Parking Places“ als Projekt einer Smart City Initiative im Bereich Smart Mobility und intelligenten Verkehrssystemen gibt es mehrere Szenarien, die in Tab. 31 aufgeführt werden.

Szenario	Beschreibung
1.	Ein Parkplatz wird über eine mobile Dapp für einen bestimmten Zeitraum reserviert und bezahlt. Das Fahrzeug bekommt ein QR-Code Aufkleber mit dem Ethereum Account über den Transaktion durchgeführt werden. Der Vorteil hierbei ist das keine Parkautomaten benötigt werden und Kosten eingespart werden können. Ein Nachteil ist, dass für das Parken die Nutzung der mobile Dapp zwingend erforderlich ist.

	Es wird ein leichtgewichtiger Ethereum Client (aktuell in der Entwicklung) und eine Internetverbindung benötigt. Die Prüfung von gültigen Parktickets wird innerhalb des Contracts über die Funktion getReservedBlock abgebildet und benötigt die Adresse des Parkenden und des Parkplatzes. Die Dapp scannt die QR-Codes und validiert sie auf Gültigkeit basierend auf der aktuellen Blocknummer oder Zeit.
2.	Die Transaktion wird über eine Web Wallet abgeschickt, da hierfür kein Client benötigt wird. Jeder Parkplatz erhält einen QR-Code mit der Contract Account Adresse als Empfänger der Transaktion. Hierfür ist ein eigener Contract erforderlich, der über seine Fallback Funktion die Transaktion entgegen nimmt und anhand des zu überweisenden Betrags die Blocknummer für das Parkende berechnet. Es gibt drei Funktionen über die eine Reservierung und Bezahlung nach Betrag, Block oder Zeit vorgenommen werden kann. Sollte jeder Parkplatz einen eigenen Contract besitzen muss ein übergeordneter Contract erstellt werden, der die Kapazitäten der freien Slots aller Parkplätze überwacht bzw. über jede Änderung in der Verfügbarkeit benachrichtigt wird und ein Event triggert z.B. für die Aktualisierung der Dapp Views.
3.	Neben oder statt Parkplätzen werden Parkhäuser angebunden. Diese und einige Parkplätze besitzen Schranken, die sich nur dann öffnen wenn ein gültiges Parkticket eingeworfen wird. Hier kann analog zu slock.it (siehe Tab. 18 in Kapitel 4.3) eine Zugangskontrolle über den Contract erfolgen, der prüft ob die Account Adresse des Parkenden eine gültige Reservierung besitzt. Parkhäuser sind oft an Parkleitsystemen und Parkautomaten angebunden. In Kooperation mit den Betreibern könnten diese in die Dapp integriert werden und bestenfalls die Reservierung und Bezahlung von Parktickets hierüber abgewickelt werden. Durch die Zugangskontrolle müssen keine Parktickets auf Gültigkeit geprüft werden. Vorteil ist, dass bei vorzeitigem Parkende der Slot wieder freigegeben werden kann. Der Slot im Contract müsste um ein Flag erweitert werden. So kann die Effizienz von Parkkapazitäten weiter optimiert werden.
4.	In einer Hybrid-Lösung ist die Anpassung der Parkautomaten erforderlich, die mit dem Internet verbunden werden müssen und analog zum zweiten Szenario die Transaktion über eine Web Wallet abschicken. Der Automat muss den QR-Code der Account Adresse des Parkenden einscannen können, damit dieser über die Angabe eines Betrages die Transaktion an die Account Adresse des Parkplatzes über seine Passphrase signiert und bestätigt. Der Blockchain Explorer EtherCamp [Et16d] bietet analog hierzu die Funktion „Transfer Balance“. So kann der Parkende entweder über die Dapp oder Web Wallet auf seinem mobilen Endgerät eine Reservierung durchführen oder über den Parkautomaten. Der QR-Code ist wie Parktickets sichtbar im oder am Fahrzeug zu platzieren, falls dieses kontrolliert wird.
5.	In einer Hybrid-Lösung ist die Anpassung der Parkautomaten erforderlich, die mit dem Internet und Ethereum verbunden werden müssen. Hier kann der Parkende entweder über seinen Ethereum Account den Parkplatz reservieren oder über eingeworfenes Bargeld. Letzteres startet intern eine Reservierung, wobei der Contract selbst über seine Adresse diese ausführt und somit einen Slot belegt. Dabei wird die anhand der eingeworfenen Währung über den Price Feed der entsprechende ETH Betrag und hieraus das Parkende ermittelt. In diesem Fall bekommt der Parkende ein Parkticket ausgedruckt, welches analog zum QR-Code sichtbar im oder am Fahrzeug zu platzieren ist. Auch hier kann der Parkende entweder den Parkautomaten oder die Dapp oder Web Wallet auf seinem mobilen Endgerät nutzen. Dieses Szenario lässt sich auch auf Parkhäuser übertragen

TAB. 31: SZENARIEN FÜR "PARKING PLACES" IN SMART CITIES

Für alle Szenarien aus Tab. 31 gilt dass eine Transaktion über mindestens einen Block bestätigt werden muss. Mit der Ethereum Version Homestead wurde das durchschnittliche Zeitintervall zwischen zwei aufeinander folgenden Blöcken von 17 auf 14 Sekunden reduziert. Über den GAS Preis kann die Priorität der Transaktion beeinflusst werden, so dass Miner diese früher verarbeiten. Mit Versand und Empfang muss ca. eine halbe Minute eingeplant werden. Sollte nach ca. 30 Sekunden keine Bestätigung erfolgen, kann über den Hash der Transaktion geprüft werden ob diese abgebrochen oder in Bearbeitung ist. Weiter ist die Angabe der Parkdauer als zukünftige Blocknummer dem Benutzer nicht vertraut. Daher sollte die Parkdauer bekannter Weise in Minuten angegeben werden. Hierüber kann über das durchschnittliche Zeitintervall von 14 Sekunden die Blocknummer als Parkende ermittelt werden. Da der Durchschnitt auf einen Tag berechnet wird und hierüber temporäre Abweichungen aufgefangen werden, sollte das Parkende als Timestamp zusätzlich in den Slot im Contract aufgenommen werden, um eindeutig Parktickets auf ihre Gültigkeit zu bestimmtem Zeitpunkt oder Blocknummer validieren zu können. Weitere Parameter die bei der Anwendung von „Parking Places“ berücksichtigt werden müssen sind Öffnungszeiten, Parkkosten, maximale Parkdauer und besondere Parkplätze wie z.B. Behindertenparkplätze. Die Anzahl freier Parkplätze, Öffnungszeiten und Standorte können auch für Navigations- und Parkleitsysteme verwendet werden. Der Dapp Prototyp muss auf einen mobilen Client portiert und für das entsprechende Szenario aus Tab. 31 adaptiert werden.

Nach der Definition aus Kapitel 2.5 von erfolgreichen Smart City Initiativen müssen konkrete Auswirkungen und Ergebnisse präsentiert werden können, die messbar und skalierbar sind. Diese werden in individuellen Wert, Einsparungen und positive externe Effekte als Nutzlevel eingeteilt (siehe Abb. 5 aus Kapitel 2.5). „Parking Places“ besitzt über seine Infrastruktur und IKT Lösung bestehend aus dem Ethereum Netzwerk und angebundenen Parkhäusern und Parkplätzen wie auch mobilen Dapps einen individuellen Wert. Einsparungen werden durch Zeit und Kosten sowie Reduktion der CO_2 Emission (Ziel aus Europa 2020) angestrebt über direkte Navigation zum nächsten freien Parkplatz, ortsunabhängiger Reservierung von Parktickets und Verzicht von Parkautomaten in Szenario 1 und 2 aus Tab. 31 durch mobile Dapps. Positive externe Effekte sind z.B. Innovation durch die Blockchain als neues IKT Paradigma oder der anfangs beschriebenen Stimulationen weiterer Eigenschaften der Smart City. Als Indikatoren hierfür können Sensoren zur Messung der CO_2 Emission, des Lärmpegels sowie Bewegungs- und Drucksensoren zur Messung von Verkehrsbewegungen dienen. Darüber hinaus können über Ethereums historisierte Contract Account States die Auslastung von Parkplätzen und Parkhäusern heran gezogen werden. Dabei sind die durchschnittliche Parkdauer, Anzahl von belegten und unbelegten Parkplätzen wie auch die Einnahmen über Parktickets für Monitoring und Trendanalyse relevant. Diese wären mit aktuellen und vergangenen Daten sofern vorhanden zu vergleichen. Szenarien 1 und 2 aus Tab. 31 sind über das Hinzufügen weiterer Parkplätze zum Contract gut skalierbar. Szenarien 4 und 5 aus Tab. 31 sind bedingt skalierbar, da Parkautomaten ausgetauscht bzw. angepasst werden müssen. Szenario 3 aus Tab. 31 ist am wenigsten skalierbar, da dies von der Kooperation mit den jeweiligen Parkhausbetreibern abhängt. Szenarien 3 bis 5 aus Tab. 31 sind bedingt durch

die Automatenintegration am kostenintensivsten. Für das Überprüfen von Parktickets sind für alle Szenarien außer Szenario 3 aus Tab. 31 mobile Endgeräte anzuschaffen, die mit einer mobilen Dapp und einem leichtgewichtigen Ethereum Client ausgestattet sind, der sich über das Internet mit dem Ethereum Netzwerk verbindet.

Das Referenzprojekt „The Smart Parking Network Barcelona“ im Bereich Smart Mobility und intelligenten Verkehrssystemen profitiert nach Manville et al. [Ma14] stark von allen in Tab. 4 in Kapitel 2.5 aufgeführten Erfolgsfaktoren Vision, Personen und Prozess. Für den Faktor Vision sind „Quick-Wins“ und Inklusion bzw. die Beteiligung aller Einwohner unabhängig von Alter und Herkunft entscheidend. Mit Szenario 5 aus Tab. 31 benötigt der Parkende nicht zwingend einen Ethereum Account und kann in bekannter Weise ein Parkticket einlösen. Ein „Quick-Win“ kann am ehesten mit Szenario 1 aus Tab. 31 erzielt werden, da keine Automaten erforderlich sind sondern ein mobiles Endgerät mit Internetverbindung, einem leichtgewichtigen Ethereum Client und der mobilen Dapp. Hierüber werden alle ohne mobiles Endgerät mit Internetverbindung und Ethereum Account ausgeschlossen. Innerhalb des Erfolgsfaktors Personen ist ein nutzerzentrierter Ansatz über Crowdfunding und Crowdsourcing zu verfolgen. So kann ein ParkingCoin als Token und kryptografische Währung erstellt werden, mit dem die Benutzer später ein Parkticket bezahlen können und bspw. Parkautomaten finanzieren. Ethereum basiert auf diesem Ansatz und beschreibt ein Contract für einen Crowdsale in einem Tutorial [Et16r]. Die (Weiter-)Entwicklung von leichtgewichtigen Ethereum Clients und mobilen Dapp kann im Rahmen von Kollaboration und Kooperation über Crowdsourcing erfolgen. Der Prozess als letzten Erfolgsfaktor sieht eine lokale Koordination des Projekts vor und alle Daten und Informationen sollen öffentlich verfügbar sein. Verteiltes Wissen und Lernen vor allem in Bezug auf die Ethereum und Blockchain Technologie sind entscheidend. Über eine Kooperation mit Herstellern von (Park)-Automaten im Rahmen eines „Living Lab“ nach Schuurmann et al. [SBM12] kann eine offene, nutzerzentrierte, innovative Umgebung für Entwicklung und Evaluation eines Automaten über die Ethereum Blockchain erfolgen. Hierbei müssen zwecks Akzeptanz auch die User Experience (DIN EN ISO 9241-210) in einem mobilen Kontext berücksichtigt werden.

Im nächsten Kapitel wird die Blockchain als neues IKT Paradigma über Smart Mobility hinaus auf alle Eigenschaften einer Smart City diskutiert. Hierbei werden die Vorteile von Ethereum als dezentralisiertes Framework für die Ausführung von Smart Contracts hervorgehoben.

7.2 Blockchain Technologie

In dem vorherigen Kapitel wurde der Use Case und Dapp-Prototyp „Parking Places“ im Kontext von Smart Cities diskutiert. An dieser Stelle wird die Blockchain Technologie in Smart City Definitionen und Eigenschaften im Modell von Manville eingeordnet. Dabei werden die Blockchain Versionen und Anwendungsszenarien für Dapps sowie die Ethereum Plattform berücksichtigt.

Alle Smart City Definitionen aus Kapitel 2.3 basieren auf verwandten Konzepten aus Kapitel 2.2 die Nam und Pardo [NP11] in die Dimensionen Technologie, Institution und Human eingeteilt haben. Viele dieser Definitionen legen ihren den Schwerpunkt auf die Bedeutung von Information -und Kommunikationstechnologien. In dem Modell einer Smart City von Manville et al. [Ma14] werden die Schwerpunkte unterschiedlicher Smart City Definitionen als Eigenschaften und die Dimensionen verwandter Konzepte zusammengeführt. Nach ihrer Definition von ist eine Smart City erfolgreich, wenn sie ein ausgeglichenes Portfolio dieser Eigenschaften und Dimensionen besitzt. Viele der von Manville betrachteten Projekte mit den Eigenschaften Smart Mobility und Smart Environment verwenden Netzwerke mit Sensoren und Apps aus den Bereichen IoT und Mobile Computing. IoT ist mit 1,1 Billionen verbundenen Geräten in 2015 (Gartner [Ga14]) und nach McKinsey [Mc15] die drittgrößte disruptive Technologie hinter dem mobilen Internet und wird im IBM Businessreport „Device Democracy“ [BP14] mit der Blockchain als „Internet of Decentralized Autonomous Things“ bzw. die Demokratisierung der digitalen Welt beschrieben. Swan [Sw15] beschreibt das aktuelle Jahrzehnt als Zeitalter einer verbundenen Welt mit der Blockchain, Wearables, IoT-Sensoren, Smart Homes, Smart Cars und Smart Cities. Dabei bezeichnet sie die Blockchain als fünftes revolutionäre Computerparadigma nach Mobile Computing und sozialen Netzwerken aus dem letzten Jahrzehnt.

Die Blockchain als IKT wurde mit Bitcoin 2008 entwickelt um Schwächen elektronischer Zahlungssysteme zu beheben. Diese sind limitierte hohe Transaktionskosten, limitierte Transaktionsgrößen, reversible Transaktionen und das Vertrauen in Drittparteien. Die Blockchain ist eine kryptografisch verifizierte, pseudonymisierte und über ein P2P- Netzwerk dezentrale und transparente Datenbank. Über das auf Hashberechnung basierte Mining werden Blöcke mit Transaktionen innerhalb dieser Datenbank chronologisch verkettet und so die kryptografische Währung (z.B. BTC oder ETH) geprägt. Über die integrierte stapelverarbeitende Skriptsprache werden Smart Properties, Smart Contracts und autonome Agenten wie Dapps unterstützt. Im Bereich IoT- und M2M-Kommunikation können hierüber z.B. Mikrozahlungen abgebildet werden. Smart Properties und Smart Contracts basieren darauf, dass sich zwei oder mehr Parteien nicht kennen oder vertrauen müssen. Dies ist aufgrund der hohen Anzahl verbundener IoT-Geräte ein entscheidender Vorteil. Weitere Vorteile sind der Verzicht auf zentralisierte Infrastruktur und die hiermit verbundene Kostenersparnis sowie bedingt durch das dezentralisierte P2P-Netzwerk die Eliminierung eines „Single Point of Failure“. Dies führt zu einer nahezu 100%-igen Ausfallsicherheit. IoT-Geräte müssen als Smart Properties einen Client besitzen und mit dem Internet und dem Blockchain Netzwerk verbunden sein. Ein Nachteil ist die Größe der Blockchain und Zeit für die Erstellung und Propagierung eines Blocks innerhalb des Netzwerks. In der Bitcoin Blockchain wird ca. alle zehn Minuten ein neuer Block hinzugefügt, was der Wartezeit für eine Transaktionsbestätigung entspricht. Die meisten IoT-Geräte können eine vollständige Blockchain mit über `50 GByte` und die erforderliche Rechenleistung nicht aufbringen. Sowohl die Bitcoin mit SPV als auch Ethereum besitzen leichtgewichtige Clients für IoT- und mobile Geräte. Dies ist ein wichtiges Kriterium für den Einsatz der Blockchain in Smart Cities

neben der Verfügbarkeit (Netzabdeckung) von mobilem Internet z.B. über freie WLAN Spots. Die globale Verfügbarkeit und Anbindung des Blockchain Netzwerk ist der Eigenschaft Smart Mobility zuzuordnen. Die Einsparung zentralisierter Infrastruktur und ggf. der Ausbau des mobilen Internets und IoT-Netzwerke gehört auch zu der Eigenschaft Smart Environment. Pseudoanonyme Identifikation und Authentifizierung für Transaktionen zwischen fremden, sich nicht vertrauenden Personen ist ein Merkmal der Eigenschaft Smart People.

