h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Hochschule Darmstadt
- Fachbereich Informatik -

Permissioned Blockchains für B2B

Prototypische Implementierung eines dezentralisierten Wartungsmarktes

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt von

Eric Nagel
Matrikelnummer

Referent:	Prof. Dr. Andreas Müller
Korreferent:	Björn Bär

Abstract

Traditionelle B2B-Anwendungen mit multiplen Geschäftspartnern als Teilnehmer bringen verschiedene Probleme mit sich. Wenn jedes Unternehmen seine eigenen Daten speichert, erfolgt der Zugriff auf diese, für Kooperationspartner, aufwändig über Schnittstellen. Die Daten könnten sich auch bei einer einzelnen, eventuell nicht vertrauenswürdigen Instanz befinden, welche die Kontrolle über diese hat. Um dieses Problem zu lösen wird eine dezentrale B2B-Anwendung mittels der Blockchain-Technologie entwickelt. Die bekanntesten Implementationen dieser, wie Bitcoin und Ethereum, bringen jedoch Nachteile hinsichtlich Datenschutz, Sicherheit und Transaktionsdurchsatz mit sich, welche im B2B-Bereich nicht wünschenswert sind. Diese werden analysiert, um anschließend eine Aussage über den Nutzen von B2B-Blockchain-Anwendungen zu treffen, und sinnvoll den dezentralen Wartungsmarkt zu entwickeln.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Kapitel 1

Einführung und Motivation

Klassische B2B-Anwendungen bringen diverse Probleme hinsichtlich der Datenhaltung mit sich. Eigene Daten können bei jeden Geschäftspartner selber gespeichert werden, was jedoch den Zugriff auf diese, aufgrund von aufwendig einzurichtenden Schnittstellen und uneinheitlichen Datenformaten, erschwert. Eine andere Möglichkeit ist die Speicherung bei einem zentralen Unternehmen. Dieses hätte jedoch die Kontrolle über die Daten, womit alle anderen Parteien diesem vertrauen müssten. Diese Faktoren machen B2B-Anwendungen für die Teilnehmer unattraktiv und erschweren die Entwicklung [57] [76] [63].

Um diese Probleme zu lösen, wird ein Prototyp einer dezentralen B2B-Applikation basierend auf der Blockchain-Technologie entwickelt. Sie erlaubt es dezentrale Systeme aufzubauen, in welchen sich die Parteien nicht vertrauen. Alle Daten würden bei jedem Teilnehmer des Netzwerks gespeichert werden. Trotzdem sind diese nicht lösch- oder manipulierbar, alle Transaktionen sind lückenlos nachvollziehbar und es besteht ein gemeinsamer Konsens über den Datenbestand [47]. Bei der zu entwickelnden Applikation handelt es sich um einen automatisieren und dezentralisierten Wartungsmarkt. Teilnehmer an diesem sind Unternehmen und Wartungsdienstleister. Die Unternehmen besitzen IoT-Geräte, welche automatisch erkennen, dass sie eine Wartung benötigen. Sie legen für die Wartung einen Smart-Contract an, welcher von Wartungsdienstleistern angenommen werden kann. Diese melden sich an dem Gerät an und loggen die durchgeführten Wartungsschritte. Die Maschine schließt nach durchgeführter Wartung den Vertrag. Somit besteht ein automatisierter Wartungsmarkt zwischen mehreren Unternehmen, in welchen Wartungen verfolgbar und unveränderbar dokumentiert werden sowie kein Vertrauen zwischen den Parteien nötig ist.

Bekannte Blockchain-Implementationen, wie Bitcoin oder Ethereum, bringen jedoch Probleme mit sich, welche im B2B-Bereich von Nachteil sind. So sind alle Daten öffentlich einsehbar, der Transaktionsdurchsatz ist gering und die Konsensmechaniken sind unter bestimmten Umständen unsicher und resultieren in hohen Energieverbrauch [38][62][24].

Ziel dieser Arbeit ist es, die Probleme der Blockchain-Technologie für den B2B-Bereich zu analysieren und basierend auf den Ergebnissen die Entwicklung einer dezentralen B2B-Anwendung zu beschreiben sowie zu evaluieren. Dazu werden zunächst die grundlegenden Kon-

zepte der Blockchain-Technologie erklärt, um ein besseres Verständnis für die Vor- und Nachteile dieser zu erhalten. Anschließend werden die Probleme für B2B-Anwendungen anhand der Anforderungen an dem Wartungsmarkt genauer betrachtet und analysiert. Daraufhin erfolgt die Beschreibung der Anwendungsentwicklung. Zuletzt wird ein Fazit zur Lösung der Probleme und des entwickelten Systems gezogen.

Kapitel 2

Blockchain-Grundlagen

2.1 Funktionsweise

Die Funktionsweise der Blockchain wird hauptsächlich an Bitcoin erklärt. Als erste Blockchain-Anwendung [77] und aufgrund der relativ geringen Komplexität liefert es die Grundlage für die Funktion der Technologie. Andere Implementationen, wie Ethereum oder Ripple, funktionieren nach dem gleichen Prinzip.

2.1.1 Allgemein

Wenn der Begriff "Die Blockchain" auftaucht, ist damit meistens die Blockchain-Technologie gemeint. Es gibt nicht nur eine global bestehende Blockchain und auch nicht nur eine Implementation der Technologie, was man an Bitcoin oder Ethereum sehen kann.

Allgemein kann man die Blockchain als Datenstruktur bezeichnen, welche verteilt, nicht löschbar und unmanipulierbar gespeichert werden kann. Weiterhin verifizieren jegliche Teilnehmer am Netzwerk ausgeführte Transaktionen, womit ein gemeinsamer Konsens über den Datenbestand besteht [47].

In einer Blockchain werden Transaktionen in Blöcken gespeichert. Dabei handelt es sich um Operationen, welche Daten erstellen, verändern, oder löschen. Aus diesen lässt sich letztendlich der aktuelle Datenbestand ermitteln. So erfolgt z.B. bei Bitcoin keine Speicherung des aktuellen Guthabens der Teilnehmer. Es wird nur aus allen bestehenden Transaktionen berechnet [38]. Die Daten welche letztendlich bestehen, können z.B. Geldtransferinformationen (Bitcoin), Smart Contracts (Ethereum, selbstausführende Verträge mit selbst erstellter Programmlogik, siehe 2.2), oder simple Dokumente oder Informationen sein [24][62][73]. Die Blöcke setzen sich zusammen aus den Transaktionen sowie den Block Header, welcher verschiedene Metadaten, wie zum Beispiel den kombinierten Hash[1] aller Transaktionen, enthält [38].

Die Blöcke sind miteinander verkettet. Jeder Block Header enthält den Hash des vorherigen

[1] Hash: Ergebnis einer Operation, welche "eine Zeichenfolge beliebiger Länge auf eine Zeichenfolge mit fester Länge abbildet" [25].

Block Headers (Siehe Abb. 2.1). Dies ist ein wichtiges Feature zum Schutz der Blockchain vor Angriffen. Wenn ein Angreifer die Transaktionen eines Blocks zu seinen Gunsten verändern würde, würde sich der Hash des Block Headers ändern. Dieser müsste dann im darauffolgenden Block Header stehen, womit sich allerdings auch der Hash dieses Blocks ändert. Letztendlich müssten alle folgenden Blöcke manipuliert werden, um eine gültige Blockchain zu erhalten [62]. Diese Manipulation wird durch verschiedene Verfahren erschwert, welche genauer in den Kapiteln 2.1.1 und 2.1.2 erklärt werden.

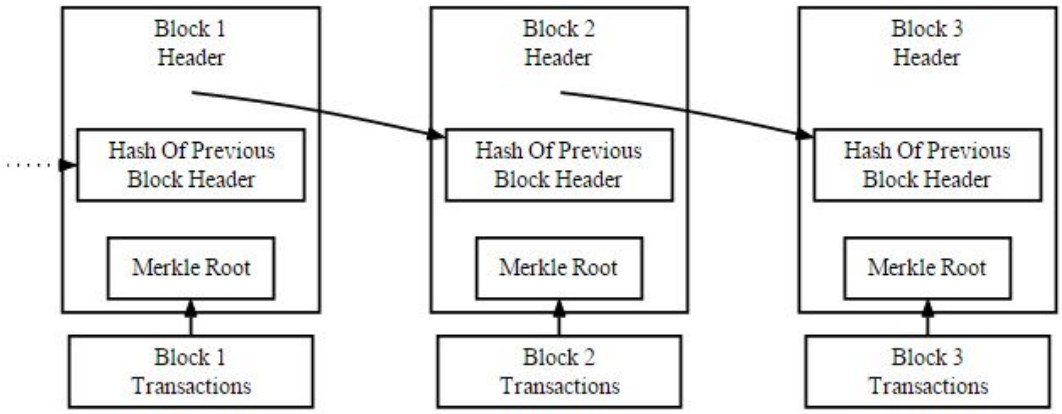

Abbildung 2.1: Verkettung von Blöcken durch Block Header Hashes

Die Blockchain ist verteilt gespeichert. Jeder Teilnehmer hat die Möglichkeit Sie auf seinen Rechner zu speichern. Somit besteht keine zentrale Instanz, welche die Kontrolle über die Daten hat. Weiterhin gibt es keinen Single Point of Failure[2], [47].

2.1.2 Konsensmechanismen

Aufgrund der verteilten Datenhaltung, muss es Verfahren geben, um die Daten synchron, und auf einen Stand, auf welchen sich alle Teilnehmer geeinigt haben, zu halten. Dazu gibt es die sogenannten Konsensmechanismen, welche gleichzeitig die Unmanipulierbarkeit der Daten sicherstellen. Bevor diese erklärt werden können, muss zunächst genauer auf die Funktion des Netzwerks eingegangen werden.

Wenn ein Teilnehmer eine Transaktion ausführt, wird diese, vorausgesetzt dass sie valide ist (Genauer im nächsten Absatz erklärt), an alle Nodes[3] (Teilnehmer, welche die Blockchain speichern) im Netzwerk weitergeleitet und im Transaktionspool aufgenommen. Dieser enthält alle noch nicht in Blöcken vorkommenden Transaktionen. Diese werden in einen neuen Block aufgenommen, und jede Node beginnt mit der Erstellung von diesem. Das Erstellen wird durch verschiedene Konsensmechaniken realisiert. Bei Bitcoin und Ethereum findet der Proof-of-Work

[2]Single Point of Failure: Komponente eines Systems, dessen Ausfall den Ausfall des gesamten Systems bewirkt [22].

[3]

Anwendung (Genauer im folgenden Absatz erklärt). Sobald eine Node einen Block erstellt, wird dieser im Netzwerk verteilt. Jede Node hängt ihn an ihre lokale Blockchain an, und beginnt mit der Erstellung des nächsten Blocks [**?**].

Damit eine Transaktion valide ist, muss sie bestimmte Voraussetzungen erfüllen. So muss sie unter anderen mit den Private Key des Senders signiert sein. Mittels seines Public Keys kann überprüft werden, ob wirklich er der Sender der Nachricht ist und ob die Transaktion manipuliert wurde. Dieses Verfahren wird auch in der Abbildung 2.2 visualisiert. Das Signieren trägt zur Sicherheit der Blockchain bei, da ein Angreifer somit keine Transaktionen manipulieren oder im Namen eines anderen ausführen kann. In Bitcoin ist eine weitere Kondition, dass der Transaktionsersteller die zu sendenen Bitcoins besitzt [38]. In Systemen wie Ethereum und Hyperledger Fabric, in welchen eigene Programmlogik abgebildet werden kann, können weitere Konditionen festgelegt werden. So muss z.B. ein Teilnehmer die nötigen Rechte haben um eine Transaktion auszuführen [1].

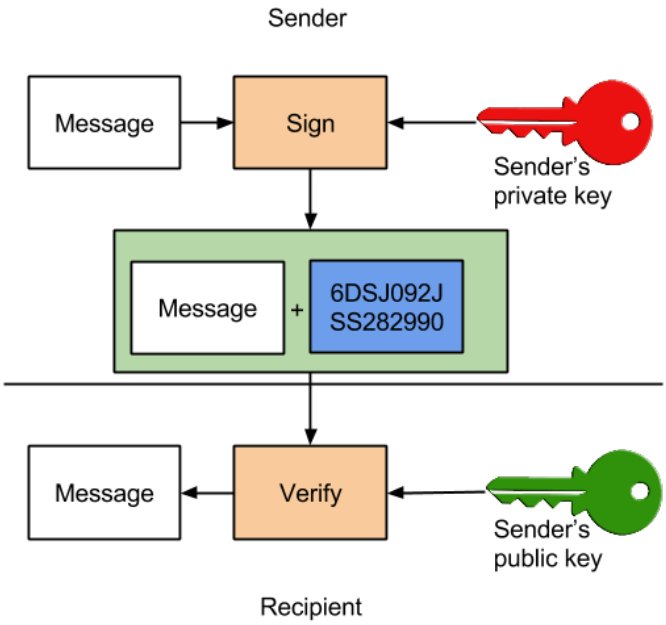

Abbildung 2.2: Signieren und Verifizieren von Nachrichten. Der Sender signiert die Nachricht mit seinen Private Key und der Empfänger kann diese mit den Public Key des Senders verifizieren.

Der Proof-of-Work ist nur einer der zur Verfügung stehenden Konsensmechanismen (Siehe Kapitel 5.1.3). Er bedarf jedoch genauerer Erklärung, da er in aktuellen Blockchain-Implementationen vorwiegend genutzt wird. Der Proof-of-Work ist eine Art Rätsel, welches mit der Rechenleistung von Nodes gelöst werden muss, um einen Block zu erschaffen. Genauer gesagt, muss für einen Block ein Hash gefunden werden, welcher einen bestimmten Wert unterschreitet. Desto kleiner dieser ist, desto höher ist die Schwierigkeit. Um wachsender Rechenleistung und Teil-

nehmerzahl entgegen zu wirken, also die Zeit für die Erstellung eines Blockes ungefähr gleich zu halten, kann die Schwierigkeit angepasst werden. Dies ist aufgrund verschiedener Faktoren nötig, welche genauer im Kapitel 5 erläutert werden. Um unterschiedliche Hashwerte für gleiche Blöcke zu erhalten, gibt es im Block Header eine Nonce[4], welche verändert wird [62]. Alle Nodes im Bitcoin-Netzwerk benötigen im Durchschnitt 10 Minuten um einen Proof-of-Work zu erbringen [38], bei einer Hash Rate[5] von ca. 13.000.000 TH/s (Terrahashes pro Sekunde) [13]. Bei Ethereum beträgt die Zeit ungefähr 15 Sekunden [10], bei einer Hash Rate von ca. 150 TH/s [11]. Damit die Nodes eine Motivation haben, Rechenleistung für das Erstellen von Blöcken zu nutzen, erhalten sie bei Erbringung des Proof-of-Work eine Belohnung in Form von Währung [62] [24].

Um vollständig zu verstehen, wie der Proof-of-Work funktioniert, muss das Forking erklärt werden. Wenn eine Node einen Proof-of-Work erbringt, also einen Block erstellt, wird dieser an alle anderen Nodes weitergeleitet. Im Bitcoin-Netzwerk dauert es bei einer maximalen Blockgröße von 1MB [38], zwischen 6 und 20 Sekunden, bis ein Block mindestens 90% aller Nodes erreicht hat [3]. Dies stimmt auch mit den Paper von Decker und Wattenhofer überein, wo eine durchschnittliche Zeit von 12,6 Sekunden angegeben wird, bis ein Block 95% aller Nodes erreicht [49]. In dieser Zeit kann es vorkommen, dass eine weitere Node einen Block erstellt. Auch dieser wird im Netzwerk verteilt, womit 2 Versionen der Blockchain existieren: Eine endet mit Block A, und die andere mit Block B. Dies ist der sogenannte Fork. Das Netzwerk muss sich nun darauf einigen, welche der beiden Versionen beibehalten werden soll. Deshalb gilt: Die Blockchain in welche mehr Arbeit eingeflossen ist, ist die gültige. Im Falle von Bitcoin wäre dies die längere Blockchain. Die Nodes probieren an den zuerst erhaltenen Block (A oder B) einen neuen anzuhängen. Gelingt dies, ist eine der beiden Blockchains länger als die andere. Diese wird dann von allen Nodes als die richtige akzeptiert. Dieser Vorgang wird auch in den Abbildungen 2.3 bis 2.6 dargestellt. Theoretisch ist es möglich, dass ein Fork über mehrere Blöcke besteht. Die Wahrscheinlichkeit dafür ist jedoch gering, da mehrmals nacheinander mindestens 2 Nodes zur ungefähr gleichen Zeit einen Block erstellen müssen. Auch zu erwähnen ist, dass in einem Fork-Branch weitere Forks entstehen können. Diese Forks sind der Grund, warum Transaktionen erst als bestätigt gelten, sobald sie in einem Block stehen, welcher eine gewisse Anzahl an Nachfolgern hat. Denn erst dann ist die Sicherheit gegeben, dass die Transaktion nicht in einem Fork vorhanden ist, welcher eventuell verworfen wird [38]. Wie genau der Proof-of-Work das Netzwerk absichert, wird im Kapitel 2.1.2 erklärt.

Neben dem Proof-of-Work gibt es noch weitere Konsensmechanismen, wie Proof-of-Stake, Proof-of-Authority oder Practical Byzantine Fault Tolerance [69], [48] . Diese werden im Kapitel 5.1.3 genauer beschrieben und analysiert.

[4]Nonce: Eine "Zahlen- oder Buchstabenkombination, [...] die nur ein einziges Mal in dem jeweiligen Kontext verwendet wird"[26].

[5]Hash Rate: Anzahl der in einer Zeiteinheit berechneten Hashwerte [12].

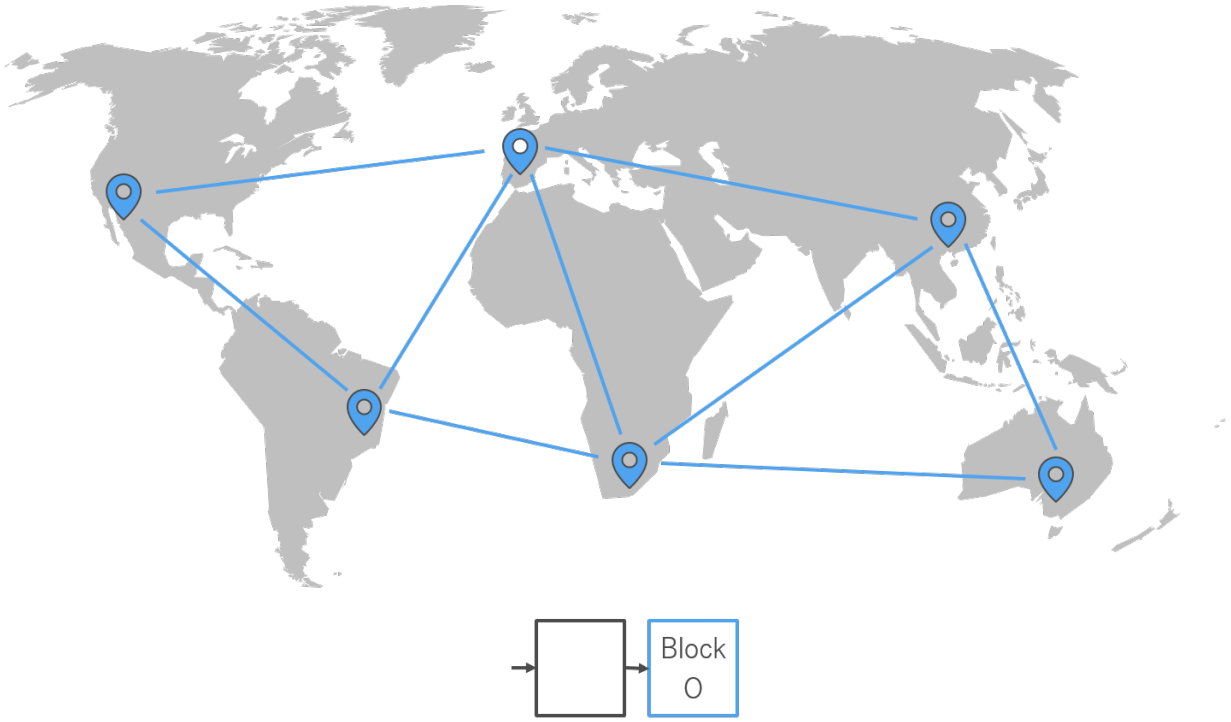

Abbildung 2.3: Fork-Visualisierung - Vor dem Fork besitzen alle Nodes Block O als letzten Block [38].

