HOCHSCHULE
MITTWEIDA
University of Applied Sciences

MASTER-THESIS

Herr
Erik Neumann

Vergleich der Leistung von Konsensalgorithmen in privaten Blockchain-Netzwerken für industrielle Anwendungen

2021

HOCHSCHULE
MITTWEIDA
University of Applied Sciences

MASTER-THESIS

Mr.
Erik Neumann

Comparison of consensus algorithm performance in private blockchain networks for industrial applications

2021

Fakultät **Angewandte Computer- und Biowissenschaften**

MASTER-THESIS

Vergleich der Leistung von Konsensalgorithmen in privaten Blockchain-Netzwerken für industrielle Anwendungen

Autor:
Erik Neumann

Studiengang:
Blockchain & Distributed Ledger Technologies

Seminargruppe:
BC19w1-M

Erstprüfer:
Prof. Dr.-Ing. Andreas Ittner

Zweitprüfer:
Dipl.-Volkswirt Mario Oettler

Mittweida, Oktober 2021

Bibliografische Angaben

Neumann, Erik: Vergleich der Leistung von Konsensalgorithmen in privaten Blockchain-Netzwerken für industrielle Anwendungen, 65 Seiten, 30 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Master-Thesis, 2021

Referat

Private Blockchain-Netzwerke können von Unternehmen für die Integritätssicherung von Produktionsdaten verwendet werden. Die Hochschule Mittweide entwickelt im Rahmen des Forschungsprojektes safe-UR-chain ein derartiges Blockchain-Netzwerk. Teil dieser Entwicklung ist die Auswahl und Optimierung eines Konsensverfahrens für das Netzwerk. Für diese Aufgabe fehlt es momentan an einem System, das den Vergleich von Konsensverfahren anhand ihrer Leistungsmetriken ermöglicht. Diese Arbeit befasst sich mit der Entwicklung eines derartigen Systems sowie der Implementierung dreier Konsensverfahren, die anhand ihrer vom System erfassten Leistungsmetriken verglichen werden.

I. Inhaltsverzeichnis

Inhaltsverzeichnis I
Abbildungsverzeichnis II
Tabellenverzeichnis III
1 Einführung 1
2 Technische Übersicht 3
2.1 Projektvorstellung 3
2.2 Node-Typen 4
2.3 Kryptographie 7
2.4 Netzwerk 8
2.5 Transaktionen 12
2.6 Blöcke 13
2.7 Interne Abläufe 15
2.8 Datenherkunft 16
2.9 Konfiguration 17
3 Konsensverfahren 19
3.1 Grundlagen 19
3.2 Vergleichbarkeit von Konsensverfahren 21
3.3 Generisches Interface 25
3.4 Proof of Work 27
3.5 Proof of Elapsed Time 34
3.6 Proof of Network Signatures 37
4 Vergleichssystem 41
4.1 Erfassung der Metriken 41
4.2 Simulationsumgebung 50
4.3 Definition von Testfällen 50
5 Versuchsdurchführung 53
5.1 Ergebnisse 53

5.2 Auswertung 57

5.3 Betrachtung 58

6 Fazit und Ausblick 61

Literaturverzeichnis 63

II. Abbildungsverzeichnis

2.1 Unternehmensnetzwerk mit lokalen Blockchains innerhalb der Unternehmen, die über einen HTTPs-Broker verknüpft werden 4

2.2 Node-Typen 5

2.3 Beispielhafte Netzwerktopologie 6

2.4 Node- und Netzwerk-Schlüssel 7

2.5 Nodes vernetzen sich 10

2.6 Gegenseitige Rückbestätigungen 11

2.7 Zusammenfassung von Daten in einen Block 14

2.8 Baumstruktur der Blöcke mit der “korrekten” Kette in Blau, einer verworfenen Spaltung in Grau und nicht angehängten Blöcke in Rot. 15

2.9 Blockproduktion 16

3.1 Weg von Daten in die Blockchain 20

3.2 Wertebereich für ein einzelnes Byte nach dem u8 Wert der Schwierigkeit 28

3.3 Überprüfung des Block Hashes bei Proof of Work 29

3.4 Scoringfunktion in Proof of Elapsed Time ($y = 10^3 \cdot 1,3^{-\frac{x^2}{10^5}}$) 36

5.1 PoW: Transaktionsdurchsatz in Transaktionen pro Sekunde 54

5.2 PoW: Latenz in Millisekunden 54

5.3 PoW: Abweichung vom theoretisch maximalen Durchsatz in Prozent 54

5.4 PoW: Blockzeiten in Millisekunden 54

5.5 PoET: Transaktionsdurchsatz in Transaktionen pro Sekunde 55

5.6 PoET: Latenz in Millisekunden 55

5.7 PoET: Abweichung vom theoretisch maximalen Durchsatz in Prozent 55

5.8 PoET: Blockzeiten in Millisekunden 55

5.9 PoNS: Transaktionsdurchsatz in Transaktionen pro Sekunde 56

5.10 PoNS: Latenz in Millisekunden 56

5.11 PoNS: Abweichung vom theoretisch maximalen Durchsatz in Prozent 56

5.12 PoNS: Blockzeiten in Millisekunden 56

5.13 Wiederholbarkeit von PoW-Tests 59
5.14 Wiederholbarkeit von PoET-Tests 59
5.15 Wiederholbarkeit von PoNS-Tests 59
5.16 Blockzeiten im Test mit maximal 10.000 Transaktionen pro Block 60
5.17 Blockzeiten im Test mit maximal 1.500 Transaktionen pro Block 60

III. Tabellenverzeichnis

3.1 Beispiele für Schwierigkeiten als Tupel (links) und in ihrer numerischen Form (rechts) . 32

4.1 Konfigurationen der einzelnen Node Typen 42

4.2 Testfälle in der Versuchsreihe 51

1 Einführung

Moderne Produktionsnetzwerke sind von einer immer stärkeren Digitalisierung und Vernetzung geprägt. Insbesondere das Tracking innerhalb von Lieferketten findet zunehmend auf digitaler Ebene statt [1]. Mithilfe der erhobenen Daten können Produktionsprozesse optimiert und auch automatisiert werden. Dies trifft nicht nur für Prozesse innerhalb einzelner Unternehmen, sondern auch für unternehmensübergreifende Kooperationen zu. So können Unternehmen ihre Arbeit genauer aufeinander abstimmen und ggf. auftretende Fehler gemeinsam zurückverfolgen und beheben.

Mit dieser zunehmenden Vernetzung von Unternehmen wächst allerdings auch die Gefahr durch Cyberkriminalität. Dazu zählen beispielsweise das Abfangen von Daten (Spionage) und die Datenmanipulation. Besonders die Manipulation von Daten kann Unternehmen stark beeinträchtigen. Zulieferer könnten beispielsweise ihre Produktionsdaten so verändern, dass die Güte ihrer Produkte höher erscheint als sie tatsächlich ist. Auch Akteure außerhalb des Wertschöpfungsnetzwerkes können Prozesse stören, indem sie z.B. Daten verändern, auf die Maschinen für ihren normalen Betrieb angewiesen sind.

Um diesem Problem entgegenzuwirken, müssen Daten so gespeichert werden, dass ihre Integrität gesichert ist. Dies gilt sowohl für die direkte Speicherung der Daten innerhalb eines Unternehmens, als auch für die Übertragung der Daten zwischen Unternehmen. Aus einer Befragung von 150 Entscheidungsträgern in Supply Chain Positionen [1] geht der Bedarf dafür hervor. So werden die Integrität von Daten (“Data integrity”) und der Datenaustausch (“Data sharing/exchange across your supply chain”) als die zwei größten Herausforderungen in diesem Bereich angegeben.

Die Blockchain-Technologie bietet Lösungen für diese Probleme, insbesondere die Sicherung der Datenintegrität, sowie eine hohe Verfügbarkeit bei Ausfällen von Teilen des Netzwerkes. Diese Eigenschaften wurden bereits in öffentlichen Blockchain-Netzwerken, wie der Bitcoin-Blockchain, nachgewiesen und sind in diesem bereits seit mehr als einem Jahrzehnt durchgehend verfügbar. Der Ursprung dieser Eigenschaften liegt in der Funktionsweise der Blockchain als Datenstruktur und ihrer Verteilung innerhalb des Blockchain-Netzwerkes. Daten werden dabei in sog. Blöcken zusammengefasst, die nach bestimmten Regeln gültig sein müssen. Diese Blöcke werden zu einer Blockkette zusammengeführt, indem jeder Block auf seinen Vorgänger verweist. Dabei ist der Verweis auf einen Vorgänger gleichzeitig der Verweis auf dessen Vorgänger. Diese Eigenschaft besteht für alle Blöcke. Somit verweist jeder neue Block auf die gesamte Historie der Blockchain bis zu seinem Erscheinen, wodurch Daten in älteren Blöcken unveränderbar werden. Durch diesen Vorgang wird die Integrität der Daten gesichert. Zusätzlich wird die Blockchain in einem Netzwerk verteilt, das fortwährend neue Blöcke

erzeugt und diese anhängt. Durch die Speicherung der Blockchain bei allen Teilnehmern, bleiben die Daten verfügbar, selbst wenn einige von ihnen ausfallen.

Durch die Nutzung eines Blockchain-Netzwerkes können Daten innerhalb eines Unternehmens gegen Manipulation geschützt werden. Dieser Schutz kann weiter verstärkt werden, indem die Blockchain nicht nur in einem Unternehmen, sondern über mehrere Unternehmen verteilt wird. Mit einem derartigen Netzwerk können Daten nicht nur sicher gespeichert, sondern auch beweisbar ausgetauscht werden. So kann mithilfe einer Blockchain z.B. die Existenz von Daten zu einem gewissen Zeitpunkt zweifelsfrei nachgewiesen werden. Damit können Zulieferer gegenüber einem Inspektor/Auditor ihre Produktionsdaten in beweisbarer, chronologischer Ordnung offenlegen.

Einige Unternehmen haben bereits begonnen, ihre Wertschöpfungsketten mit der Blockchain-Technologie abzusichern. Walmart verfolgt beispielsweise die Produktion von Schweinefleisch und Mangos über eine Blockchain-Lösung [2] und das von A.P. Moller, Maersk und IBM entwickelte TradeLens [3] [4] macht die Verschiffung von Gütern auf Blockchain-Basis nachvollziehbar. Immer mehr Unternehmen (darunter BMW, Amazon und General Electric) setzen auf industrielle Blockchain-Lösungen, teilweise zur Schaffung von Verantwortlichkeit in ihrem Wertschöpfungsnetzwerk, aber auch als neue Einkommensquelle mit Servicelösungen [5].

Auch die Hochschule Mittweida beteiligt sich im Rahmen des Projektes safe-UR-chain aktiv an der Entwicklung eines solchen privaten Blockchain-Netzwerkes. Zum Zeitpunkt des Schreibens dieser Arbeit soll ein geeignetes Konsensverfahren für dieses Netzwerk ausgewählt werden. Dafür fehlt es jedoch an einer praktikablen Möglichkeit zum Vergleich von Konsensverfahren innerhalb dieses Netzwerkes. Um eine derartige Möglichkeit zu schaffen, beschäftigt sich diese Arbeit mit der Entwicklung eines Systems zum Vergleich von Konsensverfahren im safe-UR-chain Projekt, sowie einem ersten Vergleich dreier Konsensverfahren.

Dafür wird im zweiten Kapitel der grundlegende Aufbau des safe-UR-chain Blockchain-Netzwerkes erläutert. Das dritte Kapitel befasst sich mit der Funktionsweise von Konsensverfahren und der Implementierung solcher Verfahren im Projekt. Im vierten Kapitel werden die Implementierung des Vergleichssystems beschrieben und Testfälle für dieses System definiert. Anschließend beschäftigt sich das fünfte Kapitel mit der Durchführung erster Versuche und betrachtet deren Ergebnisse.

2 Technische Übersicht

Als Grundlage für die Forschung innerhalb dieser Arbeit dient ein von der Hochschule Mittweida entwickeltes Blockchain-Netzwerk. Dieses Kapitel beschreibt den Einsatzbereich dieses Netzwerkes und gibt eine Übersicht über dessen wesentlichen Softwaremodule und deren Zusammenspiel.

2.1 Projektvorstellung

Ziel des Forschungsprojektes “safe-UR-chain” [6] ist die manipulationssichere Speicherung von produktionsrelevanten Daten (Produktionsdaten, Daten aus Qualitätskontrollen, Maschinenkonfigurationen, etc.). Um zu erforschen, wie ein derartiges System umgesetzt werden kann, arbeitet die Hochschule Mittweida mit Vitesco Technologies, dem Fraunhofer IWU, dem Maschinenbauer Xenon und der Consulting-Firma Capgemini zusammen. Dieses Team aus Unternehmen und Forschungseinrichtungen hat die Aufgabe, ein industrielles Datenspeichernetzwerk auf Blockchain-Basis prototypisch umzusetzen. Unternehmen können dieses System dann beispielsweise nutzen, um im Garantiefall zurückzuverfolgen, ob und in welchem Produktionsschritt möglicherweise ein Fehler aufgetreten ist. Jedes Unternehmen bzw. jeder Partner soll ein lokales Blockchain-Netzwerk betreiben. Diese lokalen Netzwerke können, je nach Teilnehmer, unterschiedlich groß sein. So kann eine Fabrik jegliche Maschinen in ihr Netzwerk einbinden oder ein Auditor nur einen einzelnen Knoten betreiben.

Die unternehmensinternen Netzwerke arbeiten dabei weitgehend autonom, synchronisieren aber regelmäßig Block-Hashes, die in die Blockchains der anderen Unternehmen aufgenommen werden. Dieser Austausch findet über einen HTTPs-Broker statt, der es ermöglicht, Nachrichten zwischen sonst getrennten Unternehmensnetzwerken zu verschicken (s. Abb. 2.1). Damit werden Rückbestätigungen zwischen den Unternehmen etabliert, nach denen kein Teilnehmer seine lokale Blockchain bis zu diesem Punkt verändern kann. Zusätzlich können die Netzwerke in Form von Datenabfragen untereinander kommunizieren. So können relevante Daten in anderen Netzwerken eingesehen und mithilfe der zuvor synchronisierten Block-Hashes validiert werden. Durch diesen Mechanismus muss nur die minimale Menge an Daten das lokale Blockchain-Netzwerk verlassen. Dieser Ansatz soll helfen, den strengen IT-Sicherheitsrichtlinien der Teilnehmer gerecht zu werden. Die Hochschule Mittweida befasst sich innerhalb dieses Projektes mit der Entwicklung des privaten Blockchain-Netzwerkes, bei dieser kommt die Programmiersprache Rust[1] zum Einsatz.

[1] https://www.rust-lang.org

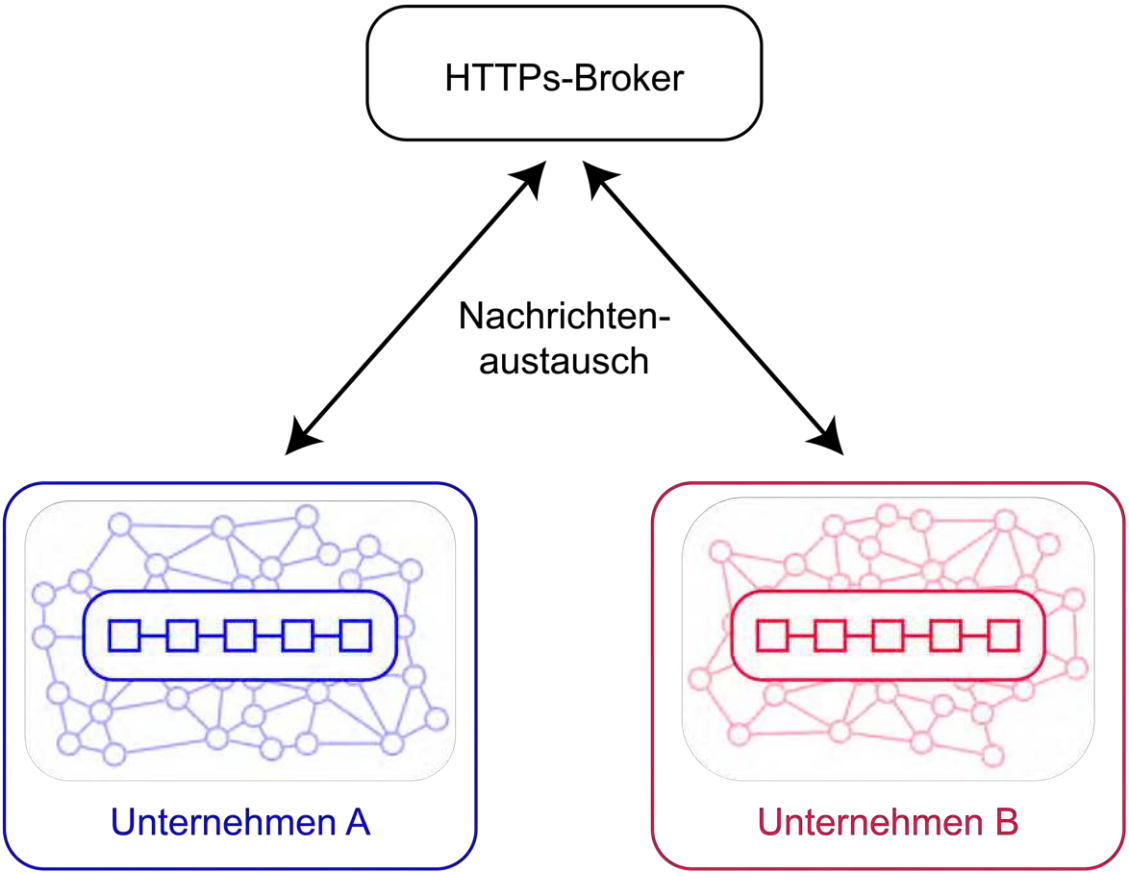


Abbildung 2.1: Unternehmensnetzwerk mit lokalen Blockchains innerhalb der Unternehmen, die über einen HTTPs-Broker verknüpft werden

2.2 Node-Typen

Innerhalb der unternehmensinternen Netzwerke arbeiten vier Arten von Nodes. Diese wurden insbesondere für den Einsatz in Fabriken konzipiert. In dieser Umgebung sind die Computer, auf denen die Blockchain-Software läuft, nicht notwendigerweise sehr leistungsfähig, was den Betrieb einer kompletten Blockchain-Node erschweren kann. Aus diesem Grund wurde die Software so entwickelt, dass sich Funktionen, die besonders viel Rechenleistung oder Speicherplatz benötigen, abschalten lassen.

Jede Node hat ein Blockchain-Interface (“**BCI**”), mit dem die Daten der lokalen Blockchain auf der Node zugänglich gemacht werden. Außerdem bietet dieses Modul grundlegende Funktionen, wie die Validierung von Blöcken und das Erstellen von signierten Transaktionen ab. Eine weitere Basisfunktion jeder Node ist die Interaktion mit dem **Netzwerk**. Nodes, die nur über diese beiden Module verfügen, werden als **Thin-Nodes** bezeichnet und sind insbesondere für den Einsatz direkt an Maschinen konzipiert. Ihre Hauptaufgabe besteht in der Einspeisung von Daten in das Netzwerk.

Zusätzlich zu den Grundfunktionen können Nodes als **Validator**en auftreten. Diese Nodes sammeln Transaktionen aus dem Netzwerk und fassen sie nach bestimmten Regeln zu Blöcken zusammen. Diese Aufgabe kann eine hohe Rechenleistung erfordern. Validatoren müssen die Transaktionsdaten jedoch nicht dauerhaft speichern, denn für eine dauerhafte Speicherung der gesamten Blockchain kann viel Speicherplatz notwendig sein. Nodes mit diesem Speicherplatz können als **Archiv** auftreten. Sie speichern alle neuen Blöcke mitsamt aller darin enthaltenen Daten und machen diese per API zugänglich. Letztendlich können Nodes auch alle der hier gelisteten Aufgaben übernehmen. Diese Nodes werden als **Full-Nodes** bezeichnet. Abbildung 2.2 stellt die einzelnen Node-Typen mit ihren jeweils aktiven Modulen dar.

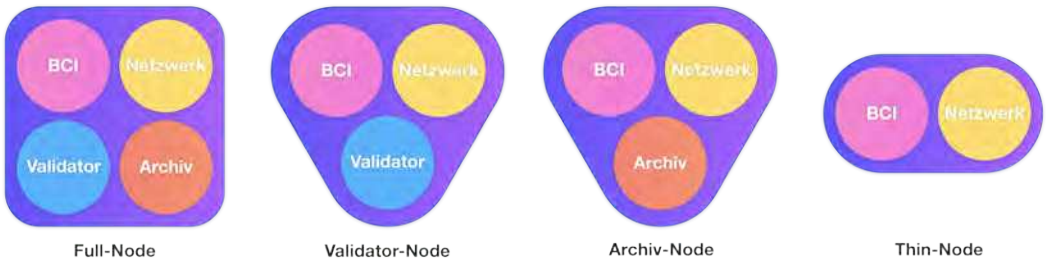


Abbildung 2.2: Node-Typen

Zusätzlich zu diesen Funktionen können Nodes eine Reihe an weiteren Aufgaben übernehmen, dazu zählen:

- Kommunikation mit dem HTTPs-Broker
- Datenaufnahme aus dem Dateisystem
- Datenaufnahme über eine direkte Schnittstelle zur Maschine
- Serielle Kommunikation zu Funk-Hardware, um die Blockchain für mobile Geräte offenzulegen
- Betrieb einer API für die Anbindung an ein Web-Interface

Mit diesen Node-Typen kann das jeweilige unternehmensinterne Netzwerk auf die Anforderungen und Ressourcen des Unternehmens individuell angepasst werden. Die nachfolgende Abbildung 2.3 zeigt eine beispielhafte Netzwerk-Topologie für eine Fabrik, bei der die Maschinen über Thin-Nodes ihre Daten in das Netzwerk einspeisen können. Die Daten werden im Netzwerk verteilt und von den Validatoren und der Full-Node zu Blöcken zusammengefasst. Für die Langzeit-Speicherung der Daten steht ein Cluster mit Archiv-Nodes bereit.

