
IP6: Blockchain Transaktionsmanager

Bachelorthesis

Faustina Bruno, Serge Jurij Maïkoff

Studiengang:

- iCompetence
- Informatik

Betreuer:

- Markus Knecht
- Daniel Kröni

Experte:

- Konrad Durrer

Auftraggeber:

Fachhochschule Nordwestschweiz
FHNW Campus Brugg-Windisch
Bahnhofstrasse 6
5210 Windisch



University of Applied Sciences and Arts Northwestern Switzerland
School of Engineering

2019-10-01

Abstract

Die Fachhochschule Nordwestschweiz (FHNW) möchte den Studierenden und Mitarbeitenden eine kostenneutrale Blockchain anbieten. Diese soll für innovative Projekte, Experimente und Schulungszwecke genutzt werden.

Eine Blockchain schützt sich vor Denial of Service (DoS) Attacken mit Transaktionsgebühren. Dieser Schutzmechanismus ist für eine Lernumgebung nicht praktikabel. Für die Studierenden und Mitarbeitenden sollen keine Kosten anfallen. Das Ziel dieser Arbeit ist, einer definierten Benutzergruppe gratis Transaktionen zu gewähren, ohne dabei die Sicherheit der Blockchain zu kompromittieren.

Mit einer Literaturrecherche ist das nötige Wissen zu Blockchains, Transaktionen und DoS Attacken erarbeitet worden. Für die Ethereumblockchain ist mit Parity ein Client identifiziert worden, der bereits über die Möglichkeit verfügt, einer Benutzergruppe gratis Transaktionen zu erlauben. Auf dieser Basis, ist ein Kontrollsystem konzipiert worden. Mit diesem Transaktionsmanager kann das Verhalten der Benutzergruppe überwacht werden. Stellt ein Benutzer eine Bedrohung dar, wird ihm das Privileg für gratis Transaktionen temporär entzogen.

Der Transaktionsmanager ist als Java-Applikation implementiert worden. Mit der Bibliothek web3j wird der jeweils neuste Block auf Transaktionen untersucht. Gefundene gratis Transaktionen werden analysiert.

Der Transaktionsmanager verwaltet die Benutzergruppe. Pro Benutzer wird die Anzahl getätigter gratis Transaktionen und die damit verursachten Rechenkosten der Blockchain überwacht. Die FHNW kann für jeden Benutzer individuelle Limiten definieren. Die Dauer der Suspendierung kann ebenfalls individuell festgelegt werden. Um die Datenpersistenz auch bei einem Stopp der Applikation zu gewährleisten, wird der Zustand regelmässig als Datei gespeichert. Bei einem Neustart wird dieser gelesen und wiederhergestellt. Für die individuelle Suspendierung wird eine Priority-Queue in Kombination mit einem Command Pattern verwendet. So kann auch eine grosse Anzahl Benutzer, mit hoher Ressourceneffizienz, bearbeitet werden. Für die Entwicklung und Testung ist eine private single Node Blockchain verwendet worden.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Problemstellung und Ziel	1
1.2	Blockchain-Transaktionsmanager	1
1.3	Resultat	3
1.4	Strukturierung des Berichts	3
2	Theoretische Grundlagen	5
2.1	Anwendungsbereich	5
2.2	Komponenten	5
2.2.1	Ethereum Blockchain	6
2.2.2	Smart Contracts	6
2.2.3	Transaktionen	7
2.2.4	Gas	8
2.2.5	Account	9
2.2.6	Blockchain Wallet	10
2.2.7	Denial of Service (DoS) Attacken	11
2.3	Ethereum Client	12
2.3.1	Parity	12
2.3.2	Geprüfte Alternativen	17
2.4	Lösungsansätze	17
2.4.1	Architektur	18
2.4.2	Evaluation der Architektur	27
2.4.3	DoS-Algorithmus	29
2.4.4	Evaluation DoS-Algorithmus	32
2.4.5	Konfiguration des Algorithmus	34
3	Praktischer Teil	36
3.1	Parity	37
3.1.1	Konfiguration der Blockchain	37
3.1.2	Name Registry	41
3.1.3	Certifier	41

3.1.4	Interaktionsübersicht auf einem Parity Node	41
3.2	Transaktionsmanager	45
3.2.1	Wrapperklassen	45
3.2.2	Überwachung von Transaktionen	45
3.2.3	Command Pattern und Priority Queue	47
3.2.4	Persistenz	48
3.2.5	Konfiguration	51
3.2.6	Initialisierung	52
3.2.7	Run	53
3.2.8	Tests	54
4	Fazit	57
5	Quellenverzeichnis	60
6	Anhang	64
6.1	Glossar	64
6.2	Weitere Lösungsansätze	67
6.2.1	Super Smart Wallet	67
6.3	Abnahmekriterien	69
6.4	Report der Abnahmetests	71
6.4.1	Abnahmetest 1	71
6.4.2	Abnahmetest 2	72
6.4.3	Abnahmetest 3	73
6.4.4	Abnahmetest 4	73
6.4.5	Abnahmetest 5	75
6.4.6	Abnahmetest 6	76
6.4.7	Abnahmetest 7	77
6.4.8	Abnahmetest 8	78
6.4.9	Abnahmetest 9	79
6.4.10	Abnahmetest 10	80
6.4.11	Abnahmetest 11	81
6.4.12	Abnahmetest 12	82
6.4.13	Abnahmetest 13	83
6.4.14	Abnahmetest 14	84
6.4.15	Abnahmetest 15	86
6.5	Betriebsanleitung	87
6.5.1	Hochfahren der Parity Instanz	87
6.5.2	Deployment	87

6.5.3	Konfigurieren des Transaktionsmanagers	88
6.5.4	Starten der Blockchain Überwachung mit dem Blockchain Transaktionsmanager	88
6.6	Verlinkung von Code	90
6.6.1	Name Registry und Certifier	90
6.6.2	Transaktionsmanager	90
6.6.3	Dateien für Parity	90
7	Ehrlichkeitserklärung	91

1 Einleitung

Dieses Kapitel liefert eine ausführliche Zusammenfassung der Bachelorthesis. Weiter ist eine Übersicht über die Strukturierung des Berichts gegeben.

1.1 Problemstellung und Ziel

Die Fachhochschule Nordwest Schweiz[1] (FHNW) möchte für die Studierenden und Mitarbeiter eine Blockchain zur Verfügung stellen. Ihnen sollen so eine Möglichkeit geboten werden, erste Erfahrungen und Wissen im Umgang mit einer Blockchain zu gewinnen.

Der beste Schutz einer Blockchain gegen eine DoS Attacke, sind die anfallenden Kosten für den Angreifer. Auf jede Transaktion wird eine Gebühr erhoben. Bei Ethereum werden zusätzlich anfallender Aufwand auf der Blockchain berücksichtigt. Die benötigte Rechenleistung wird mit Gas beziffert. Je komplexer der Task, desto mehr Gas wird benötigt. Es existiert ein Maximum an Gas, dass durch eine Transaktion verbraucht werden kann. Wird dieses Maximum erreicht bevor die Berechnungen fertig sind, werden diese Abgebrochen. Das ist ebenfalls ein Schutzmechanismus gegen DoS Attacken. Ohne diesen Schutz, könnten zum Beispiel Schleifen ohne Abbruchbedingung für einen Angriff verwendet werden.

Um die Kosten einer Transaktion zu berechnen, wird das verbrauchte Gas mit dem Gas Preis[2] multipliziert. Diese Kosten werden vom Sender einer Transaktion an den Node gezahlt, der die Transaktion verarbeitet hat.

Für eine Lernumgebung sind anfallende Kosten nicht praktikabel. Daher soll ein Weg gefunden werden, um einer definierten Benutzergruppe gratis Transaktionen zu ermöglichen, ohne die Blockchain anfällig gegen DoS Attacken zu machen.

1.2 Blockchain-Transaktionsmanager

Es wurde eine Ethereumblockchain[3][4] aufgesetzt, die es einer spezifischen Benutzergruppe erlaubt, gratis Transaktionen zu tätigen. Diese Transaktionen werden durch den Blockchaintransaktionsmanager überwacht. Der entwickelte Transaktionsmanager stellt sicher, dass gratis Transaktionen nicht für eine Denial of Service (DoS) Attacke[5] genutzt werden können.

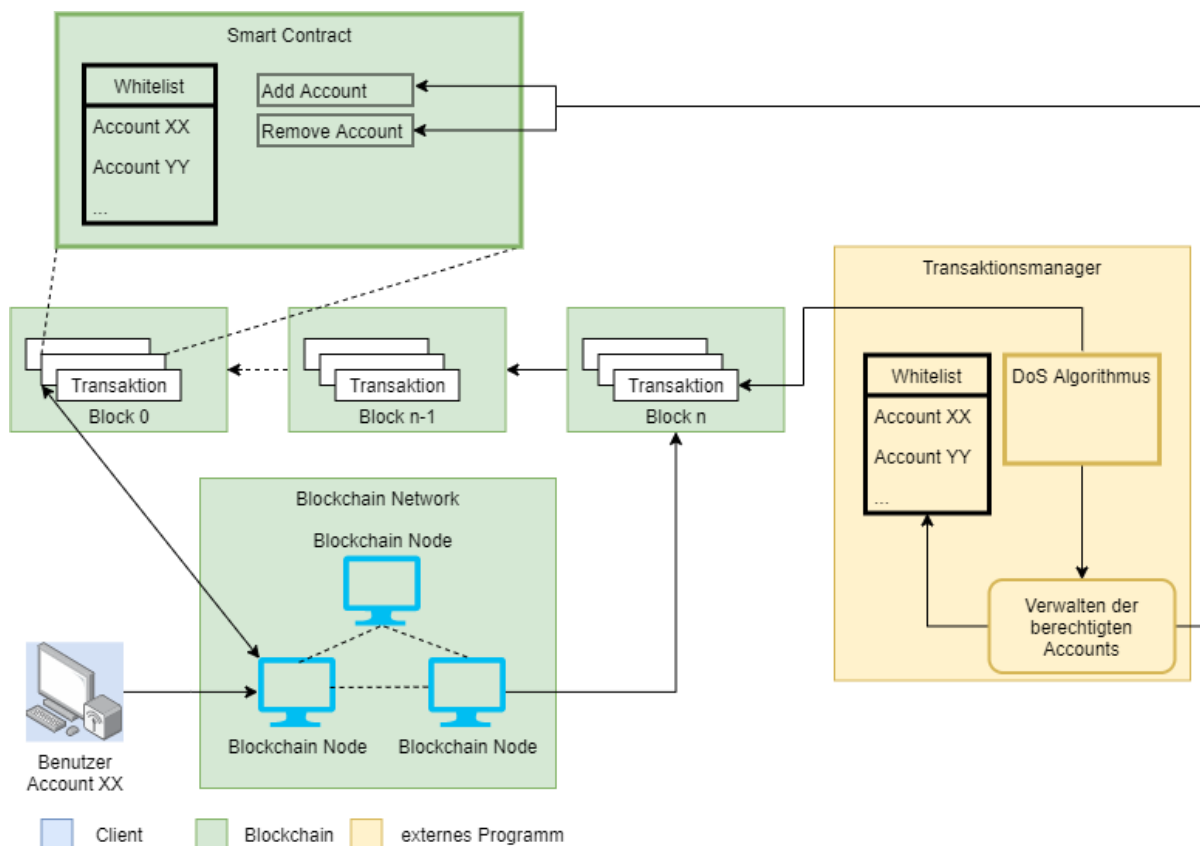


Abbildung 1.1: Interaktion von Benutzer, Blockchain und Transaktionsmanager

Auf der Grafik 1.1 sind die Interaktionen zwischen einem Benutzer, dem Blockchainnetzwerk und dem Transaktionsmanager ersichtlich.

Ein Benutzer sendet mit seinem Account eine gratis Transaktion an einen Node der Blockchain. Dieser prüft, ob der verwendete Account für gratis Transaktionen berechtigt ist. Dafür wird eine Whitelist[6] verwendet. Diese befindet sich auf der Blockchain. Ist diese Prüfung erfolgreich, wird die Transaktion in den nächsten Block aufgenommen.

Der Transaktionsmanager registriert, dass eine neue gratis Transaktion in die Blockchain aufgenommen worden ist. Der Sender der Transaktion wird ermittelt. Mit dem DoS Algorithmus wird geprüft, ob der verwendete Account mit dieser Transaktion gegen eine Richtlinie verstößt. Sollte diese Prüfung ergeben, dass ein Verstoß vorliegt, wird der Account von der Whitelist in der Blockchain gelöscht. Im Transaktionsmanager wird vermerkt, dass sich dieser Account momentan nicht mehr auf der Whitelist befindet. Das hat zur Folge, dass mit diesem Account keine gratis Transaktionen mehr getätigt werden können.

Der Transaktionsmanager kann den Account nach gewissen Regeln und in Abhängigkeit der Zeit, wieder auf die Whitelist setzen. Der Benutzer kann den Account somit wieder für gratis Transaktionen

verwenden.

Der Transaktionsmanager kann durch die Betreiber konfiguriert werden. Die Parameter für den DoS Algorithmus sind pro Account individuell definierbar.

Wird die Whitelist im Transaktionsmanager mit weiteren Accounts ergänzt, werden diese an das Blockchainnetzwerk übermittelt und dort ebenfalls in die Whitelist aufgenommen. Accounts können auch permanent von gratis Transaktionen ausgeschlossen werden.

1.3 Resultat

Parity ist zur Zeit der einzige Ethereumclient, der über ein Benutzermanagement für gratis Transaktionen verfügt. Durch die Verwendung von Smart Contracts[7] ist eine Whitelist für gratis Transaktionen möglich. Da die Whitelist auf der Blockchain gespeichert ist, kann sie von allen Nodes verwendet werden.

Sobald auf einem Node eine gratis Transaktion eingeht, wird geprüft, ob sich der verwendete Account auf der Whitelist befindet. Nur dann wird die Transaktion vom Node angenommen und weiterverarbeitet.

Der Blockchaintransaktionsmanager ist als Java-Applikation realisiert worden. Diese interagiert mit der Whitelist von Parity. Die Interaktion mit dem Betreiber des Transaktionsmanagers findet über die Kommandozeile und Konfigurationsdateien statt.

Es wird der aktuelle Block der Blockchain aufgerufen und auf Transaktionen untersucht. Bei gefundenen gratis Transaktionen wird das Verhalten des Senderaccounts evaluiert. Anhand der Anzahl getätigten gratis Transaktionen und dem dabei verbrauchtem Gas, wird bestimmt, ob die Transaktion Teil einer DoS Attacke ist. Fällt diese Prüfung positiv aus, wird der Account für eine bestimmte Dauer von der Whitelist entfernt. Das bedeutet, dass er keine gratis Transaktionen mehr tätigen kann. Reguläre, also kostenpflichtige Transaktionen, sind weiterhin möglich.

Die verfügbaren gratis Transaktionen und Gas, sowie die Dauer einer Suspendierung von der Whitelist können konfiguriert werden. Der Betreiber hat die Möglichkeit, diese Parameter für jeden Account individuell zu definieren.

Um eine individuelle Suspendierung von der Whitelist zu ermöglichen, wird eine Priority-Queue und ein Command-Pattern verwendet.

1.4 Strukturierung des Berichts

Der Bericht ist in einen theoretischen und praktischen Teil gegliedert. Geprüfte Tools, der aktuelle Stand der Ethereumblockchain, sowie die konzipierten Lösungsansätze und deren Evaluation werden im theoretischen Teil behandelt.

Im praktischen Teil wird beschrieben, wie das gewonnene Wissen umgesetzt wird. Es wird auf die implementierte Lösung und deren Vor- und Nachteile eingegangen. Geprüfte Alternativen und deren Argumente sind ebenfalls enthalten.

Das Fazit bildet den Abschluss des eigentlichen Berichts. Im Anhang ist eine Beschreibung der Entwicklungsumgebung, die Installationsanleitung und Links zum verwendeten Code zu finden.

2 Theoretische Grundlagen

Dieses Kapitel befasst sich nebst dem Kontext der Arbeit mit den erarbeiteten Lösungsansätzen. Weiter wird der Anwendungsbereich der Lösung behandelt.

2.1 Anwendungsbereich

Die FHNW möchte zu Ausbildungszwecken eine eigene Ethereumblockchain betreiben. Die Blockchain soll dieselbe Funktionalität wie die öffentliche Ethereumblockchain vorweisen. Sie soll den Studenten und Mitarbeitenden die Möglichkeit bieten, in einer sicheren Umgebung Erfahrungen zu sammeln und Wissen zu gewinnen.

Obwohl eine öffentliche Blockchain für jedermann frei zugänglich ist, ist jede Transaktion mit Kosten verbunden. Die Kosten sind ein fixer Bestandteil einer Blockchain. Diese ermöglichen nicht nur deren Verarbeitung, sondern garantieren auch Schutz vor Attacken.

Im Gegensatz zu einer öffentlichen Blockchain, sind Transaktionsgebühren in einer Lernumgebung nicht praktikabel. Die Benutzer sollen gratis mit der Blockchain agieren können, ohne dass der Betrieb oder die Sicherheit der Blockchain kompromittiert werden.

Die FHNW bietet die kostenlose Verarbeitung von Transaktionen zu Verfügung. Damit sichert sie den Betrieb der Blockchain. Die Implementation von gratis Transaktionen für eine definierte Benutzergruppe und einem Schutzmechanismus wird in diesem Bericht behandelt.

2.2 Komponenten

Die folgenden Abschnitte behandeln das Resultat der gemachten Literaturrecherchen. Für jedes Thema sind die gewonnen Erkenntnisse aufgeführt. Dabei ist nebst einem grundsätzlichen Verständnis für die Materie immer der Schutz vor einer DoS Attacke im Fokus.

2.2.1 Ethereum Blockchain

Eine Blockchain ist eine kontinuierlich erweiterbare Liste von Datensätzen, „Blöcke“ genannt, die mittels kryptographischer Verfahren miteinander verkettet sind. Jeder Block enthält dabei typischerweise einen kryptographisch sicheren Hash (Streuwert) des vorhergehenden Blocks, einen Zeitstempel und Transaktionsdaten[4].

Ein speziell erwähnenswerter Block, ist der sogenannte Genesisblock[8]. Dieser ist der erste Block in einer Blockchain. Der Genesisblock beinhaltet alle nötigen Parametern und Einstellungen, um eine Blockchain zu starten.

Blockchains sind auf einem peer-to-peer (P2P) Netzwerk[9] aufgebaut. Ein Computer der Teil von diesem Netzwerk ist, wird Node genannt. Jeder Node hat eine identische Kopie der Historie aller Transaktionen.

Es gibt keinen zentralen Server der angegriffen werden kann. Das erhöht die Sicherheit der Blockchain. Es muss davon ausgegangen werden, dass es Nodes gibt, die versuchen die Daten der Blockchain zu verfälschen. Dem wird mit der Verwendung von diversen Consensus Algorithmen[10] entgegen gewirkt. Die Consensus Algorithmen stellen sicher, dass die Transaktionen auf der Blockchain valide und authentisch sind.

Im Gegensatz zu Bitcoin[11] kann bei Ethereum[3] auch Code in der Chain gespeichert werden, sogenannte Smart Contracts, siehe 2.2.2.

Ethereum verfügt über eine eigene Kryptowährung, den Ether (ETH).

2.2.2 Smart Contracts

Der Begriff Smart Contract, wurde von Nick Szabo[12] in den frühen 1990 Jahren zum ersten Mal verwendet. Es handelt sich um ein Stück Code, das auf der Blockchain liegt. Es können Vertragsbedingungen als Code geschrieben werden. Sobald die letzte Transaktion eintrifft und alle Bedingungen erfüllt sind, führt sich der Smart Contract selbst aus.

Der Code kann von allen Teilnehmern der Blockchain inspiziert werden. Da er dezentral auf der Blockchain gespeichert ist, kann er auch nicht nachträglich manipuliert werden. Das schafft Sicherheit für die beteiligten Parteien.

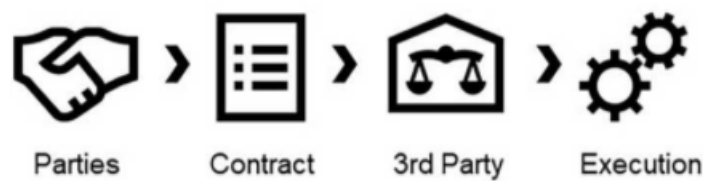


Abbildung 2.1: Ein traditioneller Vertrag[13]

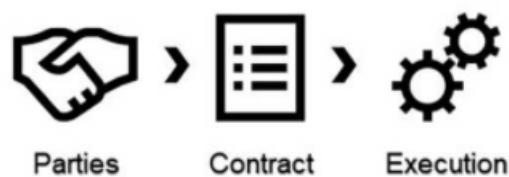


Abbildung 2.2: Ein Smart Contract[13]

Der grosse Vorteil von Smart Contracts ist, dass keine Drittparteien benötigt werden. Das ist auf den Bildern 2.1 und 2.2 dargestellt. Der Code kontrolliert die Transaktionen, welche nachverfolgbar und irreversibel sind. Bei einem traditionellen Vertrag werden diese durch Drittparteien kontrolliert und meistens auch ausgeführt.

Sobald ein Smart Contract auf Ethereum deployed ist, verfügt er über eine Adresse, siehe Abschnitt 2.2.5.1. Mit dieser, kann auf die Funktionen des Smart Contracts zugegriffen werden.

2.2.2.1 Decentralized application (DApp)

Eine DApp ist eine Applikation (App), deren backend Code dezentral auf einem peer-to-peer Netzwerk läuft, zum Beispiel die Ethereum Blockchain. Der frontend Code kann in einer beliebigen Sprache geschrieben werden, sofern Aufrufe an das Backend möglich sind.

DApp's für die Ethereum Blockchain werden mit Smart Contracts realisiert. Ein Beispiel ist der Ethereum Name Server (ENS)[14]. Analog zu DNS[15], können Adressen leserliche Namen zugeordnet werden.

2.2.3 Transaktionen

Um mit der Blockchain zu interagieren, werden Transaktionen verwendet. Sie erlauben es, Daten in der Blockchain zu erstellen oder anzupassen. Eine Transaktion verfügt über folgende Felder:

From Der Sender der Transaktion. Wird mit einer 20 Byte langen Adresse, siehe Abschnitt 2.2.5.1, dargestellt.

To Der Empfänger der Transaktion. Wird ebenfalls mit einer 20 Byte langen Adresse dargestellt. Falls es sich um ein Deployment von einem Smart Contract handelt, wird dieses Feld leer gelassen.

Value Mit diesem Feld wird angegeben, wieviel Wei[16] übertragen werden soll. Der Betrag wird von „From“ nach „To“ übertragen.

Data/Input Dieses Feld wird hauptsächlich für die Interaktion mit Smart Contracts, siehe Abschnitt 2.2.2, verwendet. Wenn ein Smart Contract deployed werden soll, wird in diesem Feld dessen Bytecode[17] übertragen. Bei Funktionsaufrufen auf einen Smart Contract wird die Funktionssignatur und die codierten Parameter mitgegeben. Bei reinen Kontoübertragungen wird das Feld leer gelassen.

Gas Price Gibt an, welcher Preis pro Einheit Gas man gewillt ist zu zahlen. Mehr dazu im Abschnitt 2.2.4

Gas Limit Definiert die maximale Anzahl Gas Einheiten, die für diese Transaktion verwendet werden können, siehe Abschnitt 2.2.4 [18]

Damit eine Transaktion in die Blockchain aufgenommen werden kann, muss sie signiert[19] sein. Dies kann beim Benutzer offline gemacht werden. Die signierte Transaktion wird dann an die Blockchain übermittelt.

Die Übermittlung der Transaktionen wird mittels Remote procedure call(RPC)[20] gemacht.