2.1.3 Nichtangreifbarkeit/Immutability

Viele Faktoren tragen zur Nichtangreifbarkeit und Unveränderlichkeit der Blockchain bei. Da alle Nodes die ausgeführten Transaktionen auf Validität prüfen, können diese nicht ohne Berechtigung, im Namen einer anderen Identität, oder mit unzureichenden Konditionen ausgeführt werden. Der wichtigste Faktor ist jedoch der genutzte Konsensmechanismus in Verbindung mit den verketteten Blöcken. Durch ihm wird sichergestellt, dass bestehende Daten nicht gelöscht oder manipuliert werden können.

Ein Beispiel dafür kann am Proof-of-Work gezeigt werden. Ein Angreifer probiert eine Transaktion aus einen bestehenden Block zu entfernen. Dazu würde er die Transaktion bei seiner lokalen Blockchain entfernen. Nun ist jedoch der Hash des Blockes sowie der Block selber nicht valide und würde von keiner Node akzeptiert werden. Der Angreifer muss also erneut einen Proof-of-Work für den manipulierten Block erbringen. Dies wäre für eine Einzelperson jedoch Zeitaufwändig, wenn man bedenkt, das extra für diesen Zweck produzierte Hardware eine Hash Rate von bis zu 13,5 TH/s erreicht [61]. Dies Wenn der manipulierte Block nun noch Nachfolger hat, muss aufgrund des neuen Hashes auch für diese der Proof-of-Work erbracht werden. Hinzu kommt, dass die Blockchain des Angreifers erst von allen Nodes akzeptiert wird, wenn sie länger ist. Er müsste also schneller als das gesamte Bitcoin-Netzwerk Blöcke erschaffen können. Dies ist nur möglich, wenn er 51% der Rechenleistung des Netzwerks besitzt. Deshalb wird dieser Angriff auch 51%-Angriff genannt [70] [24].

An dieser Stelle sollte erähnt werden, dass auch wenn ein 51%-Angriff erfolgt, die Angriffs-

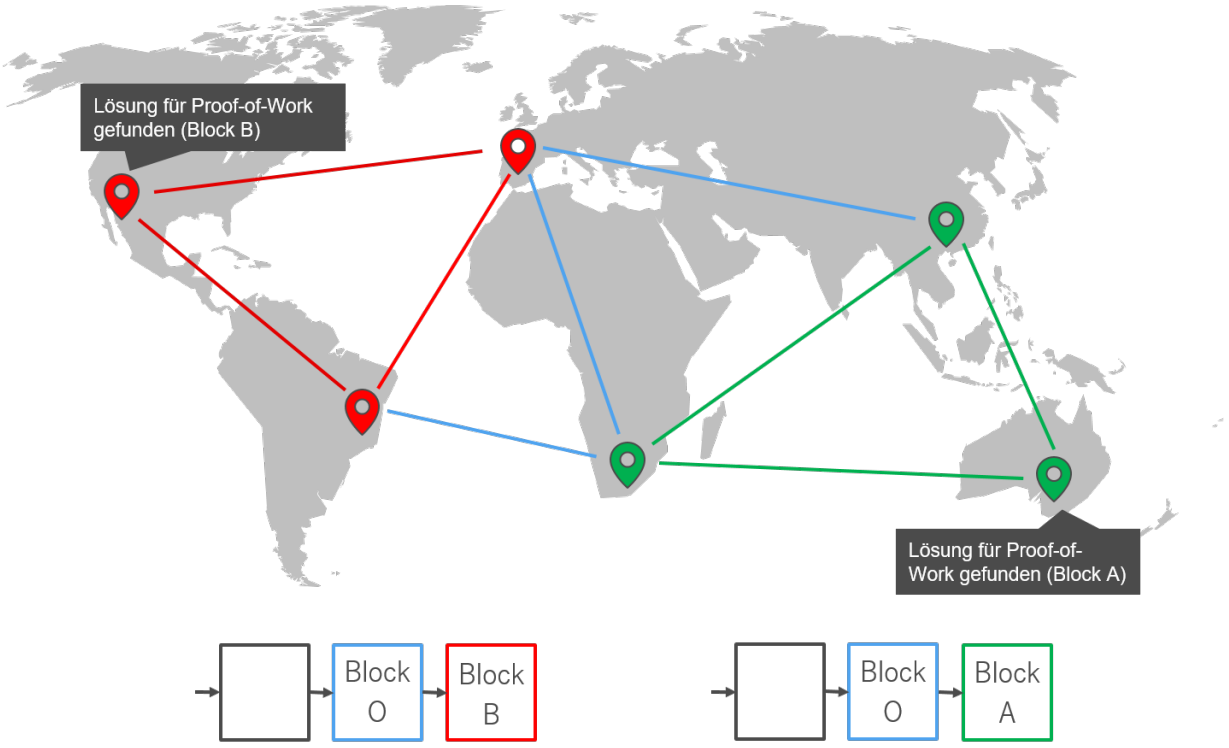

Abbildung 2.4: Fork-Visualisierung - 2 Nodes finden zur ungefähr gleichen Zeit einen Block und verbreiten ihn im Netzwerk, womit 2 Versionen der Blockchain bestehen [38].

möglichkeiten beschränkt sind. Der Angreifer kann keine unvaliden Transaktionen sowie Blöcke erstellen. Ihm ist es möglich DoS-Angriffe auszuführen indem er verhindert das bestimmte Transaktionen in Blöcke aufgenommen werden. Genau so kanner die Historie der Daten verändern, indem er eine Transaktion aus einem Block entfernt und diese nicht erneut in einen Block aufnimmt, oder dafür sorgt, dass sie ungültig werden. So gibt es z.B. im Falle von Kryptowährungen folgende Form eines Double-Spending-Angriffs: Ein Angreifer sendet z.B. Bitcoins an einen Händler. Dieser wartet auf die Bestätigung der Transaktion in einen Block sowie auf nachfolgende Blöcke. So stellt er sicher, dass die Transaktion nicht in einem eventuell verworfenen Fork stand. Erst dann versendet er die Ware. Anschließend ersetzt der Angreifer die Transaktion durch eine Zahlung an sich selber und erstellt die längere Blockchain, womit der Händler letztendlich kein Geld erhalten hat [24]. Auch zu bedenken ist, dass ein Nutzer mit 51% der Rechenleistung wenig Motivation hat Angriffe auszuführen, da er für jeden erstellten Block Kryptowährung als Belohnung erhält. Der Wert der Kryptowährung würde sinken, wenn Angriffe auf die Blockchain entdeckt werden. Deshalb besteht für die sogenannten Miner eine Motivation, ehrlich zu arbeiten [38].

2.2 Blockchaintypen

Es gibt 3 Typen von Blockchains, welche die zugelassenen Teilnehmer bestimmen. Bisher wurden nur Public Blockchain-Anwendungen, wie Bitcoin und Ethereum erwähnt. In diesen gibt

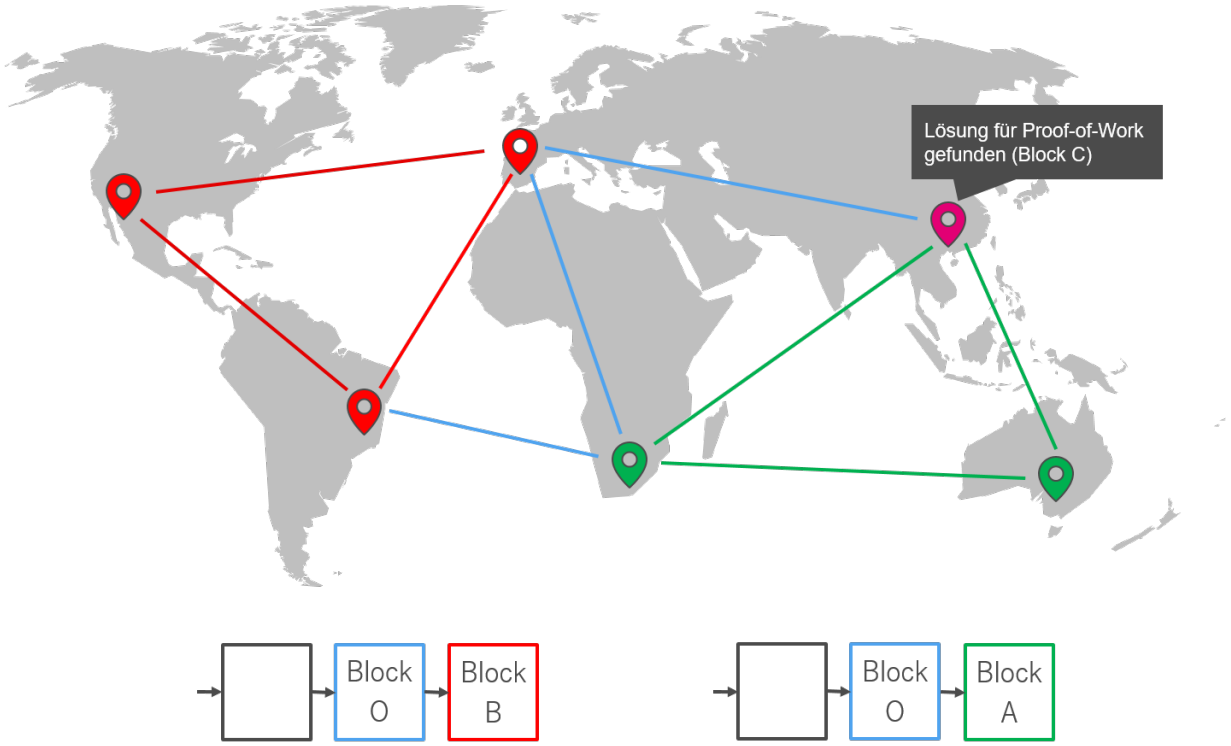

Abbildung 2.5: Fork-Visualisierung - Eine Node, welche Block A zuerst erhalten hat, hängt daran einen neuen Block C an [38].

es keine Teilnehmerbeschränkungen, jeder kann am Netzwerk teilnehmen und die Blockchain öffentlich einsehen. Anders ist dies bei Permissioned (oder auch Consortium [39]) und Private Blockchains. Die beiden Begriffe werden in einigen wissenschaftlichen Arbeiten gleichgesetzt (Siehe [53], [?], [60]). Hier folgt jedoch eine Unterscheidung. Dabei ist eine Private Blockchain, eine Blockchain welche nur von einen Nutzer verwendet wird. Da eine solche Anwendung keinen Sinn macht, da keine Vorteile der Blockchain genutzt werden können, wird darauf nicht genauer eingegangen. Interessanter sind Permissioned Blockchains, an welchen nur zugelassene Nutzer teilnehmen dürfen. Nur diese sind berechtigt, Transaktionen auszuführen und die Daten einzusehen [60]. Dies bietet sich vor allem bei B2B-Anwendungen an, welche von verschiedenen Unternehmen genutzt werden sollen. In diesen kann es aufgrund von z.B. sensiblen Daten nötig sein, das nur bestimmte Parteien Zugriff auf die Blockchain haben.

2.3 Exemplarische Anwendungsfälle

Die Blockchain wird als revolutionäre Technologie angepriesen (Siehe [71]). Trotzdem ist es wichtig zu wissen, für welche Zwecke sie wirklich geeignet ist. Grundsätzlich macht eine Blockchain Sinn, wenn mehrere Parteien, welche sich nicht vertrauen, mit einen System interagieren wollen, welches von keiner dritten zentralen Instanz verwaltet wird [76]. Um eine bessere Vorstellung zu solchen Anwendungen zu erhalten, werden im Folgenden verschiedene Exemplarische Anwendungsfälle genannt und beschrieben.

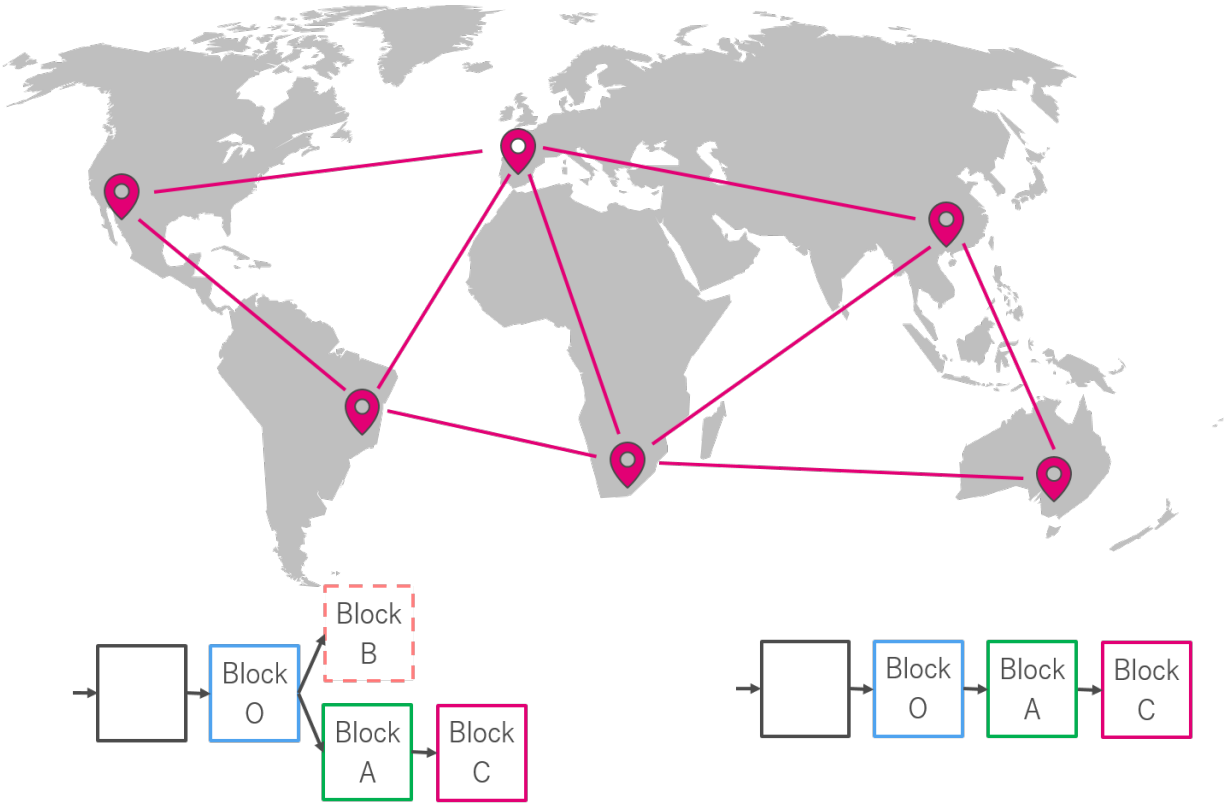

Abbildung 2.6: Fork-Visualisierung - Block C verbreitet sich im Netzwerk, rote Nodes sehen zwei Blockchains und akzeptieren die längere [38].

Der erste Anwendungsfall, mit welchen die Blockchain-Technologie auch entstanden ist, sind Kryptowährungen. Mit Ihnen ist es möglich Geld zwischen beliebigen Parteien zu übertragen, ohne dass die Transaktionen von einer eventuell nicht vertrauenswürdigen Bank oder ähnlichem kontrolliert und verwaltet werden [70].

Weitere Anwendungsfälle ergeben sich mit der Möglichkeit Programmlogik auf der Blockchain abzubilden. So können beispielsweise dezentrale Online-Wahlen realisiert werden. Die Stimmen würden in der Blockchain gesammelt werden, und können so letztendlich nicht mehr von z.B. einer korrupten Regierung manipuliert werden [44].

Ein weiterer Anwendungsfall, insbesondere für den B2B-Bereich, wäre Supply Chain Management. Über eine digitale Lieferkette sollen Material- und Informationsflüsse zu Produkten und Dienstleistungen aufgebaut und verwaltet werden [58]. Dies erlaubt Unternehmen das automatisieren von Prozessen und das verbesserte reagieren auf Ereignisse (z.B. Lieferverspätungen). Weiterhin ist es dem Unternehmen möglich, dem Kunden exakt aufzuzeigen wo es und seine Unterprodukte produziert wurden. In klassischen B2B-Anwendungen müsste jedes Unternehmen, welches ein Teil der Supply Chain ist, die relevanten Daten durch z.B. APIs[6] bereitzustellen. Dies bedeutet Aufwand, da diese erstmal entwickelt werden müssen. Weiterhin müsste ein System von all diesen unterschiedlichen Schnittstellen die Daten abfragen und in einem System zusammenführen um die Supply Chain zu erstellen. Diese müssten dann abgefragt werden um

[6] API: Schnittstelle zur Anwendungsprogrammierung[27]

die Supply Chain zu erstellen. Aufgrund dieses Aufwandes werden oft Dritte eingestellt, welche sich um den Aufbau und um die Datenintegration der Supply Chain kümmern. Den Aufwand, sowie die eventuell nicht vertrauenswürdige dritte Partei könnte man durch die Nutzung der Blockchain überspringen. In dieser könnte jedes Unternehmen die relevanten Daten speichern, ohne das aufwändige Erstellen von Schnittstellen. Die Supply Chain wäre direkt in der Blockchain vorhanden, und kein Unternehmen muss Datenmanipulation oder ähnliches befürchten [57].

Auch dezentrale Märkte sind für B2B-Anwendungen interessant. Der zentrale Marktplattformbetreiber, wie z.B. Ebay oder Amazon, welcher persönliche Informationen speichert und Gebühren für den Verkauf von Artikeln verlangt, wäre hinfällig. Nutzer könnten Waren untereinander verkaufen, während die Blockchain als Notar für den Warenaustausch dient [39].

Ein weiteres Beispiel wären Blockchain Sharing-Systeme. So könnte ein dezentrales Fahrradleihsystem aufgebaut werden. Nutzer würden mit ihrem Smartphone, über ihre in der Blockchain hinterlegte Identität (Im Falle von z.B. Ethereum die Wallet-Adresse[7]), das Fahrrad entsperren. Dieser erkennt automatisch die gefahrene Distanz sowie die Nutzungsdauer. Über einen Smart Contract würde anschließend die automatische Zahlung erfolgen. Neben der Automatisierung besteht der Vorteil, dass mehrere Unternehmen oder auch Privatpersonen Leihfahrräder anbieten können, ohne dass sie einer zentralen Instanz mit der Verwaltung vertrauen müssen [4], [51].

[7]Wallet: Speichert z.B. bei Ethereum und Bitcoin den Private Key des Nutzers und wird z.B. als Adresse für Zahlungen genutzt [6].

