Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Entwurf und Implementierung von Smart-Contracts für die Sensordatenerfassung

Carlo Schamberger

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer/in:	Dr. rer. nat. Frank Dürr
Beginn am:	30. März 2019
Beendet am:	30. September 2019

Kurzfassung

In den vergangen Jahren haben Corwd-Sensing-Anwendungen immer mehr an Relevanz gewonnen. Mit der Zunahme von mobilen Geräten nahm auch die Verwendung dieser als Sensoren zur Datenerhebung zu. In dieser Arbeit betrachten wir hierbei die Nutzung eines solchen Systems zur Verfolgung mobiler Objekte. Smart-Contracts bieten seit wenigen Jahren die Möglichkeit, dezentral auf einer Blockchain Anwendungen zu implementieren, welche nicht auf einen einzelnen Server angewiesen sind. Smart-Contracts sind transparent und können außerhalb der ersichtlichen, implementierten Möglichkeiten nicht manipuliert werden. Mobile-Target-Tracking-Anwendungen selbst sind keine Neuheit mehr, wohl aber die dezentrale Ausführung dieser auf einer Blockchain. Im Rahmen dieser Arbeit wurde eine Mobile-Target-Tracking-Anwendung implementiert, welche mithilfe von Crowd-Sensing ein Objekt sucht. Die erfassten Daten werden an den Smart-Contract in der Ethereum-Blockchain gesendet, dort gespeichert und nach erreichen bestimmter Kriterien ausgewertet. Die korrekten Meldungen sollen vom Smart-Contract mit einem Ether-Betrag belohnt werden. Dazu werden in dieser Arbeit verschiedene Algorithmen zum Finden einer konsistenten Menge von Sichtungen diskutiert. Die Herausforderungen der Implementierung und die Evaluation der entstandenen Smart-Contracts werden in dieser Arbeit vorgestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Akronyme

API Application Programming Interface. 24

CPU Central Processing Unit. 48

DAG Directed Acyclic Graph - Gerichteter Azyklischer Graph. 7

dApp dezentralisierte Applikation. 17

EVM Ethereum Virtual Machine. 22

ID Identifikator. 27

IDE Integrated Development Environment - Integrierte Entwicklungsumgebung. 48

IoT Internet of Things. 17

JSON JavaScript Object Notation. 24

lat Latitude - Längengrad. 30

LLL Lisp-like Language. 23

lon Longitude - Breitengrad. 30

NPM Node Package Manager. 24

OS Operating System - Betriebssystem. 48

PoS Proof of Stake. 21

PoW Proof of Work. 17

RAM Random-access memory. 48

RPC Remote Procedure Call. 24

UTM Universal Transverse Mercator (Koordinatensystem). 56

1 Einleitung

1.1 Motivation

Kryptowährungen gewinnen, seit der Einführung von Bitcoin [Nak+08] 2008, immer mehr an Bedeutung in der Gesellschaft. Und das nicht nur, weil sie eine interessante Geldanlage sind oder aus verschiedensten Gründen häufig in der Presse ein Thema sind, sondern vor allem auch, da sie viele neue technische Möglichkeiten bieten. Der PoW-Mechanismus (Proof of Work) zur Konsensfindung bot erstmals die Möglichkeit für eine dezentrale digitale Währung, die bis auf 51%-Attacken sicher ist. Bereits Bitcoin ermöglichte die Implementierung einer schwachen Version von Smart-Contracts, allerdings noch nicht Turing-vollständig.

Die erste Kryptowährung, die auf Smart-Contracts ausgelegt ist, war Ethereum. Bei Ethereum gibt es nicht nur „Wallet Accounts“ wie bei Bitcoin, die jeweils einem Nutzer gehören und zwischen denen Beträge transferiert werden können, sondern auch „Contract Accounts“. Ist ein solcher „Contract Account“ erst einmal in der Blockchain, kann er von außen nur noch über die im Kontrakt implementierten Schnittstellen beeinflusst werden. Der Code des Kontrakts ist öffentlich und kann von allen Nutzern der Blockchain eingesehen werden. Damit ist es erstmals möglich, einen Betrag in der Blockchain zwischenzulagern, der nach Erfüllung vorher spezifizierter Bedingungen an die entsprechenden Accounts transferiert wird oder Code in die Blockchain zu migrieren, der nur noch über Transaktionen mit Parametern angesprochen werden kann. [But+14] In solchen Smart-Contracts können ganze Programme geschrieben werden. Diese sind heutzutage eher bekannt unter dem Namen dApp (Dezentralisierte Application).

Obwohl es seit dem ersten Release von Ethereum 2015 möglich ist, Smart-Contracts zu verwenden, finden sie in der breiten Bevölkerung bisher wenig Anwendung. Im August 2018 gab es gerade mal zwei Ethereum-dApps, die über 1.000 tägliche Nutzer hatten [Pen18] und das ist ein Jahr später, im August 2019, noch genauso [Dap19]. Bei anderen günstigeren Kryptowährungen, die Smart-Contracts unterstützen, sieht es nur wenig besser aus. Die Kosten, die bei der Nutzung von dApps an die Miner anfallen und der Mehraufwand der Beschaffung entsprechender Kryptowährung, stehen nicht in Relation zum Nutzen der dezentralen Ausführung der Applikation, solange diese nicht unbedingt erforderlich ist.

Die Idee ist es nun zu untersuchen, in wie fern es möglich und effizient ist, Crowd-Sensing-Anwendungen in Form von Smart-Contracts zu implementieren. Crowd-Sensing bezeichnet das Sammeln größerer Mengen Sensordaten von verschiedenen mobilen Geräten, in der Regel Smartphones, Smart Wearables oder ähnlichen, welche kollektiv diese teilen und auswerten, um etwas zu analysieren, messen, mappen oder etwas vorauszusagen. Gründe für das partizipieren der Nutzer bei solchen Anwendungen können Unterhaltung, Leistungen oder finanzieller Vorteil sein [ZYS+15]. Außerdem ist Crowd-Sensing ein essenzieller Bestandteil vieler IoT-Anwendungen [LSN+18].

Oft ist es allerdings undurchsichtig, was genau mit den erfassten Daten geschieht und ob man eine entsprechende Gegenleistung erhält. Hier könnten Smart-Contracts Abhilfe schaffen, da ein solcher nicht mehr manipuliert werden kann, sobald er in der Blockchain ist und die Funktionsweise von jedem Nutzer einsehbar ist. Was für Nachteile, Kosten und Sicherheitsrisiken das allerdings mit sich bringt, soll in dieser Arbeit anhand einer exemplarischen Anwendung analysiert werden.

1.2 Aufgabenstellung

In dieser Arbeit sollen jetzt verschiedene Versionen eines Smart-Contracts für eine dezentrale mobile Crowd-Sensing-Anwendung vorgestellt und auf ihre Praktikabilität überprüft werden. Sensordaten Anforderer, die daran interessiert sind Daten zu sammeln, richten den Smart-Contract ein und spezifizieren dabei die gesuchten Sensordaten. Der Anforderer bietet eine Belohnung in Form eines Betrags einer elektronischen Währung an, der bei Empfang der gesuchten Sensordaten automatisch ausgezahlt wird.

Als Anwendungsfall für Smart-Contract-basiertes Crowd-Sensing soll eine Anwendung zur Verfolgung mobiler Ziele (Mobile-Target-Tracking) entworfen werden. Das mobile Ziel sendet ein Bluetooth-Signal aus, das von Nutzern der Anwendung erkannt werden kann. Das Bluetooth-Signal muss davor im Smart-Contract spezifiziert worden sein. Sensordatenerbringer in Form eines Smartphones oder Tablets, die das gesuchte Objekt entdecken, schicken die Position mit einem Zeitstempel an den Smart-Contract und erhalten dafür die ausgeschriebene Belohnung. Es soll allerdings vom Smart-Contract darauf geachtet werden, dass die erhaltenen Sensordaten valide sind und nur wahre Daten belohnt werden. Dafür soll überprüft werden, welche Orte miteinander eine konsistente Menge ergeben.

1.3 Gliederung der Arbeit

Die weitere Arbeit ist in folgender Weise gegliedert:
Kapitel 2 - Grundlagen stellt die, für diese Arbeit verwendeten, Technologien und Softwaretools vor.
Kapitel 3 - Entwurf behandelt die verschiedenen Ansätze, eine konsistente Menge von Sensordaten zu finden und beschreibt die Funktionsweise des zu implementierenden Smart-Contracts und die dafür benötigten Algorithmen in Pseudocode.
Kapitel 4 - Implementierung zeigt, wie der Entwurf als funktionsfähiger Smart-Contract implementiert werden konnte und was es dabei für technische Schwierigkeiten gab.
Kapitel 5 - Evaluation stellt die Ergebnisse des Vergleiches der verschiedenen Implementierungen vor.
Kapitel 6 - Zusammenfassung und Ausblick fasst die Erkenntnisse der Arbeit kompakt zusammen und zeigt Möglichkeiten auf, wie der Implementierte Smart-Contract in Zukunft verbessert werden könnte.

2 Grundlagen

In diesem Kapitel werden die für diese Arbeit verwendeten Technologien, Tools, Software und ihre Funktionsweisen vorgestellt. Außerdem werden jeweils gebräuchliche Alternativen und ihre Unterschiede aufgezählt.

2.1 Blockchain Technologie

Es existiert nach Wissen des Autors keine allgemeine einzelne formale Definition des Begriffes Blockchain, die alle existierenden Blockchains umfasst. Eine möglichst allgemeine Definition nach [Ber17] ist: „Eine Blockchain ist eine verteilte Rechnerarchitektur, bei der ein Computer ein Knoten genannt wird, wenn er am Blockchainnetzwerk teilnimmt. Jeder Knoten hat volles Wissen über alle Transaktionen, Informationen sind geteilt. Transaktionen werden in Blöcke gebündelt, welche sukzessive zur verteilten Datenbank hinzugefügt werden. Es können nur einzelne Blöcke hinzugefügt werden und jeder neue Block muss einen mathematischen Beweis enthalten der verifiziert, dass er Nachfolger der Sequenz bisheriger Blöcke ist. Es sind also alle Blöcke chronologisch miteinander verbunden."

Die Implementierung der ersten Blockchain, Bitcoin, wird im folgenden Abschnitt 2.1.1 kompakt vorgestellt. Auf die Ethereum Blockchain wird genauer in Abschnitt 2.3 eingegangen.

2.1.1 Bitcoin

2008 wurde unter dem Pseudonym Satoshi Nakamoto die digitale Währung Bitcoin vorgestellt [Nak+08]. Bis heute ist nicht eindeutig klar, wer sich hinter dem Pseudonym verbirgt. Ziel war es damals, zu ermöglichen, ohne eine dritte Partei, wie etwa einer Bank, online zu bezahlen. Die Idee hinter Bitcoin ist, dass jeder Nutzer einen Private und einen Public Key besitzt, jeweils in Form einer 256 Bit Zahl. Der Public Key ist, ähnlich der Kontonummer eines Bankaccounts, eine Adresse, an die Beträge der Währung transferiert werden können. Der Private Key wird, ähnlich wie eine PIN-Nummer, benötigt, um auf den unter dem Public Key gespeicherten Betrag zuzugreifen.

Transaktionen werden von einem Timestamp-Server mit einem Zeitstempel versehen, um sie chronologisch sortierbar zu machen. Die Verifikation einer Transaktion geschieht über das dezentrale Netzwerk, an dem jeder mit einem Computer als Miner teilnehmen kann. Diese Miner haben die Aufgabe Transaktionen zu verifizieren und sich auf eine korrekte Reihenfolge der Blöcke zu einigen. Dafür sollen die Miner ein schwieriges mathematisches Rätsel lösen, das so designt ist, dass es im Schnitt alle zehn Minuten gelöst wird. Die Korrektheit der Lösung ist dabei einfach verifizierbar. Wenn jemand das Rätsel löst, darf er den nächsten Block erstellen und erhält dafür Bitcoins. Diese Schürfmethode nennt sich PoW (Proof of Work) [GKW+16]. Außerdem erhalten die Miner für von ihnen verifizierte Transaktionen eine Transaktionsgebühr, die von der Transaktion abgezogen

wird. Da jede Transaktion auf einem der Blöcke gespeichert ist und nur einmalig vorkommen kann, kann es nicht zum Problem des double-spending kommen, welches vor Bitcoin, für unabhängige elektronische Währungen ungelöst war. [Nak+08]

2.2 Smart-Contracts

Der Begriff Smart-Contract ist zurückzuführen auf Nick Szabo, der diesen 1997 erstmals definiert hat. Die Idee ist ein Vertrag in Softwareform, den zu brechen entweder teuer oder unmöglich ist und der von außen und vor allem von Nichtvertragspartnern, nicht manipuliert werden kann. Laut Szabo sollte ein Smart-Contract folgende Charakteristiken erfüllen:

- Beobachtbarkeit - Alle Teilnehmer am Kontrakt sollen einsehen können wie gut die Vertragspartner den Vertrag einhalten.
- Online Vollstrekbarkeit - Es soll garantiert sein, dass die im Vertrag vereinbarten Bedingungen erfüllt werden.
- Verifizierbarkeit - Im Falle eines Konfliktes zwischen den Vertragspartnern soll der Vertrag überprüfbar sein.
- Privatheit - Die Daten des Smart-Contracts sollen nur für die Teilnehmer am Vertrag sichtbar sein. [Sza97]

Auf diese Punkte muss der Entwickler des Smart-Contracts bei der Implementierung selbst achten und entscheiden, welche Sichtbarkeit für welche Daten notwendig ist.

In dieser Arbeit wurde für die Implementierung Solidity verwendet, auf welches in Abschnitt 2.4 genauer eingegangen wird, unter anderem auch auf die Möglichkeiten der Sichtbarkeit. Da es 1997 die Blockchaintechnologie noch nicht gab, war ein Smart-Contract nur mit einem vertrauensvollen Prüfer möglich. Die erste Blockchain, die auf die Implementierung von Smart-Contracts ausgelegt ist, war Ethereum 2015 [But+14]. Bei Ethereum können, in Form von Smart-Contracts, ganze Programme in die Blockchain migriert werden, nur limitiert von den Berechnungs- und Speicherkosten, die an die Miner gezahlt werden müssen.

Abbildung 2.1 zeigt, wie ein Block einer öffentlichen Blockchain einen Smart-Contract hält. Der Code und Speicher des Kontrakts befinden sich dabei auf der Blockchain, die Berechnungen werden von den Minern durchgeführt. Der Smart-Contract wird ausgeführt, wenn er eine entsprechende Nachricht von einem Nutzer erhält. Dabei können Geld und/oder Daten ausgetauscht werden. Der neue Zustand des Smart-Contracts wird in den nächsten geschürften Block in die Blockchain geschrieben [DAK+15a].