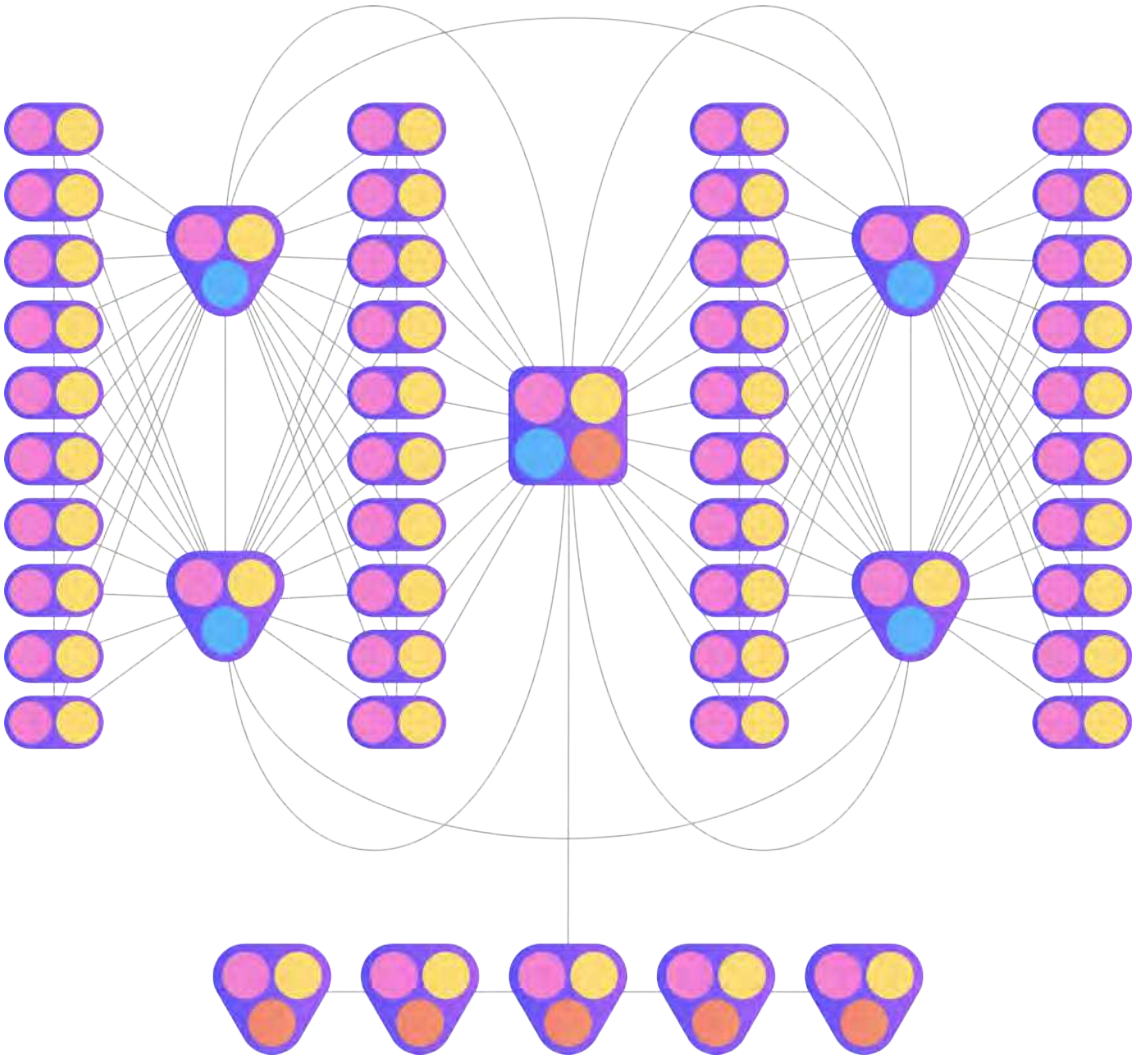


Abbildung 2.3: Beispielhafte Netzwerktopologie

2.3 Kryptographie

Identitäten/Schlüssel

Einer der Sicherheitsaspekte bei safe-UR-chain besteht darin, dass Nodes dem Netzwerk nur mit Erlaubnis eines Administrators hinzugefügt werden können. Darum brauchen die Nodes eindeutige Identitäten. Diese werden über digitale Signaturen abgebildet. Jede Nachricht, die im Netzwerk versendet wird, trägt deshalb die Identifikation der sendenden Node, sowie eine Signatur, die für jede Nachricht einzigartig ist. Identifikationen setzen sich aus der ID des Netzwerkes und einer eindeutigen Nummer zusammen (z.B.: “hsmw-2”).

Nodes, die signierte Nachrichten empfangen, müssen zu deren Verifizierung den öffentlichen Schlüssel der sendenden Node kennen. Um keine Liste mit öffentlichen Schlüsseln auf allen Nodes pflegen zu müssen, kommt ein hierarchisch-deterministisches Verfahren [7] zur Erzeugung der öffentlichen Schlüssel zum Einsatz. Alle Nodes kennen die erweiterten, öffentlichen Schlüssel aller Netzwerke. Gemeinsam mit der Nummer der jeweiligen sendenden Node kann der öffentliche Schlüssel aus dem erweiterten Schlüssel erzeugt werden. Die privaten Schlüssel können vom Administrator über den erweiterten, privaten Schlüssel des Netzwerkes erzeugt und auf neue Nodes verteilt werden. Der Zusammenhang zwischen den Schlüsseln ist in Abbildung 2.4 dargestellt. Nodes erhalten, je nach ihrem Index (z.B. `2` bei `hsmw-2`), einen privaten Schlüssel.

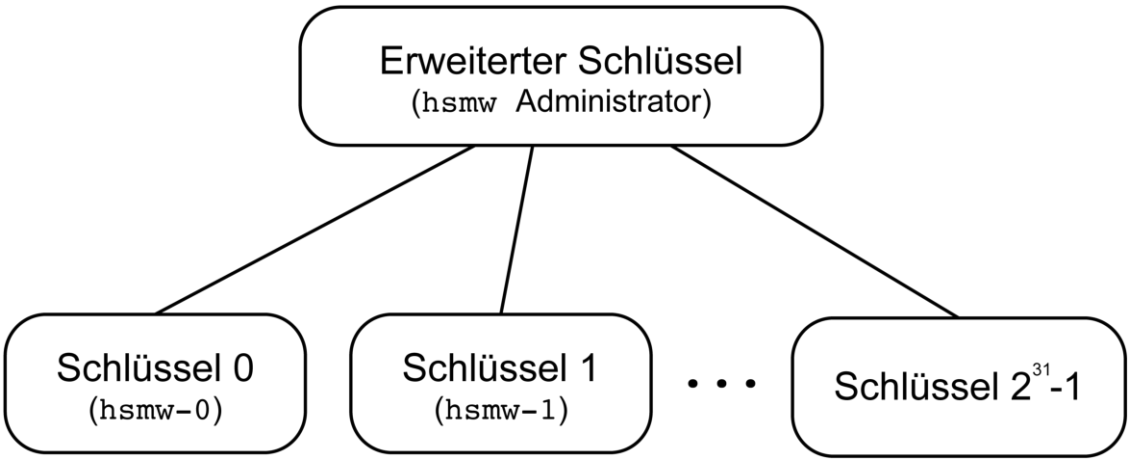


Abbildung 2.4: Node- und Netzwerk-Schlüssel

Signierte Daten werden in einer gesonderten Datenstruktur verwaltet, die neben den zum Verifizieren notwendigen Daten auch die Daten an sich in serialisierter Form beinhaltet (s. Listing 2.1).

```rust
pub struct SignedData {
    pub network_id: String,
    pub node_index: u32,
    pub data_type: SignedDataType,
    pub data: Vec<u8>,
    pub digest: Vec<u8>,
    pub signature: String,
}
```

Listing 2.1: Datenstruktur für signierte Daten

Verschlüsselung

Um eine verschlüsselte Kommunikation zwischen Nodes oder ganzen Netzwerken zu ermöglichen, kommt ein ECIES (AES GCM 256 + ECC secp256k1) zum Einsatz. Dieses verwendet asymmetrische und symmetrische Kryptographie gemeinsam, um längere Nachrichten sicher zu übertragen. In diesem Verfahren wird die Nachricht mit einem zufällig generierten AES-Schlüssel verschlüsselt. Die Nachricht kann so beliebig lang sein. Der Schlüssel selbst wird auf asymmetrische Weise verschlüsselt. Dadurch kann er nur von der empfangenden Node mit dem passenden privaten Schlüssel entschlüsselt werden [8]. Diese Node kann dann auch die Nachricht entschlüsseln. Der Vorteil dieses Verfahrens liegt darin, dass zwei Nodes keine Schlüssel austauschen müssen und ein zustandsloses Kommunikationsprotokoll verwendet werden kann.

Um die Kommunikation zwischen Netzwerken abzusichern, können Nachrichten auch so verschlüsselt werden, dass jede Node in einem bestimmten Netzwerk diese entschlüsseln kann. Dafür wird der “letzte” Schlüssel in der Ebene der Node-Schlüssel verwendet. Dieser Schlüssel trägt den Index $2^{31} - 1$ (s. Abb. 2.4) und ist allen Nodes in den jeweiligen Netzwerken bekannt. Um eine Nachricht beispielsweise für das gesamte `hsmw`-Netzwerk zu verschlüsseln, wird diese für die Node mit dem Index $2^{31} - 1$ verschlüsselt. Da alle Nodes im `hsmw`-Netzwerk Zugriff auf den passenden privaten Schlüssel haben, kann die Nachricht auch von all diesen Nodes entschlüsselt werden.

2.4 Netzwerk

Um die Funktion der Blockchain zu gewährleisten, muss ein robustes Netzwerk zwischen den Nodes bestehen. Dieses Netzwerk ist im Projekt so konzipiert, dass es sich selbst aufbaut, beim Ausfall von Teilnehmern repariert und gegen Eclipse-Attacken [9] geschützt ist. Zusätzlich werden alle Nachrichten im Netzwerk signiert und ggf. verschlüsselt. Die Grundlage des Netzwerkes bieten die eindeutigen IDs der Teilnehmer (“Peers”). Mit diesen IDs und den in der Sektion Kryptografie (2.3) beschriebenen Signaturen können verifizierbar aus dem Netzwerk stammende Nachrichten erzeugt wer-

den. Listing 2.2 zeigt die hierfür verwendeten Datenstrukturen. Diese Nachrichten werden im Netzwerk entweder auf direktem Weg zwischen Nodes verschickt oder über ein Flooding-Protokoll im gesamten Netzwerk verteilt.

```
pub struct PeerId {
    pub network: String,
    pub index: u32,
}

pub struct NetworkMessage {
    pub recipient: NetworkMessageRecipient,
    pub encrypted_for: Option<PeerId>,
    pub msg: SignedData,
    pub ttl: u8,
    pub from: Option<SocketAddr>,
    pub timestamp: u128,
}
```

Listing 2.2: Datenstrukturen im Netzwerkmodul

Bootstrapping

Um Nachrichten im Netzwerk zu verteilen, muss dieses zuerst aufgebaut werden. Dafür muss mindestens eine Node laufen und den Neuzugängen bekannt sein. Diese Nodes werden als “Seed-Nodes” bezeichnet. Diese unterscheiden sich technisch nicht von allen anderen Nodes. Sie werden nur zu Seed-Nodes, indem sie anderen Nodes per Konfiguration bekannt gemacht werden.

Nodes versuchen immer, so viele andere Nodes wie möglich zu kennen. Dafür senden sie im Normalbetrieb regelmäßige `PeerListRequest` Nachrichten an die anderen Nodes im Netzwerk. Diese antworten mit ihren jeweiligen Peer-Listen. Sollte eine Node jedoch noch keine anderen Nodes kennen (z.B. direkt nach ihrem Start), versucht sie diese Aufforderung an die in ihrer Konfiguration aufgeführten Seed-Nodes zu versenden. Die Seed-Node antwortet mit ihrer lokalen Liste und nimmt die neue Node in diese auf, wodurch die nächsten Neuzugänge auch von ihr erfahren.

Dieses Verfahren sorgt dafür, dass alle Nodes so viele andere Netzwerkteilnehmer wie möglich kennen. Es funktioniert in Netzwerken, die sich zum erstem Mal aufbauen, sowie in bereits bestehenden Netzwerken. Abbildung 2.5 veranschaulicht diese Funktionsweise in vier Phasen.

1. Es bestehen keine Verbindungen; die Seed-Node ist in der Mitte
2. Die Nodes bauen erste, unidirektionale Verbindungen zu Seed-Node auf
3. Die Seed-Node stellt Verbindungen zu den anderen Nodes her und teilt deren IP-Adressen
4. Die Nodes vernetzen sich untereinander

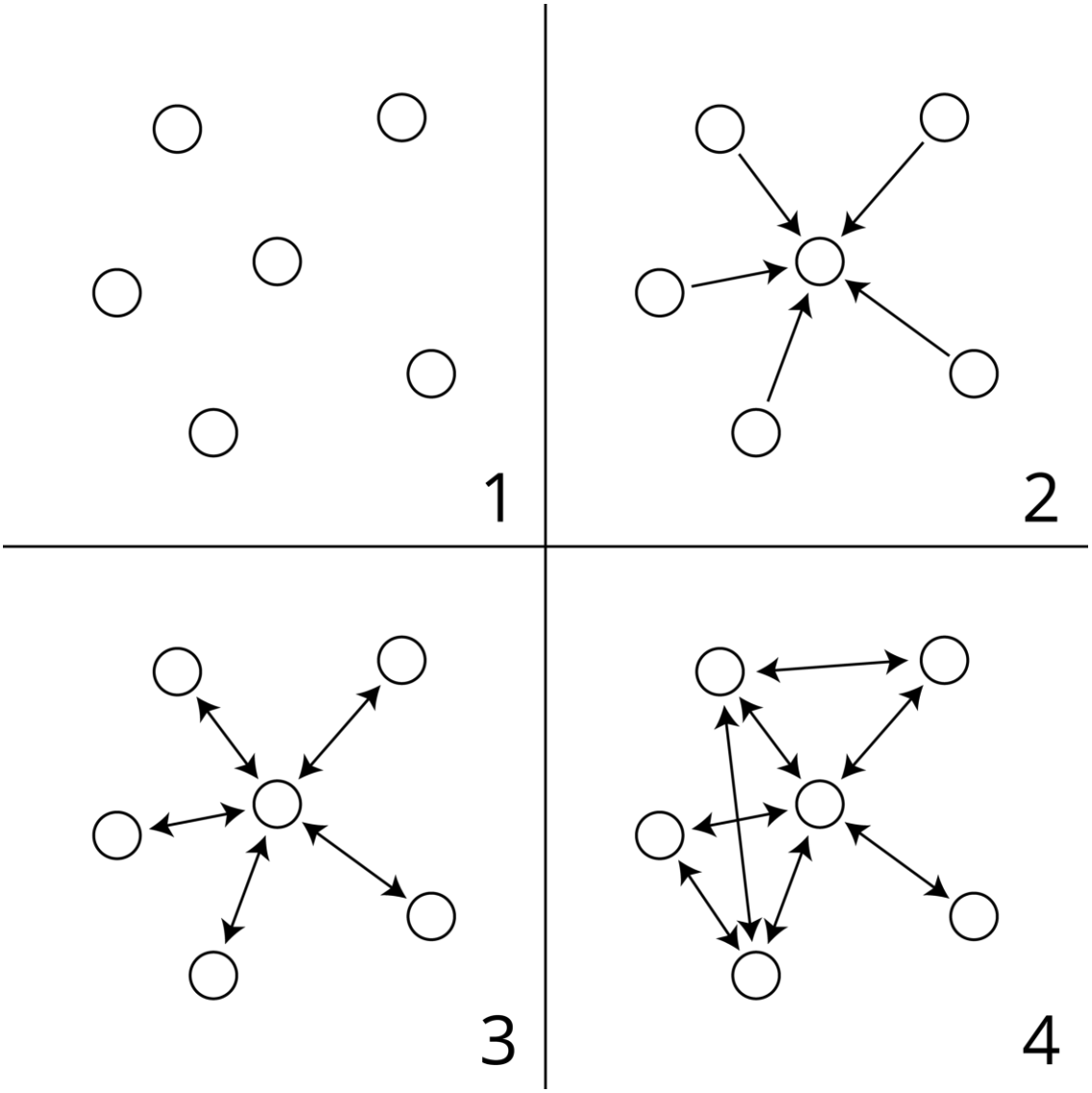


Abbildung 2.5: Nodes vernetzen sich

Peer Shuffling

Ein mögliches Angriffszenario in Blockchain-Netzwerken stellt der sog. Eclipse-Angriff dar. Dabei werden einzelne Nodes gezielt vom Netzwerk abgeschottet, indem all ihre Nachbarn von einem Angreifer kontrolliert werden. Ähnliche Abschottungen einer Node können jedoch auch zufällig im Netzwerk auftreten, da Nodes sich aufgrund des

sonst hohen Aufwandes bei der Kommunikation nicht mit allen anderen Nodes verbinden können. Um zu vermeiden, dass es zu Abschottungen von Nodes kommt, führen Nodes zwei Listen über die Peers im Netzwerk. Die erste Liste enthält die Adressen von bekannten Nodes, die zweite enthält die Adressen der von der Node aktiv genutzten Peers. Netzwerknachrichten werden nur an die Nodes in der zweiten Liste versendet und die Nodes in dieser Liste werden regelmäßig durch eine zufällige Auswahl aus der ersten Liste ausgetauscht. Das Netzwerk strukturiert sich so regelmäßig um, was dauerhafte Abschottungen unwahrscheinlich macht.

Flooding

Da ein vollvernetztes Netzwerk mit vielen Nodes sehr hohe Ansprüche an die zugrundeliegende Hardware stellen kann ($n * (n-1)$ Verbindungen), bauen Nodes nur *einige* Verbindungen auf. Um Nachrichten trotzdem im gesamten Netzwerk zu verteilen, kommt Flooding [10] zum Einsatz. Dabei werden empfangene Nachrichten an alle Nachbarn, mit Ausnahme des Senders, weitergeschickt. Außerdem werden Nachrichten nicht weitergeleitet, falls sie schon einmal auf der Node gesehen wurden.

HTTPs-Broker

Die Block-Hashes der firmeninternen Blockchains werden über einen HTTPs-Broker mit den anderen Netzwerken geteilt und dort als Transaktionen in die Blockchain geschrieben. Da jeder Block-Hash alle Transaktionen des Blockes, sowie alle Vorgängerblöcke referenziert, wird mit jedem veröffentlichten Top-Hash die gesamte Blockchain bis zu diesem Punkt von den anderen Netzwerken rückbestätigt, ohne dass Daten über den eigentlichen Inhalt der Blöcke offengelegt werden müssen. Die Abbildung 2.6 veranschaulicht diesen Vorgang. Die Grafik zeigt die Blockchains zweier Netzwerke. Diese wurden durch den Austausch der Block-Hashes `#1A` und `#2b` verknüpft.

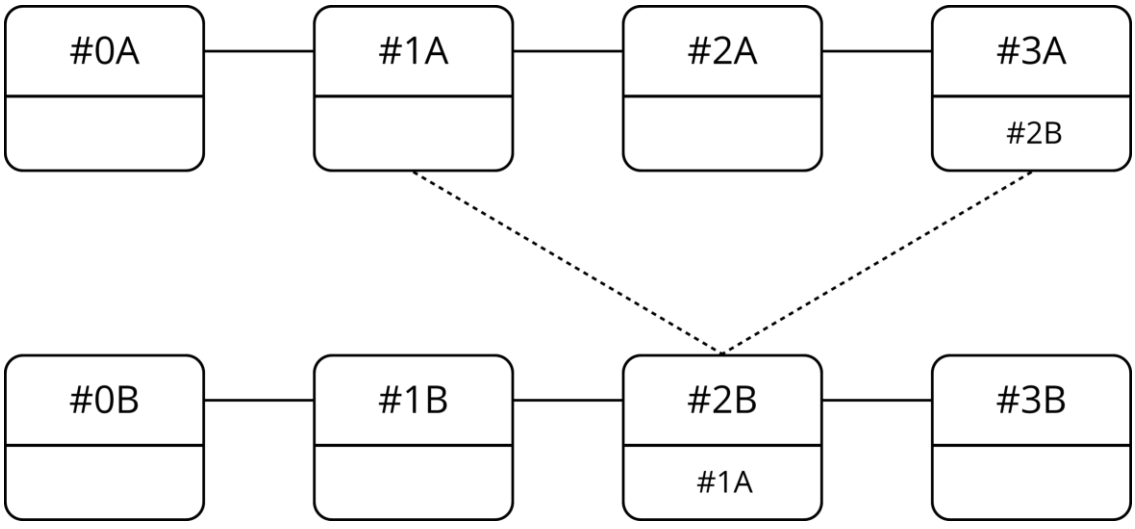


Abbildung 2.6: Gegenseitige Rückbestätigungen

Der HTTPs-Broker, der zum Datenaustausch zwischen den Netzwerken eingesetzt wird, ist so konzipiert, um möglichst wenig Angriffsfläche gegenüber externen Kräften zu bieten. Jegliche Daten, die über ihn gesendet werden, sind verschlüsselt. Lediglich die Kennung des Zielnetzwerks der jeweiligen Nachrichten wird im Klartext übertragen. Sollte ein Angreifer in den HTTPs-Broker eindringen, würde er nur verschlüsselte Datensätze vorfinden. Die Verschlüsselung für Nachrichten zum Broker und unter den Nodes wird über das in Sektion 2.3 beschriebene ECIES gelöst. Der Broker ist für den normalen Betrieb der Nodes nicht unbedingt erforderlich. Selbst ein Ausfall über längere Zeiträume kann vom System toleriert werden und wird behoben, sobald der Broker wieder zur Verfügung steht.

