



Faculteit Bedrijf en Organisatie

Gedecentraliseerd en transparant versiebeheer aan de hand van blockchain principes en IPFS: een praktische toepassing voor open source bedrijven.

Michiel Schoofs

Scriptie voorgedragen tot het bekomen van de graad van  
professionele bachelor in de toegepaste informatica

Promotor:  
Karine Samyn  
Co-promotor:  
Maurice Dalderup

Instelling: —

Academiejaar: 2019-2020

Tweede examenperiode



Faculteit Bedrijf en Organisatie

Gedecentraliseerd en transparant versiebeheer aan de hand van blockchain principes en IPFS: een praktische toepassing voor open source bedrijven.

Michiel Schoofs

Scriptie voorgedragen tot het bekomen van de graad van  
professionele bachelor in de toegepaste informatica

Promotor:  
Karine Samyn  
Co-promotor:  
Maurice Dalderup

Instelling: —

Academiejaar: 2019-2020

Tweede examenperiode



# Woord vooraf

Ik wil graag de volgende personen bedanken:

1. Karine Samyn voor de begeleiding, constructieve feedback en vooral haar geduld.
2. Maurice Dalderup voor de begeleiding en de rol van co-promotor op te nemen.
3. Ria Schoofs voor het kritisch nalezen.
4. Fatih Mercan for being a good friend and helping me pull through. Hopefully we'll meet one day outside of cyberspace.



# Samenvatting

Software is onderworpen aan constante veranderingen. Zo moeten er nieuwe functies voorzien worden en fouten opgelost. Dit is zeker moeilijk als er verschillende ontwikkelaars moeten samenwerken aan één project. Om dit te vergemakkelijken steunen software ontwikkelaars op versiebeheersystemen. Deze systemen hebben echter één groot nadeel. Ze vereisen een centrale server om bestanden en versies op te slaan. Dit is problematisch doordat een server offline of beschadigd kan worden. Hierdoor kunnen hele projecten verloren gaan. Deze bachelorproef stelt een werkwijze voor om versiebeheer te decentraliseren. Dit houdt in dat er geen centrale server meer is om de bestanden en versies op te slaan. In plaats van een centrale server wordt er geopteerd voor een netwerk van computers. Daardoor staan de verschillende versies verspreid over heel het netwerk en is er geen centrale plaats waar het fout kan gaan.

Om deze oplossing mogelijk te maken wordt er gebruik gemaakt van blockchain technologie en IPFS -, een gedecentraliseerd file-sharing protocol-. Deze begrippen worden eerst gekaderd in een literatuurstudie. Vervolgens wordt aan de hand van een prototype een werkbaar gedecentraliseerd versiebeheersysteem geïmplementeerd. Dit prototype houdt daarbij rekening met enkele succescriteria

De oplossing die ontwikkeld werd in deze bachelorproef voldoet aan alle functionele vereisten van een versiebeheersysteem. Deze bachelorproef is aldus geslaagd in haar opzet. Toch is het geen gangbaar alternatief voor populaire versiebeheersystemen zoals GitHub. Dit komt doordat de gebruikte oplossing gebouwd is bovenop het Ethereum netwerk. Dit netwerk vereist transactiekosten voor elke handeling die data wegschrijft op de blockchain. Het prototype dat ontwikkeld werd is dus niet kostenefficiënt. Zeker

als men in beschouwing neemt dat meeste versiebeheersystemen gratis zijn, lijkt het niet plausibel dat mensen zouden overstappen op deze oplossing.

Voordat dit systeem succesvol kan zijn moet men een manier vinden om deze transactiekosten te vermijden. Dit is zeker mogelijk aangezien er al netwerken zijn die geen transactiekosten vereisen. Een voorbeeld van zo een netwerk is het Loom netwerk. Een vervolgstudie zou dus kunnen zijn om het prototype dat beschreven staat in de POC na te bouwen op één van deze netwerken.



# Inhoudsopgave

<b>1</b>	<b>Inleiding .....</b>	<b>15</b>
1.1	Introductie versiebeheersystemen	15
1.2	Probleemstelling	16
1.3	Introductie decentralisatie	16
1.4	Onderzoeksvraag	17
1.5	Onderzoeksdoelstelling	17
1.6	Opzet van deze bachelorproef	18
<b>2</b>	<b>Stand van zaken .....</b>	<b>19</b>
2.1	Versiebeheer	19
2.1.1	Inleiding: De drie types van versiebeheersystemen .....	19
2.1.2	RCS: één van de eerste lokale versiebeheersystemen. ....	21
2.1.3	Branches .....	25

2.1.4	Conclusie .....	27
<b>2.2</b>	<b>Het omspringen met bestanden in een gedecentraliseerd systeem</b>	<b>28</b>
2.2.1	Duiding van dit hoofdstuk .....	28
2.2.2	Algemene begrippen en concepten .....	29
2.2.3	BitTorrent .....	31
2.2.4	IPFS .....	36
<b>2.3</b>	<b>Blockchain principles</b>	<b>37</b>
2.3.1	Inleiding .....	37
2.3.2	Basisprincipes van een blockchain .....	38
2.3.3	Blockchain mining .....	42
2.3.4	Ethereum .....	45
2.3.5	Smart contracts .....	45
<b>3</b>	<b>Methodologie .....</b>	<b>47</b>
<b>3.1</b>	<b>Hoe de onderzoeksvraag tot stand is gekomen.</b>	<b>47</b>
<b>3.2</b>	<b>Het ontleden van de onderzoeksvraag.</b>	<b>48</b>
<b>3.3</b>	<b>Tools</b>	<b>50</b>
<b>3.4</b>	<b>Meetbare criteria</b>	<b>51</b>
<b>3.5</b>	<b>Werkwijze</b>	<b>53</b>
3.5.1	Opzet .....	53
3.5.2	Repositories .....	54
3.5.3	Het opslaan van de bestanden op een gedecentraliseerde manier ..	54
3.5.4	Het opvragen van bestanden op een gedecentraliseerde manier ...	55
3.5.5	Het aanbrengen van wijzigingen .....	55
3.5.6	Het binnenhalen van wijzigingen .....	56

3.5.7	Het teruggaan naar een eerdere versie .....	56
-------	---	----

<b>4</b>	<b>Conclusie .....</b>	<b>57</b>
----------	------------------------	-----------

4.1	Verder onderzoek	58
-----	------------------	----

<b>A</b>	<b>Appendix .....</b>	<b>59</b>
----------	-----------------------	-----------

A.1	Introductie	59
-----	-------------	----

A.2	State-of-the-art	61
-----	------------------	----

A.3	Methodologie	62
-----	--------------	----

A.4	Verwachte resultaten	62
-----	----------------------	----

A.5	Verwachte conclusies	63
-----	----------------------	----

A.1	Duiding	64
-----	---------	----

A.2	Opzet van het project	64
-----	-----------------------	----

A.3	Nieuwe versie van Alice	66
-----	-------------------------	----

A.1	Basis syntax van de taal	67
-----	--------------------------	----

A.2	Overerving	69
-----	------------	----

A.3	Library voorbeeld	70
-----	-------------------	----

A.4	geavanceerde concepten	70
-----	------------------------	----

A.5	Nieuwe concepten vanaf 0.6.0	72
-----	------------------------------	----

<b>Bibliografie .....</b>	<b>73</b>
---------------------------	-----------



## Lijst van figuren

2.1	Overzicht types VCS .....	21
2.2	Overzicht concepten boomstructuur .....	22
2.3	Voorbeeld van deltas. ....	23
2.4	Voorbeeld merge proces. ....	25
2.5	Voorbeeld flow .....	26
2.6	Peer meta-informatie stap - Torrenting 1 .....	32
2.7	THP stap - Torrenting 2 .....	33
2.8	PWP stap - Torrenting 3 .....	34
2.9	Blockchain - Voorstelling 1 .....	38
2.10	Blockchain - Voorstelling 2 .....	39
2.11	Blockchain - Voorstelling 3 .....	41
2.12	Blockchain - Voorstelling 4 .....	42
2.13	Blockchain - Voorstelling 5 .....	43
3.1	POC opstart .....	54
3.2	POC Repository .....	55



## Lijst van tabellen

2.1	Concepten binnen versiebeheersystemen. ....	28
2.2	Vergelijking tussen gecentraliseerd - gedecentraliseerd systeem .	30
3.1	Succescriteria van de POC. ....	53





# 1. Inleiding

## 1.1 Introductie versiebeheersystemen

Elke software ontwikkelaar kan beamen dat software fel onderhevig is aan veranderingen. Constant moeten er nieuwe functies worden geïmplementeerd en fouten worden rechtgezet. Dit probleem wordt enkel verergerd indien er tientallen mensen samen aan projecten werken. Versiebeheersystemen bieden een oplossing voor deze problemen. Tegenwoordig zijn ze een onmisbaar deel geworden van de manier waarop software wordt ontwikkeld.

