FH Aachen
Fachbereich Elektrotechnik und Informationstechnik
Prof. Dr. rer. nat. Dr.-Ing. Georg HOEVER

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Masterarbeit

Einsatz und Vergleich verschiedener Blockchain-Technologien am Beispiel einer Glücksspielanwendung

Eingereicht von:
Dany BROSSEL
Matrikelnummer: 3024062

Studienrichtung: Informatik

2018-04-14

Betreuer: Prof. Dr. rer. nat. Dr.-Ing. Georg HOEVER
Korreferent: Prof. Dr. Marco SCHUBA

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Datum:

Unterschrift:

FH AACHEN

Zusammenfassung

Fachbereich Name
Fachbereich 5

Information System Engineering

Einsatz und Vergleich verschiedener Blockchain-Technologien am Beispiel einer Glücksspielanwendung

von Dany BROSSEL

Diese Zusammenfassung werde ich erst am Ende schreiben. Also nicht die Idee beschreiben, sondern eher zusammengefasst, was gemacht wurde und welche Resultate aus der Arbeit hervorgehen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ECDSA	**E**lliptic **C**urve **D**igital **S**ignature **A**lgorithm
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
BIP	**B**itcoin **I**mprovement **P**roposal
GUI	**G**raphical **U**ser **I**nterface
RPC	**R**emote **P**rocedure **C**all
EIP	**E**thereum **I**mprovement **P**roposal
ABI	**A**pplication **B**inary **I**nterface
JSON	**J**avaScript **O**bject **No**tation

Kapitel 1

Einleitung

Die Erfindung der Kryptowährung Bitcoin und deren inhärente Blockchain-Technologie hat in den letzten Jahren einen regelrechten Hype ausgelöst. Begriffe wie Blockchain, Smart Contracts und Dezentralität sind in aller Munde.

Ziel dieser Masterarbeit ist es den Einsatz dieser neuartigen Technologie an der beispielhaften Realisierung einer Glücksspielanwendung zu demonstrieren. Der Einsatz einer Blockchain soll dabei das Vertrauen, dass der Endnutzer der Anwendung entgegenbringen muss, auf ein Minimum reduzieren.

1.1 Motivation

Herkömmliche Glücksspielanwendungen im Internet werden in der Regel von einer zentralen Organisation angeboten. Die Organisation muss sich den länderspezifischen Regeln und Gesetzen unterwerfen und kann daher nur in einem gewissen, vorgegebenen Rahmen operieren. Diese Regeln und Gesetze dienen einerseits dazu die Interessen des Staates zu wahren und andererseits den Endnutzer zu schützen.

Das Internet bietet zahlreiche Möglichkeiten des Glücksspiel an. Eine davon ist Online-Poker. Die Internetseite Pokerstars [14] bietet beispielsweise eine Plattform auf der man im Internet gegen andere Teilnehmer Poker spielen kann. Dabei muss der Spieler dem Service von Pokerstars vertrauen, dass dieser die Karten fair verteilt und keinen der Teilnehmer bevorzugt. Der Spieler hat keine Möglichkeit nachzuprüfen ob der Algorithmus, der den Spielern die Karten zuteilt, auch wirklich fair ist. Der Spieler muss der zentrale Organisation somit ein gewisses Maß an Vertrauen entgegenbringen.

1.2 Projektidee

Die in dieser Masterarbeit betrachtete Glücksspielanwendung soll ein Spiel anbieten, bei dem N Teilnehmer in einen Geldtopf einzahlen und auf ein zufälliges Event wetten. Jeder der Teilnehmer soll dabei die gleichen Gewinnchancen haben. Sobald alle Teilnehmer eingezahlt haben, wird einer der N Teilnehmer zufällig ausgewählt und gewinnt den gesamten Geldtopf. Der Gewinner bekommt somit seinen eigenen Einsatz als auch den Einsatz aller Mitspieler ausgezahlt. Die restlichen Teilnehmer verlieren und gehen leer aus.

Die erstmalig in Bitcoin verwendete Blockchain Technologie ist für die Entwicklung einer solchen Anwendung bestens geeignet, da sie transparente, pseudonyme Zahlungen ermöglicht. Außerdem lässt sich der für die Gewinnerauswahl benötigte Zufall durch ein in der Zukunft liegenden Zustand der Blockchain abbilden. Der

Zufallsfaktor kommt somit direkt von der Blockchain und daher von außerhalb der Glücksspielanwendung.

Die genauere Erklärung der Projektidee erfordert einiges Grundwissen im Bereich der Blockchain-Technologie. Das folgende Kapitel klärt daher einige grundlegende Begriffe.

1.3 Anforderungen

Die Glücksspielanwendung muss den folgenden Anforderungen gerecht werden.

1) Transparente Einzahlungen

Die Einzahlung jedes Endnutzers ist für jeden anderen Endnutzer nachprüfbar.

2) Gewinnerauswahl durch Zufallsfaktor

Die Auswahl des Gewinners ist von einem zufälligen Faktor abhängig, auf den weder die Anwendung noch die Endnutzer einen Einfluss haben.

3) Nachprüfbarkeit des Zufallsfaktor

Jeder Endnutzer kann die Echtheit des zufälligen Faktors eigenständig nachprüfen.

4) Transparente Auszahlungen

Die Auszahlung an den Gewinner muss transparent und somit für jeden Endnutzer nachprüfbar sein.

5) Fairheit des Spiels

Jeder Endnutzer besitzt die gleiche Gewinnwahrscheinlichkeit und niemand wird benachteiligt.

1.4 Vorhandenes

1.4.1 Cyberdice Protokoll

Einen ersten Ansatz wie man im Internet Glücksspiel ohne eine vertrauenswürdige Drittpartei betreiben kann, liefert [15]. Es stellt ein Kommunikationsprotokoll vor, das mit Hilfe kryptographischer Methoden sicherstellt, dass weder die Teilnehmer noch Außenstehende betrügen können. Das zum Glücksspiel verwendete Protokoll funktioniert aber nur unter der Annahme, dass es eine zentrale Institution (Bank) gibt, bei der die Teilnehmer Geld einzahlen und im Falle eines Gewinns gegen die Vorlage eines Beweises Geld ausgezahlt bekommen. Durch die Erfindung dezentraler Kryptowährungen, die auf einer für jeden einsehbaren Blockchain basieren, fällt diese vorher noch benötigte zentrale Institution weg.

1.4.2 Glücksspielseiten

Es gibt bereits Services die dezentrales, transparentes Glücksspiel mit Hilfe von Kryptowährungen umsetzen.

Die Internetseite Crypto Games [8] bietet Würfelspiele, Blackjack, Roulette, Online Poker und Lotto an. Der Nutzer hat dabei die Möglichkeit mit der Kryptowährung seiner Wahl zu bezahlen. Für die Gewinnerauswahl bezieht diese Seite einen Zufallsfaktor von der Bitcoin Blockchain ein, sodass der Benutzer dem Online Casino nicht vertrauen muss. Eine genaue Beschreibung des verwendeten Verfahren befindet sich am Ende dieser Ausarbeitung.

Die Internetseite [16] bietet Spiele, die durch Smart Contracts auf der Ethereum Plattform umgesetzt sind, an.

Kapitel 2

Erster Ansatz: Bitcoin

2.1 Grundlagen

Bei Bitcoin handelt es sich um die erste digitale, dezentral organisierte Währung. Die Idee digitaler Währungen existiert bereits seit der Erfindung des Internets. Allerdings scheiterten diese in der Vergangenheit daran, dass sie auf einen zentralen Punkt der Kontrolle angewiesen waren und somit einen Single Point of Failure beinhalteten. Die am 03. Januar 2009 gestartete digitale Währung "Bitcoin" schaffte es erstmalig gänzlich auf die Verwendung einer zentralen Instanz zu verzichten und somit ein verteiltes, dezentrales und sicheres digitales Zahlungssystem zu realisieren. Bitcoin wurde in dem von dem Pseudonym "Satoshi Nakamoto" veröffentlichen Paper "Bitcoin: A Peer-to-Peer Electronic Cash System" das erste Mal beschrieben. Bitcoin funktioniert durch das Zusammenspiel mehrerer Komponenten und besteht aus:

- Einem dezentralen Peer-to-Peer Netzwerk, dass mit Hilfe des Bitcoin-Protokolls kommuniziert.
- Der Blockchain, die eine öffentlichen Transaktionsdatenbank darstellt, die alle validen Transaktionen seit dem Start des Netzwerkes aufzeichnet.
- Eine Menge an Konsensregeln mit Hilfe derer Netzwerkteilnehmer eigenständig Transaktionen auf ihre Richtigkeit prüfen können.
- Ein Proof-of-Work Algorithmus der es erlaubt, sich in dem globalen dezentralen Netzwerk auf den Zustand der Transaktionsdatenbank zu einigen. Das kontinuierliche Ausführen des Proof-of-Work Algorithmus wird "Mining" genannt.

2.1.1 Peer-to-Peer Netzwerk

Peer-to-Peer-Netzwerke sind Netzwerke, die auf direkten Verbindungen zwischen Rechnern beruhen, ohne dass dabei einer der Rechner eine Sonderstellung einnimmt oder ein Server die Kommunikation vermittelt. In einem reinen Peer-to-Peer-Netz sind alle Computer gleichberechtigt und können sowohl Dienste in Anspruch nehmen, als auch zur Verfügung stellen. Das Peer-to-Peer-Modell ist somit grundlegend verschieden von dem im Internet am häufigsten verwendeten Client-Server-Modell. Da jeder Knoten des Netzwerks gleichzeitig Client und Server ist, gibt es keine zentrale Instanz die einen sogenannten "Single Point of Failure" darstellt.

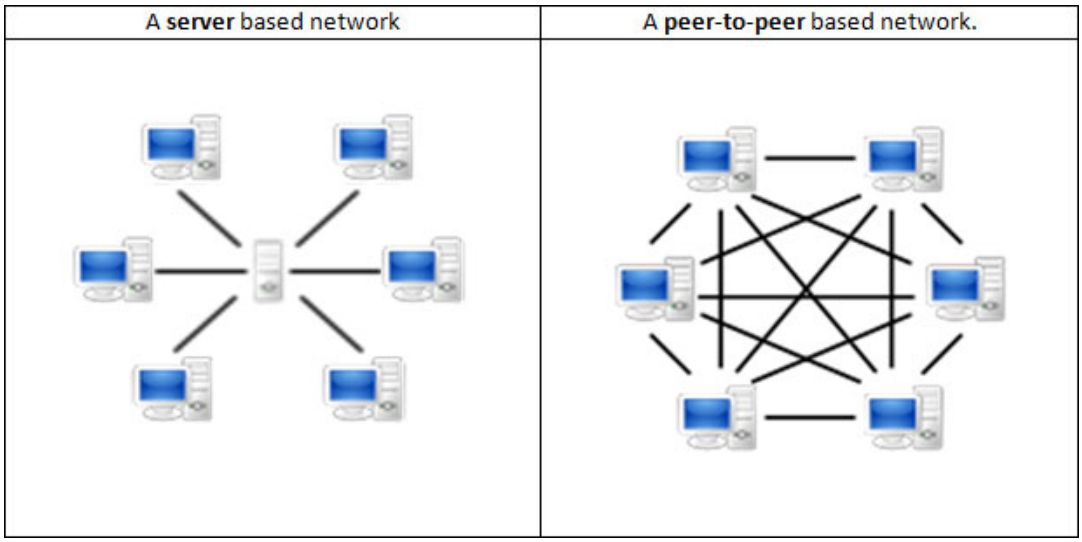


ABBILDUNG 2.1: Client-Server | Peer-to-Peer [13]

Peer-to-Peer Netzwerke sind selbstorganisierend. Das Hinzufügen neuer Teilnehmer und das Entfernen bestehender Netzwerkknoten findet ohne eine zentrale Verwaltung statt und behindert die Funktionsweise des Netzwerks nicht. Jeder Knoten des Netzwerks verwaltet eigenständig seine direkten Nachbarknoten. Die Art und Weise wie die Teilnehmer des Netzwerkes miteinander Kommunizieren ist durch das Netzwerkprotokoll vorgegeben. Um am Netzwerk teilzunehmen braucht man nur eine Software, die das Netzwerkprotokoll implementiert und einen Internetanschluss. Im Falle der Kryptowährungen nennt man die Software "Wallet" (englisch für Brieftasche), da man mit ihr Zahlungen initiieren und empfangen kann. Möchte ein Teilnehmer Bitcoins an einen anderen Teilnehmer senden, erstellt er dazu eine Nachricht die solch eine Transaktion beinhaltet und schickt sie an seine direkten Nachbarn des Peer-to-Peer Netzwerks. Die Nachbarn prüfen die Gültigkeit der Transaktion leiten diese gegebenenfalls an ihre Nachbarn weiter. Auf diese Art und Weise verteilt sich eine Transaktion im gesamten Netzwerk.

2.1.2 Blockchain

Eine Blockchain ist eine global verteilte Transaktionsdatenbank. Jeder Teilnehmer des Peer-to-Peer Netzwerkes speichert lokal eine Kopie dieser Datenbank. Dies erlaubt es ihm jegliche Datenbankeinträge zu lesen. Im Gegensatz zum lesenden Zugriff ist der schreibende Zugriff auf die Datenbank nur unter sehr strikten Regeln möglich. Über diese Regeln sind sich alle Teilnehmer des Netzwerkes einig. Daher werden diese Regeln Konsensregeln genannt. Möchte ein Teilnehmer eine Transaktion in die Datenbank schreiben, muss er sicherstellen, dass sie den Konsensregeln entspricht. Falls die Transaktion eine Konsensregel bricht wird sie vom Netzwerk verworfen und es ist ausgeschlossen, dass Sie in die Blockchain aufgenommen wird. Eine Transaktion beschreibt den Übergang von einem alten Systemzustand in einen neuen Systemzustand. Im Fall von Bitcoin handelt es sich bei dem Systemzustand um ein digitales Kontenbuch. Die Konten sind bei Bitcoin sogenannte Adressen und repräsentieren den öffentlichen Schlüssel eines ECDSA Schlüsselpaars. Um die einer Adresse zugeschriebenen Bitcoins zu überweisen, muss der Besitzer, mit Hilfe des privaten Schlüssels, eine digitale Signatur erstellen. Diese Signatur garantiert, dass die Überweisung vom Besitzer der Bitcoins autorisiert ist.

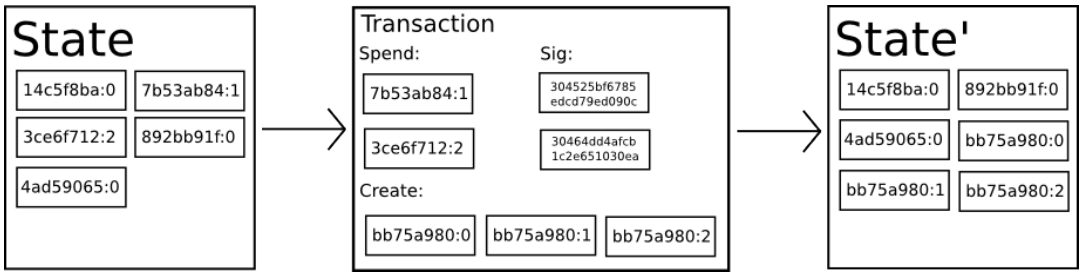


ABBILDUNG 2.2: Bitcoin Zustandsveränderung durch Transaktion

Die Transaktion aus Abbildung 2.2 überweist 1 Bitcoin von der Adresse `7b53ab84` und 2 Bitcoin von Adresse `3ce6f712` auf die Adresse `bb75a980` und überführt somit das Kontobuch in einen neuen Zustand. Möchten mehrere Teilnehmer den Systemzustand durch Transaktionen gleichzeitig anpassen, spielt die Reihenfolge in der die Transaktionen ausgeführt werden eine wichtige Rolle. Transaktionen werden in sogenannten Blöcken, in einer festen Reihenfolge, aggregiert. Somit werden nicht einzelne Transaktionen, sondern ganze Blöcke von Transaktionen in die Datenbank geschrieben. Genau wie bei den Transaktionen gibt es auch für Blöcke gewisse Konsensregeln. Sobald ein Block allen Konsensregeln entspricht, ist er bereit in die Datenbank aufgenommen zu werden. Da sich alle Netzwerkteilnehmer über die Gültigkeit des Blockes einig sind, wird somit die globale Blockchain Datenbank angepasst. Genau wie bei den Transaktionen ist auch die Reihenfolge der Blöcke wichtig. Daher beinhaltet jeder Block den Hash-Wert seines Vorgängers(siehe 2.3).

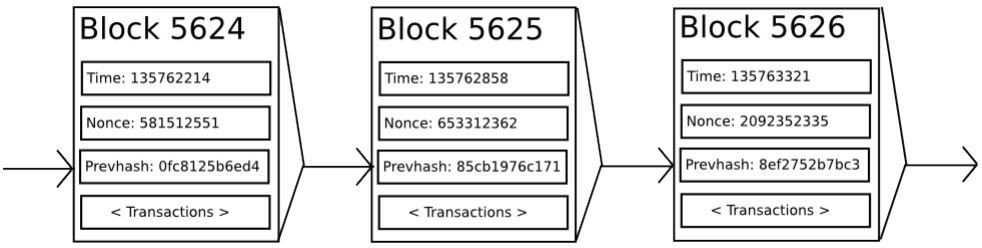


ABBILDUNG 2.3: Verkettung von Blöcken.

Durch die so erzielte Verkettung der Blöcke wird die Reihenfolge eindeutig festgelegt und es entsteht die sogenannte Blockchain. Am Anfang der Blockchain befindet sich der sogenannte Genesis Block. Auf diesen bauen alle weiteren Blöcke auf. Nachträgliche Änderungen an einem bereits eingefügtem Block sind nicht möglich, da sich dadurch der Blockhash des Blocks verändert und somit an dieser Stelle die Kette "zerbricht".

2.1.3 Konsensregeln

Konsenzregeln sind Regeln über die sich alle Teilnehmer des Peer-to-Peer-Netzwerks einig sind. Sie stellen sicher, dass die grundlegenden Eigenschaften der Kryptowährung eingehalten werden. Bei Bitcoin gibt es eine ganze Reihe an Konsensregeln. Die wichtigsten sind:

- Transaktionen dürfen kein Geld aus dem nichts schöpfen, sondern nur bereits existierende Beträge von einer Adresse auf eine andere Adresse überweisen. Die Blockreward-Konsensregel bildet hierzu die einzige Ausnahme. Sie legt fest wie neue Währungseinheiten erschaffen werden.
- Blockreward: Ein neuer Block muss genau eine Transaktion enthalten, die neue Kryptowährungseinheiten aus dem nichts erschafft. Sowohl die Höhe des Betrags als auch die Anpassung des Betrags über die Zeit ist in den Konsensregeln des Protokolls hart verankert. Bei Bitcoin startete der Blockreward bei 50 Bitcoin und halbiert sich seitdem alle 4 Jahre. Dies stellt sicher, dass es eine feste Obergrenze an Währungseinheiten gibt.
- Transaktionen, die Geld von Adresse A nach Adresse B überweisen, müssen durch eine Signatur beweisen, dass sie von dem rechtmäßigen Besitzer getätigt werden.
- Blockgröße: Diese Konsensregel legt die maximale Größe eines Blocks in Megabyte fest. Sie beeinflusst wie viele Transaktionen in einem Block gebündelt werden können. Dies ist wichtig, da sie zusammen mit der Blockzeit-Konsensregel das Wachstum der Blockchain-Datenbank steuert.
- Blockzeit: Diese legt fest in welchem durchschnittlich Zeitabstand es erlaubt ist einen neuen Block in die Blockhain einzufügen. Bei Bitcoin ist diese Zeit auf 10 Minuten festgelegt.
- Längste Blockchain-Kette: Teilnehmer des Peer-to-Peer Netzwerks folgen immer der Kette, die am meisten Proof-of-Work beinhaltet und betrachten "kürzere" Ketten als ungültig.

Die Konsensregeln ermöglichen es, dass jeder Teilnehmer eigenständig lokal seine Version der Blockchain Datenbank verwalten kann, ohne dabei einem anderen Teilnehmer vertrauen zu müssen. Die Konsensregeln stellen somit sicher, dass alle Teilnehmer die gleiche Blockchain Datenbank lokal aufbauen und sich dadurch auf den Systemzustand einigen.

2.1.4 Proof-of-Work und Mining

Das Einfügen eines neuen Blocks in die Blockchain Datenbank ist nur unter sehr strikten Regeln möglich. Bei einer dezentralen Währung wie Bitcoin gibt es keine zentrale Instanz, die solch eine Anpassung des Systemzustands koordiniert beziehungsweise autorisiert. Um dieses Problem zu lösen verwendet Bitcoin einen Proof-of-Work Algorithmus. Ziel des Algorithmus ist es durch das kontinuierliche[1] Hashen des neusten Blocks einen Hashwert zu finden der unterhalb eines dynamisch angepassten Zielwerts liegt. Der Zielwert wird durch die Blockzeit Konsensregel so angepasst, dass im gesamten Netzwerk im Durchschnitt alle 10 Minuten ein neuer Bitcoin Block gefunden wird. Findet ein Teilnehmer den Blockhash eines den Konsensregeln entsprechenden Blocks, leitet er diesen Block an das Peer-to-Peer Netzwerk weiter und erhält im Gegenzug den Blockreward. Teilnehmer, die unbestätigte Transaktionen empfangen, diese in Blöcke zusammenfassen und unter Zuhilfenahme des Proof-of-Work Algorithmus in die Blockchain einfügen, werden "Miner" genannt. Dieser Name stammt daher, dass bei diesem rechenintensiven Prozess gleichzeitig neue Bitcoins erschaffen werden.

[1] Das `nonce`-Feld des neusten Blocks wird nach jeder Berechnung des Hashwerts erhöht, damit ein neuer Hashwert entsteht.

Beim Mining werden die in Tabelle 2.1 aufgeführten Felder des Blockheaders als Eingabe für die kryptographische Hashfunktion SHA256 verwendet[2].

TABELLE 2.1: Bitcoin Blockheader

Feld	**Verwendungszweck**	**Aktualisiert falls...**	**Bytes**
version	Block Versionsnumber	man die Software aktualisiert und diese eine neue Version spezifiziert.	4
hashPrevBlock	256-bit Hash des vorherigen Blockheaders	ein neuer gültiger Block empfangen wird.	32
hashMerkleRoot	256-bit Hash aller Transaktionen des Blocks	eine neue Transaktion akzeptiert wird.	32
time	Aktueller Zeitstemple in Sekunden seit 1970-01-01	Alle paar Sekunden...	4
bits	Aktuelles Schwierigkeitsziel	wenn das Schwierigkeitsziel angepasst wird.	4
nonce	32-bit Nummer (von 0 aus erhöht)	der Hash ausprobiert wurde. (nonce+1)	4

Die Transaktionen des Blocks gehen nicht direkt, nur durch den sogenannten Merkel Tree Hash[3] in den Blockhash mit ein. Das `nonce`-Feld des Blockheaders wird nach jedem Hashversuch um den Wert 1 erhöht. Dadurch wird die Ausgabe der Hashfunktion kontinuierlich verändert.

Die Sicherheit des Bitcoin Netzwerkes und die Unmanipulierbarkeit der Blockchain ergeben sich aus der im Protokoll verankerten Spieltheorie. Die Spieltheorie macht es für einen Miner profitabler sich an die Spielregeln des Protokolls zu halten als zu versuchen das Netzwerk zu betrügen. Versucht ein Miner einen Block zu produzieren, der den Konsensregeln widerspricht, wird dieser vom Netzwerk verworfen. Somit hält kein anderer Netzwerkteilnehmer den vom Miner an sich selbst ausgeschütteten Blockreward für gültig. Da der Miner aber Ausgaben in Form von Hardwareabnutzung und Stromkosten zu bezahlen hat, schafft dies einen Anreiz sich den Konsensregeln zu unterwerfen. Solange 51 Prozent der Miner sich den Konsensregeln unterwerfen, formen diese auf Dauer die längste Blockchain. Die beim Mining anfallenden Stromkosten machen es außerdem sehr Teuer die Geschichte der Blockchain neu zu schreiben. Das Mining ist ein sich selbst regulierendes System bei dem die Anzahl Miner[4] sich mit dem Wert der Kryptowährung anpasst. Steigt der Preis pro Bitcoin, kommen neue Miner zum Netzwerk hinzu und machen das Netzwerk sicherer. Ein sinkender Preis hat zur Folge, dass der Blockreward nicht mehr für die Bezahlung der Strom und Hardwarekosten ausreicht. Dies hat zur Folge, dass die Anzahl an Miner abnimmt. Somit sinkt auch die Sicherheit des Netzwerks.

[2]Um genau zu sein wird der Blockheader bei Bitcoin zweimal mit der SHA 256 Hashfunktion gehasht. Blockhash = SHA256(SHA256(Blockheader))

[3]Der genaue Aufbau des Merkel Trees und dessen Vorteile sind in [12] genauer beschrieben.

[4]Genauer gesagt nicht die Anzahl Miner, sondern die von den Minern verwendete Rechenleistung.

2.2 Konzept

Die folgenden Schritte beschreiben den Finanzfluss zwischen den Teilnehmern und der Anwendung sowie die Gewinnerauswahl durch den in der Zukunft liegenden Blockchain-Status. Der Ablauf ist allgemein gehalten und kann nicht nur mit Bitcoin, sondern auch mit anderen Kryptowährungen, die auf einer Proof-of-Work Blockchain basieren, umgesetzt werden. Betrachtet wird ein Spiel mit N Teilnehmern, bei dem jeder Teilnehmer einen Einsatz von X Währungseinheiten zur Teilname zahlen muss.