2.5 Transaktionen

Nachdem das Netzwerk aufgebaut wurde, können Daten in die Blockchain geschrieben werden. Die Daten werden in der `Transaction` Datenstruktur (s. Listing 2.3) verarbeitet und gespeichert. Transaktionen enthalten eine eindeutige ID. Diese ist deterministisch und wird aus dem Inhalt der Transaktion und deren Zeitstempel erzeugt. Diese ID wird beispielsweise verwendet, um festzustellen, ob eine Transaktion bereits in der Blockchain enthalten ist und um sie später aus dem Speicher zu laden. Transaktionen müssen signiert sein. Dafür wird der Hash der Nutzdaten (`payload`) signiert und in die Datenstruktur eingefügt. Damit ist der Inhalt nicht mehr veränderbar und muss somit nicht zwingend in der Transaktion gespeichert werden. Wenn die Nutzdaten nicht enthalten sind, wird die Transaktion als `stripped` gekennzeichnet. Diese leeren Transaktionen ermöglichen beispielsweise das Einsparen von Speicherplatz auf Thin Nodes.

```
pub struct Transaction {
    pub id: Vec<u8>,
    pub tags: Vec<String>,
    pub signed_hash: SignedData,
    pub contents: TransactionV1Contents,
    pub payload: Vec<u8>,
    pub stripped: bool,
    pub meta: TransactionMetaData,
}
```

Listing 2.3: Datenstruktur für Transaktionen

Transaktionen können, zusätzlich zu den Nutzdaten, beliebige Tags enthalten. Diese können auch nachträglich verändert werden und ermöglichen es, sie mit einen gewissen Kontext zu versehen. Sie werden hauptsächlich verwendet, um Transaktionen durch den Endbenutzer des Systems auffindbar zu machen. Weiterhin sind Transaktionen versioniert, um eine spätere Erweiterung des Systems abwärts kompatibel zu machen.

2.6 Blöcke

Wie bereits beschrieben, können Nodes als Validatoren im Netzwerk auftreten. Ihre Aufgabe besteht darin, Transaktionen zu Blöcken zusammenzufassen und diese im Netzwerk zu verteilen. Die für Blöcke verwendete Datenstruktur besteht aus zwei Feldern, dem `BlockHeader` und einem `MerkleTree`. Der Header beinhaltet die Daten, die zur Erzeugung des Block-Hashes erforderlich sind. Dazu zählen insbesondere der Hash des vorherigen Blockes, sowie die Wurzel des `MerkleTree`s, der die Nutzdaten des Blockes in einem Hash zusammenfasst. Außerdem beinhaltet der Header weitere Datenfelder, die in verschiedenen Konsensverfahren unterschiedliche Nutzen haben können. Die Datenstrukturen für Blocke und deren Header sind in Listing 2.4 abgebildet.

```
pub struct BlockHeader {
    pub timestamp: u128,
    pub previous_digest: Vec<u8>,
    pub difficulty: Difficulty,
    pub nonce: Vec<u128>,
    pub height: usize,
    pub merkle_root: Vec<u8>,
    pub signatures: Vec<SignedData>,
}

pub struct Block {
    pub header: BlockHeader,
    pub data: MerkleTree,
}
```

Listing 2.4: Datenstrukturen für Blöcke

Die `MerkleTree`-Datenstruktur ist so implementiert, dass die eigentlichen Nutzdaten der Transaktionen auch aus den Blöcken ausgelassen werden können. Stattdessen wird nur der jeweilige Hash der Daten abgespeichert. Damit können Nodes, die nicht über viel Speicherplatz verfügen, die Blockchain auch nur mit anteiligen Nutzdaten speichern. Abbildung 2.7 zeigt, wie Daten in Blöcke zusammengefasst werden. Die Datenfelder mit gepunkteter Umrandung symbolisieren ausgelassene Nutzdaten.

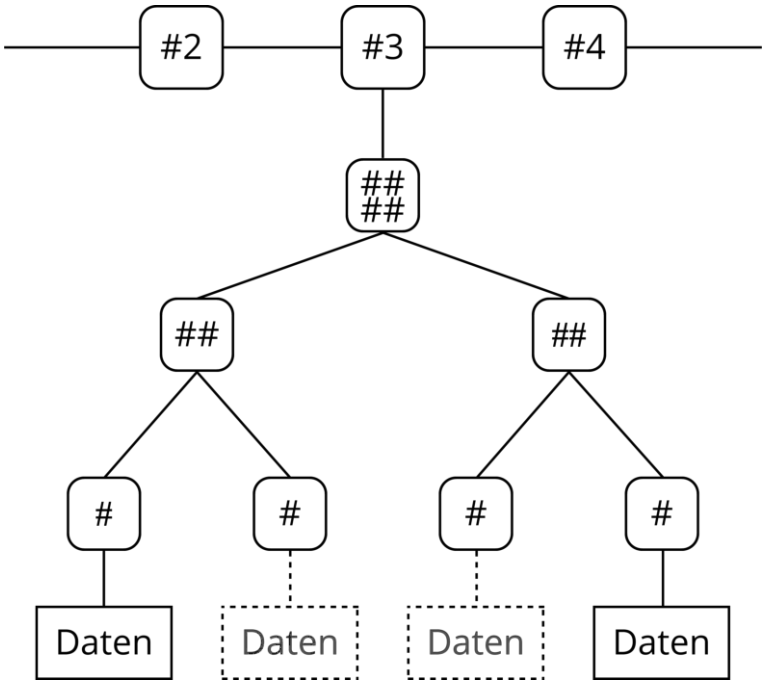


Abbildung 2.7: Zusammenfassung von Daten in einen Block

Nachdem Blöcke erzeugt und validiert wurden, werden sie an die Blockchain angehangen. Diese wird im Projekt durch eine Baumstruktur abgebildet, die alle Blöcke enthält. Die Auswahl der längsten Blockchain erfolgt durch einen austauschbaren Prozess. Die zugrundeliegende Logik deckt auch die Verwaltung von Blöcken ab, die noch keinen direkten Vorgänger in der Blockchain haben.

Nachdem Blöcke erzeugt wurden, müssen sie an die Blockchain angehangen werden. Dafür werden sie im Netzwerk verteilt, wo sie auf den einzelnen Nodes an die lokale Blockchain angehangen werden. Dabei kann es vorkommen, dass konkurrierende Blöcke empfangen werden. Diese verweisen jeweils auf den gleichen Vorgänger. Durch derartige Blöcke kann es zu einer “Spaltung” der Blockchain kommen, bei der mehrere gültige Blockchains gleichzeitig im Netzwerk existieren. Um sicherzustellen, dass das Netzwerk langfristig nur eine einzige gültige Blockchain führt, müssen diese sog. Forks aufgelöst werden. Dafür kommt eine Baumstruktur zum Einsatz, die alle empfangenen Blöcke aufnimmt und diese nach konfigurierbaren Regeln zu der “korrekten” Blockchain zusammensetzt. Dieser Vorgang läuft auf allen Nodes gleich ab, sodass, wenn alle konkurrierenden Blöcke bekannt sind, jede Node die gleiche Blockchain erhält. Diese Struktur hat auch die Aufgabe, Blöcke anzuhängen, die vor ihrem Vorgänger empfangen wurden (verwaiste Blöcke). Dafür wird beim Anhängen jedes Blockes geprüft, ob dieser der Vorgänger eines verwaisten Blockes ist. Falls ja, wird dieser auf gleiche Art und Weise angehangen, was zu einer erneuten Überprüfung führt, ob weitere verwaiste Blöcke angehangen werden können. Somit werden auch “Ketten” von verwaisten Blöcken zur Blockchain hinzugefügt. Abbildung 2.8 veranschaulicht diese Datenstruktur.

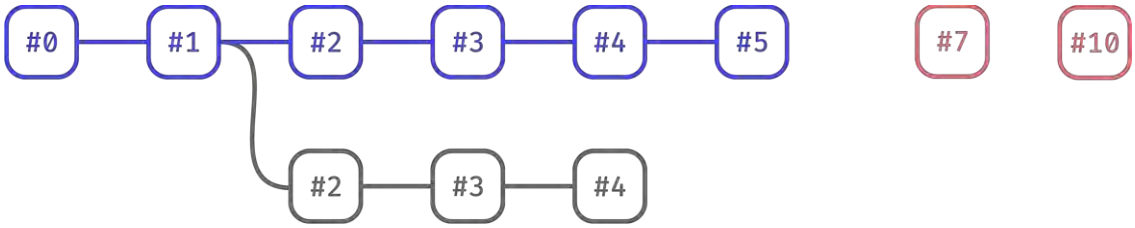


Abbildung 2.8: Baumstruktur der Blöcke mit der “korrekten” Kette in Blau, einer verworfenen Spaltung in Grau und nicht angehängten Blöcke in Rot.

2.7 Interne Abläufe

Transaktionen durchlaufen auf ihrem Weg in die Blockchain innerhalb der Nodes immer den gleichen Pfad:

1. Aufnahme aus dem Netzwerk
2. Weiterverarbeitung durch einen Worker-Thread
3. Aufnahme in den Mempool
4. Inklusion in einen Block

Nachdem Transaktionen empfangen wurden, werden diese von Worker-Threads verarbeitet. Die Anzahl dieser Threads hängt von der jeweiligen Konfiguration der Node ab. Minimal kann eine Node mit nur einem Worker betrieben werden. Sinn des Parallelbetriebs der Worker ist, dass mehrere Transaktionen gleichzeitig aufgenommen werden können, was den Datendurchsatz der Nodes verbessert. Die Worker überprüfen, ob der signierte Hash aller eingehenden Transaktionen mit dem Hash ihrer Nutzdaten übereinstimmt. Falls ja, wird die Transaktion in den Transaktions-Pool des Validator-Threads gelegt.

Bevor ein Block erzeugt wird, entnimmt der Validator seinem Transaktions-Pool je eine Transaktion, bis dieser entweder leer ist, oder er ein einstellbares Limit erreicht. Für jede entnommene Transaktion wird geprüft, ob sie sich bereits in der aktuell gültigen Blockchain befindet, falls ja, wird sie verworfen. Wenn der Validator die Transaktionen für den nächsten Block gesammelt hat, erzeugt er mit diesen einen validen Block. Die Definition von “valide” kann sich hierbei je nach Konsensverfahren unterscheiden.

Sobald ein neuer Block erzeugt wurde, wird er in das Netzwerk gesendet und, ähnlich wie eine Transaktion, von den anderen Nodes empfangen. Jede Node hängt den neuen Block in ihre Block-Baumstruktur ein und entscheidet auf Basis der Regeln des Konsensverfahrens, ob der Block zur lokal gültigen Blockchain zählt. Der Empfang neuer Blöcke kann auf Validator-Nodes auch dazu führen, dass die Erzeugung des aktuel-

len Blockes abgebrochen wird, falls der empfangene Block direkt mit Block, der aktuell erzeugt wird, kollidiert. Dieser Vorgang wird in Abbildung 2.9 schematisch dargestellt.

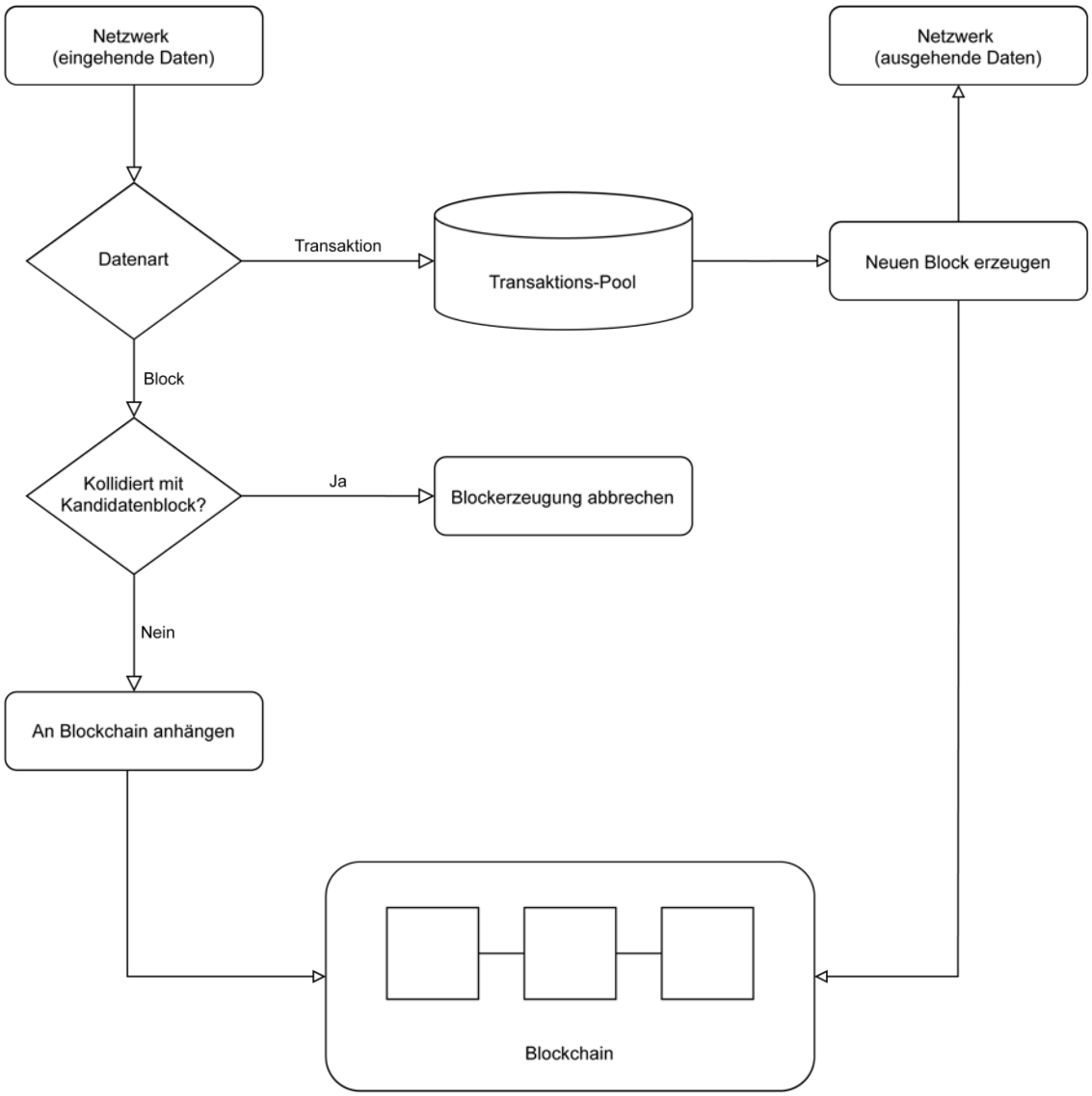


Abbildung 2.9: Blockproduktion

2.8 Datenherkunft

Die eigentlichen Nutzdaten der Transaktionen werden über das `Ingest` Modul der Nodes aufgenommen. Jeder Node-Typ kann mit diesem Modul ausgestattet werden. Das Modul stellt einige Interfaces zur Aufnahme von Daten bereit, beispielsweise eine proprietäre Netzwerkschnittstelle, die von einer der Firmen zur Einspeisung von Daten verwendet wird. Die Schnittstelle, die für die Zwecke dieser Arbeit betrachtet werden soll, ist die Dateiaufnahme. Diese Schnittstelle überwacht einen Ordner im Dateisystem und liest Dateien ein, die in diesem abgelegt werden und einem gewissen Filter entspre-

chen (bspw. nach Dateiendung). Der Inhalt dieser Dateien wird dann von der Node in Transaktionen geschrieben, die an das Netzwerk geschickt werden. Um zu verhindern, dass Daten verloren gehen, werden lokale Daten immer nur gelöscht, nachdem sie definitiv in der nächsten Verarbeitungsebene angekommen sind. Eingelesene Dateien werden vorerst markiert, indem ihr Name um eine weitere Dateiendung (`.read`) erweitert wird. Der Inhalt der Datei wird im nächsten Schritt als `payload` in eine neue Transaktion geschrieben, die an das Netzwerk geschickt wird. Dateien, die auf `.read` enden, werden nur vom System eingelesen, wenn es das erste mal startet. Damit wird sichergestellt, dass nach einem Absturz des Systems bereits eingelesene Dateien, die noch nicht verarbeitet wurden, erneut eingelesen werden. Ein ähnlicher Mechanismus sichert die ausgehenden Transaktionen ab. Noch bevor Transaktionen zum Versand in die Warteschlange des Netzwerkes gelegt werden, persistiert die Node sie vorübergehend im `pending_transactions` Ordner. Dort abgelegte Transaktionen werden regelmäßig geladen und ihr Vorhandensein in der Blockchain geprüft. Wenn sie in der Blockchain gefunden werden, löscht die Node ihre lokalen Kopien. Falls nicht, werden die betreffenden Transaktionen erneut an das Netzwerk gesendet. Dadurch dass jede Node Kopien ihrer lokalen Transaktionen behält, bis diese mit einer gewissen Sicherheit in der Blockchain enthalten sind, kann sichergestellt werden, dass Transaktionen nicht verloren gehen.

2.9 Konfiguration

Nodes können in zweierlei Hinsicht konfiguriert werden. Die erste Konfigurationsmöglichkeit besteht in der Software selbst. In dieser können Konstanten angepasst werden, die beim Kompilieren mit eingebettet werden und im Normalfall auf allen Nodes innerhalb eines Netzwerkes gleich sind. Innerhalb dieser Konfiguration können grundlegende Systemparameter eingestellt werden. Dazu zählen Netzwerkparameter wie die angestrebte Anzahl der bekannten Nodes im Netzwerk und die Anzahl der angestrebten aktiven Verbindungen, aber auch Warteintervalle für verschiedene Wartungs- und Instandhaltungsaufgaben. Zusätzlich kann hier das Verhalten der Worker-Threads eingestellt werden. Relevant hierfür sind die Anzahl dieser Threads, sowie deren Verhalten während der Synchronisierung mit anderen Nodes oder dem Fehlen von Transaktionen in der Blockchain. Weiterhin werden in dieser Konfiguration Parameter festgelegt, die das Konsensverfahren steuern, beispielsweise die angestrebte Blockzeit und -größe. Eine solche, beispielhafte Konfiguration ist in Listing 2.5 dargestellt.

```
1 // ######### NETWORKING #########
2
3 pub const TARGET_KNOWN_PEERS: usize = 20;
4 pub const TARGET_USED_PEERS: usize = 10;
5
6 pub const WAIT_TIME_TO_CHECK_PRIORITY_QUEUE: u128 = 500;
```

```
pub const WAIT_TIME_TO_MAINTENANCE: u128 = 15000;
pub const WAIT_TIME_TO_POSTMAN_STUFF: u128 = 60000;
pub const WAIT_TIME_TO_BROADCAST_PEER_LIST: u128 = 15000;
pub const WAIT_TIME_TO_SHUFFLE_PEER_LIST: u128 = 25000;

// ######### WORKER #########

pub const NUM_WORKERS: usize = 4;
pub const WAIT_TIME_TO_BLOCKCHAIN_MAINTENANCE: u128 = 30000;
pub const MAX_BLOCK_REQUESTS_IN_SYNC: usize = 20;

// ######### VALIDATOR #########

pub const MAX_TRANSACTIONS_PER_BLOCK: usize = 6000;
pub const TARGET_BLOCK_TIME_MS: u128 = 5000;
```

Listing 2.5: Node-Softwarekonfiguration (Auszug)

Die zweite Möglichkeit der Konfiguration wird durch eine Node-spezifische Konfigurationsdatei abgebildet. Innerhalb dieser Datei werden Parameter festgelegt, die für jede Node individuell sind. Beispielsweise wird die Node-Kennung in Form des Netzwerkes, sowie des Indexes der Node festgelegt. Über andere Parameter lässt sich einstellen, ob die Node neue Blöcke erzeugen (`validator`), bestimmte Transaktionen speichern (`store_transactions_from`), Daten aufnehmen (`ingest_dir`) oder mit dem HTTPs-Broker kommunizieren soll (`postman`). Eine derartige Konfiguration ist in Listing 2.6 abgebildet.

```
network = "hsmw"
index = 3

validator = true
store_transactions_from = ["hsmw-*", "iwu-1", "iwu-2"]
ingest_dir = "../../ingest/"
postman = false
```

Listing 2.6: Node-Konfigurationsdatei

3 Konsensverfahren

Innerhalb eines Blockchain-Netzwerkes arbeiten viele Computer zusammen, um eine gemeinsame zeitliche Ordnung von Daten mithilfe der Blockchain zu erzeugen. Um die Sicherheit und Funktion dieses Systems zu gewährleisten, werden sog. Konsensverfahren eingesetzt, die sich in ihren Eigenschaften stark unterscheiden können. Dieses Kapitel gibt eine Übersicht über die grundlegenden Funktionen von Konsensverfahren, deren Einsatzgebieten und beschreibt, wie derartige Verfahren verglichen werden können.

