blockchain

Richard van Dijk

2 november 2017

Inhoud

1 Inleiding

Blockchain lijkt het nieuwe hypewoord van deze tijd te worden. Sinds de lancering van Bitcoin in 2009 hebben cryptocurrency, en daarmee ook de achterliggende blockchaintechniek, een hoge vlucht genomen. De interesse in de markt groeit zo snel dat het lijkt op een nieuwe bubbel; zo stegen de aandelen van het Britse bedrijf *On-line Plc* met bijna 400% nadat de naam van het bedrijf veranderd werd in *On-line Blockchain Plc*. Maar waar ligt de kracht van blockchain en waarom wordt het de 'techniek van de toekomst' genoemd? En hoe werkt blockchain precies? Dit artikel beschrijft de achtergrond van blockchain en laat een *proof-of-concept* implementatie zien in Python 3.6.

2 Geschiedenis

Één van de voorlopers van blockchain werd beschreven in 1990, in een artikel over hoe digitale documenten een tijdsaanduiding kunnen krijgen. In dit artikel beschrijven onderzoekers Haber en Stornetta een methode waarbij de *hashes* van een document aan elkaar gelinkt worden, om rommelen met de tijdsaanduiding onmogelijk te maken.

3 Technische achtergrond

3.1 Een voorbeeldblock

Om beter te begrijpen hoe blockchain werkt is het goed om te bekijken wat de structuur van een block in de chain is. De meest simpele implementatie met *proof-of-work* bevat in ieder geval de volgende onderdelen:

- Een block ID;
- De *hash* van het voorgaande block;
- Gegevens die in het block zijn opgeslagen;
- Een *nonce*-waarde.

In veel gevallen wordt ook een tijdsaanduiding toegevoegd, zodat het block gedateerd is. We zullen nu de verschillende onderdelen van het block uitleggen.

Het block ID is het volgnummer van het block in de chain. Ieder nieuw block krijgt een oplopend nummer als unieke identificatie.

Vervolgens is er de *hash* van het voorgaande block. Een hashfunctie berekent uit een reeks bytes van variabele lengte een praktisch unieke reeks bytes van gefixeerde lengte. Dit heet de hash. Het algoritme in de hashfunctie is zo gemaakt, dat dezelfde reeks bytes altijd naar dezelfde hash evalueert en dat het origineel niet af te leiden is uit de hash. Een bekend voorbeeld van een hashfunctie is het MD5-algoritme, dat een hash van 128 bits geeft. Op Linux is dit vanuit een terminal beschikbaar met het programma `md5sum`:

```
[user@machine ~]$ echo 'abcdefg' >> file
[user@machine ~]$ md5sum file
020861c8c3fe177da19a7e9539a5dbac  file
```

Voor gebruik in een blockchain is het belangrijk een cryptografisch sterke hashfunctie te gebruiken. Dit houdt in dat er geen botsingen tussen hashes optreden. Er is sprake van een botsing tussen

hashes als twee verschillende waarden naar dezelfde hash evalueren. In het eerder genoemde MD5-algoritme zijn meerdere botsingen aangetoond en dit is dus niet bruikbaar. Momenteel wordt vaak het SHA-algoritme gebruikt.

Het volgende onderdeel in ons voorbeeldblock zijn de gegevens die in het block zijn opgeslagen. Dit kan in principe van alles zijn. In het geval van cryptocurrency worden in de blokken transactiegegevens opgeslagen.

Het laatste veld dat in ons block zit is de zogenaamde *nonce*-waarde. Dit is een speciaal getal dat wordt gebruikt om de hash van het block te manipuleren zodat hij voldoet aan de voorwaarde voor de hash. Deze vereiste is één van de kernpunten van de kracht van blockchain.

Meerdere andere zaken kunnen aan het block worden toegevoegd, zoals error checking via bijvoorbeeld CRC (Cyclic Redundancy Check), de hash van het block zelf of de eerder genoemde timestamp.

3.2 Blocks koppelen en Proof-of-Work

Nu we weten waaruit een block is opgebouwd is het mogelijk om preciezer in te gaan op de functie van twee onderdelen van het block: de hash van het vorige block en de *nonce*-waarde.

De hash van het voorgaande block is opgenomen in het huidige block, zodat deze waarde meeweegt in het bepalen van de hash van het huidige block. Deze koppeling tussen blocks verzekert dat voorgaande blocks niet veranderd kunnen worden zonder alle navolgende blocks óók te moeten herberekenen. Alle blocks wijzen immers terug naar het voorgaande block. Hier lopen we echter tegen een probleem aan. Het is namelijk voor een computer anno 2017 vrij eenvoudig om een grote hoeveelheid hashes opnieuw te berekenen. In het geval van SHA-256 is het niet ongebruikelijk voor een normale CPU om vele miljoenen hashes per seconde te kunnen uitrekenen. Om te voorkomen dat alle blokken snel herberekend kunnen worden door een kwaadwillende, moet de hash van het block aan een aantal voorwaarden voldoen. In het geval van Bitcoin moet de hash van het block met een bepaald aantal nullen beginnen. Deze vereisten heten het *target*. Hoe moeilijk het is om aan het target te voldoen wordt de *difficulty* genoemd, en het blijkt dat met een hogere difficulty de tijd voor het herberekenen van een block exponentieel stijgt.

Omdat de hash van het block niet verandert zonder iets toe te voegen, wordt er een arbitrair getal aan het block toegevoegd. Dit is de *nonce*-waarde. Door de noncewaarde te variëren (meestal door de waarde steeds met één op te hogen) kan er steeds een andere hash berekend worden totdat deze aan de voorwaarde voldoet. Het proces van het vinden van de noncewaarde wordt ook wel *mining* genoemd. Zodra de noncewaarde bepaald is wordt deze in het block opgeslagen. Dit systeem van voldoen aan een bepaalde voorwaarde voor de hash door bepaling van de overeenkomstige noncewaarde staat bekend als Proof-of-Work.

3.3 Het begin van de chain

Het eerste block van de chain is een speciaal geval. Het block waar alle andere blocks van afstammen wordt ook wel een *genesis*-block genoemd. Dit block, dat meestal als block ID 0 heeft, kan niet terugwijzen naar de hash van een voorgaand block. Dit veld wordt dan ook meestal op 0 gezet.

4 Implementatie

Met deze achtergrondinformatie is het nu mogelijk om een blockchain-implemenatie te schrijven in Python 3.6. We gebruiken hierbij twee libraries: `hashlib` en `time`. Deze libraries declareren we aan het begin van ons programma:

```
1  #!/usr/bin/env python3
2  import hashlib
3  import time
4
5
```

De library `hashlib` verleent ons de hashfunctionaliteit. In de implementatie die gedemonstreerd zal worden wordt het SHA256-algoritme gebruikt. De library `time` geeft ons de mogelijkheid om de huidige UNIX-timestamp op te vragen, die we als tijdsaanduiding in het block zullen verwerken.

We maken vervolgens een `class` aan met daarin de structuur van onze blocks; de velden in dit block zijn het block ID, de hash van het voorgaande block, de UNIX-timestamp, eventuele gegevens in het block, en tenslotte de nonce-waarde.

```
6  class Block:
7      def __init__(self, index, timestamp, data, previoushash, target):
8          self.index = index
9          self.previoushash = previoushash
10         self.timestamp = timestamp
11         self.data = data
12         self.nonce = 0
13
```

Op het moment dat het block aangemaakt wordt, moet de noncewaarde berekend worden, zodat de hash van het block aan het target voldoet. We definiëren ons target als **het aantal bytes aan het begin van de hash dat nul moet zijn**.

```
14         while True:
15             hashobject = self.calculate_hash(f'{index}'
16                                              f'{previoushash}'
17                                              f'{timestamp}'
18                                              f'{data}'
19                                              f'{self.nonce}')
20
21             if int.from_bytes(hashobject.digest()[0:target], byteorder='big') != 0:
22                 self.nonce += 1
23             else:
24                 self.hash = hashobject.hexdigest()
25                 break
26
27     def calculate_hash(self, blockdata):
28         message = hashlib.sha256()
29         message.update(blockdata.encode())
30         return message
31
32
```

5 Bronvermelding

Haber, Stornetta: How to time-stamp a digital document. https://www.anf.es/pdf/Haber_Stornetta.pdf
https://bitcoin.org/bitcoin.pdf
https://www.bloomberg.com/news/articles/2017-10-27/what-s-in-a-name-u-k-stock-surges-394-on-blockch
https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/
http://cryptomining-blog.com/4456-what-hashrate-to-expect-from-an-up-to-date-cpu/

6 Appendix: Volledige code

```python
#!/usr/bin/env python3
import hashlib
import time


class Block:
    def __init__(self, index, timestamp, data, previoushash, target):
        self.index = index
        self.previoushash = previoushash
        self.timestamp = timestamp
        self.data = data
        self.nonce = 0

        while True:
            hashobject = self.calculate_hash(f'{index}'
                                             f'{previoushash}'
                                             f'{timestamp}'
                                             f'{data}'
                                             f'{self.nonce}')

            if int.from_bytes(hashobject.digest()[0:target], byteorder='big') != 0:
                self.nonce += 1
            else:
                self.hash = hashobject.hexdigest()
                break

    def calculate_hash(self, blockdata):
        message = hashlib.sha256()
        message.update(blockdata.encode())
        return message


class Blockchain:
    def __init__(self, target):
        self.chain = []
        self.target = target

        self.chain.append(Block(0, time.time(), 'First block of the chain!',
                                'There is no previous hash', 1))

        print('Initialized chain!')

    def add_block(self, data):
        self.chain.append(Block(len(self.chain), time.time(), data, self.chain[-1].hash,
                                self.difficulty))


```

```
49 blockchain = Blockchain(2)
50
51 while True:
52     blockchain.add_block(input(f'difficulty: {blockchain.difficulty}. data to add to chain:'))
53     blockchain.difficulty += 1
```