1. Im ersten Schritt eröffnet die Anwendung ein neues Spiel in dem es N freie Plätze und einen leeren Geldtopf gibt.
2. Sobald ein Spieler am Spiel teilnehmen möchte, generiert die Anwendung eine neue Empfangsadresse und zeigt diese dem Spieler an.
3. Der Spieler verwendet die Wallet Software seiner Wahl um eine Transaktion zu erstellen, die den Einsatz an die angezeigte Empfangsadresse überweist. Die Wallet Software signiert die Transaktion und leitet sie über die mit ihr verbundenen Nachbarn an das Peer-to-Peer Netzwerk weiter.
4. Ein Miner empfängt die Transaktion und nimmt sie in den nächsten Block auf.
5. Der Miner findet den zu seinem Block passenden Proof-of-Work-Hash und schickt den Block an das Netzwerk.
6. Die Applikation empfängt den Block und merkt, dass im Block eine Transaktion auf die in Schritt 2 generierte Empfangsadresse enthalten ist. Die Applikation prüft die Höhe des Transaktionsbetrag und leitet anschließend die vom Spieler kontrollierte Auszahlungsadresse aus der Transaktion ab.
7. Die restlichen N-1 Teilnehmer überweisen ebenfalls den geforderten Betrag auf die ihnen angezeigte Empfangsadresse.
8. Sobald die letzte Transaktion in einen validen Block aufgenommen wurde, zählt die Reihenfolge in der die Transaktionen in der Blockchain stehen. Die Reihenfolge steht somit fest und kann nicht mehr nachträglich verändert werden. Der Geldtopf ist nun mit einem Betrag von N*X Krytowährungseinheiten gefüllt und wird geschlossen. Der Block nach dem Block, in dem die letzte Einzahlungstransaktion eingegangen ist, wird zur Gewinnerauswahl genutzt. Die Anwendung merkt sich die Blocknummer dieses Blocks.
9. Die Anwendung und die Teilnehmer warten darauf, dass der nächste Block von einem Miner gefunden wird. Alle Miner des Peer-to-Peer Netzwerks versuchen schnellstmöglich einen passenden Blockhash zu finden um den Blockreward zu erhalten. Ein Miner gewinnt dieses Rennen und teilt dem Netzwerk den neu gefundenen Block mit.
10. Die Anwendung empfängt den nächsten Block und ermittelt durch diesen den Gewinner. Die Berechnung erfolgt indem die Anwendung den Blockhash des Blocks vom Hexadezimalsystem ins Dezimalsystem konvertiert. Der dabei resultierende sehr hoher Wert B bildet die Grundlage für die Gewinnerauswahl. Durch die Berechnung von B modulo N resultiert eine Zahl G zwischen 0 und N-1 die den Gewinner festlegt. Der Spieler der die G+1te Einzahltransaktion gesendet hat, gewinnt den Geldtopf.

11. Die Anwendung erstellt eine Transaktion, die alle N*X Krytowährungseinheiten des Geldtopfs an die Auszahlungsadresse des Gewinners überweist, und sendet diese an das Netzwerk.

12. Die Wallet Software des Gewinners, empfängt die Transaktion und informiert den Teilnehmer, dass er den gesamten Betrag des Topfes erhalten hat.

Im folgenden Beispiel wird einen Topf mit 5 Teilnehmern, die Kryptowährung Bitcoin und einen Einzahlungsbetrag von 0,1 Bitcoin betrachtet. Dieses Beispiel verdeutlicht sowohl die Interaktion der verschiedenen Teilnehmer des Peer-to-Peer Netzwerks, als auch die Veränderung des Status der Blockchain.

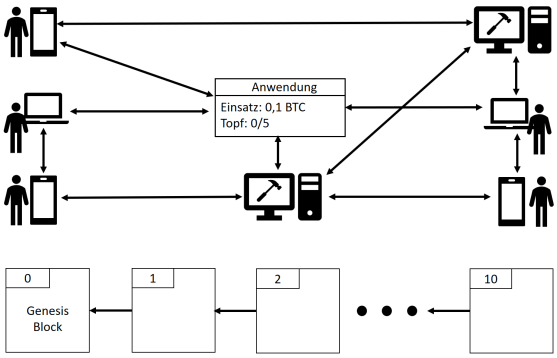


ABBILDUNG 2.4: Schritt 1

Diese Abbildung zeigt das Peer-To-Peer Netzwerk. Die 5 potentiellen Teilnehmer sind durch Notebooks und Smartphones dargestellt. Außerdem sind 2 Miner und die Glücksspielanwendung teil des Peer-To-Peer Netzwerks. Der aktuelle Status der Blockchain, die jeder Teilnehmer des Netzwerks lokal speichert ist unterhalb des Netzwerkes dargestellt.

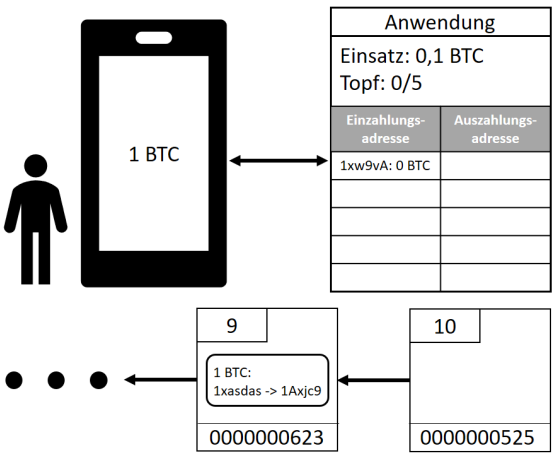


ABBILDUNG 2.5: Schritt 2

Die Bitcoin Client Software der Glücksspielanwendung generiert eine neue Bitcoinadresse und speichert den dazugehörigen privaten Schlüssel in der Wallet. Sobald Bitcoins auf dieser Adresse empfangen werden, können sie nur durch den Besitz des privaten Schlüssels weiter transferiert werden. Die Anwendung zeigt dem Benutzer eine frisch generierte Empfangsadresse über die Benutzeroberfläche an. Der Zustand der Blockchain verändert sich nicht.

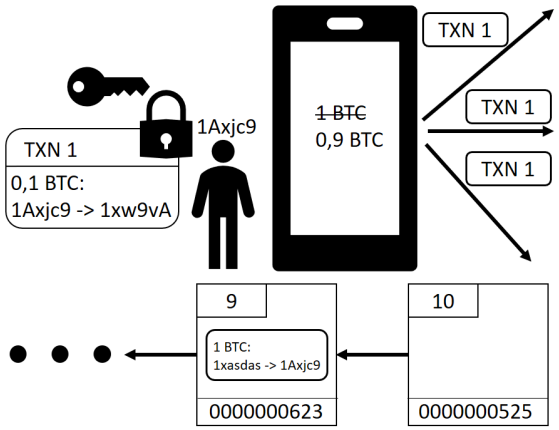


ABBILDUNG 2.6: Schritt 3

Nun zahlt der Spieler mit Hilfe seiner Bitcoin Wallet Software in den Geldtopf ein. Dazu erstellt er eine Transaktion, die Bitcoin von seiner Adresse auf die generierte Adresse der Glücksspielanwendung transferiert. Durch die Signierung mit seinem privaten Schlüssel autorisiert er die Überweisung. Anschließend schickt er die Transaktion seinen Nachbarn.

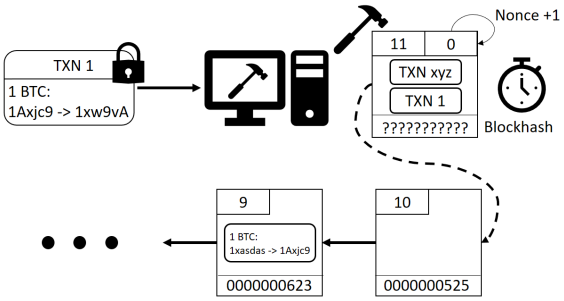


ABBILDUNG 2.7: Schritt 4

Sobald die Transaktion TXN 1 einen Miner erreicht, prüft dieser ob die Transaktion in Einklang mit den Konsensregeln ist. In diesem Beispiel existiert in Block 9 eine Transaktion von einem Bitcoin auf die Adresse des Teilnehmers. Unter der Annahme, dass dieser Bitcoin nicht in Block 10 weiter überwiesen wurde, befindet sich auf der Adresse des Teilnehmers somit ein Bitcoin. Außerdem prüft der Miner ob die Signatur der Transaktion gültig ist. Da die Transaktion valide ist, fügt er sie dem aktuell zu generierenden Block Nummer 11 hinzu.

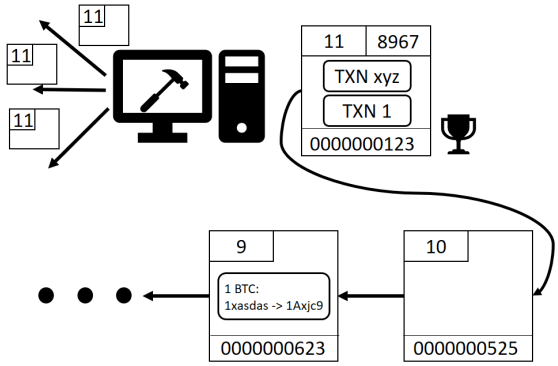


ABBILDUNG 2.8: Schritt 5

Der Miner berechnet nun mithilfe der SHA256 Hashfunktion den Hash des Blocks. Falls der Blockhash-Wert den durch die Konsensregeln dynamischen angepassten Schwierigkeits-Wert unterschreitet, gilt der Block als valide. Überschreitet der Blockhash den Wert, erhöht der Miner den Nonce-Wert des Blocks und berechnet den Blockhash erneut. Diesen Prozess wiederholt er solange bis er entweder einen gültigen Blockhash findet oder einen gültigen Block Nummer 11 von einem anderen Netzwerkteilnehmer empfängt. In diesem Beispiel findet der Miner einen gültigen Blockhash, leitet den Block ans Netzwerk weiter und wird dadurch mit neu erschaffenen Bitcoin für seinen Rechenaufwand belohnt.

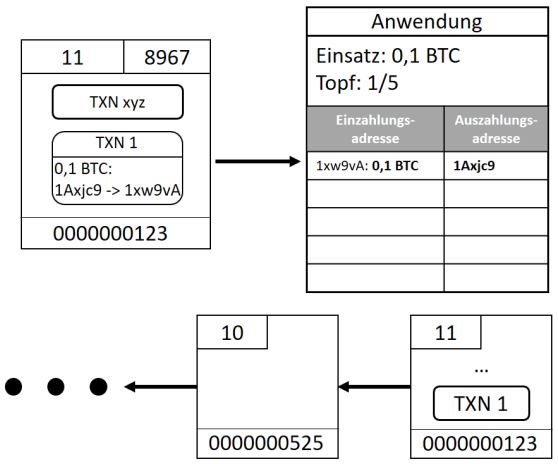


ABBILDUNG 2.9: Schritt 6

Die Glücksspielanwendung empfängt den Block Nummer 11 und überprüft ob er im Einklang mit den Konsensregeln ist. Dies ist der Fall. Somit wird die lokale Blockchain Datenbank um einen Block erweitert. Die Glücksspielanwendung hat somit den Einsatz des ersten Spielers erhalten. Aus der Einzahlungstransaktion des Spielers leitet die Anwendung die Auszahlungsadresse **1Axjc9** des Spielers ab.

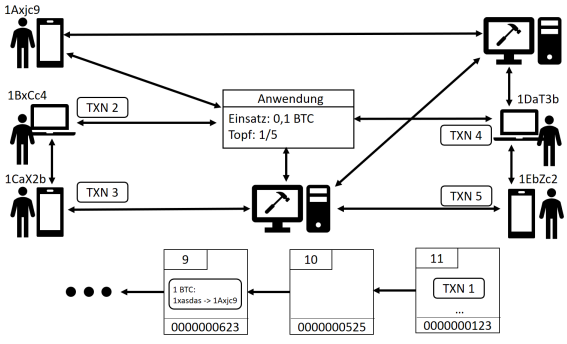


ABBILDUNG 2.10: Schritt 7

Die restlichen Spieler senden ihre signierten 0,1 Bitcoin Transaktionen ins Peer-to-Peer Netzwerk. Diese sind in Abbildung 7 durch die Transaktionen TXN 2 bis 5 dargestellt.

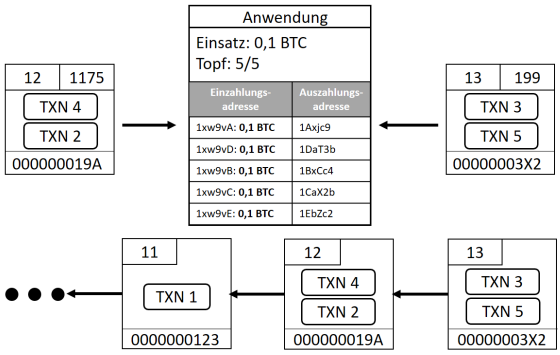


ABBILDUNG 2.11: Schritt 8

Beliebige Miner fügen die Transaktionen in ihre Blöcke ein. Sobald die Glücksspielanwendung die Blöcke empfängt, prüft sie diese gegen die Konsensregeln und fügt sie in die lokale Blockchain ein. Die Applikation merkt nun, dass alle Spieler bezahlt haben und schließt den Geldtopf. Dabei merkt sie sich die Nummer des Blocks in der die letzte Einzahlungstransaktion vorhanden ist. Der darauffolgende Block mit Nummer 14 wird für die Ziehung des Gewinners verwendet.

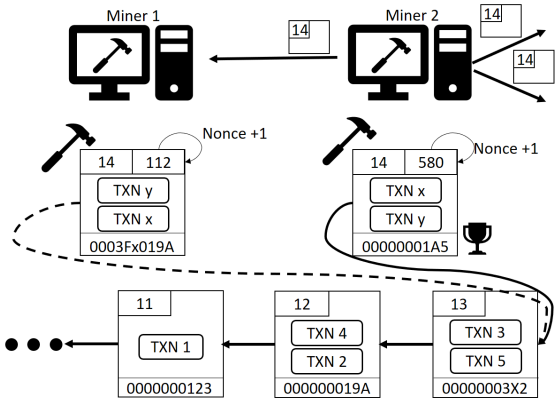


ABBILDUNG 2.12: Schritt 9

Alle Miner des Netzwerkes versuchen nun gleichzeitig so schnell wie möglich den nächsten Block zu finden. Da sie dazu eine kryptographische Hashfunktion benutzen bei der die Ausgabe ein unkontrollierbarer zufälliger Wert ist, hat keiner der Miner einen direkten Einfluss auf den resultierenden Blockhash. In diesem Beispiel findet Miner 2 einen gültigen Blockhash vor Miner 1. Miner 2 leitet seinen gültigen Block Nummer 14 so schnell wie möglich an das Netzwerk weiter und erhält den Blockreward als Belohnung. Miner 1 geht leer aus.

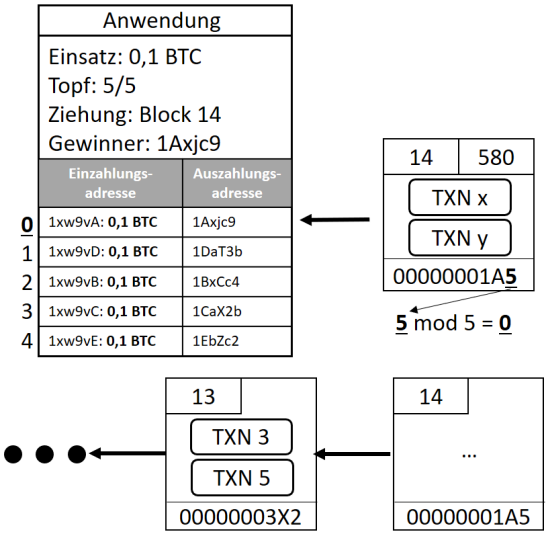


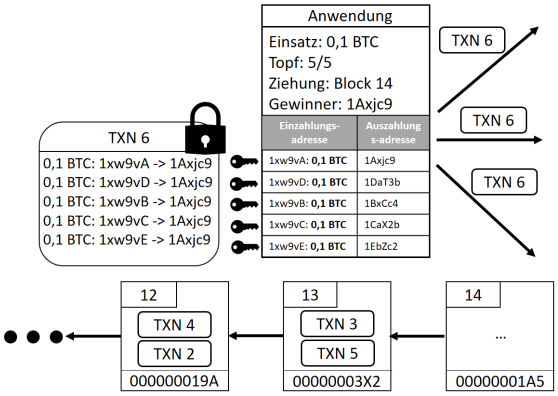
ABBILDUNG 2.13: Schritt 10

Die Anwendung empfängt Block 14 und prüft ihn gegen die Konsensregeln. Der Block ist valide. Daher verwendet die Anwendung den im Block enthaltenen Blockhash um den Gewinner des Geldtopfes zu ermitteln. Statt des gesamten Blockhashs verwendet die Anwendung nur die letzte Ziffer des Blockhashs zur Gewinnerauswahl. Dies hat den Vorteil, dass die Teilnehmer die Korrektheit der Gewinnerauswahl leichter eigenständig nachprüfen können. Da die letzte numerische Stelle des Blockhashs 10 verschiedene Werte annehmen kann, ordnet die Anwendung jedem der 5 Teilnehmer 2 Gewinnzahlen zu.

Dies erreicht die Anwendung indem sie die letzte Blockhash-Ziffer modulo 5 nimmt. Dadurch ergibt sich die Verteilung der Gewinnzahlen folgendermaßen:

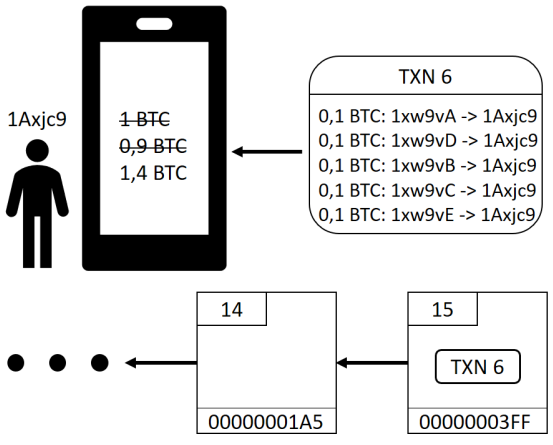
- Spieler 1 mit Adresse 1xw9vA gewinnt bei 0 und 5,
- Spieler 2 mit Adresse 1xw9vD gewinnt bei 1 und 6,
- Spieler 3 mit Adresse 1xw9vB gewinnt bei 2 und 7,
- Spieler 4 mit Adresse 1xw9vC gewinnt bei 3 und 8,
- Spieler 5 mit Adresse 1xw9vE gewinnt bei 4 und 9.

Jeder Teilnehmer besitzt nun eine Gewinnwahrscheinlichkeit von 1/5. Block Nummer 14 hat den Blockhash **00000001A5**. Die zur Gewinnerauswahl benutze Ziffer ist somit die 5. Da 5 modulo 5 den Wert 0 ergibt, gewinnt Spieler 1 mit der Adresse **1xw9vA**.



Die Anwendung erstellt nun eine Transaktion die alle Spieleinsätze an die Auszahlungsadresse **1Axjc9** von Spieler 1 überweist. Um die Transaktion zu signieren verwendet die Anwendung die, zu den 5 Einzahlungsadressen passenden, privaten Schüssel. Anschließend leitet die Anwendung die Transaktion an das Peer-to-Peer Netzwerk weiter.

ABBILDUNG 2.14: Schritt 11



Das Smartphone von Spieler 1 empfängt die Transaktion noch bevor sie von einem Miner in einen validen Block aufgenommen wurde. Die Wallet Software zeigt die Transaktion erst als unbestätigt an. Sobald sie durch die Aufnahme in Block 15 bestätigt wurde, gilt sie für die Wallet Software als bestätigt.

ABBILDUNG 2.15: Schritt 12

2.3 Umsetzung

2.3.1 Interaktion mit dem Bitcoin Netzwerk

Möchte man mit dem Bitcoin Netzwerk kommunizieren benötigt man einen Client der das Bitcoin Protokoll implementiert. Dieser kommuniziert dann mit dem Peer-to-Peer Netzwerk und wird dadurch zu einem Knoten (”Node”) des Peer-to-Peer Netzwerks. Man unterscheidet zwischen sogenannten ”Full Nodes” und ”Light Nodes”.

Full Node

Knoten die eigenständig alle Transaktionen und Blöcke auf Gültigkeit mit Hilfe der Konsensregeln prüfen nennt man ”Full Node”. Diese Knoten speichern die gesamte Blockchain und bilden das Rückgrat des Netzwerkes. Full Nodes stellen in der Regel eine RPC Schnittstelle zur Verfügung. Diese Schnittstelle bietet die Möglichkeit von einer beliebigen Programmiersprache aus mit dem Full node zu interagieren. Abbildung 2.16[1] zeigt, dass man über die RPC Schnittstelle auf die gespeicherten Daten des Nodes (Blöcke, Blockheader und Adressen) als auch auf die Wallet zugreifen kann.

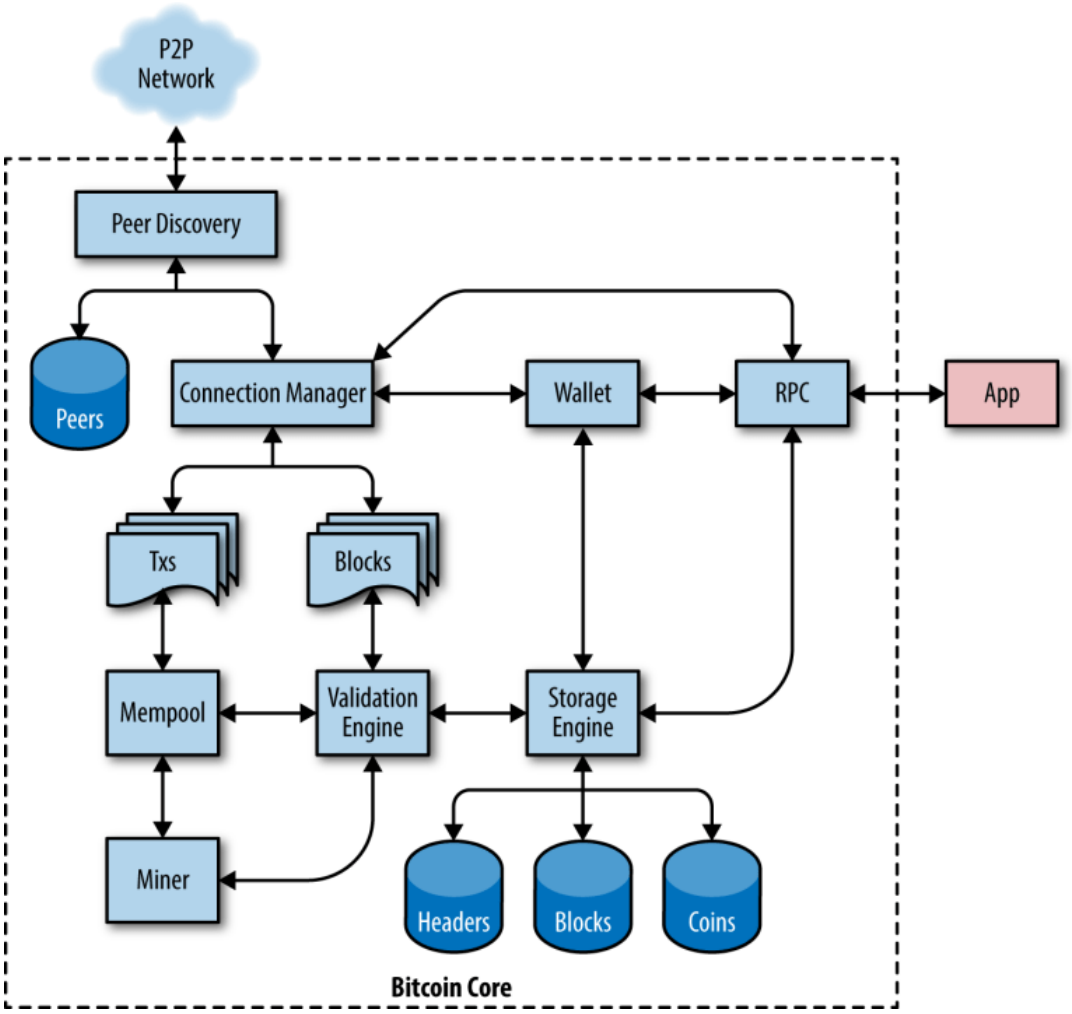


ABBILDUNG 2.16: Bitcoin Core: Full Node Aufbau

Abbildung 2.16 zeigt außerdem:

- Peer Discovery, Peer Datenbank und Connection Manager: Diese kümmern sich um die Kommunikation mit dem Peer-to-Peer Netzwerk.
- Mempool: Im sogenannten Mempool werden empfangene, unbestätigte Transaktionen im Speicher gehalten.
- Validation Engine: Diese validiert, ob die empfangenen Blöcke und deren Transaktionen die Konsensregeln einhalten. Falls ja werden die somit bestätigten Transaktionen aus dem Mempool gelöscht und der Block wird an die Storage-Engine zur Abspeicherung in der Blockchain Datenbank weitergereicht.
- Miner: Die Bitcoin Full Node Software enthält einen CPU Miner mit der man mithilfe des Proof-of-Work Algorithmus nach neuen Blöcken suchen kann. Bitcoin Mining ist heutzutage nur noch mit sogenannten ASICs profitabel. ASIC steht für **A**pplication-**S**pecific **I**ntegrated **C**ircuit. Es handelt sich um Hardware, die auf die Berechnung der SHA256 Hashfunktion spezialisiert ist.

Light Node

"Light Nodes" speichern nicht die gesamte Blockchain, sonder in der Regel nur die Blockheader der Blöcke der Blockchain. Beim Mining gehen nur die Daten des Blockheaders in den Blockhash ein. Der Node empfängt Blockheader, prüft ihre Gültigkeit und fügt sie gegebenenfalls in die Headerkette ein. Der Node kann somit eigenständig, d.h. ohne seinen Nachbarn vertrauen zu müssen, die längste Proof-of-Work Kette bilden. Da diese Kette nur aus Headern besteht und keine Transaktionen enthält, kann der Light Node empfangene Transaktionen nicht eigenständig auf ihre Gültigkeit prüfen. Light Nodes verwenden das in [12] beschriebene "Simplified Payment Verification" Verfahren zur Prüfung von Transaktionen. "Light Nodes" werden daher oft auch "SPV Client" genannt. Ein "SPV Client" prüft die Gültigkeit einer Transaktion indem er sie an der richtigen Stelle der Headerkette einordnet und dann den passenden Merkel-Branch von einem seiner Nachbarknoten anfragt. Durch diese Zusätzlichen Daten kann er nun wie in Abbildung 2.17[6] gezeigt nachprüfen ob der Hash der Transaktion wirklich in den Wurzelknoten des Merkel-Trees mit eingegangen ist.

