Blockchain – selber machen

Thomas Deniffel, Skytala GmbH

Mit Code können wir uns unmissverständlich ausdrücken. Dies ermöglicht, das Thema „Blockchain" in diesem Artikel klar an einem Beispiel zu erläutern und eine Blockchain inklusive Kryptowährung zu erstellen. Nach den Grundlagen entdecken wir während der Implementierung, welche Komponenten eine Blockchain ausmachen, was sie so besonders und fehlerresistent macht und wie genug Vertrauen in einem anonymen Netzwerk so hergestellt werden kann, dass sich sogar Währungen umsetzen lassen. So kann man während des Artikels mittels Git die Implementierung in der IDE seiner Wahl mit verfolgen.

„Blockchain = Bitcoin". Die Begriffe werden oft fälschlicherweise als Synonym verwendet – die Unterscheidung fällt jedoch einfach: Die Kryptowährung Bitcoin wird mithilfe einer Blockchain implementiert. Das Besondere an solchen Blockchains ist, dass Parteien, die sich gegenseitig nicht vertrauen, sicher miteinander interagieren und handeln können. Im anonymen Internet, das unsere globale Wirtschaft vernetzt, ist das eine vielversprechende Eigenschaft. Kryptographie spielt für die sichere Interaktion anonymer Akteure eine entscheidende Rolle, wodurch der Begriff „Kryptowährung" entstanden ist.

Seit dem Jahr 2009 hinterfragt Bitcoin, was wir unter Geld verstehen. Viele assoziieren Geld mit Banken, Geldnoten und Überweisungen, wobei Bitcoin gleich die ersten beiden Punkte streicht. Doch warum brauchen wir Geld überhaupt? Jeder Mensch arbeitet (vereinfacht) acht Stunden am Tag an einem speziellen Produkt, möchte aber eine Vielzahl verschiedener Dinge kaufen. Daher muss er seine Arbeitszeit gegen andere Produkte eintauschen, die er nicht selbst herstellt. Das machen wir, indem wir unsere Arbeitszeit gegen Geld tauschen, das wir wiederum in Arbeitszeit in Form von Produkten Anderer tauschen.

Das Prinzip funktioniert, sobald alle ihre geleistete Arbeitskraft (das Einkommen) und die verbrauchte Arbeitskraft (die Ausgaben) zuverlässig dokumentieren – ohne zu betrügen. Dazu nutzen wir seit langer Zeit Konten. Ein Konto ist eine unveränderbare Liste, der neue Buchungen – positiv wie negativ – durch Transaktionen hinzugefügt

werden. Eine Transaktion beschreibt immer zwei Kontoeinträge: Abbuchung und Einzahlung in der gleichen Höhe, wodurch das Gesamtsystem konsistent bleibt. Ist die Anzahl der Konten klein genug, kann dazu ein einfaches Blatt Papier dienen, auf dem wir die Transaktionen notieren *(siehe Listing 1)*.

Es funktioniert, solange es wenige Konten gibt und niemand betrügt. Im Großen brauchen wir jedoch Banken, die diese Liste verwalten und überwachen. Wir sehen diese nach wie vor auf unseren Kontoauszügen, gefiltert auf für uns relevante Einträge. Eine Währung lässt sich also durch eine einfache lineare Liste umsetzen. Sie muss nur folgende Eigenschaften erfüllen:

- Unveränderbar
- Erweiterbar
- Invalide Operationen werden erkannt und verworfen

So wie Banken erfüllen Blockchains diese drei Eigenschaften; sie eignen sich daher als Währung. Das Besondere: Statt mit zentralem Transaktionsmanagement funktioniert eine Blockchain dezentral – selbst in einem Netzwerk, in dem sich die Teilnehmer gegenseitig nicht vertrauen. Die Datenstruktur Blockchain kann aber für mehr genutzt werden, etwa für Echtheitsnachweise von Dokumenten oder die Verfolgung von Transporten. Sobald Transaktionen zwischen Parteien stattfinden, die sich nicht kennen und/oder vertrauen, bietet sich ein Szenario für Blockchains.

