IP6: Blockchain Transactionmanager

Bachelorthesis

Faustina Bruno, Jurij Maïkoff

Studiengang:
- iCompetence
- Informatik

Betreuer:
- Markus Knecht
- Daniel Kröni

Experte:
- Konrad Durrer

Auftraggeber:
Fachhochschule Nordwestschweiz
FHNW Campus Brugg-Windisch
Bahnhofstrasse 6
5210 Windisch

2019-10-01

Abstract

// TODO

Inhaltsverzeichnis

1 Einleitung

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

1.1 Blockchain-Transaktionsmanager

Es ist eine private Ethereumblockchain[1][2] aufgesetzt worden, die es einer spezifischen Benutzergruppe erlaubt, gratis Transaktionen zu tätigen. Diese Transaktionen werden durch den Blockchaintransaktionmanager überwacht. Der entwickelte Transaktionsmanager stellt sicher, dass gratis Transaktionen nicht für eine Denial of Service (DoS) Attacke[3] genutzt werden können.

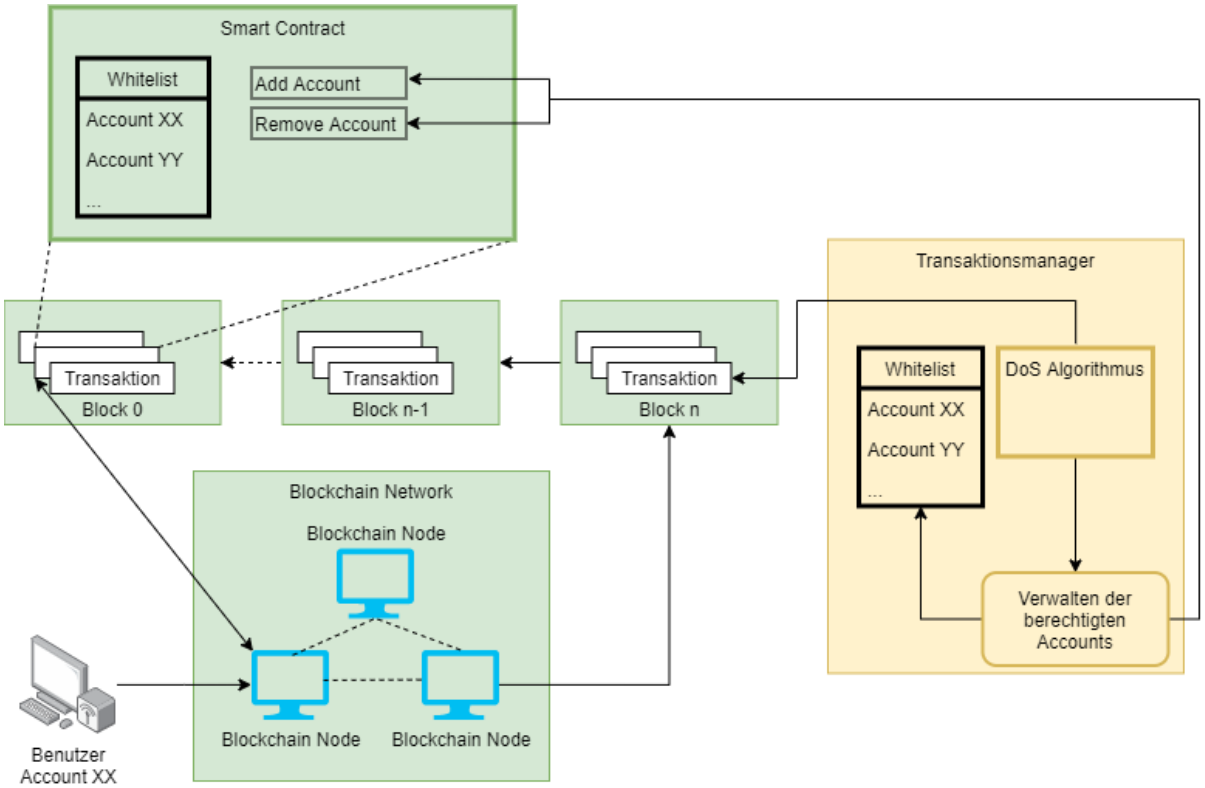

Abbildung 1.1: Interaktion von Benutzer, Blockchain und Transaktionsmanager

Auf der Grafik 1.1 sind die Interaktionen zwischen einem Benutzer, dem Blockchainnetzwerk und dem Transaktionmanager 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[wiki_whitelisting] verwendet. Diese befindet sich auf der Blockchain. Ist diese Prüfung erfolgreich, wird die Transaktion in den nächsten Block aufgenommen.
Der Transaktionmanager registriert, dass eine neue gratis Transaktion in die Blockchain aufgenommen worden ist. Der Transaktionsmanager führt eine Kopie der Whitelist. Der Absender 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 einer gewissen 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 Blockchainentzwerk übermittelt und dort ebenfalls in die Whitelist aufgenommen. Accounts können auch permanent von gratis Transaktionen ausgeschlossen werden.

1.2 Problemstellung und Ziel

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 anfallende Komputationskosten auf der Blockchain berücksichtigt. Die Komputationskosten werden 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 gegn DoS Attacken. Ohne diesen Schutz, könnten zum Beispiel Schlaufen ohne Abbruchbedingung für einen Angriff verwendet werden.
Um die Kosten einer Transaktion zu berechnen, wird das verbrauchte Gas mit dem Gas Preis[4] multipliziert. Diese Kosten werden vom Sender einer Transaktion an den Node gezahlt, der die Transaktion verarbeitet hat.
Die Fachhochschule Nordwest Schweiz[5] (FHNW) möchte für die Studierenden eine Blockchain zur Verfügung stellen. Den Studierenden soll so eine Möglichkeit geboten werden, erste Erfahrungen und Wissen im Umgang mit einer Blockchain zu gewinnen.

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.

1.3 Realisierung

//TODO ergänzen Priority Queue und Command Pattern

Parity ist zur Zeit der einzige Ethereumclient, der über ein Benutzermanagement für gratis Transaktionen verfügt. Durch die Verwendung von Smart Contracts[6] 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 Blockchaintransaktionsmanger ist als Javaapplikation realisert worden. Die Interaktion mit dem Betreiber findet über die Kommandozeile und Konfigurationsdateien statt. Mit der Bibliothek Web3j[7] ist die Anbindung an eine Blockchain sehr effizient und intuitiv. Im Transaktionsmanager wird eine Kopie der Whitelist geführt. Zu jedem Account werden zusätzliche Informationen gespeichert. Diese werden für die Beurteilung, ob der Account eine Bedrohung ist, verwendet. Um eine Datenpersistenz zu gewährleisten, wird die Liste regelmässig als JSON-Datei gespeichert.
Mit einer Subscription wird jeweils der aktuelle Block der Blockchain auf Transaktionen untersucht. Bei gefundenen gratis Transaktionen wird das Verhalten des Senderaccounts evaluiert. Anhand der Anzahl gätigten 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 bedeuted, 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. Gemachte Literaturstudiuen, geprüfte Tools, der aktuelle Stand der Ethereumblockchain, sowie die konzipierten Lösungsansätze

und derern 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 verwendeter Code zu finden.

2 Theoretische Grundlagen

Dieses Kapitel befasst sich nebst dem Kontext der Arbeit, mit den gemachteten Literaturrecherchen, welche für die Erarbeitung der Lösungsansätze nötig sind. 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 die Möglichkeit bieten, in einer sicheren Umgebung Erfahrungen zu sammeln und Wissen zu gewinnen. Obwohl eine öffentliche Blockchain für jedermann frei zugänglich ist, sind fast alle Aktionen mit Kosten verbunden. Die Kosten sind ein fixer Bestandteil einer Blockchain. So fallen zum Beispiel bei jeder Transaktionen Gebühren an. 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 Studenten sollen gratis mit der Blockchain agieren können, ohne dass der Betrieb oder die Sicherheit der Blockchain kompromitiert 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 die 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[2].
Ein speziell erwähnenswerter Block, ist der sogenannte Genesisblock[8]. Dieser ist der erste Block in einer Blockchain. Der Genesisblock ist eine JSON Datei mit allen 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 Consenus Algorithmen[10] entgegengewirkt. Die Consensus Algorithemen stellen sicher, dass die Transaktionen auf der Blockchain valide und authentisch sind.

Im Gegensatz zu Bitcoin[11] kann bei Ethereum[1] 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 erten Mal verwendet. Es handelt sich um ein Stück Code, das auf der Blockchain liegt. Es können Vertragsbedingungen als Code geschrieben werden. Sobald die 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.

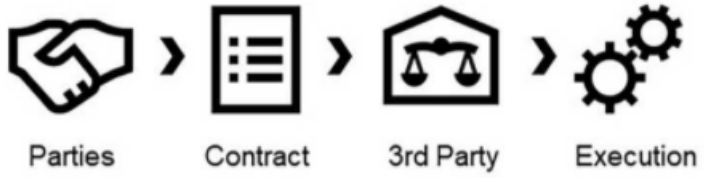

Abbildung 2.1: Ein traditioneller Vertag[13]

Abbildung 2.2: Ein Smart Contract[13]

Der grosse Vorteil von Smart Contracts ist, dass keine third parties 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 third parties 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 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. Das prominenteste Beispiel einer DApp ist CryptoKitties[14]. Die Benutzer können mit digitale Katzen handeln und züchten.

2.2.3 Transaktionen

Um mit der Blockchain zu interagieren, werden Transaktionen benötigt. 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[15] 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, wir in diesem Feld der dessen Bytecode[16] 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 gwillt 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 [17]

Damit eine Transaktion in die Blockchain aufgenommen werden kann, muss sie signiert[18] 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)[19] gemacht.

2.2.4 Gas

Mit Gas[4] ist in der Ethereum Blockchain eine spezielle Währung gemeint. Mit ihr 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 **??**, nicht lohnt, diese zu verarbeiten. Ein hoher Gas Price stellt zwar sicher, dass die Transaktion schnell verarbeitet wird, kann aber hohe Gebühren generieren.

$$TX = gasCost * gasPrice$$

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 einem 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[20] verwendet, ein „Elliptical Curve Digital Signature Algorithm“[21]. Der öffentliche Schlüssel wird verwendet um die Signatur einer Transaktion zu verifizieren.

Adresse Die Adresse wird aus dem öffentlichen Schlüssel abgeleitet. Es wird SHA3[22] 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.

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.