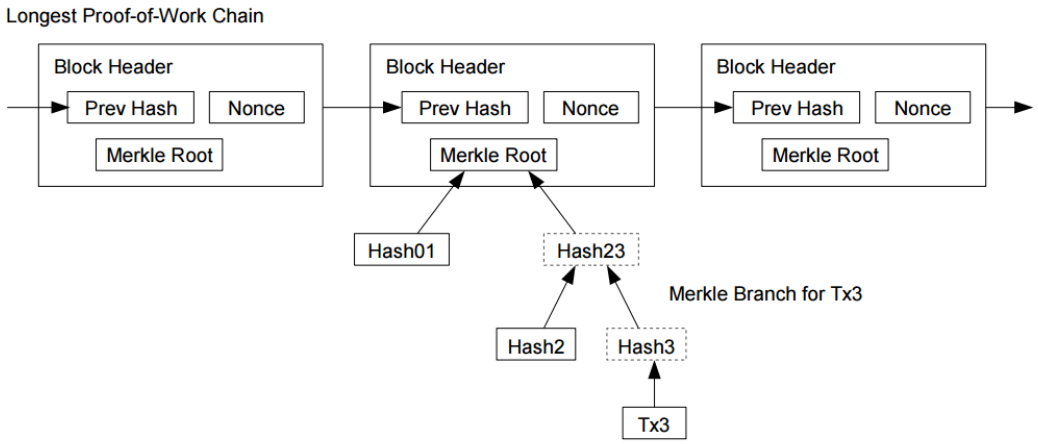


ABBILDUNG 2.17: Blockheader Kette

Für den Bitcoin Teil dieser Ausarbeitung ist die Integration mit dem Peer-to-Peer Netzwerk mit Hilfe der in Java geschriebenen BitcoinJ[4] Bibliothek umgesetzt.

2.3.2 Überblick

Abbildung 2.18 skizziert die Komponenten der Glücksspielanwendung und wie diese mit ihrer Umgebung kommunizieren. Es gibt zum einen den Server auf dem die Glücksspielanwendung läuft, die Spieler und das Bitcoin Netzwerk.

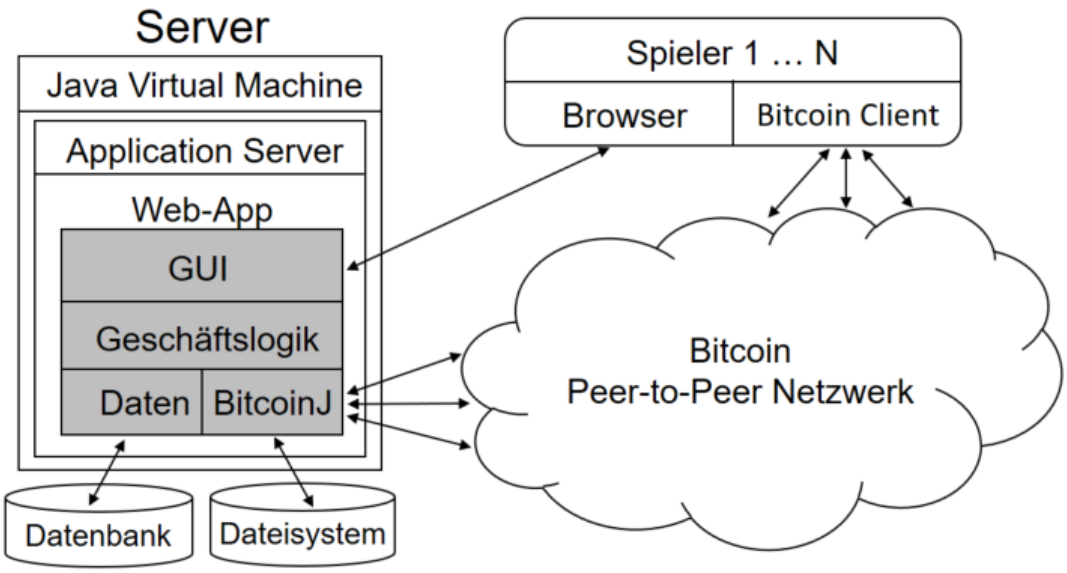


ABBILDUNG 2.18: Glücksspielanwendung Aufbau und Interaktion

Glücksspielanwedung

- Server: Die Glücksspielanwendung läuft auf einem Server, der über eine Java Virtual Machine (JVM) Laufzeitumgebung verfügt, eine MySQL Datenbank und ein gewöhnliches Dateisystem besitzt.
- Java Virtual Machine (JVM): Innerhalb der JVM läuft ein sogenannter Application Server, der eine Webanwendung nach außen bereitstellt. Auf diese Webanwendung können die Spieler über das HTTP Protokoll mittels ihres Browsers zugreifen. Die Webanwendung besteht aus mehreren Komponenten.
- Application Server: Dieser stellt die Applikation bereit. Bei der Umsetzung der Glücksspielanwendung wurde der Open Source Application Server Wildfly[5] von Red Hat verwendet.
- GUI: Die Weboberfläche stellt die zentrale Schnittstelle zwischen der Anwendung und dem Spieler da. Diese ist mithilfe des Tapestry[6] Webframeworks von Apache umgesetzt. Detaillierte Informationen findet man in [10].
- Geschäftslogik: Diese Behandelt sowohl die vom Benutzer über die GUI ausgelösten, als auch die vom Bitcoin Netzwerk ausgelösten Events.

[5] http://wildfly.org/
[6] http://tapestry.apache.org/

- BitcoinJ: Die Java Bibliothek die zur Kommunikation mit dem Bitcoin Netzwerk verwendet wird.

Spieler

Die Spieler verfügen über einen Browser und über einen Bitcoin Client. Mit dem Internetbrowser interagieren sie mit Glücksspielanwendung. Mit dem Bitcoin Client erstellen und empfangen Sie Zahlungen.

Bitcoin Peer-to-Peer Netzwerk

Das Peer-to-Peer Netzwerk besteht aus den anderen Teilnehmern des Netzwerks. Dies sind Full-, Light Nodes und Miner. Bei Kryptowährungsnetzwerken unterscheidet man in der Regel zwischen dem Test und Hauptnetzwerk. Den Bitcoins des Testnetzwerks wird kein monetärer Wert zugeschrieben. Das Testnetzwerk dient dazu Software die dem Bitcoin Netzwerk interagieren soll zu testen. Möchte ein Händler Bitcoin in seinen Onlineshop integrieren, kann er so seine Implementierung testen ohne ein finanzielles Risiko einzugehen.

2.3.3 Datenmodel

Die Grundklasse die die Anwendung verwendet ist die Klasse `Pot`. Diese repräsentiert ein Spiel und speichert alle relevanten Daten. Sie besteht aus einer Liste von Teilnehmern (`Participant`). Jeder Teilnehmer hat wie im Konzept beschrieben eine Ein- und Auszahlungsadresse.

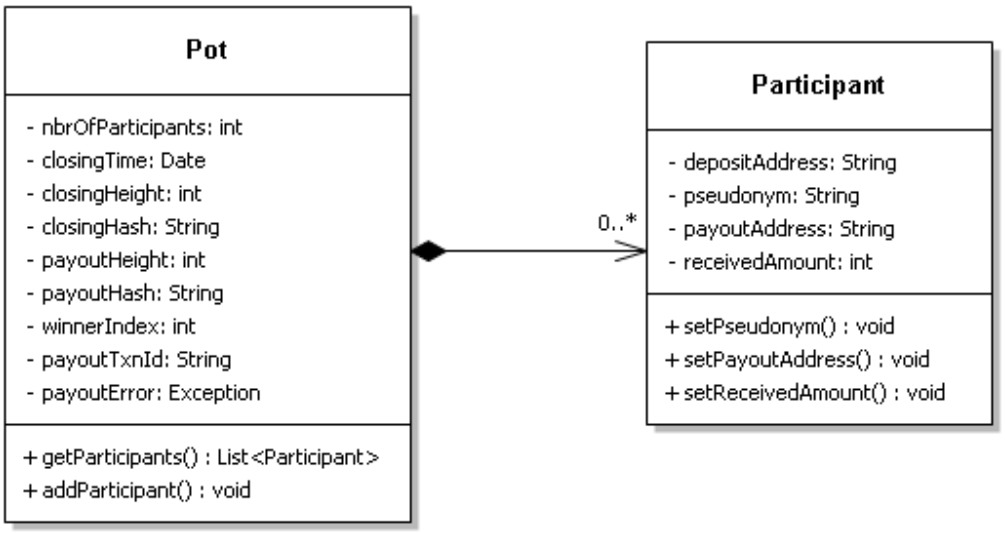


ABBILDUNG 2.19: Java Datenmodel Klassendiagramm

2.3.4 Geschäftslogik

Die Java Klassen der Glücksspielanwendung können, wie in Abbildung 2.20 gezeigt, in 3 verschiedene Gruppen unterteilt werden. Das **Core Module** enthält die Klassen des Datenmodells und ein Interface mit dem die GUI Anwendung interagiert. Das

Interface entkoppelt die Anzeigelogik der GUI von der Geschäftslogik der jeweiligen Kryptowährung. Die GUI Komponente bekommt von der Schnittstelle allgemeine Daten und kümmert sich nur um deren Anzeige. Das **Bitcoin Service Module** enthält die gesamte kryptowährungsspezifische Geschäftslogik. **BitcoinJ** enthält alle Klassen und Interfaces die benötigt werden um mit dem Bitcoin Netzwerk zu interagieren und Daten aus der Blockchain auszulesen.

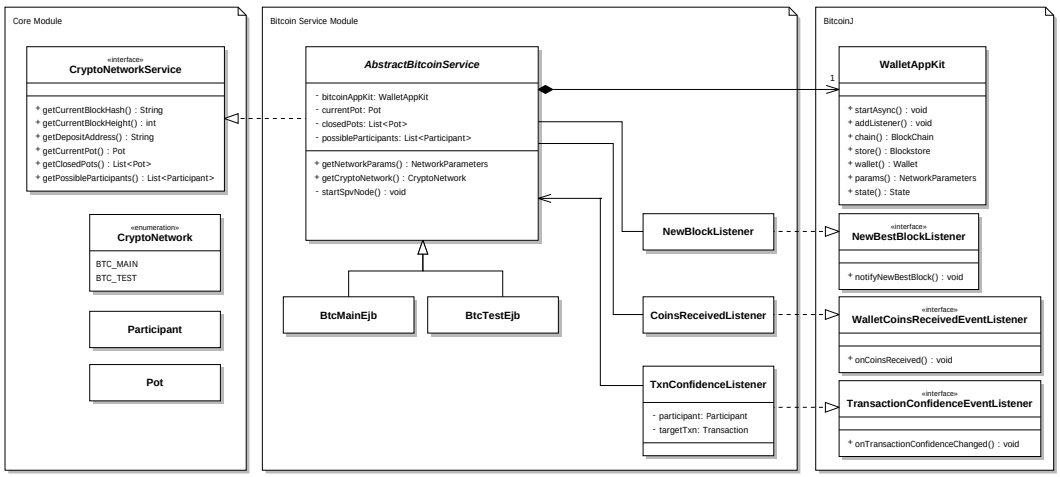


ABBILDUNG 2.20: Java Geschäftslogik Klassendiagramm

Start der Anwendung

Beim Kompilieren der Anwendung legt man über einen Konfigurationseintrag fest, ob die Anwendung mit dem Bitcoin Haupt- oder Testnetzwerk interagieren möchte. Für das Hauptnetzwerk wird die Java Klasse `BtcMainEjb` verwendet. Für das Testnetwerk wird die Klasse `BtcTestEjb` verwendet. Beide Klassen verwenden die gleiche Implementierung der abstrakten Oberklasse `AbstractBitcoinService`. Diese wiederum implementiert das von der GUI Komponente verwendete Interface `CryptoNetworkservice`.

```
1  import javax.ejb.Startup;
2  import org.bitcoinj.params.AbstractBitcoinNetParams;
3  import org.bitcoinj.params.MainNetParams;
4  import com.ossel.gamble.bitcoin.services.AbstractBitcoinService;
5  import com.ossel.gamble.core.data.enums.CryptoNetwork;
6  /**
7   * Class will be excluded for the testnet jar via the maven jar
        plugin.
8   */
9  @Startup
10 @Singleton
11 public class BtcMainEjb extends AbstractBitcoinService {
12     @Override
13     public CryptoNetwork getCryptoNetwork() {
14         return CryptoNetwork.BTC_MAIN;
15     }
16     @Override
17     public AbstractBitcoinNetParams getNetworkParams() {
18         return MainNetParams.get();
19     }
```

```
20 }
```

Die beiden Klassen `BtcMainEjb` und `BtcTestEjb` sind mit `@Startup` und `@Singleton` annotiert. Es handelt sich um sogenannte **E**nterprise **J**ava **B**eans. Dies bedeutet, dass der Applikationsserver diese eigenständig managt und genau eine Instanz der Klasse beim Starten der Applikation erzeugt. Beim Start wird dann die mit `@PostConstruct` annotierte `startSpvNode` Methode aufgerufen und abgearbeitet. Diese konfiguriert und startet das `WalletAppKit`, welches die zentrale Klasse zur Interaktion mit der BitcoinJ Bibliothek darstellt.

```
1  private void startSvpNode() {
2      log.info("#### Start Bitcoin SPV Node  ####");
3      currentPot = new Pot(2, 100000L);
4      File walletDir = CoreUtil.getWalletDirectory();
5      NetworkParameters params = getNetworkParams();
6      String fileName = "bitcoin-" + params.getPaymentProtocolId();
7      bitcoinAppKit = new WalletAppKit(params, walletDir, fileName) {
8          @Override
9          protected void onSetupCompleted() {
10             log.info("#### Bitcoin SPV Node started ####");
11         }
12     };
13     bitcoinAppKit.startAsync();
14     waitUntilStarted(bitcoinAppKit);
15     newBlockListener = new NewBlockListener(this);
16     bitcoinAppKit.chain().addNewBestBlockListener(newBlockListener);
17     coinReceivedListener = new CoinsReceivedListener(this);
18     bitcoinAppKit.wallet().addCoinsReceivedEventListener(coinReceivedListener);
19 }
```

In Zeile 4 wird zunächst ein neuer leerer Topf mit 2 Teilnehmern erzeugt. Anschließend wird das `WalletAppKit` erzeugt. Dazu bekommt dieses die gewünschten Netzwerkparameter und den Pfad zum Dateisystem in dem `BitcoinJ` die Blockchain- und Wallet-Daten speichern soll. Zeile 13 startet den durch das `WalletAppKit` repräsentierten SPV Node. Anschließend wird dem `WalletAppKit` noch ein `NewBlockListener` und ein `CoinsReceivedListener` hinzugefügt, um auf neue Blöcke und eingehende Zahlungen zu reagieren.

GUI Events

Der folgende Code zeigt die Implementierung der Methoden die von der GUI Komponente aufgerufen werden können.

```
1  public Pot getCurrentPot() {
2      return currentPot;
3  }
4
5  public String getDepositAddress() {
6      String depositAddress =
7          bitcoinAppKit.wallet().freshAddress(KeyPurpose.RECEIVE_FUNDS).toString();
8      possibleParticipants.add(new Participant(depositAddress));
9      return depositAddress;
10 }
11
12 public String getCurrentBlockHash() {
13     return
           bitcoinAppKit.chain().getChainHead().getHeader().getHash().toString();
```

```
14 }
15
16 public int getCurrentBlockHeight() {
17     return bitcoinAppKit.chain().getChainHead().getHeight();
18 }
```

Zeile 2 reicht die Daten des Topfs an die GUI Komponente weiter. Zeile 6 erzeugt eine neue Empfangsadresse und fügt einen weiteren möglichen Teilnehmer mit dieser Adresse der `possibleParticipants` Liste hinzu. Erst wenn eine Zahlung auf diese Adresse eingeht wird der Teilnehmer dem Topf hinzugefügt. Zeile 13 gibt den Blockhash des neusten Blocks der SPV Blockheader Kette zurück. Zeile 17 gibt die Blocknummer des neusten Blocks der SPV Blockheader Kette zurück.

Bitcoin Netzwerk Events

Immer wenn der SVP Node eine Transaktion auf eine vorher mittels der `bitcoinAppKit.wallet().freshAddress()` erzeugten Adresse empfängt, wird die `onCoinsReceived` Methode der Klasse `CoinsReceivedListener` aufgerufen.

```
1  @Override
2  public void onCoinsReceived(Wallet wallet, Transaction txn, Coin
       prevBalance, Coin newBalance) {
3    log.debug("Transaction details: " + txn.toString());
4    Coin value = txn.getValueSentToMe(wallet);
5    Pot currentPot = service.getCurrentPot();
6    if (currentPot.getExpectedBettingAmount() > value.getValue()) {
7      log.warn("Player did not pay enough.");
8      return;
9    }
10   List<Participant> participants =
         service.getPossibleParticipants();
11   NetworkParameters params = service.getAppKit().params();
12   for (TransactionOutput txnOutput : txn.getOutputs()) {
13     Address a = txnOutput.getAddressFromP2PKHScript(params);
14     String address = a.toString();
15     for (Participant participant : participants) {
16       String depositAddress = participant.getDepositAddress();
17       if (depositAddress.equals(address)) {
18         log.info("Received " + value.toFriendlyString() + " coins
               from " + participant.toString());
19         participant.setReceivedAmount(value.getValue());
20         String fromAddress =
               txn.getInput(0).getFromAddress().toString();
21         participant.setPayoutAddress(fromAddress);
22         wallet.addTransactionConfidenceEventListener(new
               TxnConfidenceListener(service, txn, participant));
23       }
24     }
25   }
26 }
```

Zunächst wird in Zeile 6 geprüft, ob der eingegangene Zahlungsbetrag mindestens so hoch ist, wie von aktuellen Topf gefordert. Ist dies nicht der Fall, wird die Zahlung ignoriert. Die Methode iteriert über die Output Adressen der Transaktion

(Zeile 12) und prüft mit welcher Einzahlungsadresse der Teilnehmer (Teile 17) diese übereinstimmt. Handelt es sich bei der Output Adresse um keine Einzahlungsadresse eines Teilnehmers, wird diese ignoriert, da es sich wohl um eine Wechselgeldadresse eines Teilnehmers handeln muss. Hat man den Teilnehmer identifiziert, wird aus der Transaktion die Auszahlungsadresse berechnet und dem Teilnehmer der empfangene Geldbetrag gutgeschrieben (Zeile 19-21). Da es sich um eine gültige jedoch noch unbestätigte Transaktion handelt, wird der Teilnehmer erst nachdem die Transaktion in einen gültigen Block aufgenommen wurde zum Topf hinzugefügt. Zeile 22 erzeugt einen `TxnConfidenceListener` der diese Aufgabe übernimmt.

```
1  @Override
2  public void onTransactionConfidenceChanged(Wallet wallet,
       Transaction txn) {
3    if (txn.equals(targetTxn)) {
4      log.debug("onTransactionConfidenceChanged Tx: " +
           txn.getHash());
5      switch (txn.getConfidence().getConfidenceType()) {
6        case PENDING:
7          // unconfirmed but should be included shortly
8          break;
9        case BUILDING:
10         // transaction is included in the best chain
11         Pot currentPot = service.getCurrentPot();
12         participant.setPotIndex(currentPot.getNbrOfParticipants());
13         currentPot.addParticipant(participant);
14         if (currentPot.isFull()) {
15           service.closeCurrentPot(new Date());
16         }
17         wallet.removeTransactionConfidenceEventListener(this);
18         break;
19       case IN_CONFLICT:
20         log.warn("possible double spend of txn " +
               txn.getHashAsString());
21         break;
22       case DEAD:
23         log.warn("txn " + txn.getHashAsString()+ " won't confirm
               unless there is another re-org");
24         wallet.removeTransactionConfidenceEventListener(this);
25         break;
26     }
27   }
28 }
```

Die `onTransactionConfidenceChanged` Methode wird jedes mal aufgerufen wenn dem SPV Client neue Daten zur Transaktion vorliegen. BitcoinJ unterscheidet zwischen vier verschiedenen `ConfidenceType`:

- PENDING: Bedeutet, dass die Transaktion noch unbestätigt ist und der SPV Client darauf wartet, dass er einen Block erhält in dem die Transaktion enthalten ist.
- BUILDING: Bedeutet, dass die Transaktion bereits in die Blockchain aufgenommen wurde. Durch den Aufruf von `transaction.getConfidence().getAppearedAtChainHeight()` kann man abfragen wie tief die Transaktion bereits in der Blockchain steckt. Diese Methode gibt zurück wie viele Blöcke bereits auf den Block, der die Transaktion enthält, aufbauen. Die Glücksspielanwendung betrachtet eine Transaktion

ab dem Zeitpunkt als final, ab dem sie in einen gültigen Block aufgenommen wurde.[7] Geschieht dies, wird der Teilnehmer der Transaktion in den Topf hinzugefügt

- IN CONFLICT: In diesem Fall hat der SPV Client zwei Transaktionen erhalten, die versuchen den gleichen Transaction-Output auszugeben. Man spricht von einem sogenannten "double spend" Angriff.
- DEAD: Transaktionen die diesen Status erhalten, können nicht mehr bestätigt werden, außer es kommt zu einer Blockchain Restrukturierung.

Immer wenn der SVP Node einen neuen besten Block findet, den er vorne an die Header Kette anhängen kann, wird die `notifyNewBestBlock` Methode aufgerufen.

```
1  @Override
2  public void notifyNewBestBlock(StoredBlock block) throws
       VerificationException {
3      log.info("New Block height = " + block.getHeight() + " hash = "
4              + block.getHeader().getHash().toString());
5      List<Pot> unfinishedPots =
           getUnfinishedPots(service.getClosedPots());
6      for (Pot pot : unfinishedPots) {
7          long potId = pot.getCreateTime().getTime();
8          int payoutBlockHeight = pot.getPayoutBlockHeight();
9          if (payoutBlockHeight > block.getHeight()) {
10             log.info("Pot[" + potId + "] can not be handled yet.");
11         } else if (payoutBlockHeight == block.getHeight()) {
12             log.info("Pot[" + potId + "] select temorary Winner.");
13             Block tmpPayoutBlock = new
                   ExtendedBlock(block.getHeader().getHash().toString());
14             pot.setPayoutBlock(tmpPayoutBlock);
15             selectWinner(pot, tmpPayoutBlock);
16         } else {
17             log.info("Pot[" + potId + "] select final winner.");
18             try {
19                 StoredBlock correctBlock =
                       getPastBlock(payoutBlockHeight, block);
20                 String payoutBlockHash =
                       correctBlock.getHeader().getHash().toString();
21                 Block finalPayoutBlock = new
                       ExtendedBlock(payoutBlockHash);
22                 pot.setPayoutBlock(finalPayoutBlock);
23                 Participant winner = selectWinner(pot,
                       finalPayoutBlock);
24                 log.info(winner.getDepositAddress() + " wins pot["
                       + potId + "].");
25                 startPayoutThread(pot);
26             } catch (BlockStoreException e) {
27                 log.info("Couldn't select final winner of Pot[" +
                       potId + "]:" + e.getMessage(), e);
28             }
29         }
30     }
31 }
```

[7] Vor der Auszahlung kann die Anwendung erneut nachprüfen, ob es eine Restrukturierung der Blockchain durch einen Blockchainfork gab, und ob es dadurch möglicherweise eine Transaktionen noch nicht in die längste Blockkette geschafft hat.

Diese Methode iteriert über jeden bereits geschlossenen Topf für den noch keine Auszahlung stattgefunden hat und unterscheidet 3 Fälle:

1. Der neue Block hat eine Blocknummer, die kleiner ist als die Blocknummer, die den Top entscheidet. In diesem Fall passiert nichts.

2. Der neue Block entscheidet den Topf, da die Blocknummer des Blocks gleich der PayoutHeigth des Topfs ist. Der Gewinner des Topfs wird selektiert, es findet allerdings keine Auszahlung statt. Die finale Auszahlung findet aus Sicherheitsgründen erst im nächsten Fall statt.

3. In diesem Fall gibt es mindestens einen Block, der auf dem Payout-Block des Topfs aufbaut. Ab diesem Zeitpunkt betrachtet die Anwendung den Gewinner als final.[8] Daher wird der Gewinner des Topfs überschrieben und die Auszahlung in einem neuen `Thread` gestartet.