Kapitel 3

Dezentraler Wartungsmarkt - Konzept

3.1 Allgemein

Ziel dieser Arbeit ist die Entwicklung einer prototypischen B2B-Applikation in Form eines automatisierten sowie dezentralisierten Wartungsmarktes. Teilnehmer an diesem sind multiple Unternehmen und Wartungsanbieter. Erstere besitzen IoT-Geräte, welche erkennen können, dass sie eine Wartung benötigen. Die Wartungsanbieter erhalten die Informationen zur Wartung, und können sich für diese anmelden. Anschließend würden sie diese durchführen und dabei die Wartungsschritte loggen.

In klassischen B2B-Anwendungen wäre die Realisierung dieses Systems auf 2 Arten erfolgt. Bei ersterer gäbe es eine dritte Partei, welche den Markt verwaltet, und bei welcher sich alle Unternehmen und Wartungsanbieter anmelden müssen (z.B. Ebay). Die andere Möglichkeit wäre, dass eines der teilnehmenden Unternehmen den Markt verwaltet. Bei beiden Optionen müssten die Teilnehmer am Markt ihre Daten einer eventuell nicht vertrauenswürdigen zentralen Instanz zur Verfügung stellen. Weiterhin würde bei jeden Unternehmen die Notwendigkeit bestehen, API's für den Datenzugriff zu erstellen.

Um dies zu verhindern, wird der Wartungsmarkt auf Basis der Blockchain-Technologie implementiert. Somit können beliebig viele Unternehmen und Wartungsanbieter an dem System teilnehmen, ohne dass eine Datenmanipulation durch die Teilnehmer oder eine zentrale Instanz befürchtet werden muss.

3.2 Anforderungen

Es ergeben sich verschiedene Anforderungen an das zu entwickelnde System. Die Spezifizierung dieser ist wichtig, denn auf Basis dieser wird eine Blockchain-Implementation ausgewählt und auf verschiedene Probleme analysiert. Folgende Anforderungen ergeben sich an die Anwendungen:

Registrieren und Identifizieren von Unternehmen, Wartungsanbietern und Geräten in der Blockchain Die zu entstehende B2B-Anwendung soll zwischen verschiedenen Unternehmen bestehen. Diese müssen Berechtigungen erhalten um am Netzwerk teilzunehmen, und ausgeführte Transaktionen sollen ihnen zugeordnet werden können.

Wartungsgeräte kündigen Wartungen in Form eines Smart Contracts in der Blockchain an Es kann verschiedene Gründe für die Wartung geben. So kann z.B. ein Wartungsdatum erreicht werden, oder Sensorwerte weisen auf einen Fehler hin.

Wartungsanbieter können den Smart Contract unter bestimmten Konditionen annehmen Eine dieser Konditionen könnte sein, dass der Wartungsanbieter bereits Erfahrungen mit der Wartung von bestimmten Geräten hat. Diese Information könnte ebenfalls aus der Blockchain abgefragt werden. Weiterhin darf der Vertrag z.B. noch nicht von einen anderen Anbieter akzeptiert worden sein.

Wartungsanbieter loggen Wartungsschritte in der Blockchain Die Wartungsanbieter melden sich am Gerät mit an, und loggen die durchgeführten Wartungsschritte. So entsteht eine nachverfolgbare und nich löschbare Historie an durchgeführten Wartungen mit dazugehörigen Schritten.

Gerät überprüft ob Wartung erfolgt ist, und schließt den Contract Die Überprüfung kann anhand der geloggten Schritte sowie Sensorwerten erfolgen, welche vor und nach der Wartung existiert haben.

Nur bestimmte Teilnehmer können bestimmte Transaktionen ausführen Die verschiedenen Teilnehmer haben unterschiedliche Rechte. So soll es z.B. einen Unternehmen nicht möglich sein, Wartungsverträge zu bearbeiten oder zu akzeptieren.

Private Transaktionen sollen zwischen Teilnehmern möglich sein Bei Blockchains wie Bitcoin und Ethereum sind alle Daten in der Blockchain für alle Teilnehmer einsichtbar. Aufgrund von sensiblen Daten kann es allerdings vorkommen, das nicht alle Transaktionen für alle Teilnehmer sichtbar sein sollen. Im Falle des Wartungsmarktes sollen z.B. Preisabsprachen zwischen Unternehmen und Wartungsdienstleistern privat erfolgen.

Hoher Transaktionsdurchsatz und geringe Transaktionszeiten In Bitcoin ist lediglich ein Transaktionsdurchsatz von 7 Transaktionen pro Sekunde möglich [77]. Hinzu kommt, dass es ca. zwischen 30 Minuten und 16 Stunden dauern kann, bis eine Transaktion bestätigt ist [40]. Darauf wird auch genauer im Kapitel 5.1 eingegangen. In dem zu entwickelnden System ist die Skalierbarkeit wichtig. Je nach der Anzahl der am Netzwerk teilnehmenden Unternehmen

und Wartungsanbieter wird eine höherer Transaktionsdurchsatz benötigt. Insbesondere wenn tausende von Geräten Transaktionen in der Blockchain ausführen.

Nichtangreifbarkeit der Daten Die Nichtangreifbarkeit wird durch die genutzte Konsensmechanik realisiert. Der am häufigsten genutzte Proof-of-Work ist in einem Netzwerk mit wenig Teilnehmern allerdings unsicher, da es einfach ist 51% der Rechenleistung zu erreichen. Weiterhin führt er zu einen hohen Stromverbrauch (Im Bitcoin-Netzwerk der Verbrauch von ca. 3.500.000 US-Haushalten [2]), welcher nicht erwünscht ist.

An dieser Stelle muss darauf hingewiesen werden, dass das System um viele nützliche Features erweiterbar ist. So könnte zum Beispiel eine Bewertung der Wartungsanbieter anhand bestimmter Faktoren erfolgen. Da es sich jedoch nur um eine prototypische Implementation handelt, werden nur die Features implementiert, welche für einen Proof-of-Concept eines solchen Systems benötigt werden.

Kapitel 4

Aktueller Stand der Technik

Die Blockchain wird seit 2008 erfolgreich für Kryptowährungen eingesetzt. Mit Ethereum wird das Konzept von Smart Contracts implementiert, womit es möglich ist eigene Programmlogik in der Blockchain abzubilden und so dezentrale Anwendungen zu entwickeln. Weitere weniger bekannte Blockchain-Implementationen für Public Blockchains sind Monero, Dashcoin und Litecoin [55].

Die Technologie bringt durch ihre Architektur jedoch auch Limitationen mit sich, weshalb sie nicht für alle Anwendungszwecke geeignet ist (Siehe 5. Die Probleme werden in vielen wissenschaftlichen Arbeiten analysiert und Lösungen für diese vorgeschlagen. Trotzdem bestehen gewisse Limitationen weiterhin [77][70][65].

Permissioned Blockchains, wie Hyperledger Fabric oder Quorum bieten Vorteile gegenüber dem Public Blockchains. Sie bringen allerdings auch neue Herausforderungen mit sich. So muss z.B. eine Alternative zum Proof-of-Work gefunden werden, um je nach Use-Case, Performance, Skalierbarkeit und Nichtangreiffbarkeit sicher zu stellen [60].

Dezentrale Märkte sind eine der am meisten mit Blockchain in Verbindung erwähnten Use-Cases, wie man an Quellen wie [39] und [64] sehen kann. Neben Konzepten für diese (Siehe [56]), gibt es auch Live-Systeme, wie Syscoin [66].

Dezentrale Wartungsmärkte hingegen werden nur in Verbindung mit der Supply Chain erwähnt, wie zum Beispiel in [67] oder [52]. Implementationen oder Konzeptentwürfe konnten nicht gefunden werden.

Kapitel 5

Evaluierung Permissioned Blockchains für B2B

Die Blockchain-Technologie bringt diverse Probleme mit sich, welche je nach Anwendungszweck und Blockchaintyp verschieden große Auswirkungen haben. Für den B2B-Bereich gilt es vor allem die Skalierbarkeit sowie die Konsensmechanismen zu analysieren.

5.1 Skalierbarkeit

Das CAP-Theorem besagt, dass es in einem verteilten System nur möglich ist, 2 von den 3 folgenden Eigenschaften zu erfüllen: Konsistenz, Verfügbarkeit und Ausfalltoleranz. Bei der Blockchain wären dies: Dezentralisierung, Skalierbarkeit und Nichtangreifbarkeit [65]. Im Bezug auf die Skalierbarkeit wird vor allem auf den Transaktionsdurchsatz sowie die Bestätigungszeiten von Transaktionen eingegangen. Dazu erfolgt zunächst eine Analyse an aktuellen Public Blockchains, und letztendlich an Permissioned Blockchains. Die Ergebnisse werden ebenfalls auf das CAP-Theorem angewandt.

5.1.1 Public Blockchains

5.1.1.1 Bitcoin

Das Bitcoin-Netzwerk erreicht aktuell einen maximalen Transaktionsdurchsatz von 7 Transaktionen (Unterschiedlich je nach Größe der Transaktionen) pro Sekunde (TPS), bei einer Blockgröße von 1MB. Hingegen erreicht Paypal 115 TPS, und Visa 2000 TPS (Siehe auch Abb. 5.1) [19]. Hinzu kommt, dass ungefähr 170000 unbestätigte Transaktionen[1] bestehen [50]. Berechnungen von Scherer zeigen, dass bei 11,8 Millionen Nutzern im Bitcoin-Netzwerk, sowie einem Transaktionsdurchsatz von 4 TPS, jeder Nutzer nur nur ca. 10 Transaktionen im Jahr senden kann [65].

[1] Unbestätigte Transaktion: Eine Transaktion, welche noch nicht in einen Block vorkommt [38].

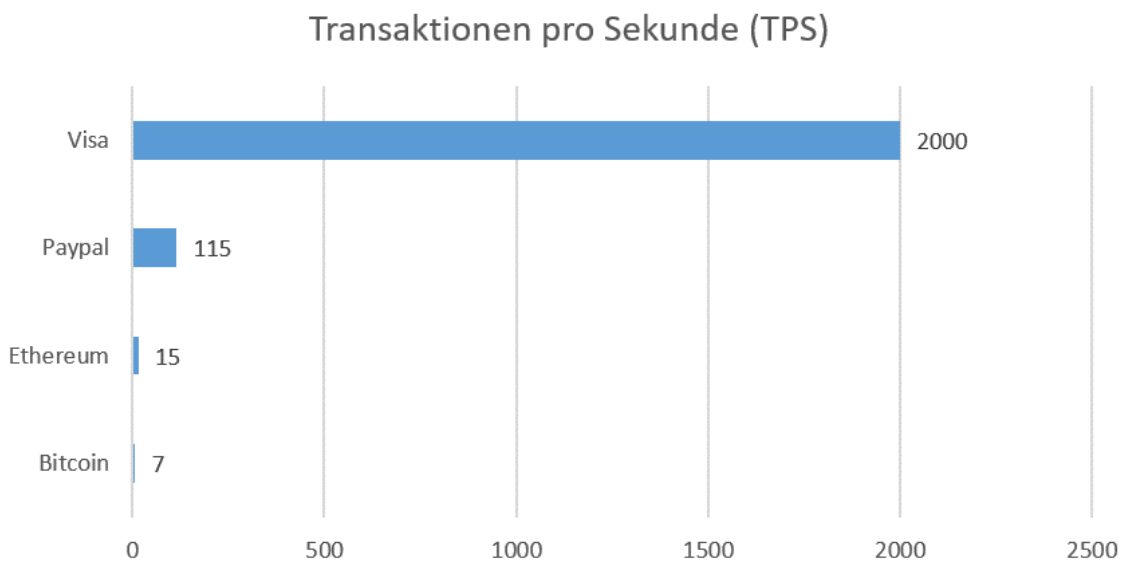

Abbildung 5.1: Möglicher Transaktionsdurchsatz bei Bitcoin, Ethereum, Paypal und Visa [19].

Der Transaktionsdurchsatz ist durch verschiedene Faktoren limitiert. Hauptsächlich durch die limitierte Blockgröße von 1MB, und dem Proof-of-Work: Nur eine bestimmte Anzahl an Transaktionen passt in einen Block, und nur alle 10 Minuten wird einer erstellt. Es gäbe also die Möglichkeit, die Blockgröße zu vergrößern, oder die Zeit für den Proof-of-Work zu verringern, indem die Schwierigkeit angepasst wird. Es gibt jedoch diverse Nachteile, welche dadurch entstehen würden. Bei einer größeren Blockgröße würde es länger dauern, bis ein Block beim Propagieren durch das Netzwerk alle Nodes erreicht. Dies würde zu öfter vorkommenden und längeren Forks führen und somit die Sicherheit des Netzwerks beeinträchtigen. Den gleichen Effekt hätte eine kürzere Proof-of-Work Zeit, da die Wahrscheinlichkeit höher ist, dass zwei Nodes zur ungefähr gleichen Zeit einen Block erstellen [65] [24] [68].

Entsteht ein Fork, probieren Nodes die längere und somit gültige Blockchain zu erschaffen. Gelingt dies, wird die kürzere Blockchain mit den nun sogenannten Stale Blocks verworfen. Die gesamte Rechenleistung, welche in die Stale Blocks und seine Nachfolger geflossen ist, trägt nicht zur Sicherheit des Netzwerks bei. Dies lässt sich auch anhand der Abbildung 5.2 erläutern. Innerhalb der Blockchain bestehen durch mehrere Forks 5 Branches. Das bedeutet, dass die Rechenleistung des Netzwerks auf diese aufgespalten ist. Es wird davon ausgegangen, dass 20% der Rechenleistung in den obersten Branch geflossen ist, welcher der längste ist. Die restlichen 4 Branches erhalten je 10% der Rechenleistung. Wenn es nun einen Angreifer mit 40% der Rechenleistung probiert eine eigene Blockchain zu erstellen, gelingt ihm dies, da er schneller die längere Blockchain erstellen kann [68]. Zusammenfassend lässt sich sagen, dass ein Angreifer nicht 51% der Rechenleistung für einen Fork benötigt, wenn das Netzwerk diese bei Forks verschwendet [42].

Ein weiteres Problem der Forks ist, dass Miner keine Belohnung für die Arbeit an verworfenen Blöcken erhalten. Dadurch kann es zur Zentralisierung durch wachsende Mining Pools

kommen. Dies wird an folgenden Beispiel ersichtlich: Ein Mining Pool A besitzt 30% der Rechenleistung, ein Mining Pool B 10%. In dem genannten Beispiel würde Mining Pool A in 70% aller Fälle einen Stale Block erzeugen, und B in 90% aller Fälle. Kein Miner würde dem Mining Pool B beitreten, da die Wahrscheinlichkeit geringer ist, dass B gültige Blöcke erschafft. A hingegen würde immer mehr Miner, und somit mehr Rechenleistung erhalten [24].

An dieser Stelle sollte auch darauf hingewiesen werden, dass schnellere Blockerstellungszeiten nicht zwingend zu schnelleren Transaktionsbetätigungen führen. Transaktionen werden zwar schneller in Blöcken aufgenommen, aber es muss auf mehr Nachfolger gewartet werden, um sicher zu gehen, dass die Transaktion nicht in einem Fork vorkommt [65].

5.1.2 Ethereum

Bessere Skalierbarkeit durch GHOST Das Ethereum Netzwerk nutzt das sogenannte GHOST-Protokoll, und erreicht damit eine Transaktionsdurchsatz von 15 TPS, bei einer durchschnittlichen Zeit von 15 Sekunden um den Proof-of-Work zu erbringen. Dieses löst Probleme des Forkings und der Benachteiligung von Minern. Ersteres wird dadurch gelöst, dass Stale Blocks in die Berechnung der gültigen Blockchain einfließen. Anders als bei Bitcoin, wo lediglich die Parents und deren Nachfolger eine Rolle spielen. Die Stale Blocks werden in Ethereum "Uncles" genannt. Kurz gesagt, ist ein Uncle ein alternativer gefundener Block welcher auf der gleichen Höhe wie der Parent bestehen würde [24].

Die Bestimmung der gültigen Blockchain wird an der Abbildung 5.2 ersichtlich. In Ethereum ist die Blockchain die gültige, für welche die meiste Arbeit aufgebracht wurde, unter Einbezug der Uncles. Das führt dazu, dass der Branch mit den meisten Uncles bestehen bleibt. Das bedeutet letztendlich, dass die gesamte Rechenleistung das Netzwerk absichert, auch wenn diese sich auf die Branches aufteilt. Ein Angreifer braucht somit weiterhin 51% der Rechenleistung um einen Angriff auszuführen [68].

Damit ist allerdings noch nicht das Problem der Zentralisierung durch Mining Pools gelöst. Es besteht weiterhin keine Motivation für Miner, Uncles zu minen. Deswegen ist das GHOST-Protokoll in Ethereum so erweitert, dass Miner Ether[2] als Belohnung für das Erstellen von Uncles erhalten (Allerdings weniger als bei vollwertigen Blöcken). Somit besteht ebenfalls die Motivation, kleineren Mining Pools beizutreten [24]. An dieser Stelle sollte auch erwähnt werden, dass Miner entscheiden können, an welchen Branch sie arbeiten [77].

Während Ethereum die Probleme löst, welche durch Forks entstehen, ist der Transaktionsdurchsatz trotzdem limitiert. Die Blockgröße muss klein genug bleiben, damit das Propagieren im Netzwerk effizient bleibt [65]. Ansonsten würden Miner unter Umständen so benachteiligt werden, dass sie nur sehr selten bis garnicht den aktuellen Block der gültigen Blockchain erhalten würden. Dies wiederum würde dazu führen, dass sie nur Uncles minen können, und so nie die volle Belohnung erhalten können. Hinzu kommt, das Uncles nur gültig sind, wenn sie maximal eine bestimmte Anzahl an Generationen vom aktuellen Block in der gültigen Chain

[2]Kryptowährung von Ethereum [24].

entfernt sind. Ansonsten hätten die Miner auch weniger Motivation ehrlich zu bleiben, da sie ohne Nachteile an der Chain eines Angreifers arbeiten könnten [24].

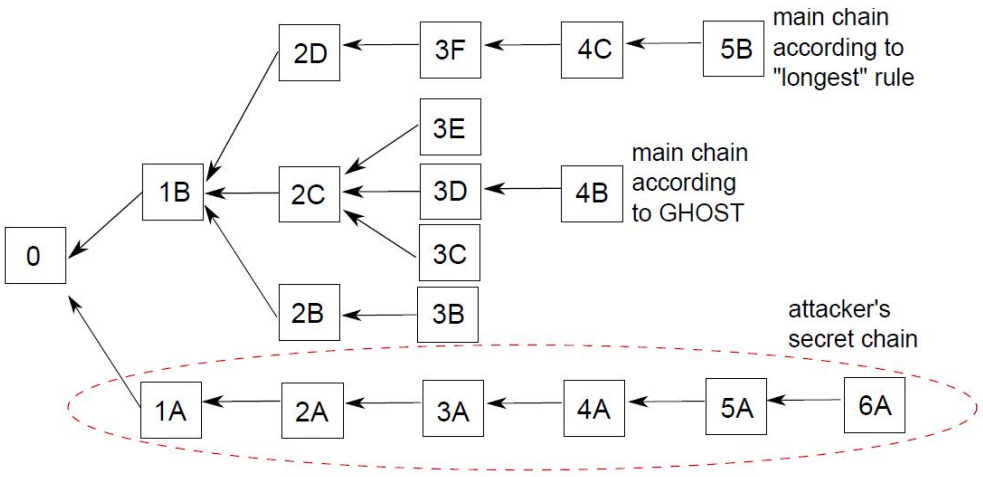

Abbildung 5.2: Auswahl der gültigen Blockchain. In Bitcoin die längere Blockchain. In Ethereum die Blockchain

Schlechtere Skalierbarkeit durch Smart Contracts Ethereum löst die Probleme von häufig auftretenden Forks und erlaubt so einen höheren Transaktionsdurchsatz sowie schnellere Transaktionsbestätigungszeiten. Weitere Probleme entstehen jedoch, wenn eine Blockchain nicht nur Geldtransfertransaktionen verarbeitet. Ethereum erlaubt das speichern und ausführen von eigenem Code durch Smart Contracts. Dadurch steigt die Komplexität der auszuführenden Transaktionen. Dadurch nimmt die Skalierbarkeit ab, da die entstehenden größeren Blöcke eine längere Propagationszeit verursachen. Ebenfalls verschlechtert sich die Performance des Netzwerks, da die Daten schwieriger zu verarbeiten sind. So muss jede Node alle Transaktionen verifizieren, Smart Contract-Code ausführen, und die Ergebnisse speichern. [65].