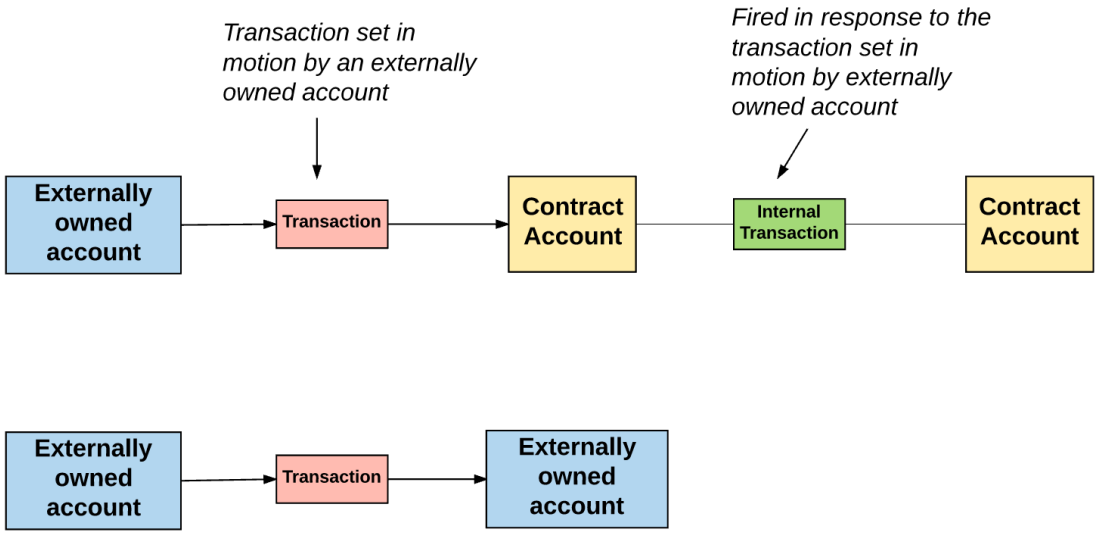

Abbildung 2.3: Transaktionen zwischen verschiedenen Accounts[23]

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[24]) des Benutzer Accounts, der das Deployment vornimmt.[25]

2.2.6 Blockchain Wallet

Eine Blockchain Wallet, kurz Wallet, ist ein digitales Portmonaie. 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, Destkop, Web und Mobile Wallets. [26], [27], [28]

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. [26], [27], [28]

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. [29], [30], [31]

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[32]. 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 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 tedentiell steigen. Um sicherzustellen, dass seine Transaktionen weiterhin zuverlässig in die Blockchain aufgenommen werden, muss der Angfreifer 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.[33]

2.2.7.1 DoS Attacke an der FHWN

Auf der Blockchain der FHNW existiert eine priviligierte 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 priviligierten Accounts überwacht werden. Falls einer dieser Accounts eine DoS Attacke einleitet, muss das frühst möglich erkannt und unterbunden werden können.

2.3 Ethereum Client

Für die Betreibung von einem Ethereum Node ist ein Client nötig. Dieser muss das Ethereum Protokoll[34] implementieren. Das Protokoll definiert die minimalen Anforderungen an den Client. Das erlaubt, dass der Client in verschiedenen Sprachen, 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)[35], Parity[36], Aleth[37] und Trinity[38]. Die Clients wurden auf die Zusatzfunktionalität untersucht, für eine definierte Gruppe von Accounts gratis Transaktionen zu ermöglichen.

2.3.1 Parity

Geschrieben in Rust[39], ist es der zweit populärste Client nach Geth[35]. 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 priviligierten 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[40] und der Service Transaction Checker[41]. Diese sind in den folgenden Abschnitten erklärt.

2.3.1.1 Name Registry

In Parity wir die Name Registry verwendet, um eine Accountaddresse 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[42].

Die Name Registry ist in Parity standardmässig immer unter derselben Addresse zu finden. 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

```
struct Entry {
    address owner;
    address reverse;
    bool deleted;
    mapping (string => bytes32) data;
}
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. Änderungn daran sind daher durch Modifiers eingeschränkt.

Listing 2.2: Modifier whenUnreserved

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

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

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

Unter 2.3 wird der Besitzer 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

```
modifier whenEntryRaw(bytes32 _name) {
    require(
        !entries[_name].deleted &&
        entries[_name].owner != address(0)
    );
    _;
}
```

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 sich um mehr als einen uninitialisierten Account handelt.

Listing 2.5: Gebühr (Fee) und Modifier whenFeePaid

```
uint public fee = 1 ether;

modifier whenFeePaid {
    require(msg.value >= fee);
    _;
}
```

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