Die grundlegende Idee

Damit eine Blockchain eine unveränderbare, sequenzielle und erweiterbare Kette von Blöcken bildet, ketten wir die Blöcke, die beliebige Nutzdaten beinhalten, durch Hashes aneinander *(siehe Abbildung 1)*. Möchten wir eine Währung implementieren, befindet sich im Block ein Teil der Liste von Transaktionen (wie im obigen Beispiel). Durch die Verkettung der Blöcke ist es nicht möglich, Elemente zu verändern, zu löschen oder einzufügen, denn jeder Bock speichert den Hash des Vorgänger-Blocks zur Zeit des Hinzufügens. Wird der Vorgänger-Block verändert, ändert sich der Hash, was dann erkannt und verworfen wird. Dazu später mehr beim Konsens-Algorithmus.

Als Use Case für eine Implementierung erschaffen wir nun unsere eigene Währung: JCoin. Die Klassenstruktur sieht wie in *Abbildung 2* aus. Es ist weder möglich noch sinnvoll, den kompletten Code abzudrucken. Es besteht allerdings die Möglichkeit, parallel mitzuentwickeln, indem man das vorbereitete Repository mit „git clone https://github.com/tom-010/jcoin.git" klont. Initial enthält es nur Boilerplate-Code. Man kann sich durch Checkouts im Artikel synchronisieren und die Entwicklung des Codes im Laufe des Artikels beobachten.

```
Alice -> Bob:    100 (Alice: 900, Bob: 1100)
Bob   -> Carol: 50 (Carol: 1050, Bob: 1050)
Carol -> Alice: 500 (Carol: 550, Alice: 1400)
...
```

Listing 1

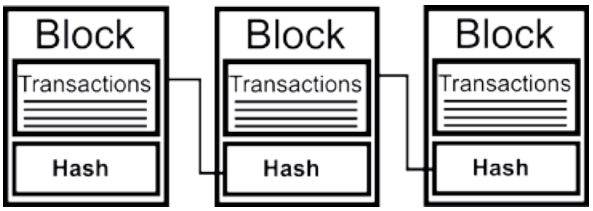

Abbildung 1: Blockchain

Block

Blöcke sind die Grundlage jeder Blockchain. Neben dem Hash sind hier die Transaktionen sowie andere Felder abgelegt, die wir entweder im Laufe des Artikels erkunden oder die als Implementierungshilfe dienen *(siehe Listing 2)*.

Jeder Block (als JSON serialisiert) beinhaltet den Hash des Vorgänger-Blocks, jeder Block besitzt also rekursiv die Hashes aller seiner Vorgänger. Um zu verifizieren, ob eine gegebene Kette nicht verändert wurde, traversieren wir die Liste an Blöcken (die Blockchain), hashen den aktuellen Block und prüfen, ob der berechnete Hash mit dem im nächsten Block übereinstimmt. Die Methode „validChain" zeigt später, wie das in Form von Code aussieht. Man sollte sich an dieser Stelle unbedingt die Zeit nehmen zu verstehen, wie das Hashing die Unveränderbarkeit garantiert.

Trotz Hashes könnte man einen Block ändern, einfügen oder löschen und die folgenden Hashes neu berechnen und aktualisieren. Diesen Angriff verhindert allerdings der Proof of Work, der zusammen mit dem Gossip-Protokoll dafür sorgt, dass manipulierte Ketten verworfen werden. Das funktioniert, weil der Proof of Work mitgehasht wird. Diese Aussage sollte man beim ersten Lesen überspringen und nach dem Artikel nochmal hierher zurückkehren.