Die Klasse `ExtendedBlock` teilt den Blockhash zur Anzeige in der GUI in die Werte `perfix`, `lastDigit` und `suffix`. Die Variable `lastDigit` speichert die letzte numerische Stelle des Blockhashs und wird zur Gewinnerauswahl verwendet.

```
1  public class ExtendedBlock extends Block {
2
3      private String prefix;
4      private String suffix;
5
6      public ExtendedBlock(String blockHash) {
7          super(blockHash, -1);
8          int position = blockHash.length() - 1;
9          while (position > 0) {
10             char c = blockHash.charAt(position);
11             int value = (int) c;
12             if (value >= 48 && value <= 57) {// numeric
13                 this.lastDigit =
                       Integer.parseInt(String.valueOf(c));
14                 break;
15             }
16             position--;
17         }
18         this.prefix = blockHash.substring(0, position);
19         this.suffix = blockHash.substring(position + 1,
               blockHash.length());
20     }
21 }
```

Auszahlungen

Auszahlung werden in einem eigenen Thread abgehandelt. Die Klasse `PayoutThread` ruft dazu die `payout` Methode auf.

```
1  private void payout(Pot pot) throws InsufficientMoneyException,
       InterruptedException, ExecutionException {
2      if (pot.isPayoutStarted()) {
3          log.error("Payout already started: " +
               pot.getPayoutTxnId() + " - " + pot.getPayoutError());
```

[8] An dieser Stelle kann man natürlich auch aus Sicherheitsgründen noch mehrere Blöcke abwarten, bevor die Anwendung eine Auszahlung startet.

```
4       } else {
5           pot.setPayoutStarted(true);
6           Address winnerAddress = new
                Address(bitcoinService.getNetworkParams(),
                pot.getWinner().getPayoutAddress());
7           Coin potValue =
                Coin.SATOSHI.multiply(pot.getParticipants().size() *
                pot.getExpectedBettingamount());
8           Wallet.SendResult result =
                bitcoinService.getAppKit().wallet()
9           .sendCoins(bitcoinService.getAppKit().peerGroup(),
                winnerAddress, potValue);
10          String txnId = result.tx.getHash().toString();
11          pot.setPayoutTxnId(txnId);
12          log.info("Payout TXN ID = " + txnId);
13          Transaction transaction = result.broadcastComplete.get();
14          log(transaction);
15      }
16  }
```

Die Methode prüft erst, dass die Auszahlung noch nicht gestartet wurde. Ist dies der Fall, wird der auszuzahlende Betrag berechnet und an die Adresse des Gewinners überwiesen. Anschließen wird die ID der Auszahlungstransaktion in den Topf geschrieben. Während der Auszahlung können von BitcoinJ 3 verschiedene Fehler auftreten:

1. InsufficientMoneyException: Falls die von der Wallet verwalteten Adressen nicht genug Bitcoin für die Auszahlung besitzen.
2. InterruptedException: Falls der Java Thread unterbrochen wird.
3. ExecutionException: Falls es zu einem unerwarteten Fehler bei der Ausführung kommt.

Sollte es bei der Auszahlung ein Problem geben, wird die Exception gefangen und abgespeichert.

2.3.5 Grafische Benutzeroberfläche

Die graphische Oberfläche der Anwendung ist mit dem Tapestry Framework von Apache realisiert. Da die GUI Komponente nur die Daten visualisiert, wird an dieser Stelle auf eine genauere Betrachtung verzichtet und lediglich die Benutzeroberfläche gezeigt.

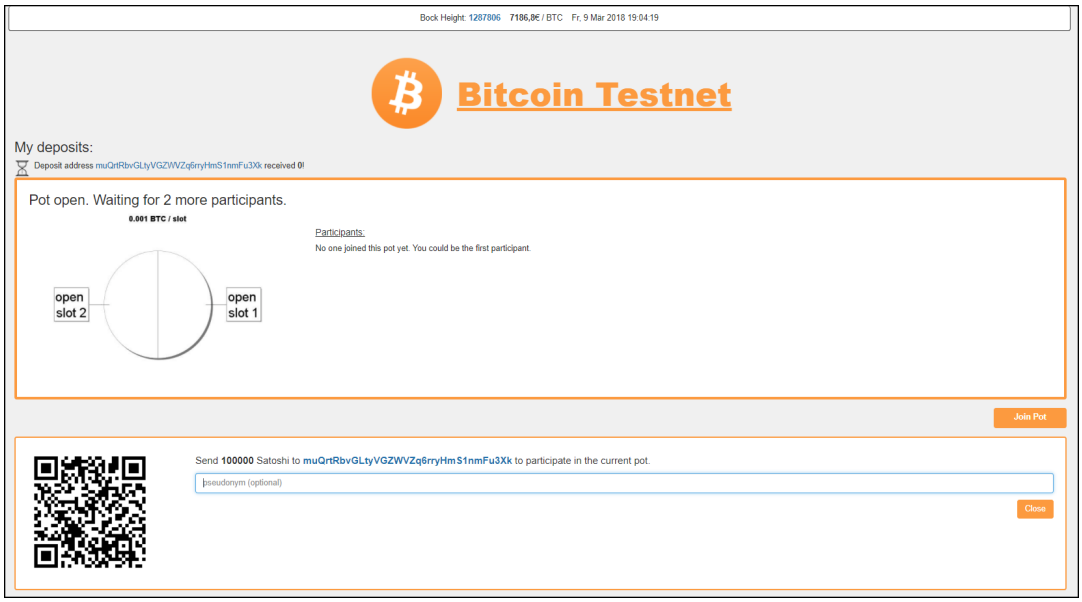


ABBILDUNG 2.21: Leerer Topf

Abbildung 2.21 zeigt einen Topf mit 2 freien Plätzen. Um dem Spiel beizutreten, muss der Spieler den Betrag von 0,001 Bitcoin an die angezeigte Adresse senden. Der angezeigte QR-Code erleichtert dem Spieler die Übertragung dieser Daten in das Überweisungsformular seines Smartphone Wallets. Das **B**itcoin **I**mprovement **P**roposal Nummer 21[3] legt fest, in welchem Format diese Daten kodiert werden müssen, damit beliebige Bitcoin Clients diese korrekt auslesen können. Folgende Daten sind in dem QR Code enthalten:
”bitcoin:muQrtRbvGLtyVGZWVZq6rryHmS1nmFu3Xk?amount=0.01”

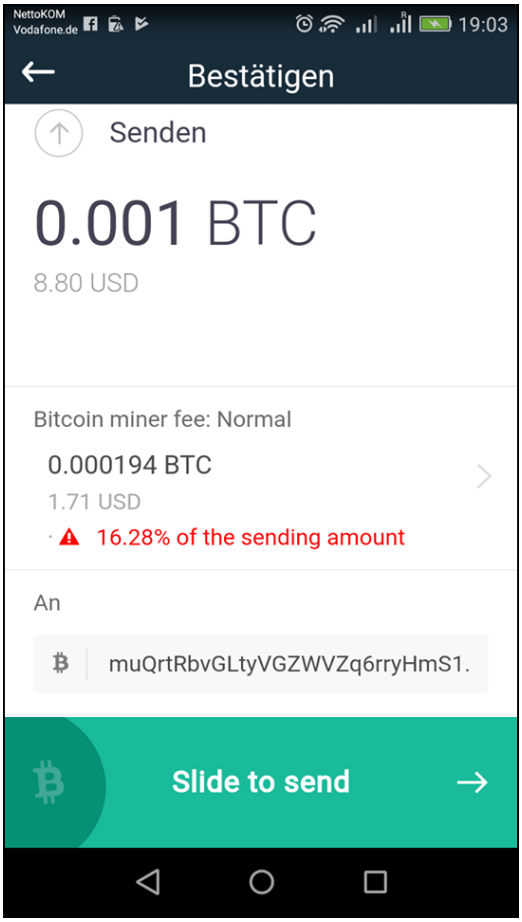


ABBILDUNG 2.22: Smartphone Überweisungsformular

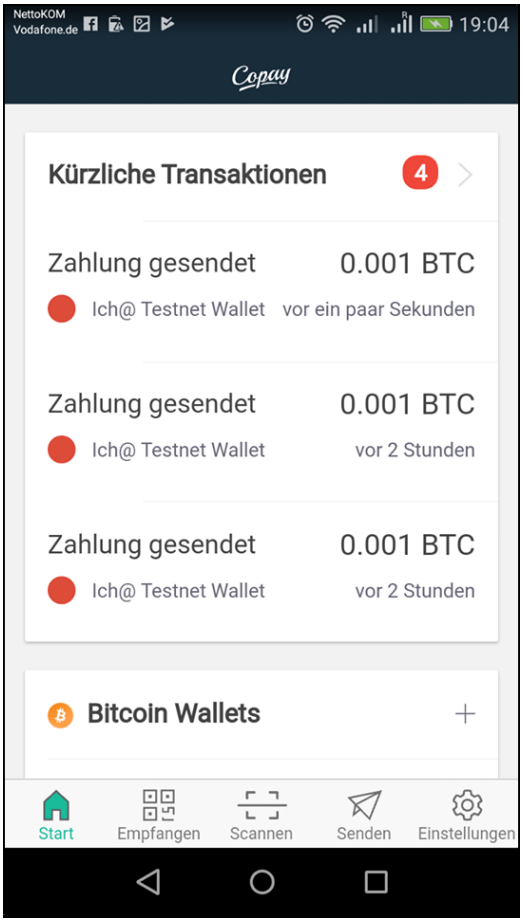


ABBILDUNG 2.23: Zahlungsbestätigung

Abbildungen 2.22 und 2.23 zeigen das Bitcoin CoPay Wallet[9]. Dieses erlaubt es Zahlungen an das Bitcoin Testnetz zu senden. Nachdem der Benutzer den QR-Code der Glücksspielanwendung mit seinem Smartphone abgescannt hat, erscheint sowohl der Betrag als auch die Empfangsadresse vor-ausgefüllt im Überweisungsformular. Die Wallet berechnet automatisch eine passende Transaktionsgebühr. Der Spieler prüft lediglich die Adresse und den Betrag und autorisiert anschließend die Zahlung.

[9] https://copay.io/

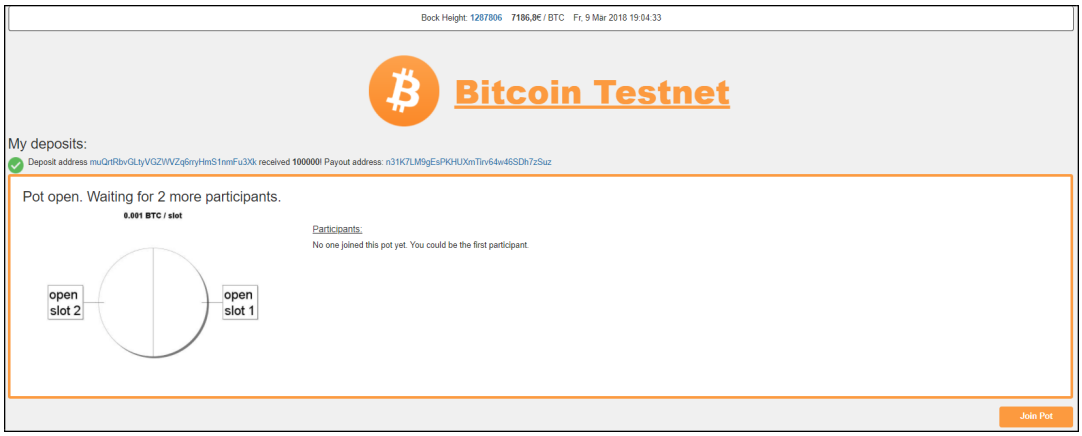


ABBILDUNG 2.24: Transaktion empfangen

Sobald die Anwendung die Transaktion empfängt, zeigt sie dies durch einen grünen Haken an. Dies ist in Abbildungen 2.24 zu sehen. Zu diesem Zeitpunkt handelt es sich um eine unbestätigte Transaktion, die noch in keinen Bock der Blockchain aufgenommen wurde.

ABBILDUNG 2.25: Spieler zu Topf hinzugefügt.

Sobald die Anwendung einen neuen Block empfängt, der die Transaktion enthält, gilt die Transaktion als bestätigt und der Spieler wird zum Topf hinzugefügt. Nun gibt es nur noch einen offenen Platz im Topf.

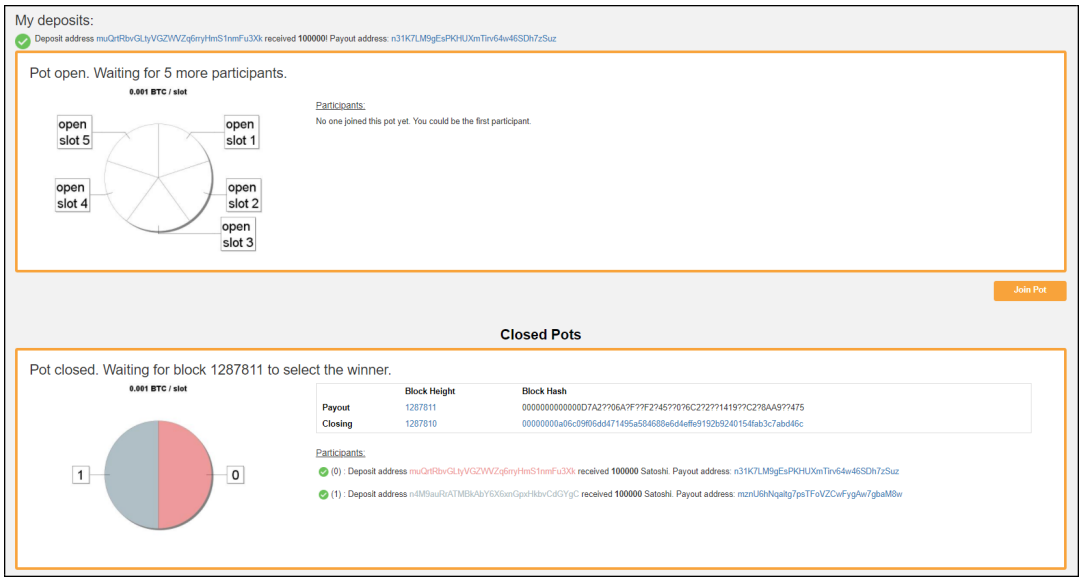


ABBILDUNG 2.26: Topf geschlossen

Nachdem ein weiterer Spieler in den Topf eingezahlt hat, ist der Topf voll. Die Anwendung schließt den aktuellen Topf und öffnet einen neuen Topf. Der in 2.26 gezeigte neue Topf hat fünf freie Plätze. Die Anwendung wartet nun auf den Payout Block, um den Gewinner des geschlossenen Topfs festzulegen. Da der Blockhash noch nicht feststeht, zeigt die Anwendung die Animation eines sich sehr schnell wechselnden Block Hash, der mehrere Fragezeichen enthält, an.

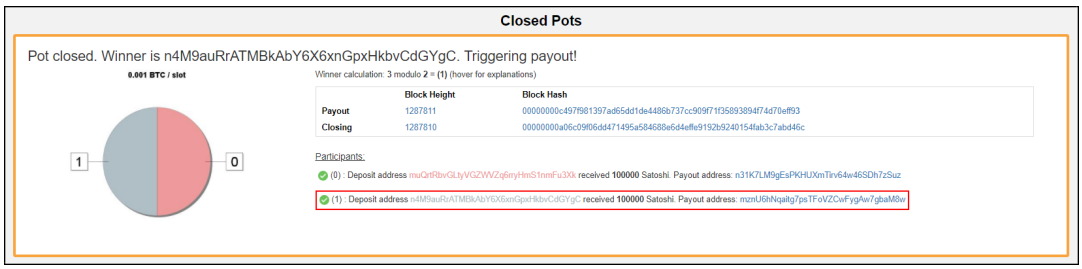


ABBILDUNG 2.27: Gewinner ermittelt

Sobald der entscheidende Block empfangen wird, wird der voraussichtliche Gewinner des Topfs durch ein rotes Rechteck markiert. Außerdem zeigt die Anwendung durch welche Berechnung der Gewinner festgelegt wurde. Nun wartet die Anwendung bis ein weiterer Block gefunden wird, bevor sie die Auszahlung an den Gewinner startet.

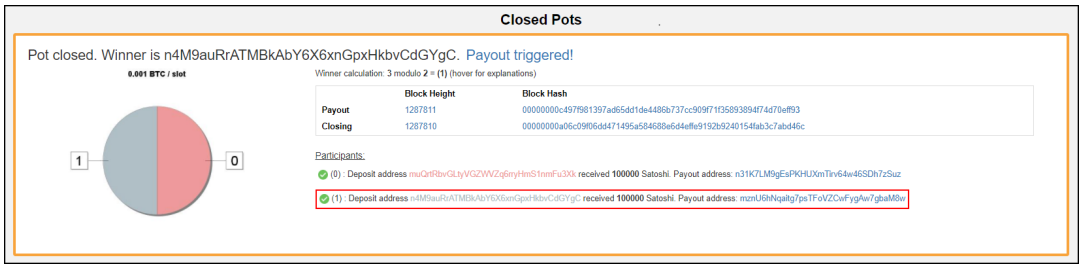


ABBILDUNG 2.28: Auszahlung beendet

Klickt der Benutzer auf den *Payout triggered* Link aus Abbildung 2.28, gelangt er zu einem Blockexplorer. Dieser zeigt ihm die Transaktionsdetails an. Eine genauere Betrachtung findet in Abschnitt 2.4.6 statt.

2.4 Evaluation

2.4.1 Prüfung der Anforderungen

Dieser Abschnitt behandelt in wie weit das beschriebene Konzept die in Kapitel 1 aufgelisteten Anforderungen erfüllt. Die jeweilige Anforderung wird zunächst wiederholt und anschließend genauer untersucht.

1) Transparente Einzahlungen

Die Einzahlung jedes Endnutzers ist für jeden anderen Endnutzer nachprüfbar.

Diese Anforderung ist erfüllt, da jede Transaktion in der lokalen Datenbank jedes Peet-to-Peer Netzwerkteilnehmers aufgezeichnet wird. Auf der Webseite [5] kann man die Bitcoin Blockchain mithilfe eines sogenannten Blockchain-Exploers durchsuchen. Mit diesem Werkzeug kann man die Blöcke und die darin enthaltenen Transaktionen untersuchen. Nutzt man den Explorer einer Drittpartei, muss man darauf vertrauen, dass dieser auch den "wahren" Status der Blockchain anzeigt. Um dieses Risiko zu vermeiden kann jeder Teilnehmer mithilfe eines eigenen Bitcoin Full Node am Netzwerk teilnehmen. Dieser speichert die gesamte Blockchain und prüft alle Transaktionen und Blöcke gegen die Konsensregeln.
Der Bitcoin Full Node stellt eine API bereit, über die man den aktuellen Status der Blockchain abfragen kann. Der Befehl **getblockchaininfo** liefert den aktuellen Zustand der Blockchain zurück. Dieser beinhaltet die Blocknummer des neusten Blocks und dessen Blockhash. Der Befehl **gettransaction** gefolgt von der Transaktions-ID liefert Details über eine Transaktion. Die Webseite [2] dokumentiert diese Schnittstelle detailliert.

2) Gewinnerauswahl durch Zufallsfaktor

Die Auswahl des Gewinners ist von einem zufälligen Faktor abhängig, auf den weder die Anwendung noch die Endnutzer einen Einfluss haben.

Diese Anforderung wird nur bedingt erfüllt, da ein Teilnehmer des Peer-to-Peer Netzwerks sowohl ein Spieler als auch ein Miner sein kann. Ist dies der Fall besteht die Möglichkeit, dass der Miner einen validen Blockhash verwirft, sobald er merkt, dass er durch diesen Blockhash nicht zum Gewinner des Geldtopfes wird. Verwirft der Teilnehmer einen Blockhash, riskiert er den dadurch ausgeschütteten Blockreward. Ein solcher Angriff ist für einen Miner nur rentabel, falls die Spieleinsätze des Geldtopfes den Blockreward um ein vielfaches übersteigen. Betrachten wir dazu das Bitcoin Netzwerk Anfang Februar 2018. Der Preis pro Bitcoin beträgt 8000 Euro. Der Mining Reward liegt bei 12,5 Bitcoin pro Block[10]. Für das Lösen eines gültigen Blocks erhält ein Miner somit 100000 Euro. Angenommen ein Miner besitzt 20 Prozent der Hashrate des gesamten Bitcoin-Netzwerks und nimmt an einem Topf mit 2 Personen teil. Dies bedeutet, dass sowohl der Miner als auch der andere Teilnehmer eine Gewinnwahrscheinlichkeit von 0,5 für einen zufälligen Blockhash haben. Da der Miner eine Hashrate von 20 Prozent hat, liegt die Wahrscheinlichkeit das der Miner den nächsten Block findet bei 0,2. Falls er den gefundene Blockhash verwirft, da er durch diesen seinen Glücksspieleinsatz verlieren würde, muss er es

[10] Dieses Beispiel vernachlässigt die durch das Minen des Blocks erhaltenen Transaktionsgebühren.

schaffen vor einem anderen Miner einen weiteren Blockhash zu berechnen. Ansonsten verliert er den Blockreward. Die Wahrscheinlichkeit das der Miner zwei gültige Blockhashs hintereinander findet, liegt bei 0,2 * 0,2 = 0,04 und ist somit verschwindend gering.

3) Nachprüfbarkeit des Zufallsfaktor

Jeder Endnutzer kann die Echtheit des zufälligen Faktors eigenständig nachprüfen.

Da das Verfahren der Gewinnerauswahl im Vorhinein festgelegt ist und die Reihenfolge der Einzahlungstransaktionen in der Blockchain festgeschrieben steht, kann jeder Teilnehmer die Berechnung des Gewinners eigenständig nachvollziehen. Der Blockhash der die Grundlage für die Gewinnerauswahl liefert kann durch die Verwendung eines Bockchain-Explorers oder eines Full Nodes nachgeprüft werden.

4) Transparente Auszahlungen

Die Auszahlung an den Gewinner muss transparent und somit für jeden Endnutzer nachprüfbar sein.

Genau wie die Einzahlungen ist auch die Auszahlung für jeden Spieler mithilfe eines Blockchain-Explorers oder eines Bitcoin Full Nodes möglich. Jeder Teilnehmer kann somit für alle bereits abgeschlossenen Spiele nachprüfen ob die Anwendung sich korrekt verhalten und eine Auszahlung getätigt hat.

5) Fairheit des Spiels

Jeder Endnutzer besitzt die gleiche Gewinnwahrscheinlichkeit und niemand wird benachteiligt.

Die Zuordnung der Spieler auf die Gewinnzahlen ist durch die Reihenfolge der Transaktionen in der Blockchain festgeschrieben. Eine nachträgliche Veränderung dieser Reihenfolge ist weder durch die Nutzer, noch durch die Glücksspielanwendung möglich.[11]

Damit keiner der Spieler einen Vorteil hat, muss jeder Topf-Platz die gleiche Gewinnwahrscheinlichkeit haben. Dies ist gegeben, falls a) jeder Teilnehmer die gleiche Anzahl Gewinnzahlen zugeordnet bekommt und b) falls die möglichen Blockhash-Werte für die Gewinnerauswahl gleichverteilt sind.

a) Die Gewinnerauswahl kann entweder wie im Konzept beschrieben durch den gesamten Blockhash-Wert oder wie im Beispiel auf Basis der letzten Blockziffer vorgenommen werden. Beide Methoden haben Vor- und Nachteile.
Variante eins erlaubt beliebige Topfgrößen, ist dafür aber schwieriger für den Endnutzer zu verifizieren. Die Verifizierung erfordert die Konvertierung des Blockhashs ins Dezimalsystem und eine Modulo-Rechnung einer sehr große Zahl.
Variante zwei ist dagegen leicht zu verifizieren, erlaubt allerdings nur die Topfgrößen zwei, fünf und zehn. Bei der Topfgröße von zwei sind beiden Spielern fünf Gewinnzahlen zugeordnet. Bei der Topfgröße von fünf besitzt jeder Teilnehmer genau 2 Gewinnzahlen. Bei einer Topfgröße von zehn wird jedem Teilnehmer genau eine

[11] Eine Veränderung der Reihenfolge ist nur durch einen sogenannten Blockchain-Fork möglich. Kapitel 2.4.5 erörtert welche Auswirkungen dies auf die Glücksspielanwendung hat.

Gewinnzahl zugeordnet. Nimmt man hingegen eine Topfgröße von 1,3,4,6,7,8 und 9 führt dies dazu das manche Teilnehmer eine signifikant höhere Gewinnchance haben. Bei der Topfgröße von 3 sind die Gewinnzahlen durch die Modulo-Funktion folgendermaßen verteilt:

- Spieler 1 hat die Gewinnzahlen 0, 3 und 9.
- Spieler 2 hat die Gewinnzahlen 1 und 4.
- Spieler 3 hat die Gewinnzahlen 2, 5 und 8.

Somit haben Spieler 1 und 3 eine Gewinnwahrscheinlichkeit von $\frac{3}{10}$, Spieler 2 hingegen nur eine Gewinnwahrscheinlichkeit von $\frac{2}{10}$.

Es kommt also vor, dass eine Teilmenge der Spieler genau eine Gewinnzahl mehr als der Rest der Teilnehmer hat. Nimmt man den gesamten Blockhash zur Gewinnerauswahl ist die dadurch entstehende Ungerechtigkeit verschwindend gering und kann vernachlässigt werden. Dies ist der Fall, da die aus dem Blockhash resultierende Dezimalzahl in der Praxis sehr groß ist und jeder Spieler somit mehrere Millionen von Gewinnzahlen hat.

b) Der Blockhash eines Blocks wird durch die verwendete kryptographische Hashfunktion der Kryptowährung festgelegt. Die Verteilung der Werte ist somit von der verwendeten kryptographische Hashfunktion abhängig.
Eine kryptografische Hashfunktion ist eine stark kollisionsresistente Einweg-Hashfunktion. Eine Hashfunktion h heißt

- Einwegfunktion genau dann, wenn es schwierig ist, zu gegebenem $_{Y0}$ ein $_{X0}$ zu finden mit $h(x_0) = y_0$.
- schwach kollisionsresistent genau dann, wenn es schwierig ist, zu einem gegebenen x ein x' zu finden mit $h(x) = h(x')$.
- stark kollisionsresistent genau dann, wenn es schwierig ist, x und x' zu finden mit $x \neq xx$ und $h(x) = h(xx)$.

Die Eigenschaften der starken Kollisionsresistenz und der Einwegfunktion sagen nichts über die Verteilung der ausgegebenen Werte aus. Bei der Auswahl der Kryptowährung muss also gesondert auf die Verteilung der verwendete Hashfunktion geachtet werden. Sollte die verwendete kryptographische Hashfunktion keine Gleichverteilung liefern, kann der Blockhash dennoch den nötigen Zufall liefern indem dieser mit einer geeigneten Hashfunktion erneut gehasht wird.