3.1 Grundlagen

Die Blockchain fungiert im Netzwerk als eine Datenstruktur, in der Daten nur hinzugefügt werden können. Sie wird von allen Netzwerkteilnehmern gespeichert und ohne zentrale Koordination erweitert. Um sicherzustellen, dass jedem Netzwerkteilnehmer letztendlich die gleiche Version der Blockchain vorliegt, werden sog. Konsensverfahren eingesetzt. Ein Konsensverfahren ist dabei als ein geteiltes Regelwerk anzusehen das sicherstellt, dass nur legitime Blöcke in die Blockchain aufgenommen werden [11].

Um Daten in die Blockchain aufzunehmen, wird immer der gleiche Prozess durchlaufen. Im ersten Schritt erzeugt ein Netzwerkteilnehmer eine Transaktion. Diese wird an das Netzwerk geschickt, wo sie unter den restlichen Netzwerkteilnehmern verteilt wird. Einige der Netzwerkteilnehmer treten als Blockerzeuger auf, fassen also viele Transaktionen zu Blöcken zusammen. Diese Erzeugung von Blöcken wird auf Basis von Regeln durchgeführt, sodass ein gültiger Block erzeugt wird. Nach der Blockerzeugung wird der neue Block ähnlich wie eine Transaktion im Netzwerk verteilt, wobei wieder alle Netzwerkteilnehmer den Block auf dessen Richtigkeit prüfen und ihn schlussendlich an ihre lokale Blockchain anhängen [12]. Dabei kann es vorkommen, dass mehrere Blöcke darum konkurrieren, an den gleichen Vorgänger angehangen zu werden. Dies wird als *Fork* bezeichnet. Der Umgang mit bzw. die Auflösung von Forks wird ebenfalls über die Regeln des Konsensverfahrens gesteuert. Eine Möglichkeit, mit Forks umzugehen, ist die Gewichtung von Blöcken anhand von bestimmten Kennzahlen. Mit dieser Gewichtung kann Blöcken sowohl einzeln, aber auch in Ketten von Blöcken eine “Punktzahl” zugeordnet werden, anhand deren sich entscheiden lässt, welche Blöcke die kanonische[2] Kette bilden. Dieser Vorgang wird schematisch in Abbildung 3.1 dargestellt.

[2] Auch als “längste” oder “richtige” Kette bezeichnet.

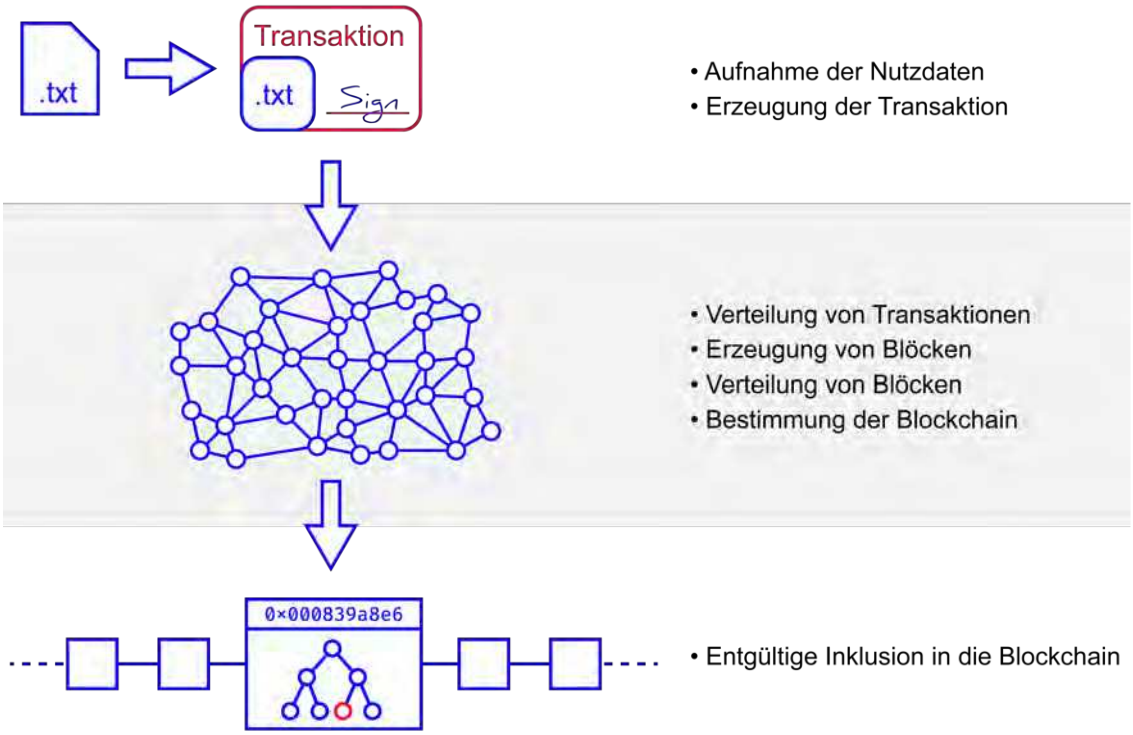


Abbildung 3.1: Weg von Daten in die Blockchain

Ein Konsensverfahren lässt sich mithilfe von Regeln beschreiben, die die Richtigkeit von Blöcken und die Wahl der kanonischen Kette festlegen. Durch die Anpassung dieser Regeln kann das Blockchain-Netzwerk verschiedene Eigenschaften erhalten. Beispielsweise kann durch die Nutzung von *verifiable delay functions* [13] bei der Erzeugung von validen Blöcken eine gewisse Zeit zwischen zwei aufeinanderfolgenden Blöcken angestrebt werden. Ein derartiges Verfahren kann beispielsweise dafür verwendet werden, um die Blockzeit so zu regulieren, dass Forks mit einer geringeren Wahrscheinlichkeit auftreten. Außerdem kann festgelegt werden, dass nur Blöcke bis zu gewissen Größe zulässig sind, um den Versand großer Datenmengen im Netzwerk einzuschränken. Wenn die Regeln für Blockzeit und Blockgröße vereint werden, erhält die Blockchain einen theoretischen maximalen Durchsatz. Diese und weitere Interaktionen der Regeln innerhalb des Konsensverfahrens können unterschiedliche Eigenschaften hervorbringen, mit denen das Netzwerk an spezielle Anwendungsfälle angepasst werden kann.

Ein wichtiger Unterscheidungspunkt zwischen Konsensverfahren liegt in der Natur der Netzwerke, in denen sie eingesetzt werden. *Öffentlich* zugängliche Netzwerke, oftmals ohne jegliche Rollen- oder Befugnissysteme, stellen ein Anwendungsgebiet für Konsensverfahren dar. Beispielhaft dafür sind frühe Kryptowährungen wie Bitcoin und Ethereum.

Der Konsens in derartig offenen Netzwerken wird meist anhand eines Regelwerkes gefunden, das die Verwendung von knappen Ressourcen zur Grundlage hat. So können

Blöcke u.a. nur erzeugt werden, indem ein kryptografisches Puzzle gelöst wird, für das kein effizienter Lösungsweg bekannt ist. Derartige Puzzles lassen sich nur lösen, indem zufällige Eingabewerte getestet werden, die mit einer gewissen Wahrscheinlichkeit zu einer Lösung führen. Die Schwierigkeit einer solchen Aufgabe kann angepasst werden, um eine gewisse durchschnittliche Lösungszeit zu erreichen. Damit wird die Erzeugung von Blöcken an die Nutzung von Ressourcen gebunden. Im Fall von Bitcoin und Ethereum ist diese Ressource die Elektrizität, die für den Betrieb von Rechenhardware genutzt wird, sowie diese Rechenhardware selbst.

Ein weiterer Aspekt dieser Systeme stellt die Anreizsetzung für die regelkonforme Teilnahme am Konsensverfahren dar. In Blockchain Netzwerken mit einer eigenen Währung kann diese für die erfolgreiche Teilnahme am Konsens ausgezahlt werden. Durch diese Wertschöpfung können die Erzeuger von Blöcken in die zur deren Erzeugung notwendigen Ressourcen (Rechenkraft, Speicher, etc.) investieren und den Konsens weiter absichern.

Die zweite Art von Blockchain Netzwerken sind *private* Netzwerke. Diese haben zumeist Zugangsbeschränkungen und dementsprechend weniger Nutzer. Solche Netzwerke werden für spezielle Einsatzzwecke genutzt und können somit stärker spezialisiert sein als öffentliche Netzwerke. Da in privaten Blockchain Netzwerken nicht zwingend ein monetärer Anreiz für die Teilnahme besteht, müssen teilweise andere Konsensverfahren genutzt werden. So können Konsensverfahren, die auf dem Verbrauch von Ressourcen basieren, möglicherweise nicht kosteneffektiv zum Einsatz kommen. Jedoch ist es möglich, andere Kontrollen auszuüben. So kann zum Beispiel sichere Hardware oder eine spezielle Netzwerkumgebung verwendet werden.

Konsens im Projekt

Ziel des Projektes safe-UR-chain ist die Schaffung von Verantwortlichkeit für Daten innerhalb und zwischen Firmen. Die Blockchain wird dabei verwendet, um Daten innerhalb eines Netzwerkes miteinander zu verknüpfen und die Existenz dieser Daten zu einem bestimmten Zeitpunkt zu belegen. Da die Top-Hashes regelmäßig zwischen den Firmennetzwerken synchronisiert werden, ist der Zeitraum, in dem ein Angreifer Veränderungen vornehmen kann, eingeschränkt. Zusätzlich besteht in den Netzwerken eine hohe IT-Sicherheit und Nodes können nur durch einen Administrator hinzugefügt werden. Dadurch werden Angriffe erschwert.

3.2 Vergleichbarkeit von Konsensverfahren

Die Arbeitsweise von verschiedenen Konsensverfahren kann starke Unterschiede aufweisen, jedoch verfolgen alle Konsensverfahren das Ziel, die Blockchain im Netzwerk synchron zu halten und mit neuen Blöcken zu erweitern. Um Konsensverfahren zu

vergleichen, kann diese Erweiterung der Blockchain analysiert werden. Zu den Metriken [14], die hierfür verwendet werden können, zählen:

- Transaktionsdurchsatz
- Datendurchsatz
- Latenz

Der **Transaktionsdurchsatz** beschreibt die Anzahl der Transaktionen, die in einem bestimmten Zeitintervall in die Blockchain aufgenommen werden. Es kann zwischen der Aufnahmerate von einzelnen Blöcken, oder auch in Teilen der Blockchain unterschieden werden. Um den Durchsatz einzelner Blöcke zu errechnen, wird die Anzahl der Transaktionen innerhalb des Blockes seiner Blockzeit gegenübergestellt. Der Transaktionsdurchsatz der gesamten Blockchain lässt sich ermitteln, indem mehrere aufeinanderfolgende Blöcke analysiert werden. Dafür wird die Anzahl der Transaktionen innerhalb der betrachteten Blöcke ermittelt und dann durch die Zeit geteilt, in der die betrachteten Blöcke erzeugt wurden [3]. Als Einheit für den Transaktionsdurchsatz wird häufig TPS (transactions per second) verwendet. Eine weitere Metrik, die sich auf ähnliche Art feststellen lässt, ist der **Datendurchsatz**. Er kann wie auch der Transaktionsdurchsatz auf Block- oder Blockchain-Ebene errechnet werden. Hierfür wird anstelle der Transaktionsanzahl die Größe der Transaktionen verwendet. Dabei können entweder die gesamte Transaktion oder nur die enthaltenen Nutzdaten verwendet werden. Der Datendurchsatz kann in Bytes pro Sekunde bzw. auch mit den aus Bytes abgeleiteten Speichereinheiten angegeben werden (z.B. MB/s).

Zusätzlich zu diesen beiden Metriken kann die **Latenz** gemessen werden. Dabei handelt es sich um die Zeit, die zwischen der Erzeugung von Transaktionen und deren Aufnahme in die Blockchain vergeht. Auch bei dieser Metrik kann zwischen Block- und Blockchain-Ebene unterschieden werden. Um die Latenz innerhalb eines Blockes zu berechnen, werden die Zeitstempel der Transaktionen mit dem Zeitpunkt verglichen, zu dem der Block zur Blockchain hinzugefügt wurde. Aus den Differenzen kann dann eine durchschnittliche Transaktionslatenz im Block errechnet werden. Für die Berechnung der Latenz innerhalb der Blockchain für einen bestimmten Zeitraum wird der Durchschnitt der Block-Latenzen in diesem Zeitraum gebildet. Bei der Berechnung von Latenzen kann weiterhin die Definition der Erzeugung von Transaktionen ausschlaggebend sein. Wird beispielsweise die erste Veröffentlichung im Netzwerk als Erzeugung angesehen, wird die Latenz geringer ausfallen, als wenn der Zeitpunkt ihrer tatsächlichen Erzeugung verwendet wird, da Transaktionen ggf. innerhalb der Node nicht sofort nach ihrer Erzeugung verschickt werden.

Um diese Metriken für verschiedene Konsensverfahren zu erheben, können unterschiedliche Ansätze verwendet werden. So ist es möglich, Konsensverfahren auf rein theore-

[3] $Transaktionsdurchsatz = \frac{Transaktionszahl}{Zeitintervall}$

tischer Ebene zu betrachten und die Metriken über Berechnungen zu erheben. Mit diesem Ansatz können theoretische Maximalwerte für die Metriken in einem perfekten System errechnet werden. Somit lassen sich Konsensverfahren ohne die Einschränkungen der Hardware, Netzwerkgeschwindikeit oder variierender Blockzeiten betrachten. Beispielhaft für eine derartige Betrachtung ist die in [15] durchgeführte Analyse des theoretisch maximalen Transaktionsdurchsatzes der Bitcoin-Blockchain. Dafür wurde ausgehend von der 1 MB Blockgröße und von kleinstmöglichen Transaktionen berechnet, wie viele Transaktionen maximal in einem Block Platz finden können. Mit dieser Transaktionsmenge wurde anschließend der maximale Transaktionsdurchsatz bei einer Blockzeit von exakt 10 Minuten errechnet. Als maximaler Durchsatz wurden 27 TPS festgestellt. Dieser Wert beruht jedoch auf Transaktionstypen, die in der realen Nutzung von Bitcoin keine Anwendung finden. Ein realistischerer Wert von 7 Transaktionen pro Sekunde wurde unter Annahme von Transaktionstypen errechnet, die tatsächlich verwendet werden [16] [15]. Selbst dieser Wert weicht jedoch vom höchsten Transaktionsdurchsatz ab, der durch das Bitcoin-Netzwerk mit 5,68 TPS jemals erzielt wurde [17][4]. Ein ähnlicher Vergleich für Ethereum zeigt, dass die üblicherweise angegebenen 15 Transaktionen pro Sekunde [18] überschritten werden können [19][5].

Diese Gegenüberstellung zeigt, dass theoretisch ermittelte Metriken für den Durchsatz von Konsensverfahren von Annahmen beeinflusst werden, die ein teilweise vereinfachtes Bild der realen Anwendung zeigen. Zusätzlich können Konsensverfahren beispielsweise auch von der Implementierung des restlichen Systems abhängen. Ein Beispiel hierfür ist die Datenaufnahme in das System, die in [14] untersucht wurde. Diese Untersuchung zeigt, dass drei private Blockchain-Netzwerke u.a. sehr unterschiedlich mit neuen Transaktionen umgehen und teilweise die Datenrate pro Netzwerkteilnehmer begrenzen. Derartige Unterschiede erschweren die Untersuchung von Konsensverfahren in verschiedenen Systemen, da diese nicht isoliert betrachtet werden können.

Um Konsensverfahren miteinander zu vergleichen, sodass die Ergebnisse des Vergleichs auch Implikationen für die reale Anwendung haben, sollten diese innerhalb des gleichen Systems und unter gleichen Bedingungen geprüft werden. Für die Auswahl der zu testenden Konsensverfahren können jedoch ihre theoretischen Eigenschaften herangezogen werden. Da im Projekt ein komplett selbst entwickeltes Blockchain-System zum Einsatz kommt, ist dieser Ansatz unkompliziert möglich. Zu dessen Realisierung wurden vier Schritte unternommen:

1. Implementierung eines generischen Interfaces für Konsensverfahren
2. Implementierung einiger Konsensverfahren

[4] Am 14.12.2017 wurden 490.459 Transaktionen zur Bitcoin-Blockchain hinzugefügt, $\frac{490.459}{24 \cdot 60 \cdot 60} = 5,68$ TPS über den Tag

[5] Am 09.05.2021 wurden 1.716.489 Transaktionen zur Ethereum-Blockchain hinzugefügt, $\frac{1.716.600}{24 \cdot 60 \cdot 60} = 19,87$ TPS über den Tag

3. Implementierung eines Test-Frameworks mit dem sich ein lokales Node-Netzwerk starten und überwachen lässt
4. Implementierung einer automatischen Versuchsauswertung

3.3 Generisches Interface

Die Blockchain-Software wurde vorerst mit einer einfachen Implementierung von Proof of Work [12] entwickelt. Dabei waren die Funktionen des Konsensverfahrens nicht von denen der Blockchain getrennt. Diese Funktionen wurden im nächsten Schritt in ein eigenes Modul ausgelagert und so angepasst, dass sie vom Code der gesamten Node aus aufgerufen werden konnten. Die Auslagerung in ein externes Modul macht es möglich, das Konsensverfahren direkt bei der Kompilierung auszuwählen und in das System einzubetten. So kann nur Code von jeweils einem Konsensverfahren ohne weitere Änderungen in die Node-Software aufgenommen werden. Das Interface, nach dem jedes Konsensverfahren in diesem Modul implementiert werden muss, ist in Listing 3.1 aufgeführt.

```
pub trait ProofOfSomething {
    pub const NAME: &str;

    pub fn block_is_valid(
        block: &Block
    ) -> bool;

    pub fn block_is_partially_valid(
        block: &Block
    ) -> BlockValidity;

    pub fn next_block(
        crypto_logic: CryptoLogic,
        transaction_pool: &mut Vec<Transaction>,
        blockchain: Arc<Mutex<Blockchain>>,
        shutdown: Arc<std::sync::RwLock<bool>>,
    ) -> Option<Block>;

    pub fn next_diff(
        look_back_blocks_headers: &[BlockHeader]
    ) -> Difficulty;

    pub fn block_work(
        parent_header: &BlockHeader,
        child_header: &BlockHeader
    ) -> u128;
}
```

Listing 3.1: Konsensverfahren-Interface

Die Konstante `NAME` hat einen rein informativen Charakter. Sie wird zur späteren Identifizierung des Konsensverfahrens in Testprotokolle geschrieben.

Die eigentliche Funktionalität der Konsensverfahren lässt sich mit fünf Funktionen abbilden. Die zwei ersten Funktionen bilden die definitive oder teilweise Validität von Blöcken ab. Dabei wird `block_is_valid` verwendet, um Blöcke in valide und invalide einzuteilen. Die Funktion wird aufgerufen, bevor Blöcke zur Blockchain hinzugefügt werden und stellt sicher, dass nur valide Blöcke gespeichert werden.

Um die Erzeugung von validen Blöcken auf mehrere Nodes zu verteilen, ist es notwendig, teilweise invalide Blöcke erkennen zu können, beispielsweise für Konsensverfahren, bei denen Blöcke von einer gewissen Anzahl Netzwerkteilnehmer signiert werden müssen. Für derartige Anwendungsfälle kommt die Funktion `block_is_partially_valid` zum Einsatz. Im Gegensatz zu `block_is_valid`, kann diese Funktion einen Block auch als teilweise gültig einstufen. Dafür kommt eine `Enumeration` zum Einsatz (s. Listing 3.2), die die Werte `Valid`, `Invalid` und `PartiallyValid` annehmen kann. Dabei enthält `PartiallyValid` einen Wert der angibt, was mit teilweise gültigen Blöcken geschehen soll. Blöcke können beispielsweise signiert und dann weitergeschickt oder ab einem gewissen Alter verworfen werden.

```
1  pub enum BlockValidity {
2      Valid,
3      Invalid,
4      PartiallyValid(PartiallyValidBlockAction),
5  }
6
7  pub enum PartiallyValidBlockAction {
8      SignIfValidatorAndResend,
9      Discard,
10 }
```

Listing 3.2: Datenstrukturen zur Repräsentation der Block-Gültigkeit

Die `next_block` Funktion wird verwendet, um einen neuen, (ggf. teilweise) gültigen Block zu erzeugen. Dafür nimmt sie u.a. eine Liste der einzufügenden Transaktionen entgegen. Diese Funktion wird je nach Konsensverfahren eine gewisse Zeit laufen und dann entweder einen neu erzeugten Block oder den Wert `None` zurückgeben. Diesem Design liegt zugrunde, dass während der Laufzeit der Funktion beispielsweise ein auf einer anderen Node erzeugter Block empfangen und in die Blockchain eingefügt worden sein kann. In diesem Fall sollte die Funktion keinen neuen Block zurückgeben. Innerhalb dieser Funktion kann auch weitere Logik, beispielsweise zur Deduplizierung von Transaktionen, eingefügt werden.

Jeder Block hat in seinem `BlockHeader` das Feld `difficulty`, über das beispielswei-

se bei Proof of Work die Blockzeit beeinflusst wird. Die Schwierigkeit für den nächsten Block wird hierbei ausgehend von den Blockzeiten und Schwierigkeiten der Vorgänger angepasst. Um eine derartige Logik abzubilden, kann die `next_diff` Funktion verwendet werden. In Verfahren, die das Konzept der Schwierigkeit nicht nutzen, kann `next_diff` auch einen konstanten Rückgabewert erhalten.