```
function reserve(bytes32 _name)
    external
    payable
    whenUnreserved(_name)
    whenFeePaid
    returns (bool success)
{
    entries[_name].owner = msg.sender;
    emit Reserved(_name, msg.sender);
    return true;
}
```

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 ans Netzwerk gesendet.

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 Eintags (`_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 ans Netzwerk gesendet.

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 geändert werden. Mit der Registrierung des Certifiers bei der Name Registry. 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 Absender zertifiziert ist. Transaktionen von zertifizierten Accounts werden selbst mit einem Gas Price von 0 in die Blockchain aufgenommen. Gratis Transaktionen von unzertifiziereten Benutzern werden weiterhin abgewiesen.

In diesem Abschnitt sind besonders wichtige Abschnitte des SimpleCertifiers 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 **??** ist die zentrale Datenstrucktur 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 Absender 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. Als Paramater 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 an das Netzwerk gesendet.

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`) zertifizierzt ist. Mit `view` auf Zeile 3 ist deklariert, dass es sich um eine Abfrage ohne weitere Komputationskosten handelt. Solche Abfragen sind daher 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 an die Blockchain gesendet.

2.3.2 Geprüfte Alternativen

Die Clients Geth, Aleth und Trinity sind ebenfalls evaluiert worden. Bei diesen Clients ist keine Möglichkeit vorhanden, bestimmte Accounts für gratis Transaktionen zu priviligieren. Daher sind sie zu diesem Zeitpunkt nicht für die FHNW geeignet.

2.4 Lösungsansätze

In diesem Kapitel werden die erarbeiteten Lösungsansätze 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 herkömlichen 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 Betreibung der Blockchain der Client Parity mit einer Withelist verwendet.

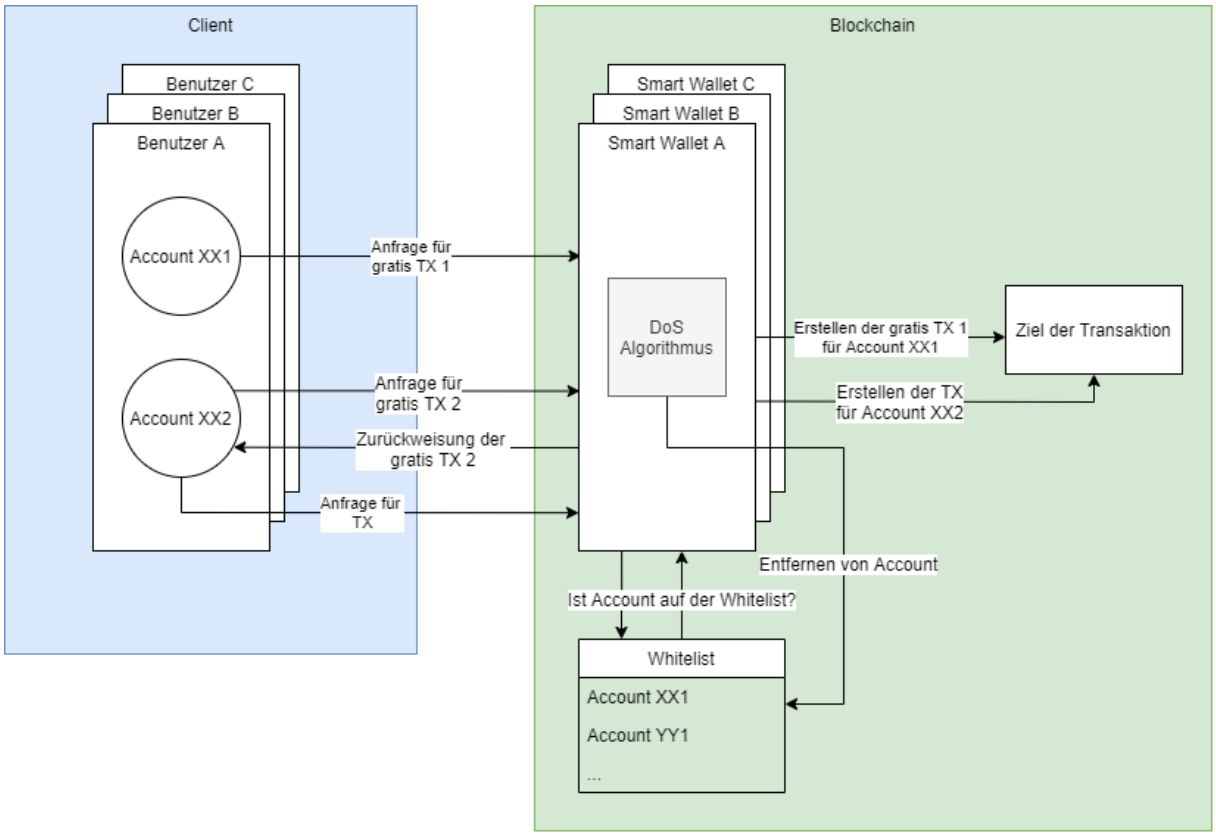

Abbildung 2.4: Architektur mit Smart Wallet

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 Alogorithmus 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 bedeuted, 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 auf 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 umzusetzten, muss der Blockchain Client, Parity, erweitert werden. Es ist unklar, wie weitreichend die Anpassungen an Parity sind. Zusätzlich wird eine zusätzliche Programmiersprache, Rust[39], 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.

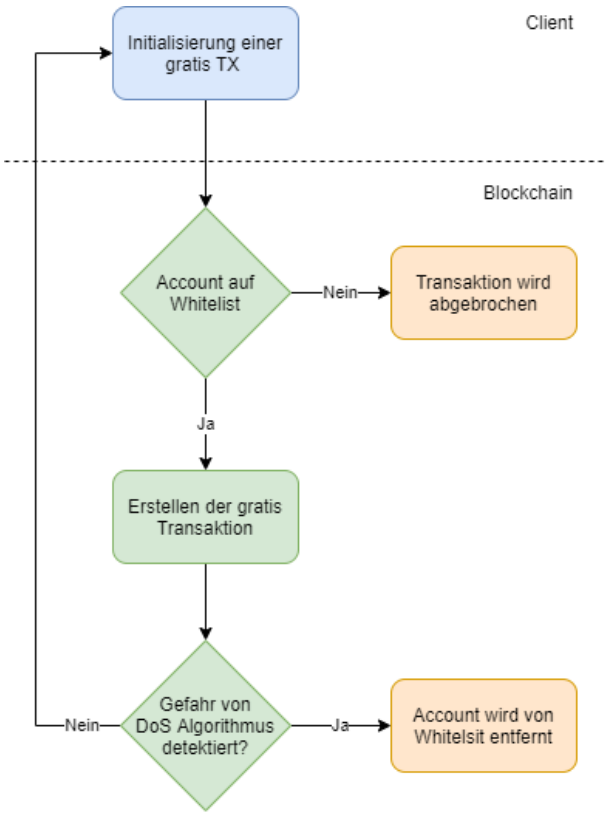

Abbildung 2.5: Flowchart für Smart Wallet

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 werden kann, kann der Account auch 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 Schutzmechnismus gegen DoS Attacken in einem externen Programm, dem Transaktionsmanager, implementiert. Dieser Ansatz verwendet ebenfalls die Whitelist von Parity, siehe 2.3.1.

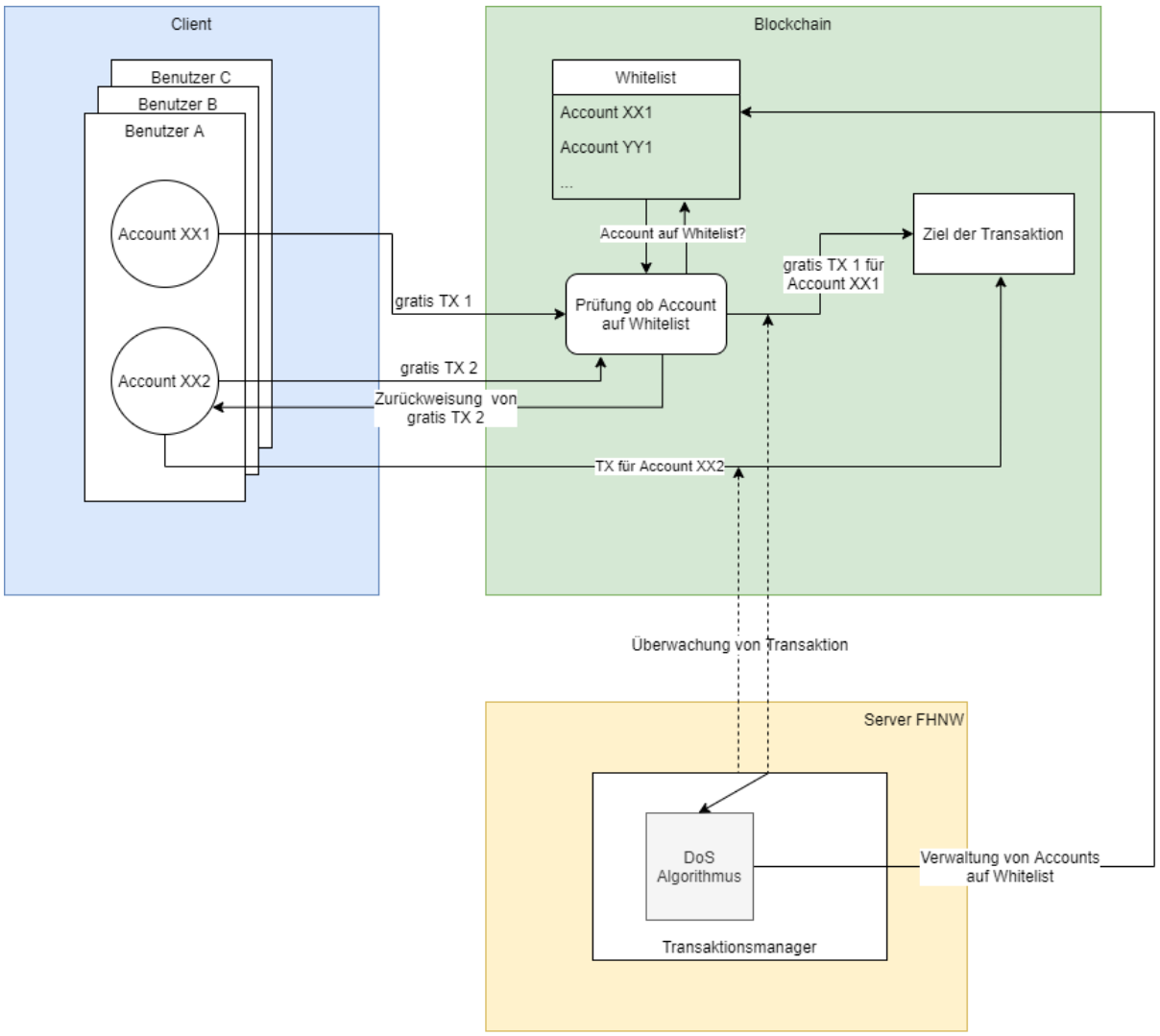

Abbildung 2.6: Externes Programm für die Verwaltung der Whitelist

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 Whitelit. 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 Transaktionsmanger 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 entpsrechende 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 sicher umsetzbar in der zur Verfügung stehenden Zeit.
Falls eine Anpassung des DoS Schutzalgorithmus nötig ist, muss nur das externe Programm neu ausgerollt werden. Eine aktualisierung der Whitelist ist nicht nötig.

Contra Es wird das Hauptprinzip, Dezentralität, einer Blockchain verletzt. Der Transaktionsmanager ist eine zentrale Authorität, die von der FHNW kontrolliert wird.
Für die Betreibung des Transaktionsmangers wird ein Server benötigt. Eine weitere Komponente, die administriert werden muss.

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

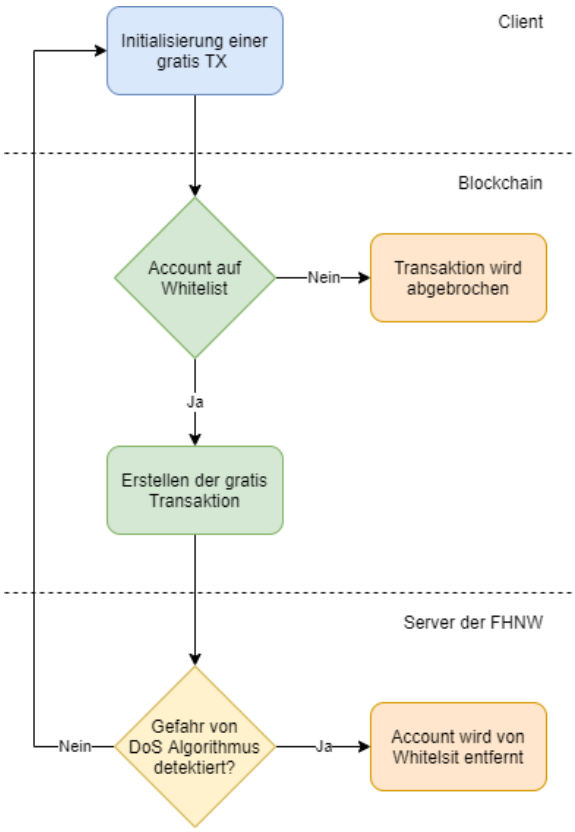

Abbildung 2.7: Flowchart Transaktionsmanager für Verwaltung der Whitelist

Der Benutzer initialisiert eine gratis Transaktion. Beim Parity Node wird überprüft, ob der verwendete Account sich auf der Whitelist befindet. Ist er das nicht, wird die Transkation 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 Transaktionmanager überwacht alle Transaktionen die auf die Blockchain gelangen. Bei gefunden 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.
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.

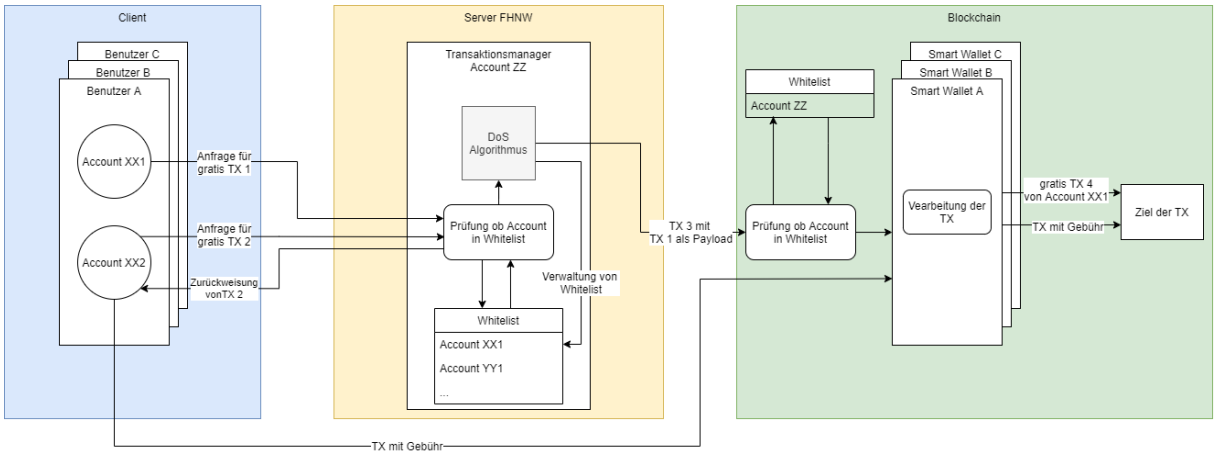

Abbildung 2.8: Transaktionsmanager mit Whitelist

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 fällte 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 gesendet. 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 gegeben Zeit umsetzbar.
Falls eine Anpassung des DoS Schutzalgorithmus nötig ist, muss nur das externe Programm neu ausgerollt werden. Eine aktualisierung der Whitelist ist nicht nötig.

Contra Es wird das Hauptprinzip, Dezentralität, einer Blockchain verletzt. Der Transaktionsmanager ist eine zentrale Authorität, 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.

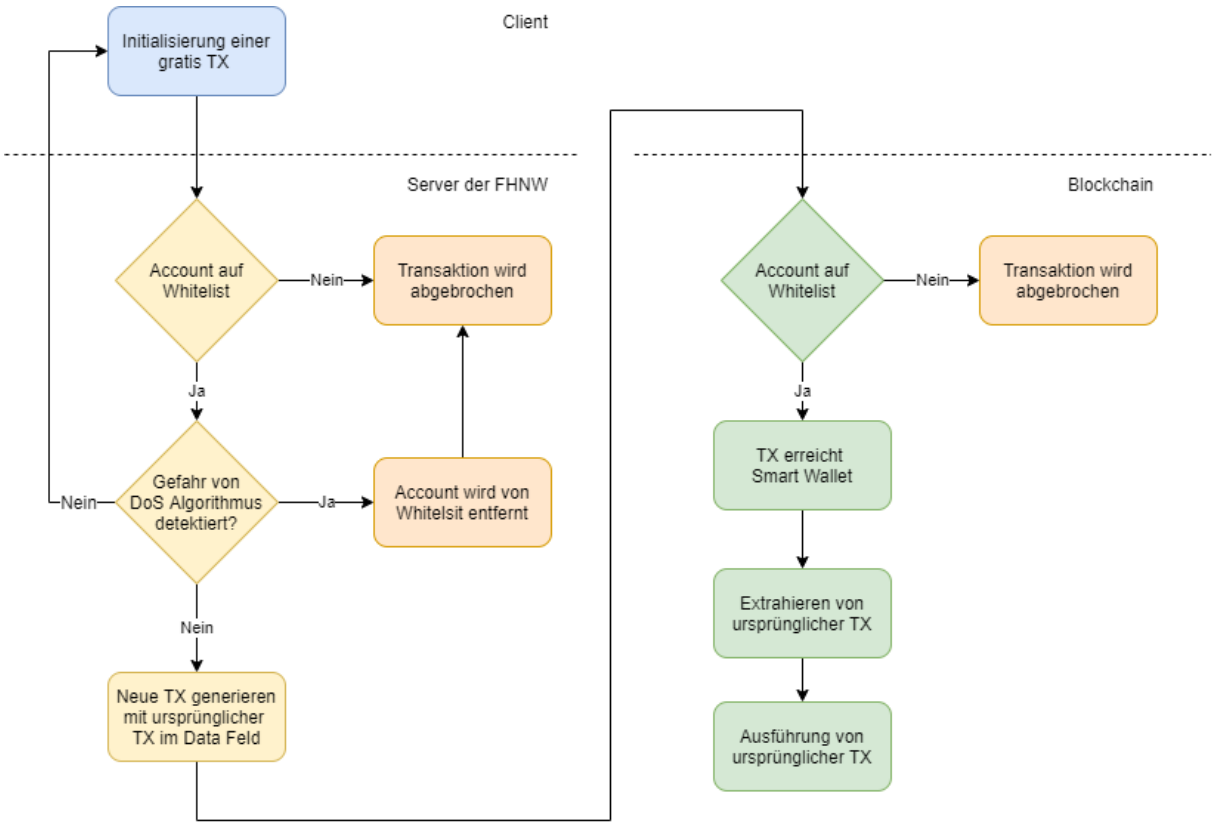

Abbildung 2.9: Flowchart für Transaktionsmanager mit Whitelist

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 weiter bearbeitet. 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 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, wurden folgende Kriterien definiert:

Machbarkeit (MK) Bewertet die Machbarkeit des Ansatzes. Das berücksichtigt den gegeben 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 mittelere Gewichtung.
Gewichtung 2

Betrieb (BT) Bewertet den administrativen Aufwand im Betrieb und die Möglichkeit zur Automatisierung. Das umfasst Deployment Smart Contracts, Anpassungen der Whitelist und Betreibung 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 3 und 1 Punkt 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 neu deployed werden. Das bedingt, dass die Whitelist ebenfalls aktualisiert wird. Die Adressen aller bestehenden Smart Wallets müssen ersetzt werden. Alle Studierenden müssen informiert werden, dass sie für ihre Smart Wallet eine neue Adresse verwenden müssen. Die Automatisierung dieser Prozesse 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 sicher realisierbar. Daher erhält der ALA für Machbarkeit die volle Punktzahl.

Mit der Verwendung eines externen Programms, wird eine zentrale Authorität verwendet. Diese ist nicht dezentral und wird von der FHNW administriert. Da das Programm die Transaktionshistorie der Blockchain überwacht und nur bei einer DoS Attacke aktiv ist, wird 2 Punkte für Blockchainprinzipien gegeben.

Falls eine Anpassung am DoS Algorithus 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 Betreibung 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. Somit ist liegt die Machbarkeit zwischen dem von ALA 1 und ALA 2. Daher werden 2 Punkte für Machbarkeit vergeben.

Mit der Verwendung eines externen Programms, wird eine zentrale Authorität verwendet. Diese ist nicht dezentral und 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 Transaktion wird an das Programm übermittelt und dort verarbeitet. Da die zentrale Authorität im Vergleich zu ALA 2 viel aktiver ist, ist für Blockchainprinzipien 1 Punkt vergeben worden.

Für die Betreibung 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.
Das externe Programm hat eine sehr zentrale Rolle, da es die Whitelist und den DoS Schutzalgorithmus enthält. Die Automatisierung wird daher als einfach eingestuft, da das externe Programm mit Java geschrieben wird und somit sehr viel zugänglicher ist. 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[7] sind Interaktionen mit der Blockchain effizient und intuitiv.

Im Anhang ist unter 6.3 ein weiterer Lösungsanatz aufgelistet. Dieser ist sehr früh in der Evaluierung als nicht realisierbar eingestuft worden und ist hier deshalb nicht aufgeführt.

2.4.3 DoS-Algorithmus

//TODO Spellcheck

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 identfiziert 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 Dieser Parameter ist zwingend nötig um eine gratis Transaktionen mit einem Account zu verknüpfen.

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

Resett-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 Progammstart 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.

Individuelles Intervall Das Intervall ist relativ zum Zeitpunkt einer getätigter 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.

Anzahl getätigte Transaktionen Pro Account wird verfolgt, wie viele gratis Transaktionen pro Zeitintervall gemacht werden. Hier werden die Transaktionen unabhängig von Typ oder verursachten Komputationskosten 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 Komputationskosten 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 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 Montag um 8:00 UTC wieder auf die Whitelist genommen.

Nach Zeitintervall 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 begannen Verstössen berechnet.

Beispiel mit einer quadratischen Abhängigkeit zwischen Verstössen Sperrung und einem Basisintervall von 5 Minuten:

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 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 von Sender frei gewählt werden. Es wird auch kein Einverständnis des Empfängers für eine Transaktion benötigt. Jeder Benutzer ist weiter in der Lage, selbst neue Accounts zu erstellen und diese als Empfänger zu verwenden. Der Parameter hat somit keine Aussagekraft und wird nicht verwendet.

Resett-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.

Die Auswirkung des genannten Nachteils bei einem allgemeinen Intervall ist stark von dessen Länge abhängig. Je kürzer das Intervall gewählt wird, umso kleiner sind die möglichen Folgen. Falls damit gerechnet werden muss, dass das Programm regelmässig gestoppt wird, muss für einen optimalen Betrieb, das Resett-Intervall entsprechend angepasst werden. Das Resett-Intervall sollte kleiner gewählt werden als die zu erwartende Dauer zwischen den Programm-Stopps.

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 Zeitpuntk 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 gnügt, 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 noch unklar.

Wir haben uns entschieden, ein Zeitintervall ab Zeitpunkt des Vergehens zu implementieren. Dieser Ansatz ist fair und verständlich für die Benutzer.

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, für die Parameter nicht explizit definiert werden. So kann die Mehrheit der Accounts über Standardparameter und 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 einer Textdatei vorgenommen. Für alle Parameter müssen natürliche Zahlen verwendet werden. Folgende Parameter können pro Account gesetzt werden:

Gratis Transaktionen Limite

