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

Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC. This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel liefert eine ausführliche Zusammenfassung der Bachelorthesis.

1.1 Problemstellung

Die Aufgabe beinhaltet ein Blockchain Netzwerk [1] für die Fachhochschule Nordwest Schweiz[2] (FHNW) zur Verfügung zu stellen, welches von den Studierenden zu Testzwecken genutzt werden kann. Blockchains verfügen über verschiedene Mechanismen, um sich gegen Attacken abzusichern. Eine davon ist eine Gebühr auf jeder Transaktion, der sogenannte Gas Price 2.2.4 [3]. Dadurch können Denial of Service (DoS) Attacken 2.2.7 [4], bei denen das Netzwerk mit unzähligen Transaktionen geflutet wird, effizient bekämpft werden. Der Angreifer kann die Attacke nicht aufrecht erhalten, da ihm die finanziellen Mittel zwangsläufig ausgehen. Obwohl dieser Schutzmechanismus auf einer öffentlichen Blockchain sehr effizient und elegant ist, eignet er sich nicht für eine Lernumgebung. Hier sollen Anwender die Möglichkeit haben, Transaktionen ohne anfallende Gebühren ausführen zu können. Dadurch wird jedoch die Blockchain anfällig für DoS Attacken.

1.2 Ziel

Das Ziel der Arbeit ist es ein Test Blockchain Netzwerk aufzubauen, welches für eine definierte Gruppe von Benutzern gratis Transaktionen erlaubt und trotzdem ein Schutzmechanismus gegen DoS Attacken hat.

1.3 Methodik

//TODO Kapitel besprechen und beschreiben Hier wird beschrieben wie und was gemacht wurde

!!muss besprochen überarbeitet werden Wir haben zu Beginn Meilensteine und grössere Arbeitspakete definiert. Die kleineren Arbeitspakete wurden nach neugewonnen Wissen und Arbeitsstand definiert.

Durch die erarbeiteten Lösungsansaätze, der Evaluation und die Besprechung nach der Zwischenpräsentation, wurden die Meilensteine geändert und die Planung anderst gestaltet.

Agiles Vorgehen, –> mit neuem Wissen weiter geplant

//TODO möglicher Text besprechen und überarbeiten Zu Beginn wurde ein provisorischer Projekt Plan mit möglichen Arbeitspaketen und Meilensteine definiert. Da die Thematik komplett unbekannt war, wurde auf ein agiles Vorgehen gesetzt, um neue Erkentnisse in die Planung einfliessen zu lassen. Nach der Einlese- und Probierphase, wurden Lösungskonzepte konzipiert, evaluiert und an der Zwischenpräsentation dem Experten und den Betreuern präsentiert. Hier wurde das weitere Vorgehen besprochen und die neuen Meilensteine definiert. Die Arbeitspakete werden alle zwei Wochen definiert.

1.4 Strukturierung des Berichts

Der Bericht ist in einen theoretischen und praktischen Teil gegliedert. Gemachte Literaturstudiuen, geprüfte Tools, der aktuelle Stand der Ethereum Blockchain, 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 Ethereum Blockchain betreiben. Die Blockchain soll die selbe Funktionalität wie die öffentliche Ethereum Blockchain 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 und einem Schutzmechanismus wird in diesem Bericht behandelt.

2.2 Komponenten

//TODO Spellcheck

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 Denial of Servie (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.[1] Ein speziell erwähnenswerter Block, ist der sogenannte Genesisblock[5]. 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[6] 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[7] entgegengewirkt. Die Consensus Algorithemen stellen sicher, dass die Transaktionen auf der Blockchain valide und authentisch sind.

Im Gegensatz zur Bitcoin[8] kann bei Ethereum[9] 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[10] 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.

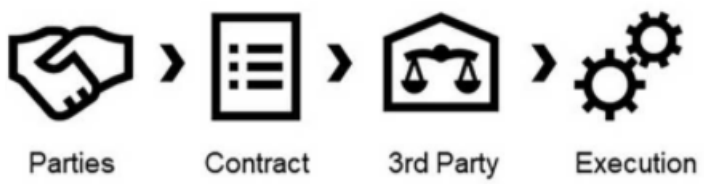

Abbildung 2.1: Ein traditioneller Vertag[11]

Abbildung 2.2: Ein Smart Contract[11]

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[12]. 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[13] 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[14] 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 [15]

Damit eine Transaktion in die Blockchain aufgenommen werden kann, muss sie signiert[16] 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)[17] gemacht.

2.2.4 Gas

Mit Gas[3] 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 Berchnung 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[18] verwendet, ein „Elliptical Curve Digital Signature Algorithm“[19]. Der öffentliche Schlüssel wird verwendet um die Signatur einer Transaktion zu verifizieren.

Adresse Die Adresse wird aus dem öffentlichen Schlüssel abgeleitet. Es wird SHA3[20] verwendet um einen 32 Byte langen String zu bilden. Von diesem bilden die letzten 20 Bytes, also 40 Zeichen, die Adresse von einem 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 initieren, sondern reagieren nur auf zuvor eingegangene. Das wird auf der Abbildung 2.3 dargestellt. Ein Benutzer Accounts wird als „Externally owned account“ bezeichnet.

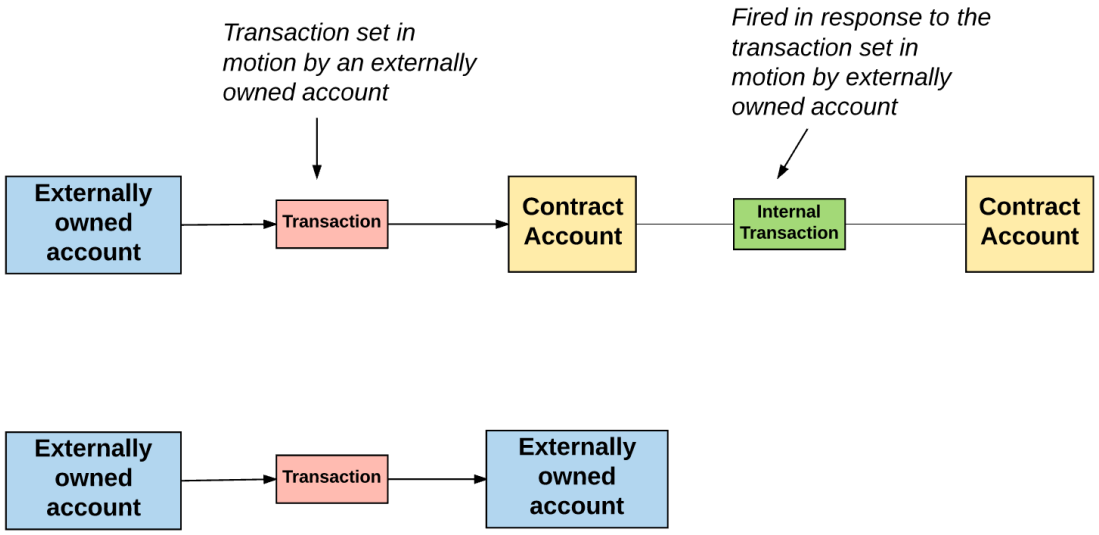

Abbildung 2.3: TX zwischen Accounts

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[21]) des Benutzer Accounts, der das Deployment vornimmt.[22]

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. [23], [24], [25]

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. [23], [24], [25]

2.2.6.1 Smart Wallet

Smart Wallets basieren auf Smart Contracts. Der Benutzer ist der Besitzer der Smart Contracts und somit der Wallet. Die Verwendung von Smart Contract bei der implementierung der Wallet ermöglicht mehr Benutzerfreundlichkeit ohne die Sicherheit zu kompromittieren. [26], [27], [28] //TODO ..