Bitcoin verwendet die kryptographische Hashfunktion SHA256. Die folgende Monte-Carlo-Simulation zeigt, dass die Resultate der SHA256 gleichverteilt sind.

```
h=SHA256 n=1000000
for i 1 -> n
    hash = h(i);
    result[lastDigit(hash)]++
```

```
Ausgabe:
result[0] =  99765
result[1] = 100488
result[2] =  99913
result[3] = 100745
result[4] = 100272
result[5] =  99649
result[6] =  99430
result[7] =  99788
result[8] =  99666
result[9] = 100284
```

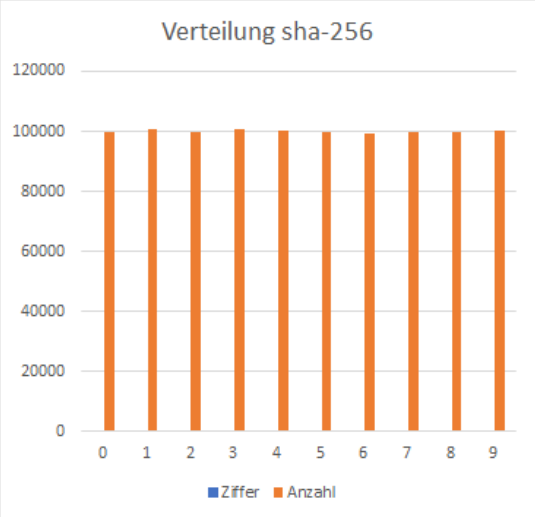


ABBILDUNG 2.29: Verteilung der SHA256 Hashfunktion

2.4.2 Betrugsmöglichkeiten

Dieser Abschnitt betrachtet in wie weit die Glücksspielanwendung potentielle Spieler betrügen kann, sollte sie gehackt und zu ausschließlich diesem Zweck modifiziert werden. Die Anwendung hat die volle Kontrolle darüber welche Ausgabe sie dem Benutzer anzeigt. Sie hat allerdings keine Kontrolle über den Status der Blockchain.

Die Anwendung könnte beispielsweise anzeigen, dass der Topf nach der Einzahlung durch einen Spieler immer noch leer ist. Die Transaktion auf die Einzahlungsadresse existiert dann zwar in der Blockchain allerdings reagiert die Anwendung nicht entsprechend. Dies hat zur Folge, dass jeder Spieler der eine Einzahlung tätigt, sein Geld verliert. Allerdings merkt der Benutzer dies und kann somit eine weitere Verwendung der Anwendung unterlassen. Ein solch plumper Manipulationsversuch fällt somit direkt auf.

Ein weitere Betrugsmöglichkeit ist, dass die Glücksspielanwendung sich bis zur Gewinnerauswahl korrekt verhält, dann allerdings keine Auszahlung tätigt. Alle einzahlenden Spieler merken den Betrug, verlieren aber dennoch ihr Geld.

Bei beiden vorgestellten Betrugsversuchen fällt der Betrug immer mindestens einem Spieler auf. Die Verwendung einer solchen Anwendung macht nur Sinn, falls man den Betreiber des Services kennt und diesen im Zweifel juristisch haftbar machen kann.

Das folgende Kapitel betrachtet wie sogenannte ”Smart Contract”s dieses Problem lösen. Ein Smart Contract erlauben es die Geschäftslogik der Glücksspielanwendung in die Blockchain zu schreiben. Die Geschäftslogik wird somit nicht mehr von der Anwendung, sondern von jedem Teilnehmer des Peer-to-Peer Netzwerks ausgeführt.

2.4.3 Angriff durch Miner

2.4.4 Blockchain Mining Varianz

Der Begriff Blockchain Mining Varianz beschreibt den Umstand, dass die Miner des Netzwerkes entweder Glück oder Pech bei der Suche nach dem nächsten gültigen Blockhash haben können. Bei Bitcoin gibt es daher nicht genau alle 10 Minuten einen neuen Block, sonder durchschnittlich alle 10 Minuten. Für die Glücksspielanwendung bedeutet dies, dass die Zeit zwischen der letzten Einzahlung und der Auswahl des Gewinners variieren kann. In der Praxis kommt es vor, dass man 30 Minuten und mehr auf den nächsten Block warten muss. Dies ist für den Spieler eine recht lange Zeit. Der Forscher XXX schlägt in dem Proposal namens Bobtail ein Verfahren vor, dass die Blockchain Mining Varianz stark verringert. Dieser Vorschlag ist bisher allerdings noch nicht in den Bitcoin Sourcecode eingeflossen. Eine andere Möglichkeit die Wartezeit für den Spieler zu verringern ist es eine Kryptowährung mit einer geringeren Blockzeit zu verwenden. Beispiele hierfür sind LiteCoin[12] mit einer Blockzeit von 2,5 Minuten, die auf Privatsphäre spezialisierte Währung Monero[13] mit einer Blockzeit von 2 Minuten und Ethereum[14] mit einer Blockzeit von 12 Sekunden. Bei einer geringen Blockzeit kommt es häufiger zu sogenannten Blockchain Forks. TODO check Bobtail: A Proof-of-Work Target that Reduces Blockchain Mining Variance (Brian N. Levine) https://stanford2017.scalingbitcoin.org/presentations

2.4.5 Blockchain Forks

Blockchain Forks entstehen, falls 2 Miner unabhängig voneinander mehr oder weniger gleichzeitig einen validen Block finden. Beide Miner broadcasten ihren Block schnellstmöglich an die Teilnehmer des Peer-to-Peer Netzwerks. Aufgrund von Netzwerkverzögerungen kommt es nun dazu, dass ein Teil des Netzwerks Block 1 und der restliche Teil des Netzwerks Block 2 zuerst enthält. Beide Blockchain-Ketten sind nun gleich lang.

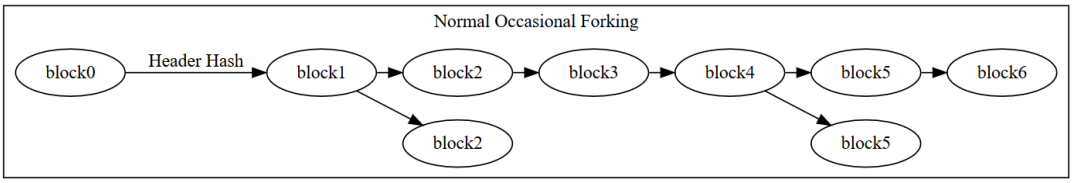


ABBILDUNG 2.30: Bitcoin Fork

Der nächste gefundene Block entscheidet, auf welche Kette sich das Netzwerk einigt[15]. Die Bitcoin Konsensregeln legen fest, dass die Teilnehmer des Netzwerks immer der längsten Kette, die somit am meisten Proof-of-Work beinhaltet, folgen. Dies erlaubt es jedem Bitcoin Knoten, ohne Trusted Third Party festzustellen, welche Version der Blockchain die echte ist. Forks kommen bei Bitcoin durch die hohe Hashrate des Netzwerks und die somit sehr hohe Difficulty recht selten vor. Die Webseite[16] zeigt an, wie oft gültige Blöcke gefunden werden, die es nicht in die Blockchain schaffen.

[12] https://litecoin.com/
[13] https://getmonero.org/
[14] https://www.ethereum.org/
[15] Unter der Annahme, dass es nicht erneut zu einem Blockchain Fork kommt.
[16] https://blockchain.info/orphaned-blocks

2.4.6 Auszahlungstransaktion

Die Glücksspielanwendung erzeugt für jede Einzahlung eine eigene Einzahlungsadresse statt für jeden Benutzer die gleiche Adresse zu verwenden. Dies hat den Vorteil, dass die Anwendung dem Benutzer anzeigen kann, dass sein Bitcoin Einzahlung eingegangen ist. Der Nachteil ist, dass dadurch die Größe der Auszahlungstransaktion steigt und man somit eine höhere Transaktionsgebühr zahlen muss. Im folgenden wird die Auszahlungstransaktion des Beispiels aus 2.3.5 betrachtet.

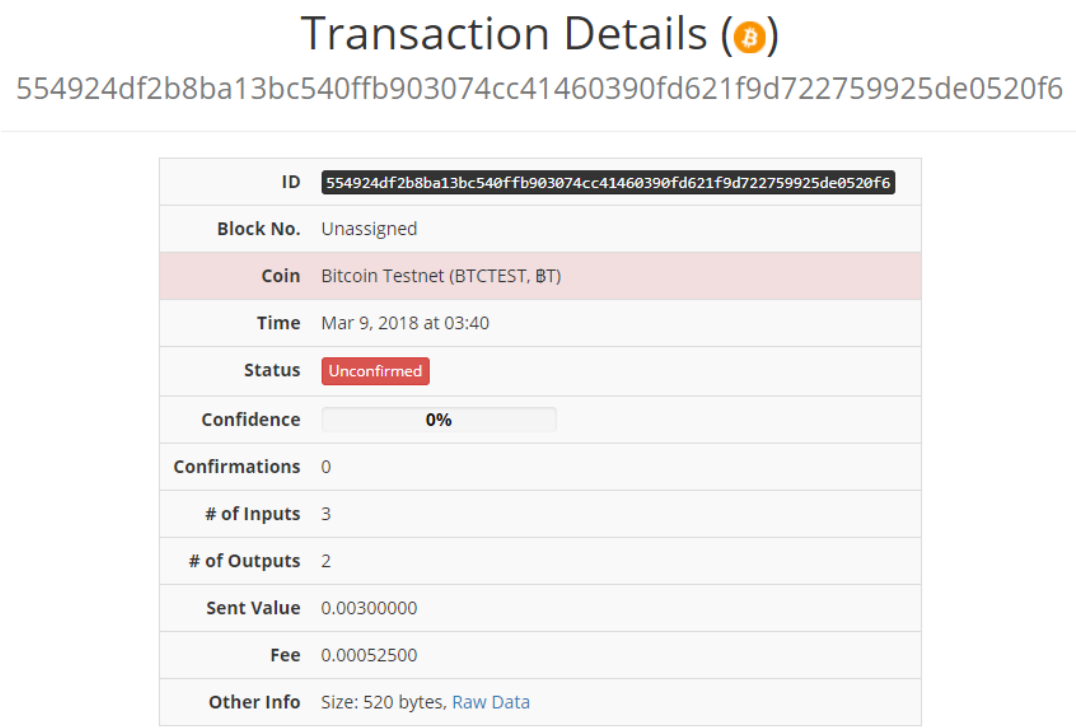


ABBILDUNG 2.31: Auszahlungstransaktion Details

Abbildung 2.31 zeigt in welchen Block die Transaktion aufgenommen wurde. Den Status, den Wert, die Transaktionsgebühr und die Größe der Transaktion. [17]

[17] Momentan zahlt die Glücksspielanwendung die Auszahlungstransaktionsgebühr aus eigener Tasche. Eigentlich müsste die Transaktionsgebühr von dem Gewinnbetrag abgezogen werden.

Inputs / Senders			
	Index	**Address**	**Value (BT)**
‹	**0**	mgyNmyrJExWoFRP4oGVM5Dt9BbZiqfAmBW	0.00100000
‹	**1**	mnnSU1EzgPgva3eFep8x2sNunoYCg71YsK	0.00100000
‹	**2**	mvGPrihYSaHz22XCmH78Cy97RSLhWUq6Xj	0.00100000
			0.00300000

Outputs / Receivers			
Index	**Address**	**Value (BT)**	
0	mznU6hNqaitg7psTFoVZCwFygAw7gbaM8w	0.00200000	›
1	n3MSpm7PhCLDx3CfmWWF55n8jQMRJprWvv	0.00047500	›
		0.00247500	

ABBILDUNG 2.32: Auszahlungstransaktion Inputs und Outputs

Abbildung 2.32 zeigt welche Inputs und Outputs für die Transaktion verwendet wurden. Output Adresse 0 gehört dem Wallet der Glücksspielanwendung und stellt die Wechselgeldadresse dar.

Input, Output Scripts		
Index		**Script**
Input	**0**	30440220170de6082da75e45ca88323ffc378efa7006ecbde8d666134b3aca3c644e343602205f02b0ea117cbdee5722b6cc8415bac8db0d8f185d1ebb839fc22f08bdef7053010237d4f81b1bc5c115d719126e14992456cd6b5b9d1b807f38fd641fb7a8ebfe8b
Input	**1**	30450221009602c4995821fdbaf6e0810d7e86b49efa49863865f9c3a09612722486476aad02205530dcd679d6025aa2dbfcc6a89fd93b415a1cea84901a55dc0f1811d209fd0701033ab2c32ea895b8a72756127b4d70ce0a4d0a6f2b8de08007ee98cb1b388674d7
Input	**2**	304402203b71208e03fe7dc9af6b4f7f928198b93ee4427ab7a8b8417f8d9d3088a9cd1c02203d3036e9b532ce05f96992b52e348b736463ea8d872045f2af229ec029d874980103c2151045e755d42a8e26820d18b47767f047133412bcef34d5270a8fe8482f9e
Output	**0**	OP_DUP OP_HASH160 d3598233c4436478702e9f9b91f98b4fd3b6464e OP_EQUALVERIFY OP_CHECKSIG
Output	**1**	OP_DUP OP_HASH160 ef866c3c9608fa1785922c94c30e14514be7bf81 OP_EQUALVERIFY OP_CHECKSIG

ABBILDUNG 2.33: Auszahlungstransaktion Skripts

Da die Anwendung für jeden Benutzer eine eigene Adresse generiert, muss die Anwendung in der Auszahlungstransaktion für jede Input Adresse eine gültige Signatur angeben. Abbildung 2.33 zeigt, dass die Transaktion dadurch wesentlich größer wird. Hier könnte in Zukunft die Verwendung sogenannter Schnorr Multi-Signaturen

aushelfen. Durch diese lassen sich alle Signaturen der Inputs durch eine einzige Signatur ersetzen. [18]

Kapitel 3

Zweiter Ansatz: Ethereum

3.1 Grundlagen

Ethereum ist genau wie Bitcoin ein Peer-to-Peer Netzwerk Protokoll, dass auf einer öffentlichen Blockchain basiert und einen Systemzustand verwaltet. Es handelt sich im Gegensatz zu Bitcoin um eine generalisierte Blockchain die nicht nur Finanztransaktionen speichern kann, sondern sogenannte Smart Contracts[1]. Ethereum sieht sich als eine Open Source Plattform, die es ermöglicht dezentrale Applikationen zu entwickeln und bereitzustellen.

3.1.1 Smart Contracts

Ethereum ermöglicht es Geschäftsprozesse in Form von Smart Contracts zu programmieren und von einem globalen dezentralen Netzwerk ausführen zu lassen. Ein Contract ist ein Programm bestehend aus einer Reihe von Anweisungen, die ausgeführt werden, sobald das Programm eine Nachricht in Form einer Transaktion erhält. Contracts haben die Möglichkeit Daten aus der Blockchain auszulesen und auf ihr Daten zu speichern. Smart Contracts können sowohl von Menschen als auch von anderen Contracts ausgelöst werden. Contracts können daher mit anderen Contracts über die vom Programmierer festgelegte Schnittstelle interagieren. Es handelt sich also nicht um einen Vertrag, der erfüllt werden muss sondern um ein öffentliches Stück Software, dass den Zustand eines Ethereum Accounts verwaltet.

3.1.2 Ethereum Accounts

Um Ethereum nutzen zu können braucht man einen Account. Ethereum Accounts bestehen aus:

- Einer 20 Byte lange Adresse,
- einem Kontostand in Ether(siehe Abschnitt 3.1.5),
- einem Nonce-Wert der sicherstellt, dass Transaktionen nur einmal ausgeführt werden können,
- Smart Contract Code (optional),
- und Speicherplatz (optional).

[1] Bitcoin Transaktionen beinhalten im Grunde auch Smart Contracts allerdings sind diese weniger Mächtig. Abschnitt 3.1.9 geht genauer auf die Unterschiede zwischen Bitcoin und Ethereum ein.

In Ethereum gibt es 2 Arten von Accounts. Die erste Art ist eine Adresse die durch den Besitzer des privaten Schlüssels kontrolliert wird. Diese Art von Account ist vergleichbar mit einer Bitcoin Adresse, die man durch eine Bitcoin Wallet Software kontrolliert. Die zweite Art ist ein Contract Account, der durch den Smart Contract Code kontrolliert wird. Der Smart Contract Code verwaltet eigenständig den Kontostand des Accounts und verhält sich genauso wie der Programmierer es festgelegt hat.

3.1.3 Transaktionen

Interaktionen mit dem Ethereum Netzwerk finden in Form von sogenannten Transaktionen statt. Transaktionen sind, durch den privaten Schlüssel eines Ethereum Account, signierte Datenpakete, die eine Nachricht(siehe Abschnitt 3.1.4) beinhalten. Transaktionen beinhalten:

- den Empfänger der Nachricht,
- eine digitale Signatur, die den Absender der Nachricht identifiziert,
- eine Anzahl an Ether die an den Empfänger versandt wird,
- Daten (Bytes), die vom Contract ausgelesen werden können (optional),
- einen sogenannten Gas-Wert, der die maximale Anzahl Rechenschritte festlegt, die die Transaktion bei der Ausführung nutzen darf
- und den GAS-Preis, der festlegt, wie viel der Absender bereit ist für die Ausführung eines Rechenschritts der Transaktion zu zahlen.

Es gibt zwei verschiedene Arten von Transaktion. Sie unterscheiden sich durch die in der Transaktion angegebene Empfangsadresse. Wird die Empfangsadresse durch einen privaten Schlüssel kontrolliert, handelt es sich lediglich um eine Finanztransaktion, die Ether vom Sender zum Empfänger Account transferiert. Anderenfalls handelt es sich um eine Transaktion, die den Contract Code der Empfangsadresse ausführt. Welche Funktion ausgeführt werden soll, wird vom Sender im Datenfeld der Transaktion kodiert.

3.1.4 Nachrichten

Bei Nachrichten handelt es sich um virtuelle Objekte, die im Gegensatz zu Transaktionen nicht abgespeichert werden, sondern lediglich in der Ethereum Virtual Machine verwendet werden. Ein weiterer Unterschied ist, dass Transaktionen im Gegensatz zu Nachrichten immer von ”außen” erzeugt werden. Nachrichten können hingegen auch von Smart Contracts erzeugt werden. Nachrichten bestehen aus einem impliziten Sender, einem Empfänger, einem Ether-Betrag, einem Gas-Wert und einem optionalen Datenfeld. Genau wie bei Transaktionen führt das versenden von Nachrichten zu der Ausführung des Conrtact Codes der Empfangsadresse.

3.1.5 Ether

Ether ist die Kryptowährung des Ethereum Netzwerks. Sie entsteht wie bei Bitcoin durch Mining und wird benutzt um für Transaktionsgebühren und die Ausführung von Contract Code zu bezahlen. Ether bildet durch die Zuhilfenahme des Gas-Werts und Gas-Preis den anti Spam Schutz des Ethereum Netzwerks.

Gas-Wert

Da Smart Contracts Schleifen beinhalten und andere Smart Contracts durch Nachrichten aufrufen können, muss sichergestellt werden, dass die Ausführung terminiert. Da es für den Sender der Transaktion schwierig, in manchen Fällen sogar unmöglich[2] ist, die Anzahlt Schritte der Ausführung im Vorhinein zu wissen, legt der Absender durch den Gas-Wert die maximale Anzahl Rechenschritte fest, die die Transaktion bei der Ausführung dauern darf.

Gas-Preis

Durch den Gas-Preis gibt der Sender einer Transaktion an, wie viel Ether er pro Rechenschritt bereit ist zu bezahlen. Die Ausführung einer Transaktion kann einen Ethereum Account somit um maximal `Gas-Wert * Gas-Preis` Ether belasten.

3.1.6 Ethereum Virtual Machine

In der Ethereum Virtual Machine wird der Smart Contract Code ausgeführt. Sie arbeitet die Anweisungen des Contracts der Reihe nach ab und bricht ab, falls der Gas-Wert nicht für die Ausführung der Transaktion ausreicht. Durch das sequenzielle Abarbeiten der Transaktionen eines Blocks wird der Systemzustand nach und nach von der Ethereum Virtual Machine angepasst.

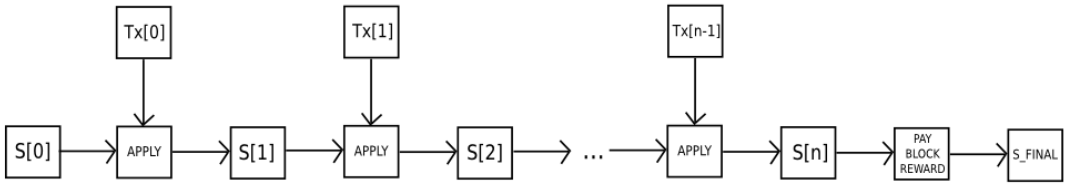


ABBILDUNG 3.1: Veränderung des Systemzustand

Der Systemzustand wird bei Ethereum durch den Zustand aller Accounts beschrieben.

3.1.7 Systemzustand Übergangsfunktion

Die Ethereum Virtual Machine geht bei der Abarbeitung einer Transaktion folgendermaßen vor:

1. Sie prüft ob die Transaktion gemäß der Konsensregeln korrekt formatiert ist, eine valide Signatur besitzt und ob der Nonce-Wert der Transaktion mit dem Nonce-Wert des Absender-Accounts übereinstimmt. Ist eine dieser Vorbedingungen nicht erfüllt, wird die Bearbeitung der Transaktion abgebrochen.

2. Es wird die maximal fällige Transaktionsgebühr durch das Multiplizieren des Gas-Werts mit dem Gas-Preis berechnet. Anschließend wird geprüft ob der Account des Senders genug Ether besitzt um die berechnete Transaktionsgebühr zu bezahlen. Ist dies der Fall wird die maximale Transaktionsgebühr vom Sender-Account abgezogen. Anderenfalls wird die Bearbeitung der Transaktion abgebrochen.

[2]Beispielsweise wenn der nächste Block eine Transaktion enthält, die bewirkt, dass die eigentlichen Transaktion ein unterschiedlicher Code-Pfad durchlaufen wird.

3. Nun wird vom Gas-Wert der Transaktion für jedes Byte der Transaktion verringert. Der Sender zahlt auf diese Weise für den Platz den die Transaktion in der Blockchain einnimmt.

4. Der Ether Wert der Transaktion wird vom Sender auf den Empfänger Account überwiesen. Falls der Empfänger Account noch nicht existiert wird er angelegt. Wird der Empfänger Account durch einen Smart Contract verwaltet, wird der Contract Code entweder vollständig ausgeführt oder aufgrund fehlenden Gas abgebrochen.

5. Falls der Sender nicht genug Ether oder verbleibendes Gas für die Überweisung besitzt, werden alle bisherigen Veränderungen des Systemzustands rückgängig gemacht. In diesem Fall wird ausschließlich die Transaktionsgebühr auf den Account des Miners überschrieben.

6. Im erfolgreichen Fall werden die Gebühren des verbleibenden Gas an den Sender zurückerstattet und die Gebühren des verbrauchten Gas an den Account des Miners überschrieben.

3.1.8 Systemzustand Beispiel

Das folgende Beispiel betrachtet, wie das Anwenden eine Transaktion einen alten Systemzustand in einen neuen Systemzustand überführt.

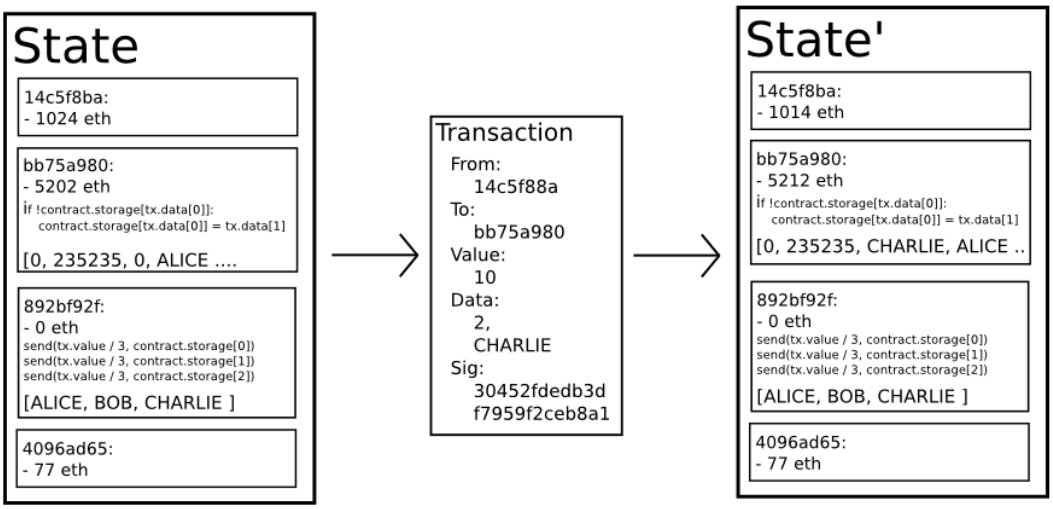


ABBILDUNG 3.2: Veränderung des Systemzustand

Die in Abbildung 3.2 betrachtete Transaktion ruft den Smart Contract des Accounts `bb75a980` auf[3]. Angenommen die Transaktion überweist einen Betrag von 10 Ether, besitzt einen Gas-Wert von 2000, einen Gas-Preis von 0,001 Ether, enthält 64 Bytes an Daten(`data[0]=2,data[1]='CHARLIE'`) und besitzt eine Gesamtgröße von 170 Bytes. Dann erfolgt die Abarbeitung der Transaktion folgendermaßen:

1. Es wird durch die Prüfung der Signatur einerseits sichergestellt, dass die Daten der Transaktion nicht manipuliert wurden und andererseits, dass die Überweisung der 10 Ether vom Sender autorisiert wurde.

[3]Transaktionsgebühren werden in Abbildung 3.2vernachlässigt.

2. Da der Account des Senders über `2000 * 0,001 = 2` Ether besitzt, wird die Bearbeitung der Transaktion fortgesetzt und die 2 Ether vom Account des Senders abgezogen.

3. Unter der Annahme, dass man 5 Gas pro Transaktionsbyte bezahlen muss, wird der Gas-Wert von 2000 auf 1150 Gas verringert. (170 Bytes * 5 = 850 Gas)

4. Der Transaktionsbetrag von 10 Ether wird vom Sender Account `14c5f88a` auf den Account `bb75a980` überwiesen.

5. Nun wird der Conract Code des Empfängeraccounts ausgeführt:
   `if !contract.storage[tx.data[0]]: contract.storage[tx.data[0]] = tx.data[1]`
   Da der Speicherplatz des Contracts an Stelle 2 noch nicht verwendet ist, wird der Wert `CARLIE` aus den Transaktionsdaten abgespeichert. Unter der Annahme, dass diese Operationen 187 Gas verbrauchen ergibt sich ein verbleibender Gas-Wert von `1150 -187 = 963`.