Mit den bisher genannten Funktionen lassen sich neue Blöcke erzeugen und validieren. Für die Aufnahme in die Blockchain ist es jedoch auch notwendig, Blöcke miteinander zu vergleichen, damit etwaige Forks aufgelöst werden können. Dafür verwendet die bereits beschriebene Baumstruktur (s. 2.6), die als `BlockTree` bezeichnet wird, den Begriff der "Arbeit", um die kanonische Kette zu erzeugen. Die `block_work` Funktion kann verwendet werden, um Blöcken diese "Arbeit" zuzuweisen, bzw. ein Scoring zu ermöglichen. Dafür nimmt die Funktion den `BlockHeader` von je einem Block und dessen Vorgänger entgegen und gibt einen numerischen Wert als Arbeit zurück.

Zusätzlich zu diesen Funktionen wird das Konsensverfahren über Konstanten konfiguriert, die auch dann gleich bleiben, wenn das Konsensverfahren ausgetauscht wird. Diese Konstanten (s. Listing 3.3) beeinflussen die Blockgröße- und -zeit (`MAX_TRANSACTIONS_PER_BLOCK` und `TARGET_BLOCK_TIME_MS`), sowie die Anzahl der Blöcke, auf die für die Berechnung der nächsten Schwierigkeit zurückgeblickt werden soll (`LOOK_BACK_FOR_DIFF`). Mit der `UPDATE_DIFF_EVERY_N_BLOCKS` Konstante lässt sich einstellen, nach wie vielen Blöcken die Schwierigkeit angepasst werden soll.

```
1 pub const MAX_TRANSACTIONS_PER_BLOCK: usize;
2 pub const TARGET_BLOCK_TIME_MS: u128;
3 pub const LOOK_BACK_FOR_DIFF: usize;
4 pub const UPDATE_DIFF_EVERY_N_BLOCKS: usize;
```

Listing 3.3: Globale Konstanten zur Konfiguration vom Konsensverfahren (Ausschnitt)

3.4 Proof of Work

Bei der Implementierung von Proof of Work, die im Bitcoin-Netzwerk zum Einsatz kommt, sind Blöcke nur dann gültig, wenn ihr Hash-Wert mit einer bestimmten Anzahl von `0`-Bits beginnt. Da die Wahrscheinlichkeit für jedes Bit mit diesem Wert immer bei 1:1 liegt, halbiert sich die Wahrscheinlichkeit, einen gültigen Hash zu finden mit jedem `0`-Bit, das dazu kommt. Um den Hash Wert eines Blockes zu verändern, kann beispielsweise der Inhalt des Nonce-Felds verändert werden [12] [20].

Ein ähnlicher Mechanismus kommt auch bei der Proof of Work-Variante im Projekt zum Einsatz. Hier werden jedoch nicht die Bits, sondern die Bytes des Block-Hashes ausgewertet, um dessen Gültigkeit zu prüfen. Grund dafür ist, dass Hash-Werte innerhalb der Rust-Implementierung von SHA-256 als Byte-Array behandelt werden (`Vec<u8>`).

Ziel der Prüfung ist es, einen Hash zu fordern, der nur mit einer gewissen Wahrscheinlichkeit auftritt und diese Wahrscheinlichkeit einstellen zu können. Dafür werden die Bytes des Hashes, begonnen bei Index 0, auf ihren numerischen Wert geprüft. Dabei muss der Wert des Bytes unter dem in der Schwierigkeit gespeicherten Wert liegen. Die Schwierigkeit wird als Tupel von einem `usize` und einem `u8` Wert dargestellt (s. Listing 3.4). Der `usize` Wert gibt an, wie viele der Bytes den Wert `00000000` (binär) haben müssen. Und der `u8` Wert gibt an, in welchen Wertebereich das letzte Byte im Anfangsbereich fallen muss. Dieser Wertebereich ist so definiert, dass bei einer `u8` Schwierigkeit von `0` jeder Wert zwischen `0` und `255` akzeptiert wird, bei einem `u8` Wert von `100` jedoch nur Werte zwischen `0` und `155`. Der Wertebereich wird berechnet, indem die Schwierigkeit von 255 abgezogen wird. Das Byte muss dann $Byte \leq 255 - u8$ erfüllen, um gültig zu sein. Dieser Zusammenhang wird in Abbildung 3.2 veranschaulicht.

```
1 pub type Difficulty = (usize, u8);
```

Listing 3.4: Schwierigkeit

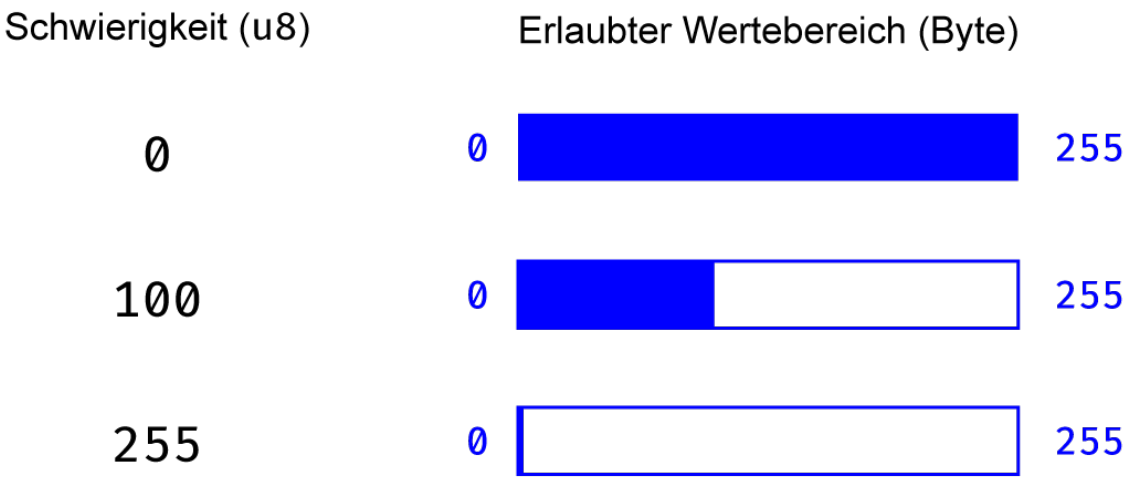


Abbildung 3.2: Wertebereich für ein einzelnes Byte nach dem u8 Wert der Schwierigkeit

Nachdem das “letzte” Byte, das im Hash für die Schwierigkeit relevant ist, überprüft wurde, werden die Bytes *bis* zu diesem Byte überprüft. Wenn all diese Bytes einen Wert von `0` haben, besteht der Hash die Gültigkeitsprüfung (s. Abb. 3.3).

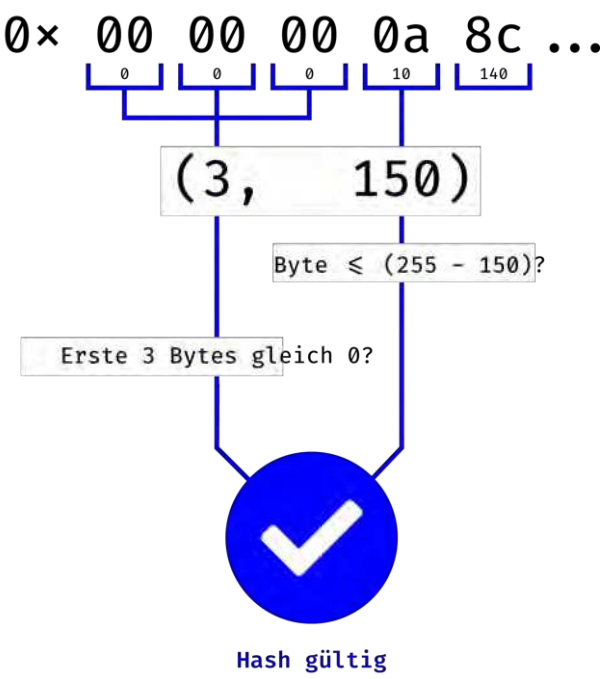


Abbildung 3.3: Überprüfung des Block Hashes bei Proof of Work

Dieser Prozess wird vom Code in Listing 3.5 abgebildet. Die Prüfungen sind jedoch invertiert, um eine logische Operation zu sparen. $!(Byte \leq 255 - u8)$ wird zu $(Byte > 255 - u8)$.

```
pub fn block_is_valid(block: &Block) -> bool {
    let digest = block.get_digest();

    if digest[block.header.difficulty.0] > 255 -
                block.header.difficulty.1 {
        return false;
    }

    let mut i = 0;
    while i < block.header.difficulty.0 {
        if digest[i] > 0 {
            return false;
        }

        i += 1;
    }

    return true;
}
```

Listing 3.5: Prüfung auf Gültigkeit eines Blockes für Proof of Work

In Proof of Work können Blöcke nur entweder gültig oder ungültig sein. Diese Tatsache wird von der `block_is_partially_valid` Funktion (s. Listing 3.6) widergespiegelt:

```
pub fn block_is_partially_valid(block: &Block) -> BlockValidity {
    if block_is_valid(block) {
        return BlockValidity::Valid;
    }

    BlockValidity::Invalid
}
```

Listing 3.6: Prüfung auf teilweise Gültigkeit eines Blockes in Proof of Work

Blöcke können nun validiert werden. Bevor jedoch neue Blöcke erzeugt werden können, muss als Nächstes die Schwierigkeitsanpassung zur Regulierung der Blockzeit umgesetzt werden. Diese Anpassung soll die Schwierigkeit erhöhen, wenn die Blockzeit zu gering ist und sie senken, wenn zu viel Zeit zwischen Blöcken vergeht. Die angestrebte Blockzeit ist dabei in der Konstante `TARGET_BLOCK_TIME_MS` festgesetzt. Für die Festlegung der nächsten Schwierigkeit wurde für Proof of Work folgende Schrittfolge umgesetzt:

1. Überprüfung, ob die Schwierigkeit im nächsten Block angepasst werden muss
2. Sammlung der letzten Block-Zeitstempel
3. Berechnung der durchschnittlichen Abstände zwischen den Blöcken (Blockzeiten)
4. Schwierigkeitsanpassung je nach angestrebter Blockzeit

Dafür wird, wie bereits erwähnt, folgende Funktionssignatur eingesetzt:

```
pub fn next_diff(look_back_blocks_headers: &[BlockHeader])
                -> Difficulty;
```

Um festzustellen, ob die Schwierigkeit überhaupt angepasst werden soll, wird die Blockhöhe des letzten Blockes *modulo* der Konstante `UPDATE_DIFF_EVERY_N_BLOCKS` genommen. Wenn dieser Wert `0` ist, wird die Schwierigkeit angepasst, andernfalls wird die letzte Schwierigkeit zurückgegeben.

Im nächsten Schritt werden die Timestamps der vorangegangenen Blöcke gesammelt, um die Blockzeiten zu berechnen. Dafür werden die Timestamps aus den Block-Headern kopiert und in einer Liste abgelegt. Diese Liste wird umgekehrt und jeder Timestamp wird vom Timestamp des vorhergehenden Blockes abgezogen und die Differenzen werden aufsummiert, um den Durchschnitt zu errechnen (s. Listing 3.7).

```
let timestamps: Vec<u128> = look_back_blocks_headers
    .iter()
    .rev()
    .map(|h| h.timestamp)
    .collect();

let mut deltas = 0;
for i in 1..timestamps.len() {
    deltas += timestamps[i - 1] - timestamps[i];
}

let delta_t = deltas / timestamps.len() as u128;
```

Listing 3.7: Berechnung der durchschnittlichen Blockzeit

Die ermittelte Blockzeit wird nun mit dem Zielwert für die Blockzeit verglichen. Falls sie höher ist als der Zielwert, wird die numerische Repräsentation der Schwierigkeit um 1 verringert und als Tupel-Schwierigkeit zurückgegeben. Wenn sie geringer als der Zielwert ausfällt, wird die Schwierigkeit reduziert und falls sie exakt dem Ziel entspricht, wird die Schwierigkeit nicht angepasst (s. Listing 3.8).

```
let last_diff = last_block.difficulty;
let mut last_diff_num = diff_to_num(last_diff);

// return the next difficulty
match delta_t.cmp(&TARGET_BLOCK_TIME_MS) {
    Ordering::Less => num_to_diff(last_diff_num + 1),
    Ordering::Greater => num_to_diff(last_diff_num - 1),
    Ordering::Equal => last_diff,
}
```

Listing 3.8: Schwierigkeitsanpassung

Um diesen Prozess zu vereinfachen, werden die Helferfunktionen `diff_to_num` und `num_to_diff` herangezogen. Sie dienen dazu, die Schwierigkeit, die als Tupel definiert ist, in eine numerische Form zu bringen, mit der einfacher gearbeitet werden kann und um die numerische Repräsentation wieder in einen Tupel umzuwandeln. In `diff_to_num` wird die numerische Form gebildet, indem der erste Wert im Tupel mit 256 multiplizert und zum zweiten Wert addiert wird. Damit kann die Schwierigkeit als fortlaufende Zahl dargestellt werden. Mithilfe dieser Funktionen kann leichter mit Schwierigkeiten umgegangen werden, ohne bei der eigentlichen Überprüfung die Effizienz der Datenstruktur einzuschränken. Tabelle 3.1 enthält einige Tupel-Wert-Paare zur Veranschaulichung.

(0,0)	0
(0, 255)	255
(1,0)	256
(1,1)	257

Tabelle 3.1: Beispiele für Schwierigkeiten als Tupel (links) und in ihrer numerischen Form (rechts)

Mit der Überprüfung auf Block-Gültigkeit und der Berechnung der nächsten Schwierigkeit können Blöcke nun erstellt werden. Dafür wird die Funktion `next_block` (s. Listing 3.9) mit einer Liste von Transaktionen, einer Referenz auf die Blockchain, einem Objekt für die Erstellung von kryptografischen Signaturen und ein `shutdown` Handle zum Abbruch der Blockerzeugung übergeben. Die Funktion gibt einen optionalen Wert zurück, der einen Block enthält, falls dessen Erzeugung erfolgreich war.

```
pub fn next_block(
    crypto_logic: CryptoLogic,
    transaction_pool: &mut Vec<Transaction>,
    blockchain: Arc<Mutex<Blockchain>>,
    shutdown: Arc<std::sync::RwLock<bool>>,
) -> Option<Block>;
```

Listing 3.9: Funktionskopf der Funktion für die Erstellung des nächsten Blockes

Der erste Schritt innerhalb der Erzeugung eines neuen Blockes ist die Auswahl der Transaktionen, die in diesen eingefügt werden sollen. Dafür wird die Helferfunktion `transactions_for_next_block` genutzt, diese stellt sicher, dass keine Duplikate in den Block aufgenommen werden und gibt maximal so viele Transaktionen zurück, wie durch die Konstante `MAX_TRANSACTIONS_PER_BLOCK` festgelegt wurden. Diese Transaktionen werden als Nächstes in einem `MerkleTree` zusammengefasst und in den zu erstellenden Block (auch "Kandidatenblock") eingefügt. Der neue Block erhält außerdem den Hash seines Vorgängers und die Angabe der Schwierigkeit. Diese Werte werden über den `Block::new()` Konstruktor in den Block eingegeben. Der neue Block ist nun erstellt und besteht aus einem Header und den Nutzdaten, ist jedoch noch nicht gültig. Bei Proof of Work muss der Block jetzt nach `block_is_valid` einen gültigen Hash erhalten. Dafür wird der Wert des `nonce` Feldes im Header so lange mit Zufallszahlen ausgefüllt, bis der Block einen gültigen Hash erhält.

Dieser Vorgang kann bei Verfahren wie Proof of Work durchaus einige Zeit in Anspruch nehmen. In dieser Zeit kann jedoch ein anderer, gültiger Block mit dem gleichen Vorgänger empfangen werden. Es ist ebenso möglich, dass während des Minings eine Fork aufgelöst wird und sich dementsprechend der Vorgängerblock ändert. In diesen Fällen sollte die Blockproduktion abgebrochen bzw. angepasst werden, um eine Fork zu verhindern. Für diese Funktionalität wird der Validierungsvorgang regelmäßig kurz

unterbrochen und es wird geprüft, ob die Blockproduktion abgebrochen oder der Kandidatenblock angepasst werden muss. Falls der Vorgang abgebrochen werden musste, gibt die Funktion `None` zurück und falls der Block erfolgreich erzeugt wurde, wird er als `Some(Block)` zurückgegeben (s. Listing 3.10).

```
let transactions =
    transactions_for_next_block(transaction_pool,
                blockchain.clone(), crypto_logic);

let merkle_tree = merkle_tree_from_transactions(transactions);

let mut candidate = Block::new(
    blockchain.get_top_block().get_digest(),
    blockchain.next_diff(),
    merkle_tree,
    blockchain.len(),
);

let mut check_interrupt_in = 10000;

while !candidate.is_valid() {
    candidate.update_nonce(5);
    check_interrupt_in -= 1;

    if check_interrupt_in == 0 {
        check_interrupt_in = 10000;

        // check if:
        // - a rival block has been added
        // - a shutdown command was issued
        // - the parent block has changed
    }
}

Some(candidate)
```

Listing 3.10: Logik zur Blockerzeugung in Proof of Work

Nachdem Blöcke erzeugt wurden, werden diese an die Blockchain angefügt. Sollten mehrere Blöcke mit dem gleichen Vorgänger erzeugt worden sein, kommt es zu einer Fork. Diese wird im System aufgelöst, indem die bereits beschriebene Baumstruktur vom Block mit dem höchsten kumulativen “Arbeit”/“Work” rückwärts durchlaufen wird. Um den “Work”-Wert für jeden einzelnen Block zu bestimmen, wird die `block_work` Funktion (s. Listing 3.11) aufgerufen. Bei Proof of Work gibt diese den `difficulty`

Wert aus dem `BlockHeader` als numerischen Wert zurück. Damit wird die Blockchain gültig, für die die kumulierte Schwierigkeit am höchsten ist.

```
pub fn block_work(
        _parent_header: &BlockHeader,
        child_header: &BlockHeader
) -> u128 {
    diff_to_num(child_header.difficulty) as u128
}
```

Listing 3.11: Bestimmung der Block-Arbeit in Proof of Work

3.5 Proof of Elapsed Time

Das zweite Konsensverfahren, das für das System implementiert wurde, ist Proof of Elapsed Time. Dieses Verfahren wurde ursprünglich von Intel für die Verwendung in Hyperledger Sawtooth entwickelt [21] [22]. Die grundlegende Idee ist die gleiche wie bei Proof of Work: die Blockerzeugung wird über ein probabilistisches Verfahren umgesetzt, bei dem Blöcke so erzeugt werden, dass sich die Blockzeiten im Durchschnitt einem bestimmten Mittelwert annähern. Während dies bei Proof of Work durch die Suche nach einem gültigen Block-Hash umgesetzt wird, die auf den Eigenschaften von Hash-Funktionen beruht, setzt Proof of Elapsed Time stattdessen auf eine zufällige Wartezeit bei den Blockerzeugern.

In der hier beschriebenen Implementierung nutzt Proof of Elapsed Time folgenden Ablauf:

1. Wartezeit generieren
2. Wartezeit abwarten
3. Block erstellen
4. Block veröffentlichen

Der Blockerzeuger, der die kürzeste Wartezeit generiert hat, wird somit den nächsten Block erstellen. Ein einfacher Angriff auf dieses Verfahren ließe sich durch einen böswilligen Blockerzeuger durchführen, indem er immer die kleinstmögliche Wartezeit generiert. Um dem entgegenzuwirken, wurde in der ursprünglichen Spezifikation von Proof of Elapsed Time [21] auf einen von Intel entwickelten Sicherheitschip zurückzugreifen. Dieser kann genutzt werden, um die Wartezeit manipulationssicher zu erzeugen und diese zu signieren. Blöcke sind dann nur gültig, wenn die Wartezeit ordnungsgemäß signiert ist und eingehalten wurde. Im Projekt wurde vorerst nur eine “naive” Variante von Proof of Elapsed Time umgesetzt, die nicht auf spezielle Hardware angewiesen ist.

Da bei Proof of Elapsed Time anstelle der Suche nach einem gewissen Hash ein Timer zum Einsatz kommt, ist es nicht notwendig, eine Schwierigkeit für Blöcke festzulegen oder diese anzupassen. Ebenso entfällt die Prüfung der Blockgültigkeit auf Hash-Basis, die Block-Hashes werden in diesem Konsensverfahren lediglich für Verweise auf den Vorgänger verwendet.

Die einzigen Funktionen, die für die Implementierung von Proof of Elapsed Time eine Rolle spielen, sind `next_block` und `block_work`. Die `next_block` Funktion ähnelt der, die auch in Proof of Work zum Einsatz kommt. Jedoch wird die wiederholte Neuberechnung des Block-Hashes durch eine Wartezeit ersetzt. Diese Wartezeit ist abhängig von der angestrebten Blockzeit (`TARGET_BLOCK_TIME_MS`). Sie wird zufällig und mit einer Abweichung von bis zu 20 % zur angestrebten Zeit angesetzt. Diese Zeit wird als Nächstes abgewartet. Dabei wird alle 100 Millisekunden überprüft, ob die Blockproduktion abgebrochen werden muss (s. Listing 3.12).