2.2.4 Gas

Mit Gas[2] ist in der Ethereum Blockchain eine Quantifizierung von Aufwand gemeint. Damit werden Transaktionskosten gezahlt. Jede Aktion in der Blockchain kostet eine bestimmte Menge an Gas (Gas Cost). Somit ist die benötigte Menge an Gas proportional zur benötigten Rechenleistung. So wird sichergestellt, dass die anfallenden Kosten einer Interaktion gerecht verrechnet werden. Die anfallenden Gas Kosten werden in Ether gezahlt. Für die Berechnung der Transaktionskosten wird der Preis pro Einheit Gas (Gas Price) verwendet. Dieser kann vom Sender selbst bestimmt werden. Ein zu tief gewählter Gas Price hat zur Folge, dass die Transaktion nicht in die Blockchain aufgenommen wird, da es sich für einen Miner, siehe Abschnitt 3.1.1.1, nicht lohnt, diese zu verarbeiten. Ein hoher Gas Price stellt zwar sicher, dass die Transaktion schnell verarbeitet wird, kann aber hohe Gebühren generieren.

$$TxPrice = gasCost * gasPrice$$

Die Transaktionskosten werden nicht direkt in Ether berechnet, da dieser starken Kursschwankungen unterworfen sein kann. Die Kosten für Rechenleistung, also Elektrizität, sind hingegen stabiler Natur. Daher sind Gas und Ether separiert.

Ein weiterer Parameter ist Gas Limit. Mit diesem Parameter wird bestimmt, was die maximale Gas Cost ist, die man für eine Transaktion bereitstellen möchte. Es wird aber nur so viel verrechnet, wie auch wirklich benötigt wird, der Rest wird dem Sender wieder gutgeschrieben. Falls die Transaktionskosten höher als das gesetzte Gas Limit ausfallen, wird die Ausführung der Transaktion abgebrochen. Alle gemachten Änderungen auf der Chain werden rückgängig gemacht. Die Transaktion wird als „fehlgeschlagene Transaktion“ in die Blockchain aufgenommen. Das Gas wird nicht zurückerstattet, da die Miner bereits Rechenleistung erbracht haben.

2.2.5 Account

Um mit Ethereum interagieren zu können, wird ein Account benötigt. Es gibt zwei Arten von Accounts, solche von Benutzern und jene von Smart Contracts. Ein Account ermöglicht es einem Benutzer oder Smart Contract, Transaktionen zu empfangen und zu senden.

2.2.5.1 Benutzer Account

Der Account eines Benutzers besteht aus Adresse, öffentlichen und geheimen Schlüssel. Diese Art von Accounts haben keine Assoziation mit Code. Sie werden von Benutzer verwendet um mit der Blockchain zu interagieren.

Geheimer Schlüssel Der geheime Schlüssel ist ein 256 Bit lange zufällig generierte Zahl. Er definiert einen Account und wird verwendet, um Transaktionen zu signieren. Daher ist es von grösster Wichtigkeit, dass ein geheimer Schlüssel sicher gelagert wird. Wenn er verloren geht, gibt es keine Möglichkeit mehr auf diesen Account zuzugreifen.

Öffentlicher Schlüssel Der öffentliche Schlüssel wird aus dem geheimen Schlüssel abgeleitet. Für die Generierung wird Keccak[21] verwendet, ein „Elliptical Curve Digital Signature Algorithm“[22]. Der öffentliche Schlüssel wird verwendet um die Signatur einer Transaktion zu verifizieren.

Adresse Die Adresse wird aus dem öffentlichen Schlüssel abgeleitet. Es wird SHA3[23] verwendet, um einen 32 Byte langen String zu bilden. Von diesem bilden die letzten 20 Bytes, also 40 Zeichen, die Adresse eines Account. Die Adresse wird bei Transaktionen oder Interaktionen mit einem Smart Contract verwendet.

2.2.5.2 Contract Accounts

Contract Accounts sind durch ihren Code definiert. Sie können keine Transaktionen initiieren, sondern reagieren nur auf zuvor eingegangene. Das wird auf der Abbildung 2.3 dargestellt. Ein Benutzer Accounts wird als „Externally owned account“ bezeichnet.

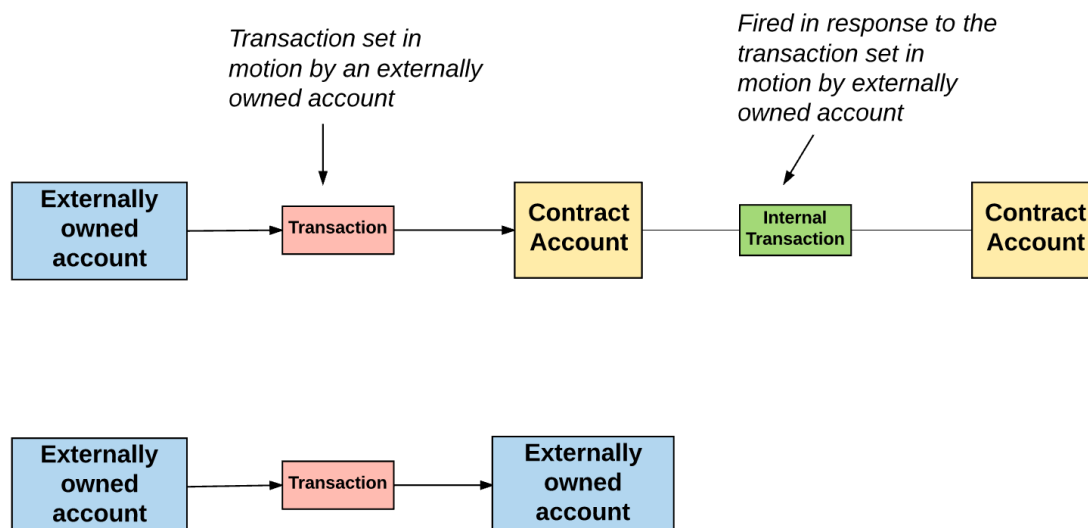


Abbildung 2.3: Transaktionen zwischen verschiedenen Accounts[24]

Im Gegensatz zu einem Benutzer Account hat ein Contract Account keine Verwendung für einen geheimen oder öffentlichen Schlüssel. Es wird nur eine Adresse benötigt. Analog zu einem Benutzer Account, wird diese benötigt, um Transaktionen an diesen Smart Contract zu senden.

Sobald ein Smart Contract deployed wird, wird eine Adresse generiert. Verwendet wird die Adresse und Anzahl getätigte Transaktionen (nonce[25]) des Benutzer Accounts, der das Deployment vornimmt.[26]

2.2.6 Blockchain Wallet

Eine Blockchain Wallet, kurz Wallet, ist ein digitales Portemonnaie. Der Benutzer hinterlegt in der Wallet seinen geheimen Schlüssel, siehe 2.2.5.1. Dadurch erhält er eine grafische Oberfläche für die Verwaltung seines Accounts. Nebst dem aktuellen Kontostand, wird meistens noch die Transaktionshistorie angezeigt.

In der Wallet können mehrere Accounts verwaltet werden. So muss sich der Benutzer nicht selbst um die sichere Aufbewahrung der geheimen Schlüssel kümmern. Bei den meisten Wallets ist es möglich

verschiedene Währungen zu verwalten.

Es existieren zwei unterschiedliche Arten von Wallets, Hot und Cold Wallets:

Hot Wallet Ein Stück Software, welches die geheimen Schlüssel verwaltet. : Es existieren drei unterschiedliche Typen, Desktop, Web und Mobile Wallets. [27], [28], [29]

Cold Wallet Der geheime Schlüssel wird in einem Stück Hardware gespeichert. Dadurch können die geheimen Schlüssel offline gelagert werden. Das erhöht die Sicherheit der Wallet, da Angriffe aus dem Internet ausgeschlossen werden können. [27], [28], [29]

2.2.6.1 Smart Wallet

Smart Wallets basieren auf Smart Contracts. Der Benutzer ist der Besitzer des Smart Contracts und somit der Wallet. Die Verwendung von Smart Contracts bei der Implementierung der Wallet ermöglicht mehr Benutzerfreundlichkeit ohne die Sicherheit zu kompromittieren. [30], [31], [32]

2.2.7 Denial of Service (DoS) Attacken

Bei einer DoS Attacke versucht der Angreifer einen Service zu überlasten. Die Überlastung schränkt die Verfügbarkeit stark ein oder macht den Service gänzlich unverfügbar für legitime Anfragen.

Zurzeit sind Blockchains noch relativ langsam bei der Verarbeitung von Transaktionen. Ethereum kann ungefähr 15 Transaktionen pro Sekunde abarbeiten[33]. Dadurch ist ein möglicher Angriffsvektor, die Blockchain mit einer sehr hohen Zahl Transaktionen zu fluten.

Ein anderer Angriffsvektor, sind Transaktionen mit einem sehr hohen Bedarf an Rechenleistung. Hier wird Code auf der Blockchain aufgerufen, dessen Verarbeitung sehr lange dauert.

Beide Angriffe haben zur Folge, dass Benutzer sehr lange auf die Ausführung ihrer Transaktionen warten müssen.

Blockchains schützen sich vor diesem Angriff mit einer Transaktionsgebühr. Diese werden durch Angebot und Nachfrage bestimmt. Das heisst, wenn es viele Transaktionen gibt, steigt der Bedarf an deren Verarbeitung und es kann davon ausgegangen werden, dass auch die Transaktionsgebühren steigen. Das bedeutet, dass bei einer DoS Attacke die Transaktionsgebühren tendenziell steigen. Um sicherzustellen, dass seine Transaktionen weiterhin zuverlässig in die Blockchain aufgenommen werden, muss der Angreifer seinen Gas Price kontinuierlich erhöhen.

Ein DoS Angriff auf eine Blockchain wird dadurch zu einem sehr kostspieligen Unterfangen. Die hohen Kosten schrecken die meisten Angreifer ab und sind somit ein sehr effizienter Schutzmechanismus.[34]

2.2.7.1 DoS Attacke an der FHNW

Auf der Blockchain der FHNW existiert eine privilegierte Benutzergruppe. Diese dürfen gratis Transaktionen ausführen. Diese Gruppe von Benutzer ist eine potenzielle Bedrohung. Ohne Transaktionskosten ist die Blockchain anfällig für eine DoS Attacke.

Aus diesem Grund muss das Verhalten der privilegierten Accounts überwacht werden. Falls einer dieser Accounts eine DoS Attacke einleitet, muss das frühestmöglich erkannt und unterbunden werden können.

2.3 Ethereum Client

Für die Betreuung von einem Ethereum Node ist ein Client nötig. Dieser muss das Ethereum Protokoll[35] implementieren. Das Protokoll definiert die minimalen Anforderungen an den Client. Das erlaubt, dass der Client in verschiedenen Sprachen und von verschiedenen Teams realisiert werden kann. Nebst der verwendeten Programmiersprache unterscheiden sich die Clients bei implementierten Zusatzfunktionen, die im Protokoll nicht spezifiziert sind. Die populärsten Clients sind Go Ethereum (GETH)[36], Parity[37], Aleth[38] und Trinity[39]. Die Clients wurden auf die Zusatzfunktionalität untersucht, für eine definierte Gruppe von Accounts, gratis Transaktionen zu ermöglichen.

2.3.1 Parity

Geschrieben in Rust[40], ist es der zweit populärste Client nach Geth[36]. Verfügbar ist Parity für Windows, macOS und Linux. Die Entwicklung ist noch nicht abgeschlossen und es wird regelmässig eine neue Version vorgestellt.

Konfiguriert wird das Programm mittels Konfigurationsdateien. Interaktion zur Laufzeit ist über die Kommandozeile möglich.

Parity ist der einzige Client, der es erlaubt, einer definierten Gruppe von Benutzern gratis Transaktionen zu erlauben.

Die Verwaltung der privilegierten Accounts geschieht mittels eines Smart Contracts. Die Accounts sind in einer Liste, der sogenannten Whitelist, gespeichert.

Für die Verwendung der Whitelist sind zwei Smart Contracts nötig, die Name Registry[41] und der Service Transaction Checker[42]. Diese sind in den folgenden Abschnitten erklärt.

2.3.1.1 Name Registry

In Parity wird die Name Registry verwendet, um eine Accountadresse in eine lesbare Form zu übersetzen. Smart Contracts können für eine Gebühr von einem Ether registriert werden. Dabei wird die Adresse des Smart Contracts zusammen mit dem gewählten Namen registriert. Das erlaubt das Referenzieren von Smart Contracts, ohne dass hart kodierte Adressen verwendet werden müssen. Dieses System ist analog zu einem DNS Lookup[15].

Um eine Whitelist verwenden zu können, muss der zuständige Smart Contract, siehe 2.3.1.2, bei der Name Registry registriert werden.

Nachfolgenden sind die involvierten Methoden und Modifier[43] der Name Registry aufgeführt und erklärt. Der vollständige Code ist im Anhang unter 6.6.1 verlinkt.

Listing 2.1: Eintrag bei der Name Registry

```
1 struct Entry {
2     address owner;
3     address reverse;
4     bool deleted;
5     mapping (string => bytes32) data;
6 }
7 mapping (bytes32 => Entry) entries;
```

Im Codeausschnitt 2.1 ist die Map `entries` aufgeführt. Sie enthält alle registrierten Accounts in Form eines `Entry`. Pro Eintrag wird der Besitzer (`owner`), die Adresse (`address`), ein Flag ob der Eintrag gelöscht ist (`deleted`) und dessen Daten (`data`) gespeichert.

Die Map `entries` ist die zentrale Datenstruktur der Name Registry. Änderungen daran sind daher durch Modifiers eingeschränkt.

Listing 2.2: Modifier whenUnreserved

```
1 modifier whenUnreserved(bytes32 _name) {
2     require(!entries[_name].deleted && entries[_name].owner == 0);
3     _;
4 }
```

Der unter 2.2 gezeigte Code, stellt sicher, dass ein Eintrag zu einem Namen (`_name`) nicht bereits existiert oder zu einem früheren Zeitpunkt gelöscht worden ist. Es wird also geprüft, ob die gewünschte Position in der Map `entries` noch frei ist und somit reserviert werden kann.

Listing 2.3: Modifier onlyOwnerOf

```
1 modifier onlyOwnerOf(bytes32 _name) {
2     require(entries[_name].owner == msg.sender);
3     _;
```

```
4 }
```

Unter 2.3 wird der Sender einer Nachricht mit dem Besitzer eines Eintrags unter dem Namen `_name` in `entries` verglichen. Nur wenn dieser identisch ist, dürfen Änderungen an einem existierenden Eintrag vorgenommen werden.

Listing 2.4: Modifier `whenEntryRaw`

```
1 modifier whenEntryRaw(bytes32 _name) {
2     require(
3         !entries[_name].deleted &&
4         entries[_name].owner != address(0)
5     );
6     _;
7 }
```

Der unter 2.4 aufgeführte Modifier prüft, ob der Eintrag für den Namen `_name` nicht gelöscht ist und über einen gültigen Besitzer verfügt. Mit `!= address(0)` wird der geprüft ob es sich um mehr als einen uninitialisierten Account handelt.

Listing 2.5: Gebühr (Fee) und Modifier `whenFeePaid`

```
1 uint public fee = 1 ether;
2
3 modifier whenFeePaid {
4     require(msg.value >= fee);
5     _;
6 }
```

Im Codeausschnitt 2.5 ist auf Zeile 1 die Höhe der Gebühr (`fee`) definiert. Ab Zeile 3 folgt ein Modifier. Dieser überprüft, ob der Betrag in der Transaktion gross genug ist, um die Gebühr von Zeile 1 zu bezahlen.

Listing 2.6: Methode `Reserve`

```
1 function reserve(bytes32 _name)
2     external
3     payable
4     whenUnreserved(_name)
5     whenFeePaid
6     returns (bool success)
7 {
8     entries[_name].owner = msg.sender;
9     emit Reserved(_name, msg.sender);
10    return true;
11 }
```

Unter 2.6 wird gezeigt, wie mit der Methode `reserve` ein Eintrag in der Liste `entries` für den Namen

`_name` reserviert wird. Durch die Verwendung von `external` auf Zeile 2, kann die Methode von anderen Accounts aufgerufen werden.

Der Modifier `payable` erlaubt es, Ether an die Methode zu senden. Auf Zeile 4 wird überprüft, ob der Eintrag in `entries` noch frei ist. Schliesslich wird geprüft ob der Transaktion genügend Ether mitgegeben wird, um die Gebühr zu begleichen.

Wenn alle Prüfungen erfolgreich sind, wird in `entries` ein neuer Eintrag erstellt. Als Besitzer des Eintrags wird der Sender der Transaktion gesetzt. Auf Zeile 9 wird die erfolgreiche Reservierung in den Log geschrieben.

Listing 2.7: Methode `setAddress`

```
1 function setAddress(bytes32 _name, string _key, address _value)
2     external
3     whenEntryRaw(_name)
4     onlyOwnerOf(_name)
5     returns (bool success)
6 {
7     entries[_name].data[_key] = bytes32(_value);
8     emit DataChanged(_name, _key, _key);
9     return true;
10 }
```

Mit Methode `setAddress` unter 2.7, wird ein reservierter Eintrag in `entries` befüllt. Als erster Parameter wird der Name des Eintrags (`_name`) übergeben. Dieser muss identisch zum verwendeten Namen in der Methode `reserve` sein. Mit dem Parameter `_key` wird der Zugriff auf die innere Map `data` verwaltet. Mit `_value` wird die zu registrierende Adresse übergeben.

Auch diese Methode muss von aussen aufgerufen werden können, daher `external` auf Zeile 2. Wenn die Bedingungen von `whenEntryRaw` und `onlyOwnerOf` auf Zeile 3 und 4 erfüllt sind, wird die eigentliche Registrierung vorgenommen.

In der Map `data` wird die Adresse (`_value`) an der Position `_key` gespeichert.

Die Änderung der Daten wird auf Zeile 9 ins Log geschrieben.

2.3.1.2 Certifier

Als Standard werden alle Transaktionen mit einem Gas Price von 0 verworfen. Das heisst, diese Transaktionen werden bereits beim Node zurückgewiesen und erreichen nie die Blockchain. Dieses Verhalten kann mit der Registrierung des Certifiers bei der Name Registry, geändert werden. Beim Start von Parity wird geprüft, ob der Eintrag in `entries` vorhanden ist. Sofern vorhanden, werden Transaktionen mit einem Gas Price von 0 nicht mehr direkt abgewiesen, sondern es wird geprüft ob der Sender zertifiziert ist. Transaktionen von zertifizierten Accounts werden selbst mit einem Gas Price von 0 in die Blockchain aufgenommen. Gratis Transaktionen von unzertifizierten Accounts werden weiterhin abgewiesen.

In diesem Abschnitt sind besonders wichtige Abschnitte des Certifiers aufgeführt und erklärt. Der gesamte Code ist im Anhang unter 6.6.1 verlinkt.

Listing 2.8: Die Whitelist im Certifier

```
1 struct Certification {
2     bool active;
3 }
4 mapping (address => Certification) certs;
```

Das `mapping` unter 2.8 ist die zentrale Datenstruktur des Certifiers, die Whitelist. In der Liste `certs` sind zertifizierte Accounts gespeichert.

Listing 2.9: Modifier `onlyDelegate`

```
1 address public delegate = msg.sender;
2
3 modifier onlyDelegate {
4     require(msg.sender == delegate);
5     _;
6 }
```

Auf Zeile 1 des Listings 2.9, wird der Besitzer (`msg.sender`) des Smart Contracts gespeichert und der Variabel `delegate` zugewiesen. Mit dem Modifier wird geprüft ob es sich beim Sender der aktuellen Anfrage um den Besitzer des Smart Contracts handelt.

Listing 2.10: Methode `certify`

```
1 function certify(address _who)
2     external
3     onlyDelegate
4 {
5     certs[_who].active = true;
6     emit Confirmed(_who);
7 }
```

Mit der unter 2.10 aufgeführten Methode, wird ein Account registriert und dadurch zertifiziert. Als Parameter wird die zu registrierende Adresse (`_who`) angegeben. Mit `external` auf Zeile 2 ist die Methode von aussen aufrufbar.

Zeile 3 stellt sicher, dass nur der Besitzer des Certifiers einen Account registrieren kann. Ist diese Prüfung erfolgreich, wird der Account `_who` der Liste `certs` hinzugefügt. Der Account ist nun für gratis Transaktionen berechtigt.

Der Event wird auf Zeile 6 in den Log geschrieben.

Listing 2.11: Methode `certified`

```
1 function certified(address _who)
2     external
3     view
4     returns (bool)
5 {
6     return certs[_who].active;
7 }
```

Mit der Methode `certified`, unter 2.11, kann jederzeit überprüft werden, ob ein Account (`_who`) zertifiziert ist. Mit `view` auf Zeile 3 ist deklariert, dass es sich um eine Abfrage ohne weitere Rechenkosten handelt. Solche Abfragen sind mit keinen Transaktionskosten verbunden.

Listing 2.12: Methode `revoke`

```
1 function revoke(address _who)
2     external
3     onlyDelegate
4     onlyCertified(_who)
5 {
6     certs[_who].active = false;
7     emit Revoked(_who);
8 }
```

Die Methode `revoke` entfernt einen zertifizierten Account (`_who`) von der Whitelist. Auf Zeile 3 wird wiederum sichergestellt, dass nur der Besitzer des Certifiers Änderungen vornehmen kann. Weiter wird auf Zeile 4 verifiziert, dass der Account `_who` in der Whitelist `certs` registriert ist.

Sind alle Bedingungen erfüllt, wird der Account von der Whitelist entfernt. Der Event wird auf Zeile 7 in den Log geschrieben.

2.3.2 Geprüfte Alternativen

Bei der Auswahl des Ethereumclients sind die populärsten[44] Clients berücksichtigt worden. Nebst Parity sind daher noch Geth, Aleth und Trinity evaluiert worden. Bei diesen Clients ist keine Möglichkeit vorhanden, bestimmte Accounts für gratis Transaktionen zu privilegieren. Daher sind sie zu diesem Zeitpunkt nicht für die FHNW geeignet.

2.4 Lösungsansätze

In diesem Kapitel werden die erarbeiteten Lösungsansätze für die Implementation eines Transaktionsmanagers vorgestellt. Die Stärken und Schwächen von jedem Lösungsansatz (LA) werden analysiert und

dokumentiert. Mit der vorgenommenen Analyse wird ein Favorit bestimmt. Dieser wird weiterverfolgt und implementiert.

2.4.1 Architektur

Die erarbeiteten Architektur-Lösungsansätze (ALA) werden in diesem Abschnitt behandelt.

2.4.1.1 ALA 1: Smart Wallet

Es wird selbst eine Smart Wallet entwickelt. Diese benötigt die volle Funktionalität einer herkömmlichen Wallet. Zusätzlich ist ein Schutzmechanismus gegen DoS Attacken implementiert. Für jeden Benutzer existiert eine Smart Wallet.

Wie unter 2.3.1 beschrieben, wird für die Betreuung der Blockchain der Client Parity mit einer Withelist verwendet.