2.2.7 Denial of Service (DoS) Attacken

//TODO ergänzen

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

Zurzeit sind Blockchains noch relativ langsam bei der Verarbeitung von Transaktionen. Ethereum kann ungefähr 15 Transaktionen pro Sekunde abarbeiten.[29] 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 Vorgehen 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.[30]

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 potentielle Bedrohung. Ohne Transaktionskosten hat die Blockchain keinen Schutzmechanismus gegen 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[31] implementieren. Das Protokoll definiert die minimal Anforderungen an den Clienten. Das erlaubt, dass der Client in verschiedenen Sprachen, von verschiedenen Teams, realisiert werden kann. Nebst der verwendeten Programmiersprache, unterscheiden sich die Clienten bei implementierten Zusatzfunktionen, die im Protokoll nicht spezifiziert sind. Die populärsten Clients sind Go Ethereum (GETH)[32], Parity[33], Aleth[34] und Trinity[35]. Die Clients wurden auf die Zusatzfunktionalität untersucht, für eine definierte Gruppe von Accounts gratis Transaktionen zu ermöglichen.

2.3.1 Parity

Geschrieben in Rust[36], ist es der zweit populärste Client nach Geth[32]. 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[37] und der Service Transaction Checker[38]. 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[39].

Die Name Registry ist in Parity standardmässig immer unter der selben 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[40] der Name Registry aufgeführt und erklärt. Der vollständige Code ist im Anhang unter 6.6 zu finden.

```
1     struct Entry {
2         address owner;
3         address reverse;
```

```
4           bool deleted;
5           mapping (string => bytes32) data;
6       }
7
8       mapping (bytes32 => Entry) entries;
```

In der Map `entries` sind alle registrierten Accounts festgehalten. 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.

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

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.

```
1       modifier onlyOwnerOf(bytes32 _name) {
2           require(entries[_name].owner == msg.sender);
3           _;
4       }
```

Der Besitzer einer Nachricht wird 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.

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

Prüft ob der Eintrag für 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.

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

Auf Zeile 1 ist 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.