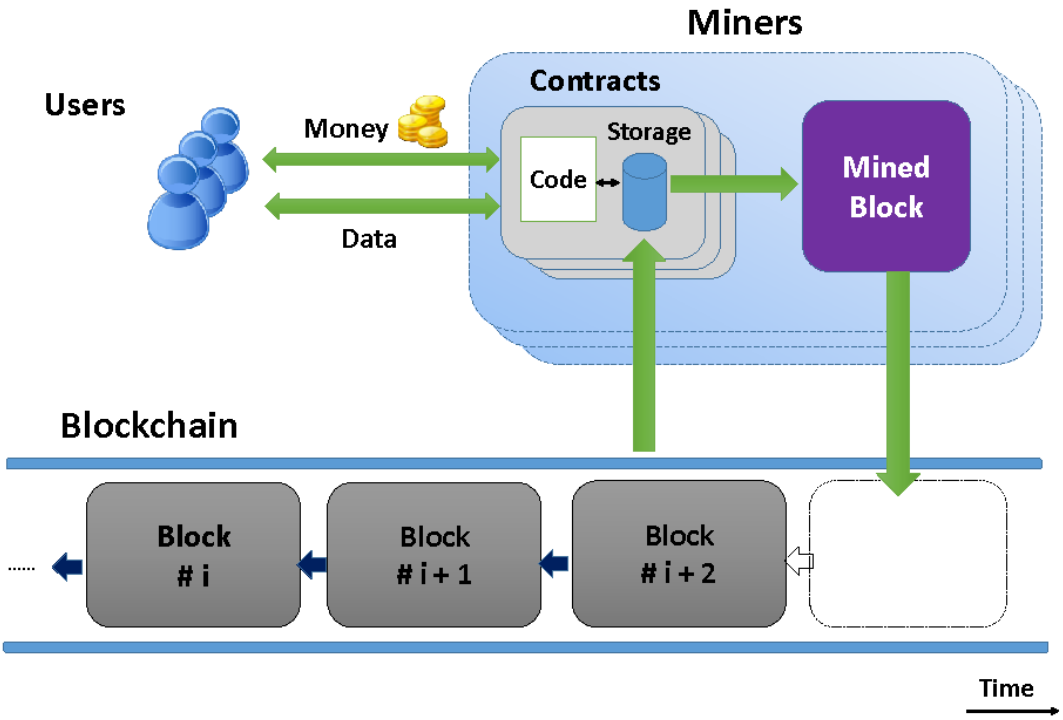

Abbildung 2.1: Funktionsweise eines Smart-Contracts auf der Blockchain [DAK+15a]

2.3 Ethereum

Die Intention bei der Entwicklung von Ethereum war nicht, wie bei Bitcoin, das Versenden von Geld ohne Drittpartei, sondern das Ermöglichen von Smart-Contracts dezentral auf einer Blockchain. Fester Bestandteil der Ethereum-Blockchain ist eine Turing-vollständige Bytecode Sprache, die aus einer höheren Programmiersprache, wie zum Beispiel Solidity [WRB+19], Viper [But17] oder LLL [Edg17], compiliert werden kann [But+14].

Die Konsensfindung der Ethereum Blockchain verwendet derzeit noch PoW, soll in Zukunft aber auf PoS (Proof of Stake) umgestellt werden. Bei PoS wird nicht mehr über ein mathematisches Rätsel entschieden wer den nächsten Block minet, sondern aus einer Kombination aus Zufall, Betrag besitzendem Ether und/oder Alter. Dadurch soll dem hohen Energiebedarf beim Schürfen von Ethereum entgegengewirkt werden [KRDO17].

Die Ethereum Blockchain ist essenziell eine transaktionsbasierte Zustandsmaschine (Automat). Ein Zustand besteht aus mehren Objekten, die Accounts genannt werden. Jeder Account hat eine 20-Byte Adresse. Diese Adresse ist quasi der Public Key des Accounts. Ein Account muss hierbei aber nicht mehr wie bei Bitcoin einen Besitzer haben. Es gibt die sogenannten Contract Accounts, zu denen es keinen Private Key gibt und die stattdessen durch ihren Contract Code kontrolliert werden. An einen solchen Contract Account kann, wie an jeden anderen Account, Ether transferiert werden, er verfährt damit allerdings nur genau so wie im Contract des Accounts spezifiziert wurde. Ein Ethereum Account enthält vier Felder:

- nonce: Ein Zähler um sicherzustellen, dass jede Transaktion nur einmal verarbeitet wird.

Einheit	Ein Ether entspricht
wei	10^{18}
Kwei	10^{15}
Mwei	10^{12}
Gwei	10^{9}
Szabo	10^{6}
Finney	1000
Ether	1

Tabelle 2.1: Tabelle der kleineren Einheiten von Ether [DAK+15b]

- ether balance: der aktuelle Ether-Betrag auf dem Account
- contract code: falls es sich um einen Contract Account handelt steht hier der Code
- storage: der Speicher auf dem der Contract arbeitet

Zwischen Ethereum Accounts können Nachrichten oder Transaktionen ausgetauscht werden. Nachrichten können entweder von Nutzern initiiert werden, oder von einem Smart-Contract. Sie können optional Daten beinhalten und es kann auf sie geantwortet werden. Das zeigt, dass Ethereum Nachrichten das Konzept von Funktionen beinhalten. [But+14]

Transaktionen bezeichnen in Ethereum Datenpakete, die eine Nachricht von einem Nutzeraccount beinhalten. Eine Transaktion beinhaltet den Empfänger der Nachricht, eine Signatur des Absenders, einen Ether Betrag, Daten und zwei Werte genannt STARTGAS und GASPRICE. Um extrem lange und unendliche Laufzeit von Code auf der Blockchain zu vermeiden und die Miner zu entschädigen, kostet jeder Berechnungsschritt Gas. STARTGAS ist das Gas-Limit und GASPRICE der Preis in Wei, der an die Miner für jeden Berechnungsschritt gezahlt wird. Wei ist die kleinste Einheit von Ether, siehe Tabelle 2.1 für alle Ethereinheiten.

Wird das Gaslimit einer Transaktion überschritten, werden alle Änderungen, die die Transaktion bewirkt, bis auf die Bezahlung der Miner, rückgängig gemacht. [But+14]

Die Ethereum Virtual Machine (EVM) ist eine quasi-Turing-vollständige Maschine. Quasi nur, weil die Anzahl der Berechnungsschritte durch das Gaslimit begrenzt ist. Die von der EVM interpretierte Sprache heißt EVM-Bytecode. Der EVM-Bytecode wird aus einer höheren Programmiersprache, in dieser Arbeit der Kontrakt-orientierten Sprache Solidity, kompiliert [Woo14].

Die EVM hat folgende drei Möglichkeiten Daten zu speichern:

- storage: Jeder Account hat diesen Speicher, der persistent zwischen Funktionsaufrufen und Transaktionen ist. Die Kosten steigen Linear zum belegen Speicher.
- memory: Diesen Speicher bekommt ein Smart-Contract bei jedem Aufruf neu. Memory muss mit Gas bezahlt werden. Die Kosten skalieren quadratisch zum benötigten memory-Speicher.
- stack: Die EVM ist eine Stack-Maschine. Hier werden Berechnungen durchgeführt.

In Solidity muss der Entwickler explizit angeben, was in storage und memory gespeichert werden soll.

2.4 Solidity

Ethereum unterstützt verschiedene Skriptsprachen zum Schreiben von Smart-Contract-Code. Die verbreitetste unter ihnen ist Solidity, eine objektorientierte Sprache mit statischem Typsystem und einer JavaScript-ähnlichen Syntax. Als Datentypen gibt es in Solidity nur Booleans, Integers, statische und dynamische Arrays und Address, welcher ein spezieller Datentyp für eine 20 byte Ethereum Adresse ist. [WRB+19]

Zu beachten ist hier, dass es gar keine Fließkommazahlen gibt, da diese relativ hohe Gaskosten mit sich bringen würden. Festpunktzahlen werden momentan auch noch nicht vollständig unterstützt. [WRB+19, S.47]

Funktionen in Solidity können vier verschiedene Sichtbarkeiten haben, Variable drei:

- external: Funktionen mit dieser Sichtbarkeit sind Teil des Kontrakt-Interface. Sie können also von anderen Kontrakten oder über Transaktionen aufgerufen werden. External Funktionen können nicht intern aufgerufen werden. Variable können nicht external sein.
- public: Funktionen, die public sind, können extern und intern aufgerufen werden. Für public Variable werden automatisch getter-funktionen generiert.
- internal: Auf Funktionen und Variable, die internal sind, kann nur intern zugegriffen werden, also vom jeweiligen Kontrakt oder einem davon abgeleiteten Kontrakt aus.
- private: Private Funktionen und Variable sind nur von dem Kontrakt aus sichtbar, in dem sie definiert sind.

Zu beachten gilt allerdings, dass alles in einem Kontrakt von außerhalb der Blockchain sichtbar ist. Private hindert nur andere Smart-Contracts am lesen und modifizieren von Daten. [WRB+19, S.82f]

Solidity 0.1.0 wurde erstmals 2015 veröffentlicht. Die Sprache wird seitdem kontinuierlich weiterentwickelt. Eine komplette und aktuelle Version der Solidity-Dokumentation befindet sich online auf dem ethereum/solidity Github. In dieser Arbeit wurde mit Solidity Version 0.5.11 gearbeitet [WRB+19].

Solidity ist neben Viper [But17] die einzige Sprache für Ethereum, die noch aktiv und offen weiterentwickelt wird. Viper ist in der Funktionalität ähnlich zu Solidity, die Syntax orientiert sich aber an Python. Außerdem gibt es noch den veralteten Vorgänger von Viper, Serpent [DAK+15b], die Lisp ähnliche Sprache LLL und Mutan, eine weitere nicht länger unterstützte Sprache.

In dieser Arbeit wurde sich für Solidity entschieden, da es die am weitesten verbreitetste der in Frage kommenden Sprachen ist und auch in der Ethereum Homestead Dokumentation [Hom16] empfohlen wird.

2.5 Solidity Compiler

Die schnellste Möglichkeit Solidity-Code zu kompilieren ist die Remix IDE [yan19], ein Onlinecompiler, mit dem man einfach im Browser Smart-Contracts in Echtzeit kompilieren und testen kann. Für größere Projekte bietet sich allerdings ein Offlinecompiler an. Die Solidity Dokumentation [WRB+19] empfiehlt den von C++ solc abgeleiteten Compiler solcjs. Die Installation von solcjs erfolgt über den Node Package Manager (NPM) [WSR16], welcher in Node.js enthalten ist. Node.js ist eine asynchrone, event-basierte Laufzeitumgebung zur Entwicklung von Anwendungen in JavaScript [Spr18]. Für diese Arbeit wurden Node.js 12.2.0 und NPM 6.9.0 verwendet.

Solcjs compiliert Solidity Code zuerst zu JSON-Dateien. Diese müssen dann miteinander verlinkt werden. Wie genau das manuell funktioniert, kann man auf dem solcjs-Repository nachlesen [19c].

2.6 Truffle Suite

Die Truffle Suite ist eine open-source Entwicklungsumgebung und ein Testframework für Blockchains, die die EVM verwenden. Dabei werden die Sprachen Solidity und Viper unterstützt. Truffle hat den solcjs-Compiler integriert und automatisiert die Verlinkungen und verwaltet die Binärdateien. Diese Suite ermöglicht es mit einem einzigen Befehl, „truffle migrate“, einen Smart-Contract zu compilieren und anschließend in die Blockchain zu migrieren. Die Parameter für die Migration müssen hierzu zuvor in einem JavaScript-File [Osm12] spezifiziert werden. Außerdem umfasst die Truffle Suite eine Konsole, von der aus man direkt mit Smart-Contracts auf der Blockchain interagieren kann. Diese Interaktion ist der Konsole mittels web3.eth (siehe 2.7) möglich.

Für diese Arbeit wurde die Truffle Suite 5.0.20 verwendet.

2.7 web3.eth

Ethereum-Clients sind direkt ansprechbar mittels JSON-RPC. Zum Testen von Smart-Contracts ist JSON-RPC [19b] aber sehr unpraktisch, da Kontrakte auf der Blockchain nur EVM-Bytecode verstehen. Um es Entwicklern zu ermöglichen, einfach mit Smart-Contracts zu interagieren, wurden die in Tabelle 2.2 aufgeführten Bibliotheken implementiert. Diese dienen auch als Schnittstelle zu Anwendungen in der dazu aufgeführten Sprache [Hom16].

Web3.eth ist die Ethereum JavaScript API, die für diese Arbeit verwendet wurde. Das web3.eth-Paket ist ebenfalls über NPM verfügbar.

2.8 Ganache

Ganache ist ein open-source Tool zur Erstellung und Verwaltung einer lokalen Ethereum Blockchain. Mit der Truffle Suite (siehe 2.6) lässt sich ein Smart-Contract in die lokale Ganache-Blockchain migrieren. Ganache ermöglicht es, den Zustand des Smart-Contracts zu verfolgen und zu jeder Zeit die Werte aller Variable einzusehen [Kar18]. Außerdem bietet Ganache die Möglichkeit,

mehrere lokale Ethereum-Accounts zu erstellen. Alle Interaktionen mit der Blockchain werden in einem Logfile aufgezeichnet. In Abbildung 2.2 sieht man die grafische Oberfläche von Ganache in Defaulteinstellung, mit mehreren Ethereum-Accounts, ihren jeweiligen Public Keys und jeweils 100 Ether.

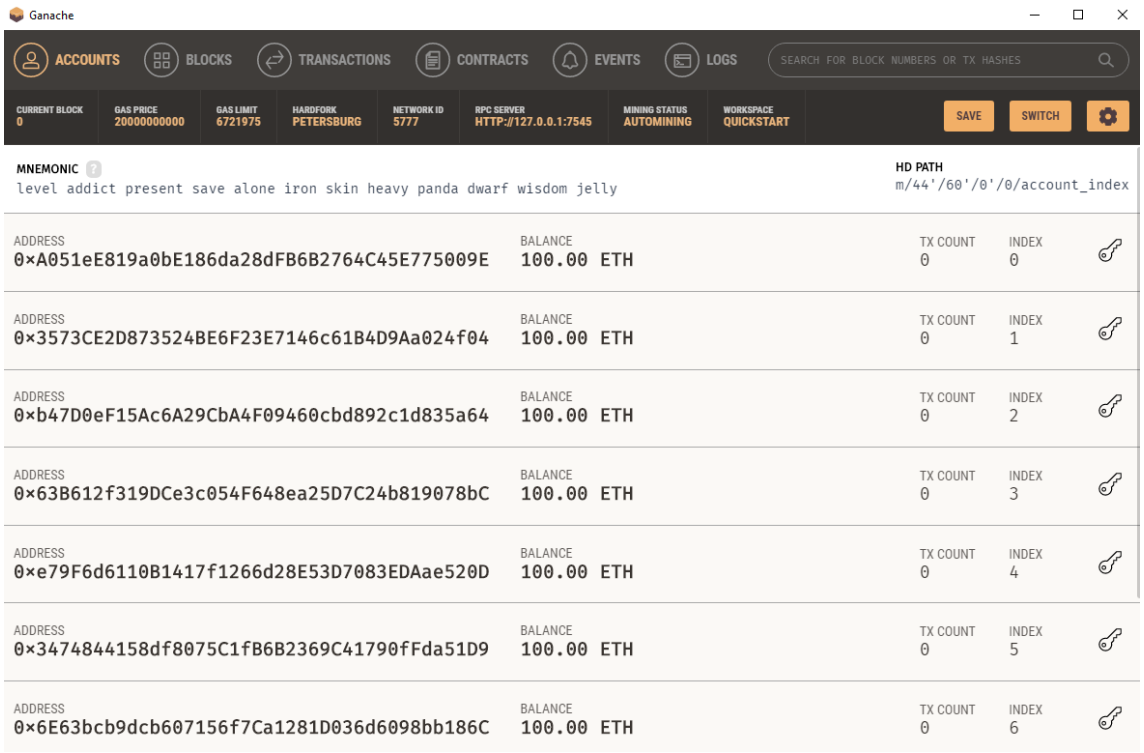

Abbildung 2.2: Grafische Oberfläche von Ganache in Defaulteinstellung

In dieser Arbeit wurde mit Ganache v2.1.1 gearbeitet.

2.9 Crowd-Sensing und Mobile-Target-Tracking

Crowd-Sensing, oder auch Mobile-Crowd-Sensing, benutzt Geräte, die mit Sensoren ausgestattet sind, um Daten aus der Umgebung zu erheben. Sensorgeräte sind hierbei meistens Smartphones, aber auch Tablets, Smart-Wearables, Smartwatches oder Voice Services, wie Alexa von Amazon, können zur Sensordatenerhebung verwendet werden.

Bibliothek	Programmiersprache
web3.js	JavaScript
web3.j	Java
Nethereum	C # .NET
ethereum-ruby	Ruby

Tabelle 2.2: Bibliotheken für die Kommunikation mit Ethereum-Clients [Hom16]

Erfasste Daten können GPS-Position, Geräusche, Temperatur, Helligkeit, Bilder oder wie in der hier Implementierten Anwendung, Bluetoot-Signale aus der Umgebung sein. Dabei wird hier nach dem Bluetooth-Signal gesucht, die übermittelten Daten sind aber GPS-Position und Uhrzeit.