```
1 Definiert die maximale Anzahl gratis Transaktionen die pro Resett-
    Intervall
```

getätigt werden können.

Gratis Gas Limite

```
1 Definirt die maximale Menge an Gas die mit gratis Transaktionen
    innerhalb
```

eines Resett-Intervalls verbraucht werden können.

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

Folgende Parameter gelten für alle Accounts:

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

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

Standardwert gratis Gas Limite Gilt für Accounts die ohne Gaslimite erfasst werden. Definirt die maximale Menge an Gas die mit gratis Transaktionen innerhalb eines Resett-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 Resett-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.

3.1 Parity

In diesem Abschnitt ist beschrieben, wie die Blockchain konfiguriert ist. Als Client wird die stable Version[44] 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. Eine einfache Konfiguration ist somit 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 wurde auf einen Mining-Algorithmus verzichtet. 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“[45].

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/"
```

```
4
5  [rpc]
6  cors = ["all"]
7  apis = ["net", "private", "parity", "personal", "web3", "eth"]
8
9  [mining]
10 min_gas_price = 1000000000
11 refuse_service_transactions = false
12 tx_queue_no_unfamiliar_locals = true
13 reseal_on_txs = "all"
14 reseal_min_period = 0
15 reseal_max_period = 6000
16
17 [misc]
18 unsafe_expose = true
```

Der oben aufgeführte Codeblock ist in Sektionen gegliedert. Diese sind durch einen Namen in eckigen Klammern definiert. Innerhalb einer Sektion existieren bestimmte Schlüssel mit einem Wert. Jede Sektion ist in den folgenden Abschnitten erklärt.

Parity In dieser Sektion sind die grundlegenden Eigenschaften der Blockchain definiert. Dazu gehören Genesisblock und der Speicherort.

Zeile 2 Der zu verwendende Genesisblock. Es wird der Pfad zu der entsprechenden JSON Datei[46] angegeben.

Zeile 3 Mit „base_path“ wird angegeben, wo die Blockchain abgespeichert werden soll. Hier wird das gewünschte Verzeichnis angegeben.

RPC Diese Sektion definiert, wie die Blockchain erreichbar ist.

Zeile 6 „cors“ steht für Cross-Origin Requests. Dieser Parameter wird benötigt, um die Interaktion von Remix[47] oder Metamask[48] mit der Blochchain zu ermöglichen.

Zeile 7 Hier sind die API’s definiert, welche über HTTP zur Verfügung gestellt werden.

Mining Diese Sektion regelt das Verhalten beim Mining von Blocks.

Zeile 10 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 11 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 12 Dieser Parameter wird benötigt, dass Transaktionen die mittels RPC an Parity übermittelt werden, nicht als lokal betracht werden. Das ist sehr wichtig, da lokale Transaktionen standarmässig auch über einen Gas-Preis von null verfügen dürfen. So wird sichergestellt, dass nur die definierte Benutzergruppe gratis Transaktionen tätigen darf.

Zeile 13 Durch die Einstellung „tx_queue_no_unfamiliar_locals = true“ werden alle eingehenden Transaktionen behandelt, als ob fremd, also nicht lokal, behandelt. Standardmässig, werden aber nur lokale Transaktionen verarbeitet. Daher muss hier explizit definiert werden, dass alle Transaktionen verarbeitet werden.

Zeile 14 Gibt an, wieviele Milisekunden im Minimum zwischen der Kreation von Blöcken liegen müssen.

Zeile 15 Defniert die maximale Zeitspanne in Millisekunden zwischen der Kreation von Blöcken. Nach Ablauf dieser Zeit wird automatisch ein Block generiert. Dieser kann leer sein.

Misc In dieser Sektion sind Parameter, die sonst nirgends reinpassen.

Zeile 18 Wird für die Interaktion mit Remix und Metamask benötigt.

3.1.1.2 Blockchainspezifikation

Mit dieser Datei wird die Blockchain definiert. Sie enthält nebst der Spezifikation den Genesis Block. Weiter können Benutzeraccounts und Smart Contracts definiert werden. Diese können verwendet werden, sobald die Blockchain gestartet ist.