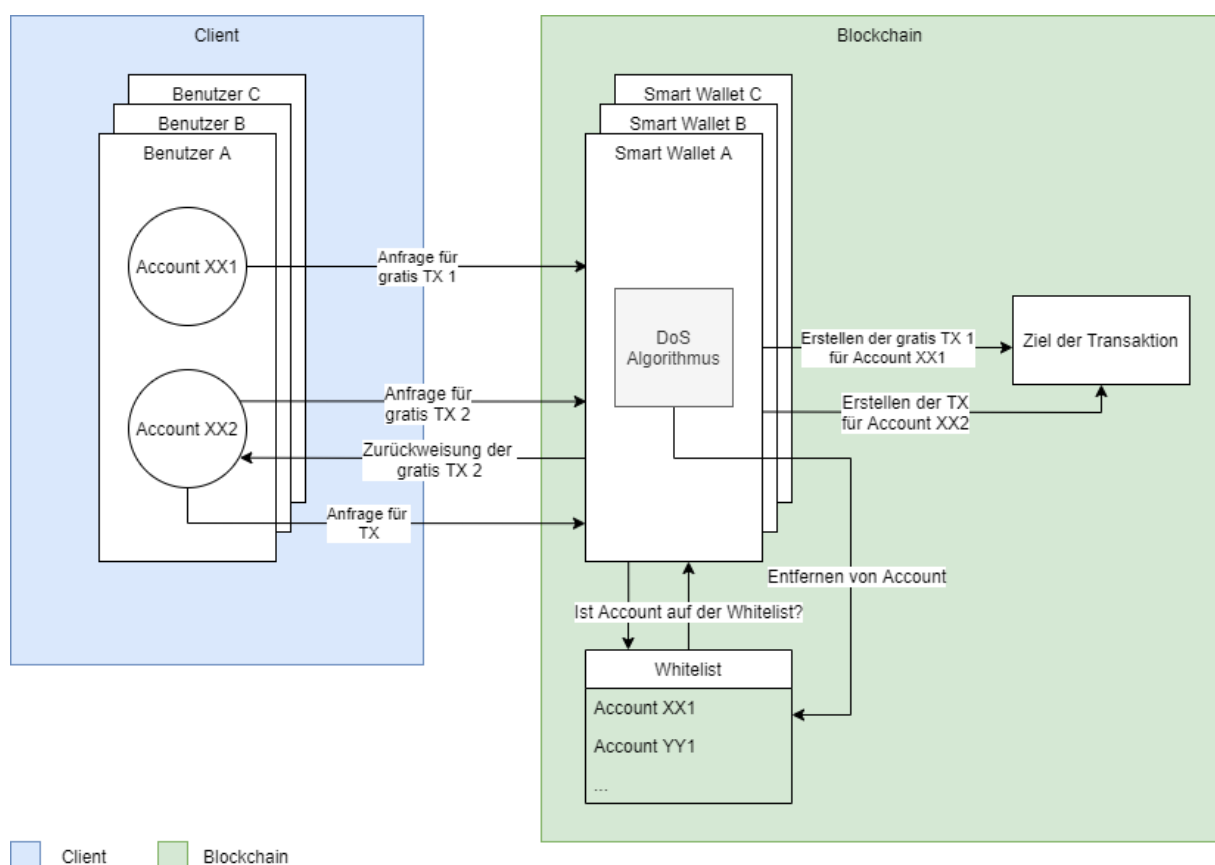


Abbildung 2.4: Architektur mit Smart Wallet

Auf dem Diagramm 2.4 ist zu sehen, dass ein Benutzer über mehrere Accounts verfügen kann. Diese sind in der Smart Wallet des Benutzers registriert. Gratis Transaktionen die von einem dieser Accounts durch die Smart Wallet gesendet werden, müssen vom DoS Algorithmus geprüft werden.

In diesem Beispiel möchte Benutzer A zwei gratis und eine reguläre Transaktion (TX) tätigen. Für die erste Transaktion wird Account XX1 verwendet. In der Smart Wallet wird überprüft, ob sich der Account auf der Whitelist befindet. Diese Prüfung ist erfolgreich. Die Smart Wallet erstellt die gewünschte Transaktion für den Account XX1.

Anschliessend wird die abgeschlossene Transaktion vom DoS Algorithmus ausgewertet. Dafür kann nicht nur eine einzige Transaktion betrachtet werden. Es muss das Verhalten aller Accounts von Benutzer A betrachtet werden, die sich auf der Whitelist befinden. Sollte der DoS Algorithmus, das Verhalten als Gefahr einstufen, müssen alle registrierten Accounts dieser Wallet von der Whitelist gelöscht werden. Der Benutzer kann weiterhin alle seine Accounts benutzen, muss aber für die Transaktionsgebühren aufkommen.

Die zweite gratis Transaktion möchte der Benutzer A mit seinem Account XX2 tätigen. Der Account befindet sich jedoch nicht in der Whitelist. Folglich wird die Transaktion nicht erstellt. Der Benutzer erhält einen Error.

Die dritte Transaktion wird ebenfalls mit Account XX2 erstellt. Es handelt sich jedoch nicht um eine gratis Transaktion. Diese wird erstellt und erreicht ihr Ziel ungehindert. Eine Prüfung durch den DoS Algorithmus ist folglich nicht nötig.

Wie in 2.3.1 beschrieben, prüft Parity bei einer gratis Transaktion nur, ob sich der Account in der Whitelist befindet. Das bedeutet, dass mit einem whitelisted Account auch gratis Transaktionen getätigt werden können, die nicht an die Smart Wallet gerichtet sind. Somit kann der Benutzer den DoS Schutzmechanismus umgehen. Deswegen muss ein Weg gefunden werden, der den Benutzer zwingt, Transaktionen über die Smart Wallet abzuwickeln.

Eine Möglichkeit ist Parity selbst zu erweitern. Anstelle einer Liste mit Accounts, muss in der Whitelist eine Liste von Verbindungen geführt werden. So kann definiert werden, dass nur eine Transaktion an die Smart Wallet gratis ist.

Pro Dieser Ansatz besticht durch die Tatsache, dass alles auf der Blockchain läuft. Somit werden grundlegende Prinzipien, wie Dezentralität und Integrität einer Blockchain bewahrt.

Contra Die Machbarkeit des Ansatzes ist unklar. Um diesen Ansatz umzusetzen, muss der Blockchain-client, Parity, erweitert werden. Es ist unklar, wie weitreichend die Anpassungen an Parity sind. Zudem wird eine zusätzliche Programmiersprache, Rust[40], für die Umsetzung benötigt.

Bei einer Anpassung am DoS Algorithmus, müssen alle Smart Wallets aktualisiert werden.

Prozessworkflow Der Prozessablauf für eine gratis Transaktion mit ALA 1 könnte folgendermassen aussehen:

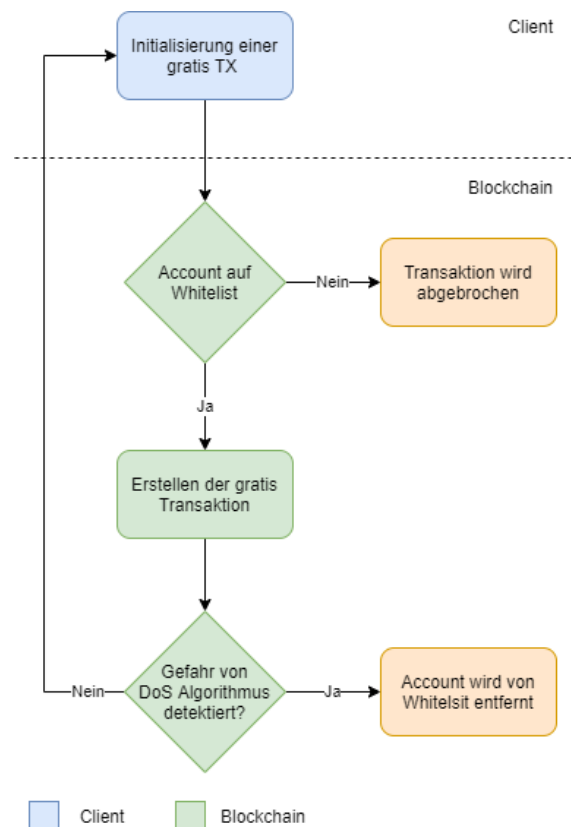


Abbildung 2.5: Flowchart für Smart Wallet

Im Diagramm 2.5 wird zu Beginn eine gratis Transaktion erstellt. Diese erreicht einen Parity Node. Nun wird geprüft ob der Sender der Transaktion sich auf der Whitelist befindet. Ist das nicht der Fall, wird die Transaktion abgebrochen. Ist die Prüfung erfolgreich, wird die Transaktion erstellt und ausgeführt. Anschliessend bewertet der DoS Algorithmus das Verhalten des verwendeten Accounts. Wird eine Gefahr für die Blockchain festgestellt, wird der Account von der Whitelist entfernt. Gratis Transaktionen sind dann für diesen Account nicht mehr verfügbar. Wenn keine Gefahr festgestellt wird, kann der Account weiter für gratis Transaktionen genutzt werden.

2.4.1.2 ALA 2: Externes Programm für die Verwaltung der Whitelist

Bei diesem Ansatz wird auf die Entwicklung einer Smart Wallet verzichtet. Stattdessen wird der Schutzmechanismus gegen DoS Attacken in einem externen Programm, dem Transaktionsmanager, implementiert. Dieser Ansatz verwendet ebenfalls die Whitelist von Parity, siehe 2.3.1.

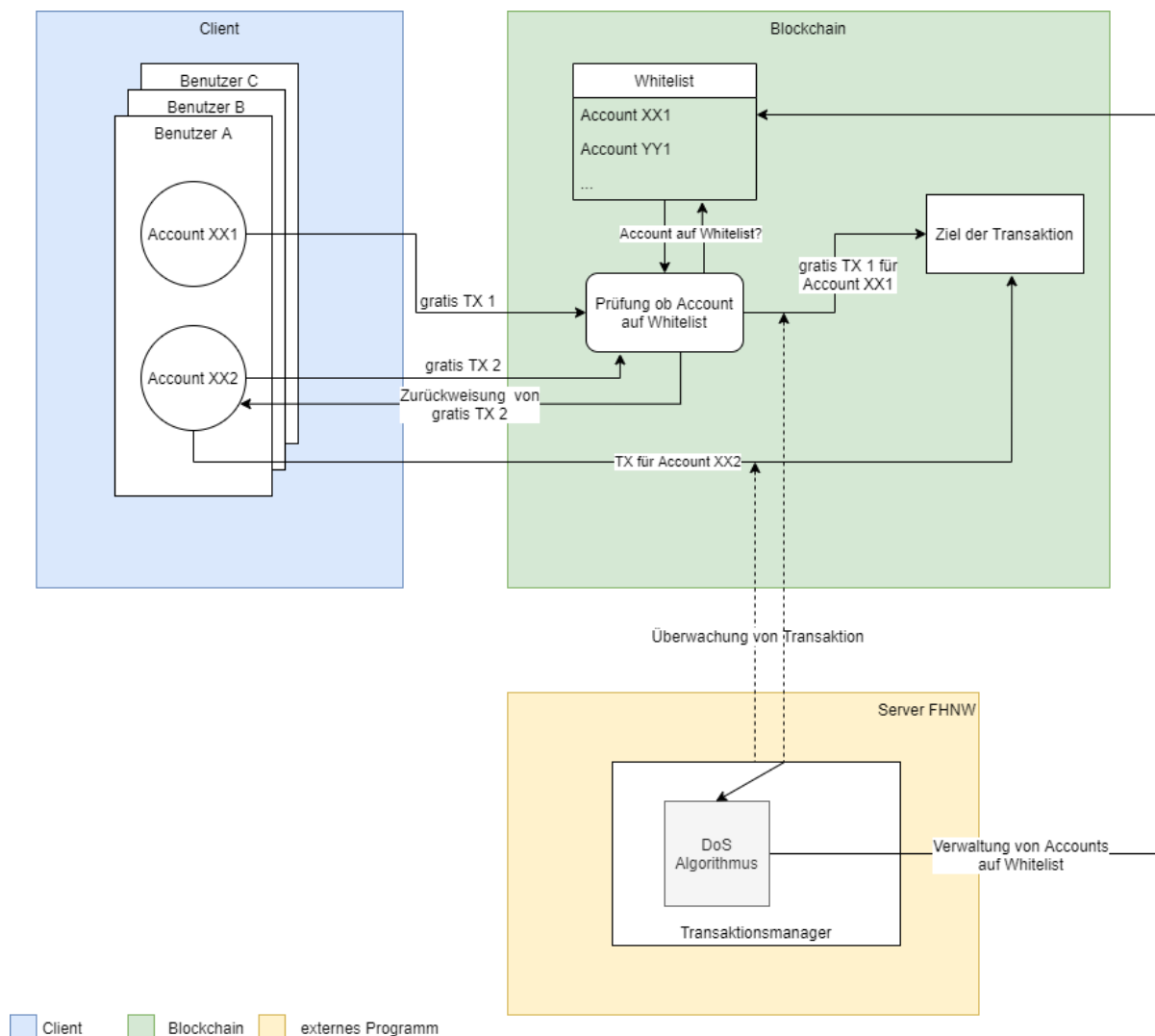


Abbildung 2.6: Externes Programm für die Verwaltung der Whitelist

Auf dem Diagramm 2.6 ist dargestellt, wie der Benutzer A zwei gratis und eine reguläre Transaktion tätigen will. Bei der ersten gratis Transaktion (gratis TX 1) wird der Account XX1 verwendet. Die Prüfung, ob sich der Account auf der Whitelist befindet, ist erfolgreich. Die Transaktion kann daher erfolgreich verarbeitet werden und erreicht ihr Ziel.

Für die zweite gratis Transaktion (gratis TX 2) wird der Account XX2 verwendet. Dieser befindet sich nicht auf der Whitelist. Die Prüfung schlägt deshalb fehl. Die Transaktion wird zurückgewiesen und der Benutzer erhält einen Error.

Auch die dritte Transaktion (Tx für Account XX2) verwendet den Account XX2. Da es sich nicht um eine gratis Transaktion handelt, kann diese problemlos ausgeführt werden.

Der Transaktionsmanager wird auf einem Server der FHNW ausgeführt. Er überwacht alle Transaktionen

die erstellt werden. Es werden jedoch nur gratis Transaktionen mit dem DoS Algorithmus geprüft. Sollte diese Prüfung eine Gefahr für die Blockchain detektieren, wird der entsprechende Account von der Whitelist gelöscht. Dies wird mit dem Pfeil „Verwaltung von Accounts auf Whitelist“ dargestellt.

Der Transaktionsmanager verfügt über einen eigenen Account, siehe 2.2.5. Dieser ist berechtigt, die Whitelist zu manipulieren. Dadurch kann bei einer identifizierten Attacke, der angreifende Account automatisch von der Whitelist gelöscht werden.

Pro Dieser Ansatz ist in der zur Verfügung stehenden Zeit sicher umsetzbar.

Falls eine Anpassung des DoS Schutzalgorithmus nötig ist, muss nur das externe Programm neu ausgerollt werden. Änderungen an der Blockchain sind nicht nötig.

Contra Es wird das Hauptprinzip, Dezentralität, einer Blockchain verletzt. Der Transaktionsmanager ist eine zentrale Autorität, die von der FHNW kontrolliert wird.

Für die Betreuung des Transaktionsmanagers wird ein Server benötigt. Eine weitere Komponente, die administriert werden muss.

Prozessworkflow Das Diagramm 2.7 zeigt, wie der Prozessworkflow beim ALA 2 aussieht.



Abbildung 2.7: Flowchart Transaktionsmanager für Verwaltung der Whitelist

Der Benutzer initialisiert eine gratis Transaktion. Beim Parity Node wird überprüft, ob sich der verwendete Account auf der Whitelist befindet. Ist er das nicht, wird die Transaktion abgebrochen und der Benutzer erhält einen Error. Ist ein Account verwendet worden, der sich auf der Whitelist befindet, wird die Transaktion in die Blockchain aufgenommen.

Der Transaktionsmanager überwacht alle Transaktionen die auf die Blockchain gelangen. Bei gefundenen gratis Transaktionen wird das Verhalten des Senders mit dem DoS Algorithmus analysiert. Wird eine Bedrohung für die Blockchain detektiert, wird der Account von der Whitelist entfernt.

Diese Prüfung findet nicht auf der Blockchain statt. Um einen Account von der Whitelist zu entfernen, generiert der Transaktionsmanager eine Transaktion.

2.4.1.3 ALA 3: Externes Programm mit Whitelist

Wie in Abbildung 2.8 illustriert, ist der Blockchain ein externes Programm, der Transaktionsmanager, vorgelagert. Dieser führt eine Whitelist für Accounts die gratis Transaktionen ausführen dürfen. Der DoS Algorithmus befindet sich ebenfalls im Transaktionsmanager. Transaktionen die den Transaktionsmanager erreichen, werden in einer weiteren Transaktion verpackt und an das Blockchainnetzwerk

gesendet. Dies ist nötig, um den ursprünglichen Sender zu bewahren.

Weiter wird eine Smart Wallet entwickelt. Diese ist nötig, um die verschachtelten Transaktionen des Programms zu verarbeiten. Aus dem Data-Feld wird die eigentliche Transaktion extrahiert und abgesetzt.

Jeder Benutzer besitzt eine eigene Smart Wallet um die Sender Identität für jeden Benutzer einmalig zu halten.

Auf der im Abschnitt 2.3.1 beschriebenen Whitelist ist nur der Account des externen Programmes aufgelistet. So ist sichergestellt, dass nur Transaktionen, die vom Programm weitergeleitet werden, kostenfrei durchgeführt werden können. Der Benutzer kann immer mit kostenpflichtigen Transaktionen auf die Smart Wallet zugreifen. Dies ist insbesondere wichtig, falls das Programm nicht aufrufbar ist, wenn z.B. der Server ausfällt.

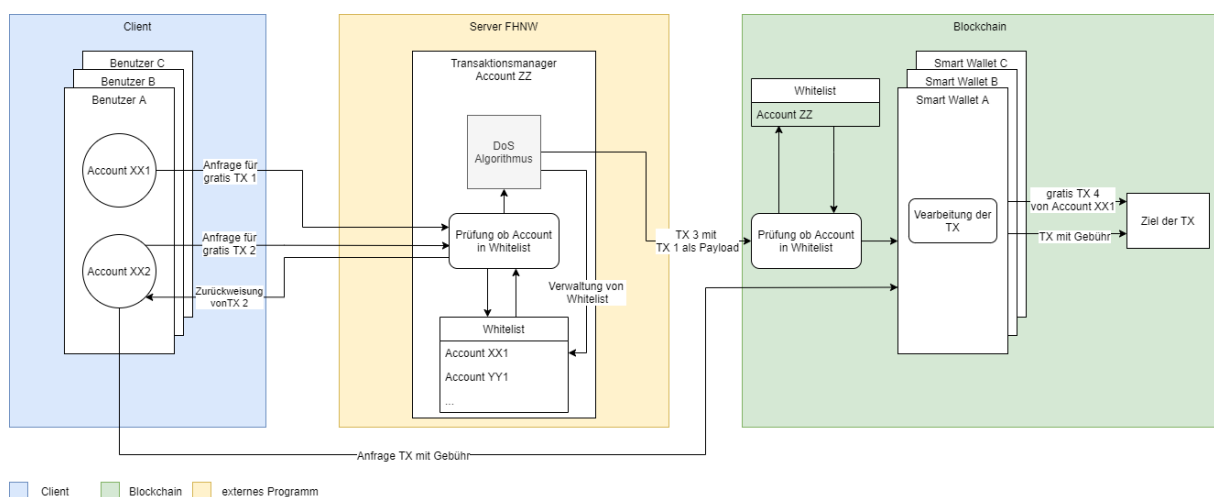


Abbildung 2.8: Transaktionsmanager mit Whitelist

Diagramm 2.8 zeigt die Benutzer mit ihren Accounts auf der linken Seite. Benutzer A initiiert in diesem Beispiel zwei gratis und eine gebührenpflichtige Transaktion.

Die erste gratis Transaktion (TX 1) wird an den Transaktionsmanager übermittelt. Hier wird zuerst geprüft, ob sich der verwendete Account auf der Whitelist befindet. Bei dieser Transaktion fällt die Prüfung positiv aus. Das Verhalten des Accounts wird daher durch den DoS Algorithmus untersucht. Sollte eine Gefahr detektiert werden, wird der betroffene Account von der Whitelist des Transaktionsmanagers gelöscht. Hier ist dies nicht der Fall. Die Transaktion „TX 1“ wird im Data Feld, siehe 2.2.3, einer neu generierten Transaktion (TX 3) an die Blockchain geschickt. Bei „TX 3“ handelt es sich ebenfalls um eine gratis Transaktion. Ansonsten würden die Kosten einer Transaktion nicht wegfallen, sondern nur umgelagert werden.

Sobald „TX 3“ beim Node ankommt, wird überprüft ob sich der Sender auf der Whitelist befindet. Das ist der Account des Transaktionsmanagers. Auf der Whitelist von Parity ist nur dieser Account vorhanden.

Nach erfolgter Prüfung, kommt „TX 3“ bei der Smart Wallet an und wird verarbeitet. Die ursprüngliche Transaktion „TX 1“ wird aus dem Data Feld extrahiert und versendet. Dies ist mit „TX 4“ dargestellt. Als Absender ist wieder der Account XX1 gesetzt.

Bei der zweiten gratis Transaktion (TX 2), befindet sich der verwendete Account nicht auf der Whitelist des Transaktionsmanagers. Die Transaktion wird zurückgewiesen und der Benutzer erhält einen Error. Bei der dritten Transaktion werden die Gebühren bezahlt. Es besteht daher kein Bedarf, die Transaktion an den Transaktionsmanager zu senden. Sie kann direkt über die Smart Wallet erstellt und verarbeitet werden.

Pro Dieser Ansatz ist in der gegebenen Zeit umsetzbar.

Falls eine Anpassung des DoS Schutzalgorithmus nötig ist, muss nur das externe Programm neu ausgerollt werden. Eine Aktualisierung der Blockchain ist nicht nötig.

Contra Es wird das Hauptprinzip, Dezentralität, einer Blockchain verletzt. Der Transaktionsmanager ist eine zentrale Autorität, die von der FHNW kontrolliert wird. Dieser benötigt einen Server, es kommt also eine weitere Komponente dazu. Diese muss ebenfalls administriert werden.

Dieser Ansatz bietet keine Vorteile im Vergleich zum ALA 2, ist aber mit der Verschachtelung von Transaktionen komplexer.

Prozessworkflow Wie auf Diagramm 2.9 gezeigt, beginnt der Prozess mit der Initialisierung einer gratis Transaktion durch den Benutzer.



Abbildung 2.9: Flowchart für Transaktionsmanager mit Whitelist

Diese wird an den Transaktionsmanager übermittelt. In erster Instanz wird geprüft, ob der verwendete Account für gratis Transaktionen berechtigt ist. Nur wenn diese Prüfung erfolgreich ist, wird die gratis Transaktion weiterbearbeitet. Anschliessend wird das Verhalten des Accounts evaluiert. Wird hier eine Gefahr detektiert, wird der Account von der Whitelist gelöscht und die Transaktion abgebrochen. Ist der Account berechtigt und es wird keine Gefahr detektiert, wird die Transaktion im Data Feld einer neuen gratis Transaktion an die Blockchain übermittelt.

Beim Node muss wiederum verifiziert werden, dass sich der Sender auf der Whitelist befindet. Da nur der Account des Transaktionsmanagers auf der Whitelist von Parity ist, ist sichergestellt, dass gratis Transaktionen nur durch den Transaktionsmanager auf die Blockchain gelangen können.

Anschliessend gelangt die neu generierte Transaktion zur Smart Wallet des Benutzers. Die ursprüngliche Transaktion wird aus dem Data Feld extrahiert und ausgeführt.

2.4.2 Evaluation der Architektur

Die erarbeiteten Lösungsansätze werden gegeneinander verglichen. Um zu bestimmen, welcher Ansatz weiterverfolgt wird, sind folgende Kriterien definiert worden:

Machbarkeit (MK) Bewertet die Machbarkeit des Ansatzes. Das berücksichtigt den gegebenen Zeitrahmen und die Komplexität des Ansatzes.

Da dieses Projekt im gegebenen Zeitrahmen abgeschlossen werden muss, ist es das wichtigste Kriterium. Daher wird es auch mit der höchsten Gewichtung versehen.

Gewichtung 3

Blockchainprinzipien (BCP) Gibt an ob die Prinzipien einer Blockchain berücksichtigt werden. Wie Dezentralität, Trust und Security.

Die Einhaltung der Prinzipien ist wichtig, aber für die FHNW nicht zwingend. Daher eine mittlere Gewichtung.

Gewichtung 2

Betrieb (BT) Bewertet den administrativen Aufwand im Betrieb und die Möglichkeit zur Automatisierung. Das umfasst Deployment von Smart Contracts, Anpassungen der Whitelist und Betreuung von zusätzlichen Servern.