Crowd-Sensing-Anwendungen können, je nachdem was für Daten erfasst werden, in die Kategorien Umwelt-Anwendungen, Infrastruktur-Anwendungen und Soziale Anwendungen kategorisiert werden [GYL11]. Crowd-Sensing ist kritischer Bestandteil vieler IoT-Anwendungen [LSN+18].

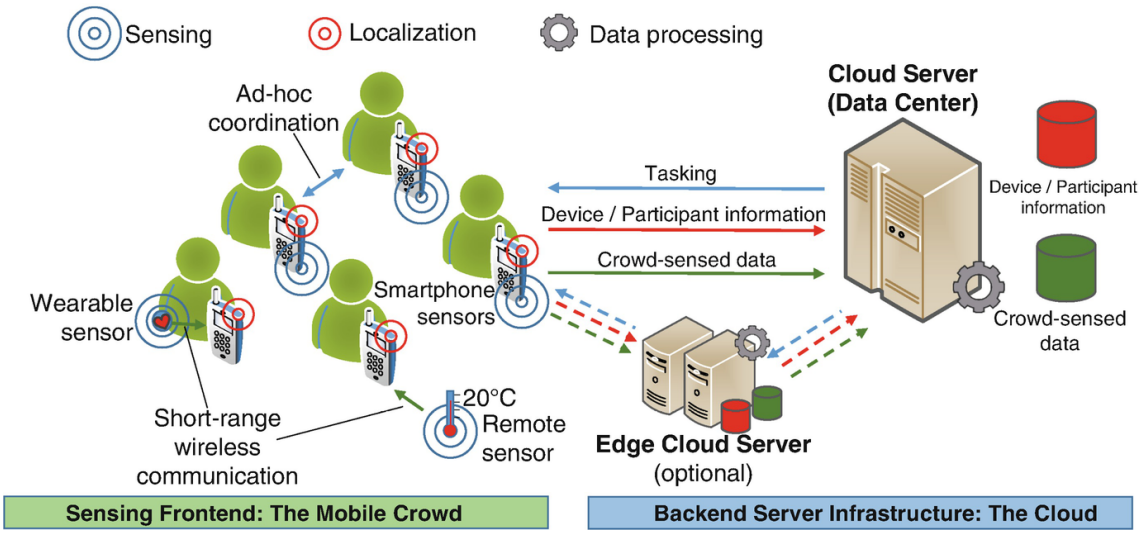

Abbildung 2.3: Herkömmliche Systemarchitektur von Crowd-Sensing [RD18]

In der Regel werden für Crowd-Sensing viele Teilnehmer mit Sensorgeräten benötigt. Abbildung 2.3 zeigt die Systemarchitektur, wie Crowd-Sensing sie normalerweise verwendet. In dieser Arbeit soll aber die Cloud durch eine Blockchain ersetzt werden. Die Anwendung, die entworfen wurde, verwendet Crowd-Sensing zur Verfolgung mobiler Objekte. Das Lokalisieren und Verfolgen eines Objektes mittels beliebiger Sensoren nennen wir Mobile-Target-Tracking [BX09; WWDR11].

Im folgenden Kapitel wird eine Anwendung zum Mobile-Target-Tracking mittels Crowd-Sensing, unter Verwendung der Blockchain zum Speichern und Auswerten der Daten, entworfen.

3 Entwurf

3.1 Übersicht über den Smart-Contract

Es soll ein Smart-Contract für die, in Kapitel 1.2 beschriebene, mobile Crowd-Sensing Anwendung entwickelt werden. Der Kontrakt soll durch den Aufruf eines Konstruktors erstellt werden, der als Parameter die ID des gesuchten Gegenstandes und eine Zeit erhält, bis zu der der Kontrakt spätestens beendet werden soll. Die Nachricht, die den Konstruktor aufruft, ist teil einer Transaktion, welche als Betrag die Belohnung an den Smart-Contract sendet. Ist der Kontrakt auf die Blockchain migriert worden, soll er darauf warten, dass Nutzer der Anwendung das gesuchte Objekt sichten.

Die erste Adresse, die mit einem Smart-Contract interagiert, also in der Regel der Erzeuger, wird „owner“ genannt [But+14]. In unserem Fall ist das der Account, von dem aus der Konstruktor aufgerufen wird, also die Person, die nach dem Objekt sucht und Sensordaten anfordert.

Definition 3.1.1 (Besitzer)
Im weiteren wird der Account, welcher den Smart-Contract erzeugt, der Besitzer genannt.

Definition 3.1.2 (Nutzer)
Die Accounts, welche das gesuchte Objekt finden und Daten an den Kontrakt senden, werden im weiteren Nutzer genannt.

Definition 3.1.3 (Location)
Ein Datenpaket, bestehend aus Längen-, Breitengrad, einem Zeitstempel und der Absenderadresse, das von einem Nutzer an den Smart-Contract gesendet wird, soll fortan Location genannt werden.

Sendet ein Nutzer eine Location an den Smart-Contract, soll dieser sich wie in Abbildung 3.1 dargestellt verhalten:

1. Zuerst soll festgestellt werden, ob der Nutzer bereits eine Location eingesendet hat. Dadurch soll sichergestellt werden, dass kein Nutzer mehrfach Locations in der größten möglichen Menge von konsistenten Locations haben kann und somit nicht mehrfach die Belohnung ausgezahlt bekommt, oder die Wahl der größten Menge beeinflussen kann.
2. Falls nein: Hat der Nutzer bereits eine Location eingesendet, soll die neue gesendete verworfen und der Aufruf des Smart-Contracts beendet werden. Dieser wartet also weiter in der Blockchain auf neue Locations.
3. Falls ja: Hat der Nutzer zuvor noch nie eine Location an den Smart-Contract gesendet, soll die neue Location gespeichert werden.
4. Nach dem Speichern der neuen Location soll entschieden werden, ob eine der Bedingungen erfüllt ist, um den Kontrakt auszuwerten und gegebenenfalls die Belohnung auszuzahlen. Mögliche Bedingungen sind:

   a) Eine bestimmte Anzahl x von Locations im Speicher wurde erreicht.

   b) Die vom Besitzer angegebene Zeit wurde erreicht oder überschritten.

   c) die mächtigste konsistente Menge von Locations hat eine bestimmte Größe erreicht, wozu diese Menge allerdings zuerst ermittelt werden müsste.

5. Falls nein: Es wurde noch kein Auswertungskriterium erfüllt. Der Kontrakt soll also in der Blockchain auf Einsendungen weiterer Locations warten.

6. Falls ja: Die mächtigste konsistente Menge von Locations soll ermittelt werden. Der Smart-Contract soll der Mehrheit glauben, er nimmt also die mächtigste gefundene Menge als Wahrheit an. Die Nutzer der Locations in dieser Menge sind die Gewinner der ausgesetzten Belohnung. Die chronologisch letzte Location wird als der wahre Ort des gesuchten Objektes angenommen. Ist die ermittelte Menge kleiner als ein geforderter Wert, kann der Smart-Contract auch auf weitere Locations warten, oder den Kontrakt terminieren.

7. Die Belohnung, also der Betrag an Ether im Smart-Contract, soll nun durch die Anzahl der Gewinner geteilt und dann an die Gewinner transferiert werden.

8. Zuletzt soll der erfüllte Kontrakt, der jetzt keine Belohnung mehr enthält, zerstört werden, um Platz auf der Blockchain frei zu machen. Falls übrige wei existieren, welche entstehen können wenn die Belohnung nicht genau durch die Gewinner teilbar ist, werden diese zurück an den Besitzer transferiert.

Die Punkte eins bis drei und fünf sind leicht umsetzbar: Ob ein Nutzer bereits eine Location eingesendet hat, kann durch einen Vergleich seiner Adresse mit den Adressen aller gespeicherter Locations ermittelt werden. Die eingesendete Location zu verwerfen und den Aufruf zu beenden, ebenso wie die neue Location zu speichern ist einfach. Die Implementierung hiervon ist in Kapitel 4 erklärt.

Die ersten beiden Auswertungskriterien von Punkt vier lassen sich auch einfach überprüfen. Eine mächtigste Konsistente Menge von Locations zu ermitteln, ist aber aufwändig. Würde man dies beim Empfangen jeder Lokation aufs neue durchführen, hätte man einen erheblichen Rechenaufwand, der ja vom Nutzer in Form von Gas und letztendlich in Ether, bezahlt werden muss. Deshalb bietet es sich an, diese Menge nur ein einziges mal zu ermitteln, und zwar dann, wenn der Kontrakt beendet werden soll. Dies erfolgt unter Punkt 6. Die Konsistenzkriterien werden in Abschnitt 3.2 erläutert. Mögliche Algorithmen zur Ermittlung der mächtigsten konsistenten Menge von Locations werden in Abschnitt 3.3 diskutiert.

Die Verteilung der Belohnung und das Zerstören des Kontrakts werden in Kapitel 4 gezeigt.

3.2 Konsistenzkriterium

Man kann davon ausgehen, dass sich das gesuchte Objekt nicht schneller, als mit einer bestimmten Geschwindigkeit, bewegen kann. Zwei Locations, die räumlich weit voneinander entfernt sind, aber zeitlich nah nacheinander an den Smart-Contract gesendet werden, können also nicht zueinander konsistent sein.

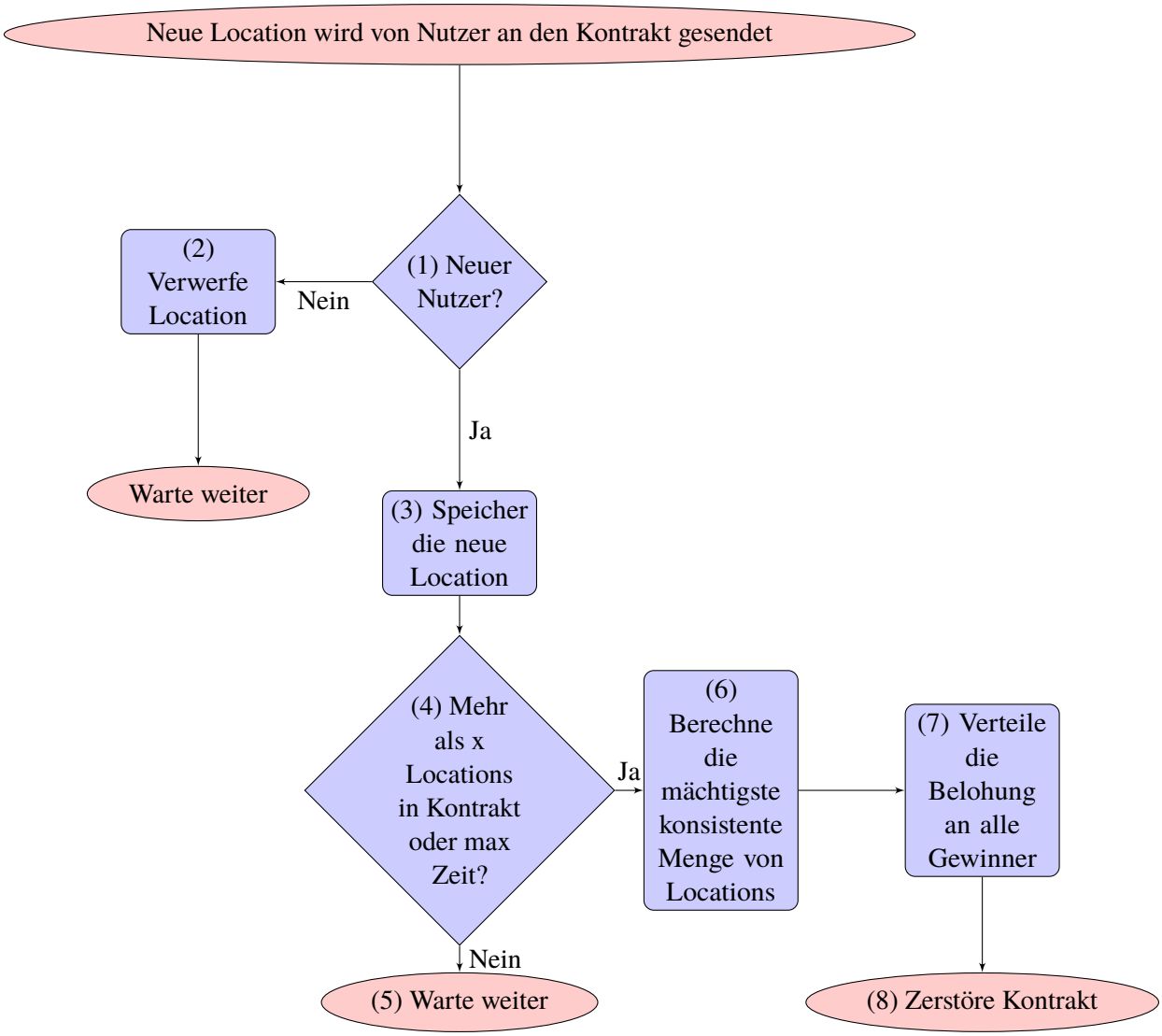

Abbildung 3.1: Flussdiagramm für den groben Ablauf des Smart-Contracts

Die maximale Distanz d_{max}, die zwischen zwei Locations liegen darf, um miteinander eine Konsistente Menge zu bilden, ist abhängig von der maximalen Geschwindigkeit v_{max}, mit der sich das Objekt physisch bewegen kann und den Zeitstempeln der beiden Locations t_1 und t_2:

Definition 3.2.1 (Maximale Distanz konsistenter Locations)

$$d_{max} = |t_1 - t_2| * v_{max}$$

d_{max} muss also kleiner sein, als die räumliche Distanz d zwischen den Locations.

In dieser Arbeit wurde sich dafür entschieden, mit dem geographischen Koordinatensystem zu arbeiten. Eine Location besitzt einen Längen- und einen Breitengrad. Die Abstandsberechnung erfolgt trigonometrisch mit einer, der in Abschnitt 3.2.2 und Abschnitt 3.2.3 vorgestellten Formeln.

3.2.1 Geforderte Genauigkeit

Komplexe Berechnungen auf der Ethereum-Blockchain werden schnell sehr teuer, da jede Berechnung Gas und somit Ether kostet. Hinzu kommt, dass Solidity keine Fließkommazahlen unterstützt und selbst das Rechnen mit Festpunktzahlen momentan noch nicht möglich ist, nur das Speichern dieser. Unterstützt werden von Haus aus nur die vier Grundrechenarten, was zur Folge hat, dass selbst die Betragsfunktion entweder von Hand implementiert werden muss, oder von einem anderen Smart-Contract als Bibliothek importiert werden muss.

Möchte man eine exakte Genauigkeit des Abstands würde es sich also anbieten, diese nicht auf der Blockchain, sondern auf dem Rechner eines der Nutzeraccounts oder des Besitzers, durchzuführen. Diese wäre dann aber von der entsprechenden Partei manipulierbar, was den Vorteil des Smart-Contracts, transparent und nicht manipulierbar zu sein, zunichte machen würde.

Ein anderer Aspekt ist die Geschwindigkeit von Transaktionen und Berechnungen auf der Blockchain. Diese variiert stark und ist davon abhängig, was für einen Gaspreis man bereit ist zu bezahlen. Der Gaspreis wiederum ist, wie der Ethereumkurs ebenfalls, stark schwankend. Auf Webseiten wie der „ETH Gas Station“ [19a] kann man einsehen, welche Dauer man aktuell für eine Transaktion, bei entsprechendem Gaspreis, erwarten kann. Wie teuer die Verwendung der hier vorgestellten Smart-Contracts ist, wird in Kapitel 5 untersucht.