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

Mit der Methode `reserve` kann ein Eintrag in der Liste `entries` für den Namen `_name` reserviert werden. 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` eine neuer Eintrag erstellt. Als Besitzer des Eintrags wird der Sender der Transaktion gesetzt. Auf Zeile 9 wird die erfolgreiche Reservierung ans Netzwerk gesendet.

```
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 dieser Methode 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 gesammte Code ist im Anhang unter 6.7 zu finden.

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

Die zentrale Datenstrucktur des Certifiers, die Whitelist. In der Liste `certs` sind zertifizierte Accounts gespeichert.

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

Auf Zeile 1 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.

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

Mit dieser 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.

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

Mit der Methode `certified` kann jederzeit überprüft werden, ob ein Account (`_who`) zertifizierert 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.

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

2.3.2 Geprüfte Alternativen

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

2.4 Lösungsansätze

//TODO Spellcheck über ganze Seite

//TODO Erläuterungen zu Flow Charts

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. Wie in Abbildung **??** ersichtlich, wird für jeden Benutzer eine Smart Wallet deployed. Dies wird von der FHNW übernommen. So fallen für die Benutzer keine Transaktionsgebühren an. Wie unter 2.3.1 beschrieben, wird für die Betreibung der Blockchain der Client Parity mit einer Withelist verwendet.

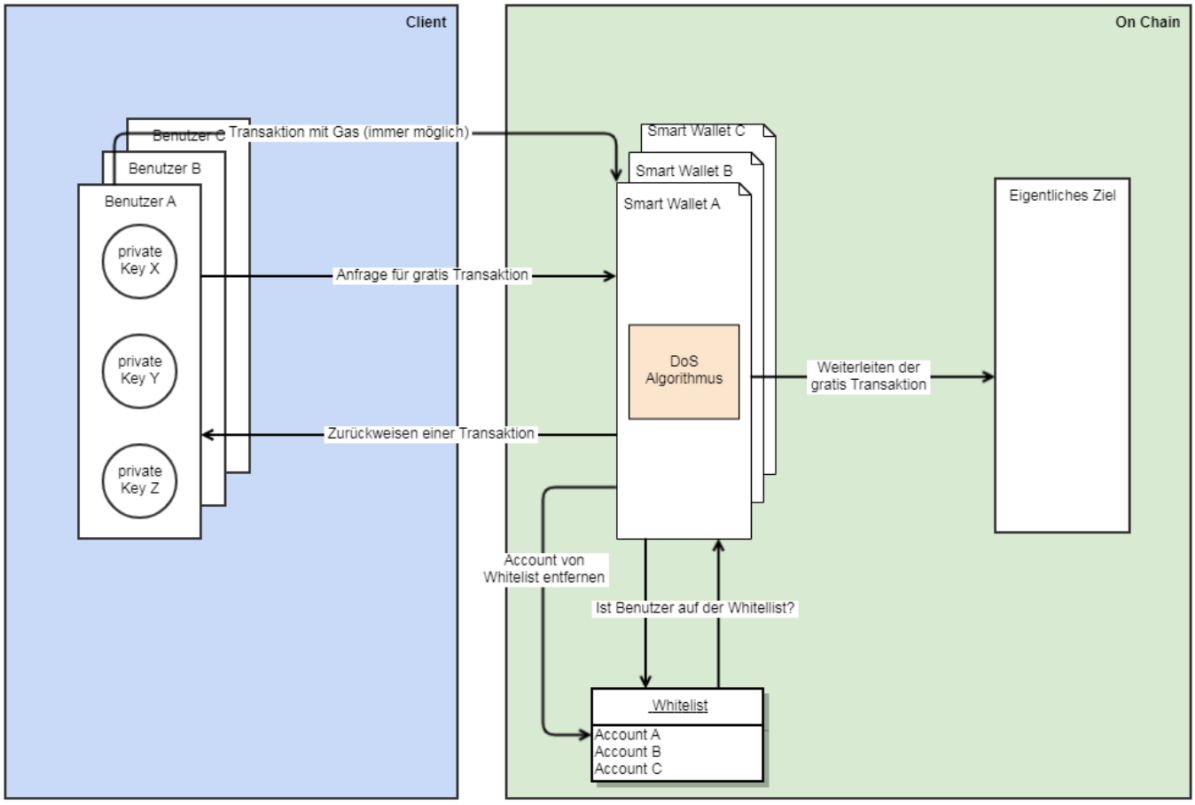

Abbildung 2.4: Architektur mit Smart Wallet

Es muss sichergestellt werden, dass ein Benutzer auf seine Smart Wallet zugreifen kann, unabhängig davon ob er gratis Transaktionen tätigen darf oder nicht. Dies ist in der Abbildung 2.4 dargestellt.

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 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[36], benötigt. Ein weiterer Nachteil ist, dass bei einer Änderung am DoS Schutzalgorithmus eine neue Smart Wallet für jeden Account deployed werden muss. Das bedingt, dass die Whitelist ebenfalls mit den neuen Accounts aktualisiert wird.

Prozessworkflow In der Abbildung 2.5 ist der Prozessablauf für eine gratis Transaktion dargestellt.

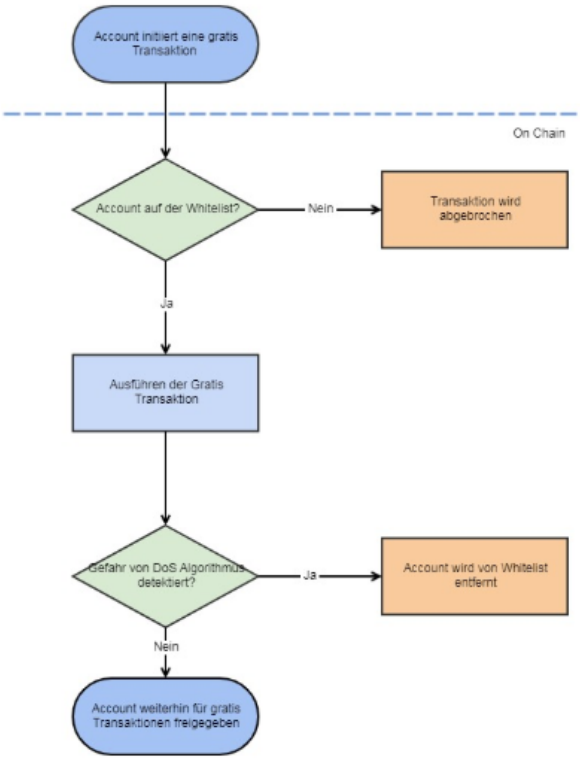

Abbildung 2.5: Flowchart für Smart Wallet

//TODO weitere Erläuterung?

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 implementiert, dargestellt in Abbildung 2.6.

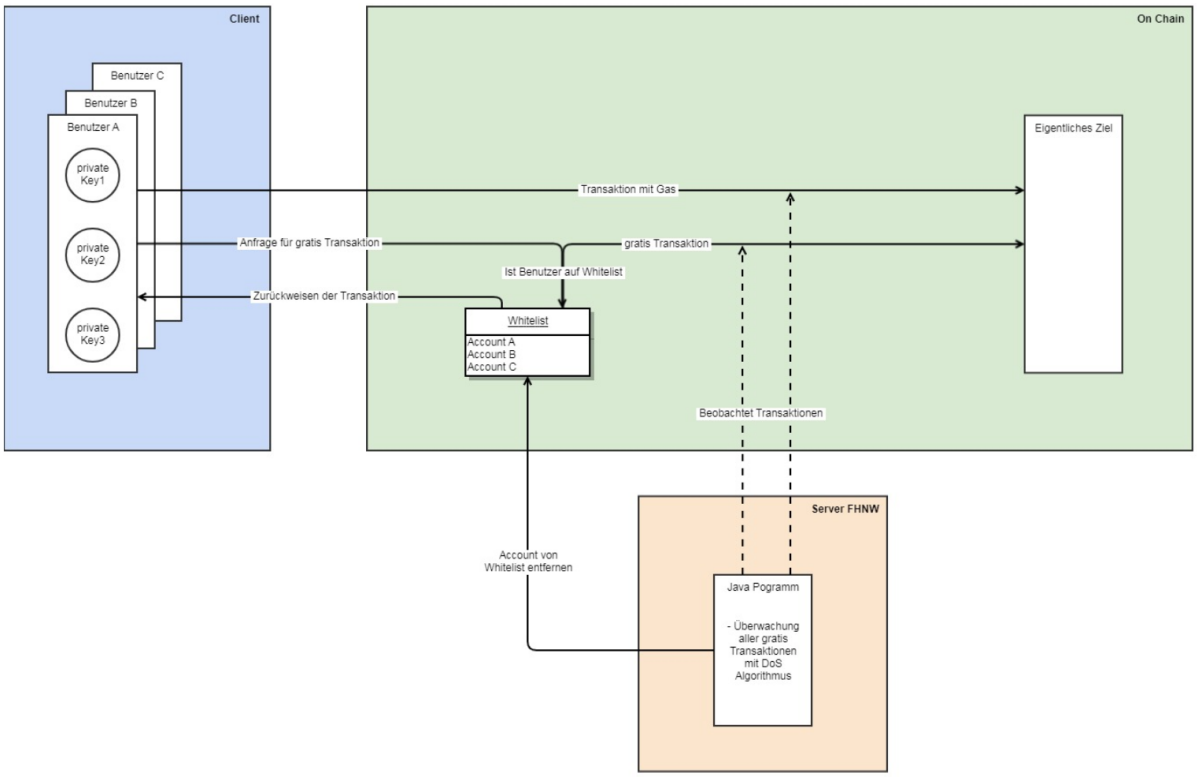

Abbildung 2.6: Externes Programm für die Verwaltung der Whitelist

Es wird auch für diesen Ansatz die Whitelist von Parity verwendet, siehe 2.3.1. Im externen Programm werden alle gratis Transaktionen analysiert, die das Blockchain Netzwerk erreichen. Das Programm verfügt über einen eigenen Benutzer 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.

Transaktionen für die ein Transaktionsgebühren gezahlt werden sind immer möglich. Diese werden vom externen Programm auch nicht überwacht. Die anfallenden Gebühren sind Schutz genug.

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 deployed werden. Eine aktualisierung der Whitelist ist nicht nötig.

Contra Es wird das Hauptprinzip, Dezentralität, einer Blockchain verletzt. Das externe Programm ist eine zentrale Authorität, die von der FHNW kontrolliert wird. Durch das externen Programm kommt eine weitere Komponente dazu. Diese muss ebenfalls administriert werden.