In Ethereum werden Transaktionen sequentiell bei allen Nodes ausgeführt. Dazu gehört das ausführen von Smart Contract-Code sowie das verifizieren der Ergebnisse. Nur so können in Konflikt stehende Transaktionen, wie zum Beispiel beim Double-Spend) erkannt werden. Eine Parallelausführung ist nicht möglich. Dies verschlechtert letztendlich die Performance des Netzwerks, da es länger dauert Transaktionen auszuführen [65]. Dies wird auch durch ein Beispiel klar. Ein Angreifer kann DoS-Attacken ausführen, indem er komplex auszuführende Smart Contracts schreiben. Die Ausführung von diesem bei jeder Node führt dazu, dass keine anderen Operationen ausgeführt werden können. Ethereum löst dieses Problem, indem der Transaktionssender für jeden Berechnungschritt eine Gebühr zahlen muss. Dies funktioniert jedoch nur, wenn in der Blockchain-Anwendung eine Kryptowährung genutzt wird [75].

Ebenfalls behauptet Vukolic, dass der Code der Smart Contracts nicht bei allen Nodes ausgeführt werden muss. Um Konsens zu erreichen genügt es, dass alle Nodes den gleichen Stand der Daten erhalten. Deshalb könnte die Codeausführung von nur von bestimmten Nodes

ausgeführt werden. Das Problem dabei ist, dass man eine genügend große Anzahl an vertrauenswürdige Teilnehmer festlegen muss [75]. Damit geht allerdings auch das vertrauenslose Modell der Blockchain verloren.

Letztendlich lässt sich sagen, dass Public Blockchains nicht skalieren. Um dies zu lösen, müsste die Netzwerktopologie verbessert werden um schnelle Blockpropagrationszeiten zu erlauben [65]. Weitere Schwierigkeiten bestehen sobald nicht nur Geldtransferaktionen verarbeitet werden müssen. Betrachtet man das CAP-Theorem wird ersichtlich, dass nur die Eigenschaften Dezentralisierbarkeit und Sicherheit gegeben sind. Es ist jedoch zu bedenken, dass viele Probleme der Skalierbarkeit aufgrund der genutzten Konsensmechanik bestehen. Auch wenn es teilweise Lösungsvorschläge für diese gibt, genügen sie bisher nicht um Skalierbarkeit herzustellen. Deshalb gilt es, die Limitationen von Permissioned Blockchains sowie alternative Konsensmechaniken für diese zu analysieren.

5.1.3 Permissioned Blockchains

Permissioned Blockchains werden eingesetzt, wenn nur bestimmte Teilnehmer an der Blockchain teilnehmen sollen. Dadurch entsteht eine stärkere Zentralisierung als bei Public Blockchains. Bezieht man sich auf das CAP-Theorem, müssten sich dadurch die Sicherheit und/oder Skalierbarkeit verbessern. Dies führt allerdings auch dazu, dass ein größeres Maß an Vertrauen zwischen den Teilnehmern gegeben sein muss. Dies wird dadurch sichergestellt, dass jeder Teilnehmer die Rechte zur Teilnahme am Netzwerk erhalten hat und die Identitäten dieser bekannt sind. Duch letzteres ist nachverfolgbar, welche Teilnehmer welche Transaktionen ausführt [65].

Scherer behauptes, dass das größere Vertrauen es erlaubt den Nodes verschiedene Aufgaben zuzuteilen. Dies beschreibt er am Beispiel von Hyperledger Fabric, einer Permissioned-Blockchain. In dieser gibt es Peer und Ordering Nodes. Erstere simulieren das ausführen von der Transaktionen und der damit verbundenen Datenänderungen. Letztere bestimmen die Reihenfolge der auszuführenden Transaktionen in den Blöcken. Peer Nodes führen die bereits simulierten Transaktionen nacheinander aus und erkennen Konflikte in den Transaktionen (Genauer im Kapitel 6.1 erklärt). Die Ordering Nodes sind also letztendlich für den Konsens verantwortlich. In Gegensatz zu Ethereum können Peer Nodes so parallel Transaktionen verarbeiten. Sie müssen sich nicht um eventuelle Konflikte oder die Reihenfolge der Transaktionen kümmern. Letztendlich würde die Skalierbarkeit, im Rahmen des Verarbeitens von Transaktionen, nur von der Hardware der Peers abhängen [65].

Scherer führt ebenfalls Tests durch um die Performance von Hyperledger Fabric zu analysieren. Dazu nutzt er eine frühe und unstabile Version 1.0. Das Netzwerk besteht aus einer Ordering und einer Peer Node. Es wird kein Konsensmechanismus genutzt. Die Anwendung selber unterstützt die Zahlung mittels digitaler Assets (z.B. Tokens bzw. Coins) zwischen 2 Accounts. Dabei erreicht er einen maximalen Transaktionsdurchsatz von 350 TPS. Dabei ist allerdings zu bedenken, dass der Test auf einer Maschine mit limitierten Ressourcen ausgeführt wird. Um einen maximalen Transaktionsdurchsatz zu erreichen, müssten mehrere leistungs-

starke Computer für den Test eingesetzt werden. Scherer stellt ebenfalls fest, dass der Transaktionsdurchsatz abnimmt, desto mehr Peers es gibt, welche Transaktionen bestätigen. Dies liegt daran, dass diese sogenannten Endorser untereinander kommunizieren müssen. Pro Node müssten $O(n^2)$ Nachrichten gesendet werden, wobei n die Anzahl an Nodes ist. Die Anzahl an effizient nutzbaren Endorsern ist also beschränkt. Auch hier ist jedoch zu bedenken, dass ein Test mit leistungsstarken Computern ausgeführt werden muss um die Skalierbarkeit dieser festzustellen [65].

Ein Paper von Pongnumkul vergleicht die Leistung von Hyperledger Fabric mit Ethereum. Er nutzt dazu die stabile Version 0.6. Er führt die Test ebenfalls mit nur einer Peer Node durch. Zur Ordering Node macht er keine Angabe. Die Anwendung ist die gleiche wie bei Scherer und es wird ebenfalls kein Konsensmechanismus genutzt. Pongnumkul stellt fest, dass die Performance von Fabric in allen Kriterien besser ist als bei Ethereum. So betrug die Zeit, bis eine Beispieltransaktion verarbeitet wurde bei Ethereum 41 Sekunden und bei Ethereum 478 Sekunden. Tests zum maximalen Transaktionsdurchsatz haben ergeben, dass Ethereum 40 TPS und Hyperledger Fabric 300 TPS erreicht hat. Die dazugehörige Abbildung 5.3 zeigt auch, dass die Unterschiede zwischen Ethereum und Fabric signifikanter sind, desto mehr Transaktionen verarbeitet werden müssen [**?**].

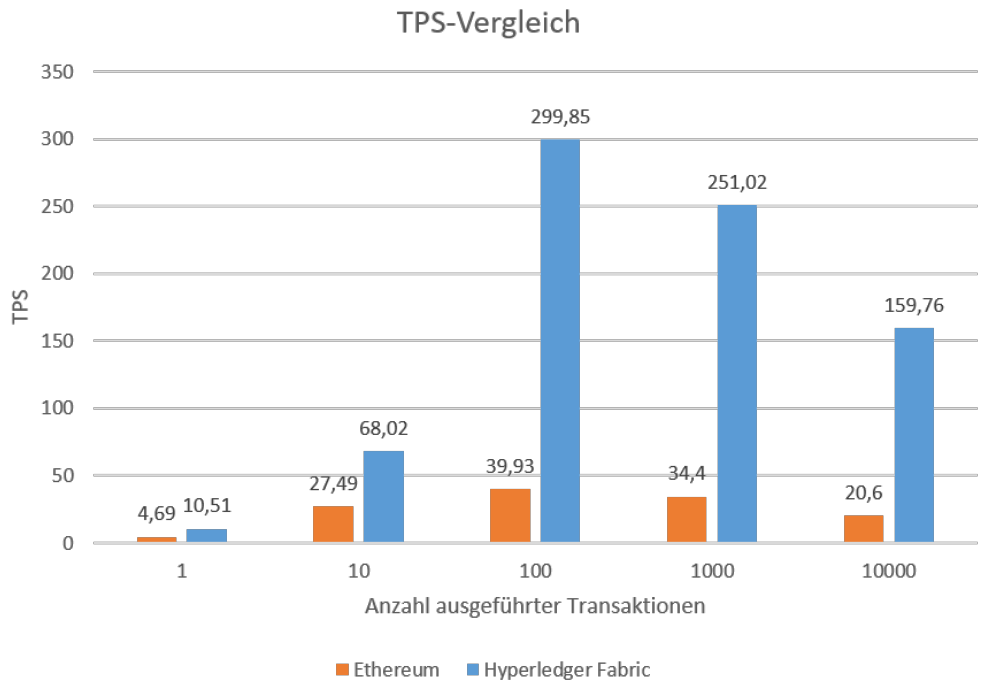

Abbildung 5.3: Vergleich des Transaktionsdurchsatzes von Ethereum und Hyperledger Fabric [**?**].

Bei beiden Tests ist zu bedenken, dass sie mit nicht aktuellen Versionen von Hyperledger Fabric ausgeführt wurden. Mittlerweile gibt es eine stabile Version 1.0, sowie eine Preview von Version 1.1.0 [32]. Es ist also möglich das die Performance sich mittlerweile verbessert hat.

Letztendlich lässt sich sagen, dass Permissioned Blockchains, was die Verabeitung von

Transaktionen betrifft, eine bessere Performance erzielen. Damit bestätigt sich auch das CAP-Theorem bezüglich der Skalierbarkeit. Die Performance wurde allerdings noch nicht unter der Nutzung verschiedener Konsensmechanismen betrachtet. Weiterhin muss das CAP-Theorem noch auf die Sicherheit untersucht werden. Deshalb erfolgt im nächsten Kapitel die Analyse der Skalierbarkeit und Sicherheit von Konsensmechanismen.

5.2 Konsensmechanismen

Das Erreichen von Konsens in einer Blockchain, ist eine Abwandlung des Byzantine Generals Problem. In diesem gibt es Generäle, welche Armeen kommandieren welche eine Stadt umzingeln. Ein Angriff auf diese ist nur erfolgreich, wenn alle Armeen gleichzeitig angreifen. Die Generäle müssen untereinander kommunizieren und Konsens darüber herstellen, ob ein Angriff erfolgen soll. Allerdings gibt es Verräter unter diesen. Es handelt sich also um ein vertrauensloses Umfeld. Genau so kann es in einer Blockchain zu den sogenannten Byzantine Faults kommen, wenn nicht vertrauenswürdige Nodes Daten manipulieren können. Deshalb müssen verteilte Systeme Byzantine Fault Tolerance (BFT) herstellen, um eine gewisse Anzahl an nicht vertrauenswürdigen Teilnehmern zu tolerieren. Dies geschieht in Blockchains über die Konsensmechanismen [77] [23].

Eine Blockchain, welche den Proof-of-Work als Konsensmechanismus nutzt, ist nicht skalierbar. Ebenfalls würde er in Netzwerken mit relativ wenig Teilnehmern die Sicherheit beeinträchtigen, da ein Teilnehmer einfacher 51% der Rechenleistung erreichen kann. Für Permissioned Blockchains muss also ein Konsensmechanismus gefunden werden, welche Skalierbarkeit und Sicherheit herstellt. Aufgrund des höheren Vertrauens in Permissioned Blockchains, behauptet Scherer, dass ein Konsensmechanismus genutzt werden kann, welcher Vertrauen in geringerem Maße als der Proof-of-Work garantiert. Somit könnten Skalierbarkeit und Sicherheit hergestellt werden [65]. Im folgenden werden verschiedene Konsensmechanismen verschiedener Blockchain-Technologien miteinander verglichen. Dabei ist zu bedenken, dass nur auf die Konsensmechanismen, welche Sicherheit und Skalierbarkeit in Permissioned Blockchains sicherstellen, im Detail eingegangen wird.

5.2.1 Proof of Stake

Beim PoS hängt die Wahrscheinlichkeit, einen Block zu minen, von der Menge des Eigentums (z.B. Kryptowährung) eines Nutzers ab. Desto mehr Bitcoins man beispielsweise hat, desto höher ist die Wahrscheinlichkeit für das Mining ausgewählt zu werden. Ähnlich wie beim PoW könnte ein Teilnehmer mit 51% aller Bitcoins das Netzwerk angreifen, da er die längere Blockchain erstellen kann. Aber selbst wenn es ein Teilnehmer schafft, 51% der Bitcoins zu besitzen, hätte er keine Motivation dazu. Denn letztendlich würde ein Angriff den Kurs von Bitcoin senken, und somit würde der Miner sich selber schaden [77]. Da der PoS hauptsächlich nur bei Kryptowährungen genutzt werden kann, ist er für Permissioned Blockchains uninteressant.

5.2.2 Proof of Elapsed Time

Der PoET wird in Intels Blockchain-Technologie Hyperledger Sawtooth genutzt. Die grundlegende Idee ist, dass eine Node eine zufällige Zeit generiert, welche Sie warten muss um einen Block zu erstellen. Um sicherzustellen, dass die generierte Zeit nicht verfälscht wurde, wird Trusted Computing [3] genutzt. So stellen Intels Software Guard Extensions (SGX) sicher, dass Code nicht modifiziert werden kann. Eine Node muss also über solchen unmodifizierbaren Code eine Zeit generieren. Weiterhin erfolgen statistische Tests, um zu verhindern dass eine Node Blöcke zu schnell und somit zu oft erstellt. Letztendlich ist die Blockerstellung damit fair verteilt, und kein Teilnehmer kann die Blockchain kontrollieren. Im Prinzip funktioniert der Mechanismus wie der PoW. Dort wird eine Wartezeit durch das Finden eines Hashes sichergestellt, während dies beim PoET durch die Hardware sichergestellt wird. Dadurch, dass es keine rechenintensiven Aufgaben gibt, ist die Skalierbarkeit sicher gestellt. Es ist jedoch zu bedenken, dass es bisher wenige Analysen zu der Sicherheit des PoET gibt. Ein Paper von Chen stellt fest das der Konsensmechanismus unter bestimmten Umständen unsicher ist, schlägt aber auch Lösungen dafür vor. Ebenfalls kommt hinzu, dass man der Hardware von Intel für das Trusted Computing vertrauen muss. [45].

5.2.3 Diversity Mining Consensus

Der Diversity Mining Consensus ist ähnlich dem PoW. In diesem ist der Block einer Node nur valide, wenn sie eine bestimmte Anzahl an vorherigen Blöcken nicht erstellt hat. Dies führt letztendlich dazu, dass eine Wartezeit für jede Node nicht durch das Finden eines Hashes (Siehe PoW) oder durch eine generierte Zeit (Siehe PoET) realisiert wird. Auch hier kann es zu Forks kommen, wenn 2 Nodes zur ungefähr gleichen Zeit einen Block erstellen. Ebenfalls wird die entstehende längere Blockchain akzeptiert. Durch die Wartezeit wird sichergestellt, dass der Branch mit den meisten Minern die längere Blockchain erstellt. Die Gefahr bei diesen Konsensmechanismus ist, dass bösartige Miner sich zusammen tun können. Unter den richtigen Umständen erstellt der Zusammenschluss die meisten Blöcke, und kann somit auch bei Forks eine längere Blockchain erstellen, um so Double-Spend-Angriffe auszuführen. Die Skalierbarkeit ist aufgrund des fehlenden Rechenaufwands gegeben. [54][43].

5.2.4 QuorumChain

Quorum ist ein Fork von Ethereum für Permissioned Blockchains. Dieser nutzt den Konsensmechanismus QuorumChain. Es gibt Voter und Block-Maker Nodes. Die Block-Maker schlagen Blöcke zum erstellen vor, und die Voter stimmen für diese ab. Um Forks zu verhindern, warten die Block-Maker eine zufällige Zeit und erstellen den Block. Die Voter validieren diesen und stimmen für ihn ab, indem Sie u.a. die Transaktionen ausführen und überprüfen ob der vorherige Block genug Stimmen erhalten hat. Die Nodes hängen letztendlich den Block an ihre

[3] Trusted Computing: Soft- und Hardware stellen sicher, dass ein Computer sich wie erwartet verhält [37]

lokale Chain an, welcher einen bestimmten Schwellwert an Stimmen überschritten hat, oder im Falle eines Forks die meisten Stimmen erhalten hat. Die Sicherheit des Mechanismus ist fragwürdig. Schon allein mit einen bösartigen Block-Macker würde es ständig zu Forks kommen, welche Double-Spend-Angriffe ermöglichen und die Zeit erhöhen bis eine Transaktion nicht mehr verworfen werden kann [43].

5.2.5 Practical Byzantine Fault Tolerance

Der PBFT gehört zu der Familie der BFT-Protokolle. Beim PBFT wählen die Teilnehmer eine Leader-Node. Jede Runde schlägt diese diese einen neuen Block mit auszuführenden Transaktionen vor. Dieser wird an alle anderen Nodes weitergeleitet. Anschließend wird der Konsens hergestellt. 2/3 der Nodes müssen dem im Block enthaltenen Transaktionen zustimmen, damit er erstellt wird. Erst dann werden die Transaktionen bei jeden Teilnehmer ausgeführt. Deshalb können bis zu 1/3 der Nodes unvertrauenswürdig sein. Ein Angreifer müsste für einen Angriff die Kontrolle über 2/3 der Nodes haben [69][77].

Vukolic behauptet, dass es BFT-Protokolle gibt, welche einen Transaktionsdurchsatz von mehreren 10000 TPS unterstützen. Die Skalierbarkeit dieser bezüglich der Anzahl an Nodes ist jedoch begrenzt [74]. Croman erzielt bei seinen Tests mit dem PBFT, bei 8 Nodes und 8192 auszuführenden Transaktionen einen Transaktionsdurchsatz von 14000 TPS. Weiterhin wird ersichtlich wie die Performance mit der Anzahl an Nodes abnimmt. Mit 64 Nodes und 8192 auszuführenden Transaktionen wird ein Transaktionsdurchsatz von 4500 TPS erreicht [46]. Im Gegensatz zum PoW besteht hier eine bessere Skalierbarkeit bezüglich des Transaktionsdurchsatzes, allerding ist sie bezüglich der Anzahl an Teilnehmern begrenzt [74].

Ein weiterer Vorteil von BFT-Protokollen ist, dass es Consensus Finality gibt. Das bedeutet, dass es nicht zu Forks kommen kann. Somit müssten Nutzer nicht darauf warten, dass mehrere Blöcke nach einer Transaktion erstellt werden, damit die Sicherheit gegeben ist, dass diese endgültig bestehen wird. Somit entfällt auch die Gefahr von Double-Spend-Attacken [74].