Aber selbst wenn man dazu bereit ist, einen hohen Gaspreis zu bezahlen, kann eine Transaktion frühestens auf dem nächsten geschürften Block verifiziert werden. Im September 2019 liegt die durchschnittliche Zeit pro geschürften neuen Block bei ca. 14 Sekunden [19a]. Hinzu kommt aktuell das Skalierbarkeitsproblem von Ethereum, weshalb bis zur erwarteten Einführung von Ethereum 2.0 [But19] eine Transaktionszeit von 2 Minuten bereits als schnell gilt.

Die lange Dauer einer Transaktion hat zur Folge, dass eine übermäßige Genauigkeit der Entfernungsberechnung, von dem in Definition 3.2.1 durch den Fehler der Zeit von t_1 und t_2 entstehenden Ungenauigkeit, überlagert wird. Deshalb genügt uns im Folgenden die Erde als eine Kugel anzunehmen und die Berechnung mithilfe von Ellipsoiden zu vernachlässigen.

3.2.2 Abstandsberechnung auf der Kugeloberfläche

Nimmt man an, dass die Erde eine Kugel ist, lässt sich die Entfernung zweier Punkte auf ihrer Oberfläche und damit das gesuchte d, mithilfe des Kugeldreiecks, wie folgt berechnen:

Definition 3.2.2 (Abstandsberechnung auf der Erde als Kugel)

$$d = 6378.388 * \arccos(\sin(lat_1) * \sin(lat_2) + \cos(lat_1) * \cos(lat_2) * \cos(lon_2 - lon_1))$$

Hierbei ist 6378.388 der Radius der Erde. lat und lon stehen für die Längen- und Breitengrade der beiden Locations. Die errechnete Distanz d liegt in Kilometern vor. [Sch16, S.70]

Da Winkel aber ebenfalls noch recht aufwändig zu berechnen sind vereinfachen wir diese Formel in Abschnitt 3.2.3.

3.2.3 Vereinfachte Abstandsberechnung

Möchte man den Abstand berechnen ohne Sinus und Cosinus zu benötigen, bietet es sich an die Erdoberfläche als Ebene anzusehen und die Distanz mithilfe des Satzes des Pythagoras zu berechnen:

Definition 3.2.3 (Abstandsberechnung auf der Erde als Ebene)

$$d_x = 71.5 * (lon_1 - lon_2)$$
$$d_y = 111.3 * (lat_1 - lat_2)$$
$$d = \sqrt{(d_x^2 + d_y^2)}$$

111,3 ist dabei der Abstand zwischen zwei Breitenkreisen in Kilometern. Der durchschnittliche Abstand zwischen zwei Längengraden entspricht am Äquator auch 111,3 Kilometern. Dieser Abstand wird zu den Polen hin aber immer kleiner. 71,5 km entspricht dem durchschnittlichen Abstand zwischen zwei Längengraden in unseren Breiten. Mit dieser Formel lässt sich der Abstand zwischen zwei Punkten innerhalb Deutschlands bis auf den Kilometer genau berechnen, wird die Distanz aber größer, nimmt auch der Fehler zu [Kom19].

Angenommen, die Geschwindigkeit mit der sich das gesuchte Objekt maximal bewegen kann sei 200 km/h und das Senden einer Location an den Smart-Contract benötigt eine Minute, dann kann sich das gesuchte Objekt in dieser Minute bereits um 3,3 km bewegt haben. Der Fehler, welcher durch die Verzögerung bei der Kommunikation mit der Ethereum-Blockchain entsteht, ist also immer noch größer, als der Fehler der vereinfachten Formel, zumindest auf kurze Distanzen, bei denen dieser unter einem Kilometer bleibt. Soll das Objekt also nur im Radius von wenigen hundert Kilometern um einen Punkt gesucht werden, genügt es den durchschnittlichen Abstand zwischen zwei Längengraden an diesem Punkt zu ermitteln und die 71,5 in der Formel, aus 3.2.3, durch diesen zu ersetzen.

3.3 Ermittlung der Gewinner

In diesem Kapitel sollen die verschiedenen Ansätze gezeigt werden, um eine möglichst mächtige, konsistente Teilmenge von Locations zu ermitteln. Dies entspricht Punkt 6 in Abbildung 3.1.

3.3.1 Naiver Ansatz

Alle Locations werden in einer mengenartigen Datenstruktur gespeichert. Das kann in diesem Fall ein Array sein, da Locations durch ihre Absenderadresse einzigartig sind. Der naive Ansatz, die mächtigste konsistente Teilmenge von Locations zu berechnen, ist es, alle möglichen konsistenten Teilmengen zu bestimmen und dann die Größte/n auszuwählen. Algorithmus 3.1 zeigt, wie alle konsistenten Teilmengen aufgestellt werden können. Hierbei wird jede Location mit allen Locations jeder Teilmenge auf Konsistenz überprüft. Jede Überprüfung auf Konsistenz beinhaltet eine Abstandsberechnung, welche für sich bereits teuer sein kann. Dies hat eine sehr schlechte Worst-Case-Laufzeit zur Folge.

Algorithmus 3.1 Naiver Algorithmus zur Bestimmung der mächtigsten Menge von konsistenten Locations

```
1:  function GEWINNERNAIV
2:      for all x ∈ Locations do
3:          for all K ∈ konsistenteTeilmengen do
4:              konsistent ← true
5:              for all y ∈ K do
6:                  if x und y nicht zueinander konsistent sind then
7:                      konsistent ← false
8:                  end if
9:              end for
10:             if konsistent == true then
11:                 konsistenteTeilmengen ← x ∪ K
12:             end if
13:         end for
14:         konsistenteTeilmengen ← x // x alleine wird neue konsistente Teilmenge
15:     end for
16:     return G = K ∈ konsistenteTeilmengen|max(|K|) // mächtigste Teilmenge
17: end function
```

Der Worst-Case tritt auf, wenn alle Locations zueinander konsistent sind. In diesem Fall berechnet der Algorithmus 3.1 die Potenzmenge der Locations. Damit kommt man auf 2^n Teilmengen, wobei n die Anzahl der Locations ist, was zur Folge hat, dass der gesamte Algorithmus in $O(2^n)$ liegt. Dies ist für Berechnungen auf der Ethereum-Blockchain, mit größeren Mengen an Locations, nicht praktikabel.

3.3.2 Clustering

Der Algorithmus 3.1 kann aber noch nicht die optimale Lösung für unser Problem sein, da viele unnötige Teilmengen berechnet werden und nicht alle bekannten Informationen, welche über die Locations zur Verfügung stehen, berücksichtigt wurden.

Die Idee ist es jetzt, den Ansatz der dynamischen Programmierung zu verwenden. Es sollen nur noch die Teilmengen gespeichert werden, welche noch am größten werden können. Der vorgeschlagene Algorithmus 3.2 verwendet die Methode des hierarchischen, agglomerativen Clustering [HP13; XT15].

Der Algorithmus 3.2 liefert aber nicht für jede Eingabe das korrekte Ergebnis. Da eine Location hier nie in mehreren Clustern gleichzeitig sein kann, wird immer nur das Cluster gespeichert, von dem aufgrund der geringen Distanz vermutet wird, dass es am wahrscheinlichsten am größten wird. Dafür benötigt der Algorithmus 3.2 nur wenig Speicherplatz. Bei n Locations werden nie mehr als n Cluster im Speicher gehalten und die Tabelle benötigt $n^2/2$ Einträge. Sollte Speicher also sehr teuer sein, könnte dieser Ansatz praktikabel sein.

Ein weiteres Problem dieser Methode ist die Anzahl der benötigten Konsistenzprüfungen. Das in Abbildung 3.2 dargestellte Dendogramm kann bei einer Ausführung von Algorithmus 3.2 entstehen, beispielsweise wenn die Locations a und b ein Cluster (a,b) bilden und c, d und e ein Cluster (c,d,e)

Algorithmus 3.2 Algorithmus zur Bestimmung der mächtigsten Menge von konsistenten Locaions mithilfe hierarchischem, agglomerativen Clustering

```
1:  function GEWINNERCLUSTERING
2:      Erstelle eine Tabelle mit den Locations als Zeilen und Spaltenbeschriftungen. Jede Location
        einzeln sei nun ein Cluster.
3:      Fülle die Tabelle mit den Räumlichen Abständen zwischen den jeweiligen Locations
4:      for all Cluster do
5:          if die Locations der Cluster des kleinsten Eintrages untereinander konsistent sind then
6:              Merge die beiden Cluster
7:              Update in der Tabelle den Druchschnittlichen Abstand der beiden Cluster
8:          else
9:              Prüfe den nächst größeren Eintrag in der Tabelle
10:         end if
11:     end for
12:     return den mächtigsten Cluster
13: end function
```

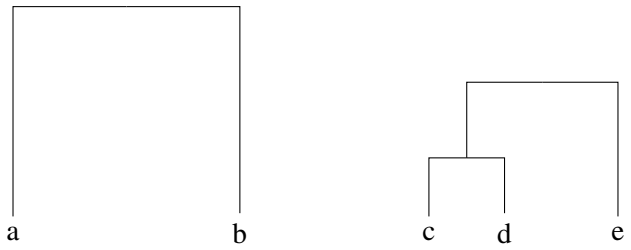

Abbildung 3.2: Dendogram einer exemplarischen Ausführung von Algorithmus 3.2. a,b,c,d und e seien Locations.

bilden. c und d haben den kürzesten Abstand zueinander, da sie zuerst vereinigt wurden. Im letzten Schritt überprüft der Algorithmus, ob die beiden Cluster (a,b) und (c,d,e) zueinander konsistent sind. Dies entspricht Zeile 5 in Algorithmus 3.2. Dazu muss jedes Element aus (a,b) mit jedem Element aus (c,d,e) miteinander auf Konsistenz geprüft werden und nur, wenn alle zu einander konsistent sind, dürfen die Cluster zu einem großen Cluster (a,b,c,d,e) vereinigt werden.

Sei n die Anzahl der Locations und damit die Anzahl der ursprünglichen Cluster, dann wird maximal $n-1$ mal die Vereinigung zweier Cluster durchgeführt, bis eventuell nur noch ein Cluster existiert. Die Vereinigung zweier Cluster benötigt im Worst-Case, wenn beide Cluster $n/2$ Locations enthalten, $n^2/4$ Konsistenzprüfungen. Die for-Schleife, von Zeile 4 bis 11, des Algorithmus 3.2, benötigt also im Worst-Case

$$O((n-1) * \frac{n^2}{4}) \in O(n^3)$$

Konsistenzüberprüfungen. Das Füllen der Tabelle in Zeile 2, ist mit $O(n^2)$ weniger komplex. Algorithmus 3.2 liegt folglich in $O(n^3)$.

3.3.3 Längster Pfad in einem DAG

Algorithmus 3.2 ist zwar schon wesentlich effizienter als Algorithmus 3.1, aber dafür weniger genau. Wäre das Zeitattribut der Locations im Konsistenztest nicht relevant, wäre es wesentlich leichter die mächtigste konsistente Teilmenge zu ermitteln. Bei den beiden bisher untersuchten Ansätzen wurde allerdings vernachlässigt, dass sich alle Locations zeitlich ordnen lassen. Diese zeitliche Relation zwischen den Locations lässt sich ausnutzen, indem man diese, in Reihenfolge des Eintreffens, in einer Liste speichert. Dadurch ist es nicht einmal notwendig, diese nach ihrem Zeitstempel topologisch zu sortieren. Die Locations liegen jetzt also nicht mehr, wie bisher angenommen, als unsortierte Menge im Speicher, sondern als zeitlich vorsortierte Liste.

Der Ansatz ist es nun, einen gerichteten azyklischen Graphen (DAG) aufzubauen und in diesem, den längsten möglichen Pfad zu ermitteln. Den längsten Pfad auf einem allgemeinen Graphen zu bestimmen, ist NP-Hart. Da wir aber, wie in Algorithmus 3.3 gezeigt, einen DAG erzeugen können, ist es leichter, den längsten Pfad zu bestimmen [Kha11]. Dieser ist dann die größte mögliche Teilmenge von Locations. In diesem DAG soll jeder Knoten eine Location sein und die Kanten jeweils konsistente Relationen in Richtung des Vorgängers. Hierzu erweitern wir die Locations um die zwei Attribute längster Pfad und längster Vorgänger, welche initial auf Null (0, nicht null) gesetzt werden. Dann wird wie in Algorithmus 3.3 vorgegangen.

Algorithmus 3.3 Algorithmus zur Bestimmung der mächtigsten Menge von konsistenten Locations, mithilfe eines gerichteten azyklischen Graphen

```
1:  function GEWINNERGRAPH
2:      //Erzeuge den Graphen in Form einer Adjazenzmatrix
3:      for all i ∈ Locations bginnend bei der Ältesten do
4:          for all j ∈ Locations zeitlich nach i, beginnend bei der Ältesten do
5:              if i und j konsistent zueinander sind then
6:                  Adjazenzmatrix in Zeile i Spalte j sei true
7:              else
8:                  Adjazenzmatrix in Zeile i Spalte j sei false
9:              end if
10:         end for
11:     end for
12:     // Längsten Pfad bestimmen
13:     for all i ∈ Locations bginnend bei der Ältesten do
14:         for all j ∈ Locations zeitlich nach i bginnend bei der Ältesten do
15:             if Adjazenzmatrix in Zeile i Spalte j = true then
16:                 if Längster Pfad von i + 1 > Längster Pfad von j then
17:                     Längster Pfad von j sei jetzt Längster Pfad von i + 1
18:                     Längster Vorgänger von j ist jetzt i
19:                 end if
20:             end if
21:         end for
22:     end for
23:     return die Location mit dem Längsten Pfad. Gewinner sind dieser und alle seine längsten
    Vorgänger
24: end function
```

Aus der Adjazenzmatrix alleine lässt sich nicht erkennen, dass es sich hierbei um einen gerichteten Graphen handelt. Erst mit der Information, dass die Locations nach ihrem Index aufsteigend zeitlich sortiert sind und Kanten nur in Richtung älterer Knoten gehen können, erhält man einen DAG.

Wird exemplarisch Algorithmus 3.3 auf dem Szenario aus Abbildung 3.3 ausgeführt, verhält sich dieser wie folgt:

- Der Knoten mit der Nummerierung 1 ist die älteste eingesendete Location. Ihr längster Pfad und längster Vorgänger sind null.
- Knoten zwei wird mit der ersten Iteration der for-Schleife in Zeile 13 auf einen längsten Pfad von eins und den längsten Vorgänger eins upgedated.
- Knoten drei wird mit der zweiten Iteration der for-Schleife in Zeile 13 auf einen längsten Pfad von zwei und den längsten Vorgänger zwei upgedated.
- Knoten fünf und sieben werden mit der dritten Iteration der for-Schleife in Zeile 13 auf einen längsten Pfad von drei und den längsten Vorgänger drei upgedated.
- In der vierten Iteration der for-Schleife in Zeile 13 geschieht nichts, da fünf bereits einen längeren Vorgänger als vier hat.
- Knoten sechs wird mit der fünften Iteration der for-Schleife in Zeile 13 auf einen längsten Pfad von vier und den längsten Vorgänger fünf upgedated.
- Der längste Pfad ist vier, von Location sechs. Gewinner sind sechs und alle seine längsten Vorgänger. Längster Pfad zählt tatsächlich die Kanten des Pfades. Der Längste Pfad enthält fünf Knoten.

Bei n Locations durchläuft die äußere Schleife zur Erstellung der Adjazenzmatrix in Algorithmus 3.3, Zeile drei bis elf n Durchläufe, die innere Schleife, Zeile vier bis zehn, $n-1+n-2+n-3...+n-(n-1)$ Durchläufe:

$$\begin{aligned} & n-1+n-2+n-3...+n-(n-1) \\ &= (1+2+3+...+(n-1)) \\ &= (\frac{n(n+1)}{2})-n \end{aligned}$$

nach der Gaußschen Summenformel.