Wird mit einer mittleren Gewichtung versehen. Ein zu hoher administrativer Aufwand ist nicht praktikabel.

Gewichtung 2

Jeder ALA wird auf diese drei Kriterien untersucht. Pro Kriterium können zwischen 1 und 3 Punkten erreicht werden, wobei 3 das Maximum ist. Die erreichten Punkte werden mit der entsprechenden Gewichtung multipliziert. Für die Evaluation, werden alle Punkte zusammengezählt. Der Ansatz mit den meisten Punkten wird weiterverfolgt.

Tabelle 2.1: Evaluation Lösungsansätze

	MK	BCP	BT	Total
Gewichtung	3	2	2	
ALA 1	1	3	2	13
ALA 2	3	2	2	17
ALA 3	2	1	3	12

2.4.2.1 ALA 1: Smart Wallet

Wir haben diesen Ansatz als sehr komplex eingestuft. Für die Anpassung von Parity muss eine zusätzliche Programmiersprache verwendet werden. Es ist nicht klar, wie weitreichend die nötigen Anpassungen sind. Zusätzlich muss eine Smart Wallet entwickelt werden.

Dieser Ansatz ist komplett dezentral und in die Blockchain integriert. Daher maximale Punktzahl bei Blockchain Prinzipien.

Falls eine Anpassung am DoS Algorithmus nötig ist, muss jede Smart Wallet aktualisiert werden. Die Automatisierung dieses Prozesses wird als komplex aber machbar eingeschätzt. Daher sind bei Betrieb 2 Punkte gesetzt.

2.4.2.2 ALA 2: Externes Programm für die Verwaltung der Whitelist

Die Entwicklung eines externen Programmes, welches getätigte Transaktionen der Blockchain prüft, ist in der gegebenen Zeit realisierbar. Daher erhält der ALA für Machbarkeit die volle Punktzahl.

Mit der Verwendung eines externen Programms, wird eine zentrale Autorität verwendet. Diese ist nicht dezentral und wird von der FHNW administriert. Da das Programm die Transaktionshistorie der Blockchain überwacht und nur gratis Transaktionen einschränkt, werden 2 Punkte für Blockchainprinzipien vergeben.

Falls eine Anpassung am DoS Algorithmus nötig ist, muss das externe Programm neu deployed werden. Es benötigt keine Anpassungen an der Blockchain selbst. Für die Verwaltung der Whitelist, braucht das Programm eine Funktion, um Accounts zur Whitelist hinzuzufügen. Diese Funktion kann einfach erweitert werden, um eine Liste von Accounts zur Whitelist hinzuzufügen. Dadurch ist das Hinzufügen von neuen Accounts für eine Klasse einfach automatisierbar.

Für die Betreuung des externen Programms wird ein zusätzlicher Server benötigt. Das bedeutet einen Mehraufwand für die FHNW.

Da der ALA einfach zu Automatisieren ist, sind für Betrieb 2 Punkte gesetzt worden.

2.4.2.3 ALA 3: Externes Programm mit Whitelist

Bei diesem ALA muss eine Smart Wallet und ein externes Programm entwickelt werden. Transaktionen werden im externen Programm verpackt und müssen von der Smart Wallet wieder entpackt werden. Dieser Vorgang wird als komplex eingestuft. Daher werden 2 Punkte für Machbarkeit vergeben.

Mit der Verwendung eines externen Programms, wird eine zentrale Autorität verwendet. Diese wird von der FHNW administriert. Im Gegensatz zu ALA 2, hat dieses Programm eine sehr viel zentralere Rolle. Das Programm interagiert nicht nur bei einer DoS Attacke mit der Blockchain, sondern ständig. Jede

gratis Transaktion wird an das Programm übermittelt und dort verarbeitet. Da die zentrale Autorität im Vergleich zu ALA 2 viel aktiver ist, ist für Blockchainprinzipien 1 Punkt vergeben worden.

Für die Betreuung des externen Programms ist ein zusätzlicher Server nötig.

Änderungen an der Smart Wallet bedingen ein erneutes Deployment.

In der Whitelist der Blockchain ist nur der Account des externen Programmes hinterlegt. Das Programm führt eine eigenen List von Accounts, die für gratis Transaktionen berechtigt sind.

Die Whitelist und der DoS Algorithmus befinden sich am selben Ort. Die Verwaltung und Automatisierung wird somit als unproblematisch bewertet. Daher sind bei Betrieb 3 Punkte vergeben worden.

2.4.2.4 Resultat Evaluation

Durch die hohe Gewichtung von Machbarkeit, erzielt ALA 2 die meisten Punkte. Im weiteren Verlauf des Projektes wird daher ALA 2 umgesetzt.

Für die Realisierung des externen Programmes ist die Programmiersprache der Wahl Java. Java ist den Teammitgliedern bereits bekannt. Mit der Bibliothek Web3j[45] sind Interaktionen mit der Blockchain effizient und intuitiv.

Im Anhang ist unter 6.2 ein weiterer Lösungsansatz aufgelistet. Dieser ist sehr früh in der Evaluierung als nicht realisierbar eingestuft worden und ist hier deshalb nicht aufgeführt.

2.4.3 DoS-Algorithmus

In diesem Abschnitt sind die Komponenten des DoS-Algorithmus aufgeführt. Verschiedene Parameter und deren Verwendung werden untersucht. Die Behandlung von Accounts, die vom Algorithmus als Gefahr identifiziert werden, wird ebenfalls evaluiert.

2.4.3.1 Parameter

Um zu bewerten, ob ein Account eine Gefahr für die Blockchain darstellt, braucht ein Algorithmus Parameter. Diese werden durch die Überwachung von getätigten gratis Transaktionen gesammelt. Dabei muss jeweils pro Account entschieden werden, ob ein Verhalten eine Gefahr darstellt. Nachfolgend sind mögliche Parameter für die Beurteilung von Accounts aufgeführt.

Sender Erlaubt die Verknüpfung von Transaktionen mit einem Account.

Empfänger Eine Transaktion wird immer an eine Adresse gesendet. Hierbei kann es sich sowohl um einen Benutzeraccount oder einen Smart Contract handeln.

Reset-Intervall Alle Interaktionen auf der Blockchain müssen relativ zu einem Zeitintervall bewertet werden. Hier werden zwei unterschiedliche Ansätze untersucht:

- Allgemeines Intervall

Gratis Transaktionen werden für alle Accounts im selben Zeitintervall betrachtet. Der Zeitpunkt ist relativ zum Programmstart. Beispielsweise ist als Intervall eine Stunde gesetzt und der Programmstart erfolgt um 8:00 UCT. Dadurch sind gratis Transaktionen die um 08:59 UTC gemacht werden, um 09:01 UTC nicht mehr relevant für die Beurteilung. Das hat zur Folge, dass Benutzer alle zulässigen Aktionen direkt vor und noch einmal, nach Ablauf eines Intervalls ausführen können.

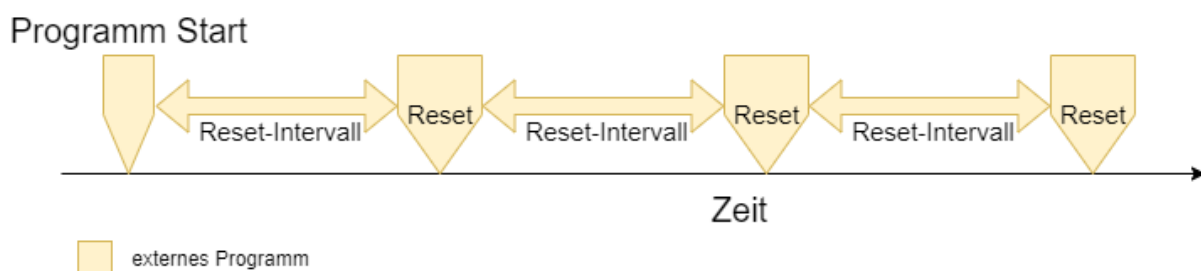


Abbildung 2.10: Allgemeines Reset-Intervall

Auf Abbildung 2.10 ist das allgemeine Reset-Intervall dargestellt. Am Ende eines Intervalls werden alle Zähler der Accounts zurückgesetzt.

- Individuelles Intervall

Das Intervall ist relativ zum Zeitpunkt einer getätigten gratis Transaktionen. Bei einer Prüfung wird untersucht, wie viele gratis Transaktionen der betroffene Account im vergangenen Zeitintervall, gerechnet ab dem Zeitpunkt der Prüfung, getätigt hat. Mit denselben Startparametern wie im oben aufgeführten Beispiel, ist eine um 08:59 UTC getätigte gratis Transaktion bis 09:59 relevant.

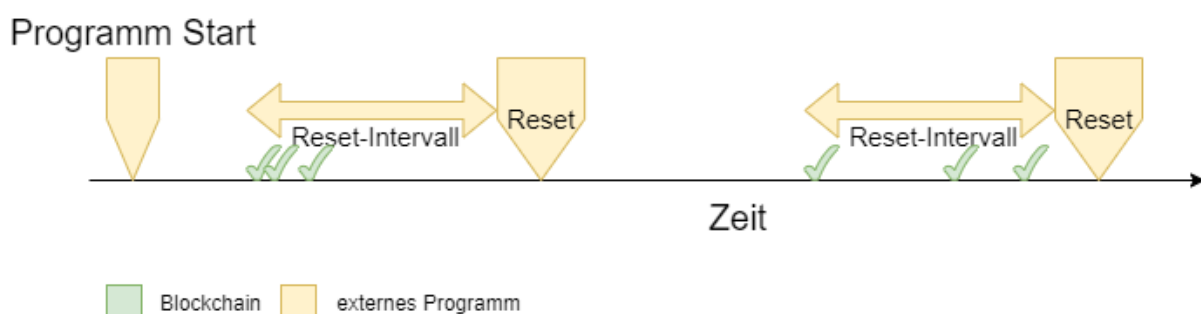


Abbildung 2.11: Individuelles Reset-Intervall

Auf dem Diagramm 2.11 ist ein individuelles Intervall graphisch dargestellt. Die grünen Haken, markieren getätigte gratis Transaktionen. Das Reset-Intervall wird erst ab einer getätigten gratis Transaktion gestartet und ist unabhängig vom Zeitpunkt des Programmstarts.

Anzahl getätigte Transaktionen Pro Account wird verfolgt, wie viele gratis Transaktionen pro Zeitintervall gemacht werden. Hier werden die Transaktionen unabhängig von Typ oder verursachten Rechengskosten auf der Blockchain gezählt.

Anzahl verbrauchtes Gas Pro Account wird verfolgt, wie viel Gas pro Zeitintervall auf der Blockchain durch dessen gratis Transaktionen verbraucht wird. Im Gegensatz zum oben genannten Parameter, werden hier die verursachten Rechengskosten auf der Blockchain berücksichtigt.

2.4.3.2 Wiederaufnahme auf die Whitelist

Falls die Prüfung durch den Algorithmus positiv ausfällt, wird der betreffende Account von der Whitelist gelöscht. In diesem Abschnitt sind mögliche Vorgehensweisen aufgeführt, um einen Account nach der Löschung automatisch wieder zur Whitelist hinzuzufügen.

Fixer Zeitpunkt für alle Analog zum allgemeinen Intervall.

Es wird ein fixes Intervall, relativ zum Programmstart, definiert. An dessen Ende werden alle Accounts zurückgesetzt. Das heisst das Kontingent wird bei allen Accounts wieder auf den konfigurierten Wert gesetzt. Von der Whitelist gelöschte Accounts werden dieser wieder hinzugefügt.

Ein Beispiel:

Als Intervall ist eine Woche definiert. Das Programm startet am Montag um 8:00 UTC.

Nun werden alle gesperrten Accounts, jeweils am nächsten Montag um 8:00 UTC wieder auf die Whitelist genommen.

Nach Zeitintervall Analog zum individuellen Intervall.

Ein Account wird für eine definierte Dauer von der Whitelist gelöscht. Die Zeit wird ab der Löschung von der Whitelist gemessen. Dadurch werden bei einem Vergehen alle Accounts gleich lange von gratis Transaktionen ausgeschlossen.

Inkrementierendes Zeitintervall Wie lange ein Account von der Whitelist entfernt wird, ist abhängig von der Anzahl bereits begangener Verstösse. Für das erste Vergehen ist eine Dauer der Suspendierung definiert. Dieses Intervall dient als Basis. Alle folgenden Verstösse und deren Suspendierung werden aus dem Basisintervall und den begangenen Verstössen berechnet.

Die Tabelle ?? zeigt eine quadratischen Abhängigkeit zwischen Verstößen und Dauer einer Sperrung. Basisintervall ist mit 5 Minuten definiert:

Tabelle 2.2: Beispiel für Suspendierung mit inkrementellen Intervallen

# Verstöße	Dauer Sperrung
1	5
2	25
3	45
4	80
5	125
6	180

2.4.3.3 Benutzermanagement

Bei der Verwaltung von Accounts geht es darum, wie die vorhergehenden Parameter und Intervalle auf die Accounts angewendet werden. Es werden drei Mögliche Ansätze betrachtet.

Kein Benutzermanagement Die Parameter werden global konfiguriert und gelten für alle Accounts. Eine Differenzierung von Accounts ist somit nicht möglich.

Parameter über Gruppen konfigurierbar Die Parameter sind über Gruppen konfiguriert. Jedem Account wird mindestens eine Gruppe zugewiesen, dieser erbt die Parameter der Gruppe. So lassen sich Strukturen der Schule, wie Studenten, Dozenten und Klassen einfach abbilden.

Parameter pro Account konfigurierbar Die Parameter sind bei jedem Account individuell konfigurierbar.

2.4.4 Evaluation DoS-Algorithmus

In diesem Abschnitt werden die Komponenten des Algorithmus evaluiert.

2.4.4.1 Parameter

Die aufgeführten Parameter werden auf ihre Relevanz für die Erkennung einer DoS Attacke geprüft.

Sender Ist zwingend nötig, um eine Transaktion einem Account zuweisen zu können.

Empfänger Dieser kann vom Sender frei gewählt werden. Es wird kein Einverständnis des Empfängers für eine Transaktion benötigt. Jeder Benutzer ist in der Lage, selbst neue Accounts zu erstellen und diese als Empfänger zu verwenden.

Der Parameter hat keine Aussagekraft und wird nicht verwendet.

Reset-Intervall Wir haben uns für die Implementierung eines allgemeinen Intervalls entschieden.

Der Ansatz ist bedeutend einfacher umzusetzen als ein individuelles Intervall und kann daher sicher in der gegebenen Zeit realisiert werden. Am Ende des Intervalls, werden die Zähler für alle Parameter pro Account zurückgesetzt.

Anzahl gratis Transaktionen Dieser Parameter wird verwendet. Er ermöglicht es eine DoS Attacke zu identifizieren, welche die Beeinträchtigung der Blockchain mittels einer grossen Zahl von gratis Transaktionen erreichen will.

Anzahl verbrauchtes Gas Wie unter 2.2.7 erwähnt, können Transaktionen mit einem sehr hohen Gas-Bedarf für eine DoS-Attacke verwendet werden. Da beim Angreifer mit der Verwendung von gratis Transaktionen keine Mehrkosten anfallen, ist dieser Angriff sehr naheliegend. Daher wird dieser Parameter ebenfalls verwendet.

2.4.4.2 Wiederaufnahme auf die Whitelist

Ein fixer Zeitpunkt ist sehr einfach umzusetzen. Allerdings werden dadurch die Accounts nicht mehr gleichbehandelt. Wie lange ein Account keine gratis Transaktionen mehr tätigen kann, ist abhängig davon, zu welchem Zeitpunkt er von der Whitelist gelöscht wird.

Mit einem Zeitintervall ab Zeitpunkt des Vergehens, werden alle Accounts gleich lange von der Whitelist gelöscht. Dieser Ansatz bietet daher mehr Fairness als ein fixer Zeitpunkt. Die Implementierung wird als komplexer betrachtet.

Je öfter mit einem Account gegen die Regeln verstossen wird, desto kleiner ist die Wahrscheinlichkeit, dass es sich um Versehen handelt. Daher kann davon ausgegangen werden, dass ein Wiederholungstäter aktiv versucht, die Blockchain zu schädigen. Mit einem inkrementierenden Intervall werden diese Accounts gezielt und härter bestraft als bei den anderen Ansätzen.

Eine inkrementierende Bestrafung ist bei einem Schutz vor DoS Attacken nicht relevant. Es genügt, wenn der Angriff unterbrochen werden kann. Eine immer stärkere Bestrafung ist nicht nötig.

Die Dauer der Suspendierung ist nicht mehr konstant. Daher muss einem Benutzer mitgeteilt werden, für wie lange er suspendiert ist. Wie diese Kommunikation realisiert werden soll, ist unklar.

Wir haben uns entschieden, ein Zeitintervall ab Zeitpunkt des Vergehens zu implementieren. Dieser Ansatz ist fair und verständlich für die Benutzer. Das Intervall ist fortan Revoke-Intervall genannt.

2.4.4.3 Benutzermanagement

Es besteht der Bedarf, dass Accounts von Dozenten toleranter behandelt werden als solche von Studenten. Daher muss ein Benutzermanagement implementiert werden.

Ein gruppenbasiertes Benutzermanagement ist intuitiv und effizient, da vorhandene Strukturen der FHWN, wie Klassen oder Dozenten, abgebildet werden können. Die Implementation wird jedoch als sehr komplex eingeschätzt. Die Realisierbarkeit in der gegebenen Zeit ist fraglich. Der Ansatz wird daher nicht implementiert.

Das lässt nur die Möglichkeit, jeden Account einzeln zu konfigurieren. Es wird erwartet, dass für die Mehrheit der Accounts kein Bedarf an individuellen Parametern besteht. Um diesen Umstand gerecht zu werden, werden Standardparameter angeboten. Diese werden verwendet, wenn Parameter für einen Account nicht explizit definiert sind. So müssen nur Ausnahmen individuell konfiguriert werden.

2.4.5 Konfiguration des Algorithmus

Um dem Betreiber die Möglichkeit zu geben, den Algorithmus an seine Bedürfnisse anzupassen, können die Parameter und Zeitintervalle, siehe 2.4.4, konfiguriert werden. Die Konfiguration wird mit über Dateien vorgenommen. Für alle Parameter müssen natürliche Zahlen verwendet werden. Folgende Parameter können pro Account gesetzt werden:

Gratis Transaktions Limite Definiert die maximale Anzahl gratis Transaktionen die pro Reset-Intervall getätigt werden können.

Gratis Gas Limite Definiert die maximale Menge an Gas, die mit gratis Transaktionen, innerhalb eines Reset-Intervalls verbraucht werden können.

Revoke-Intervall Anzahl Minuten für die ein Account, bei einer positiven Prüfung durch den Algorithmus, von der Whitelist gelöscht wird.

Folgende Parameter gelten für alle Accounts:

Reset-Intervall Einheit ist Minuten, definiert die Länge des Reset-Intervalls.

Standardwert gratis Transaktions Limite Gilt für Accounts, die ohne Transaktionslimite erfasst werden. Definiert die maximale Anzahl gratis Transaktionen, die pro Reset-Intervall getätigt werden können.

Standardwert gratis Gas Limite Gilt für Accounts, die ohne Gaslimite erfasst werden. Definiert die maximale Menge an Gas, die mit gratis Transaktionen innerhalb eines Reset-Intervalls verbraucht werden können.

Standardwert Revoke-intervall Gilt für Accounts die ohne Revoke-Intervall erfasst werden. Definiert wie lange ein Account bei einem Vergehen von der Whitelist suspendiert wird.

Bei der Konfiguration sollten die Abhängigkeiten zwischen den Parametern geachtet werden. Verfügbares Gas, Anzahl Transaktionen und das Reset-Intervall sollten immer zusammen konfiguriert werden.

3 Praktischer Teil

Dieses Kapitel beschreibt, wie die gewonnen theoretischen Grundlagen umgesetzt sind. Die realisierte Lösung wird kritisch hinterfragt und anderen Lösungsansätzen gegenübergestellt.



Abbildung 3.1: Blockchain mit Transaktionsmanager

Im folgenden Abschnitt wird die Blockchain behandelt, diese ist auf dem Diagramm 3.1 grün eingefärbt. Anschliessend wird der Transaktionsmanger behandelt, im Diagramm gelb eingefärbt.

3.1 Parity

In diesem Abschnitt ist beschrieben, wie die Blockchain konfiguriert ist. Als Client wird die Version[46] 2.5.13 von Parity verwendet.

3.1.1 Konfiguration der Blockchain

Parity wird mit der Konsole gestartet. Der Benutzer hat hier die Möglichkeit, gewisse Parameter an Parity zu übergeben. Somit ist eine einfache Konfiguration möglich. Für kompliziertere Konfigurationen wird die Verwendung von einer Konfigurationsdatei empfohlen, diese ist im nächsten Abschnitt 3.1.1.1 beschrieben.

Die hier gezeigte Konfiguration ist für die Entwicklung verwendet worden. Hierbei ist es wichtig, dass Aktionen möglichst schnell auf der Blockchain sichtbar sind. Aus diesem Grund ist eine Single Node Development Chain verwendet worden. Ein Mining-Algorithmus ist daher nicht nötig.

Für einen produktiven Betrieb sollte die Konfiguration auf die eigenen Bedürfnisse geprüft und gegebenenfalls angepasst werden.

3.1.1.1 Config.toml

Für die Konfiguration der Blockchain wird eine Konfigurationsdatei verwendet. Diese hat das Dateiformat „.toml“[47]. Die hier aufgeführten Code-Ausschnitte gehören zusammen. Um die Leserlichkeit zu erhöhen, werden sie jedoch separiert aufgeführt.

Listing 3.1: Konfigurationsdatei für Parity

```
1 [parity]
2 chain = "/home/parity/.local/share/io.parity.ethereum/genesis/
   instant_seal.json"
3 base_path = "/home/parity/"
```

Parity Der Code-Ausschnitt 3.1 zeigt wie die grundlegenden Eigenschaften der Blockchain definiert sind. Dazu gehören Genesisblock und der Speicherort.

Zeile 2 Der zu verwendende Genesisblock. Es wird der Pfad zu der entsprechenden JSON Datei[48] angegeben. Der gezeigte Pfad ist relativ zum Docker-Home.

Zeile 3 Mit „base_path“ wird angegeben, wo die Blockchain abgespeichert werden soll. Hier wird das gewünschte Verzeichnis angegeben. Der gezeigte Pfad ist relativ zum Docker-Home.

Listing 3.2: Konfigurationsdatei für Parity

```
1 [rpc]
2 cors = ["all"]
3 apis = ["net", "private", "parity", "personal", "web3", "eth"]
```

RPC Diese Sektion, 3.2, definiert, wie die Blockchain erreichbar ist.

Zeile 2 „cors“ steht für Cross-Origin Requests. Dieser Parameter wird benötigt, um die Interaktion von Remix[49] oder Metamask[50] mit der Blockchain zu ermöglichen.

Zeile 3 Hier sind die API's definiert, welche über HTTP zur Verfügung gestellt werden.

Listing 3.3: Konfigurationsdatei für Parity

```
1 [mining]
2 min_gas_price = 10000000000
3 refuse_service_transactions = false
4 tx_queue_no_unfamiliar_locals = true
5 reseal_on_txs = "all"
```

Mining Die unter 3.3 aufgeführte Sektion regelt das Verhalten beim Mining von Blocks.

Zeile 2 Der minimale Gas-Preis der gezahlt werden muss, damit eine Transaktion in einen Block aufgenommen wird. Der Preis ist in WEI angegeben. Um sicherstellen, dass nur die definierte Benutzergruppe gratis Transaktionen tätigen kann, muss dieser Wert grösser als null sein.

Zeile 3 Service Transaktionen haben einen Gas-Preis von null. Wird hier „true“ gesetzt, können keine gratis Transaktionen getätigt werden, unabhängig davon, ob eine Whitelist vorhanden ist oder nicht.