Die Angriffe, welche mit mehr als 1/3 der Voting Power erfolgen können, sind die sogenannten Censorship Attacks. So könnten Nodes verhindern, dass neue Blöcke entstehen, indem sie ihre Stimme zurückhalten. Weiterhin könnten sie bestimmte Transaktionen zensieren, indem sie dafür stimmen diese nicht in einem Block aufzunehmen [36].

5.2.6 Tendermint

Tendermint ist eine Abwandlung des PBFT-Konsensmechanismus. Der größte Unterschied besteht darin, dass die Nodes, welche eine Block vorschlagen, in einem Round-Robin Verfahren ausgewählt werden. So gibt es in jeder Runde einen neuen Blockersteller. Hinzu kommt, dass die Teilnehmer mit ihren Coins, welche die Voting Power bestimmen, abstimmen. Die Coins werden für einen Vote für eine bestimmte Zeit an die Vote gebunden und damit für die weitere Nutzung gesperrt. Aufgrund der Asynchronität des Netzwerks kann es passieren, dass eine No-

de noch keine Information über einen bereits neu erstellten Block erhalten hat. Das führt dazu, dass er einen neuen Block auf der gleichen Höhe des bestehenden Blocks vorschlägt. Es kann dann zu einen Fork kommen, wenn beide Blöcke 2/3 der Voting Power erhalten. Dies ist nur möglich, wenn mindestens 1/3 der Voting Power bösartig genutzt werden. Diese Nodes können jedoch bestraft werden. Durch die an die Abstimmung gebundenen Coins, können Nodes, welche bösartig abgestimmt haben, bestraft werden. So werden die gebundenen Coins einfach zerstört [59][41]. Es ist zu bedenken, dass Tendermint-Konsensus nur mit einer Kryptowährung im vollen Umfang funktioniert.

5.2.7 Sonstige BFT-Konsensmechanismen

Neben den bisher erwähnten BFT-Konsensmechanismen, gibt es noch diverse andere welche das Prinzip des PBFT mehr oder weniger abwandeln. So gibt es Symbiont und R3 Corda mit BFT-SMaRt, Iroha mit Sumeragi, Kadena mit ScalableBFT und Chain mit Federated Consensus [43]. Aufgrund der Vielfalt und Detailreichheit jeder dieser Konsensmechanismen, wird nicht genauer auf sie eingegangen. Es genügt die Prinzipien von BFT-Protokollen anhand des PBFT zu verstehen.

5.3 Sonstige Einschränkungen

5.3.1 Private Transaktionen

In Blockchains wie bei Bitcoin und Ethereum ist es nicht möglich, private Transaktionen auszuführen. Das bedeutet, alle Transaktionen sind für alle Teilnehmer sichtbar. In Permissioned Blockchains kann es Fälle geben, wo dies nicht erwünscht ist. So soll z.B. eine Preisabsprache zwischen 2 Teilnehmern in der Blockchain dokumentiert werden, welche für andere nicht sichtbar sein soll.

In Quorum wird dies so realisiert, dass in einer Transaktion angegeben wird, welche Teilnehmer die Transaktion sehen dürfen. Diese wird anschließend verschlüsselt, und kann nur von den angegebenen Teilnehmern entschlüsselt werden. Die Transaktion wird im Netzwerk verteilt, und nur von den Teilhabern ausgeführt [35].

In Hyperledger Fabric werden Private Transaktionen über Channels ermöglicht. Dabei ist jeder Channel eine eigene Blockchain, mit verschiedenen Teilnehmern. So würde es beispielsweise einen öffentlichen Channel geben, an welchen alle Teilnehmer der Permissioned Blockchain teilnehmen. Zusätzlich würde es private Channel geben, welche nur zwischen bestimmten Teilnehmer bestehen würden. Keine Daten können zwischen den Channeln übertragen werden [65].

5.3.2 Datenmenge

Ein noch nicht angesprochenes Problem der Blockchain ist die Datenredundanz im Bezug auf die Datenmenge. Da keine Daten in der Blockchain gelöscht werden können, wächst sie stätig

an. So ist die Bitcoin-Blockchain im Moment 151GB groß [5]. Größere Datenmengen werden schneller erreicht, wenn nicht nur Geldtransferaktionen bestehen.

Die Datenmenge stellt ein Problem dar, da Teilnehmer ab einen bestimmten Punkt eventuell nicht mehr bereit sind die Blockchain auf ihrer Node zu speichern. Dies würde zu weniger Minern, und somit zu höherer Zentralisierung führen [65].

Kapitel 6

Dezentraler Wartungsmarkt - Implementierung des Prototypen

Das vorherige Kapitel hat gezeigt, dass es möglich ist Permissioned Blockchains aufzubauen, welche die im Kapitel 3 erwähnten Anforderungen erfüllen. So ist es möglich über z.B. Hyperledger Fabric einen Transaktionsdurchsatz von mindestens 350 TPS zu erreichen. Weiterhin gibt es Konsensmechanismen, welche 1/3 an unvertrauenswürdigen Nodes tolerieren und einen Transaktionsdurchsatz von ungefähr 4500 TPS, je nach Teilnehmeranzahl, erzielen. Zusätzlich erlauben Technologien wie Hyperledger Fabric und Quorum das ausführen von privaten Transaktionen. Im Folgenden Kapitel wird eine Blockchain-Technologie ausgewählt, und der dezentrale Wartungsmarkt anhand dieser implementiert.

6.1 Technologieauswahl

Aus den Anforderungen an den dezentralen Wartungsmarkt (Siehe Kapitel 3.2) ergeben sich die folgenden Anforderungen an die zu nutzende Plattform:

- Möglichkeit Permissioned Blockchains zu erstellen
- Möglichkeit eigene Programmlogik zu implementieren (Smart Contracts)
- Höchstmögliche Performance (Transaktionsdurchsatz)
- Höchstmögliche Skalierbarkeit im Bezug auf die Anzahl der Teilnehmer
- Konsensmechanismus mit höchstmöglicher Sicherheit und Performance
- Private Transaktionen
- Mindestens Version 1
- Gute Dokumentation und Community Support

Zunächst einmal sind öffentliche Blockchain-Plattformen, wie Bitcoin, Ethereum und Sawtooth Lake entfallen aus der Auswahl entfallen. Daraufhin wurden die Permissioned Blockchains, aufgelistet in der Tabelle 6.1, miteinander verglichen. Multichain, OpenChain sowie Chain Core konnten ausgeschlossen werden, da sie keine Smart Contracts unterstützen. Die

Plattformen mit den höchsten Transaktionsdurchsatz sind Hyperledger Fabric und Hyperledger Burrow. Burrow befindet sich jedoch noch in einer frühen Version, womit es ebenfalls nicht zur Auswahl steht [29]. Letztendlich steht so nur noch Hyperledger Fabric zur Auswahl.

Version 1 ist bereits im Juli 2017 erschienen [30]. Fabric bietet eine umfassende Dokumentation, sowie Community Support über RocketChat und StackOverflow [15][16]. Private Transaktionen werden über Channels realisiert [65]. Ein großer Vorteil von Hyperledger Fabric gegenüber anderen Plattformen, sind austauschbare Konsensmechanismen. Dadurch, dass es keinen festgelegten Konsensmechanismus gibt, kann je nach Use-Case ein Konsensmechanismus ausgewählt werden, welcher die benötigte Performance, Skalierbarkeit und Sicherheit herstellt [75]. Dies ist vor allem wichtig im Prototyping. Wenn der Prototyp vom entstehenden dezentralen Wartungsmarkt erweitert werden soll (z.B. um mehr Teilnehmer), kann ein neuer Konsensmechanismus gewählt werden welcher den neuen Anforderungen entspricht. Vukolic nennt ebenfalls den Vorteil, dass Fabric eine bessere Performance als andere Plattformen erzielt, da die Nodes nach Peer und Ordering Nodes aufgeteilt werden. Aufgrund dieser Gründe behauptet Vukolic auch, dass Hyperledger Fabric die Limitationen anderer Permissioned Blockchains löst [75]. Somit ist letztendlich Hyperledger Fabric die verwendete Technologie für den dezentralen Wartungsmarkt.

Unternehmen	**Technologie**	**Performance**	**Smart Contracts**
Coin Sciences	Multichain	100-1000 TPS	Nein
J.P. Morgan	Quorum	12-100 TPS	Ja
IBM	Hyperledger Fabric	10k-100k TPS	Ja
Coinprism	OpenChain	1000+ TPS	Nein
Chain	Chain Core	N/A	Nein
R3	Corda	N/A	Ja
Monax	Hyperledger Burrow	10k TPS	Ja

Tabelle 6.1: Vergleich diverser Permissioned Blockchain Plattformen [39][28]

6.2 Hyperledger Fabric und Composer - Grundlagen

6.2.1 Hyperledger Fabric

Hyperledger Fabric ist eine eine Blockchain-Plattform für Business-Netzwerke. Es ist darauf ausgelegt modular (z.B. austauschbare Konsensmechanisment) zu sein, um es einfach erweitern, und somit für möglichst viele Use-Cases nutzbar machen zu können [73]. Im folgenden wird das grundlegende Konzept von Hyperledger Fabric erklärt.

Chaincode Fabric erlaubt den Teilnehmern das Erstellen, Interagieren und Nachverfolgen von digitalen Assets. Diese bestehen letztendlich aus Ansammlungen von Key-Value-Paaren.

Für die Interaktion werden Transaktionen genutzt. Die Assets und Transaktionen sind u.a. im Chaincode definiert. Dieser ist letztendlich bei den Nodes im Netzwerk installiert [65]. Da der Chaincode Programmlogik abbildet, kann er auch als Smart Contract bezeichnet werden [7].

Identitätsverwaltung Jede Node im Netzwerk muss eine Identität erhalten. Nur so können die Teilnehmer die Daten lesen und Transaktionen ausführen [65]. Die Registrierung sowie das Erstellen von Zertifikaten wird von einer Certificate Authority (CA) übernommen. Die Teilnehmer selber können CA's sein. So würde zum Beispiel jedes Unternehmen Identitäten und Zertifikate für seine Mitarbeiter erstellen [14].

State Database Jede Node speichert die Blockchain, und zusätzlich eine sogenannte State Database. Diese speichert den aktuellsten Status der digitalen Assets. Anders formuliert, wird sie aus den in der Blockchain enthaltenen Transaktionen erstellt. Neue in Blöcken enthaltene Transaktionen werden auf der State Database ausgeführt. Dies ermöglicht eine hohe Performance: Da die Datenbank im Arbeitsspeicher abgelegt werden kann, sind schnelle Schreib-und Lesevorgänge möglich [65].

Transaktionsfluss: Clients, Peer Nodes, Ordering Nodes In einen Hyperledger Fabric Netzwerk einigen sich die Unternehmen auf den zu nutzenden Chaincode für eine Anwendung. Dieser wird in der Blockchain gespeichert. Clients können über bestimmte Anwendungen Transaktionen über ihre Identität ausführen. Endorser Peer Nodes überprüfen die Rechte des Clients, die Validität der Transaktion, und simulieren diese. Dazu führen sie die Transaktion auf der State Database aus um die Datenänderungen zu erkennen. Diese werden jedoch noch nicht festgeschrieben. Anschließend werden die Transaktionen an eine Ordering Node geschickt. Diese sortiert die Transaktionen nach First-Come-First-Serve Prinzip in einen Block, welcher an die Committer Peer Nodes gesendet wird. Diese hängen den Block an die Blockchain an, und führen die Datenänderungen (Bereits simulierte Transaktionen) sequentiell auf der State Database durch. Dabei werden in Konflikt stehende Transaktionen erkannt, und als invalide gekennzeichnet [65].

Development Die Entwicklung für Hyperledger Fabric erfolgt über Chaincode, welcher in Java oder Go geschrieben wird [20]. Um eine schnellere und komfortablere Entwicklung zu erlauben, wird das Framework Hyperledger Composer genutzt. Dieses wird im nächsten Kapitel genauer betrachtet.

6.2.2 Hyperledger Composer

Hyperledger Composer ist ein Framework für die Anwendungsentwicklung mit Hyperledger Fabric. Es bietet verschiedene Funktionen, welche das implementieren von Blockchain-Applikationen beschleunigen. So ist es u.a. möglich Participants, Assets und Transaktionen zu modellieren,

Zugriffsregeln festzulegen und daraus Chaincode sowie REST-API's zu generieren [72]. Dies alles wird in den folgenden Kapiteln genauer erläutert. Es ist zu bedenken, dass Hyperledger Composer kontinuierlich weiterentwickelt wird, weshalb die hier gemachten Angaben nicht mehr aktuell sein müssen [31]. Die Implementierung erfolgt mit Composer v0.16.2.

6.3 Entwicklungsumgebung

Als Entwicklungsumgebung wird eine Vagrant-Box, basierend auf Ubuntu 16.04 genutzt. Dabei handelt es sich um leichtgewichtige VM, mit welcher eine SSH-Verbindung hergestellt wird. Dies erlaubt das Arbeiten mit der VM über die Kommandozeile [21]. Für die Box wird ein Provision-Script geschrieben, welches die benötigten Komponenten installiert um Hyperledger Fabric und Composer nutzen zu können. So gibt es beispielsweise das Composer Command Line Interface (CLI), damit man über die Kommandozeile u.a. Chaincode installieren kann [9]. Hyperledger Composer beinhaltet ebenfalls eine Fabric Blockchain-Konfiguration, mit welcher man eine Blockchain mit einem Peer für Entwicklungszwecke starten kann. Später wird Im Kapitel 6.6 eine eigene Netzwerkkonfiguration erstellt.

6.4 Business Network Definition

Der erste Schritt der Implementierung ist die Entwicklung der Programmlogik. Diese wird über die Business Network Definition (BND) von Composer erstellt. Aus ihr wird letztendlich der Chaincode generiert, welcher bei den Peer-Nodes installiert wird. Weiterhin besteht die Möglichkeit eine REST-API zu generieren, mit welcher die Nutzer mit den Chaincode interagieren können [8]. Die Anwendungslogik muss den folgenden Workflow erlauben: Maschinen, welche sich im Besitz von Unternehmen befinden, erkennen, dass sie eine Wartung benötigen. Sie führen mit einer von ihnen zugeteilten Identität eine Transaktion aus, welche einen Wartungsvertrag erstellt. Wartungsdienstleister können diesen annehmen. Sie melden sich beim Gerät an und führen die Wartung durch, wobei die Wartungsschritte geloggt werden. Nach ausgeführter Wartung schließt die Maschine den Vertrag. Bei allen Operationen ist zu bedenken, dass sie nur unter bestimmten Konditionen erfolgen dürfen. So kann z.B. ein Wartungsvertrag nur angenommen werden, wenn er nicht bereits von einen anderen Wartungsanbieter akzeptiert wurde. Weiterhin muss eine Preisabsprache zwischen Unternehmen und Wartungsdienstleister erfolgen können.

6.4.1 Anwendungslogik

Die BND besteht aus 4 Dateien. Ein Model-File modelliert die Participants, Assets, sowie Transaktionen. Ein JavaScript-File beschreibt den auszuführenden Code (welcher Daten erstellt und/oder bearbeitet), wenn eine Transaktion aufgerufen wird. Ein ACL-File definiert welcher Participant welche Daten lesen/bearbeiten/löschen darf. Zuletzt gibt es noch ein Query-File,

welches Queries definiert, mit welchen man Daten innerhalb der BND abfragen kann. Dieses wird im Rahmen des Prototypen jedoch nicht genutzt. Stattdessen werden die Filter der REST-API genutzt, um Datenabfragen auszuführen (Siehe Kapitel 6.5.1) [72].

Model-File Im Model File gilt es die Teilnehmer am Netzwerk, die verfügbaren Assets, die zu implementierenden Transaktionen sowie Events zu definieren. Die Modellierung erfolgt in der Composer Modeling Language [17]

Die Participants des dezentralen Wartungsmarktes sind Unternehmen (*Company*), Wartungsdienstleister (*MaintenanceProvider*) sowie Maschinen (*Machine*). Die Maschinen besitzen einen *Owner*-Key, mit welchen Sie den Unternehmen zugeordnet werden können. Der Grund, warum die Maschinen als Participant definiert sind, ist die Identitätsverwaltung. In Composer kann jeden Participant eine eindeutige Identität zugewiesen werden, über welche Transaktionen ausgeführt werden [18]. So kann z.B. eindeutig zugeordnet werden, welche Maschine einen Wartungsvertrag erstellt. Ein Beispiel für die Definition einer Machine findet sich unter dem Listing 6.1.

```
participant Machine identified by machineId {
    o String machineId
    o String type
    o String model
    --> Company owner
}
```

Listing 6.1: Modellierung einer Machine. Keys können primitive Datentypen oder Referenzen zu anderen Assets sein.

In der BNA bestehen 3 Typen von Assets. Der *MachineStatus* gehört zu einer Maschine, und gibt an ob sie funktionstüchtig ist. Der *MaintenanceContract* enthält u.a. Informationen über den Wartungsgrund und den Status der Wartung. Letztendlich gibt es noch die *PaymentAgreement*. Diese dokumentiert die abgesprochene Auszahlung zwischen Unternehmen und Wartungsanbieter für einen bestimmten Wartungsvertrag. Listing 6.2 zeigt die Definition eines MaintenanceContract.

Ebenfalls müssen die zu implementierenden Transaktionen definiert werden, welche letztendlich Assets erstellen und bearbeiten. Die Transaktion *InitMaintenance* wird von einer Maschine aufgerufen um einen Wartungsvertrag zu erstellen. *AcceptMaintenanceContract* wird von Wartungsanbietern aufgerufen um einen Vertrag zu akzeptieren. Ebenfalls wird von ihnen die Transaktion *AddPerformedStep* genutzt, um ausgeführte Wartungsschritte zu loggen. Die *CloseContract*-Transaktion wird von der Maschine aufgerufen, nachdem z.B. der Wartungsanbieter einen Knopf an der Maschine drückt, um zu signalisieren, dass die Wartung erfolgt ist. Dabei erfolgt eine Überprüfung, ob der letzte erforderliche Wartungschritt erfolgt ist. Letztendlich gibt es noch *CreatePaymentAgreement*, welche vom Unternehmen genutzt wird um zu einen Vertrag einen Vorschlag für die Zahlung zu erstellen sowie *AcceptPaymentAgreement*, womit

```
asset MaintenanceContract identified by maintenanceContractId {
    o String maintenanceContractId
    o String maintenanceReason
    o Boolean isAccepted
    o Boolean isClosed
    o String[] performedSteps
    o String requiredLastStep optional
    --> Machine owner
    --> MaintenanceProvider maintenanceProvider optional
    }
}
```

Listing 6.2: Modellierung eines MaintenanceContract.

der Wartungsanbieter diese akzeptieren kann. Listing 6.3 zeigt ein Beispiel für eine Transaktionsdefinition. Es gibt noch eine Transaktion welche nichts mit der Funktion der Anwendung zu tun hat: *SetupDemo* wird genutzt um automatisch Beispieldaten zu generieren, damit man schnell neue Funktionen sowie Änderungen an der BNA testen kann. An dieser Stelle ist es wichtig zu erwähnen, dass noch weitere Transaktionen, neben den eigen definierten, gibt. Dazu gehören die Standard-Transaktionen zum Abfragen, Erstellen, Bearbeiten und Löschen von Daten (Siehe 6.5.1).

```
    transaction AddPerformedStep {
        --> MaintenanceContract contract
        o String performedStep
    }
```

Listing 6.3: Modellierung der AddPerformedStep-Transaktion. Die Keys sind in diesem Fall die mit der Transaktion übergebenen Parameter.