Deze systemen delen software op in een aaneenschakeling van veranderingen. Deze veranderingen worden ook wel versies genoemd. Deze systemen maken het mogelijk om nieuwe wijzigingen aan te brengen en toegang te krijgen tot oudere versies. Moderne versiebeheersystemen bieden daarbovenop ook de mogelijkheid om twee conflicterende wijzigingen samen te voegen. Hierdoor wordt samenwerken aan grote projecten vergemakkelijkt.

Kort samengevat bestaat een versiebeheersysteem uit twee verschillende functies:

1. Een plaats waar bestanden kunnen worden opgeslagen en opgevraagd worden.
2. Een manier om wijzigingen aan te brengen en toegang te krijgen tot eerdere versies van deze bestanden.

Versiebeheersystemen stellen software ontwikkelaars in staat om code publiek toegankelijk te maken. Hierdoor kan elke software ontwikkelaar de broncode raadplegen en aanpassen: een principe dat mooi kadert binnen de zogenaamde Open-Source filosofie. Deze filosofie streeft naar een open en democratische ontwikkeling van software. Hierbij staat

samenwerken aan verbeteringen centraal. Software projecten worden aldus aanzien als een gezamenlijk project.

## 1.2 Probleemstelling

Toch is er een probleem met de manier waarop deze systemen functioneren. Deze systemen gebruiken een centrale server om bestanden en wijzigingen op te slaan. Dit is een probleem aangezien deze servers eventueel beschadigd of onbereikbaar kunnen zijn. Hierdoor kunnen projecten onbeschikbaar zijn of in het ergste geval zelfs verloren gaan. Dit probleem staat ook wel bekend onder de naam **Single point of failure**.

Een ander probleem is dat deze servers in de handen kunnen komen van grote tech-giganten. Zo heeft Microsoft in 2018 GitHub -een populair versiebeheer platform- overgekocht. Dit druist in tegen de eerder vermelde Open-Source filosofie, want deze systemen zijn niet gevrijwaard van commerciële invloeden.

Binnen deze bachelorproef wordt een systeem ontwikkeld dat geen centrale server vereist. Op die manier wordt het Single point of failure probleem vermeden en is de oplossing vrij van commerciële invloeden. Dit maakt de oplossing interessant voor Open-Source software bedrijven.

## 1.3 Introductie decentralisatie

Een alternatief voor deze centrale server bestaat onder de vorm van zogenaamde P2P protocollen. Deze protocollen hebben als doelstelling om de rol van centrale server op zich te nemen en de taak te verspreiden over een netwerk van computers. Het afstappen van een centrale server naar een netwerk van verschillende computers wordt ook wel decentralisatie genoemd.

De doelstelling van deze bachelorproef is om versiebeheersystemen te decentraliseren. De oplossing die hierbij wordt voorgesteld moet aldus voldoen aan de volgende drie criteria:

1. De oplossing moet volledig gedecentraliseerd zijn. Dit wilt zeggen dat er geen centrale server of andere component aanwezig mag zijn. Dit komt dus in essentie neer op het gebruik maken van P2P protocollen.
2. Er moet een manier zijn om bestanden te delen en op te vragen.
3. Er moet een mogelijkheid zijn tot het aanbrengen van wijzigingen en het bijhouden van versies.

Voor het bijhouden en opvragen van bestanden kan gebruik worden gemaakt van **IPFS**. Dit is een P2P file-sharing protocol. De wijzigingen en versies moeten worden bijgehouden in een dataopslagsysteem. Een gekend gedecentraliseerd opslagsysteem is blockchain.

## 1.4 Onderzoeksvraag

De onderzoeksvraag luidt dus als volgt: **hoe kan men door middel van Blockchain principes en IPFS een werkbaar gedecentraliseerd versiebeheersysteem ontwikkelen?**

Om deze onderzoeksvraag correct en volledig te beantwoorden wordt er ook met een aantal deelvragen gewerkt. Deze deelvragen worden sequentieel behandeld in de literatuurstudie:

1. Wat zijn de problemen die versiebeheersystemen oplossen?
2. Waaruit bestaat een versiebeheersysteem?
3. Waarom zouden we van gecentraliseerde (server-client architectuur) versiebeheersystemen overstappen naar een gedecentraliseerde variant?
4. Wat zijn de eigenschappen en valkuilen van gedecentraliseerde netwerken?
5. Wat is IPFS en hoe kadert het binnen versiebeheer?
6. Wat zijn smartcontracts en hoe bieden ze een meerwaarde aan Blockchain applicaties?

## 1.5 Onderzoeksdoelstelling

Zoals in de onderzoeksvraag wordt vermeld is de doelstelling om een gedecentraliseerd versiebeheer systeem te ontwikkelen. Er zijn dus drie criteria waar deze oplossing aan moet voldoen:

1. De oplossing moet volledig gedecentraliseerd zijn. Dit wilt zeggen dat er geen centrale server of andere component aanwezig mag zijn.
2. Er moet een manier zijn om bestanden te delen en op te vragen.
3. Er moet een mogelijkheid zijn tot het aanbrengen van wijzigingen en het bijhouden van versies.

Op het einde van deze bachelorproef is de doelstelling dus om een werkend prototype te hebben. Dit prototype zal gebruik maken van IPFS en blockchain technologie om deze

doelstelling te bereiken.

## 1.6 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie. Dit is onderverdeeld in drie verschillende delen:

1. Een literatuurstudie over versiebeheersystemen. In dit deel worden de concepten binnen versiebeheer gekaderd en verduidelijkt. Zo krijgt men een beeld van wat er reeds beschikbaar is, alsook aan welke vereisten het ontwikkelde prototype moet voldoen.
2. Een studie over file-sharing protocollen. IPFS is een complex file-sharing protocol. Hier worden de concepten uitgelegd om voldoende achtergrondinformatie te verschaffen om de achterliggende principes van het ontwikkelde prototype te begrijpen.
3. Een studie over de achterliggende principes van blockchain protocollen. Hierbij wordt uitgelegd wat blockchain is en hoe het zal verwerkt worden binnen het ontwikkelde proto-type.

In hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

## 2. Stand van zaken

### 2.1 Versiebeheer

#### 2.1.1 Inleiding: De drie types van versiebeheersystemen

Versiebeheer is een belangrijk concept binnen softwareontwikkeling. Zo waren er in 2018 in het totaal 100 miljoen projecten op het populaire versiebeheer platform GitHub (Warner, 2018). GitHub (sinds 2018 overgenomen door Microsoft) is niet de enige speler op de markt. Er zijn ook nog Code Commit van Amazon en GitLab. Veel bedrijven spelen dus in op de behoefte aan een duidelijk en efficiënt versiebeheersysteem. De vraag stelt zich dan ook: welke behoefte lossen deze systemen op?

Neem volgende scenario: Alice en Bob zijn aangenomen om te werken voor bedrijf X. Hun eerste taak is een website ontwikkelen. Ze leggen samen alle vereisten vast, bespreken de verschillende technologieën en gaan aan de slag. Op het einde van de eerste dag hebben ze elk een verschillende pagina gemaakt die ze graag met elkaar delen willen delen. Dit kan bijvoorbeeld door via mail de bestanden door te sturen. Een andere mogelijkheid is de bestanden via fysieke hardware zoals een USB-Stick aan elkaar te geven. Het nadeel is dat de code op twee verschillende plaatsen verspreid zit. Als Bob de code kwijt raakt, dan zal deze opnieuw moet worden geschreven. Om dit probleem te voorkomen kan men het project op een centrale server opslaan. Bob en Alice zullen hun veranderingen opslaan op deze centrale server. Zo hebben ze altijd toegang tot elkaars werk.

Deze manier van werken heeft nadelen. Alice kan per ongeluk een bestand overschrijven of een stuk code verwijderen. Tenzij er back-ups zijn, is het originele bestand verloren. Om dit probleem te omzeilen, wordt er gebruik gemaakt van het concept van **versies**. Elke aanpassing die er gemaakt wordt resulteert in een nieuwe versie van het project. Men kan

altijd terugkeren naar een eerdere versie. Als Alice dus het stukje code verwijdert in versie 15 kan men terug naar versie 14.

Loeliger (2012) stelt dat een versiebeheersysteem een middel is om verschillende versies van code te beheren en bij te houden. De auteur onderscheidt volgende drie eigenschappen waaraan dergelijke systemen voldoen:

- er wordt gebruik gemaakt van een centraal archief. Binnen dit archief worden alle versies van het project bewaard en bijgewerkt.
- het centraal archief geeft toegang tot eerdere versies van het project.
- alle veranderingen die worden aangebracht aan het archief worden genoteerd in een centraal logboek.