In Kapitel 3.6 wurden Anwendungsbereiche für die Blockchain von Zahlungssystemen über Smart Properties und Smart Contracts bis zu weiterführenden Anwendungen und App-Coins beschrieben. Die Entwicklung von Bitcoin bzw. der Blockchain stammt aus dem Bereich E-Commerce wie Nakamoto [Na08] im ersten Satz seines Papers *„Commerce on the Internet has come to rely almost exclusively on financial institutions serving as trusted third parties to process electronic payments."* verdeutlicht. Damit ist Bitcoin als kryptografische Währung und digitales Zahlungssystem in Smart Cities der Eigenschaft Smart Economy zuzuordnen. Diese umfasst nach Manville et al. [Ma14] E-Business und E-Commerce wie auch innovative Dienste und Geschäftsmodelle (z.B. Mining), welche über die integrierte Skriptsprache der Blockchain flexibel sind. Wesentliche Vorteile sind unlimitierte Transaktionsgrößen (Mikrozahlungen), niedrige Transaktionskosten wie auch die kryptografische Sicherheit und Pseudonymität. Vor- und Nachteile der Währung Bitcoin als Tauschmedium und Wertanlage sind in Tab. 10 aus Kapitel 3.6.1 aufgeführt. Innerhalb der Eigenschaft Smart Governance verbunden mit der Dimension Institution sind dabei rechtliche Grundlagen kryptografischer Währungen zu prüfen. Eine wesentlicher Bestandteil der Eigenschaft Smart Governance ist nach Manville et al. [Ma14] die Komponente Öffentlichkeit (OpenData) und die Kooperation über lokale, regionale, internationale wie auch globale Partnerschaften und Netzwerke im privaten und öffentlichen Sektor.

Mit einem Smart Contract aus dem Anwendungsbereich der Blockchain 2.0 (siehe Kapitel 3.6.2) lässt sich mit wenigen Zeilen Code ein Crowdfunding abbilden. Dieser ist ein Beispiel der Eigenschaft Smart Economy in dem Bereich Entrepreneurship. Crowdfunding wird als nutzerzentrierter Ansatz nach dem „Bottom-Up"-Prinzip in den Erfolgsfaktoren für Smart Cities in Tab. 4 aus Kapitel 2.5 genannt. Smart Contracts führen autonom einen initial definierten Code in einem dezentralisiertem Netzwerk aus und steuern digitale Assets wie z.B. Smart Properties. Ein Smart Property ist ein Objekt, dessen Eigentum über die Blockchain kontrolliert wird und direkten Zugang zur Blockchain hat. Da sowohl physische als auch nicht-physische Objekte als Smart Property enkodierte digitale Assets über Smart Contracts gesteuert werden können, sind alle Eigenschaften und Dimensionen einer Smart City einbezogen. [Sw15] bezeichnet die Blockchain 2.0 als Dezentralisierung der Märkte durch den Transfer digitaler Assets. Die Blockchain 3.0 aus Kapitel 3.6.3 bezieht sich insbesondere auf die Eigenschaften Smart Governance, Smart Living und Smart People. Hier sind dezentralisierte Anwendungen mit Contracts und App-Coins als Smart Properties in unterschiedlichen Bereichen enthalten. Anwendungen und App-Coins im Bereich Regierung beziehen sich auf Smart Governance, der Bereiche Gesundheit und Kultur auf Smart Living und in den Bereichen Wissenschaft, Kunst und Bildung auf die Eigenschaft Smart People.

Ethereum ist ein dezentralisiertes Framework zur Ausführung von Smart Contracts mit eigener Blockchain, Protokoll, P2P-Netzwerk und Meta-Coin mit anwendungsspezifischen Daten im Bereich der Blockchain 2.0. Wie auch die Bitcoin Blockchain ist Ethereum als Open Source öffentlich verfügbar und basiert auf dem Konzept Crowdsourcing einem Merkmal der Eigenschaft Smart Governance. Ethereum selbst wurde 2014 durch eines der größten Crowdfundings über den Verkauf der kryptografischen Währung ETH finanziert und ist seit März 2016 in der zweiten Version Homestead live. Ethereum besitzt gegenüber Bitcoin einige Vorteile, die auf Smart City Projekte übertragbar sind. In der Ethereum Blockchain wird alle 14 Sekunden ein neuer Block hinzugefügt, so dass eine synchrone Kommunikation zwischen Transaktion und Bestätigung mit kurzer Wartezeit möglich ist. Ethereum besitzt eine Turing-vollständige Skriptsprache zur Ausführung von Contracts. Jede Operation und Speicher einer Transaktion wird mit `GAS` berechnet, um Endlosschleifen und DOS Angriffe zu verhindern. Außerdem wird mit SHA-3 eine sicherere und standardisierte kryptografische Hashfunktion eingesetzt. Von der Blockzeit, der Turing-vollständigen Skriptsprache und der Verwendung von SHA-3 sind unmittelbar alle Eigenschaften einer Smart City betroffen, in denen Smart Properties über die Blockchain angebunden sind. Ethereum bietet über die Turing-vollständige Skriptsprache im Vergleich zu Bitcoin eine höhere Flexibilität in der Erstellung von dezentralisierten Anwendungen in allen Bereichen. Ethereum verwendet mit States und Accounts ein simpleres und feingranulares Konzept für Balance mit mehreren Zuständen. Hierbei wird für geringeren Speicherverbrauch und Rechenzeit ein Merkle Patricia Baum mit RLP Enkodierung verwendet, um insbesondere mobile Geräte im Bereich IoT (Smart Mobility und Smart Environment) zu unterstützen. Ein Prinzip von Ethereum ist die „Sandwich“ Komplexität. Anwender und Entwickler kennen nur die simpleren, äußeren Architekturschichten. Clients und Programmiersprachen für Contracts sind verständlich und einfach gehalten sowie über zahlreiche Beispiele gut dokumentiert. Ethereum [Et16s], [Et16r] beschreibt Contracts für Crowdfunding, eigene Währung oder DAO. Die Beispiele orientieren sich dabei sowohl inhaltlich als auch didaktisch an den Eigenschaften Smart Economy, Smart People und Smart Governance. Dabei wird über diverse Communities und Entwicklerressourcen wie Frameworks, APIs, HowTos und Wikis die Entwicklung von Dapps über der Erstellung von Contracts und GUI unterstützt.

Eine logische Konsequenz der Blockchain als innovative IKT, dem Schwerpunkt von IKT in Smart City Definitionen und dem Modell von Manville ist die Integration der Blockchain innerhalb von Smart Cities. Die Blockchain wird von Swan [Sw15] als Paradigma analog zum Internet bezeichnet und betrifft alle Eigenschaften und Dimensionen von Smart Cities. Im IBM Businessreport „Device Democracy“ [BP14] wird die Blockchain als Prinzip für die Sicherung, Verwaltung, Skalierbarkeit, Konnektivität von IoT-Netzwerken und M2M-Kommunikation basierend auf Transaktionen beschrieben. Viele Smart City Initiativen und Projekte nutzen Sensoren und Aktoren im Kontext von IoT und M2M. Mit der Ethereum Plattform können Dapps als autonome Agenten bestehend aus Smart Properties und Smart Contracts sowie geräteabhängigen GUI über die Blockchain mit anwendungsspezifischen Daten kontextunabhängig mit wenigen Zeilen Code erstellt werden.

Im nächsten und letzten Kapitel werden Entwicklung und Evaluation des Dapp-Prototyps, die Ethereum Plattform und Blockchain Technologie sowie als Anwendungsdomäne das Modell Smart Cities zusammengefasst. Außerdem wird ein Ausblick über weitere Entwicklungen in diesen Bereichen gegeben mit weiterführenden Themen, welche in dieser Arbeit nicht näher betrachtet wurden.

8 Schlussbetrachtung

In den vorherigen Kapiteln wurden eine Einführung in die Themengebiete Smart Cities und Blockchain als technologische Entwicklung von Bitcoin gegeben. Darüber hinaus wurde mit der Ethereum Plattform eine Weiterentwicklung der Blockchain betrachtet mit der ein Prototyp einer dezentralisierten Anwendung (Dapp) umgesetzt und evaluiert wurde. Diese dient als PoC für die Entwicklung von Dapps und dem Konzept Blockchain. Die Dapp wurde dabei im Rahmen eines Use Case im Kontext einer Smart City umgesetzt und anschließend zusammen mit dem Konzept der Blockchain und der Plattform Ethereum in Smart Cities diskutiert. Dieses Kapitel fasst die gewonnenen Erkenntnisse und Ergebnisse dieser Arbeit zusammen. Anschließend werden weitere Entwicklungen und Themengebiete im Umfeld Smart Cities, Smart Parking, Blockchain und Ethereum in einem Ausblick betrachtet.

8.1 Zusammenfassung

In 2015 leben laut UN ca. 7,3 Billionen ca. 54% weltweit in Städten. In 2050 werden nach einer Prognose der UN ca. 66% von ca. 9,5 Billionen Menschen in Städten leben. Diese Urbanisierung erfordert effektives Energie- und Ressourcenmanagement und smarte Lösungen basierend auf dem Fortschritt von IKT. Der G7 Klimagipfel und Europa 2020 haben internationale Leitziele hierfür festgelegt, an denen sich auch die Studie „Mapping smart cities in the EU“ von Manville orientiert. In 2015 sollten 1,1 Billionen IoT-Geräte, in 2020 schon 26 Billionen und in 2025 alle Automobile mit dem Internet verbunden sein. Smart Lösungen sind als IoT-Anwendungen häufig in Smart Cities zu finden. IBM verwendet im Bereich IoT die Blockchain IKT als Strategie in ihrem Konzept „Internet of Anonymous Decentralized Things“ und als Grundlage ihres Projekts ADEPT.

Die Historie von Smart Cities geht zurück auf die frühen 90er. Im Rahmen dieser Arbeit wurden Begriffe, verwandte Konzepte und Definitionen von 2010 bis 2014, Modell und Erfolgsfaktoren einer Smart City vorgestellt. Die ISO arbeitet an einer Standardisierung und Normung aber bislang gibt es noch keine einheitliche oder allgemeingültige Definition. Der Begriff Smart wird im Bereich IKT mit intelligent assoziiert und in der Literatur häufig synonym verwendet. In der Städteplanung wird mit Smart eine Ideologie und Strategie assoziiert. Definitionen einer Smart City besitzen einen ganzheitlichen Ansatz, Schwerpunkt auf IKT, Orientierung an internationalen Leitzielen oder unterscheiden nach dem „Bottom-Up“ und „Top-Down“ Prinzip. Giffinger hat 2007 ein Modell für eine Smart City erstellt basierend auf den Eigenschaften Smart Environment, Smart Mobility, Smart Economy, Smart People, Smart Governance und Smart Living. Manville hat dieses Modell um Dimensionen von Nam und Pardo erweitert. Diese sind Institution, Human und Technologie und umfassen verwandte Konzepte von Smart Cities. In der Studie „Mapping smart cities in the EU“ von Manville wurden 468 europäische Städte analysiert und hinsichtlich ihres Erfolgs auf die Leitziele von Europa 2020 bewertet.

Im zweiten Teil dieser Arbeit wurde mit der Blockchain die größte technologische Innovation nach mobilem Internet und sozialen Netzwerken beschrieben. Die Blockchain wurde 2008 von Nakamoto als Technologie hinter Bitcoin erfunden, um Schwächen elektronischer Zahlungssysteme im Internet zu beheben. Sie löst zwei grundlegende Probleme aus 20 Jahren Forschung mit kryptografischen Währungen und 40 Jahren Forschung in der Kryptografie. Die Blockchain ist eine verteilte, öffentliche und dezentrale Datenbank basierend auf kryptografischen Grundlagen wie AES, ECDSA, SHA und Merkle Bäume, die in Kapitel 3.2 behandelt wurden. Die Blockchain fungiert als „Timestamp Server" und verkettet alle zehn Minuten einen Block über seinen Hash (Timestamp). Mining bezeichnet hierbei das Erstellen eines Blocks mit Transaktionen über komplexe Hashberechnung (2015 mit 10^{15} Hashes pro Sekunde) wofür der Miner 25 BTC (am 24.01.2016 ca. 400 USD) erhält. Die Blockchain ist (am 18.01.2015 mit ca. 52 `GByte`) verteilt über ein P2P-Netzwerk mit dem Konsens der „längsten Kette" wobei jeder verbundene Client Blöcke erhält, kryptografisch verifiziert, in einer KV-Datenbank lokal speichert und über das Netzwerk weiterleitet. Leichtgewichtige SPV Clients wie mobile Geräte können auch nur die Blockheader (am 18.01.2016 mit ca. 30 MByte) speichern. Bitcoin nutzt eine nicht Turing-vollständige, stapelverarbeitende Skriptsprache für Transaktionstypen zum Sperren und Entsperren von Ein- und Ausgängen. Diese basieren auf den Bitcoin Adressen als Pseudonyme der Benutzer. Ein grundlegendes Prinzip hierbei ist das Bitcoin und die Blockchain keine zentrale Autorität als „Single Point of Failure" besitzt und sich Transaktionspartner nicht kennen oder trauen müssen. Im Rahmen dieser Arbeit wurde die Blockchain mit seinen Transaktionen, Blöcken und P2P-Netzwerk in Aufbau und Funktion analysiert.

In Kapitel 3.6.1 wurden Vor- und Nachteile von Bitcoin als Tauschmedium und Wertanlage gegenüber gestellt. Bitcoin ist die größte kryptografische Währung. Am 24.01.2016 lag das Marktkapital bei über 6 Milliarden USD. Die Gebühr einer Transaktion lag am 26.01.2016 bei 0,1 USD Cent im Vergleich zu 5,91 - 7,52% im dritten Quartal 2015 der durchschnittlichen, weltweiten Gebühren. Am 21.05.2016 gab es 667 ATMs, 7789 Händler und am 27.01.2016 wurden 1,5 Millionen USD durch Mining verdient. Mit der Blockchain 2.0 werden die Anwendungsbereiche von Währungen und Zahlungssysteme ausgeweitet über Smart Properties und Smart Contracts auf dezentralisierte Märkte. Die Blockchain dient als Register für den Besitznachweis digitaler Assets enkodiert als Smart Properties, die hierbei über Smart Contracts gesteuert werden. Ein Smart Property ist ein Objekt dessen Eigentum über die Blockchain kontrolliert wird und direkten Zugang zur Blockchain besitzt. Als Beispiel wurde ein Automobil angeführt, welches als Smart Property enkodiert ist und direkten Zugang zur Blockchain besitzt. Nur der Eigentümer kann dieses entsperren und nutzen. Smart Contracts werden über die Skriptsprache von Bitcoin mit seinen Transaktionstypen abgebildet und führen autonom einen definierten Code in einem dezentralisierten Netzwerk aus. In diesem Kontext wurden auch die Begriffe DAO, DAS, DAC und autonome Agenten als Dapps zusammengefasst und definiert. Für viele Apps gibt es eine dezentrale Äquivalenz wie z.B. mit Storj für die Dropbox. Smart Contracts wie auch Smart Properties erfordern das Speichern anwendungsspezifischer Daten. Dadurch sind Meta-Coins und Blockchain 2.0

Protokolle wie Ethereum entstanden. Die Blockchain 3.0 erweitert die bislang bekannten Szenarien um die Bereiche Kunst, Kultur, Bildung, Wissenschaft, Gesundheit und Regierung. Damit sind alle Eigenschaften einer Smart City nach dem Modell von Giffinger und Manville abgedeckt. Mit der Blockchain lassen sich alle Quantitäten und Aktivitäten verwalten und organisieren wie in Kapitel 3.6.3 anhand vieler Beispiele gezeigt wurde.