Listing 3.2: Blockchainspezifikation mit Genesisblock

```
1  {
2      "name": "BachelorBlockChain",
3      "engine": {
4          "instantSeal": {
5              "params": {}
6          }
7      },
8      "params": {
9          "networkID" : "0x11",
10         "registrar" : "0x0000000000000000000000000000000000001337",
11         "maxCodeSize": 24576,
12         "maximumExtraDataSize": "0x20",
13         "minGasLimit": "0x1388",
14         "gasLimitBoundDivisor": "0x0400"
15     },
16     "genesis": {
17         "seal": {
18             "generic": "0x0"
19         },
20         "difficulty": "0x20000",
```

```
21          "gasLimit": "0x7A1200"
22      },
23      "accounts": {
24          "0000000000000000000000000000000000001337": { "balance": "1", "
               constructor": "Platzhalter für Bytecode von SimpleRegistry"
               },
25          "00a329c0648769a73afac7f9381e08fb43dbea72": { "balance": "
               1606938044258990275541962092341162602522202993782792835301376
               " }
26      }
27  }
```

Oben aufgeführt ist die Blockchainspezifikation. Im folgenden Abschnitt ist diese Zeilenweise erläutert.

Zeile 2 Name der Blockchain

Zeile 3 - 7 Der Abschnitt `engine` definiert, wie die Blöcke verarbeitet werden.

Zeile: 4 Mit `instantSeal` wird angegeben, dass kein Miningalgorithmus verwendet wird. Die Blöcke, sofern valide, werden sofort in die Blockchain aufgenommen.

Zeile 5 Die Engine InstantSeal braucht keine weiteren Parameter. Falls ein anderer Algorithmus verwendet wird, kann dieser hier konfiguriert werden.

Zeile 8 - 15 Im Abschnitt `params` sind die generellen Parameter für die Blockchain aufgeführt.

Zeile 9 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 10 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 11 Die maximale Grösse eines Smart Contracts welcher in mit einer Transaktion deployed wird.

Zeile 12 Spezifiziert die maximale Anzahl Bytes, welche im Feld `extra_data` des Headers eines Blockes mitgegeben werden kann.

Zeile 13 Definiert den minimalen Gasbetrag, der bei einer Transaktion mitgegeben werden muss.

Zeile 14 Schränkt die Schwankungen der Gas Limite zwischen Blöcken ein.

Zeile 16 - 22 Mit dem Abschnitt `genesis` ist der Genesis Block, also der erste Block, der Blockchain definiert.

Zeile 17 - 19 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 20 Gibt die Schwierigkeit des Genesis Blocks an. Da als Engine InstantSeal verwendet wird, hat dieser Parameter keinen Einfluss.

Zeile 21 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.

Zeile 23 - 26 Dieser Abschnitt erlaubtes, 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 24 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 wir mit einem Ether initalisiert. Der Wert für `constructor` ist der Bytecode des kompilierten Smart Contracts. Dieser ist aufgrund seiner Grösse durch einen Platzhalter ersetzt worden.

Zeile 25 Definition von einem Benutzeraccount. Der erste Parameter ist die Adresse. Dem Account kann ein beliebiges Guthaben zugewiesen werden.

3.1.2 Docker

Um eine möglichst realitätsnahe Entwicklungsumgebung zu erhalten, wird Docker[49] für die Betreibung der Blockchain verwendet. Mehr Details zur Verwendung von Docker sind im Anhang unter 6.2.4 vorhanden.

3.1.3 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.

3.1.4 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.

3.1.4.1 Deployment und Registrierung

Für eine Erfolgreiche Verwendung des Certifiers, die Name Registry in der Blockchainspezifikation definiert sein. Sobald Parity gestartet ist, kann mit dem Deplyoment des Certifiers begonnen werden. Hierfür wird Java und die Bibliothek Web3j[7] 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)[50] und der Solidity Compiler (solc)[51] verwendet. Die Wrapper für die SimplyRegistry und den SimpleCertifier sind im Anhang unter **??** verlinkt.

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

Listing 3.3: Deplyoment 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();
```

Die Verbindung zu einem Node wird mit einer Instanz von `Web3j` auf Zeile 1 definiert. Auf der zweiten Zeile wird ein `TransactionManager` instanziert. Dieser definiert, wie und mit welchem Account auf die Ethereumblockchain 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 `DefaultGasProvicer` 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 3.4: 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 }
```

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 wird die Verbindung zur Blockchain mit `Web3j`, einem `Transactionmanager` und einem `DefaultGasProvider` definiert. Der Rückgabewert ist eine Instanz der

```
1
2 Mit den zur Verfügung stehenden Instanzen, kann die Registrierung des
     Certifiers
3 bei der Name Registry gemacht werden.
4
5    {caption="Reservierung und anschliessende Registrierung bei der Name
      Registry" label=li_reservation_and_registration .java .numberLines}
6 private static BigInteger REGISTRATION_FEE = BigInteger.valueOf
     (1000000000000000000L);
7
8 String str_hash = "6
     d3815e6a4f3c7fcec92b83d73dda2754a69c601f07723ec5a2274bd6e81e155";
9 private byte[] hash = new BigInteger(str_hash, 16).toByteArray();
10
11 try {
12     simpleRegistry.reserve(hash, REGISTRATION_FEE).send();
13     simpleRegistry.setAddress(hash, "A", simpleCertifier.
          getContractAddress()).send();
14 } catch (Exception e) {
15     e.printStackTrace();
16 }
```

Für die Registrierung wird eine Gebühr von einem Ether erhoben. Dafür wird auf Zeile 1 eine Variabel vom Typ `BigInteger` instanziert.
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 Variabel 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.

3.1.5 Interaktionsübersicht auf einem Parity Node

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

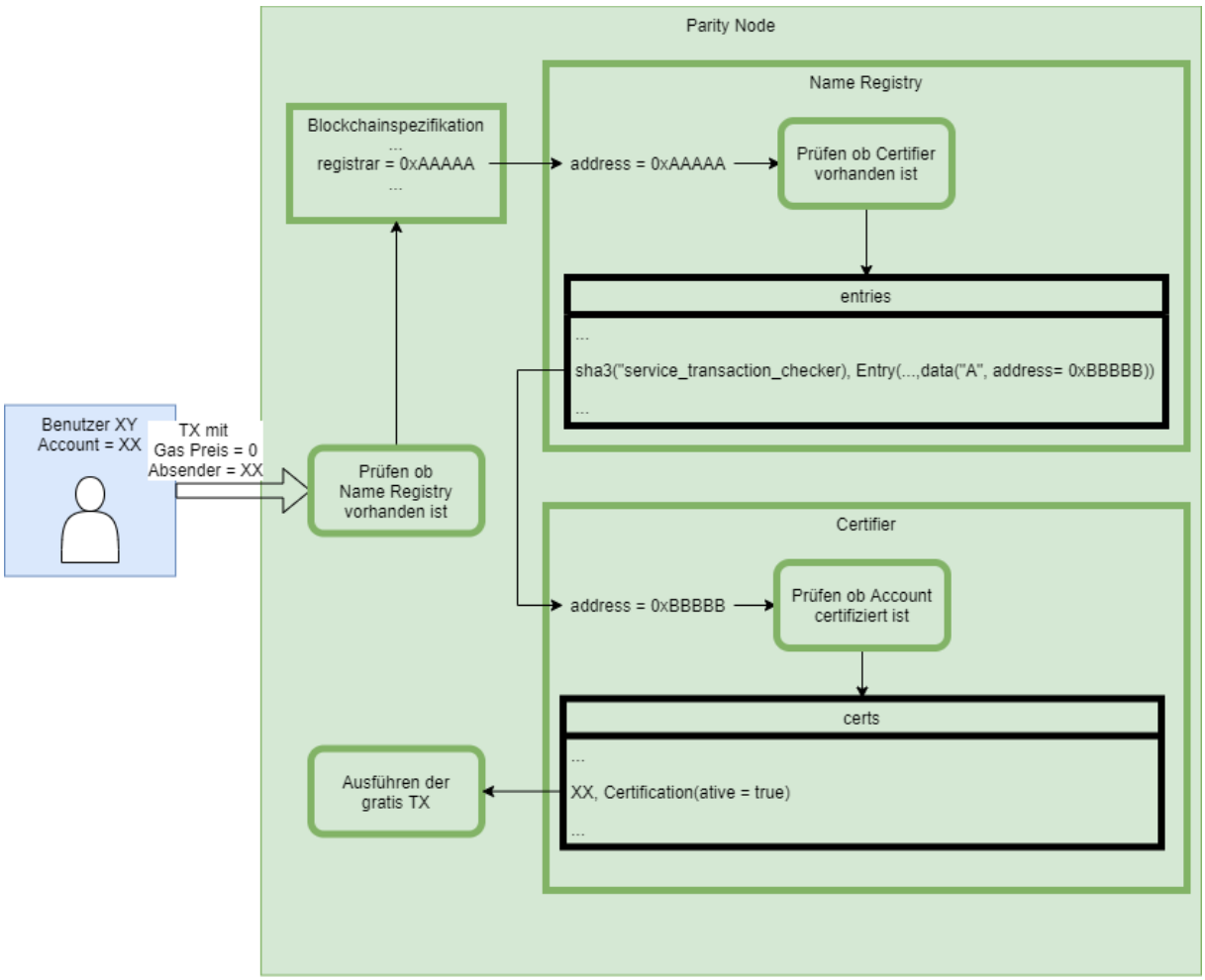

Abbildung 3.1: Interaktion zwischen Name Registry und Certifier

Auf dem Diagramm 3.1 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 Certifiers. Im Struct `Entry` ist `data`, ein weiteres Mapping, 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“ zertifiert ist.
Wenn einer dieser Schritte fehlschlägt, erhält der Benutzer einen Error, da er einen ungültigen Gas Preis verwendet.
Konnten alle Schritte erfolgreich durchgeführt werden und der Account ist zertifiziert, wird die Transaktion in die Blockchain aufgenommen.

3.2 Transaktionsmanager

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

3.2.1 Übersicht

Das folgende Klassendiagramm dient der Übersicht.