Zeile 4 Dieser Parameter wird benötigt, dass Transaktionen die mittels RPC an Parity übermittelt werden, nicht als lokal betrachtet werden. Das ist sehr wichtig, da lokale Transaktionen standardmässig auch über einen Gas-Preis von null verfügen dürfen. So wird sichergestellt, dass nur die definierte Benutzergruppe gratis Transaktionen tätigen darf.

Zeile 5 Durch die Einstellung auf Zeile 4 werden alle eingehenden Transaktionen behandelt, als ob sie von ausserhalb kommen. Standardmässig, werden nur lokale Transaktionen verarbeitet. Daher muss hier explizit definiert werden, dass alle Transaktionen verarbeitet werden.

Listing 3.4: Konfigurationsdatei für Parity

```
1 [misc]
2 unsafe_expose = true
```


Zeile 2 Die verwendete Netzwerk ID. Die grossen Netzwerke haben eine definierte ID. Falls einem bestehenden Netzwerk beigetreten werden soll, muss diese korrekt gewählt werden. Der Wert 11 ist keinem Netzwerk zugeordnet, daher kann dieser für ein privates Netzwerk genutzt werden.

Zeile 3 Der `registrar` hat als Wert die Adresse der Name Registry, siehe 2.3.1.1. Da bereits beim ersten Start von Parity die Adresse der Name Registry hinterlegt werden muss, findet das Deployment direkt in der Blockchainspezifikation statt.

Zeile 4 Die maximale Grösse eines Smart Contracts.

Zeile 5 Spezifiziert die maximale Anzahl Bytes, welche im Feld `extra_data` des Headers eines Blockes mitgegeben werden kann.

Zeile 6 Definiert den minimalen Gasbetrag, der bei einer Transaktion mitgegeben werden muss.

Zeile 7 Schränkt die Schwankungen der Gas Limite zwischen Blöcken ein.

Listing 3.7: Blockchainspezifikation mit Genesisblock

```
1      "genesis": {
2          "seal": {
3              "generic": "0x0"
4          },
5          "difficulty": "0x20000",
6          "gasLimit": "0x7A1200"
7      },
```

Code-Ausschnitt 3.7:

Zeile 1 - 7 Mit dem Abschnitt `genesis` ist der Genesis Block, also der erste Block, der Blockchain definiert.

Zeile 2 - 4 Hier kann weiter definiert werden, wie Blöcke verarbeitet werden sollen. Da für dieses Projekt, valide Blöcke sofort in die Blockchain eingefügt werden, sind keine weiteren Einstellungen nötig.

Zeile 5 Gibt die Schwierigkeit des Genesis Blocks an. Da als Engine InstantSeal verwendet wird, hat dieser Parameter keinen Einfluss.

Zeile 6 Gibt an, was die Gaslimite des Genesis Blockes ist. Da die Gaslimite für Blöcke dynamisch berechnet wird, hat dieser Wert einen Einfluss auf zukünftige Gaslimiten.

Listing 3.8: Blockchainspezifikation mit Genesisblock

[illegible]

Code-Ausschnitt 3.8:

Zeile 1 - 4 Dieser Abschnitt erlaubt, Accounts zu definieren. Diese können für Benutzer oder Smart Contracts sein. Jeder Account wird mit einer Adresse und einem Guthaben initialisiert. Bei einem Account für einen Smart Contract, wird zusätzlich dessen Bytecode angegeben.

Zeile 2 Hier ist die SimpleRegistry, siehe Abschnitt 2.3.1 und 2.3.1.1, definiert. Der erste Parameter ist die Adresse, unter welcher der Smart Contract erreichbar sein soll. Das Guthaben wird mit einem Ether initialisiert. Der Wert für `constructor` ist der Bytecode des kompilierten Smart Contracts. Dieser ist aufgrund seiner Grösse durch einen Platzhalter ersetzt worden.

Zeile 3 Definition von einem Benutzeraccount. Der erste Parameter ist die Adresse. Dem Account kann ein beliebiges Guthaben zugewiesen werden.

3.1.2 Name Registry

Die Name Registry wird standardmässig in Parity verwendet. Die zur Verfügung gestellte Implementation der Name Registry ist `SimpleRegistry` genannt. Der vollständige Code ist im Anhang unter 6.6.1 verlinkt.

3.1.3 Certifier

Parity stellt eine Implementation des Certifiers zu Verfügung, den `SimpleCertifier`. Der vollständige Code ist im Anhang unter 6.6.1 verlinkt.

Sobald der Certifier bei der Name Registry registriert ist, können Accounts definiert werden, die gratis Transaktionen tätigen können.

3.1.4 Interaktionsübersicht auf einem Parity Node

In diesem Abschnitt sind die Interaktionen zwischen Name Registry und Certifier aufgeführt.

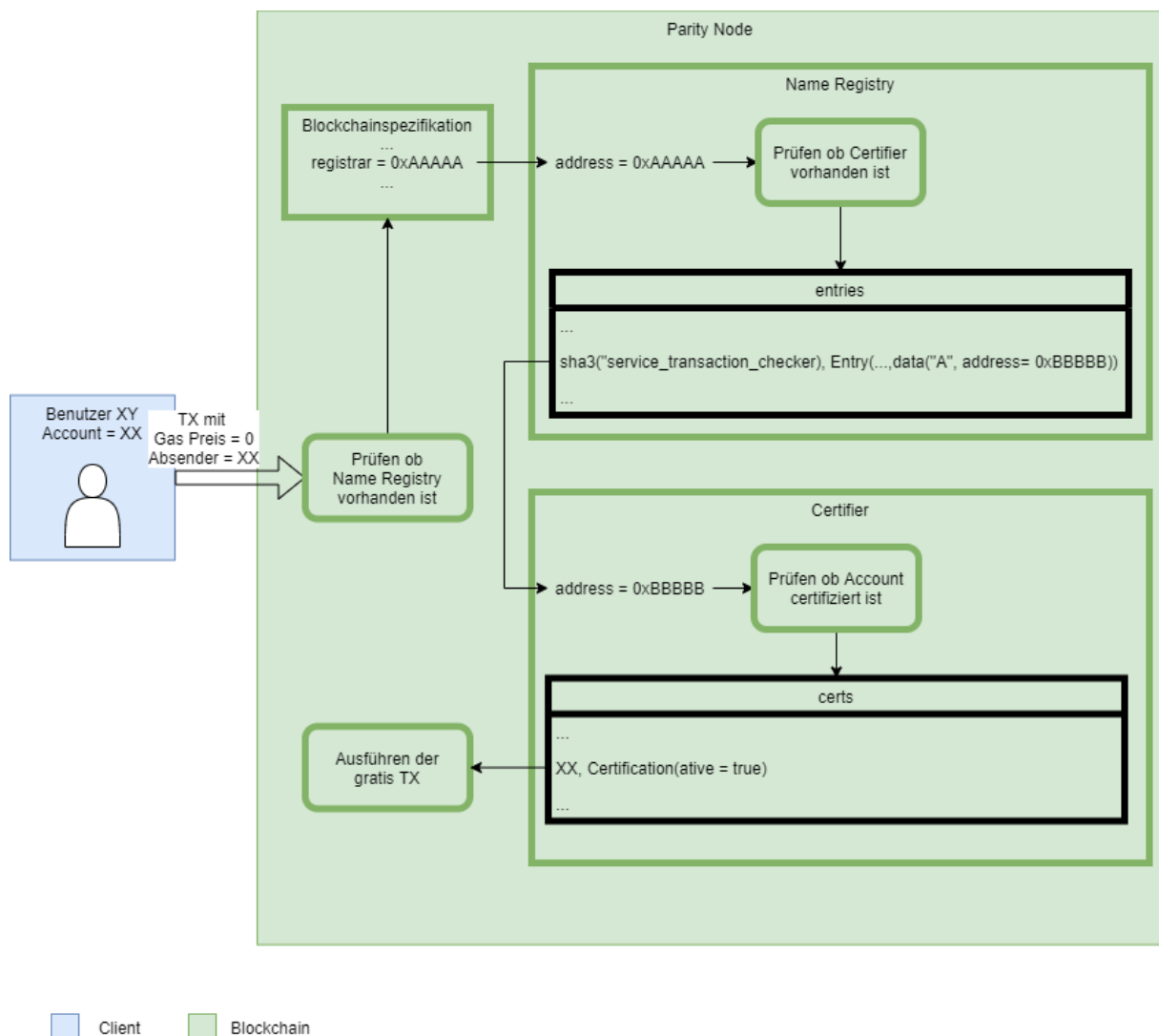


Abbildung 3.2: Interaktion zwischen Name Registry und Certifier

Auf dem Diagramm 3.2 ist links der Benutzer XY in blau dargestellt. Der Parity Node ist in grün dargestellt. Der Benutzer sendet eine Transaktion mit einem Gas Preis von null an den Parity Node. Als erstes wird in Parity geprüft, ob eine Name Registry vorhanden ist. Deren Deployment ist unter 3.1.1.2 beschrieben. Der Eintrag `registrar` hat als Wert die Adresse der Name Registry hinterlegt. In der Name Registry wird das Mapping `entries` untersucht. Der Certifier muss unter dem richtigen Namen hinterlegt sein. Der Name ist ein SHA3-Hash des Strings „service_transaction_checker“. Der Hash ist der Zugriffsschlüssel im Mapping `entries`. Der korrespondierende Wert ist ein `Entry` für den Certifier. Im Struct `Entry` ist ein weiteres Mapping, `data`, vorhanden. In `data` muss unter dem Zugriffsschlüssel „A“ die Adresse des Certifiers abgelegt sein. Das ist keine Konfiguration, sondern hart kodiert in Parity. Dieser Vorgang ist unter 3.1.4.1 ausführlich beschrieben.

Mit der so ermittelten Adresse des Certifiers, kann dieser aufgerufen werden. Im Certifier wird geprüft, ob der Sender der gratis Transaktion, also Account „XX“, zertifiziert ist.

Wenn einer dieser Schritte fehlschlägt, erhält der Benutzer einen Error, da er einen ungültigen Gas Preis verwendet.

Können alle Schritte erfolgreich durchgeführt werden und der Account ist zertifiziert, wird die Transaktion in die Blockchain aufgenommen.

3.1.4.1 Deployment und Registrierung

Für eine Erfolgreiche Verwendung des Certifiers muss die Name Registry in der Blockchainspezifikation definiert sein. Sobald Parity gestartet ist, kann mit dem Deployment des Certifiers begonnen werden. Hierfür wird Java und die Bibliothek Web3j[45] verwendet.

Um in Java mit Smart Contracts auf der Blockchain interagieren zu können, wird eine Wrapperklasse des Smart Contracts benötigt. Für dessen Generierung wird das Web3j Command Line Tool (web3j-cli)[51] und der Solidity Compiler (solc)[52] verwendet. Die Wrapper für die SimpleRegistry und den SimpleCertifier sind im Anhang unter 6.6.2 verlinkt.

Um einen Smart Contract auszurollen wird eine Instanz der generierten Wrapperklasse genutzt. Es wird die Methode `deploy` der Wrapperklasse genutzt.

Listing 3.9: Deployment des Certifiers

```
1 private Web3j web3j = Web3j.build(new HttpService("http://jurijnas.  
   myqnapcloud.com:8545/"));  
2 private TransactionManager transactionManager = new  
   RawTransactionManager(web3j, Credentials.create(privateKey));  
3  
4 private SimpleCertifier simpleCertifier;  
5 try {  
6     simpleCertifier = SimpleCertifier.deploy(web3j, transactionManager,  
       new DefaultGasProvider()).send();  
7 } catch (Exception e) {  
8     e.printStackTrace();  
9 }  
10  
11 String simpleCertifierAddress = simpleCertifier.getContractAddress();
```

Der Codeausschnitt 3.9 zeigt das Deployment des `SimpleCertifiers`. Die Verbindung zu einem Node wird mit einer Instanz von `Web3j` auf Zeile 1 definiert. Auf der zweiten Zeile wird ein `TransactionManager` instanziiert. Dieser definiert, wie und mit welchem Account auf die Ethereum-blockchain verbunden wird.

Auf Zeile 6 findet das eigentliche Deployment statt. Nebst dem `Web3j` und dem `Transactionmanager` wird ein `ContractGasProvider` benötigt. Dieser definiert den Gas Price und die Gas Limite. Mit dem

`DefaultGasProvider` werden Standardwerte verwendet. Durch das Deployment erhalten wir eine Instanz des `SimpleCertifiers`. Diese kann nun verwendet werden um weitere Aktionen auf der Blockchain auszuführen.

Auf Zeile 11 wird die Adresse des Smart Contracts in eine Variable gespeichert.

Um den Certifier bei der Name Registry registrieren zu können, muss von der Name Registry ebenfalls eine Instanz erstellt werden. Auch hier wird die Wrapperklasse verwendet.

Listing 3.10: Erstellen der Name Registry - Instanz

```
1 private Web3j web3j = Web3j.build(new HttpService("http://jurijnas.  
   myqnapcloud.com:8545/"));  
2 private TransactionManager transactionManager = new  
   RawTransactionManager(web3j, Credentials.create(privateKey));  
3  
4 private SimpleRegistry simpleRegistry;  
5 try {  
6     simpleRegistry = SimpleRegistry.load(simpleRegistryAddress, web3j,  
       transactionManager, new DefaultGasProvider());  
7 } catch (Exception e) {  
8     e.printStackTrace();  
9 }
```

Der Codeausschnitt 3.10 zeigt wie eine Instanz der `SimpleRegistry` erstellt wird. Um eine Instanz von einem bereits platzierten Smart Contract zu erhalten, wird die Methode `load` verwendet. Als erstes Argument wird die Adresse der Name Registry mitgegeben. Analog zum vorherigen Beispiel, 3.9, wird die Verbindung zur Blockchain mit `Web3j`, einem `TransactionManager` und einem `DefaultGasProvider` definiert. Der Rückgabewert ist eine Instanz der `SimpleRegistry`.

Mit den zur Verfügung stehenden Instanzen, kann die Registrierung des Certifiers bei der Name Registry gemacht werden.

Listing 3.11: Reservierung und anschließende Registrierung bei der Name Registry

```
1 private static BigInteger REGISTRATION_FEE = BigInteger.valueOf  
   (1000000000000000000L);  
2  
3 String str_hash = "6  
   d3815e6a4f3c7fcec92b83d73dda2754a69c601f07723ec5a2274bd6e81e155";  
4 private byte[] hash = new BigInteger(str_hash, 16).toByteArray();  
5  
6 try {  
7     simpleRegistry.reserve(hash, REGISTRATION_FEE).send();  
8     simpleRegistry.setAddress(hash, "A", simpleCertifier.  
       getContractAddress()).send();  
9 } catch (Exception e) {  
10     e.printStackTrace();  
11 }
```

Im Codeausschnitt 3.11 ist die Reservation und Registrierung des Certifiers bei der Name Registry aufgeführt. Für die Registrierung wird eine Gebühr von einem Ether erhoben. Dafür wird auf Zeile 1 eine Variable vom Typ `BigInteger` instanziiert.

Der auf Zeile 3 definierte String `str_hash` ist der sha3-Hash für den String `service_transaction_checker`. Dieser wird auf Zeile 4 in ein Byte-Array umgewandelt. Diese Variable hält den Namen, unter welchem der Certifier bei der Name Registry registriert wird. Die Verwendung des Strings `service_transaction_checker` und seine Umwandlung sind in Parity hart kodiert und können nicht angepasst werden.

Auf Zeile 7 wird die Reservierung bei der Name Registry vorgenommen. Hier wird der Name und die anfallende Gebühr von einem Ether gesendet.

Auf Zeile 8 wird die Registrierung abgeschlossen. In der Name Registry wird die Bindung zwischen Namen und Adresse erstellt. Als erster Parameter wird der Name übergeben. Das zweite Argument ist der Zugriffsschlüssel in der Map. Auch dieser ist von Parity vorgegeben, es muss zwingend `"A"` übergeben werden. Das dritte Argument ist die Adresse des Certifiers. Diese wird von dessen Instanz abgerufen.

3.2 Transaktionsmanager

In diesem Kapitel ist die Implementierung des Transaktionsmanagers zur Überwachung der Whitelist in Parity beschrieben. Anhand von Codeausschnitten wird die Funktionsweise von einzelnen Komponenten näher erklärt.

3.2.1 Wrapperklassen

Für die Interaktion mit Smart Contracts werden generierte Wrapperklassen verwendet. Es ist je eine Wrapperklasse für die Name Registry und den Certifier vorhanden. Für dessen Generierung und Verwendung siehe 3.1.4.1.

3.2.2 Überwachung von Transaktionen

Um die Transaktionen auf der Blockchain zu observieren wird ein `Filter`[53] von Web3j verwendet. Dieser erlaubt es, eine `Subscription` zu erstellen. Diese läuft asynchron.

Jede getätigte Transaktion wird von der `Subscription` erfasst. In einem ersten Schritt wird der verwendete Gas Preis der Transaktion betrachtet. Nur wenn dieser null ist, wird die Transaktion und der dazugehörige Account durch den DoS Algorithmus evaluiert.

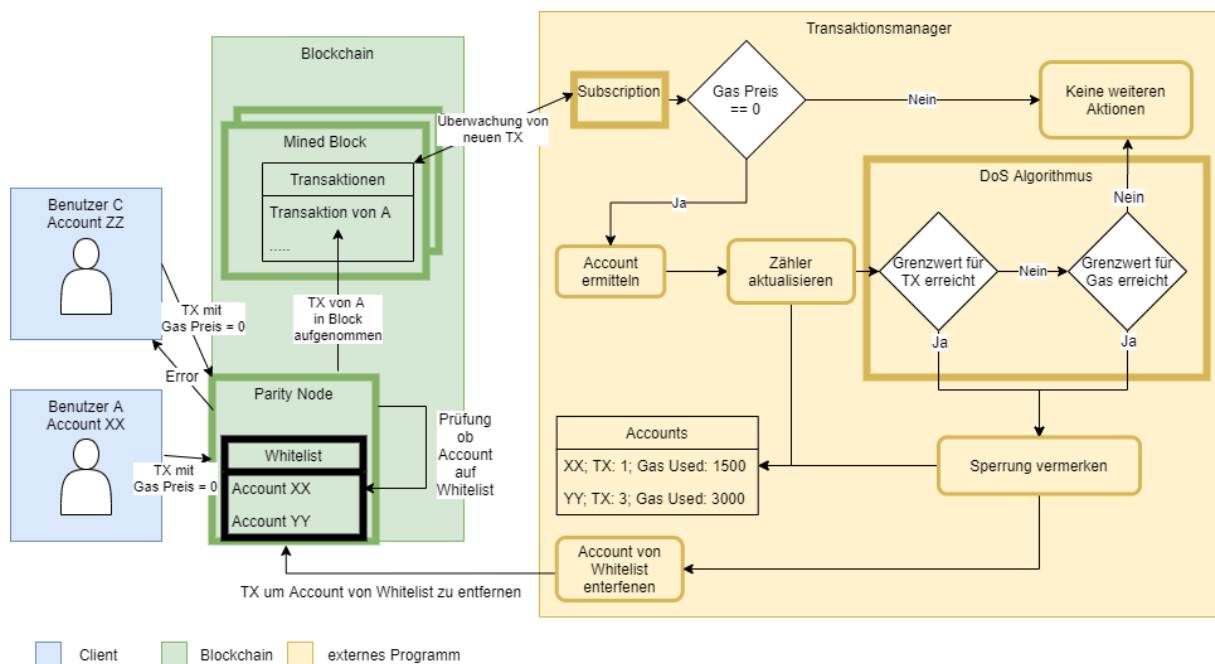


Abbildung 3.3: Prozess bei der Überwachung von Transaktionen

Auf der Abbildung 3.3 sind die Interaktionen zwischen dem Transaktionsmanager, der Blockchain und den Benutzern gezeigt. Transaktionen sind mit „TX“ abgekürzt.

Links auf dem Diagramm sind die Benutzer „A“ und „C“ mit ihrem jeweiligen Account abgebildet. In der Liste **Accounts** im Transaktionsmanager sind alle Accounts erfasst, die für gratis Transaktionen berechtigt sind. Hier ist für jeden Account vermerkt, wie viele gratis Transaktionen und gratis Gas im aktuellen Reset-Intervall bereits verbraucht wurde.

Die beiden Benutzer erstellen je eine Transaktion mit einem Gas Preis von null. Diese werden an den Parity Node übermittelt.

Beim Node wird geprüft, ob der Sender in der Whitelist erfasst ist. Dieser Vorgang ist unter 3.1.4 genauer erklärt. Der Account von Benutzer C ist nicht erfasst. Daher wird seine Transaktion vom Node abgelehnt. Der Benutzer C erhält einen Error, da er einen ungültigen Gas Preis verwendet hat.

Die Transaktion von Benutzer A wird vom Node akzeptiert, da sein Account in der Whitelist erfasst ist. Die Transaktion wird anschliessend verarbeitet und in den nächsten Block aufgenommen.

Die Subscription im Transaktionsmanager registriert, dass eine neue Transaktion in die Blockchain aufgenommen worden ist. Die Daten der Transaktion werden heruntergeladen. Der verwendete Gas Preis wird überprüft. Bei einem Gas Preis ungleich null, wird die Transaktion nicht weiter betrachtet. Falls bei der Transaktion ein Gas Preis von null festgestellt worden ist, wird der Sender ermittelt, in diesem Fall der Account XX. Die Zähler des Accounts werden in der Liste **Accounts** aktualisiert. Der Account wird anschliessend an den DoS Algorithmus übergeben.

Im DoS Algorithmus wird die Anzahl getätigter gratis Transaktionen und das verbrauchte Gas ausgewer-

tet. Es wird geprüft, ob mit der neu erfassten Transaktion ein Grenzwert im aktuellen Reset-Intervall überschritten worden ist. Wenn das nicht der Fall ist, sind keine weiteren Aktionen nötig. Ist ein Grenzwert überschritten worden, wird dies für den entsprechenden Account in `Accounts` festgehalten. Zusätzlich wird eine Transaktion erstellt, die den Account von der Whitelist in Parity entfernt.

3.2.3 Command Pattern und Priority Queue

Für die Handhabung des Reset- und Revoke-Intervalls, wird ein Command-Pattern[54] verwendet. Die Commands werden mit einem Zeitstempel versehen. Dieser gibt an, wann die Methode `execute()` ausgeführt werden soll.

Für die Zeitgerechte Ausführung wird eine Priority-Queue[55] verwendet. In einem eigenen Prozess, wird im Sekundentakt geprüft, ob der Zeitpunkt für die Ausführung des anstehenden Commands gekommen ist.

Für das Zurücksetzen aller Accountparameter wird das `ResetAccountsCommand` verwendet.

Listing 3.12: `ResetAccountsCommand` um die Accountparameter zurückzusetzen

```
1  @Override
2  public void execute() {
3      log.info(new Date().toString());
4      this.setAllCountersToMax();
5      this.jsonDefaultSettingsHandler.getDefaultSettings()
6          .setTimestampLastReset(new Timestamp(System.
7              currentTimeMillis()));
8      this.jsonAccountHandler.writeAccountList();
9      this.jsonDefaultSettingsHandler.writeDefaultSettings();
10     DoSAlgorithm.getInstance().offerCommand(new
11         ResetAccountsCommand());
12 }
```

Der Codeausschnitt 3.12 zeigt die `execute` Methode des `ResetAccountsCommand`. Dieses Command wird am Ende eines Reset-Intervalls ausgeführt. Auf Zeile 4 bis 8, werden zuerst alle Accounts zurückgesetzt und der Zeitpunkt der Ausführung wird in den `DefaultSettings` gespeichert. Anschliessend werden die Dateien `AccountList.json` und `DefaultSettings.json` neu geschrieben.