Das Aufstellen der Adjazenzmatrix liegt somit in $O(n^2)$. Die beiden for-Schleifen zur Bestimmung des längsten Pfades können maximal quadratische Laufzeit haben. Tatsächlich liegt der Algorithmus zum Finden eines längsten Pfades in einem DAG in $O(|V|+|E|)$ [Kha11]. Diese wird hier aber von der quadratischen Anahl an Konsistenztests überlagert. Der Algorithmus 3.3 liegt somit in $O(n^2)$.

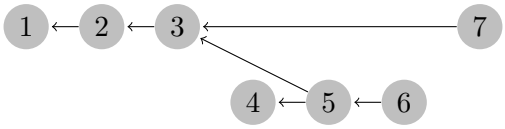

Abbildung 3.3: DAG mit einem eindeutigen längsten Pfad

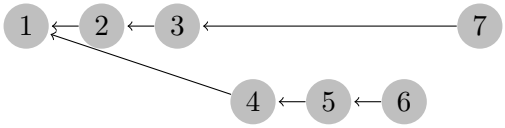

Abbildung 3.4: DAG mit mehreren längsten Pfaden

3.3.4 Mehrere längste Pfade

Das Szenario aus Abbildung Abbildung 3.3 wird von Algorithmus 3.3 korrekt gelöst. Dieser findet den längsten Pfad, der Länge fünf und damit die mächtigste Menge von Locations {1,2,3,5,6}. Bei dem Szenario aus Abbildung 3.4 dagegen findet der Algorithmus 3.3 nur einen der beiden längsten Pfade. Die if-Abfrage in Zeile 15 ergibt nur $true$, wenn der neue Vorgänger einen längeren Pfad hat. So wird Algorithmus 3.3 die Locations {1,4,5,6} als mächtigste konsistente Teilmenge zurückgeben. D ie anderen Locations gehen leer aus, obwohl alle sieben eine Belohnung verdient hätten, da sie alle Teil einer mächtigsten konsistenten Teilmenge sind. Ändert man in der if-Abfrage in Zeile 15 aber nur das „>“ in ein „=“, wird stattdessen die Menge {1,2,3,7} als Gewinner zurückgegeben, also wieder nicht alle sieben Locations.

Der Algorithmus 3.3 kann also noch verbessert werden. Algorithmus 3.4 zeigt, wie durch das Speichern mehrerer Vorgänger, mehrere längste Pfade erkannt werden können. Der Vorgänger wird dazu jetzt in einer Menge gespeichert. Das Zurückgeben der Gewinner in Zeile 23 wird dabei etwas komplexer, wie man an der Implementierung in Listing 4.8 sehen kann, da nun mehrere längste Vorgänger zurückgegeben werden müssen.

Der Algorithmus 3.4 löst jetzt auch das Szenario aus Abbildung 3.4 korrekt und gibt alle sieben Locations als Menge der Gewinner aus. Die gesamte Komplexität von Algorithmus 3.4 bleibt dabei in $O(n^2)$, da nur eine weitere if-Abfrage hinzugefügt wurde. Lediglich der Platzbedarf des Algorithmus steigt marginal.

Algorithmus 3.4 liefert auf jede gültige Eingabe das korrekte Ergebnis und verwendet alle bekannten Informationen. Hinzu kommt, dass die Anzahl der Konsistenzvergleiche, mit wachsender Anzahl von Locations, im Vergleich zu den anderen Ansätzen, am langsamsten steigt. Und zwar nur quadratisch. Dem Autor sind keine weiteren Verbesserungsmöglichkeiten bekannt. Eine weitere Untersuchung, ob der gefundene Algorithmus optimal ist, geht über den Umfang dieser Arbeit hinaus. Für die Implementierung in einem Smart-Contract ist er aber durchaus geeignet.

Algorithmus 3.4 Algorithmus zur Bestimmung der mächtigsten Mengen von konsistenten Locations mit mehreren längsten Pfaden in einem DAG

```
1: function GEWINNERGRAPH
2:     //Graph liegt bereits als Adjazenzmatrix vor
3:     for all i ∈ Locations bginnend bei der Ältesten do
4:         for all j ∈ Locations zeitlich nach i bginnend bei der Ältesten do
5:             if Adjazenzmatrix in Zeile i Spalte j = true then
6:                 if Längster Pfad von i + 1 >= Längster Pfad von j then
7:                     if Längster Pfad von i + 1 > Längster Pfad von j then
8:                         LaengsterVorgnger = ∅
9:                         Längster Pfad von j sei jetzt Längster Pfad von i + 1
10:                    end if
11:                    LaengsterVorgaenger von j fügt i zu LaengsterVorgaenger hinzu
12:                end if
13:            end if
14:        end for
15:    end for
16:    return die Location mit dem Längsten Pfad. Gewinner sind dieser und alle seine längsten
       Vorgänger
17: end function
```

3.4 Abwägung der Ansätze

In dieser Arbeit wurde sich dafür entschieden die Algorithmen 3.3 und 3.4 zu Implementieren, da von diesen die höchste Effizienz erwartet wird und 3.4 das Problem, alle Gewinner zu ermitteln, korrekt löst. In Kapitel 5 werden diese beiden Ansätze in der Praxis verglichen. Der vorgestellte Ansatz mit Clustering (Algorithmus 3.2) findet nicht auf jede Eingabe die korrekte Lösung und ist weniger effizient. Algorithmus 3.1 ist so ineffizient, dass er nicht praktikabel erscheint.

Bei der Wahl des Konsistenzkriteriums muss man für die Implementierung die Einschränkungen von Solidity beachten. Da Solidity nur das Rechnen mit Integer-Zahlen zulässt, und nur die vier Grundrechenarten als Funktionen zur Verfügung stellt, gestaltet sich die Berechnung von Winkeln mittels Sinus, Cosinus und Arcuscosinus als schwierig. Im besten Falle könnten diese über eine Lookup-Tabelle approximiert werden. Da die Implementierung hiervon aber sehr aufwändig wäre und die Verwendung dieser Funktionen unverhältnismäßig teuer, wurde sich hier gegen die Implementierung der Variante aus Abschnitt 3.2.2 entschieden. Stattdessen wurde die einfachere Distanzberechnung aus Abschnitt 3.2.3 im Konsistenztest implementiert.

Falls zwingend eine höhere Genauigkeit, oder andere komplexe Berechnungen erforderlich sind, bietet es sich an solche Berechnungen auszulagern und nicht auf der Blockchain durchzuführen.

4 Implementierung

Implementiert wurde der Smart-Contract zum tracken eines Objektes mittels Crowd-Sensing in den zwei Varianten mit Definition 3.2.3 zur Konsistenzprüfung und Algorithmus 3.3 und Algorithmus 3.4 zur Bestimmung der Gewinner.

Bei Solidity ist das umschließende, klassenartige Konstrukt der „contract“. Außerhalb diesem steht nur die Information welche Version von Solidity verwendet wird. Innerhalb stehen alle Deklarationen, Funktionen und Prozeduren.

In diesem Kapitel wird nun genauer auf die signifikanten Funktionen des Implementierten Smart-Contract eingegangen und anhand dieser, Besonderheiten und Einschränkungen bei der Implementierung von Smart-Contracts aufgezeigt. Bis auf den Algorithmus für die Auswertung der Locations sind die Funktionen der beiden Implementierten Smart-Contracts größtenteils identisch.

4.1 Konstruktor

Der Konstruktor, wie er in Listing 4.1 gezeigt ist, wird vom Besitzer aufgerufen. Dabei hat dieser nur den Tag des gesuchten Objektes und eine Zeit als Parameter mitzuschicken, bis zu der der Smart-Contract aktiv bleiben soll. Die Absenderadresse muss nicht explizit mitgeschickt werden. Diese kann der Smart-Contract mit „msg.sender“ selbst auslesen. Die Belohnung wird wie ein Parameter an den Kontrakt gesendet. Der Modifier „payable“ ist notwendig, damit der Kontrakt Ether empfangen und transferieren kann.

Die Parameter für den Konstruktor müssen bei der Migration des Kontrakts mitgegeben werden. Bei der Migration mit der Truffle Suite (Abschnitt 2.6) schreibt man diese in ein JavaScript-File. Dies zeigt das Listing 4.2. „Suchen“ ist der Name des zu migrierenden Kontrakts. Der Value-Wert ist die mitgesendete Belohnung in Wei. Dieser entspricht in Listing 4.2 zwei Ether, also $2 * 10^{18}$ wei.

Das Zerstören des Kontrakts ist ein Einzeiler. Mit dem Befehl „selfdestruct(owner)“ wird der Kontrakt von der Blockchain entfernt und der verbleibende Ether-Betrag an den Besitzer transferiert. Als Ziel der Transaktion kann auch eine Beliebige andere Adresse angegeben werden.

Listing 4.1 Konstruktor

```
constructor (uint _tag, int _finalTime) public payable {
  owner = msg.sender;
  tag = _tag;
  finalTime = _finalTime;
}
```

Listing 4.2 Aufruf des Konstruktors des Smart-Contract „Suchen“

```
module.exports = function(deployer) {
  deployer.deploy(Suchen, 1, 10, {value: 2000000000000000000});
};
```

Listing 4.3 Funktion zum Anlegen einer neuen Location

```
1  function newLocation(int _lon, int _lat, int _time) public{
2    address payable _a = msg.sender;
3    for (uint p = 0; p < locations.length; p++) {
4      require(locations[p].a != _a, "hat bereits abgestimmt");
5    }
6    //pushe die neue Location ans Ende des Arrays
7    locations.push(Location({
8    a: _a,
9    lon: _lon,
10   lat: _lat,
11   time: _time,
12   laengsterPfad: 0,
13   laengsterVorgaenger: 0
14 }));
15   if (locations.length >= 9 || _time >= finalTime) {longestPath();}
16 }
17
```

4.2 Neue Location

Die in Listing 4.3 gezeigte Funktion ist die einzige des Kontrakts, welche von außen aufrufbar sein soll. Hierzu muss diese natürlich public sein. Alle anderen Funktionen und Variable des Smart-Contract, außer dem Konstruktor, sind private.

Eine Location wurde zuvor als Struct definiert. Eine Location hat ein Adressfeld für die Absenderadresse, Felder für Längen- und Breitengrad, ein Feld für den Zeitstempel und die zwei, für den Algorithmus zur Bestimmung des kürzesten Pfades in einem DAG notwendigen, Felder zum Speichern des längsten Pfades und des längsten Vorgängers.

Davon werden Längen-, Breitengrad und der Zeitstempel vom Nutzer als Parameter des Aufrufs der Funktion mitgegeben. Die Adresse wird, wie schon beim Konstruktor für den Besitzer, mittels „msg.sender“ ausgelesen.

In Listing 4.3 Zeile drei und vier wird einmal über alle bereits gespeicherten Locations iteriert und deren Adressen mit der, der neuen Location verglichen. Hat der Nutzer bereits eine Adresse eingesendet, wird dies hier erkannt und die neue Location wird verworfen. Dies entspricht Punkt zwei in Abbildung 3.1. Wurde die Location verworfen, da die „require“-Funktion $false$ ausgibt, findet automatisch ein Rollback, der von diesem Aufruf hervorgebrachten Änderungen, statt. „require“ ist hierbei ein Solidity-Befehl. Der Rollback findet auf EVM-Ebene statt und kann nicht weiter beeinflusst werden.

Ist die Absenderadresse der neuen Location neu, kann diese gespeichert werden. Zum Speichern der Locations bietet sich ein Array als Datenstruktur an. Zum einen sind Arrays in Solidity die einzige Datenstruktur, die Solidity von Haus aus für das Speichern von Mengen eignet, zum Anderen bleibt die chronologische Reihenfolge der Locations dabei erhalten. Es muss also nicht vor dem Algorithmus 3.3 oder Algorithmus 3.4 nach den Zeitstempeln sortiert werden.

Zum Speichern der neuen Location wird diese also ans Ende des Arrays „locations“ gepusht.

In Listing 4.3 Zeile 14 befindet sich die Abfrage, ob eine der Kriterien zur Auswertung des Kontrakts erfüllt wurde. Im Entwurf entspricht dies Schritt vier von Abbildung 3.1. Die Menge der Locations, bei der es zur Auswertung kommt, ist hierbei hart gecoded. Die späteste Zeit dagegen wird beim Aufruf des Konstruktors in Listing 4.1 vom Besitzer festgelegt. In Listing 4.3 werden zehn Locations angenommen, bis es zur Auswertung des Kontrakts kommt, da ein Array bei null beginnt, solange zuvor nicht bereits die finale Zeit erreicht wurde.

4.3 Konsistenztest

Da Solidity nur das Rechnen mit Integer-Zahlen unterstützt, lässt sich mit der Eingabe von gerundeten Breiten- und Längengraden nur eine Genauigkeit auf 111,3 Kilometer erreichen, da dies die Distanz zwischen zwei Breitengraden ist. Der Abstand zwischen den Längengraden variiert, wird aber nie größer als 111,3 Kilometer, welcher am Äquator erreicht wird. Um die Genauigkeit zu erhöhen, multiplizieren wir die eingegebenen Längen- und Breitengrade mit 10^7. Mit sieben Dezimalstellen lässt sich eine Genauigkeit bis auf elf Millimeter erreichen. Die Funktion in Listing 4.4 geht davon aus, dass Längen- und Breitengrad in der geforderten Form, also mit bereits verschobenen Dezimalstellen, vorliegen. Die ermittelte Distanz zwischen den Locations wird dann durch 10^7 dividiert.

Abgesehen von der Verschiebung der Dezimalstellen, entspricht die in Listing 4.4 gezeigte Funktion Definition 3.2.3.

Die Funktionen um den Betrag („abs“) und die Quadratwurzel („sqrt“) zu Berechnen, müssen hierbei innerhalb des Smart-Contracts implementiert werden. Die Betragsfunktion lässt sich einfach lösen, indem abgefragt wird, ob die Eingabe kleiner Null ist. Die Quadratwurzel wird nach der Babylonischen Methode berechnet [IP96].

4.4 Ein längster Pfad auf einem DAG

Die Funktion „longestPath“ in Listing 4.5 wird aus Listing 4.3 aufgerufen, falls eine der Kriterien für eine Auswertung erfüllt ist. Da sie nur innerhalb des Kontrakts aufgerufen wird, erhält sie die Sichtbarkeit private. Es ist auch zu Testzwecken empfehlenswert, die Sichtbarkeit möglichst niedrig zu halten, da bei der Verwendung von public von Solidity automatisch getter-Methoden erstellt werden, was Gas kostet. Diese Funktion ist sehr teuer und kann während der Lebensdauer des Smart-Contracts nur ein einziges mal aufgerufen werden, da an deren Ende die close-Funktion, welche in Abschnitt 4.1 erklärt wurde, aufgerufen wird. „longestPath“ ist die implementierte Version von Algorithmus 3.3. Dies entspricht Schritt sechs aus Abbildung 3.1.

Listing 4.4 Funktion um die Konsistenz zwischen zwei Locations zu ermitteln

```
1    function konsistenzcheck(uint i, uint j) private returns(bool){
2      int dx = 725000000 * (locations[i].lon - locations[j].lon);
3      int dy = 1113000000 * (locations[i].lat - locations[j].lat);
4      int distance = sqrt((dx * dx) + (dy * dy))/10000000;
5      int dmax = abs(locations[i].time - locations[j].time)*200;
6      if (dmax >= distance){
7        return true;
8      }
9      else{
10       return false;
11     }
12   }
13
```

Zu Beginn der Funktion werden zwei neue interne Variable deklariert, um die Länge des aktuell längsten Pfades und dessen Startknoten zu speichern. Diese werden benötigt, um in Zeile 28 bis 32, das Array der Gewinner zu bestimmen.

In Zeile sieben bis elf wird die Adjazenzmatrix aufgestellt, in der gespeichert wird, welche Locations jeweils konsistent zueinander sind. Die ermittelte Matrix hat die Form einer Dreiecksmatrix. Nur bei der Aufstellung dieser Adjazenzmatrix wird der Konsistenztest aus Listing 4.4 aufgerufen.