Ethereum ist durch das fünftgrößte Crowdfunding über Bitcoin in 42 Tagen mit ca. 83 Millionen USD im Juni 2014 finanziert worden. Ende 2013 eröffnete Erfinder und Gründer Vitalik Buterin (Chefautor des Bitcoin Magazin) ein Whitepaper mit der Vision eines Weltcomputers basierend auf der Blockchain 2.0. Anfang 2014 gab es ein Yellowpaper mit der Formalisierung des Protokolls, ein erstes Testnetzwerk und hierzu zwei vollständig implementierte Clients. Ein Jahr nach dem Crowdfunding wurde mit Frontier für Miner die erste Version veröffentlicht und während dieser Arbeit im März 2016 mit Homestead für Dapp-Entwickler die zweite Version live gestellt. ETH wurde im Verlauf dieser Arbeit bemessen am Marktkapital zur zweistärksten, kryptografischen Währung hinter Bitcoin. Beide basieren auf Open Source und dem Blockchain Prinzip über ein verteiltes, dezentrales P2P-Netzwerk. Im Rahmen dieser Arbeit wurde die Plattform Ethereum für das Erstellen und Ausführen von Smart Contracts betrachtet. Dabei wurden dessen grundlegenden Prinzipien und Vorteile gegenüber Bitcoin herausgestellt. Ethereum verwendet bei der Blockchain über das GHOST Protokoll eine Baumstruktur für eine Blockzeit von 14 Sekunden. Darüber hinaus werden können über Accounts, Merkle Patricia Bäumen mit RLP Enkodierung in einer Turing-vollständigen, stapelverarbeitenden Skriptsprache beliebig Daten unabhängig von der Anwendung in der Blockchain gespeichert werden. Ethereum berechnet `GAS` für die Ausführung von Operationen auf dem Stapel innerhalb der EVM als zusätzliche Sicherheit wie auch SHA-3 als aktuellen Standard. Auf der Skriptsprache basieren wie Solidity höhere Programmiersprachen für Smart Contracts, die auch im Rahmen dieser Arbeit verwendet wurde. Das Entwicklungsteam von Ethereum hat hierfür einen Online Compiler als Dapp erstellt. Eine Dapp wird definiert als ein oder eine Gruppe von Contracts und einem GUI. Neben der Blockchain besteht Ethereum in seiner Vision aus Whisper und Swarm für P2P-Messaging und -Filesharing, welche in der nächsten Version Metropolis von Ethereum eingesetzt werden sollen. Auch mobile, leichtgewichtige Clients in Java (Android) und C++ sind aktuell noch in der Entwicklung. Im Rahmen dieser Arbeit wurden Clients, Dapps und Contracts mit Ethereum vorgestellt. Der Go Client `geth` ist mit 90% der meistgenutzte Client. In Kooperation mit Microsoft, Consensys und BlockApps wurde 2015 eine Haskell und Java Implementierung erstellt, Solidity in die IDE Visual Studio und die Ethereum Blockchain als Service in Microsofts Cloudlösung Azure integriert. Während dieser Arbeit hat mit „The DAO“ das größte Crowdfunding über einen Smart Contract in der Ethereum Plattform mit 160 Millionen USD im Mai 2016 für 28 Tage stattgefunden. Die Gründung der dezentralen, autonomen Organisation basierend auf Smart Contracts zeigt zunehmende Akzeptanz und Unterstützung der Blockchain und Ethereum.

Neben den Konzepten von Smart Cities, der Blockchain Technologie und der Ethereum Plattform war das Ziel dieser Arbeit die Erstellung eines PoC für eine auf der Ethereum

Plattform und Blockchain basierenden, dezentralen Anwendung im Kontext einer Smart City. Hierfür wurde innerhalb der Eigenschaft Smart Mobility ein Use Case für das Parken auf kostenpflichtigen, öffentlichen Parkplätzen erstellt angelehnt an das Projekt „Smart Parking Barcelona“. Über die Dapp sollen Parkplätze ortsunabhängig reserviert, bargeldlos bezahlt und verlängert werden können. Die Dapp sollte die Parkplätze und ihre Verfügbarkeiten in einer Google Map visualisieren. Hierfür wurde mit Docker Containern eine Ethereum Umgebung für den Smart Contract und eine JavaScript Umgebung für die GUI der Dapp erstellt und virtualisiert. Diese Umgebungen sind durch Docker (Open Source) als Images in öffentlichen Repositories reproduzierbar, versioniert und beschrieben. Die zugehörigen, öffentlichen GitHub Repositories enthalten die Quelldateien und sind mit dem DockerHub verbunden, der bei Push automatisch neue Images erstellt.

Anhand des Use Case wurde ein Smart Contract mit Solidity erstellt, im Testnetz „morden“ deployt und verifiziert. Für den Contract sind NatSpec Dokumentation für Benutzer und Entwickler sowie eine ABI als Schnittstellenspezifikation erstellt worden. Die Dapp selbst besitzt ein öffentliches GitHub Repository mit Contract und GUI. Dieses beinhaltet die Projektdaten von Meteor und der JavaScript IDE WebStorm von JetBrains. Die Auswahl des JavaScript Framework Meteor und dem „meteor-build-client“ zur Erstellung einer SPA basieren auf einer Empfehlung von Ethereum und vorhandener Pakete für Google Maps und Ethereum. Die Dapp besteht aus JavaScript, HTML und CSS bzw. LESS sowie dem Contract und adaptierter [P] Icons für Google Marker. Außerdem wurde über den „meteor-build-client“ eine als SPA gebundene Dapp erstellt, die ohne Meteor direkt im Browser lokal ausgeführt werden kann. Entwicklung und Test erfolgten iterativ und basieren auf elementaren Bestandteilen des Contract wie einzelnen State Variablen, Modifier, Events und Funktionen. Im Rahmen dieser Arbeit wurden hierfür 8 Testcases mit 77 Testfällen erstellt, wobei diese mit erhöhten Zeitinvest manuell ausgeführt wurden. Dabei wurden identifizierte Probleme behoben, Optimierungen umgesetzt und neue Test erstellt bzw. Regressionstests durchgeführt. Insgesamt wurden 93 Contracts deployt und getestet. Nach der letzten Iteration wurde der Contract in eine JavaScript Meteor App eingebunden. Die Erstellung und Tests der App erfolgte ebenfalls iterativ und erzeugte sechs Prototypen, die jeweils aufeinander aufbauen. POC1 besitzt das Templating und die View, POC2 bindet den Ethereum Client an, POC3 integriert die Google Map, POC4 integriert den Contract über Nachrichten und POC5 erweitert die Integration des Contract über Transaktionen. Die Prototypen wurden über modale Dialoge, Browserkonsole, Eventlog und über JavaScript Debugging verifiziert. Bei der Evaluation wurden identifizierte Problem behoben, Optimierungen umgesetzt und erneut getestet. Im Nachhinein betrachtet ist hier der Einsatz eines automatisierten Testframeworks für Funktions- und Regressionstests sinnvoll. Auch die Verifikation des Contracts über VM-Trace, Contract Speicher und Stapel im Blockchain Explorer sollte über eine lokale IDE für die EVM und Solidity mit der Entwicklung integriert und automatisiert werden. In Kapitel 6.3 wurde eine Anleitung zur Ausführung und Einrichtung von Ethereum und der erstellten Dapp als SPA und über die erstellten Docker Container beschrieben. Dabei wurde gezeigt wie ein Ethereum Client sich erstmalig

vollständig mit der Blockchain synchronisiert, ein Ethereum Account mit Ether erstellt wird und innerhalb der Dapp hierüber ein Parkplatz reserviert und bezahlt werden kann.

Im letzten Teil dieser Arbeit wurden Auswirkungen der Dapp, der Blockchain und Ethereum Plattform auf Smart Cities basierend auf Kapitel 2.5 diskutiert und fünf mögliche Szenarien für die Anwendung der Dapp aufgezeigt. Hierbei wurden die Dimensionen, Eigenschaften, Erfolgsfaktoren, Vor- und Nachteile für eine Anwendung im Kontext einer Smart City betrachtet. Dapp User gelten hierbei als „Citizen as a Sensor“ und sind in der Eigenschaft Smart People neuen Technologien aufgeschlossen. Analog zu anderen großen Entwicklungen wie den sozialen Netzwerken erfordern neue Technologien Zeit für deren Verständnis und Gewöhnung. Eine wichtige Voraussetzung für erfolgreiche Smart City Projekte sind neben den Erfolgsfaktoren und „Best Practises“ ein hoher Grad an Skalierbarkeit sowie die Entwicklung mobiler Dapps und leichtgewichtiger Clients. Im nächsten und letzten Abschnitt wird ein Ausblick über weitere Entwicklungen im Bereich Ethereum Blockchain, Smart Cities, Smart Mobility und Smart Parking gegeben.

8.2 Ausblick und Fazit

Die weiteren Entwicklungen der Blockchain und vor allem der Ethereum Plattform bleiben spannend. So wurde während dieser Arbeit eine neue Version von Ethereum live gestellt, neue Client Versionen bereitgestellt und Ethereum mit ETH zur zweitstärksten kryptografischen Währung hinter Bitcoin. Darüber hinaus wurde mit „The DAO“ das größte Crowdfunding abgewickelt durch *slock.it*, die bereits große Unternehmen wie Microsoft und Samsung als Partner gewinnen konnten. Mit den P2P-Protokollen Whisper und Swarm, dem Dapp-Browser und mobilen Clients sind weitere Features in der neuen Version Metropolis geplant. Das JavaScript Framework Meteor wurde während dieser Arbeit ebenfalls aktualisiert und bietet die Entwicklung mobiler Dapps auf unterschiedlichen Plattformen. Die Evaluation neuer Versionen und Features wie auch Tools und Frameworks für die Entwicklung und automatisiertes Testen von Smart Contracts und Dapps wäre eine Weiterführung dieser Arbeit ebenso wie die Optimierung des Contracts hinsichtlich GAS Kosten und indexierten Eventlogs für leichtgewichtige Clients. Zudem wäre eine Simulation zu erstellen, die realistische Parksituationen mit mehrere Parkplätzen und Reservierungen über einen längeren Zeitraum abbildet.

Des Weiteren wurden die Themengebiete Blockchain Performance, Sicherheit und Mining im Rahmen dieser Arbeit nicht im Detail behandelt und erfordern eine gesonderte Betrachtung vor allem in den Bereichen Big Data und Mobile Computing. Letzteres umschließt die Aspekte Usability (DIN EN ISO 9241-11), User Experience (DIN EN ISO 9241-210) und den mobilen Kontext insbesondere für die Akzeptanz und Toleranz des Blockchain Konzept und Computerparadigma.

Die Dapp „Parking Places“ sollte unter dem Namen „Smart Parking Dapp“ weiter entwickelt werden. Hierfür ist wie beschrieben die Entwicklungsumgebung zu aktualisieren, ein

automatisiertes Testframework einzurichten, die Dapp auf leichtgewichtige Clients und mobile Geräte zu portieren. Die jeweiligen Anwendungsszenarien aus Kapitel 7.1 erfordern eine Adaption der Dapp bzw. die Erstellung unterschiedlicher Dapps. Zudem kann die Dapp auch in der Cloud wie z.B. Microsoft Azure als Service betrieben werden, so dass der Dapp User lediglich ein mobiles Gerät mit Internetverbindung und einen Ethereum Account benötigt. Auch ist das Nutzungsmodell von Google Maps zu prüfen und ggf. durch OpenStreetMap auszutauschen. Die Dapp sollte anlehnend an Smart Mobility Lösungen um Funktionen wie Navigation und Feedback erweitert werden.

Als Smart Mobility Lösung kann die Dapp in eine Smart City Initiative als Projekt aufgenommen werden in Kooperation mit Stakeholdern im öffentlichen Bereich wie der lokalen Regierung und der Bevölkerung sowie dem privaten Sektor wie IKT-Unternehmen, Parkhausbetreiber und der Parkindustrie. Hier sollte der Use Case basierend auf aktuellen Parksituationen abgestimmt werden und eine Datenerhebung wichtiger Indikatoren stattfinden. Diese sind z.B. aktuellen Kosten durch Automaten, Wartung und Personal sowie Einnahmen über Parktickets. Außerdem dienen Metriken für Verkehr, CO_2-Emissionen und Lärmpegel dem Monitoring, Trendanalysen und der Erfolgsbewertung. Die Dapp sollte um administrative Möglichkeiten wie Öffnungszeiten, maximale Parkdauer, Parkkosten und Kontingente erweitert werden um flexibel auf besondere Ereignisse wie Events oder Ferien reagieren zu können. Weiter könnten Geschäfte Parkkosten oder die Reservierung eines Parkplatzes z.B. bei einem Restaurantbesuch übernehmen. Hauseigentümer mit eigenen Stellplätzen könnten diese über die Dapp als Parkplatz einstellen und vermieten analog zu Wohnungen über AirBnb nach dem Prinzip von $slock.it$. Jedes Fahrzeug könnte über sein Nummernschild als ID ein eigenen Ethereum Account besitzen als Smart Property innerhalb der Blockchain. Umweltschonendere Fahrzeuge könnten hierbei günstiger Parken und Restparkzeiten angerechnet werden. Ein möglicher Ansatz mit Ethereum ist das Erstellen einer eigenen Währung wie z.B. einem ParkingCoin oder einem CityCoin und hierauf aufbauend ein Crowdfunding zur Finanzierung des Projekts. Mit diesen Coins könnten z.B. Parktickets bezahlt werden. Über genutzte Parkzeiten könnten neue Coins gemined werden.

Die im Rahmen dieser Arbeit erstellte Dapp zeigt das Konzept der Ethereum Blockchain für einen Use Case im Bereich Smart Parking, Smart Mobility und Smart Cities. Es hat sich gezeigt, dass Ethereum die Testphase Frontier erfolgreich abgeschlossen und mit Homestead vermehrt produktiv eingesetzt wird. Die nächste Version Metropolis soll dieses Jahr veröffentlich werden und bietet eine breite Benutzerschnittstelle mit Dapp-Store, Whisper und Swarm. Auch vollständig entwickelte, mobile Clients sollen in Kürze verfügbar sein. In der Cloud von Microsoft Azure können über BaaS private Ethereum Netzwerke aufgebaut werden und vollständige Ethereum Clients erstellt werden. So kann eine Smart City ein privates Ethereum Netzwerk aufbauen oder ein Client Node erstellen und mit dem globalen Netzwerk verbinden. Generell lässt sich feststellen, dass Ethereum die aktuellste und verbreitete Open Source Blockchain Plattform darstellt. Sie ist im Vergleich zu Bitcoin sind Smart Contracts über Solidity flexibler und simpler sowohl in der Erstellung als auch in der Ausführung.