Versiebeheer is geen nieuw concept. Er zijn zoals eerder aangehaald verschillende software oplossingen beschikbaar. Volgens Chacon en Straub (2014) zijn er drie grote categorieën (zie 2.1 voor een grafische weergave):

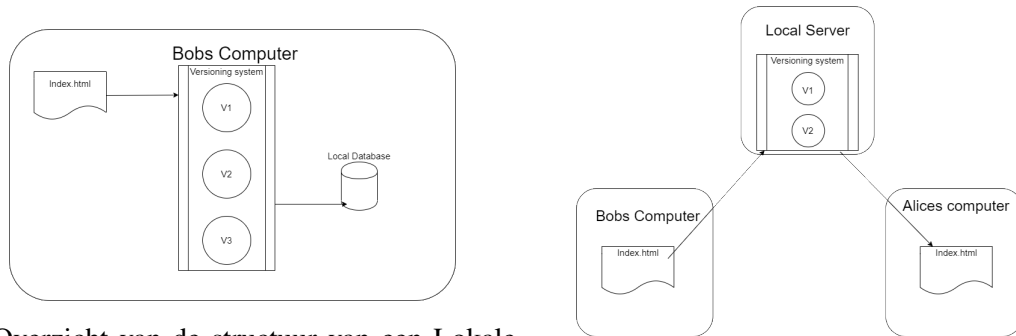
- lokale versiebeheersystemen: het centraal archief waar de veranderingen in worden bewaard, staat op een lokale computer. Het grootste voordeel is dat een lokaal systeem zeer makkelijk te onderhouden is en eenvoudig op te stellen. Toch is het niet geschikt om bestanden met elkaar te delen of samen aan bestanden te werken. Een gekend voorbeeld is RCS (Revision Control System) - zie 2.1.2 -.
- CVCS: om samen te kunnen werken aan dezelfde bestanden kan een CVCS (Centralised Version Control System) worden gebruikt. In plaats van het archief lokaal bij te houden wordt er gebruik gemaakt van een centrale server. Bestanden worden vervolgens lokaal gekopieerd. Als er veranderingen worden aangebracht zullen deze worden doorgestuurd naar de server. Doordat men verplicht is om de bestanden op een centrale plaats af te halen, kan men deze afschermen. Zo kan men de toegang beperken tot enkel de nodige bestanden per gebruiker.

Een neveneffect van alles centraal te beheren is het zogenaamde *single point of failure (SPOF)* probleem. Een SPOF is een onderdeel van een systeem dat mocht het uitvallen heel het systeem tot een halt roept. Met andere woorden: valt het centraal archief weg heeft niemand nog toegang tot het project. Een mogelijke oplossing voor dit probleem is redundantie. Dit betekent het aanbieden van kopieën. (Microsystems, 2007)

- DVCS: om het SPOF probleem te voorkomen kan men kopieën maken van het centraal archief. Deze kopieën kunnen vervolgens worden verspreid over verschillende computers. Dit is het uitgangspunt van DVCS (Distributed version control System). Elke gebruiker heeft een lokale kopie van de centrale server. De veranderingen aan de bestanden worden eerst aangebracht in het lokaal archief en vervolgens gesynchroniseerd met de centrale variant.

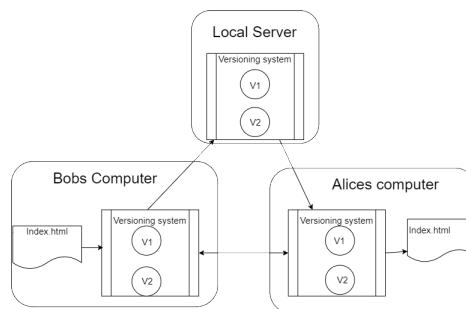
Mocht het centraal aanspreekpunt niet beschikbaar zijn is dit geen probleem. Elke gebruiker heeft immers een volledige back-up van het volledige project. In theorie

kan de gebruiker zelfs optreden als nieuwe centrale server.



(a) Overzicht van de structuur van een Lokale VCS.

(b) Overzicht van de structuur van een CVCS.



(c) Overzicht van de structuur van een DVCS.

Figuur 2.1: Overzicht van de drie types van VCS zoals aangegeven door Chacon en Straub (2014).

### 2.1.2 RCS: één van de eerste lokale versiebeheersystemen.

RCS (*Revision Control System*) is een lokaal versiebeheer systeem. Het wordt voor het eerst beschreven in een artikel geschreven door Tichy (1985). Het werd verder ontwikkeld binnen het GNU project -Een open source besturingssysteem<sup>1</sup>- waar het als vervanging voor het CSSC System (Youngman, 2016) werd gebruikt. CSSC is een systeem gebaseerd op SCCS (Source code control system) dat ontworpen is voor UNIX systemen. SCCS is in opdracht van Bell Labs ontwikkeld door Rochkind (1975).

Voordat RCS op de markt kwam is er nog tal van andere software ontwikkeld. Zo was er CA-Panvalet, een gepatenteerde oplossing voor Mainframe computers.

Waarom is het interessanter om RCS in detail te bekijken? Veel van de concepten waar het gebruik van maakt zijn aanwezig in moderne systemen (zoals GIT). Het is open source en wordt nog steeds op vrijwillige basis onderhouden, wat aansluit bij de visie van deze

<sup>1</sup>GNU is veel meer dan enkel open source. Het GNU project is sterk verbonden met de ideologie en organisatie van de free software foundation (FSF). Meer informatie omtrent deze organisatie en beweging is te vinden op: <https://www.fsf.org/>





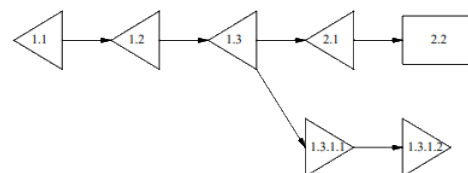
de versies weer te geven. Stel: Bob en Alice zijn bezig aan de hoofdpagina. Bob maakt een initiële versie van de pagina. Vervolgens wilt hij die graag delen met Alice. Hiervoor gebruikt hij het commando om een nieuwe versie aan te maken: `ci homepage.html`. Dit commando wordt **inchecken** genoemd. Binnen GIT is dit vergelijkbaar met het commando *git push*. Aangezien dit de eerste versie is kan men dit vergelijken met het aanmaken van een wortel. Volgende versies worden kinderen van de vorige versie. Zo wordt versie 1.4 kind van versie 1.3. Inchecken gaat niet alleen onze boomstructuur aanmaken maar ook de extensie `.v` toevoegen (`homepage.html.v`). Het originele bestand wordt ook verwijderd

Het bestand krijgt ook een **versienummer**. Dit versienummer heeft de vorm:  $x_1.x_2$ .  $x_1$  (ook wel *release* genoemd) staat voor een grote verandering. Bijvoorbeeld het in productie nemen van een nieuwe versie.  $x_2$  (*level*) staat voor een kleinere verandering. Een andere manier om  $x_2$  te bekijken is de diepte met als wortel de laatste release ( $x_1$ ). 1.1 is het versienummer van de wortel die Bob heeft aangemaakt. 1.4 is het derde kind van de wortel. Elke check-in zal het level ( $x_2$ ) met één verhogen. Het release nummer ( $x_1$ ) wordt manueel verhoogd door middel van de `-r` optie bij check-in<sup>2</sup>. Bij branches is er ook nog sprake van  $x_3$  en  $x_4$  zie 2.1.3.

Deze manier van versies te bestempelen wordt nog steeds gebruikt. Het is echter niet de enige manier. *Semantic versioning* is een gekend alternatief. Het concept van versienummers bestaat in Git onder de vorm van *tags*.

Er is nu een bestand onder de vorm `homepage.html.v`. Hoe kan Alice nu dit bestand aanpassen en een nieuwe versie publiceren? Alice zal het bestand moeten **uitchecken**. Dit kan ze doen door middel van het commando `co` en de naam van het bestand<sup>3</sup>. Het uitchecken is aldus het verkrijgen van een specifieke versie uit het archief. Als er geen specifieke versie wordt meegegeven wordt de laatste versie opgehaald. Om een specifieke versie op te halen kan er gebruik gemaakt worden van de optie `-r`<sup>4</sup>. Alice heeft nu een kopie van het originele bestand gekregen. Merk op dat in tegenstelling tot inchecken ons archief niet wordt verwijderd. Vervolgens kan ze in deze lokale kopie wijzigingen aanbrengen. Tot slot wordt het bestand weer aan het lokaal archief toegevoegd door middel van inchecken. Het equivalent van `co` binnen git is `git pull`.