Prozessworkflow //TODO Flowchart falsch.. gibt keine Smart wallet, Transaktion kommt immer durch Java wenn auf white list, da java nur passiv mithört

Auf dem Flowchart 2.7 dargestellt ist, kann ein Benutzer mit einem whitelisted Account direkt gratis Transaktionen ausführen.

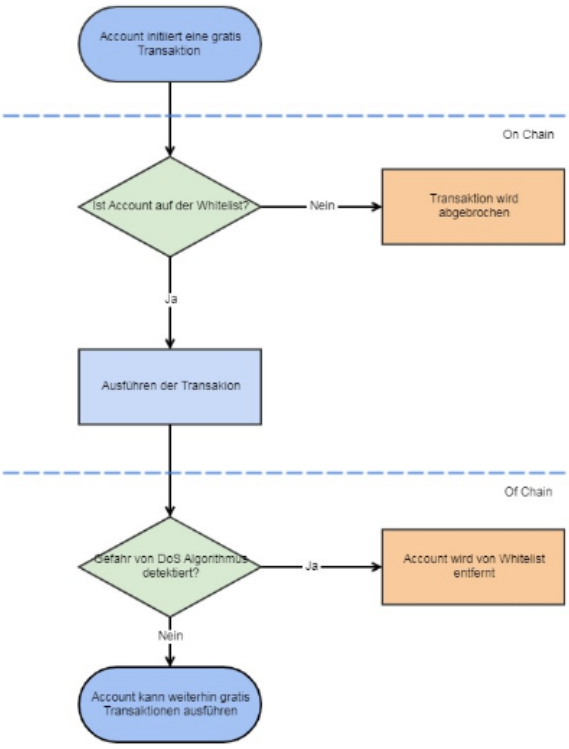

Abbildung 2.7: Flowchart externes Programm für die Verwaltung der Whitelist

2.4.1.3 ALA 3: Externes Programm mit Whitelist

Wie in Abbildung 2.8 illustriert, ist der Blockchain ein externes Programm vorgelagert. Das Programm verwaltet eine eigene Whitelist mit Accounts. Diese sind für gratis Transaktionen berechtigt. Weiter beinhaltet es den DoS Schutzalgorithmus. Dieser prüft ob der Account auf der Whitelist ist und ob die Transaktion die Schutzrichtlinien verletzt. Falls ein Account die Sicherheitsrichtlinien verletzt, wird dieser vom Algorithmus aus der eigenen Whitelist gelöscht.

Sofern keine Richtlinien verletzt werden, wird die Transaktion ins Data-Feld, siehe 2.2.3, einer neuen Transaktion gepackt. Das ist nötig, um die Transaktionsinformationen (wie z.B. Sender Identität) zu präservieren. Die neue erstellte Transaktion wird vom Programm an die Smart Wallet gesendet.

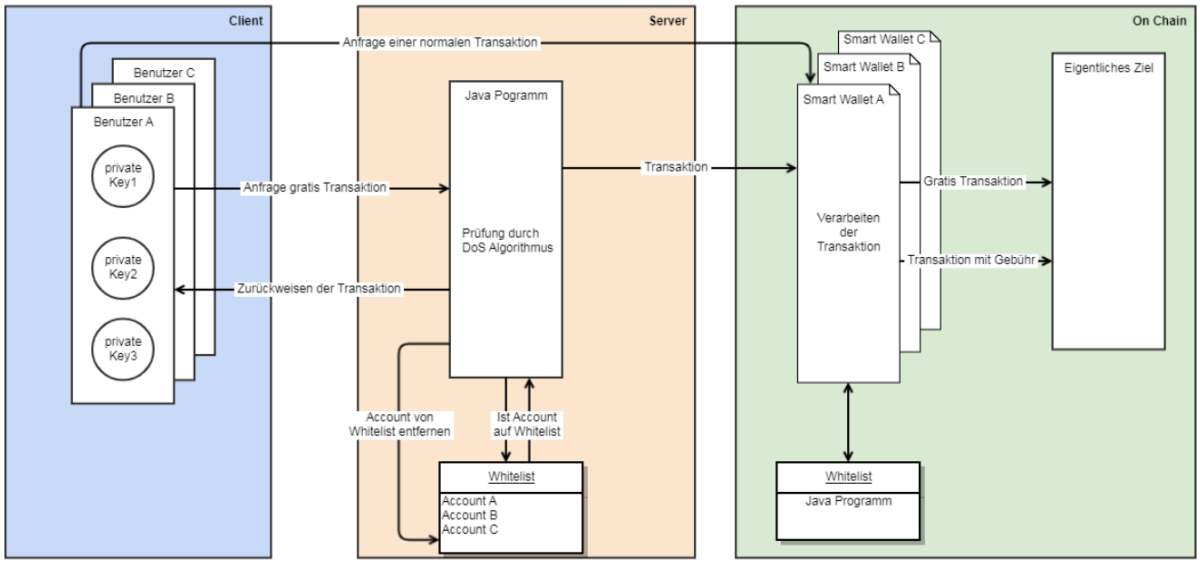

Abbildung 2.8: Externes Programm mit Whitelist

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.

Pro Dieser Ansatz ist in der gegeben Zeit umsetzbar. Falls eine Anpassung des DoS Schutzalgorithmus nötig ist, muss nur das externe Programm neu deployed werden. Eine aktualisierung der Whitelist ist nicht nötig.

Contra Es wird das Hauptprinzip, Dezentralität, einer Blockchain verletzt. Das externe Programm ist eine zentrale Authorität, die von der FHNW kontrolliert wird. Durch das externen Programm kommt

eine weitere Komponente dazu. Diese muss ebenfalls administriert werden. Dieser Ansatz bietet keine Vorteile im Vergleich zum LA 2, ist aber mit der Verschachtelung von Transaktionen komplexer.

Prozessworkflow //Todo flowchart falsch, zuerst Java dann richtige smart wallet

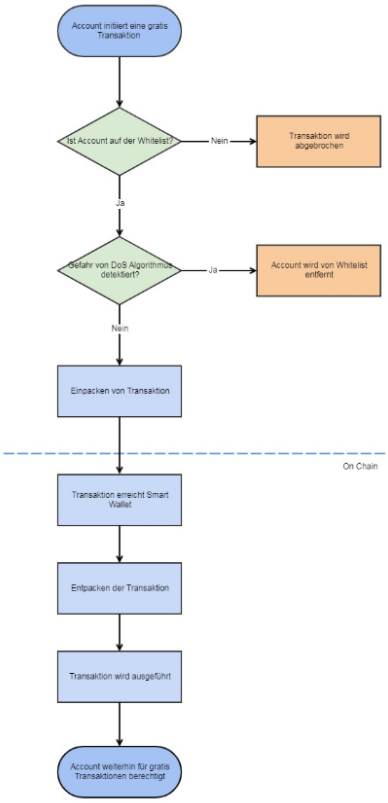

Abbildung 2.9: Flowchart externes Programm mit Whitelist

Die Abbildung 2.9 zeigt, dass alle gratis Transaktionen in erster Instanz von einem Programm geprüft werden. Falls keine Richtlinien verletzt werden, wird die Transaktion im Data-Feld einer neu generierten Transaktion an die Smart Wallet übermittelt.

2.4.2 Evaluation der Architektur

Die erarbeiteten Lösungsansätze werden gegeneinander verglichen. Um zu bestimmen, welcher Ansatz weiter verfolgt 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 ein 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 von einem externen Programm, 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 bedeuted 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 von einem externen Programm, 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.

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.

//TODO ausfleischen?

2.4.3 DoS-Algorithmus

//TODO Spellcheck

In diesem Abschnitt sind die Komponenten des DoS-Algorithmus aufgeführt. Der Algorithmus wird verwendet um getätigte gratis Transaktionen zu überwachen und falls nötig einzuschränken. Wird ein Account als Bedrohung für die Blockchain eingestuft, wird dieser Account von der Whitelist gelöscht.

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 Berurteilung 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.

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