9 Verzeichnisse

9.1 Abbildungen

Abb. 1: kontinentale Urbanisierung von 2015 bis 2050 9
Abb. 2: City Konzepte und Dimensionen [NP11] 12
Abb. 3: Eigenschaften und Faktoren einer Smart City [Gi07] 15
Abb. 4: Smart City Modell [Ma14] 16
Abb. 5: Nutzlevel von Smart City Lösungen [Ma14] 18
Abb. 6: Merkle Baum mit vier Transaktionen [An15] 26
Abb. 7: Blockhashes und verknüpfte Zeitstempel [Na08] 27
Abb. 8: Blöcke in der Bitcoin Blockchain [Bi16h] 27
Abb. 9: Bitcoin Blockchain Forks [Fr15] 28
Abb. 10: Kontroll- und Datennachrichten im Bitcoin P2P-Netzwerk [Bi16h] 30
Abb. 11: P2PKH Skript in Bitcoin [An15] 34
Abb. 12: Schwankungen im Bitcoin Handelspreis [Bl16b] 36
Abb. 13: State Transition in Bitcoin [Et16h] 51
Abb. 14: Ethereum Blockchain [Et16e] 53
Abb. 15: Merkle Patrica Baum in Ethereum [Et16f] 55
Abb. 16: Dapps in Ethereum [Bu14a] 59
Abb. 17: Interaktion unterschiedlicher Contracts [Et16e] 61
Abb. 18: Benchmark für Ethereum Clients [Et16n] 65
Abb. 19: Benchmark für Merkle Patrica Bäume [Et16n] 66
Abb. 20: Parkscheinautomat und Parkschein [Wi15] 68
Abb. 21: Dapp Use Case Diagramm "Parking Places" 69
Abb. 22: Docker-Container "geth-node" als Ethereum Client 71
Abb. 23: Docker-Container "meteor-nodejs" als JavaScript-Umgebung für Dapps 73
Abb. 24: Entwicklung und Aufbau der Dapp 81
Abb. 25: Html-View mit Bereichen nach "Dapp-Styles" 85
Abb. 26: Icons und InfoWindow für Google Marker 87
Abb. 27: Meteor Build Client bundled Dapp 88
Abb. 28: POC1 mit GUI und POC2 mit Ethereum Client 94
Abb. 29: POC3 mit Google Map 94
Abb. 30: POC4 mit Contract (Message Calls) 95
Abb. 31: POC5 mit Contract (Transactions) 96
Abb. 32: Reservierung in "Parking-Places" 100

9.2 Tabellen

Tab. 1: Übersicht von City Konzepten gruppiert nach Dimensionen 11
Tab. 2: Smart City Definitionen von 2000 bis 2014 14
Tab. 3: Dimensionen und ihre strategische Ausrichtung [NP11] 16
Tab. 4: Erfolgsfaktoren und "Best Practises" [Ma14] 19
Tab. 5: Struktur eines Bitcoin Blocks [An15] 28
Tab. 6: Message Header im Bitcoin P2P-Netzwerk [Bi16h] 30
Tab. 7: Struktur einer Bitcoin Transaktion [An15] 32
Tab. 8: Struktur eines Bitcoin Transaktionseingangs [Bi16h] 33
Tab. 9: Struktur eines Bitcoin Transaktionsausgangs [Bi16h] 34
Tab. 10: Vor- und Nachteile von Bitcoin als Währung [Fr15] 37
Tab. 11: Blockchain 3.0 für Regierung, Kultur und Kunst [Sw15] 43
Tab. 12: Blockchain 3.0 für Gesundheit, Bildung und Wissenschaft [Sw15] 45
Tab. 13: Vorteile von UTXO und Accounts als State [Et16h] 50
Tab. 14: State Transition in Ethereum 52
Tab. 15: Struktur des Ethereum Blockheader [Wo14] 53
Tab. 16: RLP Enkodierung in Ethereum [Et16j] 56
Tab. 17: Live Dapps in Ethereum [Et16l] 62
Tab. 18: Weitere Dapps in Ethereum [Da16a],[Da16c] 63
Tab. 19: Ethereum Clients - Stand 17.03.2016 [Et16t] 65
Tab. 20: State Variablen in Contract "ParkingPlaces" 75
Tab. 21: Modifier in Contract "ParkingPlaces" 76
Tab. 22: Events in Contract "ParkingPlaces" 76
Tab. 23: "Transaction"-Funktionen in Contract "ParkingPlaces" 77
Tab. 24: "Message Call"-Funktionen in Contract "ParkingPlaces" 78
Tab. 25: Zusätzliche Meteor Packages in "ParkingPlaces" 82
Tab. 26: Meteor Templates in "ParkingPlaces" 83
Tab. 27: Helper Funktionen für Template "Dapp" in "Parking-Places" 84
Tab. 28: Übersicht der Testcases für Evaluation des Contract 91
Tab. 29: Fehler und Möglichkeiten zur Optimierung des Contract 92
Tab. 30: Fehler und Möglichkeiten zur Optimierung der Dapp 97
Tab. 31: Szenarien für "Parking Places" in Smart Cities 105
Tab. 32: Testcases für Contract "Parking Places" 152

9.3 Listings

Listing 1: JavaScript zur Berechnung des Ethereum Homestead Fork 48
Listing 2: JavaScript "CreateParkingPlacesAccounts.js" 78
Listing 3: JavaScript "ShowBalances.js" 79
Listing 4: JavaScript "ParkingPlaces.js" 79
Listing 5: Auschnitt aus JavaScript "CreateParkingPlaces.js" 80
Listing 6: Auschnitt aus JavaScript "CreateParkingPlacesSlots.js" 80
Listing 7: Handling Contract Event "Transaction" 86
Listing 8: Hinzufügen eines Google Markers 86
Listing 9: Meteor Build Client Skripte in generiertem HTML 88
Listing 10: Ethereum Client über Container ausführen 98
Listing 11: Ethereum Accounts erstellen 99
Listing 12: Meteor DApp über Container ausführen 99
Listing 13: Ethereum Accounts entsperren 100
Listing 14: Dockerfile Container "geth-node" 129
Listing 15: Dockerfile Container "meteor-nodejs" 130
Listing 16: Contract Code "ParkingPlaces.sol" 134
Listing 17: NatSpec Benutzerdokumentation 135
Listing 18: NatSpec Entwicklerdokumentation 136
Listing 19: Contract ABI "ParkingPlaces" 138
Listing 20: "ParkingPlaces"- HTML mit Meteor Templates 140
Listing 21: "ParkingPlaces"- Design mit Less (CSS pre-Compiled) 141
Listing 22: "ParkingPlaces"- JavaScript 148

9.4 Formeln

Formel 1: Elliptische Kurve 23
Formel 2: Vergütung abgelaufener Blöcke in Ethereum [Wo14] 56
Formel 3: Code-Ausführung in Ethereum [Wo14] 58
Formel 4: Kostenfunktion für Operationen in Ethereum [Wo14] 58

9.5 Literatur

[An15] Antonopoulos, A. M.: Mastering bitcoin. [unlocking digital cryptocurrencies]. O'Reilly, Beijing, 2015.

[Ba12] Batty, M. et al.: Smart cities of the future. In The European Physical Journal Special Topics, 2012, 214; S. 481–518.

[Bi16a] BitPay, I.: Accept Bitcoin | Bitpay. https://bitpay.com/, 27.01.2016.

[Bi16b] Bitcoinmining: Everything you need to know about Bitcoin mining. https://www.bitcoinmining.com/, 06.02.2016.

[Bi16c] BitcoinRichList: BitcoinRichList. http://www.bitcoinrichlist.com/, 06.02.2016.

[Bi16d] Bitcoin: Bitcoin Open Source P2P Geld. https://bitcoin.org/de/, 27.01.2016.

[Bi16e] Bitcoin Wiki: Bitcoin Wiki. https://en.bitcoin.it, 01.02.2016.

[Bi16f] BitcoinExchangeRate: Bitcoin Exchange Rate. http://bitcoinexchangerate.org, 21.05.2016.

[Bi16g] BitcoinAverage: Bitcoin Average Price Index. https://bitcoinaverage.com, 27.01.2016.

[Bi16h] Bitcoin Project: Developer Documentation. Find useful resources, guides and reference material for developers. https://bitcoin.org/en/developer-documentation, 22.01.2016.

[Bl16a] BlockApps, I.: Scalable Enterprise Blockchains. http://www.blockapps.net, 15.03.2016.

[Bl16b] Blockchain: Bitcoin Block Explorer. https://blockchain.info, 20.01.2016.

[Bo13] Bonadonna, E.: Bitcoin and the Double-Spending Problem. https://blogs.cornell.edu/info4220/2013/03/29/bitcoin-and-the-double-spending-problem/, 13.01.2016.

[Bo15] Boettiger, C.: An introduction to Docker for reproducible research. In ACM SIGOPS Operating Systems Review, 2015, 49; S. 71–79.

[BP14] Brody, P.; Pureswaran, V.: Device democracy: Saving the future of the Internet of Things. In IBM Global Business Services Executive Report, 2014.

[Bu14a] Buterin, V.: Ethereum DEV PLAN. https://github.com/ethereum/www/blob/master-sale/src/extras/pdfs/Ethereum-Dev-Plan.pdf, 20.02.2016.

[Bu14b] Buterin, V.: Ethereum White Paper. A next-generation smart contract and decentralized application platform, 2014.

[Bu14c] Buterin, V.: Toward a 12-second Block Time. https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/, 26.02.2016.

[Bu15a] Buterin, V.: Merkling in Ethereum. https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/, 23.02.2016.

[Bu15b] Buterin, V.: EIP-2. Homestead Hard-fork Changes. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.mediawiki, 17.03.2016.

[Bu15c] Buterin, V.: EIP-7. DELEGATECALL. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-7.md, 17.03.2016.

[Bv15] Buterin, V.; van de Sande, A.: Frontier mining 10% reward??? https://www.reddit.com/r/ethereum/comments/2y8v14/frontier_mining_10_reward/, 20.02.2016.

[CDN09] Caragliu, A.; Del Bo, C.; Nijkamp, P.: Smart cities in Europe, 2009.

[Ce10] Certicom: SEC 2: Recommended Elliptic Curve Domain Parameters. Standards for Efficient Cryptography, 2010.

[Co16a] Coin ATM Radar: Bitcoin ATM Map. Find Bitcoin ATM, Online Rates. http://coinatmradar.com/, 21.05.2016.

[Co16b] ConsenSys: Ubuntu - ConsenSys. ConsenSys, BlockApps, and Canonical form team to deliver Nimbus uPortBiometric Digital Identity Ethereum tools on Ubuntu phones so users can access Ethereum safely. https://consensys.net/static/ubuntu.pdf, 15.03.2016.

[Co16c] Coinmap: Map of Bitcoin accepting venues. https://coinmap.org, 26.02.2016.

[Co16d] CoinMarketCap: Crypto-Currency Market Capitalizations. http://coinmarketcap.com/, 24.01.2016.

[Cr06] Crockford, D.: RFC4627: The application/json media type for JavaScript Object Notation. In Network Working Group, IETF, 2006.

[Da16a] DappsList: DappsList - DApps and Smart Contracts. https://dappslist.com/, 13.03.2016.

[Da16b] Dashjr, L.: GitHub bitcoin/bips. Bitcoin Improvement Proposals. https://github.com/bitcoin/bips, 18.01.2016.

[Da16c] DappCentral: DappCentral - Ethereum Dapps. http://dappcentral.io/, 13.03.2016.

[DBP12] Dobbertin, H.; Bosselaers, A.; Preneel, B.: The hash function RIPEMD-160. http://homes.esat.kuleuven.be/~bosselae/ripemd160.html, 07.01.2016.

[DBP96] Dobbertin, H.; Bosselaers, A.; Preneel, B.: RIPEMD-160: A strengthened version of RIPEMD: Fast Software Encryption, 1996; S. 71–82.

[De14] Department of Economic and Social Affairs, Population Division: World Urbanization Prospects: The 2014 Revision. Annual Percentage of Population at Mid-Year Residing in Urban Areas. http://esa.un.org/unpd/wup, 14.12.2015.

[Do16] Docker Inc.: Docker Docs. https://docs.docker.com/, 27.05.2016.

[Et14] Ethereum: Intented Use of Revenue. https://github.com/ethereum/www/blob/master-sale/src/extras/pdfs/IntendedUseOfRevenue.pdf, 20.02.2016.

[Et16a] Ethereum: Ethash. https://github.com/ethereum/wiki/wiki/Ethash, 26.02.2016.

[Et16b] Ethereum: Ethereum Frontier Guide. http://guide.ethereum.org, 20.02.2016.

[Et16c] Etherchain: The ethereum blockchain explorer. https://www.etherchain.org/, 06.03.2016.

[Et16d] EtherCamp: EtherCamp. https://live.ether.camp/, 06.03.2016.

[Et16e] Ethereum: Ethereum Development Tutorial. https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial, 21.02.2016.

[Et16f] Ethereum: Patricia Tree. https://github.com/ethereum/wiki/wiki/Patricia-Tree, 24.02.2016.

[Et16g] Ethereum: Solidity, Docs and ABI. https://github.com/ethereum/wiki/wiki/Solidity,-Docs-and-ABI, 06.03.2016.

[Et16h] Ethereum: Design Rationale. https://github.com/ethereum/wiki/wiki/Design-Rationale, 20.12.2016.

[Et16i] EtherScan: Ethereum BlockChain Explorer | Etherscan. http://etherscan.io, 20.12.2016.

[Et16j] Ethereum: RLP. https://github.com/ethereum/wiki/wiki/RLP, 25.02.2016.

[Et16k] Ethereum: Solidity by Example. Solidity 0.2.0 documentation. https://solidity.readthedocs.org, 06.03.2016.

[Et16l] EtherCasts: State of the Dapps. http://dapps.ethercasts.com/, 16.03.2016.

[Et16m] Ethereum: JavaScript API. Web3 JavaScript Đapp API. https://github.com/ethereum/wiki/wiki/JavaScript-API, 03.04.2016.

[Et16n] Ethereum: Benchmarks. https://github.com/ethereum/wiki/wiki/Benchmarks, 15.03.2016.

[Et16o] Ethereum: Dapp using Meteor. https://github.com/ethereum/wiki/wiki/Dapp-using-Meteor, 03.04.2016.

[Et16p] Ethernodes: The Ethereum Nodes Explorer. http://ethernodes.org, 17.03.2016.

[Et16q] Ethereum: JSON RPC. JSON RPC API. https://github.com/ethereum/wiki/wiki/JSON-RPC, 03.04.2016.

[Et16r] Ethereum: Create a crowdsale contract in Ethereum. Crowdfund your idea. https://www.ethereum.org/crowdsale, 18.05.2016.

[Et16s] Ethereum: Create a Democracy contract in Ethereum. Decentralized Autonomous Organization. https://www.ethereum.org/dao, 10.05.2016.

[Et16t] Ethereum community: Ethereum Homestead Documentation. https://ethereum-homestead.readthedocs.org, 13.03.2016.

[Eu15] Eurostat: Europe 2020 Indicators. http://ec.europa.eu/eurostat/web/europe-2020-indicators, 19.12.2015.

[Eu82] Europäisches Patentamt: Method of providing digital signatures. http://worldwide.espacenet.com/publicationDetails/biblio?CC=US&NR=4309569&KC=&FT=E&locale=en_EP, 19.01.2016.

[Fr15] Franco, P.: Understanding bitcoin. Cryptography, engineering and economics. Wiley, Chichester, 2015.

[Ga13] Gartner: Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020. http://www.gartner.com/newsroom/id/2636073?huid=mhs7JKpr9Id0f4H7xRpBQQ, 30.12.2015.

[Ga14] Gartner: Internet of Things units installed base within smart cities in 2015, by subgroup. http://www.statista.com/statistics/423094/internet-of-things-units-installed-base-in-smart-cities-segment-by-type/, 28.11.2015.

[GD16] Ghemawat, S.; Dean, J.: GitHub google/leveldb. https://github.com/google/leveldb, 18.01.2016.

[Ge15] Gerring, T.: When is Homestead? https://www.reddit.com/r/ethereum/comments/3xunoj/when_is_homestead/cy7xl5l, 20.02.2016.

[GG10] Giffinger, R.; Gudrun, H.: Smart cities ranking: an effective instrument for the positioning of the cities?, 2010.

[Gi07] Giffinger, R. et al.: Smart cities: ranking of European medium-sized cities, 2007.

[Gi15] Giffinger, R.: european smart cities. http://www.smart-cities.eu/, 21.12.2015.