Hoe worden de verschillen tussen de versies bijgehouden in ons lokaal archief? Een mogelijke oplossing zou zijn om alle versies van het bestand afzonderlijk bij te houden. Dit vraagt veel opslagruimte. RCS gebruikt voor dit probleem het concept van **deltas**. Een delta houdt bij welke regels



Figuur 2.3: Een voorbeeld van deltas. De Trunk bevat een series van achterwaartse deltas terwijl alle branches enkel voorwaardse deltas bevatten. Grafiek afkomstig uit Tichy (1985)

<sup>2</sup>`ci -r2 homepage.html`

<sup>3</sup>`(co homepage.html)`

<sup>4</sup>`(co -r1.1 homepage.html)`

veranderd zijn ten opzichte van de vorige versie. Doordat de delta enkel de relevante lijnen bijhoudt wordt de opslag beperkt <sup>5</sup>. Er zijn twee types van deltas: **voorwaartse deltas** en **achterwaartse deltas**. Bij het inchecken van een nieuwe versie zal de vorige versie worden vervangen door een achterwaartse delta. Zit men momenteel op versie 1.3 en vraagt men versie 1.2 dan zal de achterwaartse delta van versie 1.2 worden toegepast op versie 1.3. (Voorwaartse deltas komen aan bod in het gedeelte over branching (2.1.3). Het concept van deltas wordt nog eens verduidelijkt door een voorbeeld in de appendix -zie A.5-.)

Inchecken en uitchecken ligt aan de basis van het archiefsysteem. Toch is er nog een probleem aanwezig met deze manier van werken. Stel dat Alice en Bob gelijktijdig wijzigingen aanbrengen aan een bestand. Ze willen dit bestand elk afzonderlijk publiceren. Hierdoor ontstaan er twee versies die afstammen van één gezamenlijke versie. De boomstructuur wordt in twee gesplitst. Dit is niet mogelijk aangezien een versie altijd uniek moet zijn. Hoe kan men verzekeren dat elke versie slechts één kind heeft (op dezelfde branch)? Dit probleem wordt opgelost door **sloten**(engels=lock). Dit concept geeft gebruikers de mogelijkheid om een versie te versleutelen. Terwijl een versie versleuteld is kan niemand anders wijzigingen aanbrengen. Andere gebruikers kunnen deze nog bekijken. Op het moment dat Bob zijn versie gaat uitchecken kan hij deze versleutelen (door middel van de *-l* optie bij het *co* commando). Hierdoor kan Alice geen nieuwe versie meer aanmaken tot Bob zijn wijzigingen heeft doorgevoerd. Met andere woorden zolang Bob het slot niet vrijgeeft kunnen er geen nieuwe versies worden aangemaakt<sup>6</sup>. Deze manier van werken heeft een zichtbaar nadeel. Alice is verplicht om te wachten op Bobs nieuwe versie alvorens ze veranderingen kan aanbrengen. Git gebruikt het concept van sloten niet. Daar maakt men gebruik van **merges** om dit zelfde probleem aan te pakken

## Opmerkingen

In het originele artikel wordt de klemtoon gelegd op het onderling delen van de verschillende versies. Hierdoor kan men de indruk krijgen dat er een centrale server betrokken is. Dit is niet het geval. De software is ontworpen om op één besturingssysteem uitgevoerd te worden. Volgens team (2020) is GNU aangezien het gebaseerd is op UNIX een *multi-user os*. Dat wil zeggen dat meerdere gebruikers het systeem terzelfdertijd kunnen gebruiken door middel van een terminal connectie. Hierdoor kan men onderling de bestanden delen ondanks dat men niet gaat werken in een CVCS.

---

<sup>5</sup>De delta wordt opgebouwd aan de hand van het GNU commando *diff* <https://www.gnu.org/software/diffutils/>

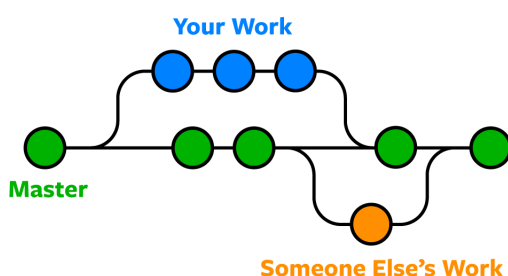
<sup>6</sup>In sommige gevallen kan het slot ook worden 'geforceerd' mocht Bob bijvoorbeeld ziek vallen

### 2.1.3 Branches

#### Duiding

Door het principe van sloten en inchecken lijkt het alsof elke versie exact één opvolger heeft. Zo zal versie 1.3 de opvolger zijn van 1.2. Toch kunnen er zich zoals in een stamboom vertakkingen voordoen. De vertakkingen worden **Branches** genoemd. De hoofdboom (niet vertakte toppen die afstammen van de wortel) wordt de **trunk** genoemd. Dit principe kan men het best demonstreren aan de hand van een figuur. Zo kan men zien in figuur 2.2 dat er twee vertakkingen zijn op versie 1.2. Het principe van vertakkingen lijkt op het eerste zicht complex en onoverzichtelijk. Tichy (1985) geeft enkele redenen om dit toch toe te passen.

1. Doorvoeren van veranderingen in oude versies: Stel dat een bedrijf een oude versie van een software product gebruikt. Dit product is ontwikkeld met behulp van RCS. Er wordt een fout in deze oude versie gevonden die om een oplossing vraagt. Aangezien er in de tussentijd nieuwere versies zijn is dit niet evident. Het bedrijf zou volledig moeten overstappen op de nieuwste versie alvorens een aanpassing kan gebeuren. Om deze situatie te vermijden kan men gebruik maken van vertakkingen. Zo kan men een vertakking maken op de gewenste versie en kleine aanpassingen doorvoeren.
2. Andere implementaties: stel dat een ontwikkelaar een nieuw stuk code wilt uittesten. Deze heeft niet het gewenste resultaat. Mocht dit stuk code bewaard worden op de hoofdboom (*trunk*) dan is er een onstabiele versie gepubliceerd. Een gebruiker die op dat moment de recentste versie opvraagt, krijgt dus een niet werkend product. Door branches te gebruiken kan men ervoor zorgen dat er enkel werkende versie op de hoofdboom terecht komen. Nieuwe stukken code worden eerst geïsoleerd en getest alvorens opgenomen te worden. Op die manier blijft het archief in een overzichtelijke en stabiele vorm.

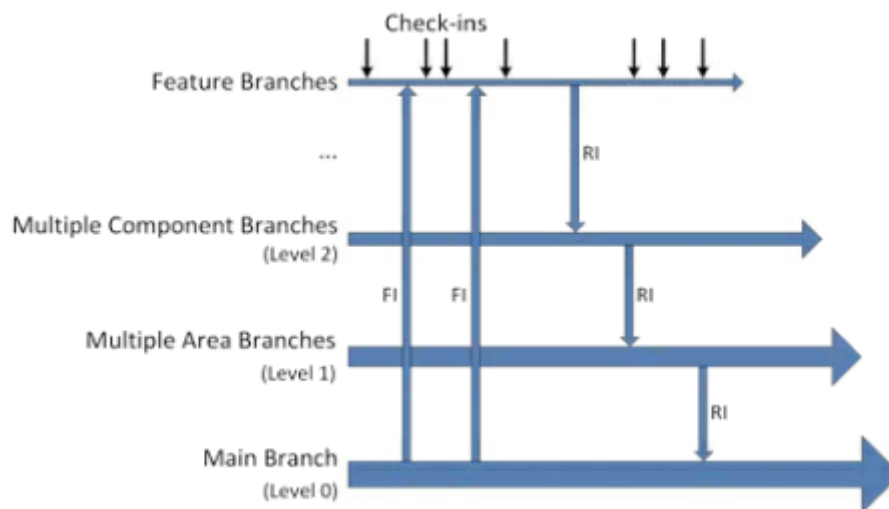


Door middel van vertakkingen kan code worden geschreven en de hoofdboom stabiel gehouden. Zelfs met veel ontwikkelaars en grote projecten kan men door deze manier van werken code conflicten vermijden. Eenmaal de code klaar is voor productie kan men deze gaan publiceren op de hoofdboom. Dit concept wordt ook wel **mergen** genoemd. Dit principe wordt geïllustreerd door de figuur 2.4.

Figuur 2.4: Een voorbeeld van een merge proces. Eerst wordt er een branch aangemaakt die na drie versies terug in de master wordt gemerged. Grafiek gepubliceerd door Desktop (2018)

#### Flows

Er is echter nog een grote vraag die niet beantwoord is: wanneer moet men gaan vertakken?