Allgemeines Intervall Gratis Transaktionen werden für alle Accounts im selben Zeitintervall betrachtet. Der Zeitpunkt ist relativ zum Programmstart. Beispielsweise ist als Intervall eine Stunde gesetzt und der 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 den selben 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 fixer Zeitpunkt definiert, bei dem alle Accounts zurückgesetzt werden. Das heisst das Kontingent wird bei allen Accounts wieder auf den konfigurierten Wert gesetzt. Von der Whitelist gelöschte Accounts werden dieser wieder hinzugefügt. Zum Beispiel könnte als Zeitpunkt Montag 8:00 UTC definiert werden.

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.

Beispiel:

# Verstösse	Dauer Sperrung
1	0.50
2	1.00
3	3.00
4	12.00
5	60.00
6	360.00

In der oben aufgeführten Tabelle ist ersichtlich, dass die Dauer der Sperrung proportional zu den Verstössen ist.

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.

Reset-Intervall Wir haben uns für die Implementierung eines allgemeinen Intervalls entschieden. Der Ansatz ist bedeutend einfacher umzusetzen als ein individuelles Intervall und kann daher sicher in der gegebenen Zeit realisiert werden. Am Ende des Intervalls, werden die Zähler für alle Parameter pro Account zurückgesetzt.

Die Auswirkung des genannten Nachteils beim allgemeinen Intervall ist stark von dessen Länge abhängig. Je kürzer das Intervall gewählt wird, umso kleiner sind die möglichen Folgen.

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 gleich behandelt. Wie lange ein Account keine gratis Transaktionen mehr tätigen kann, ist abhängig davon, zu welchem Zeitpunkt er von der Whitelist gelöscht wird. Wenn der gesetzte Zeitpunkt dem Benutzer bekannt ist, kann das System missbraucht werden. Wird ein DoS Angriff kurz vor dem Resetzeitpunkt ausführt, hat es praktisch keine Folgen für den Benutzer. Sein Account wird zwar von der Whitelist entfernt, aber mit dem entsprechendem Zeitmanagement gleich wieder entsperrt.

Mit einem Zeitintervall werden alle Accounts gleich lange von der Whitelist gelöscht. Dieser Ansatz bietet daher mehr Fairness als ein fixer Zeitpunkt.

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.
Einmalige Verstösse die versehentlich auftreten werden in einer Lernumgebung als wahrschneinlich eingeschätzt. Mit diesem System werden solche Versehn sehr milde bestraft.

Wir haben uns entschieden, eine Kombination aus einem fixen Zeitpunkt und einem inkrementierenden Intervall zu verwenden. Dieser Ansatz ist in der gegebenen Zeit realisierbar und bietet nebst einem effizienten Schutz auch eine Toleranz für einmalige Verstösse. Die Dauer einer Suspendierung von der Whitelist kann mit dem Parameter „Revoke-Faktor“ konfiguriert werden. Als Basis wird das Reset-Intervall verwendet.

$t = resInter * revFak * v$

Wobei t die Dauer der Suspendierung, $resInter$ das Reset-Intervall, $revFak$ der Revoke-Faktor und v die Anzahl bereits begangener Verstösse abbilden.

Anbei Beispiel mit einem Reset-Intervall von fünf Minuten, einem Revoke-Faktor von 3 und der daraus resultierenden Suspendierung von der Whitelist in Minuten:

resInter	revFak	Verstösse	Suspendierung (min)
5	3	1	15
5	3	2	30
5	3	3	45
5	3	4	60
5	3	5	75
5	3	6	90

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.

Um zu verhindern, dass das externe Programm angreifbar wird, kann das Reset-Intervall nur global definiert werden. Bei einem individuellen Reset-Intervall müsste für jeden Verstoss einer neuer Thread im Programm gestartet werden. Dadurch würde das Programm selbst anfällig für eine DoS Attacke.

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