Nachdem die Wartezeit verstrichen ist, werden die neuen Transaktionen per `transactions_for_next_block` abgerufen und in den Block eingefügt. Hier ergibt sich ein weiterer Unterschied zu Proof of Work, bei dem die Transaktionen schon *während* der Blockvalidierung bekannt sein müssen, da diese den Block-Hash beeinflussen. Weil Proof of Elapsed Time nicht auf dem Block-Hash basiert, können die Transaktionen auch erst nach der Wartezeit abgerufen werden.

```
let mut rng = rand::thread_rng();
// +/- 20%
let sleep_ms = rng.gen_range(
    (TARGET_BLOCK_TIME_MS - (TARGET_BLOCK_TIME_MS / 20))
        ..(TARGET_BLOCK_TIME_MS + (TARGET_BLOCK_TIME_MS / 20)),
);

let target_wake_up = get_current_timestamp() + sleep_ms;

loop {
    thread::sleep(Duration::from_millis(100));
    if get_current_timestamp() > target_wake_up {
        break;
    }

    // check if:
    // - a rival block has been added
    // - a shutdown command was issued
}
```

Listing 3.12: Abwarten der Wartezeit in Proof of Elapsed Time

Der nächste Unterschied zu Proof of Work liegt in der Definition des Arbeitsbegriffes. Bei Proof of Work wurde die numerische Repräsentation der Schwierigkeit verwendet. Dies ist bei Proof of Elapsed Time nicht möglich. Stattdessen wurde die Block-Arbeit so definiert, dass Blöcke, deren zeitlicher Abstand zum Vorgänger nahe an der angestrebten Blockzeit liegt, eine hohe Punktzahl erreichen. Dafür wird der Quotient aus der tatsächlichen und der angestrebten Blockzeit gebildet und dann an eine Scoring-Funktion gegeben, die eine Punktzahl zwischen 0 und 1000 zurückgibt. Dabei werden möglichst geringere Abweichungen belohnt. Diese Funktion ist in Listing 3.13 als Code und in Abbildung 3.4 grafisch dargestellt. Mit einer derartigen Funktion können auch bei Proof of Elapsed Time Forks aufgelöst werden.

```
pub fn block_work(parent_header: &BlockHeader,
                          child_header: &BlockHeader) -> u128 {
    let target_block_time = TARGET_BLOCK_TIME_MS;
    let block_time = child_header.timestamp
                                          - parent_header.timestamp;

    let deviation =
        (target_block_time - block_time) /
        target_block_time * 1000;
    let deviation_abs = abs(deviation);

    let score = 1000 * pow(1.3, -(pow(deviation, 2)) / 100000);
    return score;
}
```

Listing 3.13: Definition des Arbeitsbegriffes in Proof of Elapsed Time

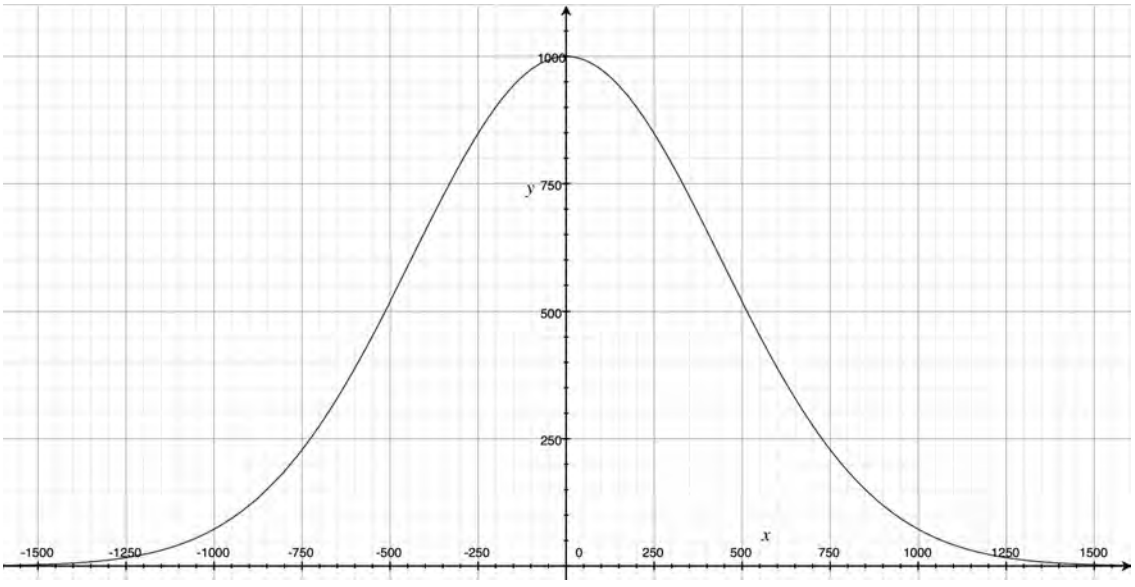


Abbildung 3.4: Scoringfunktion in Proof of Elapsed Time ($y = 10^3 \cdot 1,3^{-\frac{x^2}{10^5}}$)

3.6 Proof of Network Signatures

Das dritte Konsensverfahren im System nutzt Signaturen von speziellen Nodes, um Blöcke gültig zu machen und wird in dieser Arbeit als ”Proof of Network Signatures“ bezeichnet. Blöcke werden gültig, sobald sie von einer gewissen Anzahl von Validator-Nodes im Netzwerk signiert wurden. Damit kann beispielsweise sichergestellt werden, dass in auftretenden Netzwerk-Partitionen keine Blöcke erzeugt werden, falls in der Partition nicht eine gewisse Anzahl an Validatoren sind.

Das Verfahren funktioniert grundlegend wie Proof of Elapsed Time, jedoch mit dem Unterschied, dass Blöcke nicht direkt valide sind, sondern ihre Validität erst mit einer gewissen Anzahl von Signaturen aus dem Netzwerk erhalten. Dafür wurde zuerst eine Helferfunktion implementiert, die die Anzahl der gültigen Signaturen pro Block erfasst (s. Listing 3.14). Diese Funktion nimmt einen Block entgegen und überprüft jede seiner im `signatures` Feld gespeicherten Signaturen auf ihre Gültigkeit. Zusätzlich stellt diese Funktion sicher, dass einzelne Nodes den Block nicht mehrmals signiert haben. Mithilfe dieser Funktion wurden `block_is_valid` und `block_is_partially_valid` (s. Listing 3.15) implementiert.

```
fn count_valid_signatures(block: &Block) -> usize {
    let block_hash = block.get_digest();

    let mut valid_signature_ids = Vec::<String>::new();

    for signature in block.header.signatures {
        if CryptoLogic::raw_verify(&signature)
                        && signature.data == block_hash {
            let id = format!("{}-{}", signature.network_id,
                         signature.node_index);

            if !valid_signature_ids.contains(&id) {
                valid_signature_ids.push(id);
            }
        }
    }

    valid_signature_ids.len()
}
```

Listing 3.14: Funktion für das Zählen von Block-Signaturen in Proof of Network Signatures

```
pub fn block_is_valid(block: &Block) -> bool {
    count_valid_signatures(&block) >= SIGNATURE_COUNT_THRESHOLD
}

pub fn block_is_partially_valid(block: &Block) -> BlockValidity {
    if block_is_valid(block) {
        return BlockValidity::Valid;
    }

    if count_valid_signatures(&block)
                < SIGNATURE_COUNT_THRESHOLD {
        if get_current_timestamp() - block.header.timestamp >
                        (3 * TARGET_BLOCK_TIME_MS) {
            return BlockValidity::PartiallyValid(
                PartiallyValidBlockAction::Discard
            );
        } else {
            return BlockValidity::PartiallyValid(
                PartiallyValidBlockAction::
                        SignIfValidatorAndResend,
            );
        }
    }
}
```

Listing 3.15: Funktionen zur Prüfung der Validität von Blöcken in Proof of Network Signatures

Die definitive Prüfung des Blockes in `block_is_valid` wird durchgeführt, indem die Anzahl der gültigen Signaturen im Block mit der Konstante `SIGNATURE_COUNT_THRESHOLD` verglichen werden. Über diese Konstante lässt sich steuern, wie viele Signaturen ein Block minimal benötigt, um als valide angesehen zu werden. Damit kann das Konsensverfahren beispielsweise so konfiguriert werden, dass Blöcke nur erzeugt werden, falls mindestens 75 % der Validator-Nodes erreichbar sind. So können Forks verhindert werden, die durch einen physischen Netzwerk-Split zustande kommen.

Die teilweise Validitätsprüfung in `block_is_partially_valid` (s. Listing 3.15) beinhaltet in diesem Konsensverfahren auch eine Aktion, die mit teilweise gültigen Blöcken durchgeführt wird. Entält ein Block nicht die notwendige Anzahl von Signaturen, gibt die Funktion die Information zurück, dass der noch nicht valide Block signiert und erneut in das Netzwerk gesendet werden soll.

Wenn die Gültigkeitsprüfung für einen Block diesen Wert zurück gibt, werden Nodes, die den Validatorstatus haben, ihre Signatur dem Block hinzufügen und ihn zurück in

das Netzwerk schicken. In dieser Funktion kann auch Logik abgebildet werden, die beispielsweise eine Netzwerkbelastung verhindert, indem Blöcke nach bestimmten Kriterien verworfen werden. In dieser Implementierung werden Blöcke, die älter als die dreifache Blockzeit sind und noch nicht vom Netzwerk akzeptiert wurden, verworfen.

4 Vergleichssystem

Damit sich die Ergebnisse des Versuches auf das System von safe-UR-chain übertragen lassen, wird der Versuch auch direkt im System durchgeführt. Die ermittelten Metriken können im Projekt also direkt weiterverwendet werden. Ein weiterer Vorteil der Verwendung dieses Systems besteht darin, dass der gesamte Quellcode vorliegt. Dadurch müssen keine Drittsysteme, wie etwa zur Datenerfassung, verwendet werden. Zusätzlich können die Konsensverfahren direkt im System implementiert und gemeinsam mit dem restlichen Projekt kompiliert werden, was dafür sorgt, dass mögliche Unterschiede zwischen den Verfahren (z.B. verschieden effizient kompilierte Binärdateien) minimiert werden.

4.1 Erfassung der Metriken

Um den Versuchsablauf zu automatisieren, wurde die Node-Software dahingehend verändert, dass ein Prozess anstatt von nur einer Node nun mehrere Nodes instanziieren kann. Dafür wurde die bisherige Logik, die den Start einer Node kontrolliert, in eine Funktion ausgelagert, die den Node-Prozess innerhalb eines Threads startet, dabei wird der Node eine Datenstruktur übergeben, die ihre Konfigurationsdatei (s. Sektion 2.9) ersetzt. Zusätzlich zu einem Zeiger auf die Node-Instanz selbst, werden Referenzen auf einige Datenstrukturen der Node freigegeben. Diese werden in einem `NodeHandle` zusammengefasst (s. Listing 4.1).

```
fn start_node_with_config(config: Config) -> NodeHandle {
    // node startup logic ...
}

struct NodeHandle {
    thread: JoinHandle<()>,
    shutdown: Arc<RwLock<bool>>,
    telemetry: Arc<Mutex<Telemetry>>,
    blockchain: Arc<Mutex<Blockchain>>,
}
```

Listing 4.1: Startfunktion für Nodes und Node Handle Datenstruktur

Durch die Trennung von Hauptprozess und dem Start von Nodes kann ein Kontrollprozess implementiert werden, der viele Nodes starten und ihre Funktion überwachen kann. Dieser Prozess bildet das Grundgerüst für die Durchführung des Versuches. Jeder Versuch folgt dem gleichen Ablauf und kann über die `TestCase` Datenstruktur (s. Listing 4.2) konfiguriert werden.

```
struct TestCase {
    full_nodes: usize,
    validators: usize,
    archivists: usize,
    ingest_nodes: usize,
    min_connections_per_node: usize,
    transaction_target: usize,
    time_between_transactions_millis: u64,
    timeout_millis: u128,
    runs: usize,
}
```

Listing 4.2: Test Case Datenstruktur

Jeder `TestCase` beschreibt die grundlegende Netzwerkstruktur für den Test, sowie ein Ziel für die endgültige Transaktionsanzahl und die Rate, mit der diese erstellt werden sollen. Zusätzlich kann angegeben werden, nach welcher Zeit der Test abgebrochen werden soll, falls das Transaktionsziel nicht von allen Nodes erreicht wird. Der eigentliche Testablauf ist für jede Konfiguration gleich.

Im ersten Schritt werden die für den Start der Nodes notwendigen Konfigurationen erzeugt. Dabei werden Instanzen gemäß der Node-Typen aus Sektion 2.2 erzeugt. Listing 4.3 zeigt die Erzeugung der Konfiguration für Full Nodes. Die Konfigurationen der anderen Node-Typen werden auf die gleiche Art und Weise erstellt. Die Node-Typen im Test unterscheiden sich anhand der Einstellungen, die in Tabelle 4.1 aufgelistet sind. Innerhalb der Tests haben Thin-Nodes die Aufgabe, Daten aufzunehmen, weswegen sie auch als “Ingest-Nodes” bezeichnet werden.

	Datenaufnahme	Erzeugung von Blöcken	Transaktionsfilter
Thin Node	ja	nein	eigene
Validator Node	nein	ja	keine
Archiv Node	nein	nein	alle (*)
Full Node	nein	ja	alle(*)

Tabelle 4.1: Konfigurationen der einzelnen Node Typen

```
let mut node_count = 0;
let mut port = 10688;
let seed_port = port;
let mut configs = vec![];

for _ in 0..test_case.full_nodes {
    configs.push(Config {
        network: String::from("hsmw"),
```

```
        index: node_count,
        ingester: true,
        validator: true,
        local: true,
        listener_ip: Some(get_local_ip()),
        listener_port: Some(port),
        known_nodes: None,
        store_transactions_from: vec![String::from("*")],
        ingest_dir: None,
        base_port: seed_port,
    });

    node_count += 1;
    port += 1;
}

// do the same for the other node types ...
```

Listing 4.3: Erstellung der Full Node Konfigurationen für einen Testdurchlauf

Nachdem alle Node-Konfigurationen erzeugt wurden, werden die Nodes einzeln instanziiert. Hierbei werden die Nodes nicht gleichzeitig, sondern zeitlich etwas versetzt gestartet. Damit soll verhindert werden, dass Aufgaben, die Nodes in regelmäßigen Intervallen ausführen (z.B. die Selbstwartung des Netzwerkmoduls aller 25 Sekunden), nicht gleichzeitig von allen Nodes in der Simulation ausgeführt werden, was das Testsystem kurzzeitig auslasten kann. Die `start_node_with_config` Funktion gibt für jede gestartete Node ein `NodeHandle` zurück. Diese Handles werden für die weitere Verwendung im Test abgespeichert (s. Listing 4.4).

```
let mut node_handles: Vec<NodeHandle> = vec![];

for config in configs {
    node_handles.push(start_node_with_config(config));
    thread::sleep(Duration::from_millis(900));
}
```

Listing 4.4: Start der Nodes

Nachdem alle Nodes gestartet wurden, wartet der Kontrollprozess den Aufbau des Netzwerkes ab. Dafür wird das im `NodeHandle` enthaltene `telemetry` Feld verwendet (s. Listing 4.5). Es enthält eine Referenz auf die internen Netzwerk-Telemetriedaten der jeweiligen Node. Darin ist unter anderem eine Kopie der Liste mit den der Node bekannten anderen Nodes im Netzwerk enthalten. Der Kontrollprozess überwacht den Netzwerkaufbau, indem er die Anzahl der jeweils bekannten Netzwerkteilnehmer für je-

de Node feststellt. Wenn die minimale Anzahl der bekannten Peers erreicht ist, kann der Netzwerkaufbau als abgeschlossen angesehen werden (s. Listing 4.6).

```
pub struct Telemetry {
    pub queued_to_send: usize,
    pub peers: Vec<Peer>,
    pub global_peer_lists: Vec<TelemetryPeerList>,
}
```

Listing 4.5: Telemetrie-Datenstruktur

```
loop {
    thread::sleep(Duration::from_millis(1000));
    println!("Waiting␣for␣network...");
    let mut done = true;
    for node_handle in &node_handles {
        let telemetry = node_handle.telemetry;
        let connections = telemetry.peers.len();
        if connections < test_case.min_connections_per_node {
            done = false;
            break;
        }
    }

    if get_current_timestamp() - started_network_bootstrapping
        > test_case.timeout_millis
    {
        test_aborted = true;
        break;
    }
    if done {
        println!("Network␣online.");
        break;
    }
}
```

Listing 4.6: Abwarten des Netzwerkaufbaus

Mit dem bestehenden Netzwerk kann nun der eigentliche Versuch starten. Für diesen werden Datensätze mit einer gewissen Frequenz erzeugt und es wird abgewartet, bis eine festgelegte Anzahl an Transaktionen in das Netzwerk aufgenommen wurden.

Dafür wird für jede Thin-Node ein Thread gestartet, der Datensätze erzeugt und diese als Dateien in den jeweiligen Ordner für die Datenaufnahme (`ingest_dir` aus der Node-Konfiguration) ablegt. Nach jedem erzeugten Datensatz wird die Zeit

`time_between_transactions_millis` abgewartet. Dieser Vorgang wird so oft wiederholt, wie in der Testkonfiguration in `transactions_per_node` angegeben (s. Listing 4.7).

```
let mut tx_generators = vec![];
let mut ingest_id = 0;
for _ in test_case.validators..
             (test_case.validators + test_case.ingest_nodes) {
    if test_aborted {
        break;
    }

    let tx_generator = std::thread::spawn(move || {
        let mut tail = String::from("");
        let mut i = 0;

        for _ in 0..31 {
            tail.push_str("0000000|0000000|0000000|0000000\n");
        }

        loop {
            let mut new = File::create(format!(
                "./ingest-{}/{}{}.psv",
                ingest_id + ingest_start_index,
                &get_current_timestamp(),
                i,
            ));
            let main_data = format!(
                "{}-{}",
                ingest_id + ingest_start_index,
                i + ingest_start_index,
            );
            let contents = format!(
                "{:0>width$}|0000000|0000000|0000000\n{}",
                main_data,
                tail,
                width = 7
            );
            match new.write_all(contents.as_bytes()) {
                Ok(_) => {}
                Err(e) => {
                    println!("{:#?}", e);
                }
```

```
            }
            thread::sleep(Duration::from_millis(
                test_case.time_between_transactions_millis,
            ));
            i += 1;
        }
    });
    ingest_id += 1;
    tx_generators.push(tx_generator);
}
```

Listing 4.7: Erzeugung von Testdaten

Im nächsten Schritt werden die Blöcke mit den erzeugten Datensätzen abgewartet. Dadurch, dass die Generatoren für die Datensätze in eigenständigen Threads gestartet werden, kann der Kontrollprozess auch während der Datenerzeugung die Blockchain überwachen. Um festzustellen, wann ausreichend Transaktionen in die Blockchain aufgenommen wurden, fragt der Kontrollprozess regelmäßig die Anzahl der Transaktionen innerhalb der jeweils längsten Blockchain aller Nodes ab. Sobald alle Nodes die gewünschte Anzahl von Transaktionen in ihrer Blockchain haben, kann der Test beendet werden (s. Listing 4.8).

```
loop {
    thread::sleep(Duration::from_millis(1000));
    println!(
        "Waiting␣for␣{}␣transactions...",
        test_case.transaction_target
    );
    let mut done = false;
    if get_current_timestamp() - test_start
       > test_case.timeout_millis {
        test_aborted = true;
        break;
    }
    for node_handle in &node_handles {
        if let Ok(blockchain) = node_handle.blockchain {
            if blockchain
                .block_tree
                .longest_chain
                .last()
                .transaction_count
                >= test_case.transaction_target
            {
                done = true;
```

```
                    break;
                }
            };
        }
        if done {
            break;
        }
    }
```

Listing 4.8: Abwarten der Transaktionen

Nachdem die Transaktionen in die Blockchain aufgenommen wurden, werden alle Nodes zum Herunterfahren angewiesen. Dafür wird die `shutdown` Referenz der `NodeHandles` auf `true` gesetzt. Sobald alle Nodes erfolgreich heruntergefahren sind, beginnt der Kontrollprozess mit der Analyse der Blockchain. Dabei werden alle Transaktionen in allen Blöcken durchlaufen und ihre Latenzen werden ermittelt, indem der Transaktions-Zeitstempel mit dem Zeitpunkt verglichen wird, an dem der Block zur Blockchain hinzugefügt wurde. Danach wird der Durchschnitt der Latenzen berechnet. Mit den Zeitstempeln der ersten Transaktion, sowie dem des letzen Blockes, kann der Testzeitraum erfasst werden. Leere Blöcke am Anfang der Blockchain werden dabei ignoriert, da diese während des Netzwerkaufbaus erzeugt wurden. Mit dieser Zeitangabe kann dann der Transaktionsdurchsatz und der Datendurchsatz errechnet werden (s. Listing 4.9).

```
let blocks = blockchain.blocks_in_range(0, blockchain.len());
let mut prev_block_timestamp = 0;

for block in blocks {
    let mut block_tx_counter = 0;
    let mut block_latencies_sum = 0;
    let mut block_payload_bytes = 0;

    block_appended = block.header.time_appended;

    for leaf in block.data.leafs {
        if leaf.contents == LeafContents::Transaction {
            if let Ok(transaction) =
                bincode::deserialize::<Transaction>(&leaf.payload)
            {
                tx_ids.insert(transaction.id);
                tx_counter += 1;
                block_tx_counter += 1;

                sum_of_tx_latencies +=
                    block_appended - transaction.meta.timestamp;
```

```
                block_latencies_sum +=
                    block_appended - transaction.meta.timestamp;

                block_payload_bytes += transaction.payload.len();

                if transaction.meta.timestamp
                   < first_transaction_timestamp {
                    first_transaction_timestamp =
                        transaction.meta.timestamp;
                }
            }
        }
    }

    let avg_block_latency = block_latencies_sum
        / block_tx_counter;

    let block_time = block.header.timestamp
        - prev_block_timestamp;

    prev_block_timestamp = block.header.timestamp;
    last_block_timestamp = block.header.timestamp;

    let stats = BlockStats {
        height: block.header.height,
        block_time,
        tx_count: block_tx_counter,
        overall_tx_count: tx_counter,
        avg_latency: avg_block_latency as usize,
        payload_bytes: block_payload_bytes,
    };

    block_stats.push(stats);
}
```

Listing 4.9: Erfassung der Metriken

Sobald die gewünschte Anzahl von Transaktionen mindestens in der Blockchain einer der Nodes erfasst wurde, werden die Daten-Generatoren gestoppt und der Kontrollprozess wartet ab, bis alle lokalen Blockchains mindestens die gewünschte Anzahl an Transaktionen enthalten. Nachdem diese Bedingung erfüllt ist, wird die aktuelle Zeit als Testende gespeichert und ein Testprotokoll wird erzeugt.