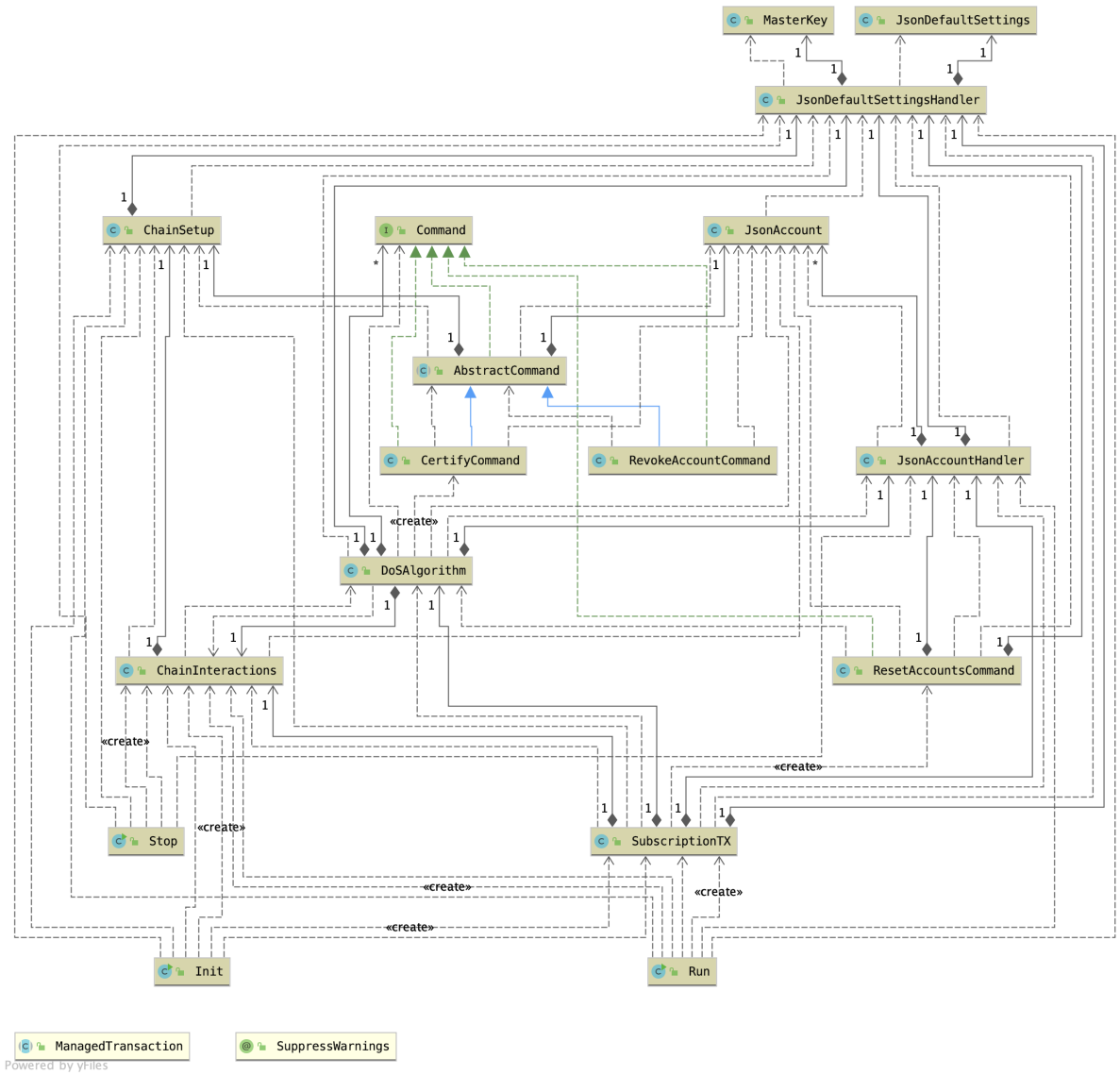

Abbildung 3.2: Klassendiagramm des Transaktionsmanagers

Das Klassendiagramm 3.2 zeigt alle verwendeten Klassen ohne Methoden oder Attribute. Die vollständigen Klassendiagramme sind im Anhang unter 6.6.2 verlinkt.

3.2.2 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.

3.2.3 Überwachung von Transaktionen

Um die Transaktionen auf der Blockchain zu Observieren wird ein Filter[52] von Web3j verwendet. Dieser erlaubt es, eine `Subscription` zu erstellen. Diese läuft asynchron in einem eigenen Thread. 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 dazugehörende Account durch den DoS Algorithmus evaluiert.

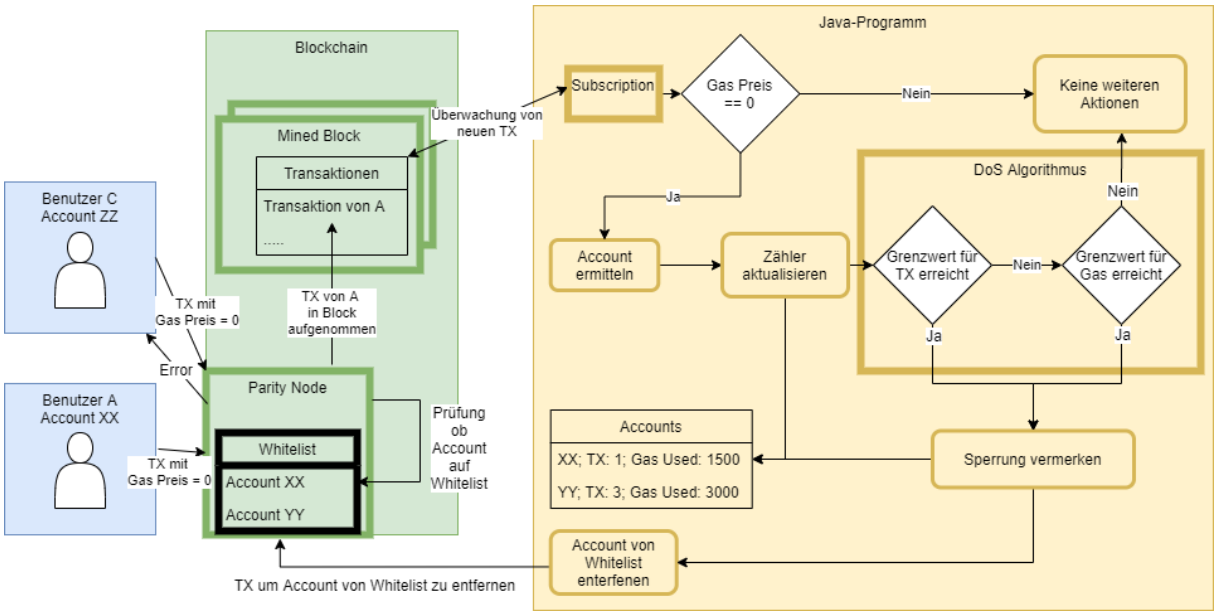

Abbildung 3.3: Prozess bei der Überwachung von Transaktionen

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 Java-Programm sind alle Accounts erfasst, die für gratis Transaktionen berechtigt sind. Hier ist für jeden Account vermerkt, wie viele gratis Transaktionen und wie viel gratis Gas im aktuellen Resett-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 3.1.5 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 Java-Programm 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 ausgewertet. Es wird geprüft, ob mit der neu erfassten Transaktion ein Grenzwert im aktuellen Resett-Intevall 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.

3.2.4 Command Pattern und Priority Queue

Für die Handhabung des Resett-Intervalls und Suspendierungen von der Whitelist, wird ein Command-Pattern[53] 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[54] 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 3.5: 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.
                      currentTimeMillis()));
7         this.jsonAccountHandler.writeAccountList();
8         this.jsonDefaultSettingsHandler.writeDefaultSettings();
9
10        DoSAlgorithm.getInstance().offerCommand(new
              ResetAccountsCommand());
11    }
```

Der Codeausschnitt 3.5 zeigt die `execute` Methode des `ResetAccountsCommand`. Dieses Command wird am Ende eines Resett-Intervalls ausgeführt. Auf Zeile 4 bis 8, werden zuerst alle Accounts zurückge-

setzt 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.

3.2.5 Persistenz

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

Dafür wird die Datei `AccountList.json` verwendet. Relativ zur JAR-Datei des Transatkionsmanagers muss sie in folgendem Verzeichnis liegen:

//TODO Filepath

In dieser Datei sind alle Accounts, die auf der Whitelist sind erfasst. Bei jedem Account können die inidviduell 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 3.2.6 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.

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

remainingTransactions Mit diesem Feld wird bestimmt, über wie viele gratis Transatkionen 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 Resett-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 bedeuted, 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 certifiziert. 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` oder via _____TODO Funktion CLI_____ gemacht werden.

Mehr zur Konfigurationsdatei und Beispiele sind im nachfolgenden Abschnitt, 3.2.6, zu finden.

3.2.5.1 Beispiele

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

Listing 3.6: AccountList.json für die Erfassung von 4 Accounts. Datei vor Programmstart

```
[
  {
    "address": "0xaf02DcCdEf3418F8a12f41CB4ed49FaAa8FD366b",
    "txLimit": 8,
    "revokeTime": 15,
    "gasLimit": 60000000
  },
  {
    "address": "0xf13264C4bD595AEbb966E089E99435641082ff24"
  },
  {
    "address": "0x00a329c0648769A73afAc7F9381E08FB43dBEA72",
    "revokeTime": 25,
    "gasLimit": 999999999999999
   },
   {
    "address": "0xf1326lasdfaösLDFJDLSNd23nlDD35641082ff24"
    "revokeTime": 2
  }

```

```
]
```

Unter 3.6 ist die Erfassung von 4 Accounts gezeigt. 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 unter 3.6 Liste von Accounts eingelesen und verarbeitet wurde, wird sie vom Transaktionsmanager neu geschrieben. Das Resultat ist abhängig von den konfigurierten Standardwerten, dazu mehr im folgenden Kapitel 3.2.6. Unter der Annahme, dass der Standarwert für Transaktionen auf 5 und für Gas auf 50000000 konfiguriert ist, sieht die Datei folgendermassen aus:

Listing 3.7: AccountList.json mit 4 Accounts. Datei nach Programmstart

```
[
  {
    "address": "0xaf02DcCdEf3418F8a12f41CB4ed49FaAa8FD366b",
    "remainingTransactions": 8,
    "txLimit": 8,
    "revokeTime": 15,
    "gasLimit": 60000000,
    "remainingGas": 60000000,
    "deleteMe": false,
    "timeStamp": null
  },
  {
    "address": "0xf13264C4bD595AEbb966E089E99435641082ff24",
    "remainingTransactions": 5,
    "txLimit": null,
    "revokeTime": null,
    "gasLimit": null,
    "remainingGas": 50000000,
    "deleteMe": false,
    "timeStamp": null
  },
  {
    "address": "0x00a329c0648769A73afAc7F9381E08FB43dBEA72",
    "remainingTransactions": 5,
    "txLimit": null,
    "revokeTime": 25,
    "gasLimit": 999999999999999,
    "remainingGas": 999999999999999,
    "deleteMe": false,
    "timeStamp": null
  },
    {
    "address": "0xf1326lasdfaösLDFJDLSNd23nlDD35641082ff24",
    "transactionCounter": 5,
    "txLimit": null,
```