Auf Zeile 10 wird rekursiv, das nächste `ResetAccountsCommand` erstellt und in die Priority Queue gelegt. Im Constructor des neuen Commands, wird der Zeitpunkt für dessen Durchführung mit den `DefaultSettings` berechnet.

3.2.4 Persistenz

Um die Datenpersistenz zu gewährleisten, wird die Whitelist des Transaktionsmanagers regelmässig in einer JSON-Datei gespeichert.

Dafür wird die Datei `AccountList.json` verwendet. Diese muss im selben Verzeichnis wie die `.jar` Datei des Transaktionsmanagers liegen.

In dieser Datei werden alle Accounts, die auf der Whitelist sind, erfasst. Bei jedem Account können die individuell konfigurierbaren Parameter gesetzt werden.

Nach einem Programmstop wird die Datei ausgelesen. Alle Accounts werden geladen. Wo nötig, die individuellen Parameter gesetzt. Für alle Parameter die nicht gesetzt sind, werden die konfigurierten Standardwerte verwendet, siehe 3.2.5. Parameter, für die der Standardwert verwendet werden soll, können entweder mit Wert `null` erfasst oder ganz weggelassen werden.

Die Datei ist als Array von Accounts aufgebaut. Für jeden Account sind folgende Felder vorhanden:

address Die Adresse des Accounts. Das ist der einzige Parameter, der zwingend definiert sein muss.

transactionLimit Definiert die maximale Anzahl gratis Transaktionen, die ein Account in einem Reset-Intervall tätigen darf.

remainingTransactions Mit diesem Feld wird bestimmt, über wie viele gratis Transaktionen ein Account noch verfügt. Dieser Wert wird vom Transaktionsmanager gepflegt und sollte nicht erfasst oder geändert werden.

gasLimit Definiert die maximale Anzahl gratis Gas, die ein Account während eines Reset-Intervalls verbrauchen darf.

remainingGas Mit diesem Feld wird bestimmt, über wie viel gratis Gas ein Account noch verfügt. Dieser Wert wird vom Transaktionsmanager gepflegt und sollte nicht erfasst oder geändert werden.

revokeTime Definiert, wie lange ein Account bei einem Vergehen von der Whitelist suspendiert wird. Die Angabe ist in Minuten.

deleteMe Dieser Parameter wird nur verwendet, wenn ein Account permanent von der Whitelist gelöscht werden soll. Nach dem der Account von der Whitelist entfernt wurde, wird der Eintrag auch aus `AccountList.json` gelöscht.

timeStamp Definiert, wann ein Account wieder in die Whitelist aufgenommen werden soll. Dieser Wert wird vom Transaktionsmanager gepflegt und sollte nicht erfasst oder geändert werden.

Wenn der Transaktionsmanager die Datei neu schreibt, werden alle Parameter geschrieben. Das bedeutet, Parameter die nicht gesetzt waren, sind nun mit einem Wert `null` erfasst. Ausnahme sind `gasUsedCounter` und `transactionCounter`, diese werden mit der entsprechenden Limite initialisiert.

Falls ein Account zum Zeitpunkt des Programmstops von gratis Transaktionen suspendiert war, ist der `timeStamp` gesetzt. Sollte der Zeitpunkt in der Vergangenheit liegen, wird der Account wieder

zertifiziert. Andernfalls wird er erneut der Priority-Queue hinzugefügt und bleibt bis zum vorgesehenen Zeitpunkt suspendiert.

Automatische Suspendierungen von Transaktionsmanager sind immer temporär. Soll ein Account permanent von der Whitelist gelöscht werden, muss dies manuell in der Datei `AccountList.json` mit dem Parameter `deleteMe` gemacht werden.

Mehr zur Konfigurationsdatei und Beispiele sind im nachfolgenden Abschnitt, 3.2.5, zu finden.

3.2.4.1 Beispiele

Erfasst werden 3 Accounts. Vor Programmstart könnte die Datei so aussehen:

Listing 3.13: `AccountList.json` für die Erfassung von 4 Accounts. Datei vor Programmstart

```
1  [  
2    {  
3      "address": "0xaf02DcCdEf3418F8a12f41CB4ed49FaAa8FD366b",  
4      "transactionLimit": 8,  
5      "revokeTime": 15,  
6      "gasLimit": 600000000  
7    },  
8    {  
9      "address": "0xf13264C4bD595AEbb966E089E99435641082ff24"  
10   },  
11   {  
12     "address": "0x00a329c0648769A73afAc7F9381E08FB43dBEA72",  
13     "revokeTime": 25,  
14     "gasLimit": 9999999999999999  
15   },  
16   {  
17     "address": "0xf1326lasdfaöSLDFJDLNd23n\DD35641082ff24"  
18     "revokeTime": 2  
19   }  
20 ]  
21 ]
```

Das Listing 3.13 zeigt die Erfassung von 4 Accounts. Der erste verwendet überall individuelle Parameter. Der zweite Account hat keine Parameter und verwendet daher überall die Standardwerte. Beim dritten und vierten Account ist gezeigt, dass auch ganz selektiv, einzelne Parameter gesetzt werden können.

Sobald die Liste von Accounts unter Listing 3.13, eingelesen und verarbeitet wurde, wird sie vom Transaktionsmanager neu geschrieben. Das Resultat ist von den konfigurierten Standardwerten abhängig, dazu mehr im folgenden Kapitel 3.2.5. Unter der Annahme, dass der Standardwert für Transaktionen

auf 5 und für Gas auf 500000000 konfiguriert ist, sieht die Datei folgendermassen aus:

Listing 3.14: AccountList.json mit 4 Accounts. Datei nach Programmstart

```
1  [  
2    {  
3      "address": "0xaf02DcCdEf3418F8a12f41CB4ed49FaAa8FD366b",  
4      "remainingTransactions": 8,  
5      "transactionLimit": 8,  
6      "revokeTime": 15,  
7      "gasLimit": 600000000,  
8      "remainingGas": 600000000,  
9      "deleteMe": false,  
10     "timeStamp": null  
11   },  
12   {  
13     "address": "0xf13264C4bD595AEbb966E089E99435641082ff24",  
14     "remainingTransactions": 5,  
15     "transactionLimit": null,  
16     "revokeTime": null,  
17     "gasLimit": null,  
18     "remainingGas": 500000000,  
19     "deleteMe": false,  
20     "timeStamp": null  
21   },  
22   {  
23     "address": "0x00a329c0648769A73afAc7F9381E08FB43dBEA72",  
24     "remainingTransactions": 5,  
25     "transactionLimit": null,  
26     "revokeTime": 25,  
27     "gasLimit": 9999999999999999,  
28     "remainingGas": 9999999999999999,  
29     "deleteMe": false,  
30     "timeStamp": null  
31   },  
32   {  
33     "address": "0xf1326lasdfaöS LDFJDLN d23n lDD35641082ff24",  
34     "transactionCounter": 5,  
35     "transactionLimit": null,  
36     "revokeTime": 2,  
37     "gasLimit": null,  
38     "remainingGas": 500000000,  
39     "deleteMe": false,  
40     "timeStamp": null  
41   }  
42 ]
```

Das Listing 3.14 zeigt die Datei `AccountList.json` nachdem der Transaktionsmanager gestartet und die initiale Datei verarbeitet wurde. Überall wo ein Standardwert verwendet wird, ist ein Wert von **null** gesetzt worden.

3.2.5 Konfiguration

Der Transaktionsmanager kann mit der Datei `DefaultSettings.json` konfiguriert werden. Diese muss im selben Verzeichnis wie die `.jar` Datei des Transaktionsmanagers liegen.

Folgende Parameter sind unterstützt:

connectionAddress URL um sich mit der Blockchain zu verbinden.

resetInterval Definiert in welchem Intervall die Zähler von allen Accounts zurückgesetzt werden. Wird in Minuten angegeben.

defaultTXLimit Der Standardwert für die gratis Transaktionslimite pro Account.

defaultGasLimit Der Standardwert für die gratis Gaslimite pro Account.

defaultRevokeTime Der Standardwert für die Dauer der Suspendierung von der Whitelist bei einem Vergehen.

certifierAddress Die Adresse des Certifiers. Er wird automatisch beim Deployment des Certifiers gesetzt. Dieser Parameter sollte nicht bearbeitet werden.

nameRegistryAddress Die Adresse der Name Registry. Hier muss dieselbe Adresse wie in der Blockchainspezifikation verwendet werden. Siehe 3.1.1.2.

timestampLastReset Hier wird der Zeitpunkt des letzten Reset-Intervalls vermerkt. Dieser Wert wird vom Transaktionsmanager gepflegt und sollte nicht bearbeitet werden.

Eine weitere, sehr zentrale Konfiguration ist der Account des Transaktionsmanagers. Dieser wird für jede Interaktion mit der Blockchain verwendet. Um zu verhindern, dass dieser fix im Code eingetragen werden muss, wird er ebenfalls aus einer Datei geladen. Hier wird die Datei `TransaktionsManagerAccount.json` verwendet. Diese muss im selben Verzeichnis wie die `.jar` Datei des Transaktionsmanagers liegen.

Die Datei hat nur einen einzigen Parameter:

Listing 3.15: `TransaktionsManagerAccount.json` mit einem privaten Schlüssel

```
1 {  
2   "privateKey": "0  
    x4d5db4107d237df6a3d58ee5f70ae63d73d7658d4026f2eefd2f204c81682cb7"  
3 }
```

Die unter 3.15 gezeigte Datei beinhaltet den Account für den Transaktionsmanager. Dieser sollte auf keinen Fall öffentlich gemacht werden. Der hier gezeigte Schlüssel ist der Standardaccount im Entwicklungsmodus von Parity.

3.2.5.1 Beispiel

Anbei ein Beispiel von `DefaultSettings.json` für eine frisch aufgesetzte Blockchain.

Listing 3.16: `DefaultSettings.json`

```
1 {
2   "connectionAddress": "http://jurijnas.myqnapcloud.com:8545/",
3   "resetInterval": 10,
4   "defaultTxLimit": 5,
5   "defaultGasLimit": 500000000,
6   "defaultRevokeTime": 10,
7   "certifierAddress": null,
8   "nameRegistryAddress": "0x00000000000000000000000000000000000000000000000000000000000000001337",
9   "timestampLastReset": null
10
11 }
```

Unter Listing 3.16 ist die Datei `DefaultSettings.json` abgebildet. So sieht die Datei aus, bevor der Transaktionsmanager den Certifier deployed hat.

Listing 3.17: `DefaultSettings.json`

```
1 {
2   "connectionAddress": "http://jurijnas.myqnapcloud.com:8545/",
3   "resetInterval": 10,
4   "defaultTxLimit": 5,
5   "defaultGasLimit": 500000000,
6   "defaultRevokeTime": 10,
7   "certifierAddress": "0xee35211c4d9126d520bbfeaf3cfee5fe7b86f221",
8   "nameRegistryAddress": "0x00000000000000000000000000000000000000000000000000000000000000001337",
9   "timestampLastReset": "Mar 5, 2020 1:53:50 AM"
10
11 }
```

Listing 3.17 zeigt die Datei `DefaultSettings.json` nachdem der Certifier deployed wurde. Dessen Adresse ist nun eingetragen. Weiter ist ein Zeitstempel gesetzt.

3.2.6 Initialisierung

Wenn die Blockchain frisch aufgesetzt wird, sind initial keine gratis Transaktionen möglich. Der dafür nötige Certifier ist weder deployed noch bei der Name Registry registriert.

Mit der Klasse `Init` wird die Blockchain für den Betrieb vorbereitet. Folgende Schritte werden dabei ausgeführt:

1. Instanziierung der Name Registry

Mit der Wrapperklasse für die Name Registry und deren Adresse auf der Blockchain wird ein Objekt für die Interaktion mit dem eigentlichen Smart Contract erzeugt.

2. Deployment des Certifiers

Die Wrapperklasse für den Certifier wird instanziiert und auf die Blockchain deployed.

3. Registrierung des Certifiers bei der Name Registry

Die Adresse des Certifiers wird bei der Name Registry registriert.

4. Zertifizierung von Account für Transaktionsmanager

Der Account des Transaktionsmanagers wird für gratis Transaktionen berechtigt.

Siehe 3.1.4.1 für mehr Informationen zu Wrapperklassen, Deployment und Registrierung von Accounts.

Die Adresse der Name Registry wird aus der Konfigurationsdatei gelesen, siehe 3.2.5. Hier muss beachtet werden, dass diese mit der Adresse in der Blockchainspezifikation, siehe 3.1.1.2, übereinstimmt. Sobald der Certifier deployed ist, wird dessen Adresse in die Konfigurationsdatei geschrieben. Beim Start der Überwachung von der Blockchain, wird die Adresse ausgelesen und der Certifier instanziiert. Der Account für den Transaktionsmanager wird in der Blockchainspezifikation definiert, siehe 3.1.1.2. Dieser Account wird verwendet um Änderungen an der Whitelist vorzunehmen. Aus Sicherheitsgründen wird der private Schlüssel von diesem Account nicht in den Quellcode des Transaktionsmanagers geschrieben. Er wird beim Start der Applikation aus einer eigenen Datei gelesen. Hier muss beachtet werden, dass diese Datei bei der Verwendung von einer Versionierungssoftware nicht inkludiert wird.

3.2.7 Run

Mit der Klasse `Run` wird der Transaktionsmanager für den regulären Betrieb gestartet. Voraussetzung ist, dass der Certifier deployed und registriert ist, siehe 3.2.6.

Beim Aufruf der Klasse `Run` werden folgende Schritte ausgeführt:

1. Laden der DefaultSettings

Die Datei `DefaultSettings.json` wird ausgelesen und das Objekt `JsonDefaultSettings` damit befüllt.

2. Laden der Accountliste

Die Datei `AccountList.json` wird ausgelesen. Die Accounts werden mit den konfigurierten Parameter instanziiert.

3. Instanziierung von Name Registry und Certifier

Mit der jeweiligen Adresse, wird die Name Registry und der Certifier instanziiert.

4. Die Klasse `ChainInteractions` wird instanziiert.

5. Zertifizierung aller geladenen Accounts

Alle geladenen Accounts werden geprüft. Sollte ein Account über einen Zeitstempel verfügen,

wird geprüft ob er suspendiert werden muss. Ansonsten werden alle Accounts zertifiziert.

6. Instanziierung der Klasse `SubscriptionTX`

Eine Subscription, die eigentliche Überwachung der Blockchain, wird gestartet. Das Command `ResetAccountsCommand` fügt sich rekursiv in die Priority Queue ein.

Sobald der Transaktionsmanager über eine Kommandozeile gestartet wurde, wird der Benutzer über aktuellen Vorgänge mit Lognachrichten informiert.

3.2.8 Tests

Um die Lauffähigkeit des Transaktionsmanagers zu verifizieren, sind diverse Tests ausgeführt worden. Mit Unit Tests ist die Funktionalität von einzelnen Aspekten geprüft worden. Dazu gehören unter anderem, isolierte Transaktionen ausführen, Lesen und Schreiben von Dateien.

Mit einem Leistungstest ist untersucht worden, wie sich der Transaktionsmanager unter Belastung verhält.

Mit den Abnahmetests ist sichergestellt, dass die Applikation als Ganzes, wie gewünscht funktioniert.

3.2.8.1 Unit Tests

Die Funktionalität von zentralen Methoden sind mit Unittests geprüft worden.

Alle Unit Tests sind erfolgreich. Sie sind im Anhang unter 6.6.2.1 verlinkt. Die Dateien `AccountList.json` und `DefaultSettings.json` welche für die Tests verwendet werden, sind im Ordner `src/test/java/resources/testFilesForWhitelist` abgelegt.

3.2.8.2 Leistungstest

Um die Leistung des Transaktionsmanagers zu testen, werden verschieden hohe Transaktionslimiten auf einem Account geprüft.

Im Test wird mit einer Schleife, in sehr rascher Folge, gratis Transaktionen getätigt. Am Ende wird geprüft, ob der Account von der Whitelist entfernt worden ist. Folgende Kombination von gratis Transaktionen und Transaktionslimite ist getestet worden:

Tabelle 3.1: Leistungstest mit gratis Transaktionen

# TX gemacht	Limite	# TX verarbeitet	Revoked
4	5	4	Nein

# TX gemacht	Limite	# TX verarbeitet	Revoked
5	5	5	Nein
6	5	6	Ja
10	5	10	Ja
100	5	100	Ja
500	450	500	Ja
1000	450	ca 980	Ja
2000	450	ca 980	Ja

Die Tabelle 3.1 zeigt die Testparameter und das Resultat. Mit „# TX gemacht“ wird angegeben, wie viele Transaktionen in der Schleife erstellt werden. „Limite“ definiert die Transaktionslimite des Accounts pro Reset-Intervall. In der Spalte „# TX verarbeitet“ ist aufgeführt, wie viele Transaktionen gratis verarbeitet worden sind. Die Spalte „Revoked“ zeigt, ob der Account am Ende des Tests von der Whitelist entfernt wurde.

Die ersten drei Tests prüfen um den Grenzwert. Diese verhalten sich wie erwartet. Fünf gratis Transaktionen dürfen ohne Konsequenzen durchgeführt werden. Nach der sechsten wird der Account von der Whitelist entfernt.

Die folgenden Tests sind erfolgreich, da der Account am Ende von der Whitelist entfernt ist. Allerdings ist festgestellt worden, dass der Transaktionsmanager nicht sofort reagiert. Bis der Transaktionsmanager reagiert, können ungehindert gratis Transaktionen gemacht werden.

Die Verzögerung ist weiter untersucht worden. Dabei wurde festgestellt, dass vom Zeitpunkt der ersten Transaktion, es immer zwischen 14 und 15 Sekunden dauert, bis diese in der Subscription registriert wird. In dieser Zeit konnten mir einer **for**-Schleife, ca 760 Transaktionen abgesetzt werden.

Sobald die Subscription die getätigten Transaktionen erhält, ist die Limite bereits überschritten worden. Während die Subscription die anfallenden Transaktionen abarbeitet, werden in der **for**-Schleife laufend neue abgesetzt.

Ab dem Moment, wo die erste Transaktion von der Subscription registriert wird, dauert es nochmals zwischen 14 und 15 Sekunden, bis der Account revoked ist.

Dieses Verhalten ist mit der Zeit zwischen Blöcken zu erklären. Sobald der Node die erste Transaktion erhält, wird ein neuer Block erstellt und befüllt. Nach 14-15 Sekunden wird dieser an die Blockchain übermittelt. Es wird ein neuer Block befüllt. Während dieser Zeit hat der Transaktionsmanager registriert, dass die Limite erreicht worden ist. Er sendet ebenfalls eine Transaktion an den Node.

Der Node hat bereits sehr viele gratis Transaktionen, deren Verarbeitung noch ausstehend sind. Die Transaktion vom Transaktionsmanager wird prioritär behandelt, da es keine gratis Transaktion ist. Nach

weiteren 14 bis 15 weiteren Sekunden, erreicht der zweite Block die Blockchain. Erst jetzt, kann das Revoke verarbeitet werden. Ab diesem Zeitpunkt, akzeptiert der Node keine weiteren gratis Transaktionen von diesem Account.

3.2.8.3 Abnahmetest

Hier wird die Applikation als Ganzes geprüft.

Um die funktionalen Aspekte zu prüfen führt ein Account auf der Whitelist gratis Transaktionen aus. Anschliessend werden die Zähler des Accounts überprüft. Am Ende des Reset-Intervalls wird geprüft, ob die Zähler zurückgesetzt werden. So wird geprüft ob die Subscription, der DoS Algorithmus, das Reset-Intervall und die Priority Queue zusammen funktionieren.

Es ist auch geprüft worden, ob der registrierte Certifier mit einer eigenen Version ersetzt werden kann. Damit könnte ein Benutzer den Transaktionsmanager umgehen. Der Schutz vor einer DoS Attacke wäre nicht mehr gewährleistet.

Beide Tests sind erfolgreich, der Transaktionsmanager verhält sich wie gewünscht.

Weiter wurden manuell alle Abnahmekriterien, siehe 6.3, geprüft. Das Abnahmetestprotokoll ist unter Kapitel 6.4 aufgeführt.

3.2.8.4 Schlussfolgerung

Die Leistungstests haben gezeigt, dass der Transaktionsmanager direkt von den Blöcken abhängig ist. Je nach Grösse und Frequenz von neuen Blöcken müssen die Parameter von Accounts und DoS Algorithmus entsprechend angepasst werden.

Mit den verwendeten Einstellungen und einer Transaktionslimite von 450, wird ein Angriff nach ungefähr 30 Sekunden unterbunden.

Tiefere Limiten ändern an diesem Verhalten nichts. Der Angreifer hat dennoch die Möglichkeit einen Block komplett zu füllen. Eine mögliche Strategie wäre, die Limite an der Kapazität von einem Block anzugleichen. Diese ist abhängig vom maximalen Gas pro Block und der Frequenz, in der Blöcke erstellt werden.

Ein Node priorisiert Transaktionen anhand des verwendeten Gas Preises. Um sicherzustellen, dass die Transaktionen des Transaktionsmanagers beim Node prioritär behandelt werden, muss ein gültiger Gas Preis verwendet werden. Es ist daher nötig, den Account des Transaktionsmanagers mit einem hohen Guthaben zu initialisieren.

4 Fazit

Für die FHNW ist eine Ethereumblockchain aufgesetzt worden. Als Client wird Parity verwendet. Zurzeit der einzige Client, welcher einer definierten Benutzergruppe kostenneutrale Transaktionen erlaubt. Diese Benutzergruppe ist eine Gefahr für die Blockchain. Gratis Transaktionen können für eine Denial of Service Attacke verwendet werden.

Um diese Gefahr einzudämmen ist ein Blockchain Transaktionsmanager konzipiert und entwickelt worden. Dieser ermöglicht eine Überwachung von getätigten gratis Transaktionen. Wenn nötig können diese auch eingeschränkt werden.

Für jeden Account der privilegierten Benutzergruppe, können individuelle Parameter für die Nutzung der gratis Transaktionen gesetzt werden. So kann pro Account festgelegt werden, wie viele gratis Transaktionen und gratis Gas pro Reset-Intervall verfügbar sind. Sobald eine dieser Limiten überschritten wird, wird das Privileg auf gratis Transaktionen für diesen Account aufgehoben. Wie lange eine solche Suspendierung dauert, kann ebenfalls pro Account festgelegt werden. Um die Benutzerfreundlichkeit auch bei vielen Accounts zu gewährleisten, können für diese Parameter auch Standardwerte gesetzt werden.

Am Ende eines Reset-Intervalls, werden die Zähler von allen Accounts zurückgesetzt. Das Reset-Intervall kann konfiguriert werden. Es ist der einzige Parameter, der nicht für jeden Account einzeln gesetzt werden kann, sondern für alle gleich ist.

Für das Reset-Intervall und die Suspendierung von einem Account, wird eine Priority-Queue in Kombination mit einem Command-Pattern verwendet. Jedes Command verfügt über einen Zeitstempel für den Ausführungszeitpunkt. Das Command für das Reset-Intervall, fügt sich rekursiv in die Priority-Queue ein.

Die für gratis Transaktionen berechtigten Accounts und Einstellungen des Transaktionsmanagers werden über zwei JSON-Dateien eingelesen. Diese Dateien werden weiter für die Sicherstellung der Datenpersistenz verwendet. Jedes Mal, wenn ein Command mit einem Zeitstempel erstellt wird, wird diese Information in einer Datei gespeichert.

Mit der Verwendung von Zeitstempel kann nach einem Unterbruch der letzte Zustand wiederhergestellt werden. Sollte die Suspendierung von einem Account während der Ausfallszeit abgelaufen sein, wird das beim Neustart ermittelt.

Mit dem entwickelten Transaktionsmanager sind gratis Transaktionen möglich, ohne dass die Sicherheit der Blockchain kompromittiert wird. Das ermöglicht der FHNW, ihren Studierenden und

Mitarbeitenden, eine sichere und kostenneutrale Lernumgebung anzubieten. Damit können alle Ziele der Projektvereinbarung erreicht werden.