```
1 Definiert die maximale Anzahl gratis Transaktionen die pro Reset-
    Intervall getätigt werden können.
```

Gratis Gas

```
1 Definirt die maximale Menge an Gas die mit gratis Transaktionen
    innerhalb eines Reset-Intervalls verbraucht werden können.
```

Wenn für einen Account individuelle Schwellenwerte für Transaktionen und Gas definiert werden, müssen immer beide Parameter gesetzt werden.

Folgende Parameter gelten für alle Accounts:

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

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

Standardwert gratis Transaktionen Giltet für Accounts die ohne Parameter erfasst werden. Definiert die maximale Anzahl gratis Transaktionen die pro Reset-Intervall getätigt werden können.

Standardwert gratis Gas Giltet für Accounts die ohne Parameter erfasst werden. Definirt die maximale Menge an Gas die mit gratis Transaktionen innerhalb eines Reset-Intervalls verbraucht werden können.

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

2.5 Externes Programm für die Verwaltung der Whitelist

//TODO

UML Diagramme und so scheiss

2.5.1 DoS Algorithmus

//TODO

Spezifikation und so scheiss

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[41] 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[42].

```
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[43] 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[44] oder Metamask[45] 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-Presi 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 standar-

mässig 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.

```
{
    "name": "BachelorBlockChain",
    "engine": {
        "instantSeal": {
            "params": {}
        }
    },
    "params": {
        "networkID" : "0x11",
        "registrar" : "0x0000000000000000000000000000000000001337",
        "maxCodeSize": 24576,
        "maximumExtraDataSize": "0x20",
        "minGasLimit": "0x1388",
        "gasLimitBoundDivisor": "0x0400"
    },
    "genesis": {
        "seal": {
            "generic": "0x0"
        },
        "difficulty": "0x20000",
        "gasLimit": "0x7A1200"
    },
    "accounts": {
```

```
24          "0000000000000000000000000000000000001337": { "balance": "1", "
                constructor": "Platzhalter für Bytecode von SimpleRegistry"
                },
25          "00a329c0648769a73afac7f9381e08fb43dbea72": { "balance": "
                160693804425899027554196209234116260252220299378279283530137 6
                " }
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 `SimpleRegistry`. Dieser Parameter und der dazugehörende Smart Contract halten und verwalten die Whitelist in Parity. Sobald eine Transaktion ohne Gas Preis auf dem Node eintrifft, wird der Smart Contract an dieser Adresse verwendet, um zu prüfen ob eine gratis Transaktion erlaubt ist oder nicht.

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 erlaubt es, 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[46] für die Betreibung der Blockchain verwendet. Mehr Details zur Verwendung von Docker sind im Anhang unter 6.2.4 vorhanden.

3.1.3 Account für gratis Transaktionen zertifizieren

3.2 Schutz vor DoS Attacken

3.2.1 Einzulesende Datei

4 Fazit

4.0.0.1 Dokumentation

Parity wird stetig weiterentwickelt. Die letzte Minorversion[47] 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[47]. Das hat zur Folge, dass praktisch alle gefundenen Tutorials nicht mehr gültig sind.

5 Quellenverzeichnis

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

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

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

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

[5] „Genesis block - Bitcoin Wiki“, 2019. [Online]. Verfügbar unter: https://en.bitcoin.it/wiki/Genesis_block.

[6] „Peer-to-peer - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Peer-to-peer.

[7] „What Is a Blockchain Consensus Algorithmen | Binance Academy“, 2019. [Online]. Verfügbar unter: https://www.binance.vision/blockchain/what-is-a-blockchain-consensus-algorithm.

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

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

[10] „Nick Szabo - Wikipedia“, 2019. [Online]. Verfügbar unter: https://en.wikipedia.org/wiki/Nick_Szabo.

[11] „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.

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

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

[14] 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.

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

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

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

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

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

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

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

[28] 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.

[29] „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.

[30] 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.

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

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

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

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

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

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

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

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

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

[40] „Common Patterns - Solidity 0.4.24 documentation“, 2020. [Online]. Verfügbar unter: https://solidity.readthedocs.io/en/v0.4.24/common-patterns.html#restricting-access.

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

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

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

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

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

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

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

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

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

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

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

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

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

[54] „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[9] 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[32] als Client den Ansprüchen genügt oder ob ein anderer Client (z.B. Parity[33], Aleth[34], etc.) zum Einsatz kommt.

Trufflesuite Trufflesuite[48] 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[49]
- Metamask[45]
- Atomic Wallet [50]
- Exodus[51]

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[52]

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[53]
- Vyper[54]

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.

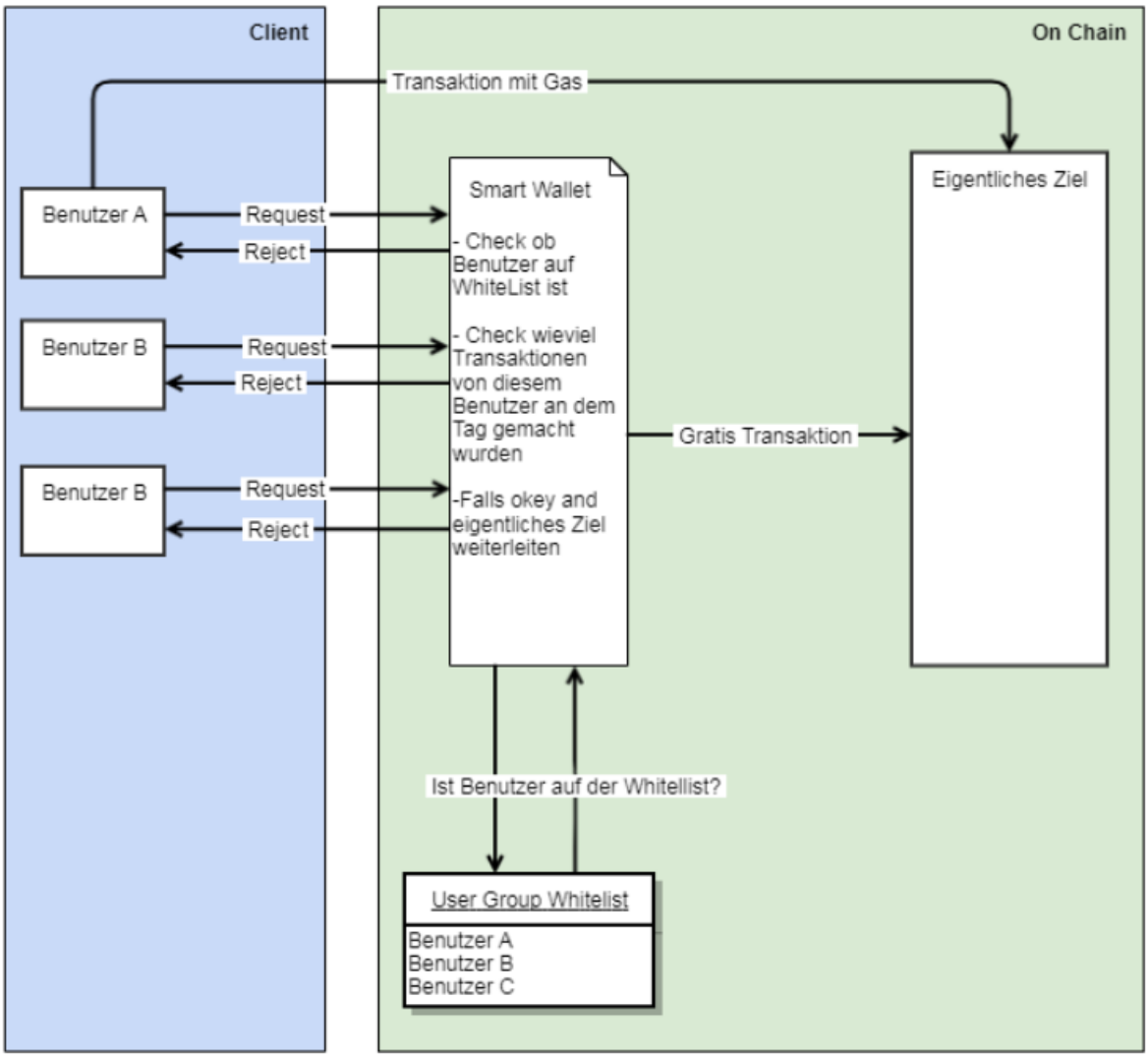

Abbildung 6.1: Super Smart Wallet

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 certifiziert ist, kann Transaktionen mit Gas Price „0“ durchführen
3.	Account aus Liste für Whitelist certifizieren	Alle Account die auf der Liste stehen sind für die Whitelist certifiziert

Nr.	Titel	Beschreibung
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 20 Transaktionen sperren	Ein Account der 20 Transaktionen betätigt hat, wird für eine Zeitspanne gesperrt
6.	Gesperrte Account entsperren	Ein gesperrter Account wird nach einer gesetzten Zeitspanne wirder entsperrt
7.	Account manuell sperren	Ein Account kann manuell gesperrt werden
8.	Counter resetten	Der Counter aller Accounts wird nach einer Woche wieder auf 0 gesetzt

6.5 Abnahme Tests Report

6.5.1 Abnahme Test 1

AK Nr.:	Titel:	Testart:
Tester:	Datum:	Status
Vorbedingung:		
Ablauf:		
Erwünschtes Resultat:		
Tatsächliches Resuktat		

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 Registry

6.6.1 ABI

```
1 [
2     {"constant":false,"inputs":[{"name":"_new","type":"address"}],"name
          ":"setOwner","outputs":[],"payable":false,"type":"function"},
3     {"constant":false,"inputs":[{"name":"_who","type":"address"}],"name
          ":"certify","outputs":[],"payable":false,"type":"function"},
4     {"constant":true,"inputs":[{"name":"_who","type":"address"},{"name"
          :"_field","type":"string"}],"name":"getAddress","outputs":[{"
          name":"","type":"address"}],"payable":false,"type":"function"},
5     {"constant":false,"inputs":[{"name":"_who","type":"address"}],"name
          ":"revoke","outputs":[],"payable":false,"type":"function"},
6     {"constant":true,"inputs":[],"name":"owner","outputs":[{"name":"","
          type":"address"}],"payable":false,"type":"function"},
```

```
7       {"constant":true,"inputs":[],"name":"delegate","outputs":[{"name":"
            ","type":"address"}],"payable":false,"type":"function"},
8       {"constant":true,"inputs":[{"name":"_who","type":"address"},{"name"
            :"_field","type":"string"}],"name":"getUint","outputs":[{"name":
            "","type":"uint256"}],"payable":false,"type":"function"},
9       {"constant":false,"inputs":[{"name":"_new","type":"address"}],"name
            ":"setDelegate","outputs":[],"payable":false,"type":"function"},
10      {"constant":true,"inputs":[{"name":"_who","type":"address"}],"name"
            :"certified","outputs":[{"name":"","type":"bool"}],"payable":
            false,"type":"function"},
11      {"constant":true,"inputs":[{"name":"_who","type":"address"},{"name"
            :"_field","type":"string"}],"name":"get","outputs":[{"name":"","
            type":"bytes32"}],"payable":false,"type":"function"}
12  ]
```

6.6.2 Owned.sol

```
1   //! The owned contract.
2   //!
3   //! Copyright 2016 Gavin Wood, Parity Technologies Ltd.
4   //!
5   //! Licensed under the Apache License, Version 2.0 (the "License");
6   //! you may not use this file except in compliance with the License.
7   //! You may obtain a copy of the License at
8   //!
9   //!     http://www.apache.org/licenses/LICENSE-2.0
10  //!
11  //! Unless required by applicable law or agreed to in writing, software
12  //! distributed under the License is distributed on an "AS IS" BASIS,
13  //! WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
        implied.
14  //! See the License for the specific language governing permissions and
15  //! limitations under the License.
16
17  pragma solidity ^0.4.24;
18
19
20  contract Owned {
21      event NewOwner(address indexed old, address indexed current);
22
23      address public owner = msg.sender;
24
25      modifier onlyOwner {
26          require(msg.sender == owner);
27          _;
28      }
29
30      function setOwner(address _new)
31          external
```