6. Die verbleibenden `963 * 0,001 = 0.963` Ether werden auf den Account des Senders zurücküberwiesen und die Anpassung des Systemzustands ist fertig.

3.1.9 Unterschiede zu Bitcoin

Turing-Vollständigkeit

Bitcoin besitzt intern auch eine Skriptsprache zur Ausführung von Transaktionen. Diese ist im Gegensatz zu Ethereum bewusst eingeschränkt und nicht turingmächtig. Da man keine Schleifen programmieren kann führt dies dazu, dass man Code mehrfach wiederholen muss. Dies führt dazu, dass Transaktionen die Smart Contracts ausdrücken in Bitcoin mehr Platz in der Blockchain einnehmen. Ethereum muss sich aufgrund der Turing-Vollständigkeit um das Halteproblem kümmern. Durch die Angabe eines maximalen Gas-Werts stellt Ethereum sicher, dass die Ausführung einer Transaktion spätestens nach einer gewissen Zeit abgebrochen wird.

Betrags-Blindheit

Der Betrag, der einer Bitcoin-Adresse zugeschrieben wird kann entweder nur ganz oder gar nicht ausgegeben werden. Dies ist bei Ethereum nicht der Fall. Der Nonce-Wert des Ethereum Accounts verhindert, dass eine Transaktion nicht zweimal ausgeführt werden kann.

Blockchain-Blindheit

In Bitcoin kann man bei der Ausführung der Transaktionen nicht auf Blockchain Daten wie Beispielsweise vorherige Blockhash-Werte, Zeitstempel oder None-Werte zugreifen. Eine Transaktion, die einen vorherigen Blockhash ausließt und als Zufallsquelle verwendet ist mit Bitcoin-Skript nicht möglich.

Fehlender Zustand

Die Ethereum Accounts ermöglichen es einen weitaus komplexeren Systemzustand abzubilden. Bei Bitcoin besteht der Systemzustand lediglich aus allen unausgegebenen Transaktionen.

3.2 Konzept

Der Ablauf des Spiels ist mit dem Ablauf aus dem Bitcoin Kapitels nahezu identisch. Die Unterschiede sind, dass das gesamte Spiel vom Nutzer initiiert wird und die Glücksspielanwendung ausschließlich den Spielstatus anzeigt. Dies führt dazu, dass die Spieler keinerlei Vertrauen in die Glücksspielanwendung setzen müssen. Statt aufgrund leichter Überprüfbarkeit die letzte numerische Stelle des Blockhashs für die Gewinnerauswahl zu nutzen, kann bei Ethereum der gesamten Blockhash zur Gewinnerauswahl genutzt werden. Der Spieler kann aufgrund der Konsensregeln darauf vertrauen, dass das Ethereum Netzwerk die Ausführung des Contract Codes korrekt durchführt und die modulo Funktion zur Gewinnerauswahl korrekt berechnet. Die Nutzung des gesamten Blockhashs als Zufallsquelle ermöglicht Töpfe beliebiger Größe. Der Ablauf des Spiels lässt sich durch den folgenden Zustandsautomat beschreiben.

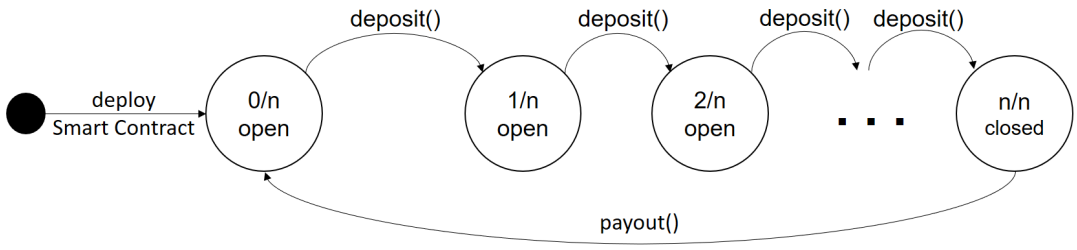


ABBILDUNG 3.3: Smart Contract Automat

Durch den Aufruf der `deploy` Funktion wir der Smart Contract in einer Transaktion an das Netzwerk gesendet und in die Blockchain aufgenommen. Innerhalb des Smart Contracts sind der Einzahlungsbetrag und die Anzahl Teilnehmer `n` fest definiert. Ab diesem Moment kann der Code des Smart Contract nicht mehr verändert werden. Im initialen Zustand ist der Topf leer, da noch kein Spieler eingezahlt hat. Nun können genau `n` Einzahlungen von Spielern durch den Aufruf der `deposit` Funktion getätigt werden. Dazu benötigen die Spieler lediglich einen Ethereum Client, ausreichend Ether[4] und die Adresse des Smart Contracts. Ab dem Zeitpunkt an dem der Smart Contract die letzte Einzahlungstransaktion empfängt, wird der Topf geschlossen und die Blocknummer für die Gewinnerauswahl festgelegt. Der Block der den Gewinner des Topfs entscheidet, muss in der Zukunft liegen und darf nicht vorher bekannt sein, da sonst Betrugsmöglichkeiten entstehen. Zum Zeitpunkt der letzten Einzahlungstransaktion ist es nicht möglich aus dem Smart Contract Code heraus auf den Blockhash des Blocks zuzugreifen in dem sich die letzte Einzahlungstransaktion befindet. Dies liegt daran, dass die Miner das Resultat der Zustandsveränderung aller Transaktionen des Blockes in den Blockheader schreiben müssen und erst anschließend den Blockhash berechnen. Die Transaktionen, die den Contract Code ausführen, können somit nicht auf den Blockhash zugreifen, da dieser zum Zeitpunkt der Codeausführung noch nicht feststeht. Es ist somit unumgänglich nach der letzten Einzahlungstransaktion eine Funktion aufzurufen, die den Gewinner auswählt, die Auszahlung startet und den Topf anschließend für ein neues Spiel wieder öffnet. Dies ist die in Abbildung 3.3 gezeigte `payout` Funktion.

[4] Einzahlungsbetrags plus Transaktionskosten und optimaler Weise genug Ether für das Bezahlen der `payout` Funktion.

3.3 Umsetzung

3.3.1 Überblick

Genau wie bei Bitcoin besteht die Möglichkeit die Glücksspielanwendung entweder mithilfe eines "Light Nodes" direkt, oder über einen "Full Node" indirekt mit dem Ethereum Netzwerk kommunizieren zu lassen. Für den Ethereum Teil dieser Masterarbeit findet die Kommunikation indirekt über einen Full Node statt. Dies ist in Abbildung 3.4 verdeutlicht. Der Full Node empfängt Transaktionen und Blöcke, validiert diese und akualisiert kontinuierlich den Zusatnd der durch die Transaktionen veränderten Ethereum Accounts. Über die RPC Schnittstelle stellt er diese Daten nach außen bereit. Die Java Bibliothek Web3J [17] erleichtert den Aufruf der RPC Schnittstelle des Full Nodes. Anders als bei Bitcoin benötigt die Glücksspielanwendung keine eigene Datenbank, da der Zustand des aktuellen Topfs im Smart Contracts und somit "in der Blockchain" gespeichert ist.

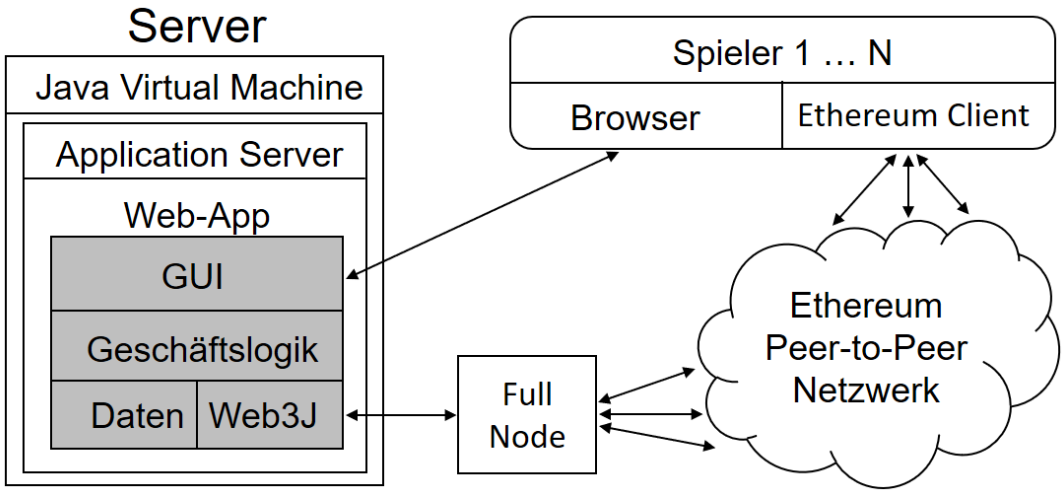


ABBILDUNG 3.4: Ethereum: Netzwerk Integration

Möchte man eine Anwendung direkt in das Ethereum Netzwerk integrieren, bietet sich in Java die Bibliothek EthereumJ [11] an.

3.3.2 Smart Contract

Die folgenden Codestücke beschreiben den TrustlessGambling Smart Contracts in der Sprache Solidity[5].

Datenmodell

Das folgenden Codestücke zeigt den Rahmen, alle Variablen und den Konstruktor des Smart Contracts.

```
1 pragma solidity ^0.4.0;
2 contract TrustlessGambling {
3     // constants
4     uint8 public constant NBR_OF_SLOTS =3;
5     uint public constant EXPECTED_POT_AMOUNT=1000;// wei
```

[5] https://solidity.readthedocs.io/en/v0.4.0/

```
6       uint8 public constant PAYOUT_BLOCK_OFFSET =1;
7       // pot values
8       uint public nbrOfParticipants;
9       address[NBR_OF_SLOTS] public depositAddresses;
10      address[NBR_OF_SLOTS] public payoutAddresses;
11      uint public closingBlockNumber;
12      uint public payoutBlockNumber;
13      bytes32 public payoutBlockHash;
14      uint public winner; // 0 -> NBR_OF_SLOTS-1
15      bool public potClosed;
16      uint public nbrOfMissedPayouts;
17      // constructor
18      function TrustlessGambling() public {
19          nbrOfParticipants = 0;
20          potClosed = false;
21          nbrOfMissedPayouts = 0;
22      }
23  }
```

Zeile 1 definiert in welcher Version der Solidity Sprache der Smart Contract geschrieben ist. Dies muss vom Compiler berücksichtigt werden. Zeile 2 legt den Namen des Smart Contracts fest. Über die beiden Konstanten in Zeile 4 und 5 kann man die Anzahl Spieler, und den von jedem Spieler erwarteten Einzahlungsbetrag festlegen. Die in Zeile 6 festgelegte Konstante legt fest, welcher Block ab der letzten Einzahlungstransaktion den Gewinner festlegt. Diese Werte können nach der Bereitstellung des Smart Contracts nicht mehr verändert werden. Die Variablen von Zeile 8 bis Zeile 16 werden vom Smart Contract manipuliert und speichern den Zustand des aktuellen Topfes. Zeile 8 speichert wie viele Teilnehmer bereits eingezahlt haben. Zeile 9 und 10 speichern die Ein- und Auszahlungsadressen der aktuellen Teilnehmer. Die Zeilen 11 bis 14 speichern alle für die Gewinnerauswahl benötigten Werte. Zeile 15 definiert über den Wahrheitswert `potClosed`, ob der Topf offen ist und Einzahlungen stattfinden können, oder ob der Topf geschlossen ist. Der Nutzen des Wertes aus Zeile 16 wird in Abschnitt 3.3.2 erklärt. Zeile 18 bis 22 beinhalten den einmalig bei der Bereitstellung des Smart Contracts aufgerufenen Konstruktor. Alle Variablen des Smart Contracts sind zur Schaffung maximaler Transparenz mit dem Schlüsselwort `public` markiert. Dies erlaubt es den Nutzern, alle Werte des Smart Contract abzurufen.

Einzahlungen

Einzahlungen finden über die beiden `deposit` Methoden statt. Diese sind mit dem Schlüsselwort `payable` markiert. Dies bedeutet, dass Transaktionen einen Ether-Betrag beim aufruf dieser Methoden angeben können.

```
1   function deposit() payable public {
2       deposit(msg.sender);
3   }
4   function deposit(address _payout) payable public {
5       assert(!potClosed);
6       assert(msg.value == EXPECTED_POT_AMOUNT);
7       depositAddresses[nbrOfParticipants] = msg.sender;
8       payoutAddresses[nbrOfParticipants] = _payout;
9       nbrOfParticipants++;
10      if (nbrOfParticipants == NBR_OF_SLOTS){
11          closingBlockNumber = block.number;
```

```
12          payoutBlockNumber = closingBlockNumber +
                PAYOUT_BLOCK_OFFSET;
13          potClosed = true;
14      }
15  }
```

Nutzt der Spieler die Methode aus Zeile 1, wir als Auszahlungsadresse einfach die Adresse der Transaktion verwendet. Nutzt der Spieler die Methode aus Zeile 4, hat er die Möglichkeit eine beliebige Auszahlungsadresse anzugeben. Bei der Einzahlung wird zunächst in Zeile 5 geprüft, ob der der Topf offen ist. Ist dies der Fall, prüft Zeile 6, dass der Eingezahlte Betrag mit dem fest definierten Wert übereinstimmt. Anschließen werden die Ein- und Auszahlungsadresse abgespeichert und die aktuelle Anzahl Teilnehmer um eins erhöht. Zeile 10 prüft, ob es sich um die letzte Einzahlungstransaktion handelt und schließt den Topf gegebenenfalls. Bevor der Topf geschlossen wird, wird allerdings noch in Zeile 11 die aktuelle Blocknummer abgespeichert und anschließend die Blocknummer für die Gewinnerauswahl berechnet.

Auszahlungen

Auszahlungen finden durch den Aufruf der `payout` Methoden statt. Diese ist nicht mit dem Schlüsselwort `payable` markiert und erwartet keinen Ether-Betrag beim Aufruf.

```
1  function payout() public{
2      assert(potClosed);
3      assert(block.number>payoutBlockNumber);
4      payoutBlockHash = block.blockhash(payoutBlockNumber);
5      if(payoutBlockHash == 0){
6          nbrOfMissedPayouts++;
7      }else{
8          winner = uint(payoutBlockHash) % NBR_OF_SLOTS;
9          address winnerAddress = payoutAddresses[winner];
10         uint amount= EXPECTED_POT_AMOUNT*NBR_OF_SLOTS;
11         amount +=
               EXPECTED_POT_AMOUNT*NBR_OF_SLOTS*nbrOfMissedPayouts;
12         winnerAddress.transfer(amount); // send pot amount to
               winner
13         nbrOfMissedPayouts = 0;
14     }
15     potClosed = false;
16     nbrOfParticipants=0;
17 }
```

Die Methode kann nur aufgerufen werden, falls der Topf geschlossen ist und die aktuelle Blocknummer bereits größer als die Blocknummer des Blocks für die Gewinnerauswahl ist. Sind diese Bedingungen erfüllt, hängt der weitere Verlauf der Abarbeitung der `payout` Methode vom Zeitpunkt des Methodenaufrufs ab. Smart Contracts können laut einer Konvention[6] bei ihrer Ausführung nur auf die Werte der 256 letzten Blockheader zugreifen[7]. Ist der in Zeile 4 angefragte `payoutBlockHash`

[6] Dies ist Effizienzgründen geschuldet. Blockheader sind in Ethereum mindestens 500 Byte groß, mit einer Blockzeit von 12 Sekunden wächst die Blockheaderkette somit täglich um 3,6 Megabyte. Der Zuwachs beträgt jährlich somit über 1 Gigabyte an Daten. Ein Ethereum Client der nicht die gesamte Blockheaderkette speichert, könnte somit nicht die Ausführung von Smart Contracts validieren, da ihm dazu die Daten aus der Vergangenheit fehlen.

[7] `http://solidity.readthedocs.io/en/develop/units-and-global-variables.html`

älter als 256 Blocks, gibt `block.blockhash(<number>)` den Wert 0 zurück und Fall 1 tritt ein.

1. Fall: Der Aufruf der `payout` Methode findet zu spät statt. Es findet keine Auszahlung statt, da der Smart Contract nicht auf den entscheidenden Blockhash zugreifen kann. Der Smart Contract erhöht die `nbrOfMissedPayouts` Variable um eins. Dies führt dazu, dass Betrag des Topfs in den nächsten Topf verschoben wird.

2. Fall: Der Aufruf der `payout` Methode findet rechtzeitig statt. Der Smart Contract berechnet in Zeile 8 den Gewinner indem er den Blockhash in einen Integer konvertiert und diese sehr hohe Zahl modulo der Anzahl Teilnehmer rechnet. Anschließend wird der korrekte Auszahlungsbetrag berechnet und in Zeile 12 an die Auszahlungsadresse des Gewinners versandt.

Zum Schluss wird der Topf wieder geöffnet und die Anzahl der teilnehmenden Spieler auf 0 gesetzt.

3.3.3 Smart Contract Bereitstellung

Nachdem man den Smart Contract programmiert hat, muss man ihn zu Bytecode kompilieren und anschließen in einer Transaktion an das Ethereum Netzwerk senden. Der in Solidity geschriebene Smart Contract Code kann mithilfe eines Online Compilers [8] kompiliert werden. Das Kompilieren erzeugt die Dateien TrustlessGambling.bin und TrustlessGambling.abi. Diese enthalten den Bytecode und das Smart Contract **A**pplication **B**inary **I**nterface. Um in der Programmiersprache Java mit dem Smart Contract interagieren zu können stellt Web3J sogenannte Comandline Tools[9] zur Verfügung. Durch den Aufruf des folgenden Befehl wird die Klasse `TrustlessGambling.java` erzeugt.

```
1 web3j solidity generate TrustlessGambling.bin
    TrustlessGambling.abi -o /path/to/src/main/java -p
    com.ossel.gamble.ethereum.generated
```

Da zur Bereitstellung des Smart Contracts auch die entsprechenden Transaktionsgebühren bezahlen muss, wird eine Wallet benötigt. Web3J hilft auch bei diesem Schritt. Der folgende Befehl leitet die Generierung einer Wallet ein.

```
1 web3j wallet create
```

Über die Kommandozeile muss der Benutzer den gewünschten Wallet-Dateinamen und ein Passwort angeben. In diesem Beispiel wird der Dateiname `ethereum.json` und das Passwort `changeit` verwendet. Anschließen wird die Wallet generiert und die Ethereum Account Adresse ausgegeben. Diese ist in diesem Beispiel die Adresse `0x2201f3919589b519135ce977cc0906c9481069b2`. Bevor der Smart Contract durch eine Transaktion veröffentlicht wird, müssen wir zunächst eins der Ethereum Netzwerke wählen und in den Besitz der auf diesem verwendeten Ether-Währung. Bei Ethereum gibt es die folgende Netzwerke zur Auswahl:

1. Mainnet: Genau wie bei Bitcoin handelt es sich bei diesem Netzwerk um das Produktionsnetzwerk.

[8] https://ethereum.github.io/browser-solidity
[9] https://docs.web3j.io/command_line.html

2. Ropsten Testnetz: Hierbei handelt es sich um ein Testnetz, dass Proof-of-Work als Konsensalgorithmus verwendet und ist dem Ethereum Mainnet am ähnlichsten. Der auf diesem Netzwerk ausgetauschten Ether-Währung wird allerdings kein finanzieller Wert zugemessen.

3. Rinkeby Testnetz: Hierbei handelt es sich um ein Testnetz, das nicht Proof-of-Work, sondern das Clique-Proof-of-Authority Protokoll als Konsens-Algorithmus verwendet. Im Gegensatz zu Proof-of-Work wird ein Konsens durch das Signieren von Blocken durch bekannte Teilnehmer gewährleistet. Das Ethereum Improvement Proposal [7] beschreibt den verwendeten Vorgang detailliert.

Das Problem bei der Verwendung des Proof-of-Work Algorithmus auf einem Testnetz ist, dass Miner für ihren Stromverbrauch nur in der wertlosen Testnetz-Währung bezahlt werden. Daraus resultiert, dass solch ein Testnetz nur eine sehr geringe Hashrate aufweist. Ein Angreifer der eine signifikante Hashrate besitzt kann beispielsweise nur noch Blöcke erzeugen, die keine Transaktionen beinhalten und dadurch das Testnetz für eine gewisse Zeit lahmlegen. Das Rinkeby Testnetz ist gegen solche Angriffe resistenter und daher im allgemeinen das stabilere Testnetzwerk. Um Ether auf dem Rinkeby Testnetz zu erhalten, verwendet man eine sogenannte Faucete[10], die in regelmäßigen Abständen Ether an eine beliebige Adresse versendet. Nachdem die Wallet über Ether verfügt kann man mit Hilfe von Web3J den geschriebenen Smart Contract bereitstellen. Abbildung 3.5 listet die dazu verwendeten Klassen und die generierte `TrustlessGambling` Klasse auf.

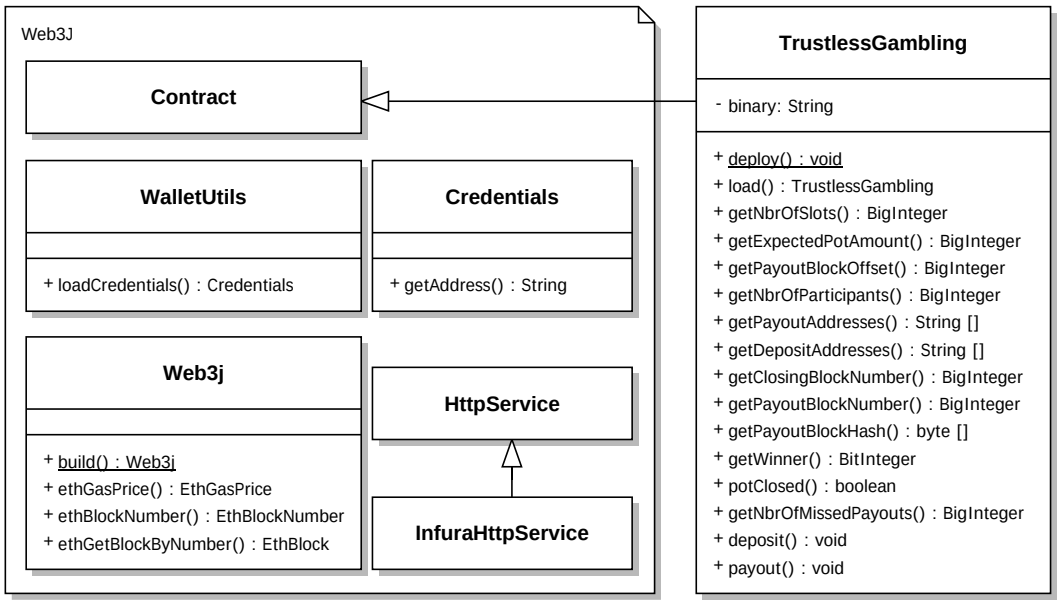


ABBILDUNG 3.5: Klassendiagramm Web3J

Der folgende Java Code sorgt dafür, dass der von Web3J angesprochene Full Node den Smart Conrtact in einer Transaktion an das Rinkeby Testnetz sendet.

```
1 public void createContract() throws Exception {
2   String WALLET_FILENAME = "ethereum.json";
3   String WALLET_PASSWORD = "changeit";
4   long GAS_LIMIT = 1000000;
```

[10] https://faucet.rinkeby.io/

```
5    Web3j web3j = Web3j.build(new
         InfuraHttpService("https://rinkeby.infura.io/" +
         UserConfiguration.API_KEY));
6    BigInteger currentGasPrice =
         web3j.ethGasPrice().send().getGasPrice();
7    ClassLoader classLoader = getClass().getClassLoader();
8    File walletFile = new
         File(classLoader.getResource(WALLET_FILENAME).getFile());
9    Credentials credentials =
         WalletUtils.loadCredentials(WALLET_PASSWORD,
         walletFile.getAbsolutePath());
10   System.out.println("Account address = " +
         credentials.getAddress());
11   TrustlessGambling contract = TrustlessGambling.deploy(web3j,
         credentials, currentGasPrice,
         BigInteger.valueOf(GAS_LIMIT)).send();
12   String status =
         contract.getTransactionReceipt().get().getStatus();
13   if ("0x1".equals(status)) {
14     String address = contract.getContractAddress();
15     System.out.println("Contract address = " + address);
16     System.out.println("TXN hash = " +
           contract.getTransactionReceipt().get().getTransactionHash());
17     System.out.println("Gas used = " +
           contract.getTransactionReceipt().get().getGasUsed());
18   } else {
19     System.out.println("Smart contract could not be deployed.");
20   }
21 }
```

In Zeile 5 wird der Web3J Service erzeugt. Dieser kümmert sich um die Kommunikation mit dem Full Node. Die übergebene URL legt die Adresse des Full Nodes fest. Infura[11] ist dabei ein Service der sich auf das Hosting von Ethereum Full Nodes spezialisiert hat. Über einen API Schlüssel kann man sich zu seinem Full Node verbinden. Statt des von Infura betriebenen Nodes kann man auch einen eigens gemanagten Full Node verwenden, um die volle Kontrolle zu behalten. In diesem Fall verwendet man statt des `InfuraHttpService` direkt die Oberklasse `HttpService`. Zeile 6 fragt den Full Node nach dem aktuell zu bezahlenden Gaspreis. In Zeile 8 wird das durch die Web3J Comandline Tools erzeugte Wallet geladen. Mithilfe dieses wird unter Zuhilfenahme des Passworts die im nächsten Schritt verwendeten `Credentials` geladen. In Zeile 11 wird der Smart Contract durch den Aufruf der statischen `TrustlessGambling.deploy` Methode in der Transaktion an das Netzwerk gesendet. Dabei wird ein Gaslimit von einer Million WEI festgelegt. Die Ausführung des oben gezeigten Java Codes führt zu der folgenden Ausgabe:

```
1 Account address = 0x2201f3919589b519135ce977cc0906c9481069b2
2 Contract address = 0x25c3136145fbd7f3b9217e58e2fabe3eb1928705
3 TXN hash =
    0x06dce3c460b4caa595c5cc0f81ac78e7c70eeb1e89d3e0e6a017ea88e60dbce1
4 Gas used = 825846
```

Das Gaslimit von einer Million WEI hat ausgereicht und der Smart Contract befindet sich nun in der Blockchain des Ethereum Rinkeby Testnetzes. In einem Blockchain Explorer kann man die Details der vom Full Node erstellten Transaktion[12] und

[11] `https://infura.io/`

[12] `https://rinkeby.etherscan.io/tx/0x06dce3c460b4caa595c5cc0f81ac78e7c70eeb1e89d3e0e6a017ea88e60dbce1`

den kompilierten Contract Code[13] anschauen.