[Go16] Google Developers: Google Maps API. https://developers.google.com/maps, 05.05.2016.

[Gr15] Gray, M.: Ethereum Blockchain as a Service now on Azure. https://azure.microsoft.com/de-de/blog/ethereum-blockchain-as-a-service-now-on-azure, 15.03.2016.

[Gu15] Gupta, V.: The Ethereum Launch Process. https://blog.ethereum.org/2015/03/03/ethereum-launch-process/, 20.02.2016.

[Ha00] Hall, R. E. et al.: The vision of a smart city: Proceedings of the 2nd International Life Extension Technology Workshop, 2000.

[Ha10] Harrison, C. et al.: Foundations for smarter cities. In IBM Journal of Research and Development, 2010, 54; S. 1–16.

[Ha12] Haque, U.: Surely there's a smarter approach to smart cities? http://www.wired.co.uk/news/archive/2012-04/17/potential-of-smarter-cities-beyond-ibm-and-cisco, 21.12.2015.

[Ho08] Hollands, R. G.: Will the real smart city please stand up? In City, 2008, 12; S. 303–320.

[Ho15] Howard, A.: See 54 Years Of Internet History In One Cool Timeline. http://www.huffingtonpost.com/entry/explore-the-history-of-the-internet-of-things-with-this-cool-timeline_55a3bff9e4b0a47ac15ccc69, 30.12.2015.

[IS14] ISO: ISO/IEC JTC1. Smart cities.

[Jo15] Johnston, D.; Yilmaz, S. O.; Kandah, J.; Bentenitis, N.; Hashemi, F.; Gross, R.; Wilkinson, S.; Mason, S.: The General Theory of Decentralized Applications, Dapps. https://github.com/DavidJohnstonCEO/DecentralizedApplications, 02.02.2016.

[Ko02] Komninos, N.: Intelligent cities: innovation, knowledge systems, and digital spaces. Taylor & Francis, 2002.

[La15] Lange, F.: EIP-8. devp2p Forward Compatibility Requirements for Homestead. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-8.md, 17.03.2016.

[Le14] Ledra Capital LLC.: Bitcoin Series 24: The Mega-Master Blockchain List. http://ledracapital.com/blog/2014/3/11/bitcoin-series-24-the-mega-master-blockchain-list.

[LSP82] Lamport, L.; Shostak, R.; Pease, M.: The Byzantine generals problem. In ACM Transactions on Programming Languages and Systems (TOPLAS), 1982, 4; S. 382–401.

[Ma14] Manville, C. et al.: Mapping smart cities in the EU, 2014.

[Mc15] McKinsey Global Institute: The Internet of Things: Mapping the Value Beyond the Hype. http://www.statista.com/study/29992/iot-global-in-depth-analysys/, 28.11.2015.

[Me16] Meteor Development Group Inc.: Build JavaScript apps and websites using Meteor. https://www.meteor.com/developers, 29.05.2016.

[Me79] Merkle, R.: Secrecy, Authentication, and Public Key Systems. http://www.merkle.com/papers/Thesis1979.pdf.

[Na08] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system, 2008.

[Na09] Nakamoto, S.: Bitcoin v0.1 released. Cryptography Mailing List Emails. http://satoshi.nakamotoinstitute.org/emails/cryptography/16/, 10.01.2016.

[Na10] Nakamoto, S.: Re: Transactions and Scripts: DUP HASH160 … EQUALVERIFY CHECKSIG. Bitcointalk. http://satoshi.nakamotoinstitute.org/posts/bitcointalk/126/#selection-25.0-25.155, 31.01.2016.

[NI01] NIST: Advanced Encryption Standard (AES). FIPS PUB 197, 2001.

[NI02] NIST: Secure Hash Standard (SHS). FIPS PUB 180-4, 2002.

[NI15] NIST: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 202, 2015.

[NK13] Nijkamp, P.; Kourtit, K.: The “New Urban Europe”. Global Challenges and Local Responses in the Urban Century. In European Planning Studies, 2013, 21; S. 291–315.

[NP11] Nam, T.; Pardo, T. A.: Conceptualizing smart city with dimensions of technology, people, and institutions. In (Bertot, J. et al. Hrsg.): the 12th Annual International Digital Government Research Conference, 2011; S. 282.

[Ox15] Oxford English Dictionary: "smart, a.". http://www.oxforddictionaries.com/de/definition/learner/smart, 19.12.2015.

[Pa04] Partridge, H. L.: Developing a human perspective to the digital divide in the’smart city’, 2004.

[Pa15] Panikkar, S.; Nair, S.; Brody, P.; Pureswaran, V.: ADEPT: An IoT Practitioner Perspective. https://pdf.yt/d/esMcC00dKmdo53-_/download, 15.03.2016.

[Pa16] PayPal: Paypal Gebühren: transparent und fair. https://www.paypal.com/de/webapps/mpp/paypal-fees, 31.01.2016.

[Pe16] Percolate Studio: Atmosphere. The trusted source for JavaScript packages, Meteor resources and tools. https://atmospherejs.com, 07.04.2016.

[Pi16] Piasecki, P.: Crypto 2.0 Comparison Chart. http://tiny.cc/Crypto, 01.02.2016.

[Po80] Postel, J.: RFC761: Transmission Control Protocol. In Network Working Group, IETF, 1980.

[PP10] Paar, C.; Pelzl, J.: Understanding cryptography. A textbook for students and practitioners. Springer, Berlin, 2010.

[Re16] Reitwiessner, C.: Solidity realtime compiler and runtime. http://chriseth.github.io/browser-solidity/, 06.03.2016.

[RI12] RIOS, P.: Creating" The Smart City". Dissertation, 2012.

[SBM12] Schuurman, D.; Baccarne, B.; Marez, L. de: Smart Ideas for Smart Cities. Investigating Crowdsourcing for Generating and Selecting Ideas for ICT Innovation in a City Context. In Journal of theoretical and applied electronic commerce research, 2012, 7; S. 11–12.

[Sc11] Schaffers, H. et al.: Smart Cities and the Future Internet: Towards Cooperation Frameworks for Open Innovation. In Future Internet Assembly, 2011, 6656; S. 431–446.

[Sm13a] Smart Cities Group: smart cities. http://smartcities.media.mit.edu/index.html, 21.12.2015.

[Sm13b] Smart Cities and Communities: Key Messages für the High-Level Group. https://eu-smartcities.eu/content/final-key-messages-hlg, 21.12.2015.

[St16] Stadt Oldenburg (Oldb): InternetStadtplan. http://gis4oldenburg.oldenburg.de, 30.04.2016.

[Su16] Summerwill, B.: Ethereum Light Client. http://doublethink.co/ethereum-light-client, 15.03.2016.

[Sw15] Swan, M.: Blockchain. Blueprint for a new economy. O'Reilly, Beijing, 2015.

[SZ15] Sompolinsky, Y.; Zohar, A.: Secure High-Rate Transaction Processing in Bitcoin. In (Böhme, R.; Okamoto, T. Hrsg.): Financial Cryptography and Data Security. 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015; S. 507–527.

[Sz97] Szabo, N.: Formalizing and securing relationships on public networks. http://szabo.best.vwh.net/formalize.html, 31.01.2016.

[Th15] The Library of Congress: Regulation of Bitcoin in Selected Jurisdictions. http://www.loc.gov/law/help/bitcoin-survey/, 26.01.2016.

[Th16] The World Bank Group: Remittance Prices Worldwide. Making Markets more transparent. https://remittanceprices.worldbank.org, 26.01.2016.

[To10] Toppeta, D.: The Smart City vision: How Innovation and ICTcan build smart, “liveable”, sustainable cities.

[Tu15a] Tual, S.: Final Steps. https://blog.ethereum.org/2015/07/27/final-steps/, 22.02.2016.

[Tu15b] Tual, S.: Frontier is coming – what to expect, and how to prepare. https://blog.ethereum.org/2015/07/22/frontier-is-coming-what-to-expect-and-how-to-prepare/, 12.03.2016.

[Tu15c] Turnbull, J.: The Docker Book. James Turnbull, 2015.

[Up16] Uphold, I.: Uphold - Homepage. https://uphold.com/, 26.01.2016.

[Vo16] Vogelsteller, F.: Meteor Build Client. https://github.com/frozeman/meteor-build-client, 06.05.2016.

[Wa09] Washburn, D. et al.: Helping CIOs understand “smart city” initiatives, 2009.

[WB13] Walravens, N.; Ballon, P.: Platform business models for smart cities: from control and value to governance and public value. In Communications Magazine, IEEE, 2013, 51; S. 72–79.

[We16] webbtc: Bitcoin Blockbrowser. Statistics. https://webbtc.com/stats, 23.01.2016.

[Wi15] Wikipedia: Parkraumbewirtschaftung. https://de.wikipedia.org/wiki/Parkraumbewirtschaftung, 30.04.2016.

[Wi16a] Wikipedia: List of highest funded crowdfunding projects. https://en.wikipedia.org/wiki/List_of_highest_funded_crowdfunding_projects, 01.02.2016.

[Wi16b] Wilcke, J.: Homestead Release. https://blog.ethereum.org/2016/02/29/homestead-release/, 12.03.2016.

[Wo14] WOOD, G.: ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. FINAL DRAFT - UNDER REVIEW, 2014.

A. Anhang

In diesem Abschnitt werden die Docker Container beschrieben durch ihre Docker Files aufgeführt mit den direkten Links zum DockerHub und GitHub Repository. Die Inhalte der GitHub Repositories jeweils mit einer Readme sind auf dem beigelegten Datenträger unter `docker` zu finden.

A.1 Docker Container „geth-node"

Dieser Container wird durch sein Dockerfile in Listing 14 beschrieben und im GitHub Repository unter *https://github.com/blakeberg/geth-node* vollständig beschrieben. Hier sind auch ein Beispiel Contract und sein JavaScript für ein Deployment enthalten. Das Repository ist mit einem automatisierten Build über DockerHub verbunden und somit ebenfalls unter *https://hub.docker.com/r/blakeberg/geth-node* beschrieben. Die zu dieser Arbeit passende Version im DockerHub ist als Tag v0.3 vorhanden.

```
1   #
2   # Official Ubuntu base image
3   #
4   FROM ubuntu:14.04.4
5   MAINTAINER blakeberg <bjoern.lakeberg@technik-emden.de>
6
7   ENV SSH_USERPASS=newpass
8
9   RUN apt-get update -y; apt-get dist-upgrade -y
10  RUN apt-get install -y openssh-server software-properties-common curl git
11  RUN add-apt-repository -y ppa:ethereum/ethereum; apt-get update -y
12  RUN apt-get install -y geth solc
13
14  RUN useradd -m geth -s /bin/bash
15  RUN echo geth:$SSH_USERPASS | chpasswd
16  ADD contracts/* /home/geth/
17
18  RUN mkdir /var/run/sshd
19  RUN sed 's@session\s*required\s*pam_loginuid.so@session optional pam_loginuid.so@g' -i
    /etc/pam.d/sshd
20  RUN echo "export VISIBLE=now" >> /etc/profile
21
22  RUN bash -c 'echo "geth ALL=(ALL:ALL) ALL" | (EDITOR="tee -a" visudo)'
23
24  EXPOSE 22 8545
25  ENTRYPOINT ["/usr/sbin/sshd", "-D"]
```

LISTING 14: DOCKERFILE CONTAINER "GETH-NODE"

A.2 Docker Container „meteor-nodejs"

Dieser Container wird durch sein Dockerfile in Listing 15 beschrieben und im GitHub Repository unter *https://github.com/blakeberg/meteor-nodejs* vollständig beschrieben. Hier sind auch ein Beispiel Meteor App mit HTML und JavaScript enthalten. Das Repository ist mit einem automatisierten Build über DockerHub verbunden und somit ebenfalls unter

https://hub.docker.com/r/blakeberg/meteor-nodejs beschrieben. Die zu dieser Arbeit passende Version im DockerHub ist als Tag v0.4 vorhanden.

```
#
# Thanks to Adam Miller <maxamillion@fedoraproject.org> from
#   https://github.com/fedora-cloud/Fedora-Dockerfiles
#
FROM centos:centos7
MAINTAINER blakeberg <bjoern.lakeberg@technik-emden.de>

ENV SSH_USERPASS=newpass

RUN curl --silent --location https://rpm.nodesource.com/setup_4.x | bash -

RUN yum -y update; yum clean all
RUN yum -y install openssh-server passwd sudo nodejs git; yum clean all

RUN useradd -m meteor
RUN echo -e "$SSH_USERPASS\n$SSH_USERPASS" | (passwd --stdin meteor)
RUN echo ssh meteor password: $SSH_USERPASS
RUN bash -c 'echo "meteor ALL=(ALL:ALL) ALL" | (EDITOR="tee -a" visudo)'
ADD dummy-dapp-example/* /home/meteor/

RUN mkdir /var/run/sshd
RUN ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key -N ''

RUN curl https://install.meteor.com/ | sh
RUN npm install -g meteor-build-client
RUN npm install solc
RUN npm install web3

EXPOSE 22 3000
ENTRYPOINT ["/usr/sbin/sshd", "-D"]
```

LISTING 15: DOCKERFILE CONTAINER "METEOR-NODEJS"

B. Anhang

In diesem Abschnitt wird der Solidity Contract „ParkingPlaces“ durch seinen Code und der NatSpec Benutzer- und Entwicklerdokumentation sowie ABI im JSON Format beschrieben. Alle Quellcodes sind ebenfalls im Repository *https://github.com/blakeberg/parking-dapp* unter `contracts` vorhanden. Die Inhalte des GitHub Repository sind auf dem beigelegten Datenträger unter `dapp` mit separater Readme zu finden.