Zuletzt müssen noch die Events erwähnt werden. Diese werden gesendet, wenn bestimmte Trigger ausgelöst werden. Client-Applikation können diese abonnieren, um so z.B. bei Datenänderungen benachrichtigt zu werden. So wird beispielsweise eine Website zum annehmen von Wartungsverträgen automatisch aktualisiert, sobald ein neuer Vertrag in der Blockchain erstellt wird. Die Events welche bestehen, sind *NewContractCreated* und *ContractClosed*. Ersteres wird im Listing 6.4 definiert.

Aufgrund der Länge des Model-Files wurden hier nur Beispiele gezeigt. Das komplette Model-File kann im Anhang A eingesehen werden.

JavaScript-File Im Model-File Kapitel wurden Transaktionen und Events nur definiert. Das Verhalten der Transaktionen sowie die Trigger der Events werden im JavaScript-File festgelegt. Aufgrund der Komplexität des Scripts wird nur beispielhaft die Implementation einer Transaktion sowie eines Events erläutert.

```
event NewContractCreated {
}
```

Listing 6.4: Modellierung eines Events, welches ausgelöst wird wenn ein neuer Vertrag erstellt wird.

```
/**
* Accept the MaintenanceContract
* @param {biz.innovationcenter.maintenance.AcceptMaintenanceContract} tx The
    ↪ transaction instance.
* @transaction
*/
function acceptMaintenanceContract(tx) {
    var error = null;

    //Check if the contract is already accepted
    if (tx.maintenanceContract.isAccepted === false) {
        //Check if the Participant has the required experience
        if (tx.maintenanceContract.owner.type === getCurrentParticipant().
            ↪ experienceWith) {
            //Set the new Contract Data for the contract parameter
            tx.maintenanceContract.isAccepted = true;
            tx.maintenanceContract.maintenanceProvider = getCurrentParticipant()
                ↪ ;

            // Get the contract from the asset registry
            return getAssetRegistry('biz.innovationcenter.maintenance.
                ↪ MaintenanceContract')
                .then(function (assetRegistry) {
                    // Update the contract in the asset registry.
                    return assetRegistry.update(tx.maintenanceContract);
                });
        } else {
            error = 'Error: Provider does not have required experience';
        }
    } else {
        error = 'Error: Contract is already accepted';
    }

    if (error !== null) {
        console.error(error);
        throw error;
    }
}
```

Listing 6.5: JavaScript-Code für die AcceptMaintenanceContract-Transaktion

Das Listing 6.5 enthält den JavaScript-Code für die AcceptMaintenanceContract-Transaktion. Der übergebene Parameter *tx* enthält die Keys, welche im Model-File für die Transaktion definiert wurden. Bevor die Transaktion ausgeführt wird, wird überprüft ob der Contract bereits akzeptiert wurde, und ob der Wartungsanbieter, welcher die Transaktion ausführt, die benötigte Erfahrung mit der zu wartenden Maschine hat (Siehe Zeile 9-12). In Zeile 14-15 werden die neuen Werte für die Keys im MaintenanceContract gesetzt, damit der Vertrag als akzeptiert gilt. Die Änderungen muss nun noch in der State Database ausgeführt werden. Dazu wird in Zeile 18-21 die Asset-Registry aufgerufen, welche alle Assets enthält, und das Update ausgeführt.

```
1  //...
2  .then(function (contractRegistry) {
3      return contractRegistry.add(contract);
4  })
5  .then(function() {
6      //Emit event
7      var factory = getFactory();
8      var newContractEvent = factory.newEvent(NAMESPACE, 'NewContractCreated');
9      emit(newContractEvent);
10 });
```

Listing 6.6: Auszug aus der InitMaintenance-Transaktion. Ein Event wird emittet, nachdem ein neuer Vertrag erstellt wurde.

Das Listing 6.7 zeigt einen Auszug aus der InitMaintenance-Transaktion. Nachdem in Zeile 2-3 ein Vertrag erstellt und der Asset-Registry hinzugefügt wurde, wird in Zeile 7-9 das im Model-File definierte Event *NewContractCreated* emittet.

Aufgrund der Länge des Model-Files wurden hier nur Beispiele gezeigt. Das komplette Model-File kann im Anhang B eingesehen werden.

ACL-File Die Zugriffsregeln (ACL-Rules) bestimmen die Schreib-und Leserechte der einzelnen Participants. Eine Art solche Regeln festzulegen wurde schon im vorherigen Abschnitt kurz gezeigt. Im JavaScript-Code im Listing 6.5 in Zeile 12 wird überprüft, ob ein Anbieter die benötigte Erfahrung mit der zu wartenden Maschine hat. Es empfielt sich jedoch solche Regeln im ACL-File festzulegen, damit man eine zentrale Stelle für diese hat, und die eventuell fehlende Berechtigung vor dem Ausführen des Codes erkennt. Ein weiteres Beispiel für eine ACL-Rule wäre, dass nur Wartungsanbieter, welche einen Wartungsvertrag angenommen haben, durchgeführte Wartungsschritte eintragen dürfen. Ein Beispiel dafür wäre Listing **??**. In Zeile 6 wird überprüft, ob der im Vertrag angegebene Wartungsanbieter die Transaktion ausführt.

Den ACL-Rules fehlt in der genutzten Version von Composer ein nützliches Feature. Man kann Participants Rechte nur auf gesamte Assets, allerdings nicht auf einzelne Keys dieser Assets geben. Damit ein Wartungsanbieter die eben genannte Transaktion ausführen kann, benötigt er Update-Rechte für den Wartungsvertrag. Damit könnte er über eine Standard-

Update-Transaktion den Status des Vertrags auf geschlossen stellen, obwohl nur eine Maschine die *CloseContract*-Transaktion ausführen kann. Das einzige was Participants davon abhält andere Teilnehmer dadurch zu schädigen, ist die Tatsache das jede ausgeführte Transaktion und die dazugehörige Identität in der Blockchain gespeichert wird. Die vollständigen Definitionen können dem Anhang C entnommen werden.

```
1 rule ProviderCanExecuteAddPerformedStepTransaction {
2     description: "Maintenance provider can add performed maintenance steps to
        ↪ his accepted contract"
3     participant(p): "biz.innovationcenter.maintenance.MaintenanceProvider"
4     operation: CREATE
5     resource(r): "biz.innovationcenter.maintenance.AddPerformedStep"
6     condition: (r.contract.maintenanceProvider.getIdentifier() == p.
        ↪ getIdentifier())
7     action: ALLOW
8 }
```

Listing 6.7: Auszug aus der InitMaintenance-Transaktion. Ein Event wird emittet.

6.4.2 Installation

Die Schritte für die Installation der BND erfolgen über die Composer CLI. Zunächst wird aus ihr ein Business Network Archive (BNA) generiert. Dieses wird bei den einzelnen Peers installiert, welche durch ein Connection Profile angegeben werden. Dieser Vorgang wird genauer im Kapitel 6.6 beschrieben. Für Entwicklungszwecke genügt zunächst die Installation bei den bereitgestellen Peer von Hyperleder Composer.

6.5 Client Applications

Nachdem die BND besteht, gilt es Client-Applikationen zu implementieren, um mit dieser zu interagieren. Dazu wird zuerst auf die REST-API eingegangen, welche die Vorraussetzung für die Anwendungen darstellt. Anschließend wird auf die Angular-Web-Anwendungen und die Gerätesimulation über eine Node.js Anwendung eingegangen. Um zu garantieren, dass die Client-Applications nicht von einer zentralen Instanz verwaltet werden, würde jeder Teilnehmer selber Rest-Server sowie die Applikationen hosten.

6.5.1 REST-API

Hyperledger Composer bietet die Möglichkeit, eine REST-Schnittstelle aus der BND zu generieren. Sie erlaubt das Abfragen, Erstellen, Bearbeiten und Löschen von Assets, sowie das ausführen von Transaktionen. Dies erfolgt über GET oder POST Requests an bestimmte URL's. Beispiele dazu werden im folgenden Kapitel genannt. Zu der generierten API gehört ebenfalls

eine Weboberfläche (Siehe Abb. 6.1, welche einen Überblick über alle zur Verfügung stehenden REST-Aufrufe gibt, sowie die Ausführung dieser erlaubt. An dieser Stelle muss erwähnt werden, dass die REST-API im Prototypen nur einen eingeloggten Nutzer unterstützt. Dieser wird im Source Code festgelegt. Dies führt dazu, dass für das Ausführen von Transaktionen mit unterschiedlichen Identitäten, unterschiedliche REST-Server genutzt werden müssen.

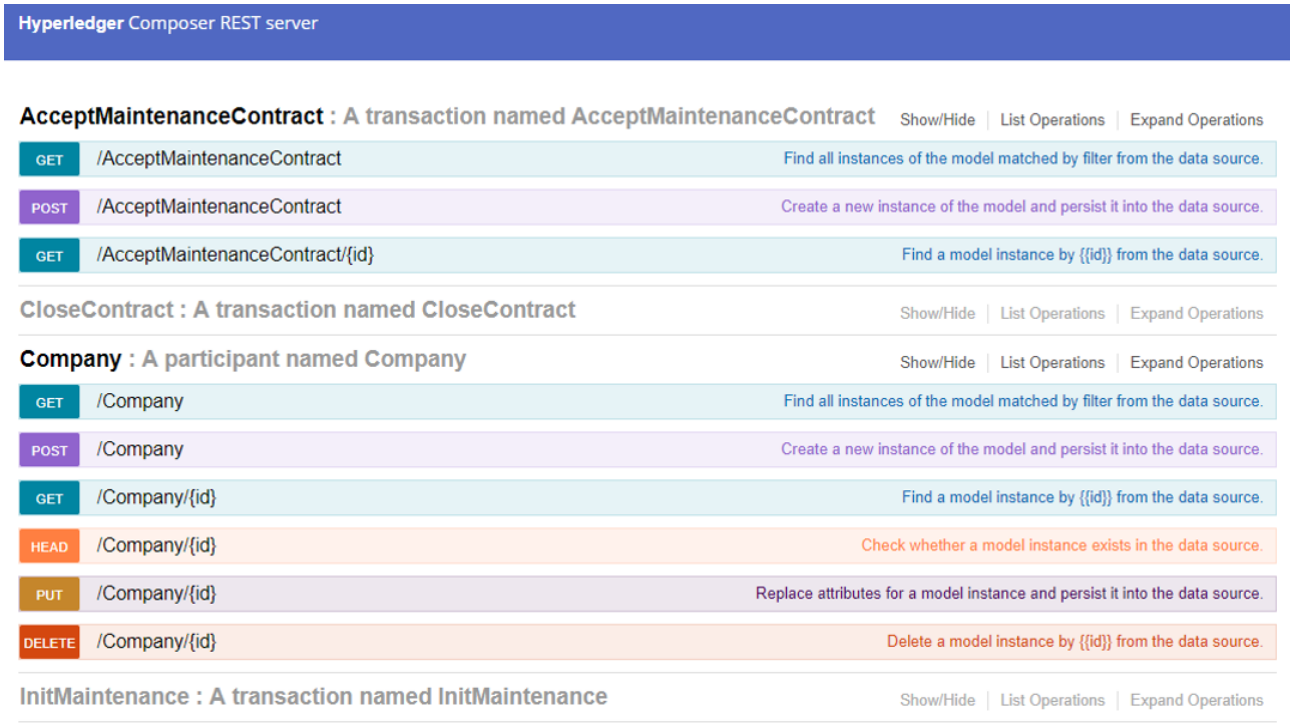

Abbildung 6.1: Generierte REST-API zu der BND. GET und POST Requests werden für Datenabfragen sowie das Ausführen von Transaktionen genutzt.

6.5.2 Webanwendungen

Maintenance-App Die Maintenance-App ist eine Angular-Webanwendung. Mit ihr können Wartungsanbieter noch nicht angenommene Wartungsverträge einsehen und akzeptieren. Weiterhin kann das Loggen von Wartungsschritten sowie das Überprüfen der Vertragsschließung erfolgen. Ein Screenshot der Anwendung findet sich bei der Abbildung 6.2. In der aktuellen Implementationen ist der eingeloggte Nutzer durch den genutzten REST-Server definiert. Im folgenden werden Codebeispiele zu den Abfragen und Erstellen von Assets sowie dem abonnieren von Events gezeigt.

Listing 6.8 zeigt die Abfrage aller existierenden Wartungsverträge über einen GET-Request. Dieser wird an die in Zeile 2 angegebene URL, an den REST-Server geschickt.

Es ist ebenfalls möglich GET-Requests mit Filtern auszuführen. Im Listing 6.9 werden nur Wartungsverträge abgefragt, welche von den eingeloggten Wartungsanbieter akzeptiert, aber

```
public fetchAllContracts() : Observable<any> {
    let API_BASE = "http://10.40.94.180:12100/api/";
    return this.http.get(API_BASE + "MaintenanceContract");
}
```

Listing 6.8: Abfrage aller existierenden Wartungsverträge

```
public getAcceptedContracts() : Observable<any> {
    let API_BASE = "http://10.40.94.180:12100/api/";
    let PROVIDERNAME = "Aintenance";

    return this.http.get(API_BASE + "MaintenanceContract?filter=%7B%22where%22%3
        ↪ A%7B%22and%22%3A%5B%7B%22isAccepted%22%3Atrue%7D%2C%7B%22isClosed%22%3
        ↪ Afalse%7D%2C%20%7B%22maintenanceProvider%22%3A%22resource%3Abiz.
        ↪ innovationcenter.maintenance.MaintenanceProvider%23" + PROVIDERNAME +
        ↪ "%22%7D%5D%7D%7D");
}
```

Listing 6.9: Abfrage aller existierenden Wartungsverträge

noch nicht geschlossen wurden. Die dazu in Zeile 5 entstandene URL wurde von der REST-Weboberfläche generiert.

```
public addPerformedStep(operation : string, contractId : string) : Observable<
    ↪ any> {
    const body = {
        "$class": "biz.innovationcenter.maintenance.AddPerformedStep",
        "contract": contractId,
        "performedStep": operation
    };
    return this.http.post(API_BASE + 'AddPerformedStep', body);
}
```

Listing 6.10: Abfrage aller existierenden Wartungsverträge

Zuletzt wird ein Beispiel zum Ausführen einer Transaktion gezeigt. Wartungsanbieter können über die *AddPerformedStep*-Transaktion die Wartungsschritte loggen. Listing 6.11 zeigt den dafür auszuführenden POST-Request. In Zeile 2-6 wird ein JSON-Objekt erstellt, welches die auszuführende Transaktion sowie ihre Parameter festlegt. Dazu gehört der zu bearbeitende Vertrag, sowie der ausgeführte Wartungschritt. Letztendlich wird das JSON-Objekt als Parameter der POST-Request übergeben, und der im JavaScript-File der BND angegebene Code der Transaktion ausgeführt.

Composer Playground Der Hyperledger Composer Playground ist eine in Composer enthaltene Webanwendung (Siehe Abb. 6.3). Sie wird hauptsächtlich während der Entwicklung

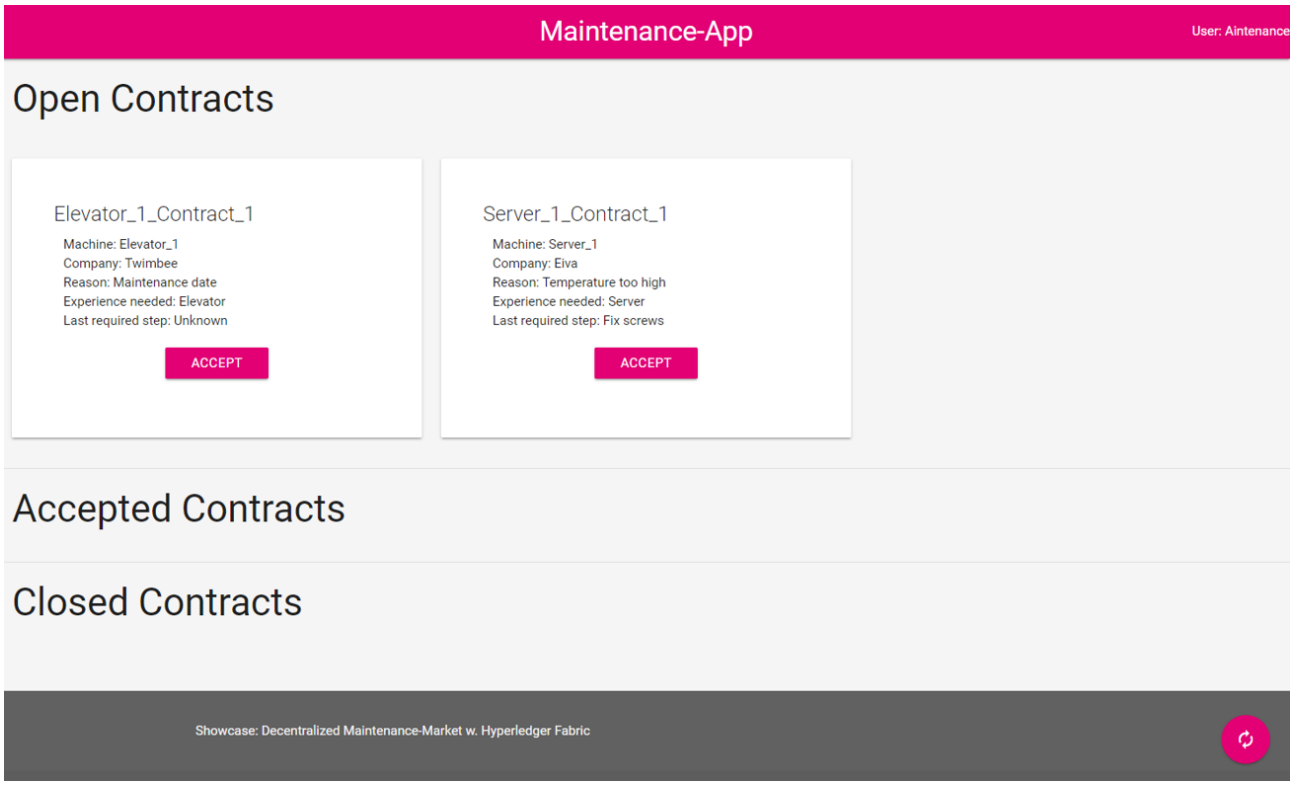

Abbildung 6.2: Angular-Maintenance-App. Vom Wartungsdienstleistern zum Verwalten der Wartungsverträge genutzt.

zum Testen der BND verwendet wird. Über den Playground können Participants sowie Identitäten für diese erstellt werden. Ebenfalls ist es möglich Assets zu erstellen und zu bearbeiten. Was die Anwendung jedoch auch für den Endnutzer attraktiv machen, ist dass sie einen Überblick über alle bestehenden Daten gibt. Weiterhin ist es möglich Transaktionen ausführen und die Historie aller durchgeführten Transaktionen einzusehen. Es ist jedoch nicht möglich Daten zu filtern. Im bestehenden Prototypen soll der Playground hauptsächlich von den Unternehmen eingesetzt werden, um die Daten der Maschinen und Wartungsverträge einzusehen. Weiterhin muss er in der aktuellen Version von den Unternehmen und Wartungsanbietern genutzt werden um die Preisabsprachen in der Blockchain zu dokumentieren.

6.5.3 XDK-Trigger

Um die Simulation von Wartungsgeräten zu realisieren, wird ein Bosch XDK genutzt. Dabei handelt es sich um ein Gerät mit diversen Sensoren. So misst es u.a. Temperatur, Luftfeuchtigkeit sowie Beschleunigung (bzw. G-Kräfte). Auf dem XDK ist ein Programm installiert, welches die Sensordaten über USB per Serial Port überträgt. Wenn bestimmte Daten angeliefert werden, sollen Transaktionen ausgeführt werden, um Wartungsverträge zu erstellen und zu schließen. Dafür wird eine Node.js-Anwendung geschrieben.