```
32          onlyOwner
33      {
34          emit NewOwner(owner, _new);
35          owner = _new;
36      }
37  }
```

6.6.3 Registry.sol

```
1   //! The registry interface.
2   //!
3   //! Copyright 2016 Gavin Wood, Parity Technologies Ltd.
4   //!
5   //! Licensed under the Apache License, Version 2.0 (the "License");
6   //! you may not use this file except in compliance with the License.
7   //! You may obtain a copy of the License at
8   //!
9   //!     http://www.apache.org/licenses/LICENSE-2.0
10  //!
11  //! Unless required by applicable law or agreed to in writing, software
12  //! distributed under the License is distributed on an "AS IS" BASIS,
13  //! WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
        implied.
14  //! See the License for the specific language governing permissions and
15  //! limitations under the License.
16
17  pragma solidity ^0.4.24;
18
19
20  interface MetadataRegistry {
21      event DataChanged(bytes32 indexed name, string key, string plainKey
            );
22
23      function getData(bytes32 _name, string _key)
24          external
25          view
26          returns (bytes32);
27
28      function getAddress(bytes32 _name, string _key)
29          external
30          view
31          returns (address);
32
33      function getUint(bytes32 _name, string _key)
34          external
35          view
36          returns (uint);
37  }
38
```

```
39
40 interface OwnerRegistry {
41     event Reserved(bytes32 indexed name, address indexed owner);
42     event Transferred(bytes32 indexed name, address indexed oldOwner,
           address indexed newOwner);
43     event Dropped(bytes32 indexed name, address indexed owner);
44
45     function getOwner(bytes32 _name)
46         external
47         view
48         returns (address);
49 }
50
51
52 interface ReverseRegistry {
53     event ReverseConfirmed(string name, address indexed reverse);
54     event ReverseRemoved(string name, address indexed reverse);
55
56     function hasReverse(bytes32 _name)
57         external
58         view
59         returns (bool);
60
61     function getReverse(bytes32 _name)
62         external
63         view
64         returns (address);
65
66     function canReverse(address _data)
67         external
68         view
69         returns (bool);
70
71     function reverse(address _data)
72         external
73         view
74         returns (string);
75 }
```

6.6.4 SimpleRegistry.sol

```
1 //! The simple registry contract.
2 //!
3 //! Copyright 2016 Gavin Wood, Parity Technologies Ltd.
4 //!
5 //! Licensed under the Apache License, Version 2.0 (the "License");
6 //! you may not use this file except in compliance with the License.
7 //! You may obtain a copy of the License at
8 //!
```

```
9  //!     http://www.apache.org/licenses/LICENSE-2.0
10 //!
11 //! Unless required by applicable law or agreed to in writing, software
12 //! distributed under the License is distributed on an "AS IS" BASIS,
13 //! WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
14 //! See the License for the specific language governing permissions and
15 //! limitations under the License.
16
17 pragma solidity ^0.4.24;
18
19 import "./Owned.sol";
20 import "./Registry.sol";
21
22
23 contract SimpleRegistry is Owned, MetadataRegistry, OwnerRegistry,
       ReverseRegistry {
24     struct Entry {
25         address owner;
26         address reverse;
27         bool deleted;
28         mapping (string => bytes32) data;
29     }
30
31     event Drained(uint amount);
32     event FeeChanged(uint amount);
33     event ReverseProposed(string name, address indexed reverse);
34
35     mapping (bytes32 => Entry) entries;
36     mapping (address => string) reverses;
37
38     uint public fee = 1 ether;
39
40     modifier whenUnreserved(bytes32 _name) {
41         require(!entries[_name].deleted && entries[_name].owner == 0);
42         _;
43     }
44
45     modifier onlyOwnerOf(bytes32 _name) {
46         require(entries[_name].owner == msg.sender);
47         _;
48     }
49
50     modifier whenProposed(string _name) {
51         require(entries[keccak256(bytes(_name))].reverse == msg.sender)
               ;
52         _;
53     }
54
55     modifier whenEntry(string _name) {
56         require(
```

```
57              !entries[keccak256(bytes(_name))].deleted &&
58              entries[keccak256(bytes(_name))].owner != address(0)
59          );
60          _;
61      }
62
63      modifier whenEntryRaw(bytes32 _name) {
64          require(
65              !entries[_name].deleted &&
66              entries[_name].owner != address(0)
67          );
68          _;
69      }
70
71      modifier whenFeePaid {
72          require(msg.value >= fee);
73          _;
74      }
75
76      // Reservation functions
77      function reserve(bytes32 _name)
78          external
79          payable
80          whenUnreserved(_name)
81          whenFeePaid
82          returns (bool success)
83      {
84          entries[_name].owner = msg.sender;
85          emit Reserved(_name, msg.sender);
86          return true;
87      }
88
89      function transfer(bytes32 _name, address _to)
90          external
91          whenEntryRaw(_name)
92          onlyOwnerOf(_name)
93          returns (bool success)
94      {
95          entries[_name].owner = _to;
96          emit Transferred(_name, msg.sender, _to);
97          return true;
98      }
99
100     function drop(bytes32 _name)
101         external
102         whenEntryRaw(_name)
103         onlyOwnerOf(_name)
104         returns (bool success)
105     {
106         if (keccak256(bytes(reverses[entries[_name].reverse])) == _name
                ) {
```

```
107             emit ReverseRemoved(reverses[entries[_name].reverse],
                    entries[_name].reverse);
108             delete reverses[entries[_name].reverse];
109         }
110         entries[_name].deleted = true;
111         emit Dropped(_name, msg.sender);
112         return true;
113     }
114
115     // Data admin functions
116     function setData(bytes32 _name, string _key, bytes32 _value)
117         external
118         whenEntryRaw(_name)
119         onlyOwnerOf(_name)
120         returns (bool success)
121     {
122         entries[_name].data[_key] = _value;
123         emit DataChanged(_name, _key, _key);
124         return true;
125     }
126
127     function setAddress(bytes32 _name, string _key, address _value)
128         external
129         whenEntryRaw(_name)
130         onlyOwnerOf(_name)
131         returns (bool success)
132     {
133         entries[_name].data[_key] = bytes32(_value);
134         emit DataChanged(_name, _key, _key);
135         return true;
136     }
137
138     function setUint(bytes32 _name, string _key, uint _value)
139         external
140         whenEntryRaw(_name)
141         onlyOwnerOf(_name)
142         returns (bool success)
143     {
144         entries[_name].data[_key] = bytes32(_value);
145         emit DataChanged(_name, _key, _key);
146         return true;
147     }
148
149     // Reverse registration functions
150     function proposeReverse(string _name, address _who)
151         external
152         whenEntry(_name)
153         onlyOwnerOf(keccak256(bytes(_name)))
154         returns (bool success)
155     {
156         bytes32 sha3Name = keccak256(bytes(_name));
```