Dieses Protokoll beinhaltet alle Daten, die im Nachgang für den Vergleich von Konsensverfahren oder von verschiedenen Konfigurationen benötigt werden. Dazu zählen Softwarekonstanten, der aktuelle `TestCase`, die errechneten Metriken und Daten darüber, ob der Test abgebrochen wurde. Mehrere dieser Protokolle lassen sich zu einer Tabelle zusammenfassen, an der weitere Analysen durchgeführt werden können (s. Listing 4.10). Es werden ebenso die Metriken der einzelnen Blöcke aufgezeichnet.

```
{
  "consensus_algorithm": "Proof␣of␣Elapsed␣Time",
  "const_target_used_peers": 6,
  "const_wait_time_to_check_priority_queue": 500,
  "const_wait_time_to_maintenance": 15000,
  "const_wait_time_to_broadcast_peer_list": 15000,
  "const_wait_time_to_shuffle_peer_list": 25000,
  "const_num_workers": 4,
  "const_target_block_time_ms": 5000,
  "const_look_back_for_diff": 5,
  "test_case_full_nodes": 0,
  "test_case_validators": 2,
  "test_case_archivists": 0,
  "test_case_ingest_nodes": 5,
  "test_case_min_connections_per_node": 5,
  "test_case_transaction_target": 30000,
  "test_case_time_between_transactions_millis": 100,
  "test_case_timeout_millis": 900000,
  "results_tx_count": 30063,
  "results_test_time": 530901,
  "results_blockchain_time": 528859,
  "results_tps": 56,
  "results_avg_latency": 23406,
  "results_test_aborted": false,
  "cycle": "1/5"
}
```

Listing 4.10: Beispielhaftes Testprotokoll (Auszug)

Der gesamte Testprozess kann beliebig oft wiederholt werden, wenn dem Programm eine Liste mit `TestCase`s übergeben wird. Damit kann der gleiche Test mehrfach durchgeführt werden, um Durchschnitte zu ermitteln. Es können aber auch verschiedene Tests nacheinander ausgeführt werden.

4.2 Simulationsumgebung

Um sicherzustellen, dass die Versuche immer unter gleichen Bedingungen durchgeführt werden können, wurde als Simulationsumgebung ein Cloud-Server mit dedizierten CPU-Ressourcen gewählt. Dieser hat folgende technische Daten:

- CPU: 32 Kerne @ 2,1GHz[6]
- RAM: 128GB
- Speicher: 600GB NVMe SSD
- Betriebssystem: Ubuntu 20.04

Auf dem Server wurde die Software installiert, die für die Kompilierung und den Betrieb des Systems notwendig ist (s. Listing 4.11).

```
$ apt install build-essential clang cmake libssl-dev pkg-config
    ↪ net-tools

$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Listing 4.11: Befehle zum Vorbereiten des Servers

Zusätzlich wurde das Programm `screen` installiert. Mit diesem Programm kann eine Terminal-Session auf dem Server erstellt werden, die unabhängig von einem `ssh` Zugriff weiterlaufen kann. Alle Tests wurden in derartigen Sessions durchgeführt, um die Tests gegen eine Störung des Testvorgangs durch eventuelle Verbindungsabbrüche eines `ssh` Zugriffs zu schützen.

4.3 Definition von Testfällen

Um die Leistung der Konsensverfahren miteinander zu vergleichen, werden aufeinanderfolgende Belastungstests mit einem immer höheren Datenaufkommen für jedes Konsensverfahren durchgeführt. Dabei werden Daten über den Durchsatz und die Latenz erhoben und anschließend verglichen. Um diesen Versuchsablauf zu automatisieren, wurden `TestCase`s definiert, in denen neue Datensätze jeweils mit einer anderen Frequenz erzeugt werden (s. Listing 4.12).

```
TestCase {
    full_nodes: 0,
    validators: 2,
    archivists: 0,
```

[6] Intel Xeon Processor (Skylake, IBRS)

```
    ingest_nodes: 5,
    min_connections_per_node: 6,
    transaction_target: 50000,
    time_between_transactions_millis: ---,
    timeout_millis: 900000,
    runs: 5,
},
```

Listing 4.12: Beispielhafter Testfall der Versuchsreihe

Das Netzwerk, dass für die Versuche aufgebaut werden soll, besteht aus sieben Nodes, die sich jeweils mit allen anderen Nodes im Netzwerk verbinden. Zwei dieser Nodes sind für die Erzeugung neuer Blöcke verantwortlich und die fünf anderen Nodes nehmen die Daten auf, die als Transaktionen in das Netzwerk geschickt werden sollen. Diese Anzahl der Nodes wurde gewählt, da sie sich bereits bei der Entwicklung des Testsystems bewährt hat. Bei der Verwendung von mehr Nodes konnten Tests besonders bei höheren Datenraten nicht mehr zuverlässig durchgeführt werden, da die maximale Anzahl von gleichzeitig geöffneten Dateien überschritten werden konnte (was zum Absturz des Systems führt).

Als Transaktionsziel für die meisten Testfälle wurden `50.000` Transaktionen festsgelegt. Dieser Wert wurde so gewählt, dass insbesondere die Tests mit geringem Datenaufkommen nicht zu viel Zeit in Anspruch nehmen. So wird beispielsweise ein Test mit einem Datenaufkommen von 100 Datensätzen pro Sekunde im Netzwerk mindestens 8 Minuten und 20 Sekunden laufen. Dazu kommt die Zeit für den Netzwerkaufbau.

Jeder Testfall wird fünffach durchgeführt, um in der Auswertung mit Durchschnittswerten je Testfall arbeiten zu können. Hierdurch soll verhindert werden, dass zufällige Ereignisse, wie die automatische Restrukturierung des Netzwerkes oder stark vom Durchschnitt abweichende Blockzeiten in Proof of Work, die Ergebnisse beeinflussen. Sollte ein Test nach 15 Minuten (`900000 ms`) nicht das Transaktionsziel erreicht haben, wird er abgebrochen. Die Testfälle der aufgestellten Versuchsreihe sind in Tabelle 4.2 aufgelistet.

Zeit zwischen Datensätzen	Datensätze pro Sekunde	Transaktionsziel
1000 ms	5	500
100 ms	50	5.000
50 ms	100	20.000
40 ms	125	50.000
30 ms	166,67	50.000
20 ms	250	50.000
15 ms	333,33	50.000
10 ms	500	50.000

Tabelle 4.2: Testfälle in der Versuchsreihe

Die erzeugten Datensätze sind je 1 KiB (1024 B) groß. Diese Größe richtet sich nach den Datensätzen, die das System in der realen Anwendung aufnehmen soll. Die Blockzeit soll fünf Sekunden betragen und die maximale Blockgröße ist auf 10.000 Transaktionen begrenzt.

5 Versuchsdurchführung

Für den Versuch wurde der Server wie bereits beschrieben aufgesetzt und die Software wurde kompiliert. Danach wurde die Versuchsreihe zuerst für Proof of Work, danach für Proof of Elapsed Time und letztendlich für Proof of Network Signatures durchgeführt. Insgesamt liefen 120 Tests über den Zeitraum von ca. einem Tag. Vor Beginn der Testreihe für ein neues Konsensverfahren wurde der Server jeweils neu gestartet.

5.1 Ergebnisse

Die folgenden drei Seiten zeigen jeweils die Daten, die in den Testreihen der einzelnen Konsensverfahren erhoben wurden. Diese sind je Seite in vier Diagrammen dargestellt. Die `x`-Achsen in allen Diagrammen zeigen die pro Testfall erzeugten Datensätze pro Sekunde (s. Tabelle 4.2). Das erste Diagramm (blauer Graph) stellt den gemessenen Transaktionsdurchsatz in *Transaktionen pro Sekunde* dar. Aus diesem Transaktionsdurchsatz lässt sich auch der Datendurchsatz errechnen, da jede Transaktion 1 KiB Nutzdaten enthält. Das zweite Diagramm (grüner Graph) zeigt die gemessene Transaktionslatenz in Millisekunden und das dritte Diagramm (graue Balken) stellt die prozentuale Abweichung des Transaktionsdurchsatzes (`y`-Achse) vom theoretisch möglichen Maximalwert (`x`-Achse) dar. Letztendlich zeigt das vierte Diagramm (roter Graph) die durchschnittliche Blockzeit, die während der Versuchsdurchführung gemessen wurde. Die gezeigten Werte wurden ermittelt, indem der Durchschnitt der fünf Wiederholungen je `TestCase` berechnet wurde und die Diagramme sind zur besseren Lesbarkeit jeweils auf den gleichen Wertebereich skaliert.

Proof of Work

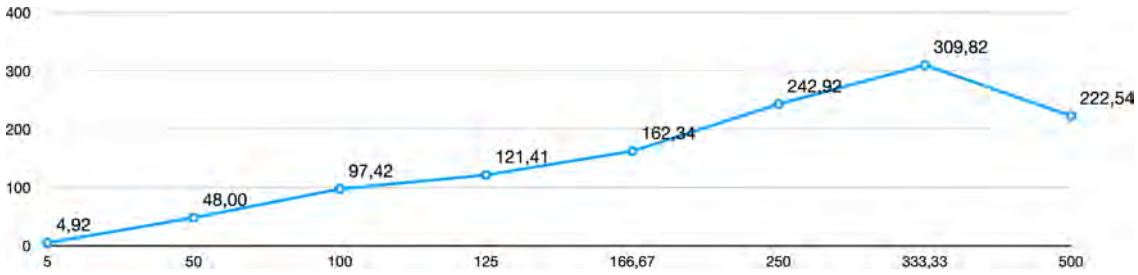


Abbildung 5.1: PoW: Transaktionsdurchsatz in Transaktionen pro Sekunde

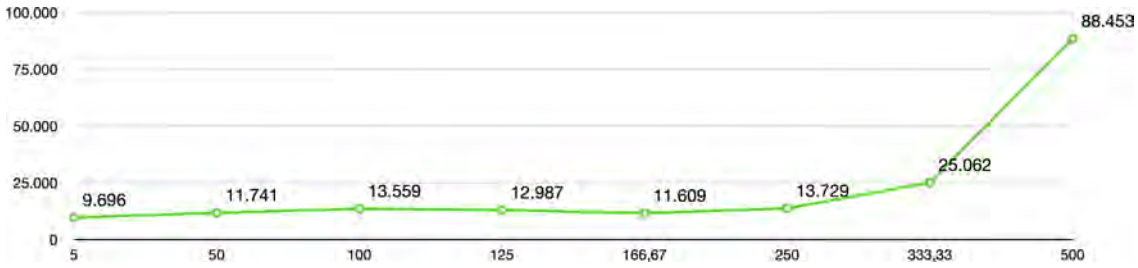


Abbildung 5.2: PoW: Latenz in Millisekunden

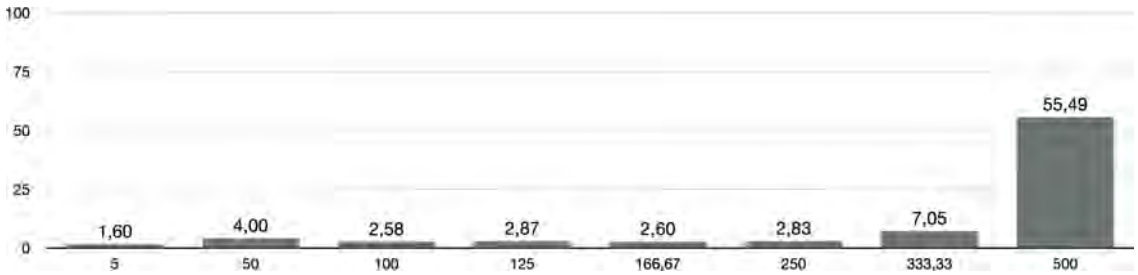


Abbildung 5.3: PoW: Abweichung vom theoretisch maximalen Durchsatz in Prozent

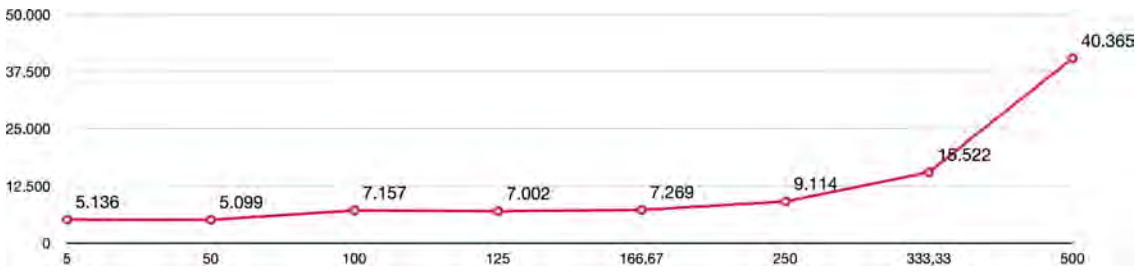


Abbildung 5.4: PoW: Blockzeiten in Millisekunden

Proof of Elapsed Time

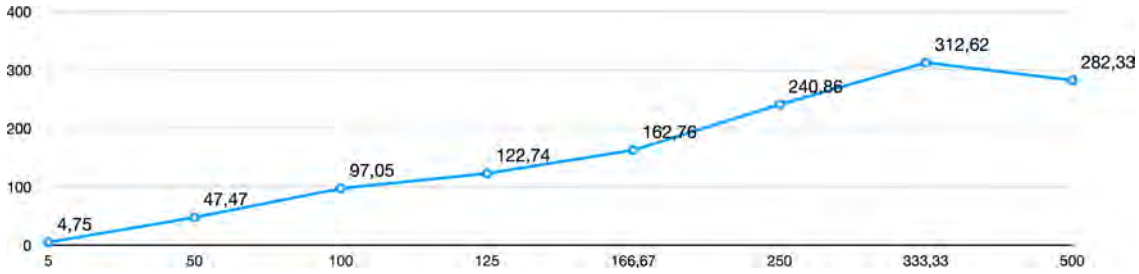


Abbildung 5.5: PoET: Transaktionsdurchsatz in Transaktionen pro Sekunde



Abbildung 5.6: PoET: Latenz in Millisekunden

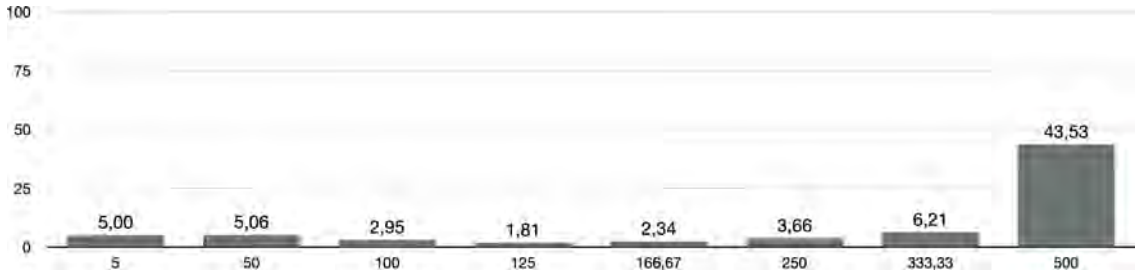


Abbildung 5.7: PoET: Abweichung vom theoretisch maximalen Durchsatz in Prozent

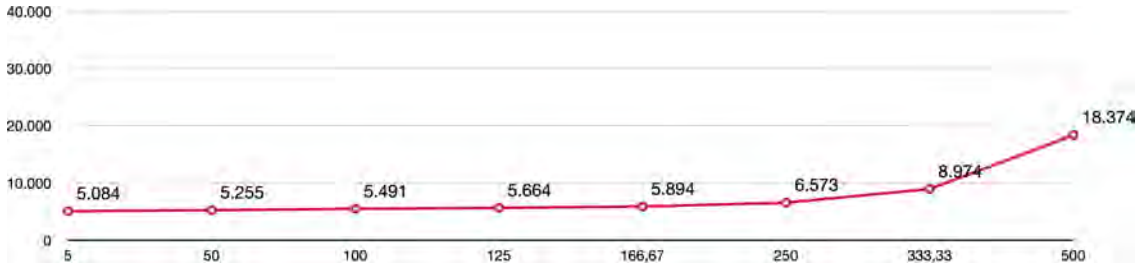


Abbildung 5.8: PoET: Blockzeiten in Millisekunden

Proof of Network Signatures

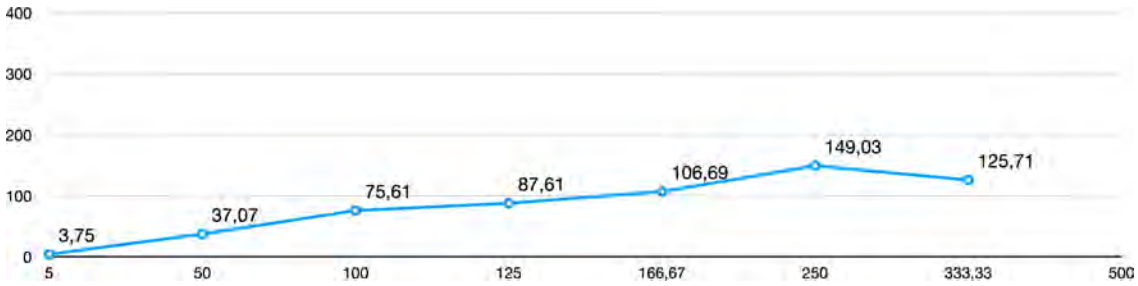


Abbildung 5.9: PoNS: Transaktionsdurchsatz in Transaktionen pro Sekunde

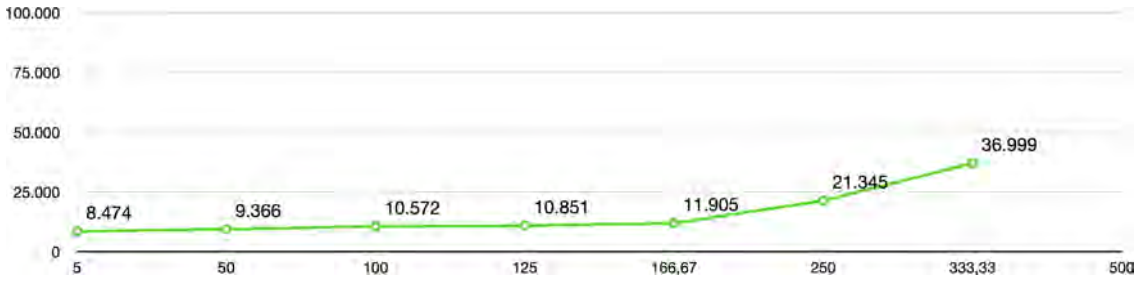


Abbildung 5.10: PoNS: Latenz in Millisekunden

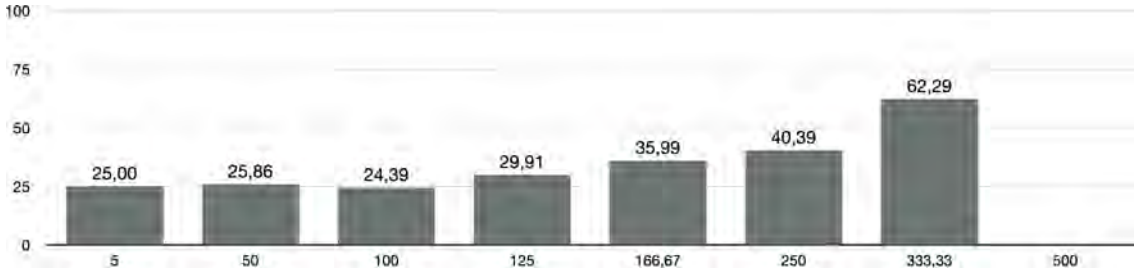


Abbildung 5.11: PoNS: Abweichung vom theoretisch maximalen Durchsatz in Prozent

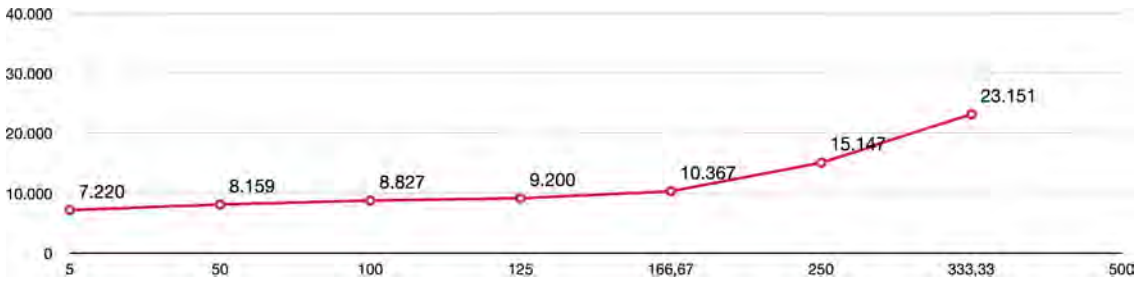


Abbildung 5.12: PoNS: Blockzeiten in Millisekunden

Die Daten für den jeweils letzen `TestCase` mit 500 erzeugten Datensätzen pro Sekunde entfallen in diesen Diagrammen, da das Transaktionsziel von 50.000 nach 15 Minuten nicht erreicht wurde.

5.2 Auswertung

Die erhobenen Daten zeigen eine starke Ähnlichkeit zwischen Proof of Work und Proof of Elapsed Time. Beide Konsensverfahren erreichen vergleichbare Durchsatzwerte, jedoch erreicht Proof of Elapsed Time durchschnittlich geringere Latenzen. Beide Konsensverfahren haben ihren höchsten Durchsatz im vorletzten `TestCase`, danach ist bei den Verfahren ein Abfall des Durchsatzes sowie ein deutlicher Anstieg der Latenzen und Blockzeiten zu beobachten.

Die Ähnlichkeit der Metriken bei diesen beiden Konsensverfahren ist damit zu erklären, dass Proof of Elapsed Time entwickelt wurde, um die Eigenschaften von Proof of Work zu imitieren, ohne jedoch die gleiche Rechenleistung in Anspruch zu nehmen. Der Vergleich dieser Verfahren im System zeigt, dass sie für geringere Transaktionsaufkommen ähnliche Leistungskennzahlen aufweisen, aber Proof of Elapsed Time besser mit einer hohen Last durch viele neue Datensätze umgehen kann. Dieser Unterschied kann damit erklärt werden, dass während der Erstellung von Blöcken bei Proof of Work mehrmals Vorgänge ablaufen, die bei Proof of Elapsed Time nur einfach ausgeführt werden.

Das dritte getestete Konsensverfahren (Proof of Network Signatures) hat durchgehend niedrigere Leistungsmetriken. Bereits bei geringen Datenaufkommen fällt der Transaktionsdurchsatz niedriger als bei den anderen beiden Konsensverfahren aus. Zudem wird der maximale Transaktionsdurchsatz in diesem Konsensverfahren bereits bei dem `TestCase` mit 250 erzeugten Datensätzen pro Sekunde erreicht und fällt danach ab. Dieses Konsensverfahren ist außerdem das einzige, bei dem der Test mit 500 erzeugten Datensätzen pro Sekunde zum Versagen des Netzwerkes geführt hat. Dieses Ergebnis kann mit der erhöhten Netzwerklast, die durch das erneute Senden von teilweise ungültigen Blöcken auftritt, erklärt werden.

Die durchgeführten Testreihen zeigen relativ stabile Latenzen und Blockzeiten für die jeweils ersten Tests. Jedoch steigen die Blockzeiten und Latenzen mit höheren Datenraten. Dies kann damit erklärt werden, dass das Zusammensetzen von Transaktionen zu Blöcken eine gewisse Zeit in Anspruch nimmt. Die benötigte Zeit steigt mit den im Block enthaltenen Transaktionen, was in den Tests mit hoher Datenrate dazu führt, dass die Blockzeiten steigen. Dies führt wiederum dazu, dass Transaktionen mehrere Blockzeiten im Transaktionspool verbringen, was einen Anstieg der Latenz zur Folge hat. Es kann davon ausgegangen werden, dass sich die Blockzeit in den jeweils letzten Tests durch die Anzahl der enthaltenen Transaktionen signifikant erhöht hat. Dieser Effekt wird noch dadurch verstärkt, dass Blöcke in den durchgeführten Tests bis zu 10.000 Transaktionen aufnehmen können. Die Blockzeit steigt dann dadurch, dass aufeinanderfolgende Blöcke immer mehr Zeit für ihre Erzeugung in Anspruch nehmen. In dieser Zeit laufen wiederum neue Transaktionen auf, die den nächsten Block noch größer werden lassen. Anhand der erhobenen Daten kann abgeschätzt werden, dass die maximale Anzahl für aufgenommene Datensätze pro Sekunde im beschriebenen System

zwischen 250 und 333 liegt. Dementsprechend sollte eine Verringerung der maximalen Blockgröße zu stabileren Blockzeiten führen.

5.3 Betrachtung

Die Versuchsreihe wurde so konzipiert, dass zwischen den einzelnen Testfällen und Konsensverfahren keine Änderungen an den äußeren Umständen zustande kommen. Dazu zählt die Vorkehrung, dass sämtliche hier dargestellten Versuche auf einem einzelnen Server ausgeführt wurden. Dies hat den Vorteil, dass das lokale Netzwerk nicht von anderen Netzwerkteilnehmern beeinflusst werden kann, bringt aber auch den Nachteil mit sich, dass alle Nodes auf geteilte Ressourcen angewiesen sind, was die maximale Netzwerkgröße auf die verwendeten sieben Nodes beschränkt. Für größer angelegte Tests sollte dementsprechend entweder ein leistungsfähigerer Server, oder gar ein isoliertes Netzwerk mit physisch separaten Nodes verwendet werden.

Mögliche Fehlerquellen liegen in der Länge der einzelnen Tests, sowie der Anzahl deren Wiederholungen. In der Versuchsreihe waren die Tests auf eine maximale Laufzeit von 15 Minuten beschränkt. Selbst mit dieser relativ kurzen Testzeit hat die Ausführung der gesamten Versuchsreihe für alle Konsensverfahren ungefähr einen Tag angedauert. Es kann vermutet werden, dass die Versuche mit einer längeren Ausführungszeit andere Ergebnisse erzielt hätten, bspw. die Annäherung des Durchsatzes an den theoretischen Maximalwert oder der Fehlschlag nach einigen Stunden in Betrieb.

Weiterhin kann die Frage gestellt werden, ob die fünffache Wiederholung von Tests zum Erhalt von Durchschnittswerten nicht diese Werte selbst beeinflusst. Es ist z.B. denkbar, dass einige Prozesse aus vorher durchgeführten Tests auf dem Server weiterlaufen und die Leistung der nachfolgenden Tests beeinflussen. Andererseits könnten die fünf Wiederholungen je `TestCase` nicht ausreichen, um einen guten Durchschnitt zu bilden. Um diesen Fehler auszuschließen, wurde ein weiterer `TestCase` zur Validierung entworfen, der 30-Mal wiederholt wird. In diesem `TestCase` werden 100 Datensätze pro Sekunde erzeugt (s. Listing 5.1). Die Abbildungen 5.13, 5.14 und 5.15 zeigen die Transaktionsraten (in TPS) und Latenzen (in Sekunden), die in diesen Versuchen erreicht wurde. Die `x`-Achse zeigt die Versuchsnummern.