```
36      "revokeTime": 2,
37      "gasLimit": null,
38      "remainingGas": 50000000,
39      "deleteMe": false,
40      "timeStamp": null
41    }
42  ]
```

Das Listing 3.7 zeigt die Datei `AccountList.json` nachdem der Transaktionmanger gestartet und die initiale Datei verarbeitet hat. Überall wo ein Standardwert verwendet ist der Paramater mit einem Wert von **`null`** gesetzt worden.

3.2.6 Konfiguration

Der Transaktionsmanagers kann mit der Datei `DefaultSettings.json` konfiguriert werden.

//TODO Filepath

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 letzen Resett-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.

//TODO Filepath

Die Datei hat nur einen einzigen Parameter:

Listing 3.8: TransaktionsManagerAccount.json mit einem privaten Schlüssel

```
{
  "privateKey": "0
     x4d5db4107d237df6a3d58ee5f70ae63d73d7658d4026f2eefd2f204c81682cb7"
}
```

Die unter 3.8 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.

3.2.6.1 Beispiel

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

Listing 3.9: DefaultSettings.json

```
{
  "connectionAddress": "http://jurijnas.myqnapcloud.com:8545/",
  "resetInterval": 10,
  "defaultTxLimit": 5,
  "defaultGasLimit": 50000000,
  "defaultRevokeTime": 10,
  "certifierAddress": null,
  "nameRegistryAddress": "0x0000000000000000000000000000000000001337",
  "timestampLastReset": null

}
```

Unter 3.9 ist die Datei `DefaultSettings.json` abgebildet. So sieht die Datei aus, bevor der Transaktionsmanager den Certifier deployed hat.

Listing 3.10: DefaultSettings.json

```
{
  "connectionAddress": "http://jurijnas.myqnapcloud.com:8545/",
  "resetInterval": 10,
  "defaultTxLimit": 5,
  "defaultGasLimit": 50000000,
  "defaultRevokeTime": 10,
  "certifierAddress": "0xee35211c4d9126d520bbfeaf3cfee5fe7b86f221",
  "nameRegistryAddress": "0x0000000000000000000000000000000000001337",
  "timestampLastReset": "Mar 5, 2020 1:53:50 AM"