3.3.4 Geschäftslogik Glücksspielanwendung

Die Glücksspielanwendung zeigt lediglich den aktuellen Zustand des Smart Contracts an. Die gesamte Geschäftslogik des Smart Contracts wird vom Etherem Netzwerk ausgeführt. Sollte die Glücksspielanwendung aufgrund technischer Fehler ausfallen, hat dies keinerlei Auswirkung auf das eigentliche Spiel. Die Geschäftslogik der Glücksspielanwendung fragt lediglich in regelmäßigen Abständen beim Full Node an, ob eine Änderung des Smart Contract Zustands stattgefunden hat. Abbildung 3.6 liefert einen ersten Überblick über die dazu verwendeten Klassen.

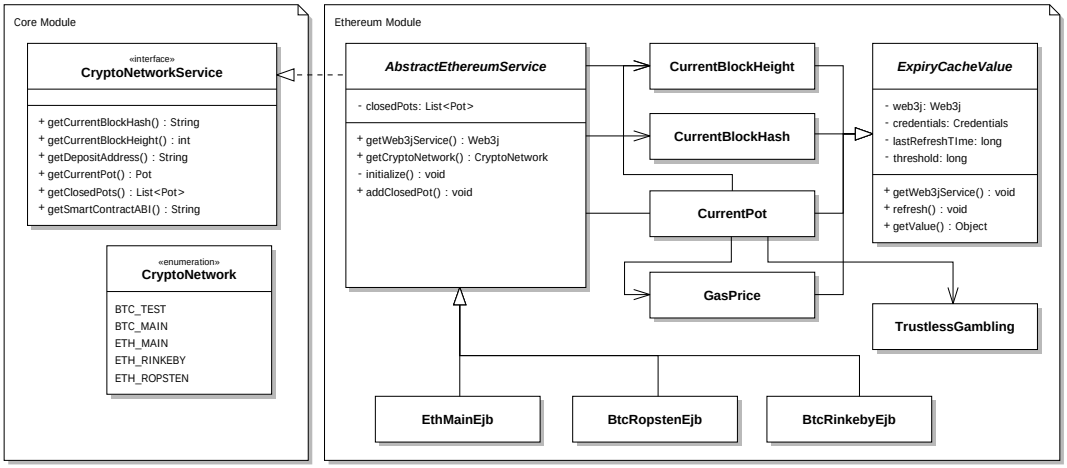


ABBILDUNG 3.6: Klassendiagramm Ethereum

Core Module

Das `CryptoNetworkService` Interface wurde um die Methode `getSmartContractABI` erweitert. Bei dem Smart Contract **A**pplication **B**inary **I**nterface handelt es sich um die statische zur Compile-Zeit bestimmte Schnittstellenbeschreibung, die festlegt wie man mit dem Smart Contract interagieren kann. Das Smart Contract ABI wird üblicherweise im JSON Format angegeben.

Zu dem `CryptoNetwork` Enum sind nun die zusätzlichen Werte `ETH MAIN, ETH RINKEBY, ETH ROPSTEN` hinzugekommen. Über diese Werte kann man steuern, mit welchem Ethereum Netzwerk die Anwendung kommunizieren soll. Die Klassen `Pot` und `Participant` haben sich nicht verändert.

Ethereum Module: AbstractEthereumService

Die abstrakte Klasse `AbstractEthereumService` implementiert die `CryptoNetworkService` Schnittstelle. Die Klassen `EthMainEjb, EthRopstenEjb und EthRinkebyEjb` legen lediglich über den verwendeten Web3J Service fest, welches Netzwerk die Anwendung ansprechen soll. Die abstrakte Klasse `AbstractEthereumService` implementiert die vom `CryptoNetworkService` Interface geforderten Methoden. Die Methoden `getDepositAdress` und `getSmartContractABI` geben lediglich die statische Smart

[13] https://rinkeby.etherscan.io/address/0x25c3136145fbd7f3b9217e58e2fabe3eb1928705#code

Contract Adresse und den Smart Contract ABI JSON String zurück. Die Methoden `getCurrentBlockHash, getCurrentBlockHeight` und `getCurrentPot` geben gecachte Werte des jeweiligen `ExpieryCacheValue` durch den Aufruf der `getValue` Methode an die Webanwendung zurück. Die Klassen `CurrentBlockHeight, CurrentBlockHash, GasPrice und CurrentPot` erweitern die abstrakte `ExpieryCacheValue` Klasse. Diese sorgt dafür, dass die Werte erst nach dem Ablauf einer gewissen konfigurierbaren Zeit (`threshold`) automatisch neu beim Full Node angefragt werden. Dies verhindert, dass der Full Node durch zu viele Anfragen überlastet wird. Alle Cache Werte werden in der `initialize` Methode der `AbstractEthereumService` Klasse initialisiert.

```
1  @PostConstruct
2  private void initialize() {
3    log.info("#### start " + getClass().getSimpleName() + " network
         service ####");
4    blockHightCache = new CurrentBlockHeight(getWeb3jService(),
         getCredentials());
5    log.info("blockHight=" + blockHightCache.getValue().intValue());
6    blockHashCache = new CurrentBlockHash(getWeb3jService(),
         getCredentials(), blockHightCache);
7    log.info("blockHash=" + blockHashCache.getValue());
8    gasPriceCache = new GasPrice(getWeb3jService(),
         getCredentials());
9    log.info("gasPrice=" + gasPriceCache.getValue().intValue());
10   currentPotCache = new CurrentPot(getWeb3jService(),
         getCredentials(), gasPriceCache, blockHightCache,
11                 UserConfiguration.CONTRACT_ADDRESS);
12   log.info("currentPot=" + currentPotCache.getValue().toString());
13 }
```

Ethereum Module: CurrentBlockHash

Der folgende Code zeigt beispielhaft die die Implementierung des `CurrentBlockHash ExpieryCacheValue`.

```
1  public class CurrentBlockHash extends ExpiryCacheValue {
2
3    CurrentBlockHeight blockHeight;
4
5    public CurrentBlockHash(Web3j web3j, Credentials credentials,
         CurrentBlockHeight blockHeight) {
6      super(web3j, credentials, 5 * SECOND);
7      this.blockHeight = blockHeight;
8    }
9
10   /**
11    * automatically refresh if cache value expired
12    */
13   @Override
14   protected void refresh() {
15     String hash = "error";
16     DefaultBlockParameterNumber number = new
           DefaultBlockParameterNumber(blockHeight.getValue());;
17     try {
18       EthBlock ethBlock =
             getWeb3jService().ethGetBlockByNumber(number,
             false).send();
```

```
19      hash = ethBlock.getBlock().getHash();
20      } catch (IOException e) {
21        e.printStackTrace();
22      }
23      setValue(blockHash);
24    }
25  }
```

In Zeile 6 wird konfiguriert, dass der aktuelle Blockhash maximal alle 5 Sekunden vom Full Node abgefragt wird. In Zeile 18 wird durch den Aufruf der `ethGetBlockByNumber` Methode der Block der aktuellen Blocknummer angefragt. Über den Wahrheitswertparameter kann man entweder die gesamten Blockdaten oder nur die Header-Informationen beim Full Node anfragen.

Ethereum Module: CurrentBlockHeight

Die Implementierung des `CurrentBlockHeight ExpieryCacheValue` greift auf die folgenden Zeilen Code zurück. Es findet maximal alle 5 Sekunden eine Anfrage an den Full Node statt.

```
1 EthBlockNumber ethBlockNumber =
      getWeb3jService().ethBlockNumber().send();
2 BigInteger currentBlockNumber = ethBlockNumber.getBlockNumber();
```

Ethereum Module: GasPrice

Die Implementierung des `GasPrice ExpieryCacheValue` greift auf die folgenden Zeilen Code zurück. Es findet maximal alle 60 Sekunden eine Anfrage an den Full Node statt.

```
1 EthGasPrice ethGasPrice = getWeb3jService().ethGasPrice().send();
2 BigInteger gasPrice = ethGasPrice.getGasPrice();
```

Ethereum Module: CurrentPot

Der `CurrentPot ExpieryCacheValue` verwendet die Klasse `TrustlessGambling` um den Zustand des Smart Contracts zu erfassen und in die Klasse `Pot` abzubilden. Im Konstruktor des `CurrentPot ExpieryCacheValue` wird die Methode `createEmptyPot` aufgerufen.

```
1 private Pot createEmptyPot() throws Exception{
2     TrustlessGambling contract =
          TrustlessGambling.load(contractAddress, getWeb3jService(),
3               getCredentials(), gasPrice.getValue(),
                    BigInteger.valueOf(5300000));
4     int nbrOfSlots = contract.NBR_OF_SLOTS().send().intValue();
5     long amount =
          contract.EXPECTED_POT_AMOUNT().send().longValue();
6     return new Pot(nbrOfSlots, amount);
7 }
```

Diese Methode fragt den Full Node, wie viele Spieler und welcher Einzahlungsbetrag vom Smart Contract erwartet wird und erzeugt anschließend einen neuen leeren Topf. Der Zustand des Topfs wird durch den folgenden Code jedes mal aktualisiert, wenn die `refresh` Methode des `ExpieryCacheValue` aufgerufen wird. Dies findet alle 10 Sekunden statt.

```
1  TrustlessGambling contract =
       TrustlessGambling.load(contractAddress, getWeb3jService(),
2          getCredentials(),
               gasPrice.getValue(),BigInteger.valueOf(5300000));
3  Pot pot = (Pot) this.value;
4  int potParticipants = pot.getNbrOfParticipants();
5  int actualParaticipants =
       contract.nbrOfParticipants().send().intValue();
6  if (actualParaticipants >= potParticipants) {
7     for (int i = potParticipants; i < actualParaticipants; i++) {
8          String depositAddress = getDepositAddress(contract, i);
9          String payoutAddress = getPayoutAddress(contract, i);
10         pot.addParticipant(new Participant(depositAddress,
               payoutAddress));
11    }
12 } else {
13    // pot has been reopened
14    int winner = contract.winner().send().intValue();
15    byte[] hashBytes = contract.payoutBlockHash().send();
16    String payoutBlockhash =
          javax.xml.bind.DatatypeConverter.printHexBinary(hashBytes);
17    if (new BigInteger(payoutBlockhash).intValue() == 0) {
18         pot.setState(
19                 "Pot closed. Payout() too late. The amount has
                       been added to the next pot.");
20    } else {
21         Block block = new Block("0x" +
               payoutBlockhash.toLowerCase(), winner);
22         pot.setPayoutBlock(block);
23         pot.setWinner(winner);
24         pot.setState("Pot closed. Winner is " +
               pot.getWinner().getPayoutAddress());
25    }
26    this.ethereumService.addClosedPot(pot);
27    this.value = createEmptyPot();
28 }
29
30 boolean potClosed = contract.potClosed().send();
31 if (potClosed) {
32    int closingBlockNumber =
          contract.closingBlockNumber().send().intValue();
33    int payoutBlockNumber =
          contract.payoutBlockNumber().send().intValue();
34    pot.setClosingBlockHeight(closingBlockNumber);
35    pot.setPayoutBlockHeight(payoutBlockNumber);
36    int currentBlockNumber =
          currentBlockHeight.getValue().intValue();
37    if (pot.getPayoutBlockHeight() > currentBlockNumber) {
38         pot.setState("Pot closed. Waiting for payout block.");
39    } else {
40         int diff = currentBlockNumber - pot.getPayoutBlockHeight();
41         int blocksLeft = (256 - diff); // solidity restriction
42         if (blocksLeft > 0) {
43              pot.setState("Pot closed. Call payout() during the
                    next " + blocksLeft
44                       + " blocks. Otherwise the whole amount will be
                             added to the next pot.");
```

```
45          } else {
46              pot.setState(
47                      "Pot closed. Payout() too late. The amount has
                           been added to the next pot. Call payout()
                           to open a new pot.");
48          }
49      }
50  }
```

Der Code unterscheidet zwischen der Anzahl der Teilnehmer, die die Glücksspielanwendung lokal zwischenspeichert(Zeile 4) und der Anzahl Teilnehmer des Datenfeldes des Smart Contracts (Zeile 5). Wenn neue Teilnehmer durch den Aufruf der `deposit` Methode in den Smart Contract einzahlen, wird der Topf durch die Zeilen 7 bis 11 aktualisiert. Die Ein- und Auszahlungsadressen der neuen Teilnehmer werden dazu aus den Smart Conract Daten geladen. Durch die letzte Einzahlung wechselt der Smart Contract in den Status `closed`. Ab diesem Moment wird der Topf durch die Abarbeitung des Codes ab Zeile 30 aktualisiert. Zunächst werden die finalen `closingBlockNumber` und `payoutBlockNumber` Werte aus den Smart Contract Daten geladen und die `currentBlockNumber` durch den Full Node bestimmt. Anschließend wird zwischen 3 Fällen unterschieden:

1. Zeile 38: Der zur Gewinnerauswahl benötigte Block wurde noch nicht gefunden. Dies bedeutet, dass ein Aufruf der `payout` Methode des Smart Contracts noch nicht möglich ist.

2. Zeile 43: Der zur Gewinnerauswahl benötigte Block wurde gefunden und die `payout` Methode kann aufgerufen werden. In diesem Fall wird dem Benutzer angezeigt, wie viel Zeit er noch für den Aufruf der `payout` Methode hat.

3. Zeile 46: Der zur Gewinnerauswahl benötigte Block wurde zwar gefunden, es sind allerdings bereits mehr als 256 Blöcke zwischen der letzten Einzahlung und dem aktuellen Zeitpunkt vergangen. Der Smart Contract wird bei dem Aufruf der `payout` Methode nicht auf den Blockhash für die Gewinnerauswahl zugreifen können und den Gewinn zum nächsten Topf hinzufügen.

Durch den Aufruf der `payout` Methode wird das Datenfeld, das die Anzahl der aktuellen Teilnehmer des Smart Contracts speichert, auf den Wert Null gesetzt. Dies hat zur Folge, dass beim erneuten Aufruf der `refresh` Methode des `CurrenPot ExpieryCacheValue` der Code von Zeile 13 bis 27 ausgeführt wird. Dieser Code lädt den Gewinner und den `payoutBlockHash` aus den Daten des Smart Contracts. Hat der `payoutBlockHash` den Wert Null, wurde die `payout` Methode zu spät aufgerufen. Nur in diesem Fall gibt es keinen Gewinner. Anschließend wird der aktuelle Topf zur Liste der abgearbeiteten Töpfe hinzugefügt und ein neuer Topf durch den Aufruf der `createEmptyPot` Methode erzeugt.

3.3.5 Grafische Benutzeroberfläche

Das folgende Beispiel betrachtet einen frisch auf dem Ethereum Rinkeby Testnetzwerk bereitgestellten Smart Contract.

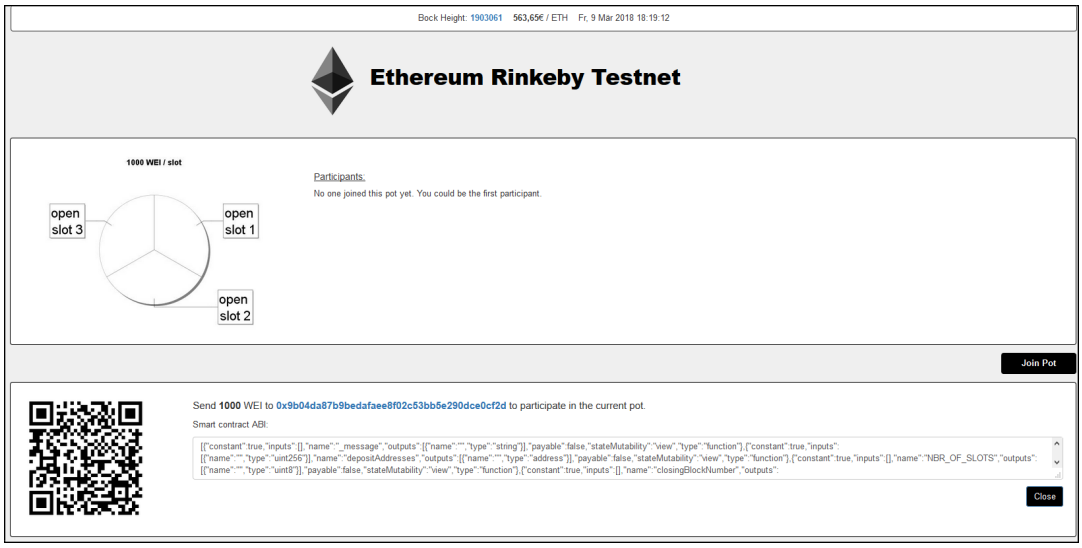


ABBILDUNG 3.7: Leerer Topf

Abbildung 3.7 zeigt einen Topf mit 3 freien Plätzen. Um dem Spiel beizutreten, muss der Spieler den Betrag von 1000 WEI (kleinste Ether Einheit) an den Smart Contract senden. Genau wie bei Bitcoin wird dem Nutzer ein QR-Code angezeigt, der die Übermittlung der Daten in einen Smartphone Client erleichtert. Das **E**thereum **I**mprovement **P**roposal Nummer 681[**eip21**] legt die Kodierung der Daten fest. Folgende Daten sind in dem QR Code enthalten: "ethereum:0x9b04da87b9bedafaee8f02c53bb5e290dce0cf2d/deposit?value=1000". In diesem Beispiel verwenden wir für die Interaktion mit dem Netzwerk keinen Smartphone Client sondern die Webanwendung namens "My Ether Wallet"[14]. Diese benötigt für die Interaktion mit dem Smart Contract sowohl die Contract Adresse als auch das **A**pplication **B**inary **I**nterface.

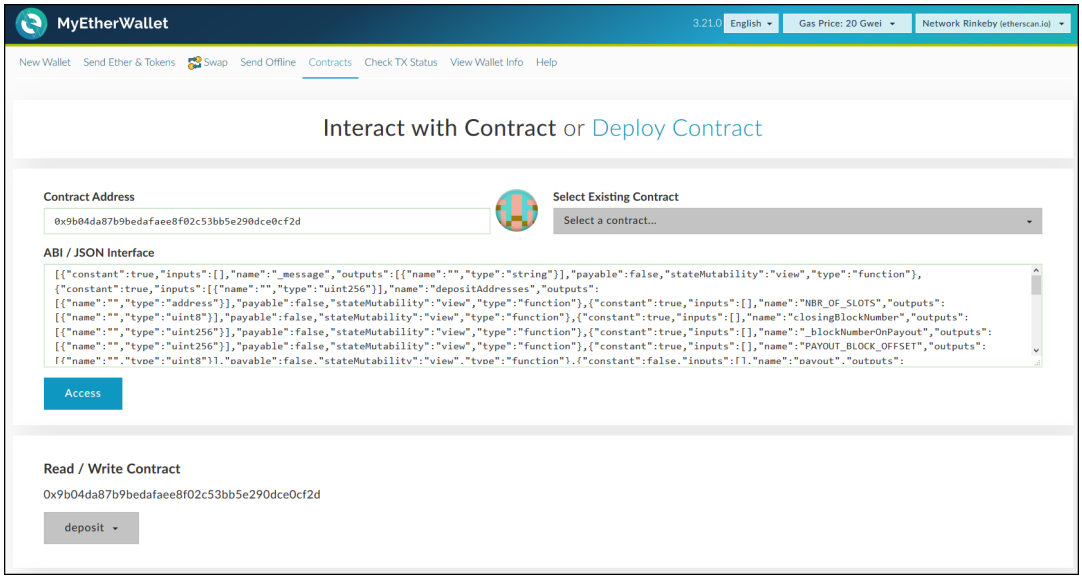


ABBILDUNG 3.8: My Ether Wallet

[14] https://www.myetherwallet.com/#contracts

Nachdem der Nutzer diese wie in Abbildung 3.8 eingegeben hat kann er über eine Dropdown-Liste die gewünschte Funktion des Smart Contracts aufrufen.

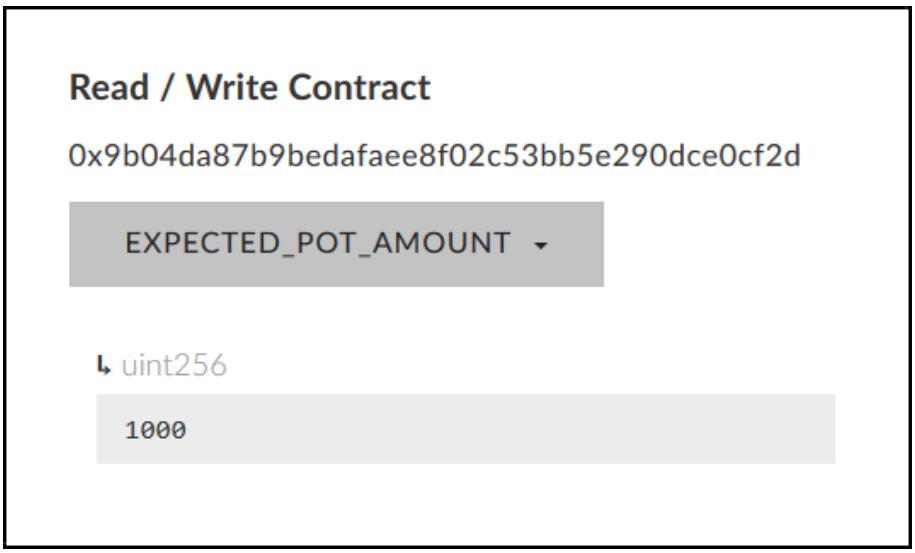


ABBILDUNG 3.9: Aufruf der `EXPECTED POT AMOUNT` Funktion

Abbildung 3.9 zeigt den Aufruf der Funktion auf, die zurückgibt, welchen Geldbetrag der Smart Contract vom Spieler erwartet. Da es sich lediglich um einen lesenden Zugriff handelt, wird keine Transaktion ans Netzwerk gesendet, beziehungsweise in die Blockchain geschrieben. Es fallen somit keine Transaktionskosten an.

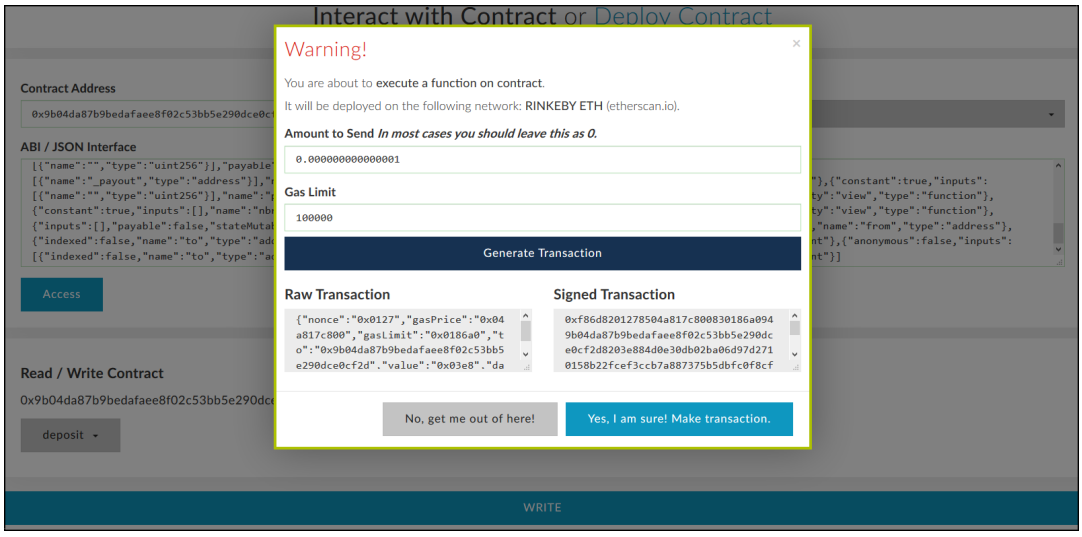


ABBILDUNG 3.10: Aufruf der `deposit` Funktion

Da der Nutzer nun nachgeprüft hat, dass der Smart Contract wirklich Zahlungen von 1000 WEI erwartet, kann er die `depsoit` Funktion mit diesem Betrag aufrufen. Die Wallet Webseite erwartet den Betrag in der Einheit Ether. Die geforderten 1000 WEI entsprechen 0.000000000000001 Ether. Die Umrechnung kann der Spieler mittels eines Online Konverters[15] durchführen. Nun muss die erstellte Transaktion nur

[15] `https://etherconverter.online/`

noch signiert werden. Der Nutzer kann der Webseite dazu seinen privaten Schlüssel mitteilen oder die Signierung eigenständig durch ein sogenanntes Hardware Wallet durchführen. Die Herausgabe seines privaten Schlüssels an eine Webseite ist aus sicherheitstechnischer Sicht keine gute Praktik. Sollte der Webseitenbetreiber böse Absichten haben oder die Webseite gehackt werden, führt dies zum Verlust des durch den Schlüssel kontrollierten Geldes. Eine sichere Variante ist die Verwendung eines Hardware Wallets. Dieses speichert alle privaten Schlüssel und führt die Signatur eigenständig durch. Der verwendete private Schlüssel verlässt somit niemals das Gerät.