Proof of Work

Könnte man ohne Aufwand Blöcke erzeugen, würde das die Synchronisierung im dezentralen Netz unmöglich machen. Zusätzlich

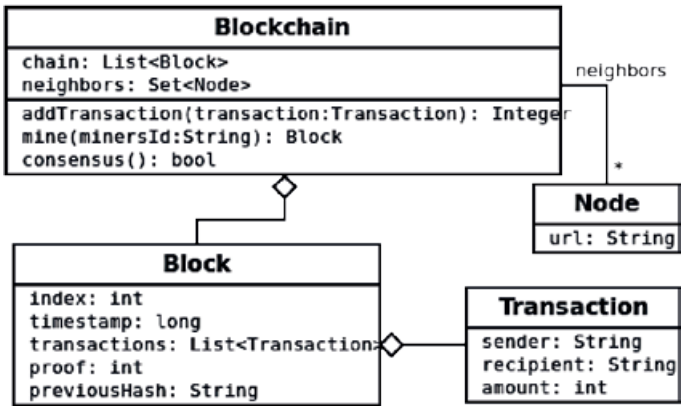

Abbildung 2: Die Klassenstruktur

```
{
  "index": 1,
  "timestamp": 1536418771131,
  "transactions": [...],
  "proof": 196759,
  "previousHash": "74234e...982b90b"
}
```

Listing 2

käme es innerhalb von Sekunden zu einer Inflation, die eine Währung wertlos macht. Der benötigte „Proof of Work" (PoW) verhindert das einfache Erzeugen eines Blocks. Der PoW ist eine Zahl, die zwei Kriterien erfüllt:

- Sehr aufwendig zu erstellen
- Sehr einfach zu kontrollieren

Der Algorithmus, der einen solchen PoW liefert, bedarf viel Rechenarbeit. Das Ergebnis wird dann jedoch von jedem Teilnehmer im Netz überprüft. Beim PoW entdecken wir die Analogie zum antiken Zahlungsmittel Gold: Es ist schwer zu finden (vor allem früher), man kann jedoch einfach durch einen Blick verifizieren, dass es wirklich Gold ist. Aus dieser Analogie stammt die Metapher „minen", was die Ausführung des PoW-Algorithmus meint. Wird ein PoW gefunden, kann ein neuer Block erstellt werden; ein neuer Block wurde also „gemint". Der PoW-Algorithmus ist leicht umzusetzen. Man spezifiziert einen PoW als Zahl x, die mit einer Zahl y (aus dem Vorgänger-Block) multipliziert wird. Der Hash des Ergebnisses muss mit einer 0 enden. Bei gegebenem y suchen wir also ein x, sodass „hash(x * y) = ...0" erfüllt ist *(siehe Listing 3)*.

Lesen wir etwa „y=4" aus dem Vorgänger-Block, ist die Lösung hier „x=37", da der erzeugte Hash mit 0 endet: „hash(4 * 37) = 12....0". Gängige Hashing-Algorithmen sorgen dafür, dass die Wahrscheinlichkeit, einen neuen PoW zu finden, über alle Versuche gleichverteilt ist. Gleichverteilung ist unabhängig von Raum und Zeit gültig, was zu einem fairen System führt. Damit ist die Wahrscheinlichkeit, einen neuen Block zu finden, im Netz gleichverteilt und jeder hat die gleichen Chancen – ohne dass eine Zentrale diese Fairness über ein Protokoll erzwingt.

Um Teilnehmer dazu zu bringen, die Rechenleistung für das Minen bereitzustellen, belohnen wir bei JCoin wie Bitcoin (dessen PoW-Algorithmus heißt übrigens „HashCash" und funktioniert ähnlich) das Mining mit einer Geldeinheit, was die Inflation der Währung an die Schwierigkeit koppelt. Um die Geschwindigkeit der Inflation anzupassen, wird die Schwierigkeit des PoW geändert, indem mehr Nullen (bei JCoin vier) am Ende des Hash gefordert sind. Das lässt die Schwierigkeit exponentiell ansteigen und die Inflation exponentiell verringern (siehe „git checkout pow").

Mining

Beim Mining geschieht die Magie einer Kryptowährung, denn hier wird der Wert der Währung erzeugt. Es besteht aus drei Schritten *(siehe Listing 4)*.