Figuur 2.5: Flow zoals vermeld in het artikel van Bird en Zimmermann (2012)

De eerste reden aangehaald door Tichy (1985) -het doorvoeren van veranderingen- is minder relevant binnen de context van DVCS. Er wordt namelijk niet meer op een gezamenlijk archief gewerkt maar op een lokale kopie. Het bedrijf kan dus op zijn eigen kopie lokaal veranderingen aanbrengen. De tweede reden is echter wel belangrijk. Stel dat een project honderden bestanden en ontwikkelaars heeft. In zo een omgeving kunnen veranderingen soms onvoorspelbare gevolgen hebben. Ontwikkelaars kunnen het project vertakken, wijzigen en testen alvorens het in productie te nemen. Dit is het principe achter **feature branches**.

Bij grote software projecten loopt men het risico dat er veel aftakkingen worden gemaakt die niet worden gebruikt (**branchmania**). Het artikel van Bird en Zimmermann (2012), benadrukt het belang van levende takken. Dit zijn vertakkingen die actief wordt gebruikt. Dit kan enkel bereikt worden via een duidelijke werkwijze. Deze werkwijze om het aantal vertakkingen zo klein mogelijk te houden wordt geïllustreerd in figuur 2.5.

Elke ontwikkelaar schrijft zijn code op een feature branch. Het risico hiervan is dat men op den duur niet meer compatibel is met de hoofdboom. Er kunnen sinds de vertakking immers verschillende andere versies gepubliceerd zijn. Daarom wordt er gewerkt met twee tussenbranches. Hier test men of de code compatibel is met de verschillende andere onderdelen van de software. Het proces waarbij men feature branches integreert in andere tussenbranches noemt men ook wel **Reverse integration** -aangeduid als RI op de figuur-. Een ander principe dat gebruikt wordt is dat van **Forward integration**. Hierbij worden de nieuwe versies van de hoofdbranch (ook wel master genoemd) geplaatst op de feature branch. Zo blijft de feature branch compatibel met alle veranderingen.

### 2.1.4 Conclusie

De verschillende types van versiebeheersystemen -zoals besproken in 2.1.1- hebben een gemeenschappelijke eigenschap. De bestanden en in veel gevallen het archief staan integraal opgeslagen op computers. Bij lokale versiebeheersystemen en CVCS staat het zelfs op één centrale computer. Dit introduceert het probleem van een SPOF (Single point of failure). DVCS vermijdt dit probleem door gebruikers kopieën te geven van het archief. Valt de centrale computer weg dan heeft elke gebruiker een back-up. Veranderingen tussen lokale versies en die op de centrale computer moet manueel gesynchroniseerd worden. Hierdoor heeft men in grote mate controle over wat er centraal wordt opgenomen. Het nadeel is dat er veel lokale kopieën zijn en veranderingen niet altijd worden gesynchroniseerd. Op deze manier heeft niet iedereen toegang tot de laatste veranderingen. Het zou dus een verbetering zijn mocht er een systeem bestaan dat één centraal archief ondersteunt zonder SPOF. Dit lijkt op eerste zicht niet mogelijk, aangezien er één centraal aanspreekpunt moet zijn. Toch is dit probleem al eerder opgelost onder de vorm van peer-to-peer (afgekort tot p2p) netwerken.

P2P is voornamelijk bekend onder de vorm van file-sharing netwerken zoals Napster. Chawathe e.a. (2003) stellen dat Napster één van de eerste systemen was die erin slaagde om een succesvol netwerk uit te bouwen. Bij dit netwerk worden files niet opgevraagd aan een centrale server. In plaats hiervan gebruikt men een netwerk van **peers**. Door de principes van dit type van netwerken toe te passen kan men een centraal archief op een gedistribueerde manier opslaan. Zo behoudt men het voordeel van CVCS zonder een SPOF.

De doelstelling van deze bachelorproef is om een werkend P2P versiebeheersysteem te implementeren. Hiervoor moet men afbakenen welke functionaliteiten deze implementatie moet voorzien. In vorige paragrafen werden de verschillende concepten besproken. Hieronder volgt een oplijsting van deze verschillende concepten alsook een korte uitleg. Hierbij worden de terminologie en concepten van Git gebruikt. Deze worden vervolgens meegenomen naar volgende hoofdstukken waar een implementatie wordt voorzien.

Concepten binnen versiebeheer	
Begrip	Uitleg
<b>Archief</b>	Een centrale plaats voor het bijhouden van bestanden. De wijzigingen van deze gearchiveerde bestanden worden bijgehouden. Men kan zowel historische als recente versies van het bestand opvragen alsook van het gehele archief.
<b>Versies</b>	Elke wijziging binnen het archief leidt tot een nieuwe versie. Men kan de verschillende versies ten alle tijden raadplegen. Men kan ook oudere versies gebruiken voor branching. In extreme gevallen kan een archief volledig worden teruggedraaid naar een eerdere versie.
<b>Logboek</b>	Een bestand waarin alle wijzigingen worden bijgehouden. Het logboek is ook onderdeel van het archief.
Pushing	Het principe waarbij wijzigingen die lokaal worden aangebracht, worden gesynchroniseerd met de centrale server.
Pulling	Het binnenhalen van veranderingen aangebracht op het centraal archief naar een lokale kopie.
Clonen Branching	Het aanmaken van een lokale kopie van een centraal archief. Het voorzien van alternatieve vertakkingen van het archief.

Tabel 2.1: Concepten binnen versiebeheersystemen.

## 2.2 Het omspringen met bestanden in een gedecentraliseerd systeem

### 2.2.1 Duiding van dit hoofdstuk

De doelstelling van deze bachelorproef is om een volledig gedecentraliseerd versiebeheersysteem uit te werken. Zoals in de vorige sectie reeds werd uitgelegd is een versiebeheersysteem een verzameling van bestanden en een logboek waarin alle veranderingen aan deze bestanden worden bijgehouden. Dit logboek is aldus een collectie van data. Hierdoor kan een logboek eenvoudig worden gedecentraliseerd aan de hand van een blockchain. **Maar ook dan blijft de vraag nog steeds waar kunnen de bestanden worden bewaard?**

In eerste instantie lijkt het eenvoudig om deze bestanden ook gewoon op een blockchain te zetten. Dit is echter niet mogelijk (hier wordt verder op ingegaan in hoofdstuk 3). Een andere eenvoudige oplossing zou zijn dat Bob gewoon rechtstreeks de bestanden deelt met Alice. Dit kan door middel van een FTP connectie of andere oplossingen. Hierdoor komen Bob en Alice echter in een klassiek server-client model terecht. De nadelen van dit model worden uitvoerig besproken in 2.2.2. Er is dus een manier nodig om bestanden te delen zonder dat men daarbij gebruik moet maken van een server. Dit principe wordt ook wel **gedecentraliseerde file-sharing** genoemd.

Dit probleem is niet nieuw. Er bestaan reeds sinds de vroege jaren 2000 verschillende protocollen die dit probleem proberen op te lossen. Doorheen dit hoofdstuk worden twee

van deze protocollen besproken: Bit-Torrent en IPFS. Waarom wordt er juist geopteerd voor deze twee? Bit-Torrent is een oud protocol dat nog steeds gebruikt wordt. Het introduceert veel van de concepten binnen file-sharing. Toch heeft het Bit-Torrent protocol enkele beperkingen. Deze beperkingen worden opgevangen binnen een meer recent protocol genaamd IPFS. IPFS wordt ook gebruikt in de POC. Deze protocollen vereisen wel enige voorkennis om te begrijpen. Daarom volgt er eerst een korte introductie van alle begrippen.

### 2.2.2 Algemene begrippen en concepten

#### Decentraliseren en P2P

Zoals in de vorige sectie uitgelegd moeten de bestanden worden opgeslagen op een gedecentraliseerd systeem. **Wat betekent het om iets te decentraliseren?** Kort gesteld is een gedecentraliseerd systeem een systeem waar geen centrale component aanwezig is.

Een gekend voorbeeld van een gecentraliseerd systeem is de manier waarop mensen surfen op het internet. Stel onderstaand scenario:

Bob is een liefhebber van Capybaras en wil informatie krijgen over de dieren. Hij surft bijgevolg naar de Wikipedia pagina. Bob zal een verzoek moeten sturen naar de server van Wikipedia. Hiervoor moet hij het publieke IP-Adres kennen. Dit kan hij verkrijgen door het DNS protocol. Bob stuurt zijn verzoek naar deze server en krijgt de gezochte pagina terug. Aangezien alle informatie zich bevindt op één plaats kan er worden gesproken van een gecentraliseerd systeem. Dit specifiek type van een centraal systeem wordt ook wel een **client-server** systeem genoemd.