```
TestCase {
    full_nodes: 0,
    validators: 2,
    archivists: 0,
    ingest_nodes: 5,
    min_connections_per_node: 6,
    transaction_target: 20000,
    time_between_transactions_millis: 50,
```

```
    timeout_millis: 900000,
    runs: 30,
},
```

Listing 5.1: Testprotokoll für die Wiederholbarkeit

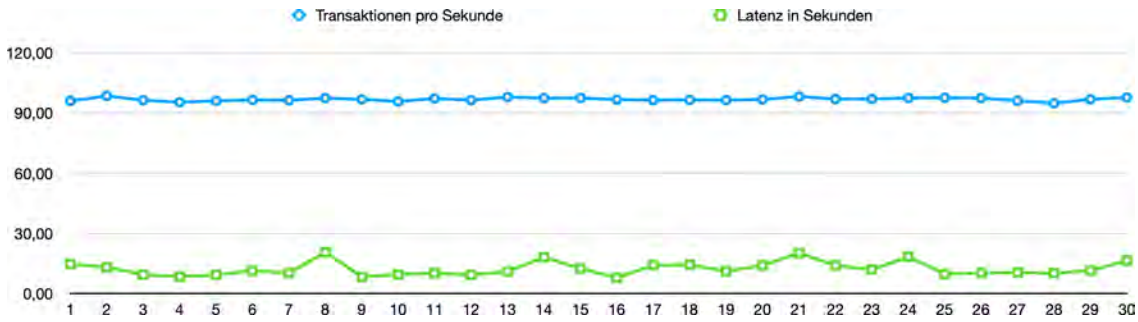


Abbildung 5.13: Wiederholbarkeit von PoW-Tests

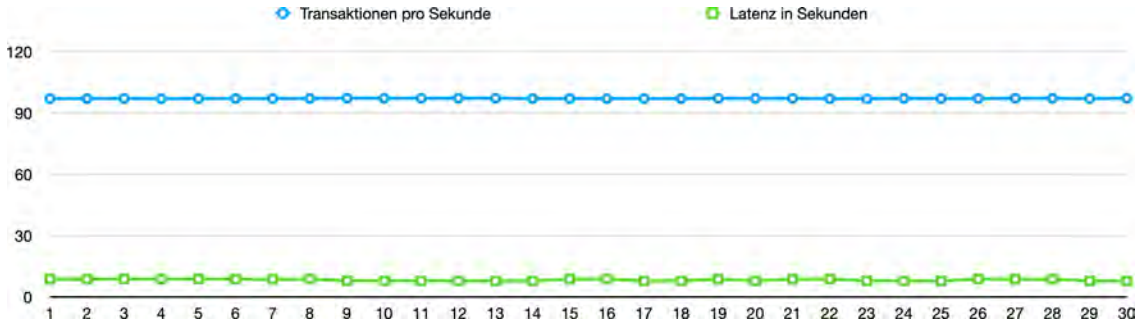


Abbildung 5.14: Wiederholbarkeit von PoET-Tests

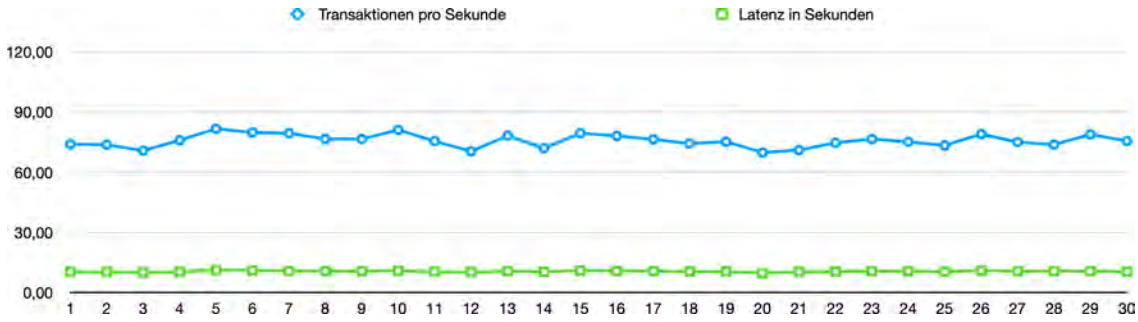


Abbildung 5.15: Wiederholbarkeit von PoNS-Tests

In den je 30 Versuchen dieser Validierung ist kein Trend einer sich verschlechternden Leistung erkennbar. Zudem weichen die Durchschnittswerte für Latenz und Durchsatz nur sehr geringfügig von den im der eigentlichen Versuchsreihe ermittelten Wert mit der gleichen Datenerzeugungsrate ab. Es kann also davon ausgegangen werden, dass die fünf Wiederholungen innerhalb der Versuchsreihe ein akkurates Bild der Metriken zulassen.

Letztendlich ist die Annahme, dass geringere Blockgrößen zu stabileren Blockzeiten führen, zu prüfen. Dafür wurden zwei Tests mit einer Datenrate von 500 Datensätzen pro Sekunde und einem Transaktionsziel von 100.000 durchgeführt. Bei einem der Tests wurde die Blockgröße auf maximal 1.500 Transaktionen herabgesetzt, bei dem anderen wurde sie bei 10.000 belassen. Die in diesen Tests gemessenen Blockzeiten zeigen einen stetigen Anstieg der Blockzeit bei einer maximalen Blockgröße von maximal 10.000 Transaktionen (s. Abb. 5.16) und konstante Blockzeiten bei einer maximalen Blockgröße von 1.500 Transaktionen 5.17) (jeweils nach konstant geringen Blockzeiten während des Netzwerkaufbaus). Diese Beobachtung belegt die Annahme, dass geringere Blockgrößen stabilere Blockzeiten zur Folge haben.

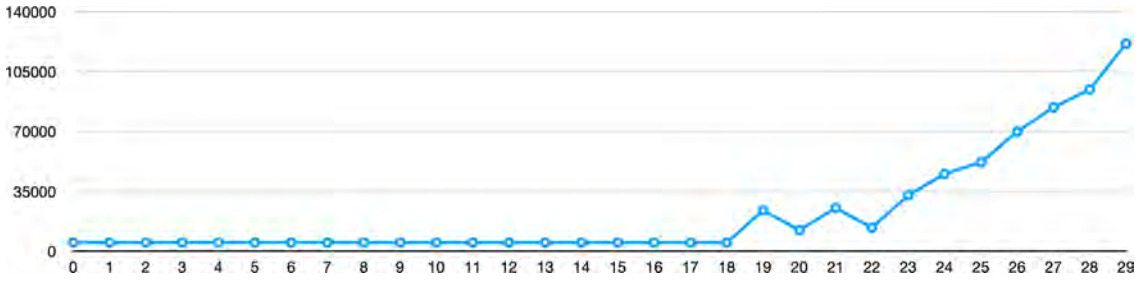


Abbildung 5.16: Blockzeiten im Test mit maximal 10.000 Transaktionen pro Block

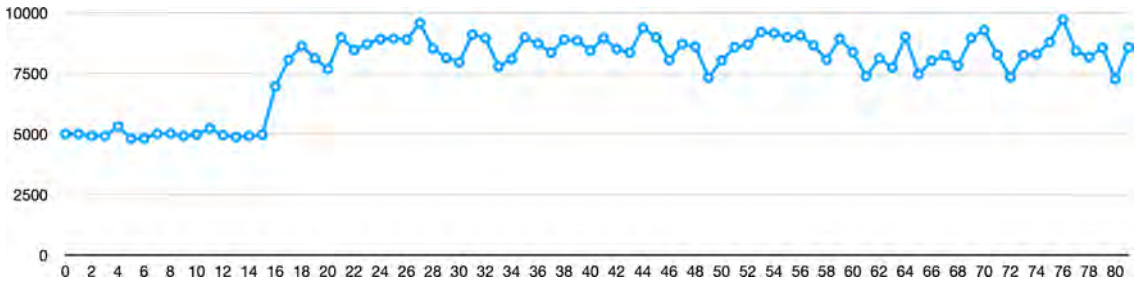


Abbildung 5.17: Blockzeiten im Test mit maximal 1.500 Transaktionen pro Block

6 Fazit und Ausblick

In der vorliegenden Arbeit wurde ein System zum Vergleich von Konsensverfahren geschaffen, das im Rahmen der Entwicklung des privaten Blockchain-Netzwerks im Projekt safe-UR-chain zum Einsatz kommt. Dafür wurde auf Grundlage der bereits entwickelten Blockchain-Software eine umfassende Testumgebung umgesetzt. Diese kann Blockchain-Netzwerke mit Nodes verschiedener Typen erstellen und überwachen. In dieser Umgebung werden nach dem Aufbau des Netzwerkes Datensätze mit einer gewissen Frequenz erzeugt, die vom Blockchain-Netzwerk als Transaktionen aufgenommen werden. Nachdem eine bestimmte Transaktionsanzahl in der Blockchain erreicht ist, weist die Testumgebung die Nodes zum Herunterfahren an und wertet die erzeugte Blockchain aus. Aus dieser Auswertung gehen Daten, wie die Transaktionsrate, Transaktionslatenz und Blockzeiten hervor. Die Testumgebung wurde so implementiert, dass sie automatisiert viele Testabläufe durchführen kann, was Tests mit verschiedenen Konfigurationen ermöglicht.

Innerhalb der Testumgebung wurden die drei Konsensverfahren Proof of Work, Proof of Elapsed Time und das selbst konzipierte Verfahren Proof of Network Signatures miteinander verglichen. Diese wurden zuvor für die Node-Software implementiert. Anschließend wurden Leistungsmetriken für die Konsensverfahren in insgesamt 120 Testdurchläufen ermittelt. Diese Tests unterschieden sich jeweils in der Frequenz der erzeugten Datensätze. Die Analyse der Testdaten hat gezeigt, dass große Ähnlichkeiten zwischen Proof of Work und Proof of Elapsed Time bestehen und das Proof of Network Signatures weniger leistungsstark ist.

Die Auswertung der Testergebnisse zeigte weiterhin, dass bei allen Konsensverfahren die Blockzeit durch die maximale Blockgröße beeinflusst wurde. Dieser Effekt hatte bei hohen Datenraten besonders starke Auswirkungen. Basierend auf dieser Beobachtung wurde die Annahme aufgestellt, dass eine Verringerung der Blockgröße zur Verringerung dieses Effektes beitragen kann. Diese Annahme wurde experimentell bestätigt. Durch diese Erkenntnis können Blockgrößen zukünftig so optimiert werden, dass sich ihr Einfluss auf die Blockzeiten minimiert.

Mithilfe der entwickelten Testumgebung können Konsensverfahren nicht nur miteinander, sondern auch mit anders konfigurierten Versionen von sich selbst verglichen werden. So können Parameter angepasst werden, um eine möglichst hohe Leistung zu erzielen. Zudem kann die Umgebung auch für die weitere Entwicklung des Blockchain-Netzwerkes im Forschungsprojekt safe-UR-chain eingesetzt werden. Beispielsweise bei der Softwareoptimierung oder für Integrationstests. Bereits während des Schreibens dieser Arbeit hat das Testsystem so einen Beitrag zur Fortentwicklung von safe-UR-chain geleistet.

In der zukünftigen Entwicklung kann das System so angepasst werden, dass es nicht nur Tests auf einem, sondern auch auf mehreren Rechnern innerhalb eines Netzwerkes ermöglicht. Damit lassen sich Tests in einer realen Umgebung durchführen und Konsensverfahren speziell auf den Einsatz innerhalb eines Unternehmens anpassen. Außerdem sollen zukünftig Tests mit weiteren Arten von Konsensverfahren ermöglicht werden.

Literaturverzeichnis

[1] Inc. Forrester Research. Forrester opportunity snapshot: A custom study commissioned by ibm. pages 5,6,8, 2020.

[2] Reshma Kamath. Food traceability on blockchain: Walmart's pork and mango pilots with ibm. pages 3–6. The Journal of The British Blockchain Association, 2018.

[3] Maersk. Tradelens blockchain-enabled digital shipping platform continues expansion with addition of major ocean carriers hapag-lloyd and ocean network express. Maersk, 2019. https://www.maersk.com/news/articles/2019/07/02/hapag-lloyd-and-ocean-network-express-join-tradelens [Online; aufgerufen am 5. Juli 2021].

[4] TradeLens. Tradelens and blockchain technology supply chain demo. pages 00:00 – 01:48. TradeLens, 2019. https://www.youtube.com/watch?v=0O2E9bCpKDk [Online; aufgerufen am 5. Juli 2021].

[5] Forbes. Forbes blockchain 50. Forbes, 2020. https://www.forbes.com/sites/michaeldelcastillo/2020/02/19/blockchain-50/ [Online; aufgerufen am 7. Juli 2021].

[6] Bundesministerium für Bildung und Forschung. Sicherheit und nachverfolgbarkeit in zivilen produktions und wertschöpfungsnetzwerken durch blockchain (safe-ur-chain). 2019. https://www.sifo.de/sifo/de/projekte/schutz-kritischer-infrastrukturen/kritische-strukturen-und-prozesse-in-produktion-und-logistik/bewilligte-projekte-aus-der-bekanntmachung-zivile-sicherheit.html [Online; aufgerufen am 30. August 2021].

[7] Bitcoin Core Developers. bip-0032. 2012. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki [Online; aufgerufen am 27. August 2021].

[8] Ecies Developers. eciesrs. 2021. https://github.com/ecies/rs [Online; aufgerufen am 15. Juni 2021].

[9] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–130, Washington, D.C., August 2015. USENIX Association.

[10] A. S. Tanenbaum and D. J. Wetherall. Computer networks (5th ed.). pages 125–127. Pearson Education, 2010.

[11] Jake Frankenfield. Consensus mechanism (cryptocurrency). 2021. https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp [Online; aufgerufen am 3. Juni 2021].

[12] Andreas Antonopoulos Et al. Mastering bitcoin. GitHub.com, 2021. https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch10.asciidoc#proof-of-work-algorithm [Online; aufgerufen am 11. Juli 2021].

[13] Dan Boneh Et al. Verifiable delay functions. page 4, 2019.

[14] Tien Tuan Anh Dinh et al. Blockbench: A framework for analyzing private blockchains. pages 6–8, 16, 2017.

[15] Evangelos Georgiadis. How many transactions per second can bitcoin really handle? theoretically. 2019.

[16] Bitcoin Wiki. Scalability. 2020. https://en.bitcoin.it/wiki/Scalability [Online; aufgerufen am 14. Juli 2021].

[17] Bitinfocharts.com. Bitcoin transactions historical chart. 2021. https://bitinfocharts.com/comparison/bitcoin-transactions.html [Online; aufgerufen am 14. Juli 2021].

[18] Ethereum Wiki. On sharding blockchains faqs. 2021. https://eth.wiki/sharding/Sharding-FAQs [Online; aufgerufen am 29. Juli 2021].

[19] Blockchair. Ethereum transactions per second. 2021. https://blockchair.com/ethereum/charts/transactions-per-second [Online; aufgerufen am 29. Juli 2021].

[20] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[21] Peter Schwarz et al. On elapsed time consensus protocols. 2017. https://github.com/hyperledger/sawtooth-core/blob/main/docs/source/architecture/poet.rst [Online; aufgerufen am 12. August 2021].

[22] Mic Bowman et al. On elapsed time consensus protocols. 2017.

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 25.10.2021