- Den Proof of Work berechnen
- Sich belohnen, indem man sich ein JCoin zuweist
- Einen neuen Block der Blockchain hinzufügen

Der Miner bekommt ein JCoin als Belohnung. Das kann nur über eine Transaktion geschehen, es gibt jedoch keinen Sender, da das Geld erschaffen wird. Deshalb setzt man „0" als Sender, was das Erzeugen der Geldeinheit darstellt: „git checkout mining". Ist ein PoW gefunden, ist man in der Lage, einen neuen Block zu erschaffen. Dazu wird alles bis dato Gesammelte zusammengefügt *(siehe Listing 5)*.

```
Integer proofOfWork(Integer lastProof) {
  Integer proof = 0;
  while(!validProof(lastProof, proof))
     proof += 1;
  return proof;
}

boolean validProof(Integer lastProof, Integer proof) {
  String guessHash = sha256(lastProof.toString()+proof.
toString());
  return guessHash.endsWith("0"); // 0000 später
}
```

Listing 3

```
Block mine(String minersNodeId) {
  // Sync mit dem rest des
  // Netzwerks. Später mehr
  consensus();

  Block lastBlock = lastBlock();
  // proof of first block is 0;
  Integer lastProof =(lastBlock != null) ?
                     lastBlock.getProof() : 0;

  Integer proof = proofOfWork(lastProof);

  // Belohnung für uns!
  addTransaction("0", minersNodeId, 1);

  String previousHash = hash(lastBlock);
  Block block = addBlock(proof, previousHash);

  // Nachbarn benachrichtigen -> Block übernehmen
  for(Node neighbor : getNeighbors())
     neighbor.resolveConflicts();
  return block;
}
```

Listing 4

```
Block addBlock(int proof, String previousHash) {
  // ...
  Block block = new Block();
  // ...
  block.setProof(proof);
  block.setPreviousHash(previousHash);
  block.setTransactions(currentTransactions);
  currentTransactions = new LinkedList<>();

  chain.add(block);
  return block;
}
```

Listing 5

```
Integer addTransaction(Transaction transaction) {
  currentTransactions.add(transaction);
  return lastIndex();
}

class Transaction {
  String sender;
  String recipient;
  Integer amount;
  // ...
}
```

Listing 6

esentri

IT'S IN ALL OF US *TO CREATE*

Jetzt bewerben unter
career@esentri.com

#inallofus
www.esentri.com

```
@RestController
public class BlockchainApi {

  Blockchain blockchain = new Blockchain();
  // Zufälligen Name für uns
  String ownNodeID = randomName();

  @GetMapping("/mine") // Mining anstoßen
  MiningMessage mine() { ... }

  @PostMapping("/transactions/new")
  TransactionAddedMessage newTransaction(Transaction transaction) { ... }
  @GetMapping("/chain") // zum kontrollieren/kopieren für andere
  List<Block> fullChain() { ... }

  @PostMapping("/nodes/register")
  String registerNodes(Node node) { ... }

  @GetMapping("/nodes/resolve") // Konsens-Algorithmus
  ResolvedMessage consensus() { ... }

  @GetMapping("/nodes/all")
  Set<Node> allNodes() { ... }
}
```

Listing 7

```
$ curl -X POST -H "Content-Type: application/json" -d ,{
 "sender": "my-id",
 "recipient": "others-id",
 "amount": 2
}‘ "http://localhost:8080/transactions/new"
```

Listing 8

Neue Transaktion

Die Transaktionen, die einem der Benutzer sendet, werden so lange zwischengespeichert (in „currentTransactions“), bis man einen neuen Block „mint“ *(siehe Listing 6)*. Damit ist eine Transaktion so lange transient, bis ein neuer Block gefunden wurde. Danach wird sie im Block in der Blockchain persistiert.