Die Berechnung des längsten Pfades ist, bis auf das Speichern des aktuell längsten Pfades und dessen Startknoten in Zeile 19 bis 22, analog zum entworfenen Algorithmus 3.3.

Wurden die längsten Pfade zu allen Knoten bestimmt, steht der Startknoten des ersten längsten Pfades in der Variable „maxLaengsterPfadStart“. Die Implementierung für mehrere längste Pfade findet sich in Abschnitt 4.5. In Zeile 28 bis 32 wird dann über alle längsten Vorgänger iteriert und jeder Knoten auf diesem Pfad in das Array „gewinner“ gepusht.

Daraufhin wird die Funktion „auszahlen“ in Listing 4.6 aufgerufen. In dieser wird die Belohnung durch die Anzahl der Gewinner geteilt, woraufhin an jeden Gewinner sein Anteil der Belohnung transferiert wird.

4.5 Mehrere längste Prade auf einem DAG

Die Implementierung von Algorithmus 3.4 zur Ermittlung mehrerer längster Pfade ist in Listing 4.7 zu sehen. Die Funktion ist ähnlich aufgebaut wie Listing 4.5, mit einigen kleinen Unterschieden.

Der Startknoten des längsten Pfades und der Startknoten des längsten Vorgängers sollen jetzt in einem Array gespeichert werden. Dafür muss die entsprechende Variable in der Definition des Location-struct und in der „newLocation“-Funktion in Listing 4.3 geändert werden.

Listing 4.5 Funktion zur Ermittlung eines längsten Pfades und Ermittlung der Gewinner

```
function longestPath() private {

  uint maxLaengsterPfadStart = 0;
  uint maxLaengsterPfad = 0;

  //Adjazenzmatrix aufstellen
  for (uint i = 0; i < locations.length; i++){
    for (uint j = i+1; j < locations.length; j++){
      adjazenzmatrix[i][j] = konsistenzcheck(i, j);
    }
  }
  //längsten Pfad berechnen
  for (uint i = 0; i < locations.length; i++){
    for (uint k = i+1; k < locations.length; k++){
      if (adjazenzmatrix[i][k] == true){
        if (locations[k].laengsterPfad < (1 + locations[i].laengsterPfad)){
          locations[k].laengsterPfad = 1 + locations[i].laengsterPfad;
          locations[k].laengsterVorgaenger = i;
          if (locations[k].laengsterPfad > maxLaengsterPfad){
            maxLaengsterPfad = locations[k].laengsterPfad;
            maxLaengsterPfadStart = k;
          }
        }
      }
    }
  }
  //gewinner Array bestimmen
  do {
    gewinner.push(locations[maxLaengsterPfadStart]);
    maxLaengsterPfad = maxLaengsterPfad-1;
    maxLaengsterPfadStart = locations[maxLaengsterPfadStart].laengsterVorgaenger;
  }while(maxLaengsterPfad+1 > 0);
  auszahlen();
  close();
}

```

Listing 4.6 Funktion, welche den Gewinnern die Belohnung auszahlt

```
function auszahlen() private{
  uint auszahlungJeweils = (address(this).balance / gewinner.length);
  for (uint i = 0; i < gewinner.length; i++){
    gewinner[i].a.transfer(auszahlungJeweils);
  }
}

```

Listing 4.7 Funktion zur Ermittlung aller längsten Pfade

```
uint[] maxLaengsterPfadStart;
uint maxLaengsterPfad = 0;

function longestPath() private {
  //Adjazenzmatrix aufstellen
  for (uint i = 0; i < locations.length; i++){
    for (uint j = i+1; j < locations.length; j++){
    adjazenzmatrix[i][j] = konsistenzcheck(i, j);
    }
  }
  //längsten Pfad berechnen
  for (uint i = 0; i < locations.length; i++){
    for (uint k = i+1; k < locations.length; k++){
      if (adjazenzmatrix[i][k] == true){
        if (locations[k].laengsterPfad <= (1 + locations[i].laengsterPfad)){
          if (locations[k].laengsterPfad < (1 + locations[i].laengsterPfad)){
            locations[k].laengsterVorgaenger.length = 0;
            locations[k].laengsterPfad = 1 + locations[i].laengsterPfad;
          }
          locations[k].laengsterVorgaenger.push(i);
          if (locations[k].laengsterPfad >= maxLaengsterPfad){
            if (locations[k].laengsterPfad > maxLaengsterPfad){
              maxLaengsterPfadStart.length = 0;
            }
            maxLaengsterPfad = locations[k].laengsterPfad;
            maxLaengsterPfadStart.push(k);
          }
        }
      }
    }
  }
  //gewinner Array ermitten in folgendem Listing
  ...
}


```

Arrays müssen bei Solidity persistent im storage-Speicher liegen und können nicht, wie Variable, innerhalb einer Funktion im memory-Speicher, also nur für die Dauer eines Aufrufs, gespeichert werden. Die Unterschiede zwischen der verschiedenen Speicher sind in Abschnitt 2.3 erläutert. Aus diesem Grund muss das Array, welches die Startknoten der längsten Pfade speichert, außerhalb der Funktion deklariert werden.

Das Aufstellen der Adjazenzmatrix bleibt identisch zu der Suche nach einem einzigen längsten Pfad in Listing 4.5.

Bei der Berechnung des längsten Pfades wird nun unterschieden, ob der neu gefundene längste Vorgänger eines Knotens gleich lang oder länger ist, als die bisher gespeicherten längsten Vorgänger:

Listing 4.8 Gewinner ermitteln, bei der Bestimmung aller längsten Pfade

```
1  ...
2    do {
3      for (uint i = 0; i < maxLaengsterPfadStart.length; i++){
4        bool istSchonGewinner = false;
5        for(uint j = 0; j < gewinner.length; j++){
6          if(locations[maxLaengsterPfadStart[i]].a == gewinner[j].a){
7            istSchonGewinner = true;
8          }
9        }
10       if(istSchonGewinner == false){
11         gewinner.push(locations[maxLaengsterPfadStart[i]]);
12       }
13     }
14     maxLaengsterPfad = maxLaengsterPfad-1;
15     for (uint i = 0; i < maxLaengsterPfadStart.length; i++){
16       if(locations[maxLaengsterPfadStart[i]].laengsterVorgaenger.length > 0){
17         if (locations[maxLaengsterPfadStart[i]].laengsterVorgaenger.length > 1){
18           for (uint k = 1; k <
19           locations[maxLaengsterPfadStart[i]].laengsterVorgaenger.length; k++){
20             maxLaengsterPfadStart.push(locations[maxLaengsterPfadStart[i]].
21             laengsterVorgaenger[k]);
22           }
23         }
24         maxLaengsterPfadStart[i] =
25         locations[maxLaengsterPfadStart[i]].laengsterVorgaenger[0];
26       }
27     }
28   }while(maxLaengsterPfad+1 > 0);
29   auszahlen();
30   close();
31 }
32
```

- Ist der neue längste Vorgänger +1 der aktuellen Location länger, als der gespeicherte längste Pfad zur Location, lösche alle Längsten Vorgänger aus dem Array (Zeile 17 in Listing 4.7) und pushe dann den neuen längsten Vorgänger auf das Array.
- Ist der neue längste Vorgänger +1 der aktuellen Location gleich lang, als der gespeicherte längste Pfad zur Location, pushe den neuen längsten Vorgänger auf das Array mit den anderen längsten Vorgängern

Dieselbe Fallunterscheidung wird nochmal bei der Bestimmung der Startknoten des gesamt längsten Pfades in Zeile 21 bis 26 durchgeführt.

Soweit ist die Berechnung von mehreren längsten Pfaden, bis auf die Fallunterscheidungen nicht schwieriger, als die in Listing 4.5 Die Berechnung des Arrays mit den Gewinnern in Listing 4.8 ist aber leider nicht ganz so simpel, wie bei der Variante mit nur einem längsten Pfad.

Da eine Location auf mehreren längsten Pfaden sein kann, reicht es nicht über alle Pfade zu iterieren, da dann solche Locations mehrfach belohnt werden würden. In Abbildung 3.4 in Abschnitt 3.3.4 beispielsweise, befindet sich der Knoten eins auf zwei längsten Pfaden. Um dies zu verhindern muss überprüft werden, ob eine Location bereits im enthalten ist.

Da es nicht nur mehrere längste Startknoten für den längsten Pfad geben kann, sondern auch jeder Knoten auf dem Pfad Verzweigungen zu mehreren längsten Vorgängern haben kann und jeder dieser Pfade auf weitere Gewinner überprüft werden muss, benötigen wir die große do-whilte-Schleife in Listing 4.8.

Ist das aber erst einmal bestimmt, kann die allgemeine Funktion zur Auszahlung aus Listing 4.6 verwendet werden. Die close-Funktion bleibt ebenfalls identisch zur ersten Variante, welche einem längsten Pfad findet.

5 Evaluation

5.1 Annahmen und Einschränkungen

In dieser Evaluation soll zum einen die Skalierbarkeit der implementierten Ansätze untersucht und zum anderen die implementierten Smart-Contracts miteinander verglichen werden.

Da alle Tests aus Kostengründen auf einer lokalen, mit Hilfe von Ganache (Abschnitt 2.8) erstellten, Blockchain durchgeführt wurden, lassen sich die benötigten Zeiten für Transaktionen nicht mit denen auf dem Ethereum-Mainnet vergleichen. Selbiges gilt für Berechnungen auf der Blockchain, da im Mainnet Berechnungen verteilt stattfinden und auch andere Anwendungen und Nutzer Bandbreite beanspruchen. Die Geschwindigkeit des Mainnets schwankt aus diesen Gründen stark.

Es ist zu erwarten, dass mit der bevorstehenden Einführung von Ethereum 2.0 diese Probleme behoben oder wenigstens verbessert werden [But19]. Wann genau Ethereum 2.0 und damit die PoS-Methode zur Konsensfindung, eingeführt werden soll steht, noch nicht fest. Unter der Annahme, dass sich die Gaspreise für die Ausführung von Operationen nicht verändern, bleiben die in diesem Kapitel durchgeführten Untersuchungen auch unter Ethereum 2.0 gültig.

Da die Preise für Gas und Ether starken Kursschwankungen unterworfen sind, ist es schwierig vorauszusagen, wie viel genau die Verwendung eines Smart-Contracts kosten wird. Alleine im Jahr 2019 schwankte bisher der Preis für ein Ether zwischen 95 und 306 Euro [Eth19a] und der Preis für eine Gaseinheit zwischen zehn und 30 Gwei im täglichen Durchschnitt [Eth19b]. Der maximale Gaspreis hat hierbei einige Ausreißer nach oben, welche aber nicht repräsentativ sind, da jeder Miner seinen Gaspreis selbst wählen kann.

Um die Kosten in Relation setzten zu können, gehen wir in der folgenden Evaluation von Kosten in Höhe von 20 Gwei pro Gaseinheit und 180 Euro pro Ether aus, was im Jahr 2019 realistische Preise sind.

5.2 Versuchsaufbau

Die Spezifikation der Testumgebung, in der die Evaluation durchgeführt wurde, ist in Tabelle 5.1 aufgelistet. Es handelt sich hierbei um einen gewöhnlichen Laptop. Es ist keinerlei spezielle Hardware oder kostenpflichtige Software für die Evaluation nötig. Außer Windows ist sämtliche verwendete Software open-source. Alternativ kann man als Betriebssystem auch eine beliebige Linux-Distribution verwenden, da sämtliche verwendete Software auch für Linux angeboten wird.

Zum Evaluieren eines Smart-Contract wurde zuerst eine leere lokale Blockchain mittels Ganache erstellt. Der zu evaluierende Smart-Contract wird aus der Windows PowerShell-Konsole mit dem Befehl „truffle migrate“ erst kompiliert, falls dieser nicht bereits in kompilierter Form vorliegt und

CPU	AMD A8-7410 APU, 4 x 2.2 GHz
RAM	8 GB
OS	Microsoft Windows 10 Home
IDE	Visual Studio Code 1.38.1
	Truffle 5.0.20
Compiler	Node.js 12.2.0 mit NPM 6.9.0 für Solidity 0.5.11
Virtuelle Blockchain	Ganache 2.1.1

Tabelle 5.1: Spezifikation des Testrechners und installierter relevanter Software

dann in die Ganache-Blockchain migriert. Hierzu müssen die Parameter für den Konstruktor, in Form einer JavaScript-Datei, im Projekt des Smart-Contracts liegen. Alle weiteren Interaktionen mit dem Kontrakt finden aus der Truffle-Konsole mittels web3.eth statt (siehe Abschnitt 2.7).

Ganache erzeugt für jede Interaktion mit der Blockchain ein detailliertes Logfile, aus welchem unter anderem Gaskosten abgelesen werden können. Allein die Migration eines Smart-Contracts erzeugt ein Logfile mit über 3400 Zeilen. Die Gaskosten aus diesem File wurden zum Vergleichen in eine CSV-Datei geschrieben. Die Gaskosten werden immer als Etherbetrag von dem Account abgezogen, von welchem der Aufruf stattfand. Ganache berechnet den voreingestellten Gaspreis von 20 Gwei pro Gaseinheit automatisch.

5.3 Migration

Der Vorgang der Migration des Smart-Contracts auf die Blockchain ist deterministisch. Jedes Migrieren desselben Smart-Contracts sollte also denselben Gasverbrauch haben. Dies wurde durch mehrmaliges Migrieren der jeweiligen Kontrakte bestätigt.

Die Migration des Smart-Contracts mit der Implementierung aus Abschnitt 4.4 kostet genau 990.007 Gas. Multipliziert mit den Gaskosten kostet dies also $990.007 * 20 = 19.800.140$ Gwei. Da ein Gwei 10^{-9} Ether entspricht, kostet das Gas $19.800.140 * 10^{-9}$ $0,01980014$ Ether. Bei einem angenommen Preis von 180 Euro für ein Ether, erhalten wir Kosten in Höhe von $0,01980014 * 180 = 3,5640252$, also ca. 3,56 Euro, für die Migration des Kontrakts. Hinzu kommt noch die Belohnung in beliebiger Höhe, welche mit an den Kontrakt gesendet wird.

Die Migration des Smart-Contracts mit der Implementierung aus Abschnitt 4.5, mit den gleichen Parametern, kostet $1.327.268$ Gas. Die Mehrkosten in Höhe von 337.261 Gas, lassen sich durch 21 Zeilen mehr unkommentierten Code und der Verschiebung einiger Variable in den Speicher erklären.

Man sieht also, dass selbst vermeintlich kleine Unterschiede in der Implementierung einen großen Unterschied bei den Kosten verursachen können. Die Migration der zweiten Variante ist mit umgerechnet ca. 4,78 Euro wesentlich teurer.

5.4 Neue Location

Die beiden implementierten Varianten des Smart-Contracts unterscheiden sich kaum, solange es nicht zur Auswertung kommt. Beim Hinzufügen einer neuen Location bei der Variante aus Abschnitt 4.5, im Vergleich zu der aus Abschnitt 4.4, wird, beim Hinzufügen einer neuen Location, lediglich anstatt der Variable „laengsterVorgaenger = 0“, jeweils ein leeres Array hinzugefügt. Diese beide Aktionen kosten genau gleich viel Gas.

Die Gaskosten für storage-Speicher steigen linear zum bereits belegten Speicher. Das hinzufügen von neuen Locations sollte also für beide Varianten gleiche Kosten erzeugen. Tatsächlich ist dies, bei der Variante mit dem Algorithmus zum Finden mehrerer korrekter Pfade aber, immer 518 Gas teurer. Dies kommt daher, dass der Kontrakt selbst bereits mehr Speicher im Storage beansprucht.

In Abbildung 5.1 wurden jeweils 100 Locations an jeden der beiden Kontrakte gesendet. Man sieht, dass die Gaskosten linear und parallel zueinander steigen. Aufgrund des geringen Unterschiedes an Gaskosten überdecken sich die beiden Plots nahezu.