```
157         if (entries[sha3Name].reverse != 0 && keccak256(bytes(reverses[
                entries[sha3Name].reverse])) == sha3Name) {
158             delete reverses[entries[sha3Name].reverse];
159             emit ReverseRemoved(_name, entries[sha3Name].reverse);
160         }
161         entries[sha3Name].reverse = _who;
162         emit ReverseProposed(_name, _who);
163         return true;
164     }
165 
166     function confirmReverse(string _name)
167         external
168         whenEntry(_name)
169         whenProposed(_name)
170         returns (bool success)
171     {
172         reverses[msg.sender] = _name;
173         emit ReverseConfirmed(_name, msg.sender);
174         return true;
175     }
176 
177     function confirmReverseAs(string _name, address _who)
178         external
179         whenEntry(_name)
180         onlyOwner
181         returns (bool success)
182     {
183         reverses[_who] = _name;
184         emit ReverseConfirmed(_name, _who);
185         return true;
186     }
187 
188     function removeReverse()
189         external
190         whenEntry(reverses[msg.sender])
191     {
192         emit ReverseRemoved(reverses[msg.sender], msg.sender);
193         delete entries[keccak256(bytes(reverses[msg.sender]))].reverse;
194         delete reverses[msg.sender];
195     }
196 
197     // Admin functions for the owner
198     function setFee(uint _amount)
199         external
200         onlyOwner
201         returns (bool)
202     {
203         fee = _amount;
204         emit FeeChanged(_amount);
205         return true;
206     }
```

```
207
208     function drain()
209         external
210         onlyOwner
211         returns (bool)
212     {
213         emit Drained(address(this).balance);
214         msg.sender.transfer(address(this).balance);
215         return true;
216     }
217
218     // MetadataRegistry views
219     function getData(bytes32 _name, string _key)
220         external
221         view
222         whenEntryRaw(_name)
223         returns (bytes32)
224     {
225         return entries[_name].data[_key];
226     }
227
228     function getAddress(bytes32 _name, string _key)
229         external
230         view
231         whenEntryRaw(_name)
232         returns (address)
233     {
234         return address(entries[_name].data[_key]);
235     }
236
237     function getUint(bytes32 _name, string _key)
238         external
239         view
240         whenEntryRaw(_name)
241         returns (uint)
242     {
243         return uint(entries[_name].data[_key]);
244     }
245
246     // OwnerRegistry views
247     function getOwner(bytes32 _name)
248         external
249         view
250         whenEntryRaw(_name)
251         returns (address)
252     {
253         return entries[_name].owner;
254     }
255
256     // ReversibleRegistry views
257     function hasReverse(bytes32 _name)
```

```
258         external
259         view
260         whenEntryRaw(_name)
261         returns (bool)
262     {
263         return entries[_name].reverse != 0;
264     }
265
266     function getReverse(bytes32 _name)
267         external
268         view
269         whenEntryRaw(_name)
270         returns (address)
271     {
272         return entries[_name].reverse;
273     }
274
275     function canReverse(address _data)
276         external
277         view
278         returns (bool)
279     {
280         return bytes(reverses[_data]).length != 0;
281     }
282
283     function reverse(address _data)
284         external
285         view
286         returns (string)
287     {
288         return reverses[_data];
289     }
290
291     function reserved(bytes32 _name)
292         external
293         view
294         whenEntryRaw(_name)
295         returns (bool)
296     {
297         return entries[_name].owner != 0;
298     }
299 }
```

6.7 Certifier

6.7.1 Certifier.sol

```
1 //! Certifier contract, used by service transaction.
```

```
2  //!
3  //! Copyright 2016 Gavin Wood, Parity Technologies Ltd.
4  //!
5  //! Licensed under the Apache License, Version 2.0 (the "License");
6  //! you may not use this file except in compliance with the License.
7  //! You may obtain a copy of the License at
8  //!
9  //!     http://www.apache.org/licenses/LICENSE-2.0
10 //!
11 //! Unless required by applicable law or agreed to in writing, software
12 //! distributed under the License is distributed on an "AS IS" BASIS,
13 //! WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
14 //! See the License for the specific language governing permissions and
15 //! limitations under the License.
16
17 pragma solidity ^0.4.24;
18
19
20 interface Certifier {
21     event Confirmed(address indexed who);
22     event Revoked(address indexed who);
23
24     function certified(address _who)
25         external
26         view
27         returns (bool);
28 }
```

6.7.2 Owned.sol

```
1  //! The owned contract.
2  //!
3  //! Copyright 2016 Gavin Wood, Parity Technologies Ltd.
4  //!
5  //! Licensed under the Apache License, Version 2.0 (the "License");
6  //! you may not use this file except in compliance with the License.
7  //! You may obtain a copy of the License at
8  //!
9  //!     http://www.apache.org/licenses/LICENSE-2.0
10 //!
11 //! Unless required by applicable law or agreed to in writing, software
12 //! distributed under the License is distributed on an "AS IS" BASIS,
13 //! WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
14 //! See the License for the specific language governing permissions and
15 //! limitations under the License.
16
17 pragma solidity ^0.4.24;
```

```
18
19
20 contract Owned {
21     event NewOwner(address indexed old, address indexed current);
22
23     address public owner = msg.sender;
24
25     modifier onlyOwner {
26         require(msg.sender == owner);
27         _;
28     }
29
30     function setOwner(address _new)
31         external
32         onlyOwner
33     {
34         emit NewOwner(owner, _new);
35         owner = _new;
36     }
37 }
```

6.7.3 SimpleCertifier.sol

```
1  //! The SimpleCertifier contract, used by service transaction.
2  //!
3  //! Copyright 2016 Gavin Wood, Parity Technologies Ltd.
4  //!
5  //! Licensed under the Apache License, Version 2.0 (the "License");
6  //! you may not use this file except in compliance with the License.
7  //! You may obtain a copy of the License at
8  //!
9  //!     http://www.apache.org/licenses/LICENSE-2.0
10 //!
11 //! Unless required by applicable law or agreed to in writing, software
12 //! distributed under the License is distributed on an "AS IS" BASIS,
13 //! WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
       implied.
14 //! See the License for the specific language governing permissions and
15 //! limitations under the License.
16
17 pragma solidity ^0.4.24;
18
19 import "./Certifier.sol";
20 import "./Owned.sol";
21
22
23 contract SimpleCertifier is Owned, Certifier {
24     struct Certification {
25         bool active;
```

```
26      }
27
28      mapping (address => Certification) certs;
29
30      // So that the server posting puzzles doesn't have access to the
            ETH.
31      address public delegate = msg.sender;
32
33      modifier onlyDelegate {
34          require(msg.sender == delegate);
35          _;
36      }
37
38      modifier onlyCertified(address _who) {
39          require(certs[_who].active);
40          _;
41      }
42
43      function certify(address _who)
44          external
45          onlyDelegate
46      {
47          certs[_who].active = true;
48          emit Confirmed(_who);
49      }
50
51      function revoke(address _who)
52          external
53          onlyDelegate
54          onlyCertified(_who)
55      {
56          certs[_who].active = false;
57          emit Revoked(_who);
58      }
59
60      function setDelegate(address _new)
61          external
62          onlyOwner
63      {
64          delegate = _new;
65      }
66
67      function certified(address _who)
68          external
69          view
70          returns (bool)
71      {
72          return certs[_who].active;
73      }
74  }
```

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 _____