B.1 Solidity Contract „ParkingPlaces“

Der Contract in Listing 16 beinhaltet State Variablen, eigene Strukturen als Gruppierung von Variablen, Modifier, Events, einem Konstruktor sowie interne und externe Funktionen.

```
1   /// @title Contract for Parking Places and Parking Reservations.
2   /// @author Bjoern Lakeberg
3   contract ParkingPlaces {
4
5       address public controller;
6       uint public blockCosts;
7       Place[] public places;
8       mapping(address => Place) placeOf;
9
10      struct Place {
11          address owner;
12          string name;
13          string latitude;
14          string longitude;
15          Slot[] slots;
16      }
17
18      struct Slot {
19          address parker;
20          uint reservedBlock;
21      }
22
23      modifier isController() {
24          if (controller != msg.sender)
25              throw;
26          else
27              _
28      }
29
30      modifier isOwner(address owner) {
31          if (owner != msg.sender)
32              throw;
33          else
34              _
35      }
36
37      modifier hasValue() {
38          if (msg.value > 0)
39              throw;
40          else
41              _
42      }
43
44      modifier allowReservation(address owner) {
45          if (owner == msg.sender || controller == msg.sender)
46              throw;
47          else
```

```
48                  _
49      }
50
51      event PlaceAdded(address place, string name, string latitude, string longitude);
52      event SlotsAdded(address place, uint amount);
53      event Reservation(address place, address parker, uint reservedBlock);
54      event Transaction(address from, address to, uint transfered, uint refund, uint block);
55
56      /// @dev msg.sender will set as controller
57      /// @notice creation of this contract
58      /// @param _blockCosts costs per block
59      function ParkingPlaces(uint _blockCosts) public {
60          controller = msg.sender;
61          blockCosts = _blockCosts;
62      }
63
64      /// @dev do nothing but throws if value is given
65      /// @notice fallback function rollback transaction if value present
66      function() hasValue public {}
67
68      /// @dev used public and intern in addPlace und addSlots
69      /// @notice verify if place with unique address exists
70      /// @param owner unique place address
71      /// @return if place exists
72      function existsPlace(address owner) constant public returns(bool exists) {
73          for (uint i = 0; i < places.length; i++) {
74              if (places[i].owner == owner)
75                  return true;
76          }
77          return false;
78      }
79
80      /// @dev only controller can add a place, throws if value is given
81      /// @notice add a new place fires an event
82      /// @param owner unique place address
83      /// @param name name of the place
84      /// @param lat latitude coordinate
85      /// @param long longitude coordinate
86      function addPlace(address owner, string name, string lat, string long)
87          isController
88          hasValue
89          public
90      {
91          if (!existsPlace(owner)) {
92              uint id = places.length++;
93              places[id].owner = owner;
94              places[id].name = name;
95              places[id].latitude = lat;
96              places[id].longitude = long;
97              places[id].slots.push(Slot(msg.sender, block.number));
98              PlaceAdded(owner, name, lat, long);
99          }
100     }
101
102     /// @dev only place owner can add slots, throws if value is given
103     /// @notice add new slots to an existing place fires an event
104     /// @param owner unique place address
105     /// @param amount amount of slots to add
106     function addSlots(address owner, uint amount) isOwner(owner) hasValue public {
107         if (existsPlace(owner)) {
108             for (uint i = 0; i < amount; i++)
109                 placeOf[owner].slots.push(Slot(msg.sender, block.number));
110                 SlotsAdded(owner, amount);
111             }
112     }
113
114     /// @dev if place not exists returns 0
115     /// @notice count all slots for a place inluding reserved ones
```

```
116     /// @param owner unique place address
117     /// @return count of all slots for a place
118     function getSlotCount(address owner) constant public returns(uint count) {
119         return placeOf[owner].slots.length;
120     }
121
122     /// @dev if place not exists or no slots free returns 0
123     /// @notice count free slots for a place from given block number
124     /// @param owner unique place address
125     /// @param atBlock block number for searching free slots
126     /// @return count of free slots
127     function getFreeSlotCount(address owner, uint atBlock)
128         constant
129         public
130         returns(uint count)
131     {
132         uint free = 0;
133         for (uint i = 0; i < placeOf[owner].slots.length; i++) {
134             if (placeOf[owner].slots[i].reservedBlock <= atBlock)
135                 free++;
136         }
137         return free;
138     }
139
140     /// @dev if place not exists returns 0
141     /// @notice get block number for next free slot
142     /// @param owner unique place address
143     /// @return block number for next free slot
144     function getNextFreeBlock(address owner) constant public returns(uint block) {
145         uint free = 0;
146         for (uint i = 0; i < placeOf[owner].slots.length; i++) {
147             if (free == 0)
148                 free = placeOf[owner].slots[i].reservedBlock;
149             else {
150                 if (placeOf[owner].slots[i].reservedBlock <= free)
151                     free = placeOf[owner].slots[i].reservedBlock;
152             }
153         }
154         return free;
155     }
156
157     /// @dev only if toBlock greater than atBlock else returns 0
158     /// @notice calculate a reservation from intervall at block numbers
159     /// @param atBlock block number as start for calculation
160     /// @param toBlock block number as end for calculation
161     /// @return amount in wei to pay for block intervall
162     function calculateEstimatedCosts(uint atBlock, uint toBlock)
163         constant
164         public
165         returns(uint costs)
166     {
167         if (toBlock > atBlock)
168             return blockCosts * (toBlock - atBlock);
169     }
170
171     /// @dev if place not exists or parker not found returns 0
172     /// @notice get the block number for a parker at place
173     /// @param owner unique place address
174     /// @param parker address to check for parking
175     /// @return reserved block number and index in slots for place
176     function getReservedBlock(address owner, address parker)
177         constant
178         public
179         returns(uint block, uint index)
180     {
181         for (uint i = 0; i < placeOf[owner].slots.length; i++) {
182             if (placeOf[owner].slots[i].parker == parker)
183                 return (placeOf[owner].slots[i].reservedBlock, i);
```

```
184         }
185     }
186 
187     /// @dev throws if place not exists, old time or sender = (place|controller)
188     /// @notice reserve a slot or extend existing reservation fires two events
189     /// @param owner unique place address
190     /// @param untilBlock until block number to reserve
191     function reserveSlot(address owner, uint untilBlock) allowReservation(owner) public {
192         if (untilBlock <= block.number || !existsPlace(owner))
193             throw;
194         var (reservedBlock, reservedId) = getReservedBlock(owner, msg.sender);
195         uint id = 0;
196         uint toReserve = untilBlock - block.number;
197         if (reservedBlock == 0) {
198             id = getNextFreeSlot(owner);
199         } else {
200             id = reservedId;
201             if (untilBlock >= reservedBlock && reservedBlock >= block.number) {
202                 toReserve = untilBlock - reservedBlock;
203             }
204         }
205         payReservation(owner, msg.sender, toReserve);
206         placeOf[owner].slots[id].parker = msg.sender;
207         placeOf[owner].slots[id].reservedBlock = untilBlock;
208         Reservation(owner, msg.sender, untilBlock);
209     }
210 
211     /// @dev called from reserveSlot and throws if given value is to low
212     /// @param owner unique place address
213     /// @param time number of blocks to pay
214     function payReservation(address owner, address parker, uint time) internal {
215         uint amount = time * blockCosts;
216         if (msg.value < amount)
217             throw;
218         owner.send(amount);
219         parker.send(msg.value - amount);
220         Transaction(msg.sender, owner, amount, msg.value - amount, time);
221     }
222 
223     /// @dev called from reserveSlot and throws if no free slot found
224     /// @param owner unique place address
225     /// @return id for next free slot in dynamic array
226     function getNextFreeSlot(address owner) internal returns(uint id) {
227         for (uint i = 0; i < placeOf[owner].slots.length; i++) {
228             if (placeOf[owner].slots[i].reservedBlock <= block.number)
229                 return i;
230         }
231         throw;
232     }
233 
234     /// @dev closing contract send value to its creator
235     /// @notice close contract only for controller
236     function close() isController {
237         selfdestruct(controller);
238     }
239 }
```

LISTING 16: CONTRACT CODE "PARKINGPLACES.SOL"

B.2 NatSpec Benutzerdokumentation

In Listing 17 sind die mit `@notice` im Contract Code annotierten Beschreibungen enthalten. Generiert wird diese mit dem Solidity Compiler aus der Konsole über `solc --userdoc ParkingPlaces.sol`.

```
{
   "methods" : {
      "addPlace(address,string,string,string)" : {
         "notice" : "add a new place fires an event"
      },
      "addSlots(address,uint256)" : {
         "notice" : "add new slots to an existing place fires an event"
      },
      "calculateEstimatedCosts(uint256,uint256)" : {
         "notice" : "calculate a reservation from intervall at block numbers "
      },
      "close()" : {
         "notice" : "close contract only for controller"
      },
      "existsPlace(address)" : {
         "notice" : "verify if place with unique address exists"
      },
      "getFreeSlotCount(address,uint256)" : {
         "notice" : "count free slots for a place from given block number"
      },
      "getNextFreeBlock(address)" : {
         "notice" : "get block number for next free slot"
      },
      "getReservedBlock(address,address)" : {
         "notice" : "get the block number for a parker at place"
      },
      "getSlotCount(address)" : {
         "notice" : "count all slots for a place inluding reserved ones "
      },
      "reserveSlot(address,uint256)" : {
         "notice" : "reserve a slot or extend existing reservation fires two events"
      }
   }
}
```

LISTING 17: NATSPEC BENUTZERDOKUMENTATION

B.3 NatSpec Entwicklerdokumentation

In Listing 18 sind die mit `@param`, `@return`, `@author`, `@title` und `@dev` im Contract Code annotierten Beschreibungen enthalten. Generiert wird diese mit dem Solidity Compiler aus der Konsole heraus über `solc --userdoc ParkingPlaces.sol`.

```
{
   "author" : "Bjoern Lakeberg",
   "methods" : {
      "addPlace(address,string,string,string)" : {
         "details" : "only controller can add a place, throws if value is given ",
         "params" : {
            "lat" : "latitude coordinate",
            "long" : "longitude coordinate",
            "name" : "name of the place",
            "owner" : "unique place address"
         }
      },
```

```
13          "addSlots(address,uint256)" : {
14             "details" : "only place owner can add slots, throws if value is given ",
15             "params" : {
16                "amount" : "amount of slots to add",
17                "owner" : "unique place address"
18             }
19          },
20          "calculateEstimatedCosts(uint256,uint256)" : {
21             "details" : "only if toBlock greater than atBlock else returns 0",
22             "params" : {
23                "atBlock" : "block number as start for calculation",
24                "toBlock" : "block number as end for calculation"
25             },
26             "return" : "amount in wei to pay for block intervall"
27          },
28          "close()" : {
29             "details" : "closing contract send value to its creator"
30          },
31          "existsPlace(address)" : {
32             "details" : "used public and intern in addPlace und addSlots ",
33             "params" : {
34                "owner" : "unique place address"
35             },
36             "return" : "if place exists"
37          },
38          "getFreeSlotCount(address,uint256)" : {
39             "details" : "if place not exists or no slots free returns 0",
40             "params" : {
41                "atBlock" : "block number for searching free slots",
42                "owner" : "unique place address"
43             },
44             "return" : "count of free slots"
45          },
46          "getNextFreeBlock(address)" : {
47             "details" : "if place not exists returns 0",
48             "params" : {
49                "owner" : "unique place address"
50             },
51             "return" : "block number for next free slot"
52          },
53          "getReservedBlock(address,address)" : {
54             "details" : "if place not exists or parker not found returns 0",
55             "params" : {
56                "owner" : "unique place address",
57                "parker" : "address to check for parking"
58             },
59             "return" : "reserved block number and index in slots for place"
60          },
61          "getSlotCount(address)" : {
62             "details" : "if place not exists returns 0",
63             "params" : {
64                "owner" : "unique place address"
65             },
66             "return" : "count of all slots for a place"
67          },
68          "reserveSlot(address,uint256)" : {
69             "details" : "throws if place not exists, old time or sender=(place|controller)",
70             "params" : {
71                "owner" : "unique place address",
72                "untilBlock" : "until block number to reserve"
73             }
74          }
75       },
76       "title" : "Contract for Parking Places and Parking Reservations."
77    }
```

LISTING 18: NATSPEC ENTWICKLERDOKUMENTATION

B.4 Contract ABI von „ParkingPlaces”

Das Abstract Binary Interface eines Contract ist streng typisiert, statisch und vor dem Kompilieren bekannt. Über das ABI im JSON Format wird ein Contract von einer bestimmten Adresse geladen und kann über die im ABI definierten Funktionen und Events genutzt werden. Listing 19 zeigt das ABI des Contract „ParkingPlaces“.

```
[{
    "constant": true,
    "inputs": [{"name": "owner", "type": "address"}],
    "name": "getSlotCount",
    "outputs": [{"name": "count", "type": "uint256"}],
    "type": "function"
}, {
    "constant": false,
    "inputs": [{"name": "owner", "type": "address"},
               {"name": "amount", "type": "uint256"}],
    "name": "addSlots",
    "outputs": [],
    "type": "function"
}, {
    "constant": true,
    "inputs": [{"name": "owner", "type": "address"}],
    "name": "existsPlace",
    "outputs": [{"name": "exists", "type": "bool"}],
    "type": "function"
}, {
    "constant": true,
    "inputs": [{"name": "atBlock", "type": "uint256"},
               {"name": "toBlock", "type": "uint256"}],
    "name": "calculateEstimatedCosts",
    "outputs": [{"name": "costs", "type": "uint256"}],
    "type": "function"
}, {
    "constant": false,
    "inputs": [],
    "name": "close",
    "outputs": [],
    "type": "function"
}, {
    "constant": true,
    "inputs": [{"name": "owner", "type": "address"},
               {"name": "atBlock", "type": "uint256"}],
    "name": "getFreeSlotCount",
    "outputs": [{"name": "count", "type": "uint256"}],
    "type": "function"
}, {
    "constant": true,
    "inputs": [{"name": "owner", "type": "address"}],
    "name": "getNextFreeBlock",
    "outputs": [{"name": "block", "type": "uint256"}],
    "type": "function"
}, {
    "constant": true,
    "inputs": [],
    "name": "blockCosts",
    "outputs": [{"name": "", "type": "uint256"}],
    "type": "function"
}, {
    "constant": false,
    "inputs": [{"name": "owner", "type": "address"}, {"name": "name", "type": "string"},
               {"name": "lat", "type": "string"}, {"name": "long", "type": "string"}],
    "name": "addPlace",
    "outputs": [],
```

```
54        "type": "function"
55    }, {
56        "constant": true,
57        "inputs": [{"name": "owner", "type": "address"},
                     {"name": "parker", "type": "address"}],
58        "name": "getReservedBlock",
59        "outputs": [{"name": "block", "type": "uint256"},
                      {"name": "index", "type": "uint256"}],
60        "type": "function"
61    }, {
62        "constant": true,
63        "inputs": [{"name": "", "type": "uint256"}],
64        "name": "places",
65        "outputs": [{"name": "owner", "type": "address"}, {"name": "name", "type": "string"},
                      {"name": "latitude", "type": "string"},
                      {"name": "longitude", "type": "string"}],
66        "type": "function"
67    }, {
68        "constant": false,
69        "inputs": [{"name": "owner", "type": "address"},
                     {"name": "untilBlock", "type": "uint256"}],
70        "name": "reserveSlot",
71        "outputs": [],
72        "type": "function"
73    }, {
74        "constant": true,
75        "inputs": [],
76        "name": "controller",
77        "outputs": [{"name": "", "type": "address"}],
78        "type": "function"
79    }, {
80        "inputs": [{"name": "_blockCosts", "type": "uint256"}],
81        "type": "constructor"
82    }, {
83        "anonymous": false,
84        "inputs": [{"indexed": false, "name": "place", "type": "address"},
                     {"indexed": false, "name": "name", "type": "string"},
                     {"indexed": false, "name": "latitude", "type": "string"},
                     {"indexed": false, "name": "longitude", "type": "string"}],
85        "name": "PlaceAdded",
86        "type": "event"
87    }, {
88        "anonymous": false,
89        "inputs": [{"indexed": false, "name": "place", "type": "address"},
                    {"indexed": false, "name": "amount", "type": "uint256"}],
90        "name": "SlotsAdded",
91        "type": "event"
92    }, {
93        "anonymous": false,
94        "inputs": [{"indexed": false, "name": "place", "type": "address"},
                     {"indexed": false, "name": "parker", "type": "address"},
                     {"indexed": false, "name": "reservedBlock", "type": "uint256"}],
95        "name": "Reservation",
96        "type": "event"
97    }, {
98        "anonymous": false,
99        "inputs": [{"indexed": false, "name": "from", "type": "address"},
                     {"indexed": false, "name": "to", "type": "address"},
                     {"indexed": false, "name": "transfered", "type": "uint256"},
                     {"indexed": false, "name": "refund", "type": "uint256"},
                     {"indexed": false, "name": "block", "type": "uint256"}],
100       "name": "Transaction",
101       "type": "event"
102   }
103 ]
```

LISTING 19: CONTRACT ABI "PARKINGPLACES"

C. Anhang

In diesem Abschnitt sind mit den Quellcodes die Implementierung der Dapp aus JavaScript, HTML und CSS bzw. LESS offen gelegt. Alle Quellcodes sind ebenfalls im Repository *https://github.com/blakeberg/parking-dapp* unter `ui/client` vorhanden. Eine Sammlung von Testcases für Contract und Dapp dient der Evaluation. Die Inhalte des GitHub Repository sind auf dem beigelegten Datenträger unter `dapp` mit separater Readme zu finden.