Binnen een gedecentraliseerd systeem is er geen sprake meer van een centrale server die alles ter beschikking stelt. Alle computers zijn onderling met elkaar verbonden en wisselen bestanden en gegevens met elkaar uit. Dit wordt ook wel bestempeld als **Peer-to-Peer** (afgekort als P2P). Peer-to-peer is een vorm van distributed computing<sup>7</sup> waarbinnen elke computer optreedt als een Servent. Dit is een combinatie van het woord **server** en **client**. Elke computer op dit netwerk is aldus in staat om verzoeken te behandelen en te versturen. Het biedt met andere woorden een gangbaar alternatief voor het client-server model.

Volgens Schollmeier (2001) zijn er twee verschillende categorieën van Peer-to-peer netwerken:

- Pure netwerken: hierbij is elke peer gelijkwaardig. Dat wilt zeggen dat iedere individuele deelnemer evenveel privileges en taken heeft. Kortom is één van die

---

<sup>7</sup>**Distributed Computing:** een collectie van individuele computers die verbonden zitten in een netwerk en met elkaar kunnen communiceren. Deze computers zijn in staat om samen computermatige taken uit te voeren. ((Attiya, 2004))

deelnemers niet meer bereikbaar dan kan elke andere peer zijn rol op zich nemen.

- Hybride netwerk: er is één centrale component aanwezig die een bepaalde taak op zich neemt. Bijvoorbeeld bij het Bit-Torrent protocol is er een centrale component nodig om toe te treden tot het netwerk. Als deze centrale component onbereikbaar is functioneert het netwerk niet.

**Waarom zouden we dus kiezen voor een gedecentraliseerd systeem?** Om dit duidelijk te maken kunnen de voor- en nadelen worden samengevat in onderstaande tabel:

Vergelijking Gecentraliseerd en Gedecentraliseerd Systeem		
Probleem	Client-Server	P2P
<b>SPOF</b>	Doordat alle informatie zich op één server bevindt is er ook één kritieke plaats waar alles kan fout lopen. Is de server onbeschikbaar zijn alle diensten en informatie hierop dat ook.	Er is niet één computer die optreedt als server. Elke deelnemer van het netwerk (ook wel <b>Peer</b> genoemd) treedt op als server en cliënt. Hierdoor is er geen kritiek punt waarop het kan fout lopen. <sup>8</sup>
<b>Netwerkbelasting</b>	Doordat al het verkeer moet verwerkt worden door dezelfde hardware kan dit leiden tot een hoge serverbelasting. Hierdoor kan het verwerken van deze verzoeken traag verlopen.	Het hele netwerk verwerkt onderling verzoeken waardoor de individuele belasting op één peer eerder laag blijft. Hierdoor is het netwerk in staat om op een efficiënte manier verzoeken te verwerken.
<b>Uitbreidbaarheid</b>	Het uitbreiden van bestaande infrastructuur is niet evident en vereist vaak grote kosten voor een bedrijf.	Een p2p netwerk is goed uitbreidbaar. Op elk moment kunnen computers toetreden tot het netwerk of uittreden.

Tabel 2.2: Vergelijking tussen gecentraliseerd - gedecentraliseerd systeem



## Hashing

Hashfuncties is een begrip dat veel aan bod komt in de verschillende protocollen. Zeker binnen IPFS speelt hashing een grote rol. Daarom is het noodzakelijk dat het begrip toch kort wordt toegelicht.

Hashfuncties liggen aan de basis van het verifiëren van de dataintegriteit bij bestanden. Het is immers belangrijk om te kunnen nagaan of het bestand dat verkregen is hetzelfde is als wat verwacht wordt. Op deze manier kan er worden nagegaan of de bestanden eventueel beschadigd zijn door de overdracht. Een *goede* hashfunctie zou moeten voldoen aan volgende eigenschappen ((Anderson, 1993)):

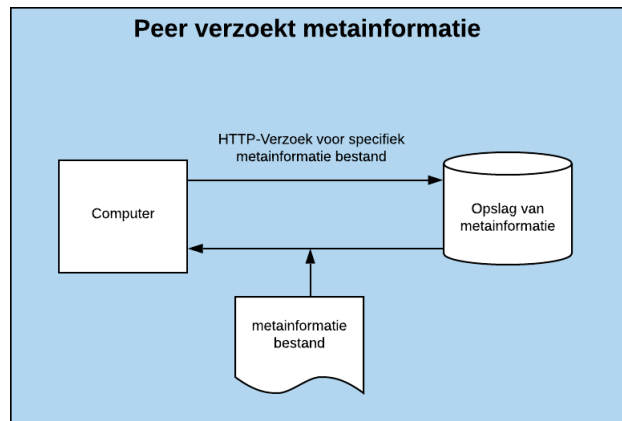
- One-way functie: het is relatief eenvoudig om voor een gegeven dataset  $x$  een resultaat te vinden  $f(x)$ . Het is echter onwaarschijnlijk om voor een gegeven resultaat  $f(x)$  de originele dataset  $x$  te vinden. Op deze manier is het makkelijk om een dataset te valideren indien de hashfunctie waarde is gekend.
- Collision free: het moet niet mogelijk zijn dat twee verschillende datasets (bijvoorbeeld:  $x$  en  $y$ ) dezelfde hashfunctie waarde hebben. Wiskundig kan dit worden voorgesteld door volgende notatie: stel een functie  $f : A \rightarrow B$ . Deze functie is collision free indien  $\forall \{x, y\} | \{x, y\} \subseteq A | f(x) \neq f(y)$
- Gelijke verdeling: een kleine verandering aan de dataset leidt tot een volledig verschillende hashwaarde. Zo is hashwaarde voor 123 en 124 sterk verschillend. Op deze manier kan er geen informatie worden afgeleid uit de hashwaarde zelf.
- Vaste grootte: ongeacht de grootte van de ingegeven dataset, is het resultaat van de lengte van de hash-functie constant. Zo zal een MD5 Hash altijd 128 bit lang zijn ongeacht of de ingegeven data 1 bit of 10000 bits beslaat.

### 2.2.3 BitTorrent

Het BitTorrent protocol is wijd verspreid. Een studie door Wang en Kangasharju (2013) stelt dat het dagelijks aantal van BitTorrent gebruikers tussen de 15-27 miljoen zit. Volgens de makers is de opzet ervan om een gangbaar alternatief te voorzien voor FTP. De uitleg van het protocol is gebaseerd op de tekst door Fonseca e.a. (2005).

Het Torrent protocol bestaat uit drie delen. Het eerste deel is het verkrijgen van een **metainfo** bestand.

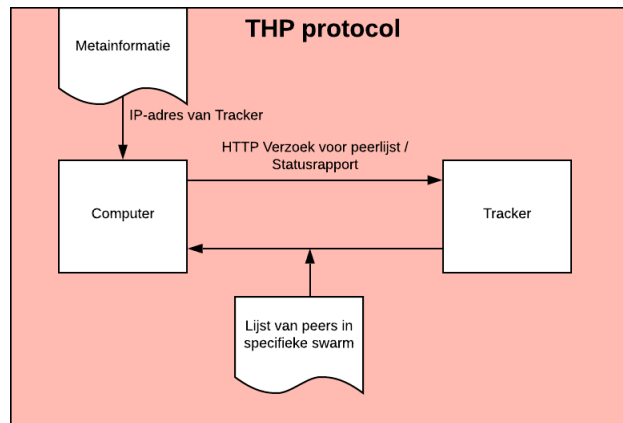
Dit metainfo bestand wordt over het algemeen afgehaald van een website of een ftp server. Een gekend voorbeeld van een website die metainfo bestanden ter beschikking stelt is



Figuur 2.6: De peer maakt een verzoek aan een opslagplaats voor een metainformatie bestand.

*The pirate bay*<sup>9</sup>. Dit bestand bevat essentiële informatie om de rest van het protocol te faciliteren. Het bevat de namen en bestandsstructuur van de verschillende bestanden binnen deze Torrent. Voor elk van deze bestanden is ook een MD5-Hashwaarde voorzien zodat eenmaal het bestand gedownload is de gebruiker kan controleren of het niet beschadigd is geraakt bij het download proces. Er wordt ook een zogenaamde infowaarde en stuklengte voorzien. Deze komen later aan bod. Het belangrijkste element in het metainfo bestand is echter het IP-adres van de **Tracker**. Deze Tracker is een computer die de andere delen van het protocol gaat overzien en aansturen. Aangezien er één centrale computer nodig is om dit protocol in goede banen te leiden is er dan ook sprake van een hybride netwerk.