Weil eine Blockchain eine dezentrale Datenstruktur ist, existiert ein Netzwerk mit gleichberechtigten Nodes. Sie kommunizieren im Beispiel mittels REST-Schnittstellen. Implementiert ein Node diese, ist er Teil des Netzwerks. Dazu kommt Spring zum Einsatz *(siehe Listing 7)*. Nun lassen sich neue Transaktionen anlegen *(siehe Listing 8)*. Um die Transaktion zur Blockchain hinzuzufügen (persistieren), wird ein neuer Block „gemint“ *(siehe Listing 9)*.

Mining bringt uns ein JCoin. Deshalb können wir auch rund um die Uhr in einem Extra-Thread minen. Haben wir zwei Blöcke „gemint“, erhalten wir die Blockchain wie in *Listing 10*.

Konsens

Nun haben wir eine funktionstüchtige Blockchain: Transaktionen werden akzeptiert und neue Blöcke können „gemint“ werden, dazu ist sie unveränderbar und sequenziell. Eine grundlegende Idee fehlt jedoch: Das Ganze soll nicht zentral auf einem Rechner laufen, sondern dezentral. Dazu werden invalide Ketten noch nicht verworfen. Es gibt keine Zentrale wie einen Server, der als „Source of Truth“ dient. Wie können wir erreichen, dass jeder Node die gleiche und korrekte Blockchain repräsentiert – und das

```
$ curl "http://localhost:8080/mine"
{
"message":"New Block Forged",
"Index":0,
"Transactions":[
  {"sender":"my-id","recipient":"others-id","amount":2},
  {"sender":"0","recipient":"my-id","amount":1}],
"proof":4,
"previousHash":"74..90b"
}
```

Listing 9

```
$ curl "http://localhost:8080/chain"
[{
  "index":0,
  "timestamp":1537730860721,
  "transactions":[...]
}]

git checkout rest
```

Listing 10

in einem Netzwerk, in dem sich keiner vertraut? Wir brauchen einen Algorithmus, der Konsens herstellt.

Minen verschiedene Nodes unkoordiniert neue Blöcke, kommt es zwangsläufig zu Konflikten. Denn immer dann, wenn zwei Nodes gleichzeitig einen Block finden, ist zu entscheiden, welcher Block aufgenommen und welcher verworfen wird. Ohne zentralen Server gibt es keine Möglichkeit, pessimistisches Locking zu koordinieren, wodurch zwangsläufig im Zuge des optimistischen Locking gemergt werden muss.

Beim Mergen lassen sich die Blöcke nicht verändern, man muss also Blöcke entweder verwerfen oder aufnehmen. Um zu entscheiden, welche Blöcke übernommen werden, definiert man eine einfache Regel: Die längste valide Blockchain wird übernommen und der Rest

```
boolean consensus() {
  List<Block> newChain = new LinkedList<>();
  Integer maxLength = chain.size();

  for(Node node : neighbors) {
     List<Block> ownChain = node.readChain();
     if(ownChain == null)
        continue; // ignore

     Integer length = ownChain.size();

     // Andere Blockchain länger? Übernehmen!
     // -> Die Idee des Konsens-Algorithmus
     if(validChain(ownChain) && length > maxLength) {
        maxLength = length;
        newChain = ownChain;
     }
  }

  if(!newChain.isEmpty()) {
     this.chain = newChain;
     return true;
  }
  return false;
}

validChain(List<Block> chain) {
  for(int idx = 1; idx < chain.size(); idx++) {
     Block lastBlock = chain.get(idx-1);
     Block block = chain.get(idx);

     // Hash korrekt?
     if(!block.getPreviousHash().
equals(hash(lastBlock)))
        return false;

     // Proof of Work korrekt?
     if(!validProof(lastBlock.getProof(), block.get-
Proof()) )
        return false;
  }
  return true;
}
```

Listing 11

verworfen; neu gefundene Blöcke werden sofort einer Version der Blockchain angehängt, es entstehen also zwei verschiedene valide Blockchains. So schafft man Konsens unter den Knoten im Netzwerk. Der offensichtliche Nachteil: Sobald eine Transaktion in einem Block untergebracht ist, ist nicht mehr sicher, dass der Block auch Teil der Blockchain bleibt, denn es kann passieren, dass der Block irgendwann (eventuell auch erst Jahre später) verworfen wird, sobald sich eine längere Kette mit einem synchronisiert. Konkret sieht das wie in *Listing 11* aus.