Es fällt auf, dass die erste gesendete Location teurer ist als ihr Nachfolger. Dies kommt vermutlich daher, dass hierbei die Erzeugung des Arrays bezahlt werden muss, indem die Location-Structs gespeichert werden. Dies kann allerdings nicht nachvollzogen werden, da die Ganache-Logfiles nur Informationen über die Kommunikation liefern und der Gasverbrauch, innerhalb eines Aufrufs des Kontrakts, nicht ersichtlich ist.

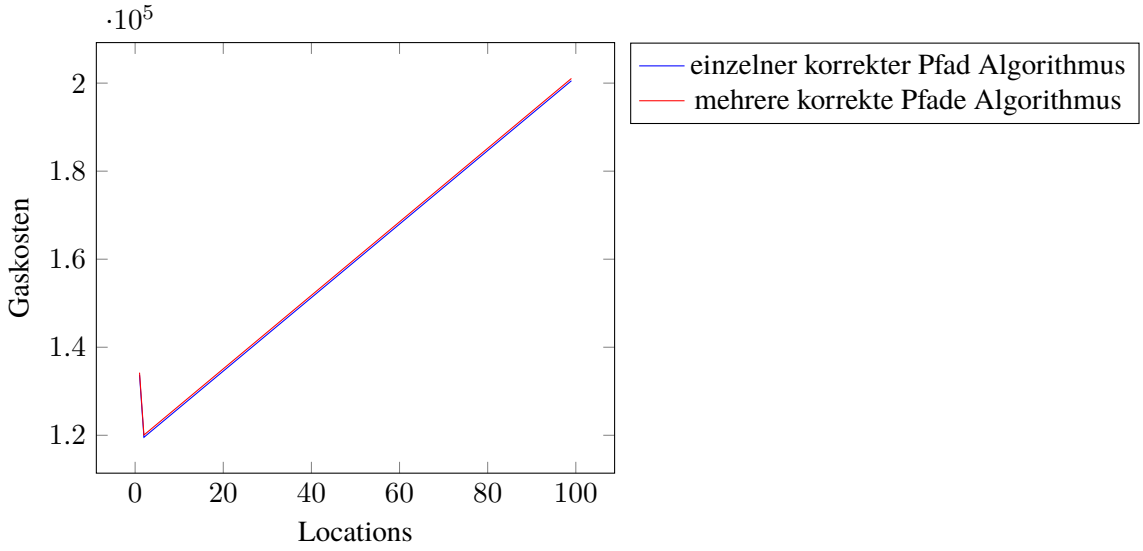

Abbildung 5.1: Gaskosten für das Hinzufügen neuer Locations

In Euro umgerechnet steigen die Kosten der ersten 100 Locations, ausgenommen der Ersten, von 0,43 Euro auf 0,72. Das Einsenden einer Location ist also wesentlich günstiger als das Erstellen des Smart-Contracts.

5.5 Auswertung der Locations

Die Erweiterung des Storage-Speichers und somit das Hinzufügen neuer Arrays wird also linear teurer. Die größten Kosten entstehen allerdings bei der Auswertung dieser Locations. Die Kosten hängen hierbei davon ab, wie viele Locations bis zur Auswertung gespeichert wurden und wie viele konsistent zueinander sind. Der Aufwand des Algorithmus hängt also von der Anzahl an Knoten und Kanten im Graphen ab. Im Folgenden werden diese verschiedenen Fälle untersucht:

- alle Locations sind konsistent zueinander
- keine der Locations ist zu einer anderen konsistent
- es existieren zwei konsistente längste Pfade

Dabei werden nur die Kosten angegeben, die bei dem Aufruf anfallen, welcher die Auswertung auslöst. Die Kosten für die Speicherung der Locations zuvor, sind in Abbildung 5.1 zu sehen.

5.5.1 Alle Locations sind zueinander konsistent

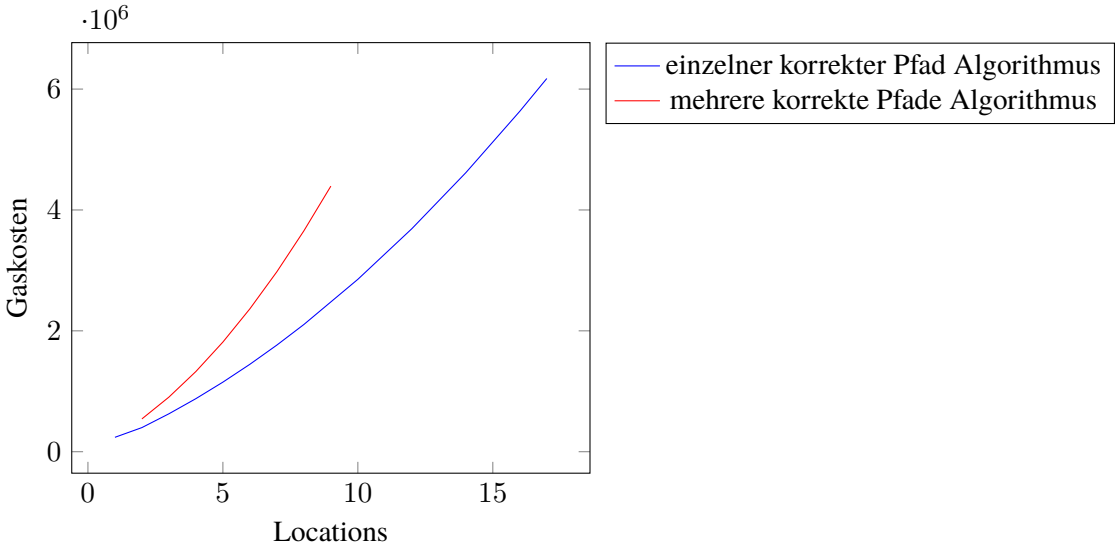

Abbildung 5.2: Gaskosten der Auswertung, wenn alle Locations konsistent zueinander sind

Bei diesem Test haben alle eingesendeten Locations denselben Längen-, Breitengrad und Zeitstempel. Die Locations werden im Arry in Reihenfolge des Eintreffens gespeichert, sind also trotz gleicher Zeitstempel zeitlich sortiert.

Beide Varianten des Smart-Contracts finden denselben längsten Pfad und zahlen die gleichen Gewinner aus. Die Kosten der Auswertung stellt der Graph in Abbildung 5.2 dar.

Hierbei fällt erstmals auf, dass die Test-Blockchain von Ganache ein Gaslimit von 6.721.975 Gas pro Transaktion hat. Dieses wird bei der Variante des Smart-Contracts, der einen korrekten Pfad findet, bei der Auswertung mit 18 Locations überschritten. Bei der Variante, welche mehrere korrekte

Pfade findet, wird dieses Gaslimit bereits bei der Auswertung von zehn Locations überschritten. Dieses Gaslimit ist auf dem Mainnet wesentlich höher, kann in dieser Testumgebung allerdings nicht erhöht werden.

Das Gaslimit entspricht ca. 0,135 Ether, was unter den in Abschnitt 5.1 getroffenen Annahmen 24,31 Euro entspricht. Die Auswertung ist also bereits für eine geringe Anzahl an Locations, in diesem Fall, sehr teuer.

Abbildung 5.3 zeigt den Status von fünf Ethereum-Accounts, welche initial jeweils 100 Ether besessen haben, nachdem dieser Test nach vier Locations ausgewertet wurde. Der Account mit dem Index 0 ist hierbei der Besitzer des Smart-Contracts und hat diesen mit einer Belohnung von zwei Ether migriert. Die Fehlenden 0,03 Ether sind die Gaskosten für die Migration. Die Kosten, welche für die Accounts mit Index 1 bis 3 entstehen, sind die aus Abbildung 5.1. Diese sind hier so gering, dass sie bei nur zwei Nachkommastellen nicht ersichtlich sind. Der vierte Account musste nach Einsendung der Location auch die Kosten für die Auswertung übernehmen, da er diese verursacht hat. Es gibt keine einfache Möglichkeit die Gaskosten für alle Nutzer gleich zu halten. Nach der Auswertung wurden die zwei Ether Belohnung an alle Accounts transferiert, welche zum längsten korrekten Pfad beigetragen haben.

Die Gaskosten der Variante des Smart-Contracts mit Algorithmus 3.4 steigen wesentlich schneller. Dies lässt sich dadurch erklären, dass der Zugriff auf eine Position eines Arrays teurer ist, als der auf eine Variable und mit jeder neuen Location mehr Zugriffe auf Arrays benötigt werden. Alle Schleifen sollten, in diesem Szenario, in beiden Varianten gleich häufig durchlaufen werden.

ADDRESS	BALANCE	TX COUNT	INDEX
0×4175eC5F4370eB53f567B8C1BbD9957737d005DC	97.97 ETH	4	0
0×aD1C0782A43D9C05161e83967fB87c61d8Ad89D7	100.50 ETH	1	1
0×9A7ed8e0Daa2629146f056046252B91c4E929429	100.50 ETH	1	2
0×75ad2A3Dba25A6586C860ddD257713F21c455CB1	100.50 ETH	1	3
0×D00458EB51920f4538D978a8cB5e9b4F4Fc5eD0A	100.47 ETH	1	4

Abbildung 5.3: Fünf Ethereum Accounts nach der Einsendung von vier Locations und darauffolgender Auswertung. Der Account mit Index 0 ist hierbei der Besitzer des Smart-Contracts. Der verwendete Algorithmus ist der zum Finden mehrerer korrekter Pfade.

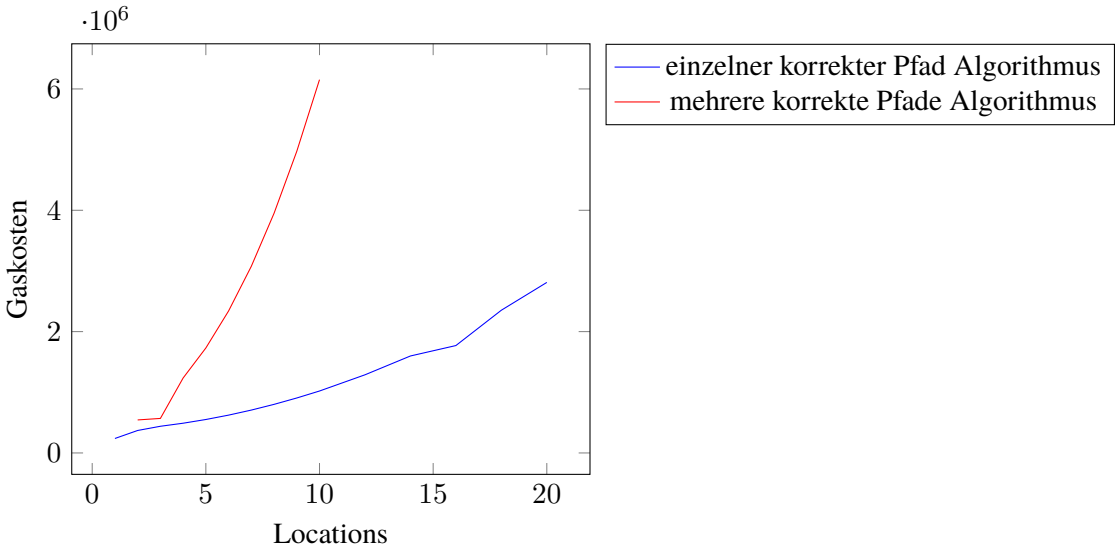

Abbildung 5.4: Gaskosten der Auswertung, wenn keine der Locations konsistent zu einer anderen ist

5.5.2 Keine der Locations ist zu einer anderen konsistent

In diesem Szenario haben alle Locations denselben Zeitstempel, aber unterschiedliche Längen- und Breitengrade. Es ist also keine Location zu einer anderen konsistent. Jede Location für sich bildet aber einen längsten Pfad der Länge null. Die Variante, welche nur einen korrekten Pfad findet, zahlt die erste eingesendete Location aus. Die, welche alle korrekten Pfade findet, zahlt alle eingesendeten Locations aus.

In Abbildung 5.4 sieht man, dass die Kosten für die Variante, welche alle Locations auszahlt, ähnlich schnell steigen, wie in dem Szenario aus Abbildung 5.2. Die Variante, welche nur eine Location auszahlt, wird dabei nur langsam teurer. Dies lässt sich dadurch erklären, dass hierbei, bei n Locations, die Transaktionsgebühren von $n - 1$ Transaktionen gespart werden und im nur eine einzige Location gespeichert werden muss. In der anderen Variante dagegen, muss für jede Location die Fallunterscheidung auf gleichen oder längeren längsten Vorgänger durchgeführt werden.

5.5.3 Zwei konsistente längste Pfade

In diesem Szenario existieren immer zwei konsistente längste Pfade. Die Eingabe besteht also aus Locations von zwei verschiednen Orten zur selben Zeit und, bei ungerader Anzahl, noch eine Location von einem dritten Ort zur gleichen Zeit. Die Variante, welche beide korrekten Pfade erkennt, hat also doppelt so viele Gewinner als die andere Variante. Eine Eingabe von drei Locations ist in diesem Szenario nicht möglich, da hierbei entweder ein oder drei längste Pfade entstehen.

In Abbildung 5.5 erkennt man bei der Variante, die nur einen Pfad auszahlt, eine stufenartige Entwicklung der Funktion. Bei einer ungeraden Anzahl von Locations steigen die Kosten langsamer, da hier keine neuen längsten Pfade entstehen. Sieht man genau hin, ist diese Stufenform auch bei der anderen Variante sichtbar, zumindest bei weniger als zehn Locations. Danach steigt die

Funktion bereits so steil, dass man sie im Graphen nicht mehr erkennen kann. In den Testlogs ist aber ersichtlich, dass die Kosten bei einer ungeraden Anzahl von Locations trotzdem noch etwas langsamer steigen.

Wie zu erwarten war, skaliert auch in diesem Szenario die Variante, welche alle Gewinner korrekt belohnt, wesentlich schlechter.

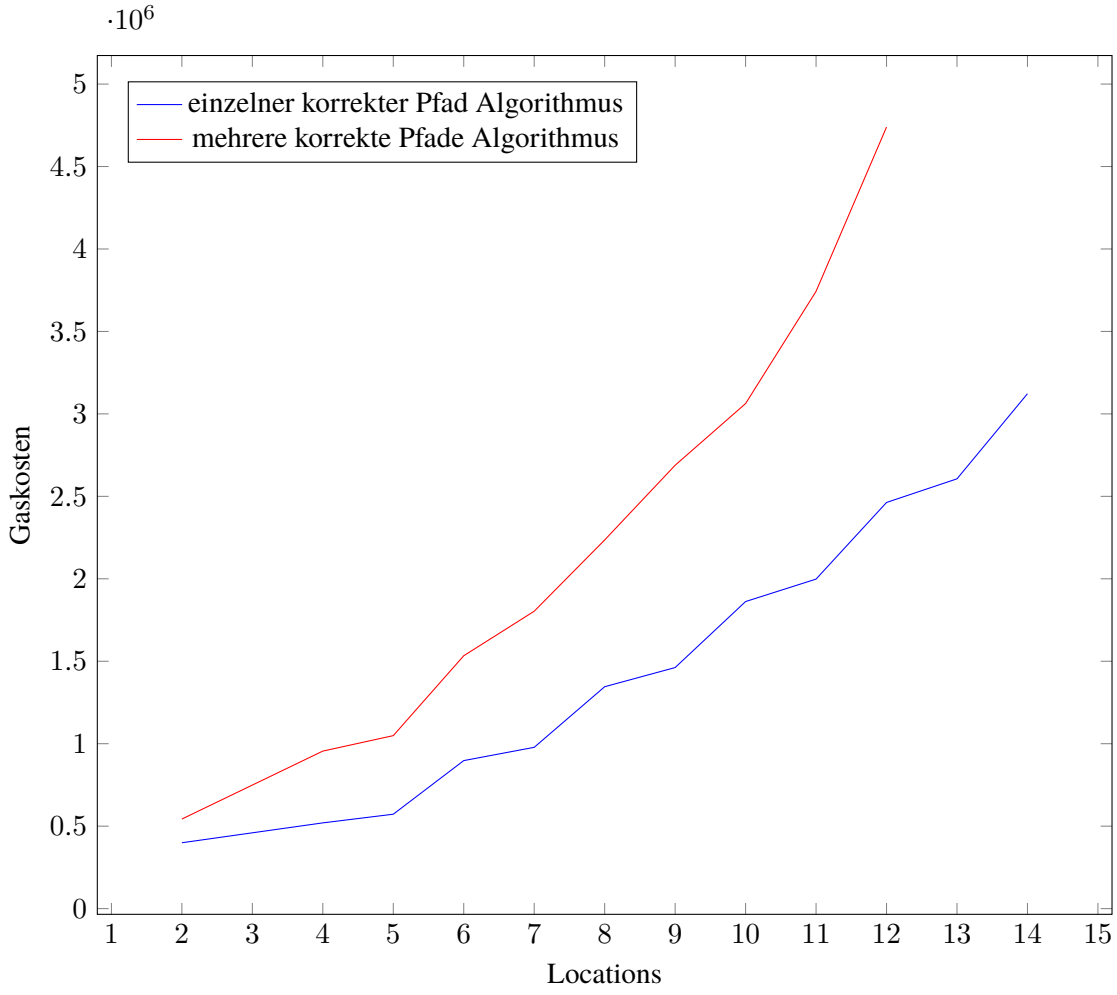

Abbildung 5.5: Gaskosten der Auswertung, wenn genau zwei konsistente längste Pfade existieren

6 Fazit und Ausblick

In dieser Arbeit wird eine kompakte Übersicht über die Entwicklung von Blockchains und Smart-Contracts im Speziellen gegeben und alle notwendigen Werkzeuge für die Implementierung eines Ethereum-Smart-Contracts werden vorgestellt. Der Schwerpunkt liegt hierbei darauf, wie die Blockchain für Crowd-Sensing-Anwendungen verwendet werden kann und wie effizient dies ist.