<sup>9</sup>Het is ook belangrijk om kort te vermelden dat file-sharing controversieel is. Zo is de gekende Torrent indexering website "*The Pirate Bay*" sinds 2011 verboden in België. Dit verbod is er gekomen aangezien file-sharing ook wordt gebruikt voor het verspreiden van auteursrechtelijk beschermd materiaal. Niet alleen het Torrent protocol is hiervoor onder vuur komen te staan. Limewire een muziek deelplatform gemaakt bovenop Gnutella werd in 2010 verplicht om te sluiten. Het probleem met deze maatregelen is dat ze niet kunnen worden doorgevoerd. Men kan individuele sites namelijk blokkeren maar doordat de protocollen bovenop P2P netwerken zijn gebouwd is het niet mogelijk ze compleet onbeschikbaar te maken. Er zijn verschillende kopieën van de piratebay nog steeds toegankelijk in België en alternatieve versies van Limewire zijn eveneens beschikbaar. Het is hierbij belangrijk om te onthouden dat de principes en protocollen niet illegaal zijn maar het verspreiden van auteursrechtelijk beschermd materiaal is dat wel.

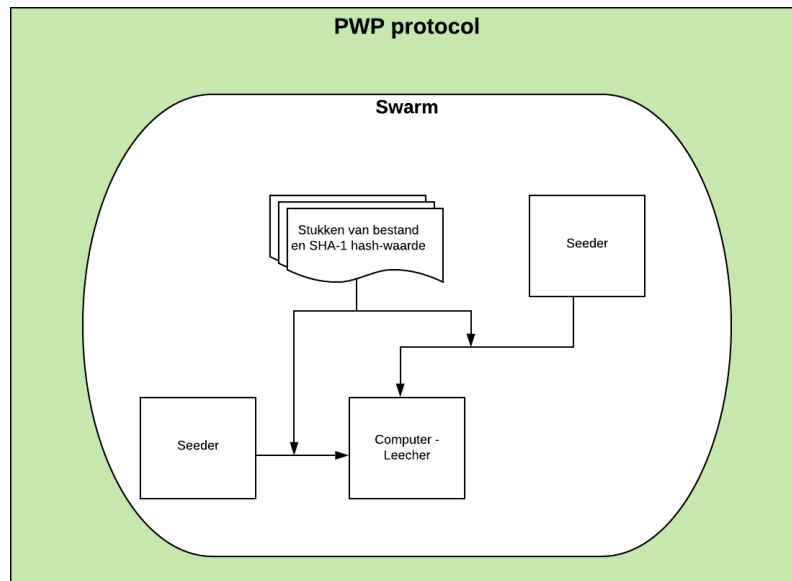


Figuur 2.7: De peer doet een verzoek voor de peerlijst aan de tracker door middel van informatie uit het metainformatie bestand.

Om deze stap te verduidelijken kan er gebruik worden gemaakt van volgend voorbeeld:

Bob en Alice maken een Torrent van hun website. Hun website bevat verschillende pagina's en afbeeldingen die zich in een media folder bevinden. Ze stellen een lijst op van deze verschillende bestanden en berekenen voor elk van deze bestanden een MD5-Hash waarde. Bobs computer zal de rest van het protocol coördineren en aldus functioneren als Tracker. Er wordt tot slot een unieke infosleutel voorzien en stuklengte (het is aangeraden deze stuklengte rond 70kb in te stellen). Op basis van deze informatie wordt een metainfo bestand aangemaakt dat wordt beschikbaar gesteld aan de buitenwereld.

In de derde stap wilt men toegang verkrijgen tot andere peers die dit bestand bezitten. Men heeft immers andere computers nodig vanwaar men het bestand kan downloaden. De verzameling van computers die een torrent toegankelijk stellen voor het netwerk wordt ook wel een **swarm** genoemd. Men kan een swarm dan ook zien als een klein netwerk met als functie toegang te verlenen tot een specifieke collectie van bestanden. De functie van de Tracker is om toegang te verschaffen aan deze Swarm en ook informatie hierover bij te houden. Het principe van toegang verschaffen en informatie bijhouden wordt ook wel *THP* (Tracker HTTP protocol) genoemd. De locatie van deze tracker staat vermeld in het metainfo bestand. Om een lijst van peers te krijgen die in een swarm zit, stuurt de computer een HTTP-GET verzoek naar de tracker. In dit verzoek moeten een aantal zaken worden meegestuurd waaronder het IP-adres van de computer, de infosleutel en de poort waarop de communicatie met andere peers kan gebeuren. De Tracker zal vervolgens aan de hand van de infosleutel een lijst van peers terugsturen die de gegeven bestanden ter beschikking stellen.



Figuur 2.8: De peer verzoekt andere peers voor stukken data en downloadt op die manier de Torrent bestanden.

Om dit principe te verduidelijken wordt verder gebouwd op bovenstaand voorbeeld:

Charlie wil de website van Bob en Alice. Hij heeft van Alice het metainfo bestand ontvangen. Aangezien dit bestand het IP-adres van Bobs computer als Tracker bestempeld, wordt er naar deze computer een HTTP-GET verzoek verstuurd. Bob en Alice hebben beiden deze bestanden lokaal staan en vormen aldus samen een swarm. Charlie zal hun IP-adres en poort waarop het verkeer kan verlopen doorgestuurd krijgen en gaat rechtstreeks met beide computers communiceren om de bestanden te downloaden.

Er dient opgemerkt te worden dat netwerken en bestanden erg veranderlijk zijn. Een computer kan elk moment uitgaan of van het netwerk treden. Het is dus noodzakelijk dat de Tracker een accurate lijst van peers bijhoudt in de Swarm. Om dit te bereiken zal een Tracker vragen dat elke peer op een vast tijdsinterval zijn status rapporteert. Zo kan de Tracker accurate informatie voorzien.

De computer heeft nu een lijst van verschillende Peers binnen een specifieke swarm. De laatste stap van het protocol is het effectief verkrijgen van de bestanden. De communicatie loopt uitsluitend tussen de verschillende peers zelf. Deze stap wordt ook wel aangeduid als het *PWP* protocol (Peer Wire Message). Bandbreedte en andere beperkingen spelen een grote rol binnen netwerkverkeer. Men wil een peer zo weinig mogelijk belasten. Op die manier stimuleert men de peer om zolang mogelijk op het netwerk te blijven. Het is aldus niet praktisch om volledige bestanden op te vragen van één peer. Stel dat het bestand bijvoorbeeld verschillende gigabytes groot is. Hierdoor zal men het bestand opdelen in verschillende kleine stukken. Deze stukken kunnen individueel worden opgevraagd aan verschillende computers. Op deze manier kan men een bestand van verschillende gigabytes

in duizend keer opvragen bij verschillende peers die elk enkele mb doorsturen. De grootte van elk stuk wordt bepaald in het meta-info bestand en staat vastgelegd in de zogenaamde stuklengte. Een ander voordeel van deze fragmentering is dat men individuele stukken kan verifiëren. Elk van deze stukken heeft namelijk zijn eigen SHA-1 hashwaarde. Op die manier kan men gedurende de overdracht van een stuk onmiddellijk nagaan of dit goed is toegekomen en indien er fouten zijn dit klein stuk opnieuw opvragen. Het PWP protocol komt neer op het opvragen en valideren van individuele stukken aan verschillende peers.<sup>10</sup> Eenmaal alle stukken zijn gedownload, worden deze samengenomen tot het origineel bestand dat op zijn beurt kan worden gevalideerd aan de hand van MD5.

Het Torrent protocol heeft een aantal voordelen. Zoals eerder vermeld is het nog steeds relevant hoewel het van 2002 dateert. Het biedt immers een relatief eenvoudige manier aan om bestanden te downloaden. Door een goede meta-informatie indexeringswebsite te gebruiken zoals *The pirate bay* vindt men op eenvoudige wijze de correcte bestanden terug. Dit is bij FTP minder evident.

Toch zijn er ook veel technische nadelen verbonden aan het Torrent protocol<sup>11</sup>:

- Doordat men gebruik moet maken van een centrale Tracker is er ook één SPOF. Indien de tracker onbereikbaar is dan kan het Torrent protocol niet lang functioneren.
- Het verkrijgen van de meta-informatie bestanden is niet evident. Men moet toegang krijgen door middel van een indexeringswebsite of de auteur van de Torrent moet dit bestand doorsturen.
- Volgens Thanekar e.a. (2010) kunnen bestanden permanent verloren gaan. Dit is het geval indien er geen Seeders of volledige kopieën beschikbaar zijn.
- Peers communiceren rechtstreeks met elkaar over TCP. Hiervoor heeft men elkaars IP-adres nodig. Dit leidt er dus toe dat men niet anoniem is bij de download en upload van bestanden.
- Er is weinig stimulans voor een peer die bestanden downloadt om deze ook toegankelijk te stellen voor het netwerk en de rol van Seeder op zich te nemen. Om dit probleem te vermijden wordt een zogenaamde *Tit for tat* beleid gehanteerd. Hierbij kunnen peers enkel bestanden downloaden indien ze effectief ook bijdragen aan de download van anderen.
- Tot slot moet men ook rekening houden dat er geen manier is om volledig te verifiëren wat men downloadt. Men kan nagaan of het bestand hetzelfde is als dat in de metafile. Dit is echter geen garantie dat het bestand veilig is en de relevante informatie bevat.