Um die Daten mit der Node.js Anwendung zu lesen wird das Package *node-serialport* [34] ge-

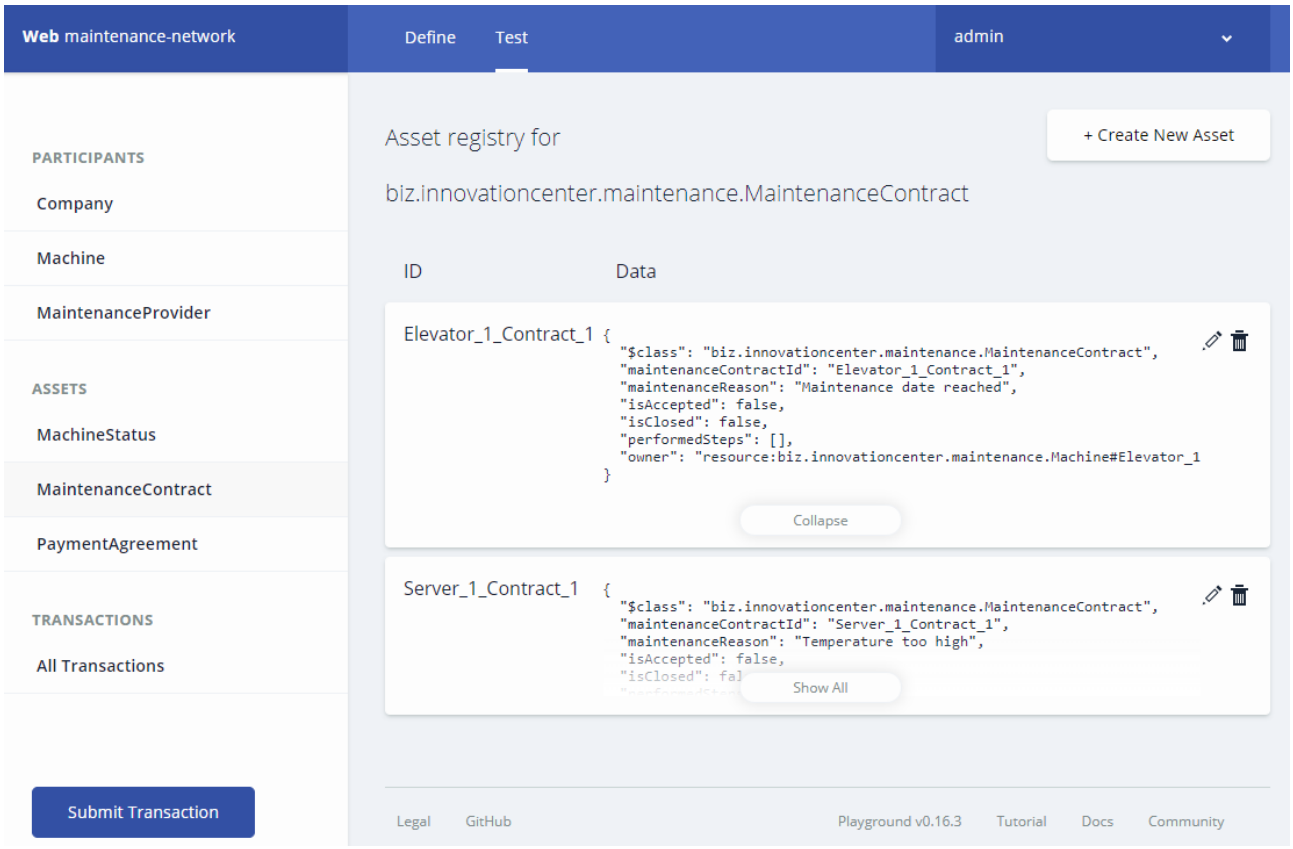

Abbildung 6.3: Hyperledger Composer Playground

nutzt. Ein Wartungsvertrag wird erstellt, sobald die Luftfeuchtigkeit einen bestimmten Schwellwert überschreitet. Das Schließen von Wartungsverträgen erfolgt, sobald das Gerät auf den Kopf gedreht wird. Dies würde das Drücken eines Knopfes an der zu wartenden Maschine simulieren, womit der Wartungsanbieter die fertige Wartung signalisiert sowie die Schließung des Vertrags beantragt. Damit die Transaktionen mit der Identität einer Maschine ausgeführt werden, muss ein dementsprechend konfigurierter REST-Server existieren.

Listing **??** zeigt, wie mit dem Überschreiten der Luftfeuchtigkeitsschwellwerts umgegangen wird. In Zeile 7 wird die Funktion *createSmartContract* aufgerufen. Dieser führt die InitMaintenance-Transaktion über einen POST-Request aus. Dabei werden zufällige Werte für die Parameter *contractId*, *maintenanceReason* und *lastRequiredStep* übergeben.

6.6 Netzwerkkonfiguration

Die Netzwerkkonfiguration besteht aus dem Erstellen einer Fabric-Netzwerk-Konfiguration, sowie dem Konfigurieren von Composer zum Installieren der BND.

```
function handleHumidityEvent() {
    var HUMIDITY_TRIGGER = 70;
    //If the humidity is too high, create a smart contract
    if (jsonData.humidity >= HUMIDITY_TRIGGER) {
        if (humidityFirstTimeExceeded) {
            //Execute InitMaintenance-Transaction via POST-Request
            console.log("HUMIDITY TOO HIGH! MY HAIR!!!");
            createSmartContract();
        }
        humidityFirstTimeExceeded = false;
    } else {
        humidityFirstTimeExceeded = true;
    }
}
```

Listing 6.11: Erstellen eines Wartungsvertrags bei der Überschreitung des Luftfeuchtigkeitsschwellwertes.

6.6.1 Fabric-Netzwerk-Konfiguration

Um das Blockchain-Netzwerk zu konfigurieren und zu starten, müssen Orderer Nodes, Peer Nodes, Certificate Authorities und Channel in mehreren Dateien definiert werden. Um diesen Prozess zu vereinfachen, wird das Tool *netcomposer* [33] genutzt. Dieses erstellt aus einem einzigen Kondifurationsfile alle benötigten Dateien um ein Blockchain-Netzwerk zu starten.

In dem zu erstellenden Netzwerk soll es 2 Unternehmen (Eiva und Twimbee), 2 Wartungsanbieter (Repairr und Aintenance), sowie 2 Maschinen (Server_1 und Machine_1), welche zu den Unternehmen gehören, geben (Siehe Abb. 6.4). Eine verkürzte Konfiguration dafür ist im Listing 6.12 angegeben. In Zeile 1 wird der genutzte Konsensmechanismus angegeben. In der aktuellen Implementation gibt es nur eine Ordering Node. Der Grund dafür wird genauer im Kapitel 6.7 erläutert. In Zeile 4 wird die für die State Database zu nutzende Datenbank definiert. Von Zeile 10-21 werden die Channels konfiguriert. Der Channel *mychannel* ist der öffentliche Channel, während *privatechannel* nur zwischen 2 Organisationen besteht.

Aus diesem Konfigurationsfile werden verschiedene Dateien generiert, u.a. auch ein Docker-Compose-File. Fabric nutzt

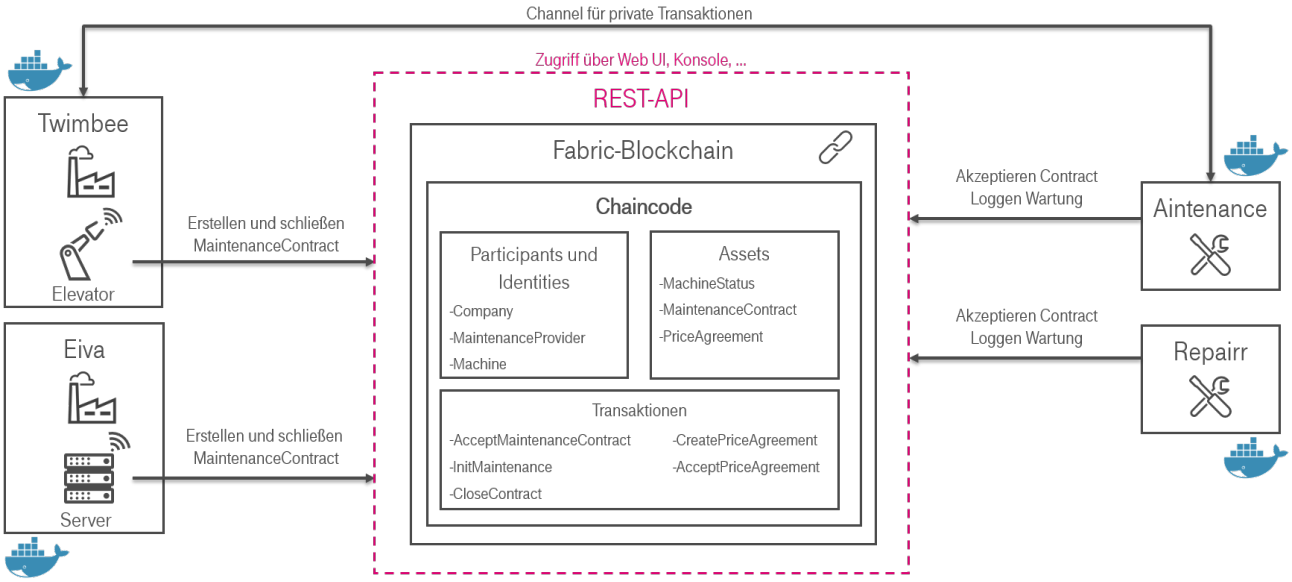

Abbildung 6.4: High-Level-Architektur des zu entstehenden Fabric-Netzwerks

6.6.2 Composer-Konfiguration

6.7 Konsensmechanismus

6.8 Showcase-Demo

6.9 Evaluierung

```
orderer:
    type: "solo"
db:
    provider: "CouchDB"

organizations:       4
peersPerOrganization:    1
usersPerOrganization:    1

channels:
    - name: mychannel
        organizations:
        - organization: 1
        - organization: 2
        - organization: 3
        - organization: 4

    - name: privatechannel
        organizations:
        - organization: 1
        - organization: 2
```

Listing 6.12: Konfiguration des Fabric-Netzwerks (Verkürzt).

Kapitel 7

Dezentraler Wartungsmarkt - Evaluierung

7.1 Evaluierung

Kapitel 8

Fazit und Ausblick

- Kurze Zusammenfassung
- Ausblick geben/Erweiterbarkeit des Systems beschreiben
- Ausblick zu Problemen von B2B-Blockchains geben

Literaturverzeichnis

[1] Access Control Language - Hyperledger Composer. https://hyperledger.github.io/composer/unstable/reference/acl_language.

[2] Bitcoin Energy Consumption Index. https://digiconomist.net/bitcoin-energy-consumption.

[3] BitcoinStats. http://bitcoinstats.com/network/propagation/.

[4] Blockchain Bikes. http://futurefluxfestival.nl/en/program/blockchain-bikes/.

[5] Blockchain Size. https://blockchain.info/blocks-size.

[6] Blockchain Wallet. https://wirexapp.com/guides/mobile-wallets/blockchain-wallet/.

[7] Chaincode - Hyperledger Fabric Documentation. http://hyperledger-fabric.readthedocs.io/en/release/chaincode.html.

[8] Developer Tutorial - Hyperledger Composer. https://hyperledger.github.io/composer/tutorials/develo tutorial.

[9] Development Environment - Hyperledger Composer. https://hyperledger.github.io/composer/installin tools.html.

[10] Ethereum Average BlockTime Chart. https://etherscan.io/chart/blocktime.

[11] Ethereum Network HashRate Growth Chart. https://etherscan.io/chart/hashrate.

[12] Glossar - Bitcoin. https://bitcoin.org/de/glossar.

[13] Hash Rate. https://blockchain.info/hash-rate.

[14] Hyperledger Fabric CA. http://hyperledger-fabric-ca.readthedocs.io/en/latest/users-guide.html#overview.

[15] Hyperledger Fabric Documentation. https://hyperledger-fabric.readthedocs.io/en/release/index.html.

[16] Hyperledger Fabric Support. https://hyperledger-fabric.readthedocs.io/en/release/questions.html.

[17] Modeling Language - Hyperledger Composer. https://hyperledger.github.io/composer/reference/cto_language.html.

[18] Participants and identities | Hyperledger Composer. https://hyperledger.github.io/composer//managing/participantsandidentities.

[19] Scalability - Bitcoin Wiki. https://en.bitcoin.it/wiki/Scalability.

[20] SDKs - Hyperledger Fabric Documentation. https://hyperledger-fabric.readthedocs.io/en/release/fabric-sdks.html.

[21] Vagrant by HashiCorp. https://www.vagrantup.com/index.html.

[22] Single Point of Failure. *Wikipedia*, July 2016. Page Version ID: 156306981.

[23] Byzantine Fault Tolerance: The Key for Blockchains. http://www.nasdaq.com/article/byzantine-fault-tolerance-the-key-for-blockchains-cm810058, June 2017.

[24] Ethereum White Paper, December 2017.

[25] Kryptologische Hashfunktion. *Wikipedia*, November 2017. Page Version ID: 170625494.

[26] Nonce. *Wikipedia*, August 2017. Page Version ID: 167799632.

[27] Programmierschnittstelle. *Wikipedia*, November 2017. Page Version ID: 170840602.

[28] Github: Hyperledger Burrow, January 2018.

[29] GitHub Releases: Hyperledger Burrow, January 2018.

[30] GitHub Releases: Hyperledger Fabric, January 2018.

[31] Hyperledger Composer - Releases, January 2018.

[32] Hyperledger Fabric Releases, January 2018.

[33] Netcomposer - Github Repository, January 2018.

[34] Node-serialport - Github Repository, January 2018.

[35] Quorum Wiki - Transaction Processing, January 2018.

[36] Tendermint: Tendermint Core (BFT Consensus) in Go, January 2018.

[37] Trusted Computing. *Wikipedia*, January 2018. Page Version ID: 819361025.

[38] Andreas M. Antonopoulos. *Mastering Bitcoin*. O'Reilly, Sebastopol CA, first edition edition, 2015. OCLC: ocn876351095.

[39] Elyes Ben Hamida, Kei Leo Brousmiche, Hugo Levard, and Eric Thea. Blockchain for Enterprise: Overview, Opportunities and Challenges. In *The Thirteenth International Conference on Wireless and Mobile Communications (ICWMC 2017)*, Nice, France, July 2017.

[40] Steven Buchko. How Long Do Bitcoin Transactions Take? https://coincentral.com/how-long-do-bitcoin-transfers-take/, December 2017.

[41] Ethan Buchman. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains*. Phd thesis, 2016.

[42] Vitalik Buterin. Toward a 12-second Block Time, July 2014.

[43] Christian Cachin and Marko Vukolić. Blockchain Consensus Protocols in the Wild. *arXiv:1707.01873 [cs]*, July 2017.

[44] Amy Castor. An Ethereum Voting Scheme That Doesn't Give Away Your Vote. https://www.coindesk.com/voting-scheme-ethereum-doesnt-give-away-vote/, April 2017.

[45] Lin Chen, Lei Xu, Nolan Shah, Zhimin Gao, Yang Lu, and Weidong Shi. On Security Analysis of Proof-of-Elapsed-Time (PoET). In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 282–297. Springer, 2017.

[46] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On Scaling Decentralized Blockchains. In *Financial Cryptography and Data Security*, Lecture Notes in Computer Science, pages 106–125. Springer, Berlin, Heidelberg, February 2016.

[47] Michael Crosby. BlockChain Technology: Beyond Bitcoin. Technical report, 2016.

[48] Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. PBFT vs proof-of-authority: Applying the CAP theorem to permissioned blockchain. 2017.

[49] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference On*, pages 1–10. IEEE, 2013.

[50] Blockchain (Firma). Blockchain Charts: Unbestätigte Transaktionen. https://blockchain.info/de/unconfirmed-transactions.

[51] Andreas Fischer. Das IoT in der Blockchain. https://www.com-magazin.de/praxis/internet-dinge/iot-in-blockchain-1228562.html.

[52] Patrick Götze. Lufthansa Industry Solutions - Mit Blockchain zu mehr Transparenz in der Luftfahrt. https://www.lufthansa-industry-solutions.com/de-de/loesungen-produkte/luftfahrt/mit-blockchain-zu-mehr-transparenz-in-der-luftfahrt/.

[53] Vincent Gramoli. On the danger of private blockchains. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers (DCCL'16)*, 2016.

[54] Gideon Greenspan. MultiChain Private Blockchain - White Paper. Technical report, 2015.

[55] Blockchain Hub. Blockchains & Distributed Ledger Technologies.

[56] Ido Kaiser. A Decentralized Private Marketplace: DRAFT 0.1.

[57] Kari Korpela, Jukka Hallikas, and Tomi Dahlberg. Digital Supply Chain Transformation toward Blockchain Integration. January 2017.

[58] Winfried Krieger. Definition » Supply Chain Management (SCM) « | Gabler Wirtschaftslexikon. http://wirtschaftslexikon.gabler.de/Definition/supply-chain-management-scm.html.

[59] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 2014.

[60] Wenting Li, Alessandro Sforzin, Sergey Fedorov, and Ghassan O. Karame. Towards Scalable and Private Industrial Blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, BCC '17, pages 9–14, New York, NY, USA, 2017. ACM.

[61] What is Bitcoin Mining? Learn about Bitcoin mining hardware. https://www.bitcoinmining.com/bitcoin-mining-hardware/.

[62] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

[63] NetEDI. How API's are shaping B2B Data Integration- NetEDI®, September 2017.

[64] Sirajd Raval. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. O'Reilly Media, Inc., 2016.

[65] Mattias Scherer. *Performance and Scalability of Blockchain Networks and Smart Contracts*. 2017.

[66] Jagdeep Sidhu. Syscoin: A Peer-to-Peer Electronic Cash System with Blockchain-Based Services for E-Business. In *Computer Communication and Networks (ICCCN), 2017 26th International Conference On*, pages 1–6. IEEE, 2017.

[67] John Soldatos. What Does Blockchain Technology Have to Do with Enterprise Maintenance Activities? https://www.solufy.com/blog/what-does-blockchain-technology-have-to-do-with-enterprise-maintenance-activities.

[68] Yonatan Sompolinsky and Aviv Zohar. Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains. Technical Report 881, 2013.

[69] H. Sukhwani, J. M. Martínez, X. Chang, K. S. Trivedi, and A. Rindos. Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric). In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 253–255, September 2017.

[70] Melanie Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly, Beijing : Sebastopol, CA, first edition edition, 2015. OCLC: ocn898924255.

[71] Don Tapscott and Alex Tapscott. *Die Blockchain-Revolution: Wie die Technologie hinter Bitcoin nicht nur das Finanzsystem, sondern die ganze Welt verändert*. Plassen Verlag, Kulmbach, 1 edition, October 2016.

[72] Hyperledger Composer Team. Introduction | Hyperledger Composer. https://hyperledger.github.io/composer/introduction/introduction.html.

[73] Hyperledger Fabric Team. Hyperledger Whitepaper. https://docs.google.com/document/d/1Z4M_qwILLRehPbVRUsJ3OF8Iir-gqS-ZYe7W-LE9gnE/edit?usp=embed_facebook, 2016.

[74] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.

[75] Marko Vukolić. Rethinking Permissioned Blockchains. pages 3–7. ACM Press, 2017.

[76] Karl Wüst and Arthur Gervais. Do you need a Blockchain? Technical Report 375, 2017.

[77] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain Challenges and Opportunities: A Survey. *International Journal of Web and Grid Services*, December 2017.