Die Leistungstests haben gezeigt, dass der Transaktionsmanager nicht sofort reagieren kann. Er ist an die publizierten Blöcke gebunden. Die Wahl der Limiten sollte daher immer in Abhängigkeit der Blockgrösse und deren Frequenz gewählt werden.

Mit der Verwendung von einem Gas Preis grösser null beim Transaktionsmanager ist sichergestellt, dass diese Transaktionen prioritär behandelt werden. Die Tests haben gezeigt, dass die Priorisierung des Nodes auch während einer DoS Attacke funktionieren.

Zu Beginn ist eine Implementation des Transaktionsmanagers auf der Blockchain mittels Smart Contracts angestrebt worden. Dieser Ansatz bedingt jedoch eine Anpassung von Parity. Aktuell hält Parity eine Liste von Accounts, die für gratis Transaktionen berechtigt sind. Um den Transaktionsmanager völlig dezentral zu realisieren, müsste die Whitelist von Parity so angepasst werden, dass nicht nur Accounts, sondern Verbindungen geprüft werden. So kann sichergestellt werden, dass gratis Transaktionen über den Transaktionsmanager abgewickelt werden müssen.

Der Umfang und Komplexität dieser Anpassungen sind im gegebenen Zeitrahmen als nicht realisierbar eingeschätzt worden. Daher haben wir beschlossen, den Transaktionsmanager als Java-Applikation zu realisieren. Diese Version des Transaktionsmanagers ist eine zentrale Autorität. Das entspricht nicht dem dezentralitäts Prinzip einer Blockchain. Der Transaktionsmanager kann nur gratis Transaktionen einschränken. Alle anderen Aspekte der Blockchain bleiben dezentral. Die Verletzung der Prinzipien ist daher geringfügig.

Das Aufsetzen einer Blockchain mit Parity ist unterschätzt worden. Bei der Konfiguration der Blockchain mussten wir feststellen, dass die Dokumentation sehr oft unvollständig oder veraltet ist. Dadurch hat sich die Bereitstellung der Blockchain stark verzögert. Aufgrund der Verzögerung musste der Fokus, stärker als geplant, auf die Kernfunktionalität gerichtet werden. Dadurch fehlte Zeit bei der Implementaion von Zusatzfunktionen.

Der Transaktionsmanager hat ein grosses Potential für Erweiterungen. Wenn die Blockchain frisch aufgesetzt wird, könnte das Deployment und die Registrierung des Certifiers automatisiert werden. Analog zur Name Registry kann der Certifier direkt in der Blockchainspezifikation deployed werden. Zusätzlich ist es möglich, den Constructor vom Certifier so anzupassen, dass er sich automatisch bei der Name Registry registriert. Dafür muss der `SimpleCertifier` und die Blockchainspezifikation angepasst werden.

Mit den Erkenntnissen aus den Leistungstests könnten die Parameter überarbeitet werden. Unter Berücksichtigung der Blockchainspezifikation könnten optimale Vorschläge für die Limiten berechnet werden. Das würde die Effizienz und die Benutzerfreundlichkeit des Transaktionsmanagers erhöhen. Um die Benutzerfreundlichkeit zu erhöhen, könnte eine leserliche Zusammenstellung aller Ereignisse

erstellt werden. Diese könnte manuell abgerufen oder automatisch via Email an die Betreiber des Transaktionsmanagers gesendet werden. Für eine Zustellung via Mail muss ein Mailserver zur Verfügung stehen.

Mit einer Validierung der Input-Dateien und einer entsprechenden Meldung, könnte mehr Benutzerfreundlichkeit erreicht werden.

5 Quellenverzeichnis

- [1] „University of Applied Sciences and Arts Northwestern Switzerland“, 2019. [Online]. Verfügbar unter: <https://www.fhnw.ch/>.
- [2] M. Inc., „What is Gas | MyEtherWallet Knowledge Base“, 2018. [Online]. Verfügbar unter: <https://kb.myetherwallet.com/en/transactions/what-is-gas/>.
- [3] Ethereum, „Home | Ethereum“, 2019. [Online]. Verfügbar unter: <https://www.ethereum.org/>.
- [4] „Blockchain - Wikipedia“, 2019. [Online]. Verfügbar unter: <https://en.wikipedia.org/wiki/Blockchain>.
- [5] „Denial-of-service attack - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Denial-of-service_attack.
- [6] „Whitelisting - Wikipedia“, 2020. [Online]. Verfügbar unter: <https://en.wikipedia.org/wiki/Whitelisting>.
- [7] „Smart Contract Definition“, 2019. [Online]. Verfügbar unter: <https://www.investopedia.com/terms/s/smart-contracts.asp>.
- [8] „Genesis block - Bitcoin Wiki“, 2019. [Online]. Verfügbar unter: https://en.bitcoin.it/wiki/Genesis_block.
- [9] „Peer-to-peer - Wikipedia“, 2019. [Online]. Verfügbar unter: <https://en.wikipedia.org/wiki/Peer-to-peer>.
- [10] „What Is a Blockchain Consensus Algorithmen | Binance Academy“, 2019. [Online]. Verfügbar unter: <https://www.binance.vision/blockchain/what-is-a-blockchain-consensus-algorithm>.
- [11] „Bitcoin - Wikipedia“, 2019. [Online]. Verfügbar unter: <https://en.wikipedia.org/wiki/Bitcoin>.
- [12] „Nick Szabo - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Nick_Szabo.
- [13] „Smart Contracts for Alpiq | ETH Zürich“, 2019. [Online]. Verfügbar unter: <https://ethz.ch/en/industry-and-society/industry-relations/industry-news/2019/04/smart-contract-for-alpiq.html>.
- [14] „Ethereum Name Service“, 2020. [Online]. Verfügbar unter: <https://ens.domains/#>.

- [15] „Domain Name System - Wikipedia“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Domain_Name_System.
- [16] „Wei“, 2019. [Online]. Verfügbar unter: <https://www.investopedia.com/terms/w/wei.asp>.
- [17] S. Fontaine, „Understanding Bytecode on Ethereum - Authereum - Medium“, 2019. [Online]. Verfügbar unter: <https://medium.com/authereum/bytecode-and-init-code-and-runtime-code-oh-my-7bcd89065904>.
- [18] K. Tam, „Transactions in Ethereum - KC Tam - Medium“, 2019. [Online]. Verfügbar unter: <https://medium.com/@kctheservant/transactions-in-ethereum-e85a73068f74>.
- [19] Y. Riady, „Signing and Verifying Ethereum Signatures - Yos Riady“, 2019. [Online]. Verfügbar unter: <https://yos.io/2018/11/16/ethereum-signatures/>.
- [20] „Remote procedure call - Wikipedia“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Remote_procedure_call.
- [21] „<https://keccak.team/>“, 2019. [Online]. Verfügbar unter: Keccak%20Team.
- [22] „Elliptic Curve Digital Signature Algorithm - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.
- [23] „SHA-3 - Wikipedia“, 2019. [Online]. Verfügbar unter: <https://en.wikipedia.org/wiki/SHA-3>.
- [24] P. Kasireddy, „How does Ethereum work, anyway? - Preethi Kasireddy - Medium“, 2019. [Online]. Verfügbar unter: <https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369>.
- [25] „Ethereum Series - Understanding Nonce - The Startup - Medium“, 2019. [Online]. Verfügbar unter: <https://medium.com/swlh/ethereum-series-understanding-nonce-3858194b39bf>.
- [26] N. Jabes, „Nu Jabe’s answer to What is an Ethereum contract address? - Quora“, 2019. [Online]. Verfügbar unter: <https://www.quora.com/What-is-an-Ethereum-contract-address/answer/Nu-Jabes>.
- [27] „Crypto Wallet Types Explained | Binance Academy“, 2019. [Online]. Verfügbar unter: <https://www.binance.vision/blockchain/crypto-wallet-types-explained>.
- [28] M. Wachal, „What is a blockchain wallet? - SoftwareMill Tech Blog“, 2019. [Online]. Verfügbar unter: <https://blog.softwaremill.com/what-is-a-blockchain-wallet-bbb30fbf97f8>.
- [29] StellaBelle, „Cold Wallet Vs. Hot Wallet: What’s The Difference?“, 2019. [Online]. Verfügbar unter: <https://medium.com/@stellabelle/cold-wallet-vs-hot-wallet-whats-the-difference-a00d872aa6b1>.
- [30] M. Wright, „So many mobile wallets, so little differentiation - Argent - Medium“, 2019. [Online]. Verfügbar unter: <https://medium.com/argenthq/recap-on-why-smart-contract-wallets-are-the-future-7d6725a38532>.

- [31] E. Conner, „smart Wallets are Here - Gnosis“, 2019. [Online]. Verfügbar unter: <https://blog.gnosis.pm/smart-wallets-are-here-121d44519cae>.
- [32] D. Labs, „Why Dapper is a smart contract wallet - Dapper Labs - Medium“, 2019. [Online]. Verfügbar unter: <https://medium.com/dapperlabs/why-dapper-is-a-smart-contract-wallet-ef44cc51cfa5>.
- [33] „Crypto Bites: Chat with Ethereum founder Vitalik Buterin“, 2019. [Online]. Verfügbar unter: https://www.youtube.com/watch?v=u-i_mTwL-FI&feature=emb_logo.
- [34] R. Greene und M. N. Johnstone, „An investigation into a denial of service attack on an ethereum network“, 2018. [Online]. Verfügbar unter: <https://ro.ecu.edu.au/cgi/viewcontent.cgi?article=1219&context=ism>.
- [35] „ethereum/yellowpaper: The Yellow Paper: Ethereum’s formal specification“, 2019. [Online]. Verfügbar unter: <https://github.com/ethereum/yellowpaper>.
- [36] go-ethereum, „Go Ethereum“, 2019. [Online]. Verfügbar unter: <https://geth.ethereum.org/>.
- [37] P. Technologies, „Blockchain Infrastructure for the Decentralised Web | Parity Technologies“, 2019. [Online]. Verfügbar unter: <https://www.parity.io>.
- [38] „<https://github.com/ethereum/aleth>“, 2019. [Online]. Verfügbar unter: <https://github.com/ethereum/aleth>.
- [39] „ethereum/trinity: The Trinity client for the Ethereum network“, 2019. [Online]. Verfügbar unter: <https://github.com/ethereum/trinity>.
- [40] „Rust Programming Language“, 2019. [Online]. Verfügbar unter: <https://www.rust-lang.org/>.
- [41] „Parity Name Registry - Parity Tech Documentation“, 2020. [Online]. Verfügbar unter: <https://wiki.parity.io/Parity-name-registry.html>.
- [42] „Permissioning - Parity Tech Documentation“, 2020. [Online]. Verfügbar unter: <https://wiki.parity.io/Permissioning#how-it-works-3>.
- [43] „Common Patterns - Solidity 0.4.24 documentation“, 2020. [Online]. Verfügbar unter: <https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html#restricting-access>.
- [44] „blockchain - What exactly is an Ethereum client and what clients are there?“, 2020. [Online]. Verfügbar unter: <https://ethereum.stackexchange.com/questions/269/what-exactly-is-an-ethereum-client-and-what-clients-are-there>.
- [45] „Web3j SDK - Where Java meets the Blockchain“, 2020. [Online]. Verfügbar unter: <https://www.web3labs.com/web3j>.
- [46] P. Technologies, „Releases - paritytech/parity-ethereum“, 2020. [Online]. Verfügbar unter: <https://github.com/paritytech/parity-ethereum/releases>.

- [47] „TOML - Wikipedia“, 2020. [Online]. Verfügbar unter: <https://en.wikipedia.org/wiki/TOML>.
- [48] „JSON - Wikipedia“, 2020. [Online]. Verfügbar unter: <https://en.wikipedia.org/wiki/JSON>.
- [49] „Remix - Ethereum IDE“, 2020. [Online]. Verfügbar unter: <https://remix.ethereum.org/>.
- [50] MetaMask, „MetaMask“, 2019. [Online]. Verfügbar unter: <https://metamask.io/>.
- [51] „Releases - web3j/web3j“, 2020. [Online]. Verfügbar unter: <https://github.com/web3j/web3j/releases>.
- [52] „Installing the Solidity Compiler – Solidity 0.6.4 documentation“, 2020. [Online]. Verfügbar unter: <https://solidity.readthedocs.io/en/develop/installing-solidity.html>.
- [53] „Getting Starte - web3j latest documentation“, 2020. [Online]. Verfügbar unter: https://web3j.readthedocs.io/en/latest/getting_started.html#filters.
- [54] „Command pattern - Wikipedia“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Command_pattern.
- [55] „PriorityQueue (Java Platform SE 7)“, 2020. [Online]. Verfügbar unter: <https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html>.

6 Anhang

6.1 Glossar

Begriff	Bedeutung
Account	Zugangsberechtigung zu einem zugangsbeschränkten IT-System. Zugrunde liegt der geheime Schlüssel. Dieser dient der Authentifikation.
Adresse	40 Zeichen langer String. Wird aus dem geheimen Schlüssel abgeleitet. Dient zur Identifizierung eines Accounts.
Aleth	Ein Blockchainclient.
Bitcoin	Eine Kryptowährung.
Blockchain	Eine Blockchain ist eine kontinuierlich erweiterbare Liste von Datensätzen, „Blöcke“ genannt, die mittels kryptographischer Verfahren miteinander verkettet sind. Jeder Block enthält dabei typischerweise einen kryptographisch sicheren Hash (Streuwert) des vorhergehenden Blocks, einen Zeitstempel und Transaktionsdaten.
Certifier	Smart Contract für die Verwaltung einer Whitelist in Parity.
Client	Ein Computer oder Programm.
Command Pattern	Standardisiertes Entwurfsmuster in der objektorientierten Programmierung.
Consens Algorithmus	Prozess, um in einem verteilten System, Einigkeit zum Zustand eines Datensatzes zu erhalten.
Decentralized application (DApp)	Applikation mit einem dezentralen Backend.

Begriff	Bedeutung
Denial of Service (DoS) Attacke	Ein Angriff, bei dem das Zielsystem überlastet wird. Die eigentlichen Benutzer des Systems können dadurch nicht mehr bedient werden.
Docker	Client für die Virtualisierung von Software.
Ether	Eine Kryptowährung
Ethereum	Globale open-Source Plattform für dezentralisierte Applikationen
FHNW	Fachhochschule Nordwestschweiz
Gas	Währung um die Kosten einer Transaktion in der Ethereumblockchain zu beziffern.
Gas Limit	Obergrenze für benötigtes Gas
Gas Preis	Preis der pro Einheit Gas bezahlt werden muss.
Geheimer Schlüssel	256 Bit langer generierter String. Dient zur eindeutigen Identifikation eines Accounts.
Genesisblock	Der erste Block in einer Blockchain. Er bildet das Fundament einer Blockchain
Geth	Ein Blockchain Client.
Hash	Umwandlung von einem Eingabeparameter in einen String fixer Länge.
Miner	Besitzer der benötigten Infrastruktur zur Validierung von Transaktionen.
Mining	Validierung von Transaktionen. Für erfolgreiche Validierung erhält der Miner eine Belohnung
Name Registry	Smart Contract in Parity. Zuständig für Mapping zwischen einem Namen und einer Adresse.
Node	Ein Computer in einem Netzwerk.
Reset-Intervall	Am Ende des Intervalls werden die Zähler aller Accounts zurückgesetzt. Definiert die Dauer zwischen zwei Resets.
Revoke-Intervall	Definiert die Dauer einer Suspendierung von der Whitelist.

Begriff	Bedeutung
Öffentlicher Schlüssel	Wird aus dem geheimen Schlüssel abgeleitet. Wird für die Signierung von Transaktionen benötigt.
Parity	Ein Ethereumclient.
Persistenz	Speicherung von Daten über einen Programmstopp hinweg.
Priority Queue	Datenstruktur um eine Warteschlange abzubilden. Die Elemente liegen sortiert vor.
Remote procedure call (RPC)	Aufruf einer Funktion auf einem entfernten System.
Smart Contract	Ein Stück Code auf der Blockchain.
Solidity	Programmiersprache für die Erstellung von Smart Contracts.
Trinity	Ein Blockchainclient.
TOML	Dateiformat für die Speicherung von Konfigurationsparametern.
Wallet	Digitales Portmonaie.
Wei	Untereinheit von Ether. 10^{18} Wei = 1 Ether
Whitelist	Liste mit berechtigten Accounts

6.2 Weitere Lösungsansätze

6.2.1 Super Smart Wallet

Es wird eine zentrale Smart Wallet entwickelt. Im Gegensatz zu LA 1, siehe Kapitel 2.4.1.1, wird nicht für jeden Benutzer eine Smart Wallet deployed, sondern nur eine einzige. Diese kann von allen Benutzern der Blockchain genutzt werden. Bei diesem Ansatz wird mit der in Absatz 2.3.1 beschriebenen Whitelist gearbeitet.

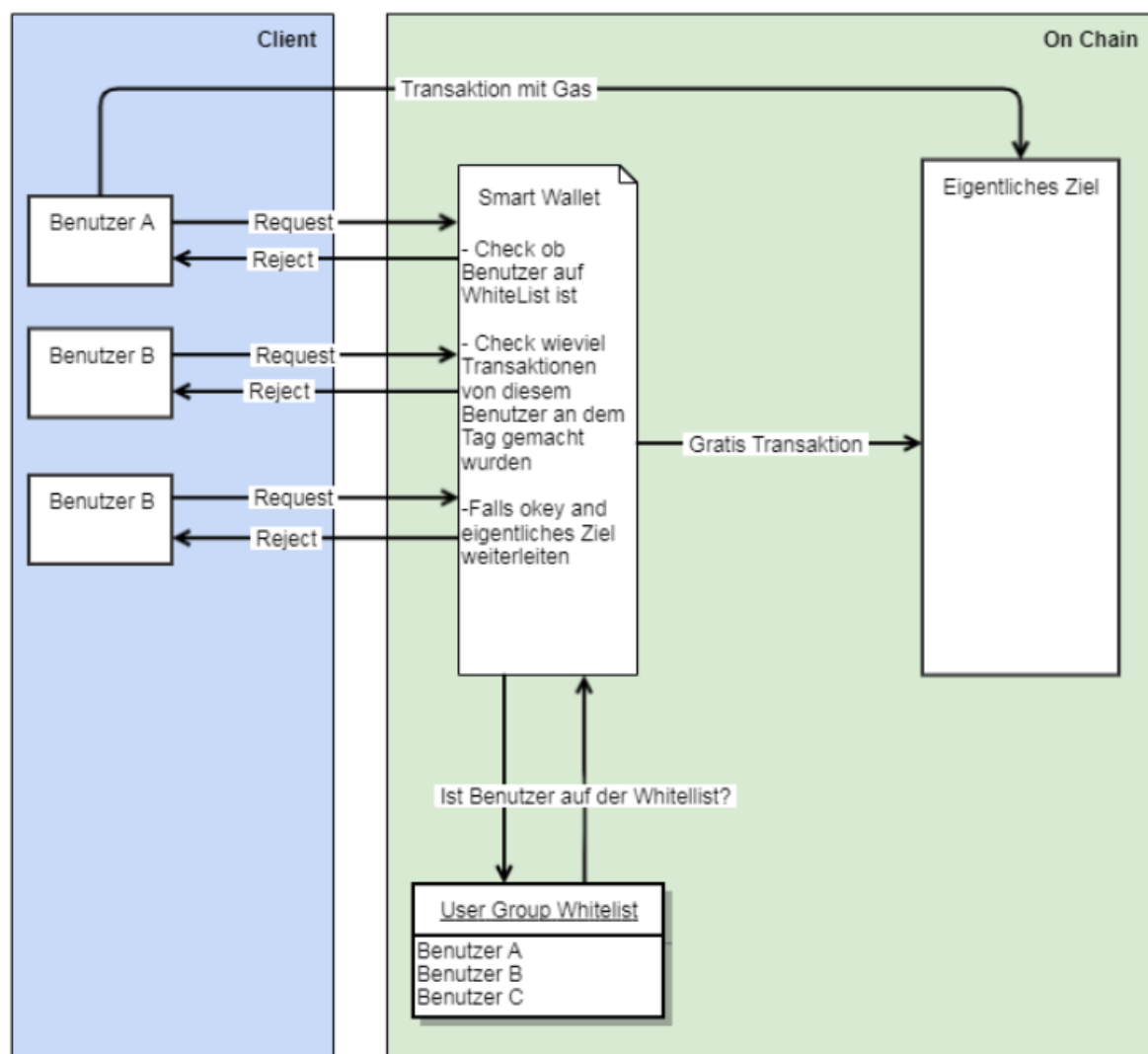


Abbildung 6.1: Super Smart Wallet

Die Smart Wallet verwaltet die Whitelist und den Schutzmechanismus gegen DoS Attacken. Das ist auf

Abbildung 6.1 ersichtlich. Wird eine DoS Attacke identifiziert, wird der entsprechende Account aus der Whitelist gelöscht.

6.2.1.1 Pro

Es existiert nur eine einzige Smart Wallet. Das Deployment ist somit weniger aufwändig. Falls eine Änderung am Code gemacht nötig ist, muss nur eine Smart Wallet neu deployed werden.

6.2.1.2 Contra

Bei diesem Ansatz ist die Machbarkeit unklar. Parity muss umgeschrieben werden, da nicht die Senderidentität der Smart Wallet genutzt werden muss, sondern die des Benutzeraccounts. Ebenfalls muss die Whitelist-Funktionalität von Parity angepasst werden, analog zu LA 1.

6.3 Abnahmekriterien

In der Tabelle 6.2 sind die Abnahmekriterien aufgeführt.

Tabelle 6.2: Abnahmekriterien

Nr.	Titel	Beschreibung
1.	Bezahlte Transaktionen für alle Accounts	Jeder Account kann Transaktionen durchführen mit einem gültigen Gas Preis durchführen
2.	Gratis Transaktionen nur für Accounts in Whitelist	Nur Accounts die in der Whitelist erfasst sind, können gratis Transaktionen durchführen.
3.	Liste von Accounts der Whitelist hinzufügen	Accounts die für die Whitelist zertifiziert werden sollen, stehen in der JSON Datei. Diese Accounts werden geladen und der Whitelist hinzugefügt.
4.	Accounts permanent von Whitelist entfernen	In der Datei <code>AccountList.json</code> wird ein Account für die Löschung markiert. Dafür wird der Parameter <code>deleteMe</code> verwendet. Der Account wird von der Whitelist und <code>AccountList.json</code> entfernt.
5.	Sperrung eines Accounts nach zu vielen Transaktionen	Ein Account der die Transaktionslimite überschreitet, wird temporär von der Whitelist gelöscht.
6.	Sperrung eines Accounts bei zu hohem gratis Gas Verbrauch	Ein Account der die Limite von gratis Gas überschreitet, wird temporär von der Whitelist gelöscht
7.	Revoke-Intervall	Beim Entfernen eines Accounts von der Whitelist, wird der Zeitstempel für das Revoke-Intervall korrekt gesetzt. Sobald der im Stempel definierte Zeitpunkt erreicht wird, wird der Account wieder auf die Whitelist gesetzt.
8.	Reset-Intervall	Die Zähler der Accounts werden nach Ablauf des Reset-Intervalls zurückgesetzt. Ein neues Command für das Reset-Intervall wird durch Rekursion in die Priority-Queue eingefügt.
9.	Zeitstempel für Reset-Intervall	Der Zeitstempel des Rest-Intervalls wird in der Datei <code>DefaultSettings.json</code> korrekt gesetzt.

Nr.	Titel	Beschreibung
10.	Standardwert für Revoke-Intervall	Der Standardwert des Revoke-Intervalls wird auf Accounts ohne individuelles Revoke-Intervall angewendet.
11.	Transaktionslimite pro Account	Die Transaktionslimite kann für jeden Account individuell gesetzt werden.
12.	Gaslimite pro User	Die Menge an verbrauchten gratis Gas kann für jeden Account individuell eingestellt werden
13.	Standardwerte für Limiten	Es können Standardwerte für Transaktionslimite und Gaslimite gesetzt werden.
14.	Registrierung Certifier	Die Registrierung des Certifiers kann nicht überschrieben werden

6.4 Report der Abnahmetests

Der Report zu den durchgeführten Abnahmetests. Jeder Test deckt mindestens ein Abnahmekriterium (AK) ab.