C.1 „ParkingPlaces" View aus HTML und CSS

Listing 20 zeigt die HTML-Struktur der Dapp und Integration der Meteor Templates.

```
1   <head>
2     <title>Decentralized Application for Parking in Oldenburg (GER)</title>
3   </head>
4
5   <body>
6     {{> dapp}}
7     {{> dapp_modalPlaceholder}}
8   </body>
9
10  <template name="dapp">
11
12    <header class="dapp-header">
13      <h1>Parking Places in Oldenburg</h1>
14      <h3>last block {{currentBlockNumber}} at {{currentBlockTime}}</h3>
15    </header>
16
17    <div class="dapp-flex-content">
18
19      <aside class="dapp-aside">
20        <h2>an Ethereum decentralized application</h2>
21        Controller address:
22        {{> dapp_addressInput placeholder=contractController disabled="true"}
23        <br>
24        Select your account to pay for a parking place reservation:
25        <div class="from">
26          {{> dapp_selectAccount accounts=accounts showAccountTypes=true}}
27        </div>
28        <br>
29        <small>
30          {{dapp_formatBalance "1000000000000000000" "0,0.00[000] UNIT"}} =
            {{dapp_formatBalance "1000000000000000000" "0,0.00[0000] UNIT" "btc"}} =
            {{dapp_formatBalance "1000000000000000000" "0,0.00 UNIT" "eur"}}
31        </small>
32        <br>
33        You have to pay {{dapp_formatBalance contractParkingCosts "0,0.00[000] UNIT"}} per
           parking block.
34        <br><br>
35        Insert place address:
36        <div class="to">
37          {{> dapp_addressInput placeholder="copy paste address from place info of map
             markers" }}
38        </div>
39        <div align="right">Reserve until block....
40          <input type="number" name="block" pattern="[0-9]*" class="block"
             placeholder="type block number in future" min={{currentBlockNumber}}>
41        </div>
42        <br>
43        <small>Your estimated costs is
```

```
44          {{dapp_formatBalance estimatedParkingCosts "0,0.00[000] UNIT"}} =
            {{dapp_formatBalance estimatedParkingCosts "0,0.00[0000] UNIT" "btc"}} =
            {{dapp_formatBalance estimatedParkingCosts "0,0.00 UNIT" "eur"}}
45        </small>
46        <button type="submit" class="dapp-block-button">Park now</button>
47        <div align="center" class="check">
48          <button type="submit" class="dapp-large">Check parking</button>
49        </div>
50        <h4>Your contract events</h4>
51        <small>
52          <div class="payment">
53            {{#each contractLogs}}
54              <li>{{this}}</li>
55            {{/each}}
56          </div>
57        </small>
58      </aside>
59
60      <main class="dapp-content">
61        {{> googleMap name="map" options=mapOptions}}
62      </main>
63
64    </div>
65  </template>
66
67  <template name="modal_info">
68    <h1>{{header}}</h1>
69    <p>{{message}}</p>
70  </template>
```

LISTING 20: "PARKINGPLACES"- HTML MIT METEOR TEMPLATES

Die in Listing 20 aufgeführte HTML View verwendet das in Listing 21 dargestellte CSS.

```
1   @import '{ethereum:dapp-styles}/dapp-styles.less'; //einziger LESS Ausdruck
2   //ab hier CSS
3   html, body {
4     height: 100%;
5   }
6
7   h1 {
8     margin-bottom: 24px !important;
9   }
10
11  input {
12    margin-top : 0 !important;
13  }
14
15  .dapp-flex-content {
16    height: 100%;
17    width: 100%;
18    background-color: #ccc6c6 !important;
19  }
20
21  .dapp-content {
22    height: 90%;
23    width: 100%;
24    margin: 0 !important;
25    padding: 0 !important;
26    max-width: 100% !important;
27  }
28
29  .dapp-block-button {
30    width: 100%
31  }
32
33  .payment {
34    width: 100%;
35    background: gainsboro;
```

```
36    border: solid gainsboro 1px;
37  }
```

LISTING 21: "PARKINGPLACES"- DESIGN MIT LESS (CSS PRE-COMPILED)

C.2 „ParkingPlaces" JavaScript

Innerhalb des JavaScripts `parking-dapp.js` in Listing 22 findet die Kommunikation zu Ethereum, dem Contract und Google Maps statt. Außerdem sind hier der Datenkontext, Events und Funktionen aller Templates enthalten. Anstelle des Contract ABI wurde der Platzhalter `ABI` eingesetzt.

```
4   //single page application (SPA) needs access to google maps and running ethereum client
5   if (Meteor.isClient) {
6     //map constants
7     const MAP_ZOOM = 15;
8     const CENTER = {lat: 53.143722, lng: 8.214059};
9     const TIMEOUT_ANIMATION = 200;
10    //update all places and markers every x blocks
11    const REFRESH_INTERVALL = 20;
12    //threshold for slot capacity of places determine icons (all % in free slots)
13    const RED_THRESHOLD = 20; //under 20% free
14    const YELLOW_THRESHOLD = 50; //under 50% free
15    //rpc address of ethereum client
16    const ETH_RPC_ADDRESS = 'http://localhost:8545';
17    //contract address
18    const CONTRACT_ADDRESS = "0xad3d7d21862dfa1f9d91569240a9ed06ac276b4d";
19
20    //initialize web3 and uri of json rpc api from running ethereum client
21    if (typeof web3 === 'undefined') {
22      web3 = new Web3(new Web3.providers.HttpProvider(ETH_RPC_ADDRESS));
23    }
24    //contract definition and contract object
25    var parkingplaces = loadContract();
26
27    //associative key-value arrays (first three with same index)
28    var markers = [];
29    var placeInfos = [];
30    var places = [];
31    var eventlogs = [];
32    //selected block to estimate or parking
33    var block;
34
35    //call when meteor client starting
36    Meteor.startup(function () {
37      //EthBlocks with last 50 block information auto updating
38      EthBlocks.init();
39      EthAccounts.init();
40      //load google package at start
41      GoogleMaps.load();
42    });
43
44    //template for block and time information
45    Template.dapp.helpers({
46      currentBlockNumber: function () {
47        updateAllMarker(EthBlocks.latest.number);
48        return EthBlocks.latest.number;
49      },
50      currentBlockTime: function () {
51        return formatTS(EthBlocks.latest.timestamp);
52      },
53      accounts: function () {
54        return EthAccounts.find().fetch();
55      },
```

```
56      contractController: function () {
57        return parkingplaces.controller();
58      },
59      contractParkingCosts: function () {
60        return parkingplaces.blockCosts();
61      },
62      contractLogs: function () {
63        // to call an update for each new block
64        EthBlocks.latest.number;
65        return eventlogs;
66      },
67      estimatedParkingCosts: function () {
68        return parkingplaces.calculateEstimatedCosts(EthBlocks.latest.number, block);
69      },
70      mapOptions: function () {
71        if (GoogleMaps.loaded()) {
72          return {
73            center: CENTER,
74            zoom: MAP_ZOOM
75          };
76        }
77      }
78    });
79
80    //handle events from dapp template
81    Template.dapp.events({
82      'click .dapp-block-button'(event) {
83        // Prevent default browser form submit
84        event.preventDefault();
85        var to = TemplateVar.getFrom('.to .dapp-address-input', 'value');
86        var estimatedCosts = parkingplaces.calculateEstimatedCosts(EthBlocks.latest.number,
              block);
87        if (isDataValid(to, block, true) && validateBalance(estimatedCosts)) {
88          var msg = "Do you want to reserve place " + to + " until block " + block +
89              " and pay " + web3.fromWei(estimatedCosts, "ether") + " ether?";
90          EthElements.Modal.question({
91            text: msg,
92            ok: function () {
93              reservation(to, block, estimatedCosts);
94              eventlogs.push("reservation for place " + to.substring(0, 8) + "... with " +
95                   web3.fromWei(estimatedCosts, "ether") + " ether until block " + block + "
                     was sent");
96            },
97            cancel: true
98          });
99        }
100     },
101     'click .dapp-large'(event) {
102       // Prevent default browser form submit
103       event.preventDefault();
104       var to = TemplateVar.getFrom('.to .dapp-address-input', 'value');
105       // pass block number cause we need only to validate the address
106       if (isDataValid(to, EthBlocks.latest.number + REFRESH_INTERVALL, false)) {
107         validateParking(to);
108       }
109     },
110     'change .block'(event) {
111       // Prevent default browser form submit
112       event.preventDefault();
113       block = event.target.value;
114     }
115   });
116
117   //actions on template creation - register contract events, load map and add markers for
       each place in contract
118   Template.dapp.onCreated(function () {
119     //adding events from contract
120     parkingplaces.PlaceAdded({}, '',
```

```
121       /**
122         * If contract event PlaceAdded to add a marker on map of google maps api
123         * @param error first callback style
124         * @param result with arguments (address place, string name, string latitude,
              string longitude)
125         */
126       function (error, result) {
127         if (!error) {
128           places[result.args.place] = [result.args.place, result.args.name,
                  result.args.latitude, result.args.longitude];
129           addMarkerWithTimeout(result.args.place, TIMEOUT_ANIMATION);
130         }
131         else {
132           console.error(error);
133         }
134       }
135     );
136     parkingplaces.SlotsAdded({}, '',
137       /**
138         * If contract event SlotsAdded update and animate marker on map of google maps api
139         * @param error first callback style
140         * @param result with arguments (address place, uint amount)
141         */
142       function (error, result) {
143         if (!error) {
144           updateMarker(result.args.place, true, true);
145         }
146         else {
147           console.error(error);
148         }
149       }
150     );
151     parkingplaces.Reservation({}, '',
152       /**
153         * If contract event Reservation show modal dialog if reservation of your account
154         * @param error first callback style
155         * @param result with arguments (address place, address parker, uint reservedBlock)
156         */
157       function (error, result) {
158         if (!error) {
159           updateMarker(result.args.place, true, true);
160           if (isOwnAccount(result.args.parker)) {
161             showMessage("Your reservation was successful", "for place at " +
162                 result.args.place + " from parker " + result.args.parker +
                    " until block number " + result.args.reservedBlock);
163             eventlogs.push("reservation for place " + result.args.place.substring(0, 8) +
164                  "... until block number " + result.args.reservedBlock + " was
                    successful");
165           }
166         }
167         else {
168           console.error(error);
169         }
170       }
171     );
172     parkingplaces.Transaction({}, '',
173       /**
174         * If contract event Transaction push sender and receiver details to eventLogs
175         * array if transaction was started from your account
176         * @param error first callback style
177         * @param result with arguments (address fromOrigin, address to, uint amount)
178         */
179       function (error, result) {
180         if (!error) {
181           if (isOwnAccount(result.args.from)) {
182             eventlogs.push("transfered to place " + result.args.to.substring(0, 8) +
183                 "... " + web3.fromWei(result.args.transfered, "ether") + " ether for " +
```

```
184                 result.args.block + " blocks " + " with " +
                    web3.fromWei(result.args.refund, "ether") + " ether refund");
185           }
186         }
187         else {
188           console.error(error);
189         }
190       }
191     );
192     //adding marker of google maps api for each place
193     GoogleMaps.ready('map', function () {
194       //load places from contract until exception
195       var next = true;
196       var i = 0;
197       while (next === true) {
198         try {
199           places[parkingplaces.places(i)[0]] = parkingplaces.places(i);
200           addMarkerWithTimeout(parkingplaces.places(i)[0], i * TIMEOUT_ANIMATION);
201           i++;
202         }
203         catch (e) {
204           next = false;
205         }
206       }
207     });
208   });
209
210   /**
211    * Update all marker in array without animating and centering
212    * @param currentBlock actual block number cause updating only every x.th block
213    */
214   function updateAllMarker(currentBlock) {
215     //only every x block
216     if ((currentBlock % REFRESH_INTERVALL) === 0) {
217       for (var key in markers) {
218         updateMarker(key, false, false);
219       }
220     }
221   }
222
223   /**
224    * Check if balance of selected account ist greater than amount
225    * @param amount of wei to check for
226    * @returns {boolean} true if balance of selected account is sufficient
227    */
228   function validateBalance(amount) {
229     var from = TemplateVar.getFrom('.from .dapp-select-account', 'value');
230     for (i = 0; i < EthAccounts.find().fetch().length; i++) {
231       if (EthAccounts.find().fetch()[i].address === from) {
232         if (amount.lessThan(EthAccounts.find().fetch()[i].balance)) {
233           return true;
234         }
235         else {
236           showMessage("Data verification", "Please choose an account with a balance of " +
237                "minimal " + web3.fromWei(amount, "ether") + " ether");
238           return false;
239         }
240       }
241     }
242     return false;
243   }
244
245   /**
246     * Validate parking for selected address and a place to show the result in a message
247     * @param to the address of the parking place
248     */
249   function validateParking(to) {
250     var from = TemplateVar.getFrom('.from .dapp-select-account', 'value');
```

```
251     var reservedBlock = parkingplaces.getReservedBlock(to, from)[0];
252     if (reservedBlock.equals(0)) {
253       showMessage("Parking validation", "You never had a parking reservation for this
              place");
254     }
255     else {
256       if (reservedBlock.greaterThan(EthBlocks.latest.number)) {
257         showMessage("Parking validation", "You can park for place until block number " +
              reservedBlock);
258       }
259       else {
260         showMessage("Parking validation", "Your parking for place is over since block
              number " + reservedBlock);
261       }
262     }
263   }
264
265   /**
266    * Verify if address exists as place with free slots in contract and block number is in
267    * future, if not show a corresponding message for all cases
268    * @param to address of a place in contract
269    * @param block number to reserve or estimate costs
270    * @param checkFreeSlots if validating free slots
271    * @returns {boolean} true if data valid
272    */
273   function isDataValid(to, block, checkFreeSlots) {
274     if (to === 'undefined') {
275       showMessage("Data verification", "Please insert place address");
276     }
277     else {
278       if (!parkingplaces.existsPlace(to)) {
279         showMessage("Data verification", "Please insert an existing place address");
280       }
281       else {
282         if (block === undefined || block <= EthBlocks.latest.number) {
283           showMessage("Data verification", "Please insert a block number in future");
284         }
285         else {
286           if (checkFreeSlots && parkingplaces.getFreeSlotCount(to,
                    EthBlocks.latest.number).equals(0)) {
287             showMessage("Data verification", "Please wait for free slots or take another
                  place");
288           }
289           else {
290             return true;
291           }
292         }
293       }
294     }
295     return false;
296   }
297
298   /**
299     * Make a reservation until future block for place if selected account is unlocked
300     * @param to the address of the parking place
301     * @param block the block until which to reserve
302     * @param value the costs in wei to pay
303     * @returns {*} the hash of the transaction
304     */
305   function reservation(to, block, value) {
306     var from = TemplateVar.getFrom('.from .dapp-select-account', 'value');
307     return parkingplaces.reserveSlot(to, block, {from: from, gas: 300000, value: value});
308   }
309
310   /**
311     * Checks if the address is one of your ethereum accounts
312     * @param address an ethereum address
313     * @returns {boolean} true if the given address is one of your ethereum accounts
```

```
314       */
315     function isOwnAccount(address) {
316       for (i = 0; i < EthAccounts.find().fetch().length; i++) {
317         if (EthAccounts.find().fetch()[i].address === address) {
318           return true;
319         }
320       }
321       return false;
322     }
323
324     /**
325       * Formatting of an unix timestamp to a 'hh:mm:ss' string
326       * @param timestamp the unix timestamp of a block in seconds
327       * @returns {string} time formatted as hh:mm:ss
328       */
329     function formatTS(timestamp) {
330       var date = new Date(timestamp * 1000);
331       var hours = date.getHours();
332       var minutes = "0" + date.getMinutes();
333       var seconds = "0" + date.getSeconds();
334       return hours + ':' + minutes.substr(-2) + ':' + seconds.substr(-2);
335     }
336
337     /**
338       * Add a new marker on map of google maps api and add infowindow with actual place data
339       * @param owner the place address and key for all key value array
340       * @param timeout for animated marker falling down
341       */
342     function addMarkerWithTimeout(owner, timeout) {
343       //add marker at location of place from contract parkingplaces
344       window.setTimeout(function () {
345         var marker = new google.maps.Marker({
346           title: places[owner][1],
347           position: {lat: Number(places[owner][2]), lng: Number(places[owner][3])},
348           map: GoogleMaps.maps.map.instance,
349           animation: google.maps.Animation.DROP,
350           icon: getIcon(owner)
351         });
352         //add marker to key-value-array
353         markers[owner] = marker;
354         //adding information for each place
355         addMarkerInfo(owner);
356       }, timeout);
357     }
358
359     /**
360       * Clear all listener for existing marker, create new click listener with infowindow
361       * @param owner the place address and key for all key value array
362       */
363     function addMarkerInfo(owner) {
364       //add info window as click event and remove existing animation and listener
365       var marker = markers[owner];
366       google.maps.event.clearInstanceListeners(marker);
367       //close infowindow if existing
368       if (placeInfos[owner] !== undefined) {
369         placeInfos[owner].setMap(null);
370       }
371       //add info window to key-value-array and update icon
372       placeInfos[owner] = getPlaceInfowindow(owner);
373       marker.setIcon(getIcon(owner));
374       //add click listener to stop animation, update and open infowindow and update icon
375       marker.addListener('click', function () {
376         if (marker.getAnimation() !== null) {
377           marker.setAnimation(null);
378         }
379         //close infowindow if existing
380         if (placeInfos[owner] !== undefined) {
381           placeInfos[owner].setMap(null);
```

```
382       }
383       //update and open
384       placeInfos[owner] = getPlaceInfowindow(owner);
385       marker.setIcon(getIcon(owner));
386       placeInfos[owner].open(GoogleMaps.maps.map.instance, marker);
387     });
388   }
389
390   /**
391     * Generates an infowindow with aktual information about a place and its slots
392     * @param owner the place address and key for all key value array
393     * @returns {google.maps.InfoWindow} an infowindow of google maps api for marker
394     */
395   function getPlaceInfowindow(owner) {
396     //add information of place and slot from contract parkingplaces
397     var slotInfo =
398             '<li><b>name: </b>' + places[owner][1] + '</li>' +
399             '<li><b>owner: </b>' + owner + '</li>' +
400             '<li><b>latitude: </b>' + places[owner][2] + '</li>' +
401             '<li><b>longitude: </b>' + places[owner][3] + '</li>' +
402             '<li><b>slots total: </b>' + parkingplaces.getSlotCount(owner) + '</li>' +
403             '<li><b>slots free: </b>' + parkingplaces.getFreeSlotCount(owner,
                                          EthBlocks.latest.number) + '</li>' +
404             '<li><b>next free slot: </b>' + parkingplaces.getNextFreeBlock(owner,
                                          EthBlocks.latest.number) + '</li>';
405     return new google.maps.InfoWindow({
406       content: slotInfo
407     });
408   }
409
410   /**
411     * Change marker image depending on slot availability and centers plus animate marker
412     * @param owner the place address and key for all key value array
413     */
414   function updateMarker(owner, toCenter, toAnimate) {
415     if (toCenter) {
416       GoogleMaps.maps.map.instance.setCenter(markers[owner].getPosition());
417     }
418     if (toAnimate) {
419       markers[owner].setAnimation(google.maps.Animation.BOUNCE);
420     }
421     else {
422       markers[owner].setAnimation(null);
423     }
424     addMarkerInfo(owner);
425   }
426
427   /**
428     * Shows a modal dialog with header and message
429     * @param header the header of a modal dialog
430     * @param message the message of a modal dialag
431     */
432   function showMessage(header, message) {
433     EthElements.Modal.show({
434       template: 'modal_info',
435       data: {
436         header: header,
437         message: message
438       }
439     });
440   }
441
442   /**
443     * Get icon for marker corresponding to its place free slots percentage.
444     * @param owner the place address and key for all key value array
445     */
446   function getIcon(owner) {
447     var slots = parkingplaces.getSlotCount(owner);
```

```
448     var slots_free = parkingplaces.getFreeSlotCount(owner, EthBlocks.latest.number);
449     var diffPercent = slots_free.dividedBy(slots).times(100).floor();
450     if (slots.equals(0) || slots_free.equals(0) || diffPercent.lessThan(RED_THRESHOLD)) {
451       return 'parking_icon_red.png';
452     }
453     if (diffPercent.lessThan(YELLOW_THRESHOLD)) {
454       return 'parking_icon_yellow.png';
455     }
456     return 'parking_icon_green.png';
457   }
458
459   /**
460     * load contract with ABI definition and contract address
461     * @returns {Contract} ethereum contract object
462     */
463   function loadContract() {
464     var contract_abi = ABI;
465     return web3.eth.contract(contract_abi).at(CONTRACT_ADDRESS);
466   }
467 }
```