„validChain" überprüft, ob eine Blockchain gültig ist. Dazu prüft sie den Hash (die Verkettung ist korrekt) sowie den Proof of Work (die Arbeit wurde auch wirklich geleistet) jedes Blocks. So entsteht die Unveränderbarkeit und invalide Ketten werden verworfen, indem der Node mit der zerbrochenen Kette schlicht von allen Nachbarn ignoriert wird und damit nicht mehr Teil des Netzwerks ist. Hat jedoch ein Nachbar eine valide längere Kette, wird diese übernommen.

Dass es Nachbarn gibt, ist der Schlüssel zum dezentralen System: Es existiert kein zentraler Server, bei dem man sich registrieren muss. Man fügt einfach einen Nachbarn hinzu und fängt an, neue Blöcke zu minen.

Gossip-Protokoll – sag es weiter!

„Mint" man einen Block, ist dies den anderen im Netz mitzuteilen, damit er nicht verworfen wir. Man kennt jedoch nur seinen direkten Nachbarn, weshalb man ihn nicht einfach allen Teilnehmern schicken kann. Dies löst das Gossip-Protokoll: Findet man einen neuen Block, wächst die Blockchain um ein Element; sie ist damit länger als die der Nachbarn. Man benachrichtigt sie, woraufhin diese unsere Blockchain mit dem Konsens-Algorithmus prüfen und übernehmen. Ist das geschehen, benachrichtigen sie wiederum ihre Nachbarn. So findet unser Block seinen Weg durch das Netz, wobei die Geschwindigkeit zeitweise exponentiell steigt.

Das inhärente Problem ist jedoch, dass wir nicht feststellen können, ob jeder den Block erhalten hat. Befindet sich irgendwo ein Subnetz, dessen Nodes in der gleichen Zeit mehr Blöcke findet, wird der Block verworfen, sobald sich das Subnetz mit dem Netz verbindet. Deshalb kann man nie ganz sicher sein, dass eine Transaktion tatsächlich gespeichert ist. Es gibt Workarounds für dieses Problem, doch das Wichtigste ist, dass der Proof of Work schwer genug ist, dass zwei Blockchains nicht besonders weit divergieren können. Das Gossip-Protokoll wurde bereits im Konsens-Algorithmus implizit mit umgesetzt (siehe „git checkout distributed").

Fazit

Das war es: Es gibt eine dezentrale Blockchain, die alle nötigen Eigenschaften erfüllt, um eine Währung zu implementieren. Der Autor empfiehlt, mit Freunden Transaktionen durchzuführen und danach zu versuchen, die Blockchain zu löschen. Sobald nur ein Node nicht heruntergefahren ist, überlebt sie. Diese Fehlertoleranz ist eine der vielversprechenden Fähigkeiten einer Blockchain und wir werden die nächsten Jahre viele interessante Anwendungen entdecken.

Thomas Deniffel
tdeniffel@acm.org

Thomas Deniffel ist CTO bei der Skytala GmbH, die durch neueste Technologien individuelle Software-Projekte umsetzt. Dort ist er für die Auswahl der richtigen Werkzeuge zuständig und evaluiert in diesem Zuge die neuesten Trends. Dabei ist er weder Fan von Hypes noch vom religiösen Bestehen auf Programmiersprachen oder Tools, denn den eigenen Kopf zu verwenden ist gefragt. In seiner Freizeit hält er regelmäßig Präsentationen und Workshops und ist Organisator zweier Meetup-Gruppen mit den Schwerpunkten „Software-Entwicklung", „Software Craftsmanship" und „Clean Code".