6.4.1 Abnahmetest 1

AK Nr.: #1

Titel: Bezahlte Transaktionen für alle

Testart: Manuell

Tester: Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. MetaMask installiert
2. Zwei Accounts in MetaMask eingerichtet: (1.Account Add: 0xdb3cc88CA30407c079deCBcfF9c34E4a23Aaa669 PK: 080BACEA441C50059857889AFD35D22E91C5F27C1639BE441C8E456DF678B28D 2. Account beliebig)
3. Blockchain ist aufgesetzt und läuft
4. DoS Algorithmus läuft und der Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Zum Account mit der Adresse „0xdb3cc88CA30407c079deCBcfF9c34E4a23Aaa669“ wechseln
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. zweiten Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 2 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist

Erwünschtes Resultat:

- Eine bestätigte Transaktion

Tatsächliches Resultat:

- Die Transaktion ist erfolgreich durchgeführt worden

6.4.2 Abnahmetest 2

AK Nr. : #2

Titel: Gratis Transaktionen nur für Accounts in Whitelist

Testart: Manuell

Tester: Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. MetaMask installiert
2. Zwei Accounts in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34
PK: FD825395E5B421CE6DDCEDBE4205FAB1F916E73E56721D5FD4D49C540BA2DB94 2. ein Account welcher nicht in der Whitelist ist)
3. Blockchain ist aufgesetzt und läuft
4. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Zertifizierten Account wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Account 2. wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld Transaktionsgebühr 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Auf den 2. Account wechseln
9. Kontrollieren ob der Betrag überwiesen wurde
10. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
11. Account 1. wählen
12. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
13. WEITER und BESTÄTIGEN wählen
14. Warten bis die Transaktion durchlaufen ist

Erwünschtes Resultat:

- Erste Transaktion erfolgreicher durchlaufen
- Zweite Transaktion abgebrochen

Tatsächliches Resultat:

- Erste Transaktion ist erfolgreich durchlaufen

- Die zweite Transaktion wurde abgebrochen

6.4.3 Abnahmetest 3

AK Nr.: #3

Titel: Account aus Liste für Whitelist zertifizieren

Testart: Manuell **Tester:** Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. MetaMask installiert
2. Alle Accounts die in der Datei `AccountList.json` erfasst sind, sind in MetaMask eingerichtet
3. Blockchain ist aufgesetzt und läuft
4. DoS Algorithmus läuft

Ablauf:

Für jeden Account wird folgender Ablauf durchgeführt

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist

Erwünschtes Resultat:

- Bei allen Accounts ein erfolgreicher Durchlauf der Transaktion

Tatsächliches Resultat:

- Bei allen Accounts ein erfolgreicher Durchlauf der Transaktion

6.4.4 Abnahmetest 4

AK Nr.: #4

Titel: Account aus Liste und Whitelist entfernen

Testart: Manuell

Tester: Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. MetaMask installiert
2. Zu löschender Account in `AccountList.json` erfasst
3. Zu löschender Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014E425395E5B421CE6DDCEDBE4205FAB1F916E73E56721D5FD4D49C540BA2DB94 2. Account beliebig)
4. Parameter `resetInterval` in `DefaultSettings.json` ist auf 10 gesetzt
5. Blockchain ist aufgesetzt und läuft
6. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Transaktionsmanager stoppen
9. In `AccountList.json` den Parameter `deleteMe` auf `true` setzen.
10. Transaktionsmanager starten
11. MetaMask starten
12. Account 1. wählen
13. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
14. Beliebigen Account wählen
15. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
16. WEITER und BESTÄTIGEN wählen
17. Warten bis die Transaktion durchlaufen ist
18. Transaktionsmanager stoppen
19. Prüfen ob Account noch in `AccountList.json` vorhanden ist

Erwünschtes Resultat:

- Erste Transaktion erfolgreicher Durchlauf
- Zweite Transaktion abgebrochen
- Account ist nicht mehr in `AccountList.json`

Tatsächliches Resultat:

- Erste Transaktion ist erfolgreich durchgelaufen
- Zweite Transaktion abgebrochen
- Account ist nicht mehr in `AccountList.json`

6.4.5 Abnahmetest 5**AK Nr.:** #5 & #11**Titel:** Account nach Transaktionen sperren**Testart:** Manuell**Tester:** Faustina Bruno**Datum:** 15.3.2020**Status:** ERFOLGREICH**Vorbedingung:**

1. MetaMask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34 PK: FD825395E5B421CE6DDCEDBE4205FAB1F916E73E56721D5FD4D49C540BA2DB94 2. Account beliebig)
3. Parameter `resetInterval` in `DefaultSettings.json` ist auf 10 gesetzt
4. Blockchain ist aufgesetzt und läuft
5. Transaktionsmanager läuft und Account 1. ist zertifiziert
6. Account hat in `AccountList.json` die Transaktionslimite von 5 und Gaslimite von 126000

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Schritte 2 bis 7 viermal wiederholen
9. Datei `AccountList.json` prüfen
10. Schritte 2 bis 7 nochmals wiederholen
11. Datei `AccountList.json` prüfen

Erwünschtes Resultat:

- Nach 5 Transaktionen ist beim Account der Parameter `remainingTransactions` gleich 0
- Die letzte Transaktion wird abgebrochen
- Nach der 6. Transaktion ist beim Account der Parameter `timeStamp` gesetzt

Tatsächliches Resultat:

- Nach 5 Transaktionen ist beim Account der Parameter `remainingTransactions` gleich 0
- Die letzte Transaktion wird abgebrochen
- Nach der 6. Transaktion ist beim Account der Parameter `timeStamp` gesetzt

6.4.6 Abnahmetest 6

AK Nr.: #6 & #11 & #12 **Titel:** Account nach GasUsed sperren

Testart: Manuell

Tester: Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. MetaMask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
3. Parameter `resetInterval` in `DefaultSettings.json` ist auf 10 gesetzt
4. Blockchain ist aufgesetzt und läuft
5. Transaktionsmanager läuft und Account 1. ist zertifiziert
6. Account hat in `AccountList.json` den Parameter `gasLimit` auf 42000 gesetzt und `transactionLimit` auf 5

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Schritte 2 bis 7 wiederholen
9. `AccountList.json` prüfen
10. Schritte 2 bis 7 wiederholen

Erwünschtes Resultat:

- Der Account hat in `AccountList.json` einen Timestamp eingetragen
- Die letzte Transaktion wird abgebrochen
- Der Parameter `remainingGas` in `AccountList.json` ist beim Account auf 0

Tatsächliches Resultat:

- Der Account hat in `AccountList.json` einen Timestamp eingetragen
- Die letzte Transaktion wird abgebrochen
- Der Parameter `remainingGas` in `AccountList.json` ist beim Account auf 0

6.4.7 Abnahmetest 7**AK Nr.:** #7 & #10**Titel:** Gesperrter Account entsperren**Testart:** Manuell**Tester:** Faustina Bruno**Datum:** 15.3.2020**Status:** ERFOLGREICH**Vorbedingung:**

1. MetaMask installiert
2. Blockchain ist aufgesetzt und läuft
3. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
4. Parameter `resetInterval` in `DefaultSettings.json` ist auf 10 gesetzt
5. Parameter `revokeTime` von Account 1. in `AccountList.json` auf 2 gesetzt
6. Parameter `transactionLimit` von Account 1. in `AccountList.json` auf 1 gesetzt
7. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist

8. Schritte 2 bis 7 zweimal wiederholen
9. 2 Minuten warten
10. Schritte 2 bis 7 wiederholen

Erwünschtes Resultat:

- Dritte gratis Transaktion kann nicht durchgeführt werden
- Nach 2 Minuten kann wieder eine gratis Transaktion durchgeführt werden

Tatsächliches Resultat:

- Dritte gratis Transaktion kann nicht durchgeführt werden
- Nach 2 Minuten kann wieder eine gratis Transaktion durchgeführt werden

6.4.8 Abnahmetest 8**AK Nr.:** #7**Titel:** Gesperrter Account entsperren **Testart:** Manuell**Tester:** Faustina Bruno**Datum:** 15.3.2020**Status:** ERFOLGREICH**Vorbedingung:**

1. MetaMask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
3. Blockchain ist aufgesetzt und läuft
4. Parameter `resetInterval` in `DefaultSettings.json` ist auf 10 gesetzt
5. Parameter `revokeTime` von Account 1. in `AccountList.json` auf 2 gesetzt
6. Parameter `transactionLimit` von Account 1. in `AccountList.json` auf 1 gesetzt
7. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist

8. Schritte 2 bis 7 zweimal wiederholen
9. Transaktionsmanager stoppen
10. 4' warten
11. Transaktionsmanager starten
12. Schritte 2 bis 7 wiederholen

Erwünschtes Resultat:

- Dritte gratis Transaktion wird nicht ausgeführt
- Nach Start des Transaktionsmanagers kann der Account wieder gratis Transaktionen durchführen

Tatsächliches Resultat:

- Dritte gratis Transaktion wird nicht ausgeführt
- Nach Start des Transaktionsmanagers kann der Account wieder gratis Transaktionen durchführen

6.4.9 Abnahmetest 9**AK Nr.:** #7**Titel:** Unterbruch mit gesperrtem Account **Testart:** Manuell**Tester:** Faustina Bruno**Datum:** 15.3.2020**Status:** ERFOLGREICH**Vorbedingung:**

1. MetaMask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
3. Blockchain ist aufgesetzt und läuft
4. Parameter `resetInterval` in `DefaultSettings.json` ist auf 10 gesetzt
5. Parameter `revokeTime` von Account 1. in `AccountList.json` auf 60 gesetzt
6. Parameter `transactionLimit` von Account 1. in `AccountList.json` auf 1 gesetzt
7. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben

6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Schritte 2 bis 7 zweimal wiederholen
9. Transaktionsmanager stoppen
10. 4' warten
11. Transaktionsmanager starten
12. Schritte 2 bis 7 wiederholen

Erwünschtes Resultat:

- Dritte gratis Transaktion wird nicht ausgeführt
- Nach Start des Transaktionsmanagers kann der Account immer noch keine gratis Transaktionen durchführen

Tatsächliches Resultat:

- Dritte gratis Transaktion wird nicht ausgeführt
- Nach Start des Transaktionsmanagers kann der Account immer noch keine gratis Transaktionen durchführen

6.4.10 Abnahmetest 10**AK Nr.:** #8**Titel:** Reset Zeitintervall setzen**Testart:** Manuell**Tester:** Faustina Bruno**Datum:** 15.3.2020**Status:** ERFOLGREICH**Vorbedingung:**

1. MetaMask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
3. Blockchain ist aufgesetzt und läuft
4. Parameter `resetInterval` in `DefaultSettings.json` ist auf 2 gesetzt
5. Parameter `transactionLimit` von Account 1. in `AccountList.json` auf 5 gesetzt
6. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten

2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Parameter `remainingTransactions` in `AccountList.json` für Account 1. prüfen
9. 2' warten
10. Parameter `remainingTransactions` in `AccountList.json` für Account 1. prüfen

Erwünschtes Resultat:

- Parameter `remainingTransactions` in `AccountList.json` ist für Account 1. auf 4 gesetzt
- Parameter `remainingTransactions` in `AccountList.json` ist für Account 1. wieder auf 5 gesetzt

Tatsächliches Resultat:

- Parameter `remainingTransactions` in `AccountList.json` ist für Account 1. auf 4 gesetzt
- Parameter `remainingTransactions` in `AccountList.json` ist für Account 1. wieder auf 5 gesetzt

6.4.11 Abnahmetest 11**AK Nr.:** #9**Titel:** Speicherung des Reset-Intervalls**Testart:** Manuell**Tester:** Faustina Bruno**Datum:** 15.3.2020**Status:** ERFOLGREICH**Vorbedingung:**

1. Metamask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
3. Blockchain ist aufgesetzt und läuft
4. Parameter `resetInterval` in `DefaultSettings.json` ist auf 15 gesetzt
5. Account 1. ist in `AccountList.json` mit `transactionLimit` 5 erfasst

Ablauf:

1. Transaktionsmanager starten
2. 1 Minute warten
3. Parameter `timestampLastReset` in `DefaultSettings.json` prüfen und notieren
4. MetaMask starten
5. Account 1. wählen
6. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
7. Beliebigen Account wählen
8. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
9. WEITER und BESTÄTIGEN wählen
10. Warten bis die Transaktion durchlaufen ist
11. Prüfen von `remainingTransactions` in `AccountList.json` für Account 1.
12. Transaktionsmanager stoppen
13. 2 Minuten warten
14. Transaktionsmanager starten
15. 1 Minute warten
16. Prüfen ob Parameter `timestampLastReset` in `DefaultSettings.json` identisch mit Schritt 3 ist
17. Prüfen von `remainingTransactions` in `AccountList.json` für Account 1.

Erwünschtes Resultat:

- Parameter `remainingTransactions` in `AccountList.json` für Account 1. ist bei 4
- Parameter `timestampLastReset` in `DefaultSettings.json` ist identisch
- Parameter `remainingTransactions` in `AccountList.json` für Account 1. ist nach Programmstopp bei 4

Tatsächliches Resultat:

- Parameter `remainingTransactions` in `AccountList.json` für Account 1. ist bei 4
- Parameter `timestampLastReset` in `DefaultSettings.json` ist identisch
- Parameter `remainingTransactions` in `AccountList.json` für Account 1. ist nach Programmstopp bei 4

6.4.12 Abnahmetest 12**AK Nr.:** #10**Titel:** Revoke-Intervall**Testart:** Manuell**Tester:** Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. MetaMask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
3. Blockchain ist aufgesetzt und läuft
4. Parameter `revokeTime` in `AccountList.json` ist für Account 1. nicht erfasst
5. Parameter `transactionLimit` von Account 1. in `AccountList.json` auf 1 gesetzt
6. Parameter `defaultRevokeTime` in `DefaultSettings.json` ist auf 2 gesetzt
7. Parameter `resetInterval` in `DefaultSettings.json` ist auf 5 gesetzt
8. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Schritte 2 bis 7 wiederholen
9. Dannach 2' warten
10. Erneut eine gratis Transaktion durchführen

Erwünschtes Resultat:

- Account ist nach der Default Revoke Zeit wieder entsperrt

Tatsächliches Resultat:

- Account ist nach der Default Revoke Zeit wieder entsperrt

6.4.13 Abnahmetest 13

AK Nr.: #13

Titel: Standardwerte für Limiten

Testart: Manuell

Tester: Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. MetaMask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
3. Parameter `transactionLimit` in `AccountList.json` ist für Account 1. nicht erfasst
4. Parameter `defaultTxLimit` in `DefaultSettings.json` ist auf 3 gesetzt
5. Blockchain ist aufgesetzt und läuft
6. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Schritte 2 bis 7 viermal wiederholen

Erwünschtes Resultat:

- Die letzte Transaktion wird abgebrochen
- Parameter `remainingTransactions` in `AccountList.json` ist für Account 1. auf 0
- Parameter `timeStamp` in `AccountList.json` ist für Account 1. gesetzt
- Parameter `transactionLimit` in `AccountList.json` ist für Account 1. mit `null` erfasst

Tatsächliches Resultat:

- Die letzte Transaktion wird abgebrochen
- Parameter `remainingTransactions` in `AccountList.json` ist für Account 1. auf 0
- Parameter `timeStamp` in `AccountList.json` ist für Account 1. gesetzt
- Parameter `transactionLimit` in `AccountList.json` ist für Account 1. mit `null` erfasst

6.4.14 Abnahmetest 14

AK Nr.: #13

Titel: Standardwerte für Limiten

Testart: Manuell

Tester: Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. MetaMask installiert
2. Account in MetaMask eingerichtet (1. Account Add: 0x3e7Beee9585bA4526e8a7E41715D93B2bE014B34, 2. Account beliebig)
3. Parameter `gasLimit` in `AccountList.json` ist für Account 1. nicht erfasst
4. Parameter `defaultGasLimit` in `DefaultSettings.json` ist auf 63000 gesetzt
5. Blockchain ist aufgesetzt und läuft
6. Transaktionsmanager läuft und Account 1. ist zertifiziert

Ablauf:

1. MetaMask starten
2. Account 1. wählen
3. Auf SENDEN und anschliessend auf ZWISCHEN MEINEN KONTAKTEN TRANSFERIEN drücken
4. Beliebigen Account wählen
5. In das Feld „Betrag“ 100 ETH eingeben und in das Feld „Transaktionsgebühr“ 0 eingeben
6. WEITER und BESTÄTIGEN wählen
7. Warten bis die Transaktion durchlaufen ist
8. Schritte 2 bis 7 viermal wiederholen

Erwünschtes Resultat:

- Die letzte Transaktion wird abgebrochen
- Parameter `remainingGas` in `AccountList.json` ist für Account 1. auf 0
- Parameter `timeStamp` in `AccountList.json` ist für Account 1. gesetzt
- Parameter `gasLimit` in `AccountList.json` ist für Account 1. mit `null` erfasst

Tatsächliches Resultat:

- Die letzte Transaktion wird abgebrochen
- Parameter `remainingGas` in `AccountList.json` ist für Account 1. auf 0
- Parameter `timeStamp` in `AccountList.json` ist für Account 1. gesetzt
- Parameter `gasLimit` in `AccountList.json` ist für Account 1. mit `null` erfasst

6.4.15 Abnahmetest 15

AK Nr.: #14

Titel: Certifier erneut registrieren

Testart: Manuell

Tester: Faustina Bruno

Datum: 15.3.2020

Status: ERFOLGREICH

Vorbedingung:

1. <https://remix.ethereum.org/> im Browser geöffnet
2. Code von `SimpleCertifier.sol` kompiliert in Remix
3. Code von `SimpleRegistry.sol` kompiliert in Remix
4. Blockchain läuft
5. Name Registry ist vorhanden
6. Certifier ist deployed
7. Certifier ist bei Name Registry registriert

Ablauf:

1. `SimpleCertifier` deployen mit Remix
2. Mit Remix die `SimpleRegistry` anhand der Adresse laden
3. Methode `reserve` der `SimpleRegistry` mit „6d3815e6a4f3c7fcec92b83d73dda2754a69c601f07723ec5a2274bd“ als Parameter ausführen.

Erwünschtes Resultat:

- Die Transaktion gibt `false` zurück

Tatsächliches Resultat:

- Die Transaktion gibt `false` zurück

6.5 Betriebsanleitung

In diesem Kapitel wird dem Benutzer erklärt wie er den Blockchain Transaktionsmanager in Betrieb nimmt. Für die Inbetriebnahme muss Docker auf dem Rechner installiert sein.

6.5.1 Hochfahren der Parity Instanz

1. Die Konfiguration der Blockchain wird über die Datei `config.toml` gemacht, siehe 3.1.1.1. Hier muss die Abhängigkeit der Pfade zum Docker-Home beachtet werden.
Parity stellt für die Konfiguration des Dockers diese Anleitung zur Verfügung:
<https://wiki.parity.io/Docker>
2. Die Blockchainspezifikation ist in der Datei `instant_seal.json` definiert. Der Speicherort muss mit dem Wert des Parameters `chain` der `config.toml` Datei übereinstimmen. Siehe 3.1.1.2 für den Aufbau der Datei `instant_seal.json`.
Die Dateien sind im Anhang unter 6.6.3 verlinkt.
3. Für dieses Projekt ist mti der Version 2.5.13 gearbeitet worden:
`docker pull parity/parity:2.5.13`
Für die „Stable“-Version von Parity kann folgender Befehl verwendet werden:
`docker pull parity/parity:stable`
4. Mit folgendem Befehl wird ein neuer Party-Container erstellt:
`docker run -ti -p 8545:8545 -p 8546:8546 -p 30303:30303 -p 30303:30303/
udp -v ~/.local/share/io.parity.ethereum/docker:/home/parity/.local/
share/io.parity.ethereum/ parity/parity:2.5.13 --config /home/parity/.
local/share/io.parity.ethereum/config.toml --jsonrpc-interface all`

6.5.2 Deployment

Der Transaktionsmanager kann zu einer `.jar` Datei gepackt werden. Um diese neu zu erstellen muss die Methode `package` von Maven verwendet werden. Dies kann über die Kommandozeile oder in der Entwicklungsumgebung gemacht werden.

Um alle Tests auszuführen wird bereits eine funktionierende Ethereum Blockchain mit einem Parity-Clients benötigt. Deshalb werden die Tests momentan nicht automatisch ausgeführt. Das kann in der `pom.xml` eingestellt werden:

Listing 6.1: pom.xml Properties

```
1 <properties>
```

```
2     <skipTests>true</skipTests>
3 </properties>
```

Um die Tests auszuführen müssen die Dateien `AccountList.json` und `DefaultSettings.json` durch die Dateien im Ordner `'BTMChainObserver/src/test/java/resources/testFilesForWhitelist'` ersetzt werden.

Wenn die Zeile 2 im Listing 6.1 gelöscht wird, werden vor dem Erstellen einer neuen `.jar` Datei, alle Tests automatisch ausgeführt.

Es werden zwei unterschiedliche `.jar` Dateien erstellt:

1. `BTMChainObserver-1.jar`
2. `BTMChainObserver-1-jar-with-dependencies.jar`

Die Datei 1. enthält keine Dependencies und ist somit viel schlanker. Im zweiten `.jar` sind alle Dependencies mitgepackt. Sie ist somit unabhängiger, dafür bedeutend grösser.

6.5.3 Konfigurieren des Transaktionsmanagers

Die Konfiguration des Transaktionsmanager geschieht über folgende Dateien:

1. `AccountList.json`
2. `DefaultSettings.json`
3. `TransaktionsManagerAccount.json`

Siehe Kapitel 3.2.4 und Kapitel 3.2.5 für mehr Informationen zu deren Inhalt und Bedeutung.

Diese Dateien müssen im selben Verzeichnis wie die `.jar` Datei des Transaktionsmanagers befinden.

6.5.4 Starten der Blockchain Überwachung mit dem Blockchain Transaktionsmanager

Der generierte `.jar` Datei wird über die Kommandozeile angesteuert. Die hier gezeigten Befehle setzen voraus, dass man sich im selben Verzeichnis befindet.

Um das Programm das erste Mal nach dem Aufsetzen der Parity Instanz laufen zu lassen, muss folgender Befehl auf der Console ausgeführt werden:

```
java -cp BTMChainObserver-1-jar-with-dependencies.jar ch.brugg.fhnw.btm.
Init
```

Sobald die Initialisierung abgeschlossen ist, kann folgender Befehl genutzt werden:

```
java -cp BTMChainObserver-1-jar-with-dependencies.jar ch.bruigg.fhnw.btm.Run
```

Dieser startet die Überwachung der Blockchain.

6.6 Verlinkung von Code

Hier ist sämtlicher Code verlinkt.

6.6.1 Name Registry und Certifier

<https://github.com/parity-contracts/name-registry/tree/master/contracts>

6.6.2 Transaktionsmanager

<https://github.com/FaustinaBrunoFHNW/BTMChainObserver/tree/master/BTMChainObserver>

6.6.2.1 Tests

<https://github.com/FaustinaBrunoFHNW/BTMChainObserver/tree/master/src/test/java/ch/brugg/fhnw/btm>

6.6.3 Dateien für Parity

<https://github.com/FaustinaBrunoFHNW/BTMChainObserver/tree/master/Parity>

7 Ehrlichkeitserklärung

Die eingereichte Arbeit ist das Resultat unserer persönlichen, selbstständigen Beschäftigung mit dem Thema. Alle wörtlichen und sinngemässen Übernahmen aus anderen Werken sind als solche gekennzeichnet.

Faustina Bruno

Serge Jurij Maïkoff

Ort

Datum