LISTING 22: "PARKINGPLACES"- JAVASCRIPT

C.3 Testcases für Contract und Dapp

Für die Evaluation des Contract und der Dapp wurde basierend auf dem Use Case aus Abb. 21 sechs Testcases mit Testfällen erstellt, die im Folgenden mit dem erwarteten Ergebnis des jeweiligen Testfalls im Detail beschrieben sind. Für die Durchführung der Testcases in Tab. 32 sind drei Ethereum Accounts und mind. 1 ETH erforderlich.

1	**Testcase "close ParkingPlaces"**	**<expected>**
1.1	Deployment Contract mit controller, Konstruktor (ohne Parameter), close Funktion und Modifier isController vom 1. Account	Contract Account mit controller im Speicher erstellt und Funktionen close und controller sind verfügbar
1.2	Message Call controller	Adresse des 1. Account wird zurückgegeben
1.3	Transaktion "close" vom 2. Account	Transaction Error
1.4	Transaktion "close" vom 1. Account	Code und Storage des Contract Account ist gelöscht und interne "suicide" Transaktion wurde ausgelöst
2	**Testcase "add ParkingPlace"**	**<expected>**
2.1	Deployment Contract wie 1.1 mit Struktur Place, Slots, Array places, Funktion addPlace, Funktion existsPlace und Event PlaceAdded	wie 1.1 nur zusätzlich sind Funktionen places, addPlace, existsPlace und PlaceAdded verfügbar
2.2	Message Call places (ohne Index)	BigNumber Exception
2.3	Transaktion "addPlace" vom 1. Account mit place owner = 1. Account	Contract Account Speicher hat Place in Array aufgenommen und Eventlog der Transaktion enthält einen Eintrag
2.4	Message Call places (ohne Index)	Place wird als Array zurück gegeben
2.5	Transaktion "addPlace" vom 1. Account mit place owner = 2. Account	Contract Account Speicher hat Place in Array aufgenommen und Eventlog der Transaktion enthält einen Eintrag

2.6	Transaktion "addPlace" vom 1. Account mit place owner = 1. Account	keine Änderungen im Contract Account Speicher und kein Eintrag im Eventlog der Transaktion
2.7	Transaktion "addPlace" vom 2. Account mit place owner = 2. Account	Transaction Error
2.8	Message Call places mit Index 0	erster Place als Array[3] wird zurückgegeben
2.9	Message Call places mit Index 1	zweiter Place als Array[3] wird zurückgegeben
2.10	Message Call places mit Index 2	BigNumber Exception
2.11	Message Call existsPlace mit place owner = 1. Account	gibt true zurück
2.12	Message Call existsPlace mit place owner = 2. Account	gibt true zurück
2.13	Message Call existsPlace mit place owner = 3. Account	gibt false zurück
2.14	Testfall 1.4 in Testcase 1 auführen	wie 1.4
3	**Testcase "update ParkingPlace"**	**<expected>**
3.1	Deployment Contract wie 2.1 mit Mapping placeOf, Funktion addSlots, Modifier isOwner und Event SlotsAdded	wie 2.1, zusätzlich sind Funktionen addSlots und SlotsAdded verfügbar
3.2	Testfall 2.3 in Testcase 2 ausführen	wie 2.3
3.3	Transaktion "addSlots" vom 1. Account mit place owner = 1. Account und 3 Slots	Contract Account Speicher zusätzlich eine Map mit 1 Eintrag, der mit dem als Key (place owner) ein Array an Slots referenziert Eventlog der Transaktion enthält einen Eintrag
3.4	Transaktion "addSlots" vom 1. Account mit place owner = 1. Account und weiteren 2 Slots	Contract Account Speicher hat in dem über die Map referenzierte Array weitere Slots aufgenommen Eventlog der Transaktion enthält einen Eintrag
3.5	Transaktion "addSlots" vom 2. Account mit place owner = 1. Account und 2 Slots	Transaction Error
3.6	Testfall 1.4 in Testcase 1 auführen	wie 1.4
4	**Testcase "show Places"**	**<expected>**
4.1	Deployment Contract wie 3.1 mit Funktionen getSlotCount, getFreeSlotCount und getNextFreeBlock	wie 3.1, zusätzlich sind Funktionen getSlotCount, getFreeSlotCount und getNextFreeBlock verfügbar
4.2	Testfall 2.3 in Testcase 2 ausführen	wie 2.3
4.3	Message Call getSlotCount mit place owner = 1. Account	gibt 0 zurück
4.4	Message Call getFreeSlotCount mit place owner = 1. Account	gibt 0 zurück
4.5	Message Call getNextFreeBlock mit place owner = 1. Account	gibt 0 zurück

4.6	Message Call getSlotCount mit place owner = 2. Account	gibt 0 zurück
4.7	Message Call getFreeSlotCount mit place owner = 2. Account und zukünftiger Blocknr	gibt 0 zurück
4.8	Message Call getNextFreeBlock mit place owner = 2. Account	gibt 0 zurück
4.9	Testfall 3.3 in Testcase 3 ausführen	wie 3.3
4.10	Message Call getSlotCount mit place owner = 1. Account	gibt 3 zurück
4.11	Message Call getFreeSlotCount mit place owner = 1. Account und Blocknr vor 4.9	gibt 0 zurück
4.12	Message Call getFreeSlotCount mit place owner = 1. Account und Blocknr nach 4.9	gibt 3 zurück
4.13	Message Call getNextFreeBlock mit place owner = 1. Account	gibt Blocknr der Transaktion von 4.9 zurück
4.14	Testfall 3.4 in Testcase 3 ausführen	wie 3.4
4.15	Message Call getSlotCount mit place owner = 1. Account	gibt 5 zurück
4.16	Message Call getFreeSlotCount mit place owner = 1. Account und Blocknr vor 4.14	gibt 3 zurück
4.17	Message Call getFreeSlotCount mit place owner = 1. Account und Blocknr nach 4.14	gibt 5 zurück
4.18	Testfall 1.4 in Testcase 1 ausführen	wie 1.4
5	**Testcase "estimate Costs"**	**<expected>**
5.1	Deployment Contract wie 4.1 mit Variable blockCosts, Konstruktor mit Argument blockCosts = 1 Finney und Funktion calculateEstimatedCosts	wie 4.1, zusätzlich sind Funktionen calculateEstimatedCosts und blockCosts verfügbar sowie Contract Account enthält blockCosts im Speicher
5.2	Message Call blockCosts	gibt 1000000000000000 (Wei) zurück
5.3	Message Call calculateEstimatedCosts mit zwei Blocknr (die Zweite ist 5 Blocks größer)	gibt 5000000000000000 (Wei) zurück
5.4	Message Call calculateEstimatedCosts mit zwei Blocknr (die Erste ist 5 Blocks größer)	gibt 0 zurück
5.5	Testfall 1.4 in Testcase 1 ausführen	wie 1.4
6	**Testcase "reserve Slot" und "check Parker"**	**<expected>**
6.1	Deployment Contract wie 5.1 mit Event Reservation und Transaction, Funktion reserveSlot, getReservedBlock, internen Funktionen payReservation und getNextFreeSlot	wie 5.1, zusätzlich sind Funktionen reserveSlot, getReservdBlock, Transaction und Reservation verfügbar
6.2	Testfall 2.3 in Testcase 2 ausführen	wie 2.3
6.3	Testfall 3.4 in Testcase 3 ausführen	wie 3.4

6.4	Message Call getReservedBlock mit place owner = parker = 1. Account	gibt ein Array mit zwei Einträgen zurück: an Index 0 die Blocknr der Transaktion von 6.3 und an Index 1 eine 0
6.5	Message Call getReservedBlock mit place owner = 1. Account und parker = 2. Account	gibt ein Array mit zwei Einträgen zurück: an Index 0 eine 0 und an Index 1 eine 0
6.6	Transaktion reserveSlot vom 2. Account mit place owner = 1. Account und aktuelle Blocknr + 15 sowie 15 Finney Value	Contract Account Speicher hat Parker und Block für Slot im Array aktualisiert, das Eventlog der Transaktion enthält zwei Einträge und es wurden zwei interne Transaktionen getriggert für den Transfer von ETH vom Contract Account in Summe von 15 Finney
6.7	Message Call getReservedBlock mit place owner = 1. Account und parker = 2. Account	gibt ein Array mit zwei Einträgen zurück: an Index 0 die übergebene Blocknr der Transaktion aus 6.6 und an Index 1 eine 0
6.8	Message Call getFreeSlotCount mit place owner = 1. Account und der aktuellen Blocknr (kleiner als die aus 6.6)	gibt 1 zurück
6.9	Transaktion reserveSlot vom 2. Account mit place owner = 1. Account und aktuelle Blocknr + 30 sowie 15 Finney Value	Transaction Error, 15 Finney werden nicht transferiert
6.10	Transaktion reserveSlot vom 2. Account mit place owner = 1. Account und aktuelle Blocknr + 30 sowie 30 Finney Value	wie 6.6 nur mit 30 Finney
6.11	Transaktion reserveSlot vom 3. Account mit place owner = 1. Account und aktuelle Blocknr + 30 sowie 30 Finney Value	wie 6.6 nur mit 30 Finney
6.12	Message Call getFreeSlotCount mit place owner = 1. Account und der aktuellen Blocknummer (kleiner als die aus 6.10)	gibt 0 zurück
6.13	Transaktion reserveSlot vom 1. Account mit place owner = 1. Account und aktuelle Blocknr + 10 sowie 10 Finney Value	Transaction Error, 10 Finney werden nicht transferiert
6.14	Message Call getReservedBlock mit place owner = 1. Account und parker = 2. Account	gibt ein Array mit zwei Einträgen zurück: an Index 0 die übergebene Blocknr der Transaktion aus 6.10 und an Index 1 eine 0
6.15	Message Call getReservedBlock mit place owner = 1. Account und parker = 3. Account	gibt ein Array mit zwei Einträgen zurück: an Index 0 die übergebene Blocknr der Transaktion aus 6.11 und an Index 1 eine 1
6.16	Message Call getNextFreeBlock mit place owner = 1. Account	gibt die kleinere Blocknr aus 6.14 und 6.15 zurück
6.17	Testfall 1.4 in Testcase 1 ausführen	wie 1.4
7	**Testcase "allowReservation"**	**<expected>**
7.1	Deployment Contract wie 6.1 mit Modifier allowReservation	wie 6.1

7.2	Transaktion "addPlace" vom 1. Account place owner = 2. Account	wie 2.3
7.3	Transaktion "addSlots" vom 1. Account place owner = 2. Account Adresse und 3 Slots	wie 3.4
7.4	Transaktion reserveSlot vom 1. Account mit place owner = 2. Account und aktueller Blocknr + 15 sowie 15 Finney Value	Transaction Error, 15 Finney werden nicht transferiert
7.5	Transaktion reserveSlot vom 2. Account mit place owner = 2. Account und aktueller Blocknr + 15 sowie 15 Finney Value	wie 7.4
7.6	Transaktion reserveSlot vom 3. Account mit place owner = 2. Account und aktueller Blocknr + 15 sowie 15 Finney Value	wie 6.6
7.7	Testfall 1.4 in Testcase 1 ausführen	wie 1.4
8	**Testcase "hasValue"**	**<expected>**
8.1	Deployment Contract wie 7.1 mit Modifier hasValue	wie 7.1
8.2	Transaktion "addPlace" vom 1. Account mit place owner = 2. Account und 15 Finny Value	wie 7.4
8.3	Transaktion "addSlots" vom 1. Account mit place owner = 2. Account und 3 Slots mit 15 Finny Value	wie 7.4
8.4	Transaktion an Contract ohne bekannte Funktion mit 15 Finny Value	wie 7.5
8.5	Testfall 1.4 in Testcase 1 ausführen	wie 1.4

TAB. 32: TESTCASES FÜR CONTRACT "PARKING PLACES"