Appendices

Anhang A

Composer BNA: Model-File

```
namespace biz.innovationcenter.maintenance

participant Company identified by companyName {
o String companyName
}

participant Machine identified by machineId {
o String machineId
o String type
o String model
--> Company owner
}

participant MaintenanceProvider identified by companyName {
o String companyName
o String experienceWith
}

asset MachineStatus identified by machineStatusId {
o String machineStatusId
o Boolean isBroken
--> Machine owner
}

asset MaintenanceContract identified by maintenanceContractId {
o String maintenanceContractId
o String maintenanceReason
o Boolean isAccepted
o Boolean isClosed
o String[] performedSteps
o String requiredLastStep optional
--> Machine owner
--> MaintenanceProvider maintenanceProvider optional
}
```

```
35
36 asset PaymentAgreement identified by paymentAgreementId {
37 o String paymentAgreementId
38 o Integer payment
39 o String maintenanceContractId
40 o Boolean isAccepted
41 }
42
43 transaction AcceptMaintenanceContract {
44 --> MaintenanceContract maintenanceContract
45 }
46
47 transaction InitMaintenance {
48 o Integer contractId
49 o String requiredLastStep optional
50 o String maintenanceReason optional
51 }
52
53 transaction CloseContract {
54 --> MaintenanceContract contract
55 }
56
57 transaction AddPerformedStep {
58 --> MaintenanceContract contract
59 o String performedStep
60 }
61
62 transaction AcceptPaymentAgreement {
63 --> PaymentAgreement paymentAgreement
64 }
65
66 transaction CreatePaymentAgreement {
67 o Integer paymentAgreementId
68 o Integer payment
69 o String maintenanceContractId
70 }
71
72 transaction SetupDemo {
73 }
74
75 event NewContractCreated {
76 }
77
78 event ContractClosed {
79 --> MaintenanceContract contract
80 --> MaintenanceProvider provider
```

```
81 }
```

Listing A.1: Composer BNA Model-File

Anhang B

Composer BNA: JavaScript-File

```
'use strict';
/**
    * Write your transction processor functions here
    */

var NAMESPACE = 'biz.innovationcenter.maintenance';
var MACHINE_STATUS_SUFFIX = '_Status';
var MAINTENANCE_CONTRACT_SUFFIX = '_Contract';
var PAYMENT_AGREEMENT_PREFIX = 'Agreement_';

/**
    * Accept the MaintenanceContract
    * @param {biz.innovationcenter.maintenance.AcceptMaintenanceContract} tx The
       ↪  transaction instance.
    * @transaction
    */
function acceptMaintenanceContract(tx) {
    var error = null;

    //Check if the contract is already accepted
    if (tx.maintenanceContract.isAccepted === false) {
        //Check if the Participant has the required experience
        if (tx.maintenanceContract.owner.type === getCurrentParticipant().
           ↪ experienceWith) {
            //Set the new Contract Data
            tx.maintenanceContract.isAccepted = true;
            tx.maintenanceContract.maintenanceProvider = getCurrentParticipant()
               ↪ ;

            // Get the contract from the asset registry
            return getAssetRegistry('biz.innovationcenter.maintenance.
               ↪ MaintenanceContract')
                .then(function (assetRegistry) {
                    // Update the contract the asset registry.
```

```
                    return assetRegistry.update(tx.maintenanceContract);
                });
        } else {
            error = 'Error: Provider does not have required experience';
        }
    } else {
        error = 'Error: Contract is already accepted';
    }

    if (error !== null) {
        console.error(error);
        throw error;
    }
}

/**
    * Initiate the maintenance. Set status to broken and create a maintenance
      ↪ contract.  Can only be executed by machines(see ACL).
    * @param {biz.innovationcenter.maintenance.InitMaintenance} tx The
      ↪ transaction instance.
    * @transaction
    */
function initMaintenance(tx) {
    //TODO: Add condition
    var factory = getFactory();

    console.log('initMaintenance');
    var machineId = getCurrentParticipant().getIdentifier()
    var machineStatusId = machineId + MACHINE_STATUS_SUFFIX;
    console.log("CURRENT MACHINE STATUS ID: " + machineStatusId);

    //Initiate the MaintenanceContract
    var contractName = machineId + MAINTENANCE_CONTRACT_SUFFIX + "_" + tx.
      ↪ contractId;
    var contract = factory.newResource(NAMESPACE, 'MaintenanceContract',
      ↪ contractName);
    if (tx.maintenanceReason != undefined) {
        contract.maintenanceReason = tx.maintenanceReason;
    }
    else {
        contract.maintenanceReason = "Unknown";
    }

    if (tx.requiredLastStep != undefined) {
        contract.requiredLastStep = tx.requiredLastStep;
    }
    else {
        contract.requiredLastStep = "Unknown";
```

```
75      }
76      contract.isAccepted = false;
77      contract.isClosed = false;
78      contract.performedSteps = [];
79
80      contract.owner = factory.newRelationship(NAMESPACE, 'Machine', machineId);
81
82      //Update the MachineStatus
83      return getAssetRegistry(NAMESPACE + '.MachineStatus')
84          .then(function (machineStatusRegistry) {
85              return machineStatusRegistry.get(machineStatusId)
86                  .then(function (machineStatus) {
87                      machineStatus.isBroken = true;
88                      return machineStatusRegistry.update(machineStatus);
89                  })
90          })
91          //Add the contract to the registry
92          .then(function () {
93              return getAssetRegistry(NAMESPACE + '.MaintenanceContract')
94                  .then(function (contractRegistry) {
95                      return contractRegistry.add(contract);
96                  })
97                  .then(function() {
98                      //Emit event
99                      var factory = getFactory();
100                     var newContractEvent = factory.newEvent(NAMESPACE, '
                            ↪ NewContractCreated');
101                     emit(newContractEvent);
102                 });
103         });
104
105 }
106
107 /**
108     * Add a performed maintenance step to the contract
109     * @param {biz.innovationcenter.maintenance.AddPerformedStep} tx The
            ↪ transaction instance.
110     * @transaction
111     */
112 function addPerformedStep(tx) {
113     //Update contract
114     if (tx.contract.isAccepted == true) {
115         if (tx.performedStep != "") {
116             tx.contract.performedSteps.push(tx.performedStep);
117             return getAssetRegistry(NAMESPACE + '.MaintenanceContract')
118                 .then(function (contractRegistry) {
119                     return contractRegistry.update(tx.contract);
120                 });
```

```
121         }
122
123     } else {
124         var error = 'Error: Contract is not accepted';
125         console.error(error);
126         throw error;
127     }
128 }
129
130 /**
131     * Close the given Contract
132     * @param {biz.innovationcenter.maintenance.CloseContract} tx The transaction
        ↪  instance.
133     * @transaction
134     */
135 function closeContract(tx) {
136     //Update contract
137     if (tx.contract.isAccepted == true) {
138         if (tx.contract.performedSteps.length != 0) {
139             if (tx.contract.requiredLastStep == "Unknown" || tx.contract.
                ↪ requiredLastStep == tx.contract.performedSteps[tx.contract.
                ↪ performedSteps.length - 1]) {
140                 tx.contract.isClosed = true;
141                 return getAssetRegistry(NAMESPACE + '.MaintenanceContract')
142                     .then(function (contractRegistry) {
143                         return contractRegistry.update(tx.contract);
144                     })
145                     .then(function () {
146                         console.log("Getting machine status registry");
147                         return getAssetRegistry(NAMESPACE + '.MachineStatus')
148                             .then(function (machineStatusRegistry) {
149                                 machineStatusRegistry.get(tx.contract.owner.
                                    ↪ getIdentifier() + MACHINE_STATUS_SUFFIX)
150                                     .then(function (machineStatus) {
151                                         //Update contract
152                                         machineStatus.isBroken = false;
153                                         return machineStatusRegistry.update(
                                            ↪ machineStatus);
154                                     })
155                                     .then(function() {
156                                         //Emit event
157                                         var factory = getFactory();
158                                         var contractClosedEvent = factory.
                                            ↪ newEvent(NAMESPACE, '
                                            ↪ ContractClosed');
159                                         contractClosedEvent.contract = tx.
                                            ↪ contract;
160                                         contractClosedEvent.provider = tx.
```

```
                                                    ↪ contract.maintenanceProvider;
161                                             emit(contractClosedEvent);
162                                         });
163                                 });
164                         });
165             } else {
166                 var error = 'Error: Required last step was not performed';
167                 console.error(error);
168                 throw error;
169             }
170         } else {
171             error = 'Error: No steps performed';
172             console.error(error);
173             throw error;
174         }
175     } else {
176         error = 'Error: Contract is not accepted';
177         console.error(error);
178         throw error;
179     }
180 }
181
182 /**
183     * Add a performed maintenance step to the contract
184     * @param {biz.innovationcenter.maintenance.CreatePaymentAgreement} tx The
            ↪ transaction instance.
185     * @transaction
186     */
187 function createPaymentAgreement(tx) {
188     var factory = getFactory();
189     var agreement = factory.newResource(NAMESPACE, 'PaymentAgreement',
            ↪ PAYMENT_AGREEMENT_PREFIX + tx.paymentAgreementId);
190     agreement.payment = tx.payment;
191     agreement.maintenanceContractId = tx.maintenanceContractId;
192     agreement.isAccepted = false;
193
194     return getAssetRegistry(NAMESPACE + '.PaymentAgreement')
195         .then(function (agreementRegistry) {
196             return agreementRegistry.add(agreement);
197         });
198 }
199
200 /**
201     * Add a performed maintenance step to the contract
202     * @param {biz.innovationcenter.maintenance.AcceptPaymentAgreement} tx The
            ↪ transaction instance.
203     * @transaction
204     */
```

```
205 function acceptPaymentAgreement(tx) {
206     tx.paymentAgreement.isAccepted = true;
207 
208     return getAssetRegistry(NAMESPACE + '.PaymentAgreement')
209         .then(function (agreementRegistry) {
210             return agreementRegistry.update(tx.paymentAgreement);
211         });
212 }
213 
214 /**
215     * Setup the demo
216     * @param {biz.innovationcenter.maintenance.SetupDemo} setupDemo - the
        ↪ SetupDemo transaction
217     * @transaction
218     */
219 function setupDemo(setupDemo) {
220     console.log('setupDemo');
221 
222     var factory = getFactory();
223 
224     //Declare the existing participants and assets
225     var companyNames = ['Twimbee', 'Eiva'];
226     var maintenanceProvNames = ['Repairr', 'Aintenance'];
227     var machineNames = ['Elevator_1', 'Server_1'];
228     var machineTypes = ['Elevator', 'Server'];
229     var machineModels = ['EL213', 'SRV4'];
230 
231     //Create the resources and add them to arrays
232     var companies = [];
233     var maintenanceProviders = [];
234     var machines = [];
235     var machineStatuses = [];
236 
237     for (var i in companyNames) {
238         var name = companyNames[i];
239         var company = factory.newResource(NAMESPACE, 'Company', name);
240         companies.push(company);
241     }
242 
243     for (var i in maintenanceProvNames) {
244         var name = maintenanceProvNames[i];
245         var provider = factory.newResource(NAMESPACE, 'MaintenanceProvider',
            ↪ name);
246         provider.experienceWith = machineTypes[i];
247         maintenanceProviders.push(provider);
248     }
249 
250     var i = 0;
```

```
251     for (var i in machineNames) {
252         var name = machineNames[i]
253         var machine = factory.newResource(NAMESPACE, 'Machine', name);
254         machine.type = machineTypes[i];
255         machine.model = machineModels[i];
256         machine.owner = factory.newRelationship(NAMESPACE, 'Company', companies[
                ↪ i].companyName)
257         machines.push(machine);
258         i++;
259     }
260 
261     var i = 0;
262     for (var i in machines) {
263         var name = machineNames[i] + MACHINE_STATUS_SUFFIX;
264         console.log("MACHINE NAME: " + name);
265         var machineStatus = factory.newResource(NAMESPACE, 'MachineStatus', name
                ↪ );
266         machineStatus.isBroken = false;
267         machineStatus.owner = factory.newRelationship(NAMESPACE, 'Machine',
                ↪ machineNames[i]);
268         machineStatuses.push(machineStatus);
269         i++;
270     }
271 
272     //Update the registries
273     return getParticipantRegistry(NAMESPACE + '.Company')
274         .then(function (companyRegistry) {
275             return companyRegistry.addAll(companies);
276         })
277         .then(function () {
278             return getParticipantRegistry(NAMESPACE + '.MaintenanceProvider');
279         })
280         .then(function (machineProvRegistry) {
281             return machineProvRegistry.addAll(maintenanceProviders);
282         })
283         .then(function () {
284             return getParticipantRegistry(NAMESPACE + '.Machine');
285         })
286         .then(function (machineRegistry) {
287             return machineRegistry.addAll(machines);
288         })
289         .then(function () {
290             return getAssetRegistry(NAMESPACE + '.MachineStatus');
291         })
292         .then(function (machineStatusRegistry) {
293             return machineStatusRegistry.addAll(machineStatuses);
294         });
```

```
295 }
```

Listing B.1: Composer BNA JavaScript File

Anhang C

Composer BNA: ACL-Rules

```
rule NetworkAdminUser {
    description: "Grant business network administrators full access to user
        ↪ resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "**"
    action: ALLOW
}

rule NetworkAdminSystem {
    description: "Grant business network administrators full access to system
        ↪ resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "org.hyperledger.composer.system.**"
    action: ALLOW
}

rule CompanyCanCreateMachineParticipants {
    description: "Allow companies to create machine-participants"
    participant(p): "biz.innovationcenter.maintenance.Company"
    operation: CREATE
    resource(m): "org.hyperledger.composer.system.Participant"
    condition: (p.getIdentifier() == m.owner.getIdentifier())
    action: ALLOW
}

rule CompanyCanDeleteMachineParticipants {
    description: "Allow companies to delete own machine-participants"
    participant(p): "biz.innovationcenter.maintenance.Company"
    operation: DELETE
    resource(m): "org.hyperledger.composer.system.Participant"
    condition: (p.getIdentifier() == m.owner.getIdentifier())
    action: ALLOW
```

```
}

rule MachineCanCreateStatusAsset {
    description: "Machine can create own Status-Asset"
    participant(p): "biz.innovationcenter.maintenance.Machine"
    operation: CREATE
    resource(m): "biz.innovationcenter.maintenance.MachineStatus"
    condition: ((p.getIdentifier() == m.owner.getIdentifier()) && (m.isBroken ==
        ↪ false))
    action: ALLOW
}

rule MachineCanUpdateStatusAsset {
    description: "Machine can update own status-asset"
    participant: "biz.innovationcenter.maintenance.Machine"
    operation: UPDATE
    resource: "biz.innovationcenter.maintenance.MachineStatus"
    action: ALLOW
}

rule MachineCanCreateAndUpdateMaintenanceContract {
    description: "Machine can create and update MaintenanceContract-Asset"
    participant(p): "biz.innovationcenter.maintenance.Machine"
    operation: CREATE, UPDATE
    resource(m): "biz.innovationcenter.maintenance.MaintenanceContract"
    condition: (p.getIdentifier() == m.owner.getIdentifier())
    action: ALLOW
}

rule MaintenanceProviderCanUpdateMaintenanceContract {
    description: "Maintenance provider can update his accepted
        ↪ maintenanceContract"
    participant(p): "biz.innovationcenter.maintenance.MaintenanceProvider"
    operation: UPDATE
    resource(r): "biz.innovationcenter.maintenance.MaintenanceContract"
    condition: ((r.maintenanceProvider == null) || (r.maintenanceProvider.
        ↪ getIdentifier() == p.getIdentifier()))
    action: ALLOW
}

rule CompanyCanCreatePaymentAgreement {
    description: "Company can create PaymentAgreement"
    participant: "biz.innovationcenter.maintenance.Company"
    operation: CREATE
    resource: "biz.innovationcenter.maintenance.PaymentAgreement"
    action: ALLOW
}

```

```
78 rule MaintenanceProviderCanUpdatePaymentAgreement {
79     description: "Maintenance provider can update PaymentAgreement"
80     participant: "biz.innovationcenter.maintenance.MaintenanceProvider"
81     operation: UPDATE
82     resource: "biz.innovationcenter.maintenance.PaymentAgreement"
83     action: ALLOW
84 }
85
86 rule MaintenanceProviderCanExecuteAcceptTransaction {
87     description: "Maintenance provider can execute AcceptMaintenanceContract"
88     participant(p): "biz.innovationcenter.maintenance.MaintenanceProvider"
89     operation: CREATE
90     resource(r): "biz.innovationcenter.maintenance.AcceptMaintenanceContract"
91     condition: (r.maintenanceContract.maintenanceProvider == null)
92     action: ALLOW
93 }
94
95 rule MachineCanExecuteCloseContractTransaction {
96     description: "Maintenance provider can execute CloseContract"
97     participant: "biz.innovationcenter.maintenance.Machine"
98     operation: CREATE
99     resource: "biz.innovationcenter.maintenance.CloseContract"
100    action: ALLOW
101 }
102
103 rule MachineCanExecuteInitMaintenanceTransaction {
104    description: "Machine can execute initMaintenance transaction"
105    participant: "biz.innovationcenter.maintenance.Machine"
106    operation: CREATE
107    resource: "biz.innovationcenter.maintenance.InitMaintenance"
108    action: ALLOW
109 }
110
111 rule ProviderCanExecuteAddPerformedStepTransaction {
112    description: "Maintenance provider can add performed maintenance steps to
       ↪ his accepted contract"
113    participant(p): "biz.innovationcenter.maintenance.MaintenanceProvider"
114    operation: CREATE
115    resource(r): "biz.innovationcenter.maintenance.AddPerformedStep"
116    condition: (r.contract.maintenanceProvider.getIdentifier() == p.
       ↪ getIdentifier())
117    action: ALLOW
118 }
119
120 rule CompanyCanExecuteCreatePaymentAgreementTransaction {
121    description: "Company can execute CreatePaymentAgreementTransaction"
122    participant: "biz.innovationcenter.maintenance.Company"
123    operation: CREATE
```

```
124     resource: "biz.innovationcenter.maintenance.CreatePaymentAgreement"
125     action: ALLOW
126 }
127
128 rule ProviderCanExecuteAcceptPaymentAgreementTransaction {
129     description: "Maintenance provider can execute AcceptPaymentAgreement"
130     participant: "biz.innovationcenter.maintenance.MaintenanceProvider"
131     operation: CREATE
132     resource: "biz.innovationcenter.maintenance.AcceptPaymentAgreement"
133     action: ALLOW
134 }
135
136  rule denyParticipantEditing {
137     description: "No participant is allowed to edit participants"
138     participant: "ANY"
139     operation: CREATE, UPDATE, DELETE
140     resource: "org.hyperledger.composer.system.Participant"
141     action: DENY
142 }
143
144 rule denyAssetEditingAndTransactionExecution {
145     description: "No participant is allowed to edit assets or execute
        ↪ transactions"
146     participant: "ANY"
147     operation: CREATE, UPDATE, DELETE
148     resource: "biz.innovationcenter.maintenance.*"
149     action: DENY
150 }
151
152 rule EverybodyCanReadEverything {
153     description: "Allow all participants read access to all resources"
154     participant: "ANY"
155     operation: READ
156     resource: "biz.innovationcenter.maintenance.*"
157     action: ALLOW
158 }
159
160
161 rule SystemACL {
162   description:  "System ACL to permit all access"
163   participant: "org.hyperledger.composer.system.Participant"
164   operation: ALL
165   resource: "org.hyperledger.composer.system.**"
166   action: ALLOW
167 }
```

Listing C.1: Composer BNA ACL-Rules