Es wurde ein Smart-Contract für eine Crowd-Sensing-Anwendung zum Mobile-Target-Tracking in zwei verschiedenen Varianten implementiert. Diese wurden ausgiebig getestet und auf einer Test-Blockchain evaluiert. Hierbei kam heraus, dass die Verwendung der Ethereum-Blockchain sehr teuer ist. Die günstigste Interaktion mit den implementierten Smart-Contracts, das Speichern eines Structs bestehend aus fünf integer Zahlen und einer Adresse, kostet, unter den hier getroffenen realistischen Preisannahmen, 0,42 Euro. Dabei steigt der Preis für Speicher auf der Blockchain linear zum bereits belegten Speicher. So kostet das Speichern des ein-tausendsten solchen Structs im selben Smart-Contract bereits ca. 400 Euro. Gerade bei Crowd-Sensing-Anwendungen, bei denen es interessant ist, viele Daten zu sammeln und auszuwerten, werden solche Preise zum Problem.

Noch schlechter sieht es bei der Ausführung von komplexeren Operationen auf der Blockchain aus. Jeder Lese- und Schreibzugriff auf den Speicher kostet Gas. Algorithmen, welche eine lineare Komplexität haben, oder komplexere Algorithmen auf sehr kleinen Datensätzen können zwar noch praktikabel sein, aber selbst der hier implementierte einfachere Smart-Contract, kam bei einer Eingabe von 17 Datensätzen auf 0,12 Ether Kosten, was über 20 Euro entspricht. Diese hohen Kosten entstehen, obwohl sich hier für eine einfache und ungenaue Berechnungsmethode, zur Konsensfindung zwischen Locations, entschieden wurde.

Unabhängig von den Kosten wurde in dieser Arbeit gezeigt, dass ein Smart-Contract auf der Ethereum-Blockchain dazu in der Lage ist, Sensordaten zu sammeln und auszuwerten. Es ist also möglich, sowohl Speicher als auch Rechenleistung, dezentral in der Blockchain zu verwenden.

Ausblick

Es existieren noch einige Möglichkeiten, den hier implementierten Smart-Contract zu verbessern. Eines der Probleme ist, dass immer der Absender des Aufrufs für die Gaskosten aufkommen muss. Bei den linear steigenden Kosten zum Speichern neuer Locations mag das vernachlässigbar sein, aber die Kosten für die Auswertung können enorm sein. Es wäre fair, diese Gaskosten an den entsprechenden Nutzer zu erstatten. Dazu müssten diese allerdings vor der Auszahlung approximiert werden, da sonst kein Ether mehr im Kontrakt vorhanden ist, oder von einem anderen Account aus bezahlt werden.

Die Genauigkeit der Distanzberechnung kann noch stark erhöht werden, wofür allerdings hohe Kosten zu erwarten sind. Da es sehr teuer ist, auf der EVM Fließkommazahlen zu simulieren, ist es auch nicht geplant, diese in Solidity zu implementieren. Es könnte allerdings in Zukunft möglich

sein, mit Festpunktzahlen in einem Smart-Contract zu arbeiten, ohne diese selbst implementieren zu müssen. Alternativ könnte auch das UTM-Koordinatensystem zur Distanzberechnung verwendet werden. Falls sich alle Locations innerhalb einer UTM-Zone befinden, könnte dies effizienter sein.

Außerdem bleibt zu untersuchen, wie sicher die hier implementierte Anwendung ist. Es wird zwar jeder Adresse erlaubt, nur eine einzige Location einzuschicken, aber es ist nicht sehr teuer neue Ethereum-Accounts zu erstellen. Der hier Implementierte Smart-Contract ist also von Sybil-Attacks angreifbar. Damit ist gemeint, dass ein einzelner Nutzer über mehrere Ethereum-Accounts mehrere Locations einsenden könnte und dadurch das Ergebnis der Auswertung zu seinen Gunsten manipulieren könnte. Mit etwa der Hyperledger Sawtooth-Plattform [DMH17] kann es ermöglicht werden, dass nur noch eine Location pro Gerät eingeschickt werden kann.

Blockchainbasierte Währungen sind starken Preisschwankungen unterworfen, die Kosten für die Ausführung eines Smart-Contracts können schwer vorauszusehen sein. Ethereum ist allerdings die teuerste Blockchain, welche Smart-Contracts unterstützt. Es kann wesentlich günstiger sein jüngere Blockchains zu verwenden, welche ebenfalls Smart-Contracts unterstützen, wie zum Beispiel TRON [TRO18] oder Stellar [Maz15].

Ethereum ist komplett open-source. Es ist also auch möglich, eine eigene Blockchain mit der Ethereum-EVM zu starten, was aber viele andere Herausforderungen mit sich bringt.

Abzuwarten bleibt, was die Umstellung auf Ethereum 2.0 und die Verwendung von PoS zur Konsensfindung, für Änderungen mit sich bringen wird.

Literaturverzeichnis

[19a] *ETH Gas Station*. 22. Sep. 2019. URL: https://ethgasstation.info/ (zitiert auf S. 30).

[19b] *JSON RPC*. 19. Juli 2019. URL: https://github.com/ethereum/wiki/wiki/JSON-RPC (zitiert auf S. 24).

[19c] *solcjs Repository*. 2019. URL: https://github.com/ethereum/solc-js (zitiert auf S. 24).

[Ber17] J. Bergquist. „Blockchain Technology and Smart Contracts: Privacy-preserving Tools“. Uppsala Universitet, 2017 (zitiert auf S. 19).

[But+14] V. Buterin et al. „A next-generation smart contract and decentralized application platform“. In: *white paper* 3 (2014), S. 37 (zitiert auf S. 17, 20–22, 27).

[But17] V. Buterin. *Viper Documentation, Release*. 10. Nov. 2017. URL: https://buildmedia.readthedocs.org/media/pdf/ethereum-viper/latest/ethereum-viper.pdf (zitiert auf S. 21, 23).

[But19] V. Buterin. *Ethereum 2.0*. 22. Sep. 2019. URL: https://github.com/ethereum/eth2.0-specs (zitiert auf S. 30, 47).

[BX09] S. Bhatti, J. Xu. „Survey of target tracking protocols using wireless sensor network“. In: *2009 Fifth International Conference on Wireless and Mobile Communications*. IEEE. 2009, S. 110–115 (zitiert auf S. 26).

[DAK+15a] K. Delmolino, M. Arnett, A. E. Kosba, A. Miller, E. Shi. „Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab“. In: *IACR Cryptology ePrint Archive* 2015 (2015), S. 460 (zitiert auf S. 20, 21).

[DAK+15b] K. Delmolino, M. Arnett, A. Kosba, A. Miller, E. Shi. „A programmer's guide to ethereum and serpent“. In: *URL: https://mc2-umd. github. io/ethereumlab/docs/serpent_tutorial. pdf.(2015).(Accessed May 06, 2016)* (2015) (zitiert auf S. 22, 23).

[Dap19] DappRadar. *DappRadar*. 17. Sep. 2019. URL: https://dappradar.com/rankings/protocol/ethereum (zitiert auf S. 17).

[DMH17] V. Dhillon, D. Metcalf, M. Hooper. „The hyperledger project“. In: *Blockchain enabled applications*. Springer, 2017, S. 139–149 (zitiert auf S. 56).

[Edg17] B. Edgington. *LLL Compiler Documentation*. 16. Sep. 2017. URL: https://buildmedia.readthedocs.org/media/pdf/lll-docs/latest/lll-docs.pdf (zitiert auf S. 21).

[Eth19a] Etherscan. *Ether Historical Prices*. 25. Sep. 2019. URL: https://etherscan.io/chart/etherprice (zitiert auf S. 47).

[Eth19b] Etherscan. *Ethereum Gas Tracker*. 25. Sep. 2019. URL: https://etherscan.io/gastrackere (zitiert auf S. 47).

[GKW+16] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, S. Capkun. „On the security and performance of proof of work blockchains“. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM. 2016, S. 3–16 (zitiert auf S. 19).

[GYL11] R. K. Ganti, F. Ye, H. Lei. „Mobile crowdsensing: current state and future challenges“. In: *IEEE Communications Magazine* 49.11 (2011), S. 32–39 (zitiert auf S. 26).

[Hom16] E. Homestead. *Ethereum Homestead Documentation*. 2016. URL: https://buildmedia.readthedocs.org/media/pdf/ethereum-homestead/latest/ethereum-homestead.pdf (zitiert auf S. 23–25).

[HP13] S. Harkanth, B. Phulpagar. „A survey on clustering methods and algorithms“. In: *International Journal of Computer Science and Information Technologies* 4.5 (2013), S. 687–691 (zitiert auf S. 32).

[IP96] S. Ilic, H. D. Petkovic Miodrag S. „A note on Babylonian square-root algorithm and related variants“. In: *Novi Sad J. Math* 26.1 (1996), S. 155–162 (zitiert auf S. 41).

[Kar18] E. Karataş. „Developing Ethereum Blockchain-Based Document Verification Smart Contract for Moodle Learning Management System“. In: *Bilişim Teknolojileri Dergisi* 11.4 (2018), S. 399–406 (zitiert auf S. 24).

[Kha11] M. Khan. *Longest path in a directed acyclic graph (DAG)*. 10. Apr. 2011. URL: http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/Docs/longest-path-in-dag.pdf (zitiert auf S. 34, 35).

[Kom19] M. Kompf. *Entfernungsberechnung*. 2019. URL: https://www.kompf.de/gps/distcalc.html (zitiert auf S. 31).

[KRDO17] A. Kiayias, A. Russell, B. David, R. Oliynykov. „Ouroboros: A provably secure proof-of-stake blockchain protocol“. In: *Annual International Cryptology Conference*. Springer. 2017, S. 357–388 (zitiert auf S. 21).

[LSN+18] J. Liu, H. Shen, H. S. Narman, W. Chung, Z. Lin. „A survey of mobile crowdsensing techniques: A critical component for the internet of things“. In: *ACM Transactions on Cyber-Physical Systems* 2.3 (2018), S. 18 (zitiert auf S. 17, 26).

[Maz15] D. Mazieres. „The stellar consensus protocol: A federated model for internet-level consensus“. In: *Stellar Development Foundation* (2015), S. 32 (zitiert auf S. 56).

[Nak+08] S. Nakamoto et al. „Bitcoin: A peer-to-peer electronic cash system“. In: (2008) (zitiert auf S. 17, 19, 20).

[Osm12] A. Osmani. *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide*. Ö'Reilly Media, Inc.", 2012 (zitiert auf S. 24).

[Pen18] M. Penke. *Sorry, aber die dApp-Revolution braucht noch ein bisschen*. 30. Aug. 2018. URL: https://www.gruenderszene.de/technologie/dapp-nutzung-kommentar?interstitial_click (zitiert auf S. 17).

[RD18] D. Reinhardt, F. Dürr. „Opportunities and Risks of Delegating Sensing Tasks to the Crowd“. In: *Handbook of Mobile Data Privacy*. Hrsg. von A. Gkoulalas-Divanis, C. Bettini. Cham: Springer International Publishing, 2018, S. 129–165. ISBN: 978-3-319-98161-1. DOI: 10.1007/978-3-319-98161-1_6. URL: https://doi.org/10.1007/978-3-319-98161-1_6 (zitiert auf S. 26).

[Sch16] B. Schuppar. *Geometrie auf der Kugel: alltägliche Phänomene rund um Erde und Himmel*. Springer-Verlag, 2016 (zitiert auf S. 30).

[Spr18] S. Springer. *Node. js: das umfassende Handbuch*. Rheinwerk Verlag, 2018. ISBN: 978-3-836-24003-1. URL: https://www.amazon.com/Node-js/dp/3836240033?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=3836240033 (zitiert auf S. 24).

[Sza97] N. Szabo. „Formalizing and securing relationships on public networks“. In: *First Monday* 2.9 (1997) (zitiert auf S. 20).

[TRO18] TRONfoundation. *TRON, Advanced Decentralized Blockchain Platform, Whitepaper Version: 2.0*. 10. Dez. 2018. URL: https://tron.network/static/doc/white_paper_v_2_0.pdf (zitiert auf S. 56).

[Woo14] G. Wood. „Ethereum: A secure decentralised generalised transaction ledger (yellow paper)“. In: *Internet: https://github. com/ethereum/yellowpaper,[Oct. 30, 2018]* (2014) (zitiert auf S. 22).

[WRB+19] G. Wood, C. Reitwiessner, A. Beregszaszi, L. Husikyan, Y. Hirai et al. *Solidity Documentation, Release 0.5.11*. 12. Aug. 2019. URL: https://buildmedia.readthedocs.org/media/pdf/solidity/v0.5.11/solidity.pdf (zitiert auf S. 21, 23, 24).

[WSR16] E. Wittern, P. Suter, S. Rajagopalan. „A look at the dynamics of the JavaScript package ecosystem“. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2016, S. 351–361 (zitiert auf S. 24).

[WWDR11] H. Weinschrott, J. Weisser, F. Dürr, K. Rothermel. „Participatory sensing algorithms for mobile object discovery in urban areas“. In: *2011 IEEE International Conference on Pervasive Computing and Communications (PerCom)* (2011), S. 128–135 (zitiert auf S. 26).

[XT15] D. Xu, Y. Tian. „A comprehensive survey of clustering algorithms“. In: *Annals of Data Science* 2.2 (2015), S. 165–193 (zitiert auf S. 32).

[yan19] yann300. *Remix Documentation Release 1*. 20. Sep. 2019. URL: https://buildmedia.readthedocs.org/media/pdf/remix-ide/latest/remix-ide.pdf (zitiert auf S. 24).

[ZYS+15] X. Zhang, Z. Yang, W. Sun, Y. Liu, S. Tang, K. Xing, X. Mao. „Incentives for mobile crowd sensing: A survey“. In: *IEEE Communications Surveys & Tutorials* 18.1 (2015), S. 54–67 (zitiert auf S. 17).

Alle URLs wurden zuletzt am 27. 09. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift