Blockchain

Richard van Dijk

2 november 2017

Inhoud

1 Voorwoord

In het kader van het onderdeel 'Rapporteren en presenteren' tijdens het opleidingstraject bij IT-Vitae kregen wij de opdracht een verslag te schrijven over een technisch onderwerp. Het onderwerp dat ik voor dit verslag gekozen heb is Blockchain. Omdat het mij interessant leek om blockchain zelf in de praktijk te brengen, heb ik tevens gekozen om naast de technische achtergrond een implementatie in Python te schrijven.

2 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.

3 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. In 1993 werd een concept beschreven waarbij computers om een bepaalde actie uit te voeren eerst een bepaalde taak op moesten lossen, een systeem dat later de naam Proof-of-Work kreeg. In 1998 werden de eerste cryptocurrencies beschreven. Onderzoeker Nick Szabo ontwierp een gedecentraliseerd cryptocurrencysysteem gebaseerd op Proof-of-Work, genaamd Bit Gold. Het ontwerp van Bit Gold lijkt veel op het ontwerp van Bitcoin, dat in 2009 beschreven werd door een persoon met het pseudoniem Satoshi Nakamoto. Sinds 2009 zijn er vele honderden cryptocurrency gemaakt die gebaseerd zijn op dit ontwerp, dat gebaseerd is op blockchain met Proof-of-Work.

4 Toepassingen van blockchain

Het hoofdgebied waar blockchain wordt toegepast zijn cryptocurrency. Door de aard van blockchain (moeilijk aan te passen, eenvoudig na te lopen) is het perfect voor het opslaan van bijvoorbeeld transactiegegevens. Ook banken doen onderzoek naar hoe zij blockchain kunnen toepassen in hoe hun systemen werken. Andere toepassingen die overwogen worden zijn onder andere e-voting en digitale eigendomsaktes. Cryptocurrency zoals Ethereum gebaseerd op blockchain voegen functionaliteit zoals cloud computing toe bovenop de bestaande digitale valuta. De meeste van deze technologieën staan nog in de kinderschoenen, maar bieden wel perspectief op de brede mogelijkheden van blockchain.

5 Technische achtergrond

5.1 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 van een datum voorzien 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.

5.2 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.

5.3 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.

6 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 het 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**.

De code voor het minen van het nieuwe block staat hieronder beschreven. In dit fragment wordt de functie `calculate_hash` (L27) gedefinieerd, die op basis van een string met de gegevens van het block een SHA-256 hashobject terugstuurt. Vervolgens kan de hash in stringvorm opgevraagd worden door middel van de method `.hexdigest()`. De bytes van de hash kunnen als bytestring opgevraagd worden door de method `.digest()` aan te roepen. Van deze bytestring nemen we een slice om zoveel bytes op te vragen als benodigd door het target. Deze bytes vertalen we met de functie `int.from_bytes()` in een getal. Als alle bytes 0 zijn, is aan de voorwaarde voor het target voldaan, en kunnen we de `hexdigest()` opslaan in het block. Is dit niet zo, dan hogen we de noncewaarde met 1 op (L22) en begint de loop opnieuw.

```
        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


```

Met deze code is de opbouw van het block-object compleet. Vervolgens wordt de class Blockchain als volgt gedefiniëerd:

```
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', self.target))
        print('Initialized chain!')

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

Het object Blockchain bevat dus de chain, het target, en een method om een block aan de chain toe te voegen. Ook wordt in de constructor van Blockchain het genesisblock aan de chain toegevoegd.

Nu we een Blockchain- en Block-object hebben gedefinieerd, is het mogelijk om een basis-frontend te maken. We doen dit met behulp van de `input()`-functie van Python. Om te demonstreren dat de tijd voor het minen van een block toeneemt met een steeds hoger target, verhogen we het target bij ieder block met 1.

```
blockchain = Blockchain(1)

while True:
    blockchain.add_block(input('Data to add to chain:'))
    blockchain.target += 1
```

De code die hier beschreven staat kan eenvoudig worden uitgebreid met meer functies. Een goede toevoeging zou het valideren van de chain zijn. Een andere plek waar ruimte voor verbetering is is het target: een granulariteit van 1 byte lijkt erg laag te zijn omdat de tijd die 1, 2 en 3 bytes benodigen om te berekenen snel heel hoog wordt. Een goede oplossing zou zijn om in plaats van bytes bits te gebruiken. We hebben echter om dit script eenvoudig te houden ervoor gekozen om dit niet te doen.

7 Samenvatting

In dit verslag beschrijven we de technische achtergrond van blockchain en mogelijke toepassingen daarvan. Een grote toepassing die al wijdverspreid is zijn cryptocurrency, met Bitcoin als één van de grootste spelers. Gebaseerd op de kennis van de technische achtergrond van blockchain demonstreren we een proof-of-concept geschreven in Python, waarin we een blockchain kunnen initialiseren en blocks aan de chain kunnen toevoegen. Enkele mogelijke verbeteringspunten voor het Python-programma zijn het toevoegen van validatie van de blockchain en fijnere controle over het target voor de hash.

8 Bronvermelding

Haber, Stornetta (1990): How to time-stamp a digital document. `https://www.anf.es/pdf/Haber_Stornetta.pdf`
Satoshi Nakamoto (2009): Bitcoin: A peer-to-peer Electronic Cash System `https://bitcoin.org/bitcoin.pdf`
Wei Dai (1998): bmoney.txt `http://www.weidai.com/bmoney.txt`
Cryptomining Blog (2015): What hashrate to expect from an up-to-date CPU. `http://cryptomining-blog.com/4456-what-hashrate-to-expect-from-an-up-to-date-cpu/`

9 Appendices

9.1 Appendix I: GitHub-repository

Om de code in dit artikel te downloaden, te forken, voor pull requests of om issues door te geven, verwijzen wij naar onze GitHub-repository:
`https://github.com/aeTIos/python_blockchain`

9.2 Appendix II: Licentie

Copyright (C) 2017, Richard van Dijk

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.