}
```

Listing 3.10 zeigt die Datei `DefaultSettings.json` nach dem der Certifier deployed wurde. Dessen

Adresse ist nun eingetragen. Weiter ist ein Zeitstempel gesetzt.

3.2.7 Initialisierung

Wenn die Blockchain frisch aufgesetzt wird, sind inital 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. Instanzierung 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 instanziert 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 3.2.6. Hier muss beachtet werden, dass diese mit der Adresse in der Blockchainspezifkation, siehe 3.1.1.2, übereinstimmen muss.
Sobald der Certifier deployed ist, wird dessen Adresse in die Konfigurationsdatei geschrieben. Sobald die Überwachung der Blockchain gestartet wird, wird die Adresse ausgelesen und der Certifier instanziert.
Der Account für den Transaktionsmanger 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.

3.2.8 Run

Mit der Klasse `Run` wird der Transaktionsmanger für den regulären Betrieb gestartet. Vorraussetzung ist, dass der Certifier deployed und registriert ist, siehe 3.2.7.
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 konifgurierten Parameter instanziert.
3. Instanzierung von Name Registry und Certifier
   Mit der jeweiligen Adresse, wird die Name Registry und der Certifier instanziert.
4. Die Klasse `ChainInteractions` wird instanziert.
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. Instanzierung 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 mit einer Kommandozeile gestartet wurde, wird der Benutzer über aktuellen Vorgänge mit Lognachrichten infomriert.

3.2.9 Tests

//TODO - Gemachte Tests, Resultat, Schlussfolgerung

4 Fazit

// TODO

4.0.0.1 Dokumentation

Parity wird stetig weiterentwickelt. Die letzte Minorversion[55] ist im April 2019 veröffentlicht worden. Obwohl es sich um eine Minorversion handelt, hat es Änderungen in der Code-Syntaxt. Daher verhält sich das Update eher wie eine neue Majorversion[55]. Das hat zur Folge, dass praktisch alle gefundenen Tutorials nicht mehr gültig sind.

4.1 Mögliche Erweiterungen

//TODO

- Priority Queue
- JSON Dataformat
- Alles on Blockchain
- Erweiterung Parity (gratis TX from - to)
- Genesisblock
  - Deployment von Certifier
  - Registrierung von Certifier
- Report (Alle vorkommnisse ..)

5 Quellenverzeichnis

[1] Ethereum, „Home | Ethereum“, 2019. [Online]. Verfügbar unter: https://www.ethereum.org/.

[2] „Blockchain - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Blockchain.

[3] „Denial-of-service attack - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Denial-of-service_attack.

[4] M. Inc., „What is Gas | MyEtherWallet Knowledge Base“, 2018. [Online]. Verfügbar unter: https://kb.myetherwallet.com/en/transactions/what-is-gas/.

[5] „University of Applied Sciences and Arts Northwestern Switzerland“, 2019. [Online]. Verfügbar unter: https://www.fhnw.ch/.

[6] „Smart COntrct Definition“, 2019. [Online]. Verfügbar unter: https://www.investopedia.com/terms/s/smart-contracts.asp.

[7] „Web3j SDK - Where Java meets the Blockchain“, 2020. [Online]. Verfügbar unter: https://www.web3labs.com/web3j.

[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] „CryptoKitties | Collect and breed digital cats!“, 2019. [Online]. Verfügbar unter: https://www.cryptokitties.co/.

[15] „Wei“, 2019. [Online]. Verfügbar unter: https://www.investopedia.com/terms/w/wei.asp.

[16] 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.

[17] K. Tam, „Transactions in Ethereum - KC Tam - Medium“, 2019. [Online]. Verfügbar unter: https://medium.com/@kctheservant/transactions-in-ethereum-e85a73068f74.

[18] Y. Riady, „Signing and Verifying Ethereum Signaturs - Yos Riady“, 2019. [Online]. Verfügbar unter: https://yos.io/2018/11/16/ethereum-signatures/.

[19] „Remote procedure call - Wikipedia“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Remote_procedure_call.

[20] „https://keccak.team/“, 2019. [Online]. Verfügbar unter: Keccak%20Team.

[21] „Elliptic Curve Digital Signature Algorithm - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm.

[22] „SHA-3 - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/SHA-3.

[23] P. Kasireddy, „How does Ethereum work, anyway? - Preethi Kasireddy - Medium“, 2019. [Online]. Verfügbar unter: https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369.

[24] „Ethereum Series - Understanding Nonce - The Startup - Medium“, 2019. [Online]. Verfügbar unter: https://medium.com/swlh/ethereum-series-understanding-nonce-3858194b39bf.

[25] 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.

[26] „Crypto Wallet Types Explained | Binance Academy“, 2019. [Online]. Verfügbar unter: https://www.binance.vision/blockchain/crypto-wallet-types-explained.

[27] M. Wachal, „What is a blockchain wallet? - SoftwareMill Tech Blog“, 2019. [Online]. Verfügbar unter: https://blog.softwaremill.com/what-is-a-blockchain-wallet-bbb30fbf97f8.

[28] 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.

[29] 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.

[30] E. Conner, „smart Wallets are Here - Gnosis“, 2019. [Online]. Verfügbar unter: https://blog.gnosis.pm/smart-wallets-are-here-121d44519cae.

[31] 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.

[32] „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.

[33] 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.

[34] „ethereum/yellowpaper: The Yellow Paper: Ethereum’s formal specification“, 2019. [Online]. Verfügbar unter: https://github.com/ethereum/yellowpaper.

[35] go-ehereum, „Go Ethereum“, 2019. [Online]. Verfügbar unter: https://geth.ethereum.org/.

[36] P. Technologies, „Blockchain Infrastructure for the Decentralised Web | Parity Technologies“, 2019. [Online]. Verfügbar unter: https://www.parity.io.

[37] „https://github.com/ethereum/aleth“, 2019. [Online]. Verfügbar unter: https://github.com/ethereum/aleth.

[38] „ethereum/trinity: The Trinity client for the Ethereum network“, 2019. [Online]. Verfügbar unter: https://github.com/ethereum/trinity.

[39] „Rust Programming Language“, 2019. [Online]. Verfügbar unter: https://www.rust-lang.org/.

[40] „Parity Name Registry - Parity Tech Documentation“, 2020. [Online]. Verfügbar unter: https://wiki.parity.io/Parity-name-registry.html.

[41] „Permissioning - Parity Tech Documentation“, 2020. [Online]. Verfügbar unter: https://wiki.parity.io/Permissioning#how-it-works-3.

[42] „Domain Name System - Wikipedia“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Domain_Name_System.

[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] P. Technologies, „Releases - paritytech/parity-ethereum“, 2020. [Online]. Verfügbar unter: https://github.com/paritytech/parity-ethereum/releases.

[45] „TOML - Wikipedia“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/TOML.

[46] „JSON - Wikipedia“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/JSON.

[47] „Remix - Ethereum IDE“, 2020. [Online]. Verfügbar unter: https://remix.ethereum.org/.

[48] MetaMask, „MetaMask“, 2019. [Online]. Verfügbar unter: https://metamask.io/.

[49] „Empowering App Development for Deveolopers | Docker“, 2020. [Online]. Verfügbar unter: https://www.docker.com/.

[50] „Releases - web3j/web3j“, 2020. [Online]. Verfügbar unter: https://github.com/web3j/web3j/releases.

[51] „Installing the Solidity Compiler – Solidity 0.6.4 documentation“, 2020. [Online]. Verfügbar unter: https://solidity.readthedocs.io/en/develop/installing-solidity.html.

[52] „Getting Starte - web3j latest documentation“, 2020. [Online]. Verfügbar unter: https://web3j.readthedocs.io/en/latest/getting_started.html#filters.

[53] „Command pattern - Wikipedia“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Command_pattern.

[54] „PriorityQueue (Java Platform SE 7)“, 2020. [Online]. Verfügbar unter: https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html.

[55] „Software versioning“, 2020. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Software_versioning.

[56] T. B. G. 2019, „Sweet Tools for Smart Contracts“, 2019. [Online]. Verfügbar unter: https://www.trufflesuite.com/.

[57] uPort, „uPort“, 2019. [Online]. Verfügbar unter: https://www.uport.me/.

[58] A. Wallet, „Atomic Cryptocurrency Wallet“, 2019. [Online]. Verfügbar unter: https://atomicwallet.io/.

[59] E. M. Inc., „Crypte Wallet - Send, Receive & Exchange Cryptocurrency | Exodus“, 2019. [Online]. Verfügbar unter: https://www.exodus.io.

[60] MyEtherWallet, „MyEtherWallet | MEW“, 2019. [Online]. Verfügbar unter: https://www.myetherwallet.com/.

[61] Solidity, „Solidity - Solidity 0.5.11 documentation“, 2019. [Online]. Verfügbar unter: https://solidity.readthedocs.io/en/v0.5.11/.

[62] „Vyper–Vyper documentation“, 2019. [Online]. Verfügbar unter: https://vyper.readthedocs.io/en/v0.1.0-beta.13/#.

6 Anhang

6.1 Glossar

Begriff	Bedeutung

6.2 Entwicklungsumgebung

In diesem Abschnitt wird die geplante Testumgebung und deren Verwendung beschrieben.

6.2.1 Blockchain

Es wird eine Test-Blockchain aufgesetzt. Diese wird benötigt, um geschriebenen Code zu testen und analysieren.

Als Blockchain wird Ethereum[1] verwendet. In den nachfolgenden Absätzen werden mögliche Tools besprochen, die für den Aufbau von einer Testumgebung genutzt werden können.

6.2.1.1 Client

In der Arbeit wird evaluiert ob Geth[35] als Client den Ansprüchen genügt oder ob ein anderer Client (z.B. Parity[36], Aleth[37], etc.) zum Einsatz kommt.

Trufflesuite Trufflesuite[56] wird verwendet, um eine simulierte Blockchain aufzusetzen. Diese kann für die Einarbeitung in die Materie genutzt werden.

6.2.2 Wallet

Wallets werden für die Verwaltung von Benutzerkonten und deren Transaktionen benötigt. Zu den möglichen Wallets gehören z.B.:

- uPort[57]
- Metamask[48]
- Atomic Wallet [58]
- Exodus[59]

Es wird davon ausgegangen, dass keine Wallet alle Bedürfnisse abdecken kann, daher wird die gewählte Wallet im Zuge dieses Projekts erweitert. Für Ethereum existiert ein offizieller Service um eine eigene Wallet zu ertellen: MyEtherWallet[60]

6.2.3 Smart Contracts

Smart Contracts werden benötigt, um zu bestimmen, wer auf einer Blockchain gratis Transaktionen ausführen kann. Sobald eigene Smart Contracts entwickelt werden, kann die Testumgebung genutzt werden, um diese zu testen.

Programmiersprache Für die Entwicklung von Smart Contracts werden folgende zwei Sprachen evaluiert:

- Solidity[61]
- Vyper[62]

6.2.4 Docker

docker run -ti -p 8545:8545 -p 8546:8546 -p 30303:30303 -p 30303:30303/u -v ~/.local/share/io.parity.ethereum/docker/:/ parity/parity:stable –config /home/parity/.local/share/io.parity.ethereum/docker.toml –jsonrpc-interface all

6.3 Weitere Lösungsansätze

6.3.1 Super Smart Wallet

Es wird eine zentrale Smart Wallet entwickelt. Im Gegensatz zu LA 1, 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.

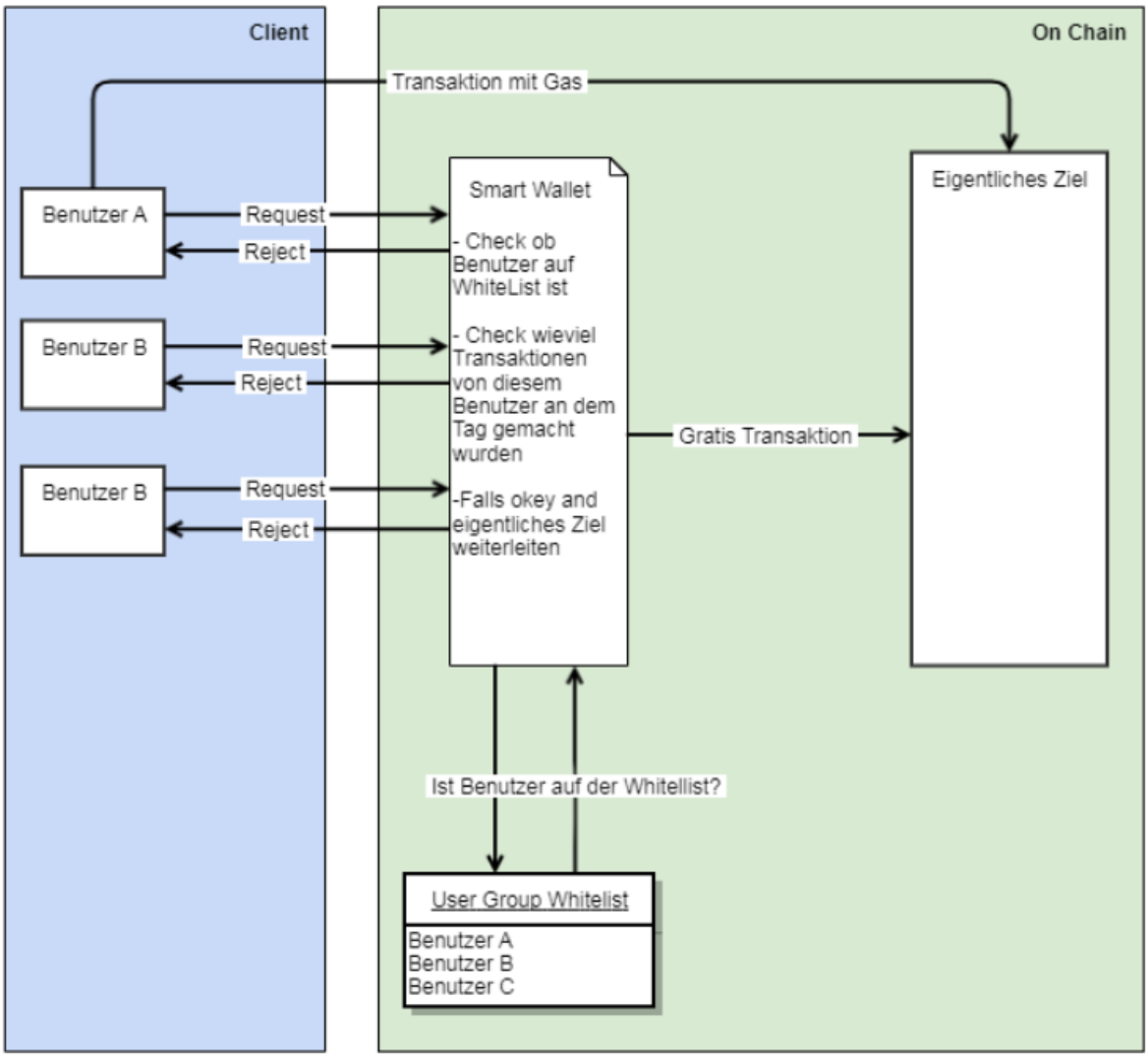

Abbildung 6.1: Super Smart Wallet

Die Smart Wallet verwaltet die Whitelist und den Schutzmechanismus gegen DoS Attacken. Das ist auf Abbildung 6.1 ersichtlich. Wird eine DoS Attacke identifiziert, wir der entsprechende Account aus der Whitelist gelöscht.

6.3.1.1 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.

6.3.1.2 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.

6.4 Abnahmekriterien

In diesem Kapitel werden alle Abnahmekriterien des Blockchain Transaktions Managers aufgelistet und Kategorisiert. Es wird zwischen funktionalen und nicht-funktionales Kriterien unterschieden. //TODO Text

Nr.	Titel	Beschreibung
1.	Bezahlte Transaktionen für alle	Jeder gültige Account kann Transaktionen mit Gas Price durchführen
2.	Gratis Transaktionen für Whitelist	Ein Account der für die Whitelist zertifiziert ist, kann Transaktionen mit Gas Price „0“ durchführen
3.	Account aus Liste für Whitelist zertifizieren	Alle Account die auf der Liste stehen sind für die Whitelist certifiziert
4.	Account aus Liste und Whitelist entfernen	Wenn ein Account gelöscht wird, wird er von der Whitelist wie auch von der Account Liste entfernt
5.	Account nach Transaktionen sperren	Ein Account der zu viele Transaktionen betätigt hat, wird für eine Zeitspanne gesperrt
6.	Account nach GasUsed sperren	Ein Account der zu viel Gas für seine Transaktionen benutzt hat wird gesperrt
7.	Gesperrte Account entsperren	Ein gesperrter Account wird nach einer gesetzten Zeitspanne wirder entsperrt
8.	Account manuell sperren	Ein Account kann manuell gesperrt werden
9.	Zeitintervall setzten	Es muss ein Zeitintervall gesetzt. Dieses gilt für die Zeitspanne wie lange Limiten und Sperrungen gelten.
10.	Sperrzeit	Die Sperrzeit ist Intervall Abhängig und gilt für alle gleich
11.	Anzahl Sperrungen speichern	Die Anzahl Sperrungen eines Accounts werden in einem File gespeichert

Nr.	Titel	Beschreibung
12.	Sperrzeit Abhängig von Anzahl Sperrungen	Die Sperrzeit wird höher, je öfter der Account gesperrt wurde
13.	Counter resetten	Der Counter aller Accounts wird nach Zeitperiode wieder auf 0 gesetzt
14.	Transaktionslimite pro User	Die Anzahl Transaktionen bis ein Account gesperrt wird, kann für jeden Account individuell eingestellt werden
15.	GasUsed Limite pro User	Die Anzahl Gas Used bis ein Account gesperrt wird, kann für jeden Account individuell eingestellt werden
16.	Default Werte	Es können default Werte für max Transaktionen und max Gas Used gesetzt werden
17.	Certifier nur von Owner deployen	Der Certifier kann nur vom Owner registriert werden
18.	Certifier nur einmal deploybar	Der Certifier kann nur einmal deplyt/registriert werden
19.	Owner Account	Der Certifier Owner Account kann nicht gesperrt werden
20.	Accounts nicht doppelt	Es gibt eine Meldung wenn Accounts doppelt in datei geführt werden
21.		

6.5 Abnahme Tests Report

6.5.1 Abnahme Test 1

AK Nr.:	**Titel:**	**Testart:**
Tester:	**Datum:**	**Status**
Vorbedingung:		
Ablauf:		

AK Nr.:	Titel:	Testart:
Erwünschtes Resultat:		
Tatsächliches Resultat		

6.5.2 Abnahme Test 2

6.5.3 Abnahme Test 3

6.5.4 Abnahme Test 4

6.5.5 Abnahme Test 5

6.5.6 Abnahme Test 6

6.5.7 Abnahme Test 7

6.5.8 Abnahme Test 8

6.5.9 Abnahme Test 9

6.6 Verlinkung von Code

Hier ist sämtlicher Code verlinkt.

6.6.1 Name Registry und Certifier

Smart Contracts von Parity

6.6.2 Transaktionsmanager

// TODO öffentliches Repository

7 Ehrlickeitserklärung

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

Datum _____

Ort _____

Faustina Bruno _____

Serge Jurij Maïkoff _____