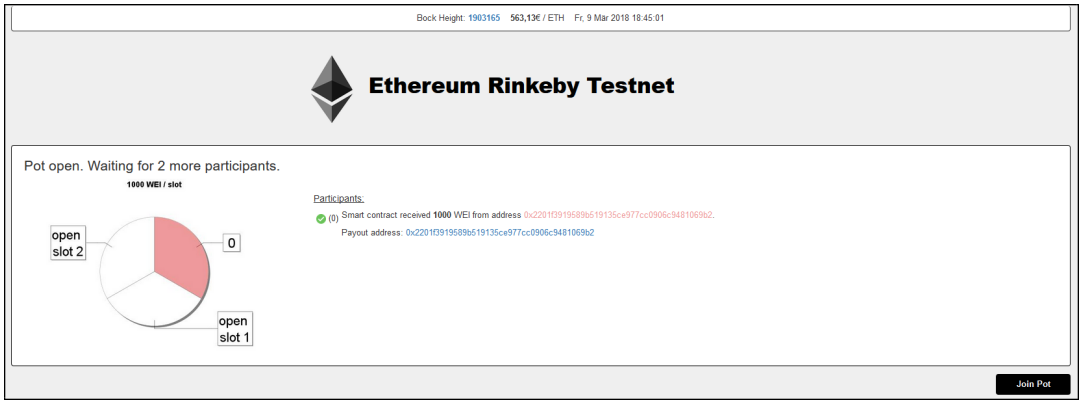


ABBILDUNG 3.11: Eingang der ersten Zahlung

Abbildung 3.11 visualisiert den Zustand des Smart Contracts nachdem die erste Einzahlungstransaktion in die Blockchain aufgenommen wurde.

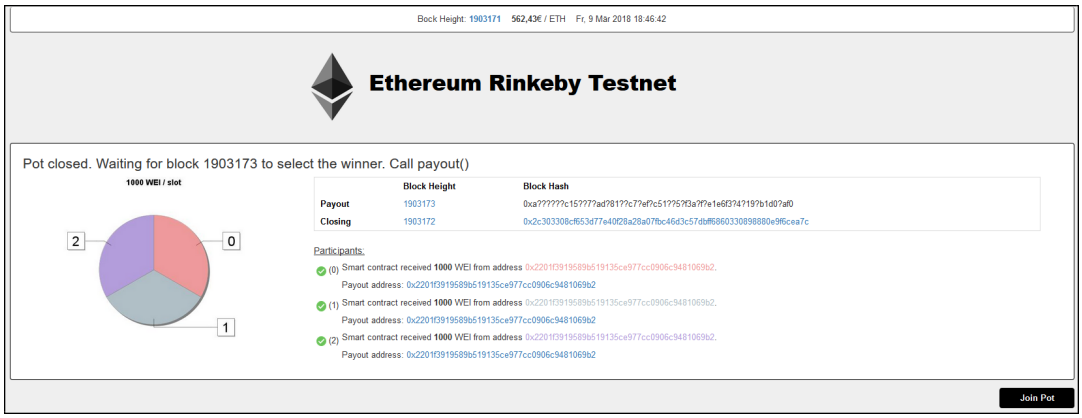


ABBILDUNG 3.12: Topf geschlossen

Abbildung 3.12 visualisiert den Zustand des Smart Contracts nachdem die letzte Einzahlungstransaktion in die Blockchain aufgenommen wurde. Der Smart Contract hat den Topf geschlossen und wartet nun, dass einer der Spieler die `payout` Funktion aufruft.

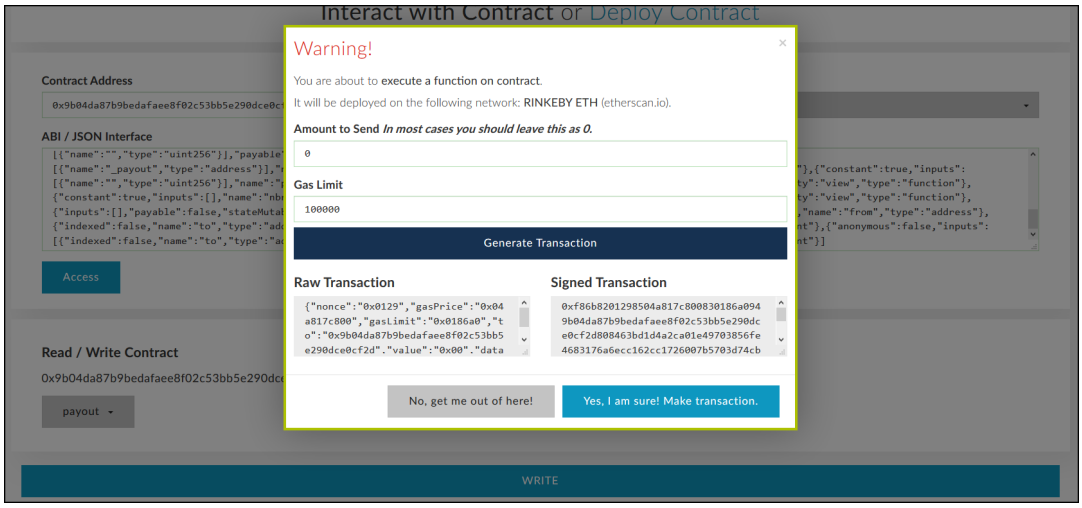


ABBILDUNG 3.13: Aufruf der `payout` Funktion

In Abbildung 3.13 ist gezeigt wie ein Spieler die `payout` Funktion aufruft. Durch den Aufruf dieser Funktion wird der Gewinner ausgewählt, die Auszahlung getätigt und der Topf wieder geöffnet.

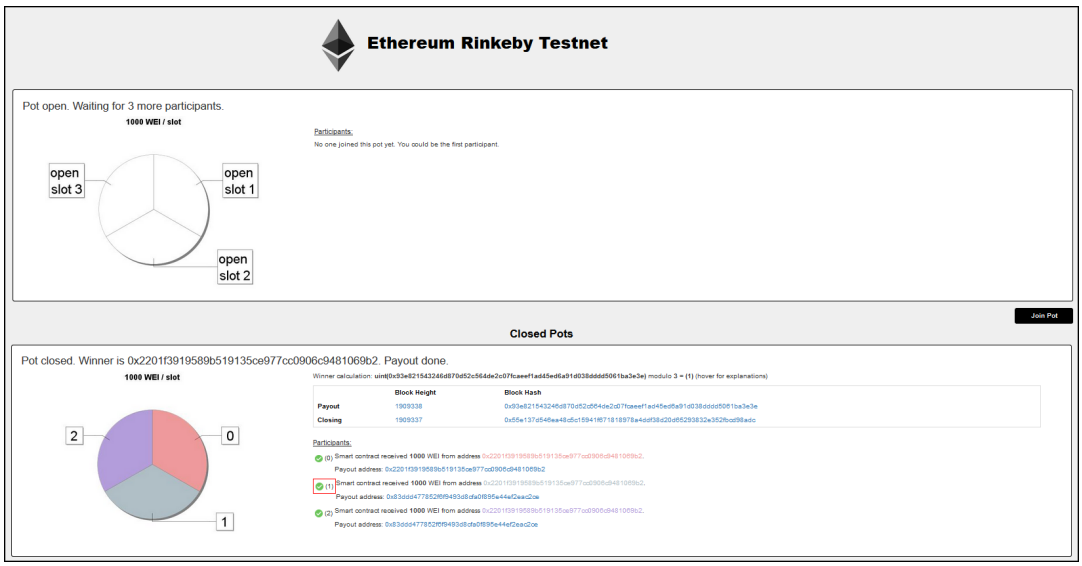


ABBILDUNG 3.14: Gewinner ausgewählt

Abbildung 3.14 zeigt den Gewinner des alten Topfs und den neu geöffneten Topf.

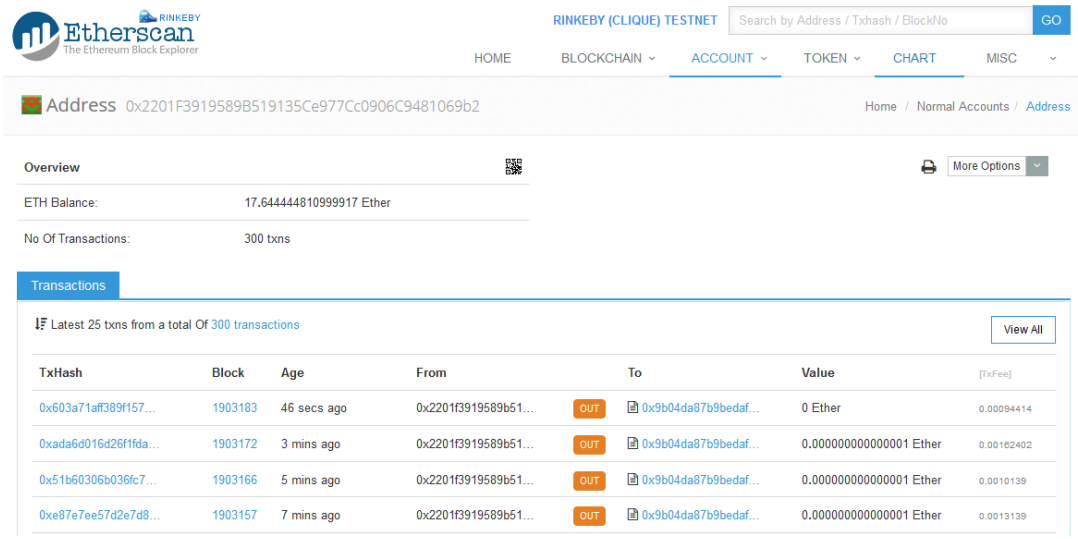


ABBILDUNG 3.15: Block Explorer: Smart Contract

Abbildung 3.15 zeigt die 3 Einzahlungstransaktionen und die Transaktion, die die `payout` Funktion aufruft.

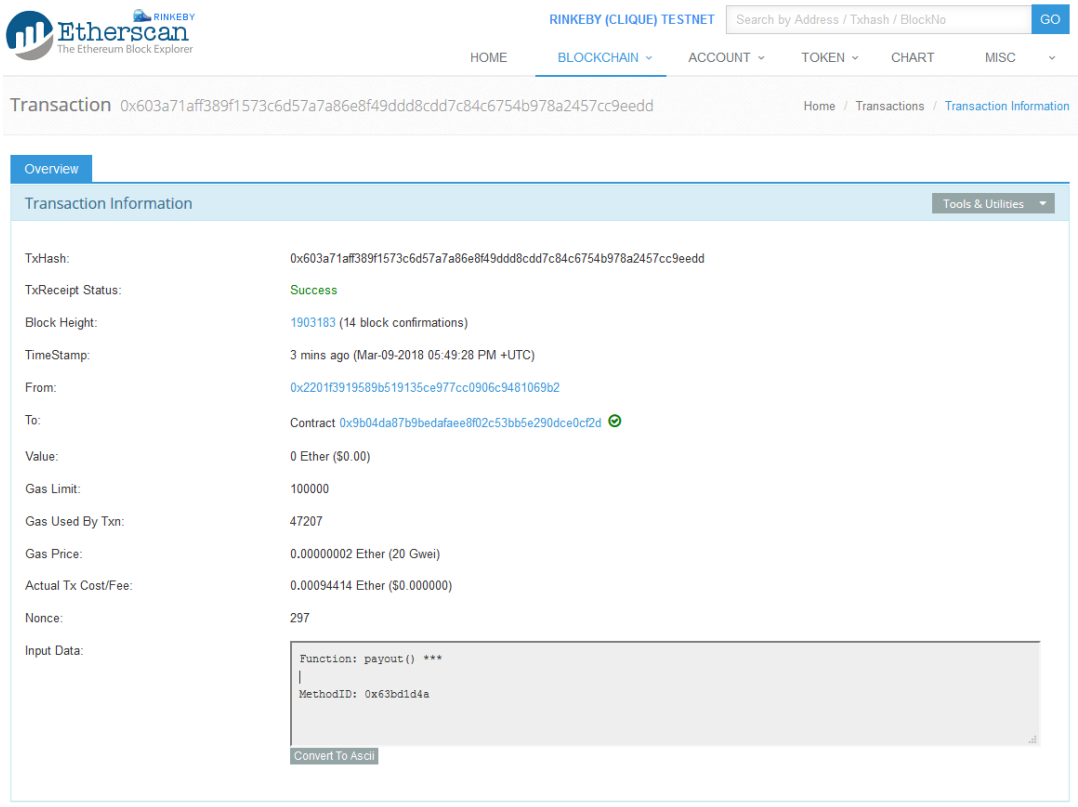


ABBILDUNG 3.16: Block Explorer: Payout Transaktion

Abbildung 3.15 zeigt die Details der Transaktion, die die `payout` Funktion aufruft.

3.4 Evaluation

3.4.1 Prüfung der Anforderungen

Dieser Abschnitt behandelt in wie weit das beschriebene Konzept die in Kapitel 1 aufgelisteten Anforderungen erfüllt. Die jeweilige Anforderung wird zunächst wiederholt und anschließend genauer untersucht.

1) Transparente Einzahlungen

Die Einzahlung jedes Endnutzers ist für jeden anderen Endnutzer nachprüfbar.

Einzahlungen geschehen genau wie bei Bitcoin innerhalb von Transaktionen, die in die Blockchain geschrieben werden. Diese Anforderung ist also auch im Falle von Ethereum erfüllt.

2) Gewinnerauswahl durch Zufallsfaktor

Die Auswahl des Gewinners ist von einem zufälligen Faktor abhängig, auf den weder die Anwendung noch die Endnutzer einen Einfluss haben.

Genau wie bei Bitcoin findet die Gewinnerauswahl basierend auf einem aus dem Proof-of-Work Blockhash statt. Die in Kapitel 2.4.1 betrachtete Analyse gilt somit genau so für Ethereum außer, dass der Mining Reward 3 Ether beträgt und durchschnittlich alle 12 Sekunden ausgeschüttet wird. In Zukunft plant Ethereum von einem Proof-of-Work Algorithmus auf einen Proof-of-Stake Algorithmus umzusteigen. Proof-of-Stake und die daraus resultierenden Auswirkungen werden in Kapitel 4.2 betrachtet.

3) Nachprüfbarkeit des Zufallsfaktor

Jeder Endnutzer kann die Echtheit des zufälligen Faktors eigenständig nachprüfen.

Der zur Gewinnerauswahl verwendete Blockhash ist zum Zeitpunkt der Auszahlung bereits in der öffentlichen Blockchain verankert und kann somit überprüft werden.

4) Transparente Auszahlungen

Die Auszahlung an den Gewinner muss transparent und somit für jeden Endnutzer nachprüfbar sein.

Die Auszahlung wird durch die vom Nutzer initiierte `payout` Transaktion ausgelöst und vom Smart Contract vorgenommen. Das Verfahren der Auszahlung ist durch den unveränderlichen Smart Contract Code in Stein gemeißelt. Die Konsensregeln und die dahinter liegende Spieltheorie garantieren, dass dieser auch genau so ausgeführt wird. Obwohl die `payout` Transaktion nicht direkt Geld auf die Auszahlungsadresse des Gewinners überweist, kann der Endnutzer dennoch sicher sein, dass eine Auszahlung stattgefunden hat, wenn die `payout` Transaktion wie in Abbildung 3.16 im Status `success` vorliegt.

5) Fairheit des Spiels

Jeder Endnutzer besitzt die gleiche Gewinnwahrscheinlichkeit und niemand wird benachteiligt.

Damit keiner der Spieler einen Vorteil hat, muss jeder Topf-Platz die gleiche Gewinnwahrscheinlichkeit haben. Dies ist gegeben, falls jeder Teilnehmer a) die gleiche Anzahl Gewinnzahlen zugeordnet bekommt und b) falls die möglichen Blockhash-Werte für die Gewinnerauswahl gleichverteilt sind.

a) Statt wie bei Bitcoin ausschließlich die letzte Ziffer des Blockhashs für die Gewinnerauswahl zu verwenden und als Konsequenz lediglich Töpfe der Größe 2, 5 und 10 anzubieten, wird bei Ethereum der gesamte Blockhash zur Gewinnerauswahl verwendet. Dies führt zu beliebig großen Töpfen, bei denen einige Teilnehmer genau eine Gewinnzahl mehr haben können. Da jeder Spieler in der Praxis mehrere Millionen von Gewinnzahlen hat, kann man diesen theoretischen Vorteil vernachlässigen.

b) Abschnitt 3.4.3 zeigt, dass die von Ethereum eingesetzte `Keccak-256` Hashfunktion gleichverteilte Werte liefert.

3.4.2 Aufruf der Auszahlungstransaktion

Wie bereits in Abschnitt 3.2 betrachtet, ist der Aufruf deiner Funktion zur Auszahlung unumgänglich. Da der Smart Contract dies nicht selber kann, muss der Aufruf entweder von außerhalb oder von einem Anderen Smart Contract kommen.

a) Aufruf von außerhalb:
Der Aufruf kann wie in der Implementierung vom Gewinner ausgeführt werden. In diesem Fall zahlt der Gewinner die Transaktionsgebühr und erhält den gesamten Topf-Betrag. Der Gewinner ist dafür zuständig die Funktion rechtzeitig aufzurufen, da der Gewinn sonst in den nächsten Topf übergeht. Eine andere Möglichkeit ist es, dass die Glücksspielanwendung den Smart Contract überwacht und die *payout* Funktion rechtzeitig aufruft. In diesem Fall müsste die Transaktionsgebühr von der Glücksspielanwendung gezahlt werden oder Funktionalität in den Smart Contract eingebaut werden, die die Transaktionskosten vom Topf-Betrag abzieht und der Glücksspielanwendung zurückerstattet. Allerdings verlässt sich der Gewinner dann auf die Anwendung und geht dadurch ein Risiko ein.

b) Aufruf durch Smart Contract:
Man kann in der Theorie den Ansatz des Ethereum Alarm Clock [16] Contracts [17] verwenden, um eine gewünschte Smart Contract Funktion zu einem späteren Zeitpunkt auszuführen. Man spezifiziert dazu welche Funktion man wann (in welchem Blockzeitraum) ausführen möchte und zahlt für die anfallenden Transaktionsgebühren im Voraus. Dies erlaubt, dass eine ganze Reihe von Funktionen sich bei dem Alarm Clock Contract registrieren. Wird nun der Alarm Clock Contract von einem durch einen privaten Schlüssel kontrollierten Account ausgelöst, werden alle registrierten Funktionen aufgerufen. Leider liefert diese Vorgehensweise keine Garantie, da eine registrierte Funktion nur aufgerufen wird, falls der Alarm Clock Contract aufgerufen wird. Die Glücksspielanwendung müsste also einspringen, sobald niemand anderes bereit ist den Alarm Clock Contract anzustoßen. Es handelt sich also lediglich um eine Vorgehensweise um Transaktionsgebühren mit anderen Ethereum Nutzern zu teilen.

[16] `http://www.ethereum-alarm-clock.com/`
[17] `https://etherscan.io/address/0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b`

3.4.3 Verteilung der Hashfunktion Keccak-256

Ethereum verwendet die kryptographische Hashfunktion Keccak-256. Die folgende Monte-Carlo-Simulation zeigt, dass die Hashwerte der Hashfunktion Keccak-256 gleichverteilt sind.

```
h=Keccak-256 n=1000000
for i 1 -> n
    hash = h(i);
    result[uint(hash)%10]++
```

```
Ausgabe:
result[0] =  99227
result[1] = 100479
result[2] = 100163
result[3] =  99804
result[4] =  99945
result[5] = 100208
result[6] = 100403
result[7] = 100438
result[8] = 100035
result[9] =  99298
```

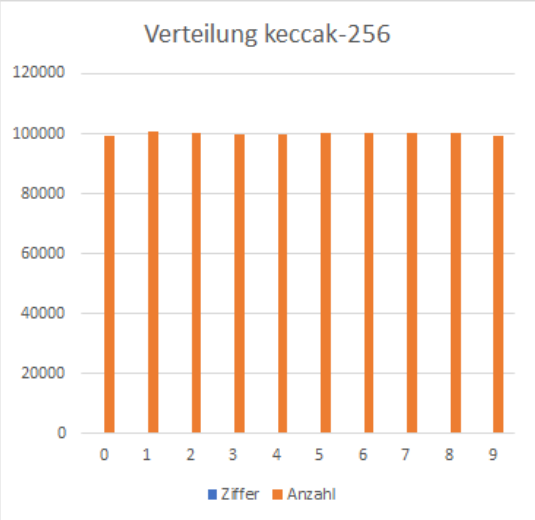


ABBILDUNG 3.17: Verteilung der Keccak-256 Hashfunktion

3.4.4 Sicherheit von Smart Contracts

Bei Smart Contracts handelt es sich um öffentliche, für jeden ausführbare und unveränderliche Software. Beinhaltet diese einen Software Fehler, ist dieser ausnutzbar und kann nicht behoben werden. Smart Contracts verwalten in der Regel Geld oder Token, die einen finanziellen Wert repräsentieren. Bei der Entwicklung eines Smart Contracts ist somit oberste Vorsicht geboten. Das Beispiel von the DAO zeigt, zu welchen Katastrophalen Folgen Sicherheitslücken in Smart Contracts führen können. Bei the DAO handelt es sich um einen von Christoph Jentzsch programmierten und am 20ten April 2016 auf der Ethereum Blockchain veröffentlicht Smart Contract[18]. The DAO ist ein Kapitalfond, der es sich zur Aufgabe gemacht hat in Blockchain Technologie zu investieren. Bei dem initialen 28zig tägigen Crowdsale wurden mehr als 150 Millionen Dollar von über 11 Tausend Investoren eingesammelt. Investoren haben Kapital in Form von Ether eingezahlt und als Gegenleistung eine entsprechende Anzahl Token als eine Art Stimmrecht erhalten. Investitionsentscheidungen dieser **d**ezentralen **a**utonomen **O**rganisation werden mithilfe des Smart Contracts durch einen dezentral erarbeiteten Konsens getroffen. Durch das Ausnutzen eines nicht trivialen Fehlers im Smart Contract Code schaffte es ein Hacker einen großen Teil des Kapitals an eine von ihm kontrollierte Adresse auszuzahlen. Eine genau Beschreibung des Angriffes findet man unter [9].

[18] https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code

Die Solidity Dokumentation [19] listet eine Reihe von Beispielen, die die Sicherheit von Smart Contracts betreffen. Entwickler sollten sich dieser bewusst sein, bevor sie einen Smart Contract veröffentlichen der Geld verwaltet.

[19] https://solidity.readthedocs.io/en/develop/security-considerations.html

Kapitel 4

Sonstige Blockchain-Technologie

4.1 Directed acyclic graph

Hier dann darauf eingehen, dass solch ein Ansatz keinen Sinn macht.

4.2 Konsensalgorithmus: Proof of stake

Hier kann man auch noch erwähnen, dass Proof of stake und solche slotbasierten Ansätze nicht geeignet sind da der Slotleader direkten Einfluss nehmen kann.

4.3 Payment Channels und Lightning Network

Hier könnte man darauf eingehen, dass off-chain Transaktionen nicht einsetzbar sind, da die restlichen Teilnehmer somit nicht die Einzahlung überprüfen können.

Kapitel 5

Ausblick

Kapitel 6

Fazit

Quellenverzeichnis

[1] Andreas M. Antonopoulos. *Mastering Bitcoin*. O'Reilly, 2015. URL: https://github.com/bitcoinbook/bitcoinbook (besucht am 03. 03. 2018).

[2] *Bitcoin Full Node API*. 2015. URL: http://chainquery.com/bitcoin-api (besucht am 09. 02. 2018).

[3] *Bitcoin Improvement Proposal 21*. 2013. URL: https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki (besucht am 03. 03. 2018).

[4] *bitcoinj*. 2011. URL: https://bitcoinj.github.io/ (besucht am 09. 02. 2018).

[5] *Blockchain Info Bitcoin Explorer*. 2011. URL: https://blockchain.info/ (besucht am 09. 02. 2018).

[6] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. Nov. 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper (besucht am 03. 03. 2018).

[7] *Clique PoA protocol and Rinkeby PoA testnet*. 2017. URL: https://github.com/ethereum/EIPs/issues/225 (besucht am 16. 03. 2018).

[8] *Crypto Games*. 2014. URL: https://www.crypto-games.net/ (besucht am 09. 02. 2018).

[9] *Deconstructing theDAO Attack*. 2016. URL: http://vessenes.com/deconstructing-thedao-attack-a-brief-code-tour/ (besucht am 31. 03. 2018).

[10] Igor Drobiazko. *Tapestry 5: Die Entwicklung von Webanwendungen mit Leichtigkeit*. Pearson Deutschland, 2010.

[11] *ethereumj*. 2016. URL: https://github.com/ethereum/ethereumj (besucht am 15. 03. 2018).

[12] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Nov. 2008. URL: https://bitcoin.org/en/bitcoin-paper (besucht am 03. 03. 2018).

[13] *Peer-to-Peer*. 2018. URL: https://de.wikipedia.org/wiki/Peer-to-Peer (besucht am 01. 04. 2018).

[14] *Pokerstars*. 2001. URL: https://www.pokerstars.eu (besucht am 09. 02. 2018).

[15] Frank Stajano und Richard Clayton. „Cyberdice: Peer-to-Peer Gambling in the Presence of Cheaters". In: *Lecture Notes in Computer Science* 62.1 (Jan. 2011), S. 1–20. URL: https://link.springer.com/chapter/10.1007/978-3-642-22137-8_9.

[16] *vDice Ether Games*. 2016. URL: https://www.vdice.io/ (besucht am 09. 02. 2018).

[17] *web3j*. 2016. URL: https://github.com/web3j/web3j (besucht am 15. 03. 2018).

[18] Sam Wouters. *Why Schnorr signatures will help solve 2 of Bitcoin's biggest problems today*. 2017. URL: https://medium.com/@SDWouters/why-schnorr-signatures-will-help-solve-2-of-bitcoins-biggest-problems-today-9b7718e7861c (besucht am 10. 03. 2018).

Abbildungsverzeichnis