---

<sup>10</sup>De manier waarop deze stukken worden opgevraagd, ligt buiten de scope van deze bachelorproef. Een goed overzicht wordt gegeven in de tekst van Fonseca e.a. (2005).

<sup>11</sup>Er zijn ook veel morele implicaties verbonden aan Torrenting maar deze vallen buiten de scope van deze bachelorproef.

Vele van deze problemen stellen zich ook in het klassieke Client-Server model.

## 2.2.4 IPFS

### Inleiding

IPFS is een modern file-sharing protocol dat veel van de tekortkomingen van Bit-Torrent oplost. Het is ook het protocol dat zal worden gebruikt in het opstellen van de POC. De uitleg in dit hoofdstuk is afkomstig van de officiële documentatie (IPFS, 2020b) en video's gepubliceerd door Explained (2018) en IPFS (2020a).

In dit hoofdstuk ligt de focus op **Content based addressing** en het concept van een **CID**. Deze twee concepten spelen namelijk een grote rol binnen de POC.

### Content Based Addressing en CIDS

IPFS beperkt zich niet louter tot bestandsoverdracht. Op de officiële website van IPFS wordt het protocol als volgt beschreven: IPFS is een gedistribueerd bestandssysteem dat een alternatief biedt voor de manier waarop bestanden op het internet worden opgevraagd. Wat wordt daar echter mee bedoeld?

IPFS heeft als doelstelling om alle bestanden op het netwerk te verspreiden over verschillende peers. Deze bestanden kunnen vervolgens eenvoudig worden opgevraagd en weergegeven in een webbrowser. Een peer kan vervolgens rechtstreeks aan het netwerk deze bestanden opvragen. In tegenstelling tot Bit-Torrent is er dus geen sprake van Trackers of Swarms. Dus wat maakt IPFS uniek?

Volgens IPFS is er een probleem met de manier waarop er klassiek wordt gezocht naar bestanden. Stel bijvoorbeeld onderstaand scenario:

Bob wilt graag een nieuwe achtergrond en gaat op zoek naar een afbeelding van een Capybara. Hij vindt twee afbeelding op google van een mooie Capybara en slaat vervolgens de links op. Drie dagen erna bezoekt hij nog eens de links. Eén van deze links leidt naar een 404 pagina. De afbeelding is immers verwijderd. De andere link leidt tot zijn verbazing tot een afbeelding van een Tapir.

Dit is eveneens een voorbeeld van **Location based addressing**. Location based addressing(=lba) is het principe dat men een bestand gaat opvragen aan de hand van waar dit bestand zich bevindt. Zo gaat de link `abc.com/cat.jpg` het bestand `cat.jpg` opvragen aan de server van `abc.com`. Bovenstaand voorbeeld illustreert ook onmiddellijk drie problemen met lba:

1. Informatie kan gemakkelijk offline worden gehaald. Er is geen enkele garantie dat afbeeldingen, websites en andere bestanden beschikbaar blijven.
2. Er wordt veel vertrouwen gesteld in servers dat de informatie die men opvraagt correct is. Omdat een bestand een bepaalde naam heeft wilt dat niet noodzakelijk iets zeggen over de inhoud ervan.
3. Informatie kan eenvoudig worden gewijzigd. De afbeelding die gisteren nog een copybara was kan veranderd zijn in een afbeelding van een Kat.

Het zijn deze problemen die worden aangepakt met Content Based Addressing (=CBA), IPFS en het concept van een CID. Binnen CBA worden bestanden opgevraagd aan de hand van hun inhoud. Bij een klassieke url moet immers zowel de naam van de server (of het ip-adress) alsook de naam van het bestand in kwestie gekend zijn. CBA vervangt dan ook het concept van een URL door een uniek identificatienummer dat gebaseerd is op de inhoud van het bestand. Dit uniek identificatie nummer wordt een CID genoemd.

## 2.3 Blockchain principes

### 2.3.1 Inleiding

Blockchain is een gedecentraliseerde opslagoplossing. Blockchain werd in 2009 ontworpen door Nakamoto (2009) en vormt de basis van Bitcoin. Zoals vermeld in het Witboek van Nakamoto (2009) is de doelstelling van Bitcoin om een elektronisch peer-to-peer betalingssysteem te voorzien. Het idee van Nakamoto was vooral revolutionair aangezien het een manier biedt om binnen een P2P netwerk gegevens op te slaan en deze gegevens ook te valideren. Het probleem dat Bitcoin oplost voor P2P netwerken staat ook wel bekend als het *Byzantine Generals' Problem*.

Dit probleem kan als volgt worden beschreven: Drie generalen moeten beslissen om een stad aan te vallen. Ze kunnen enkel communiceren via koeriers die tussen de verschillende generalen berichten overbrengen. De drie generalen moeten het eens geraken om de stad aan te vallen of met rust te laten. Eenmaal de aanval is ingezet is er geen manier om terug te trekken (Slagter, 2018). Aanvankelijk lijkt dit probleem triviaal. Ze sturen alle drie een bericht naar elkaar en bereiken een consensus door twee derde meerderheid. Toch is dit probleem complexer: wat als de boodschapper niet te vertrouwen is bijvoorbeeld? Wat als er een generaal is die twee verschillende stemmen uitbrengt?

Dit probleem demonstreert dat het niet eenvoudig is om tot een consensus te komen in een systeem waar ieder als gelijke wordt gezien. Er is immers één rotte appel nodig om de hele consensus te beïnvloeden. Blockchain is innoverend omdat het een consensus

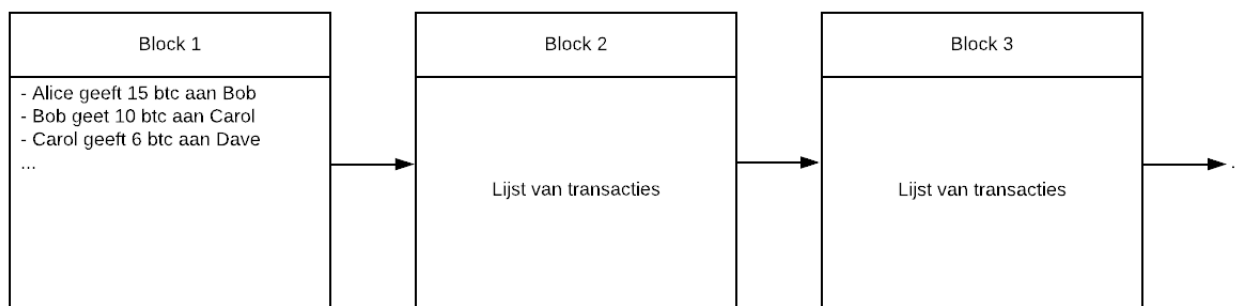
algoritme biedt om binnen een P2P netwerk -waar iedere node als gelijke wordt gezien- boodschappen en gegevens op te slaan. Deze boodschappen en gegevens worden ook wel **transacties** genoemd.

Hoe kadert blockchain binnen de POC die in deze bachelorproef wordt ontwikkeld? Zoals eerder vermeld -zie 2.2.1- bestaat een versiebeheersysteem niet enkel uit bestanden maar ook uit een logboek. Dit logboek is in essentie een overzicht van alle wijzigingen in de bestanden. Dit logboek moet worden opgeslagen en kunnen worden aangepast op een P2P netwerk. Hiervoor kan er worden gebruik gemaakt van een blockchain.

### 2.3.2 Basisprincipes van een blockchain

In deze sectie wordt blockchain uitgelegd aan de hand van het originele voorstel door Nakamoto (2009) en de video gepubliceerd door 3Blue1Brown (2017).

In het kort gesteld is blockchain een geheel van met elkaar verbonden blokken van transacties. Dit systeem van onderling verbonden blokken wordt ook wel een **ledger** genoemd. Elke peer heeft ten allen tijden een kopie van deze ledger. Een transactie is een bepaalde set van data. In het geval van bitcoin is dit een overschrijving. Een voorbeeld van een transactie op het bitcoin netwerk is Alice geeft Bob 25 bitcoins. Een blok is aldus een lijst van transacties. Om dit principe te verduidelijken, kan er gebruik worden gemaakt van onderstaande grafische voorstelling:



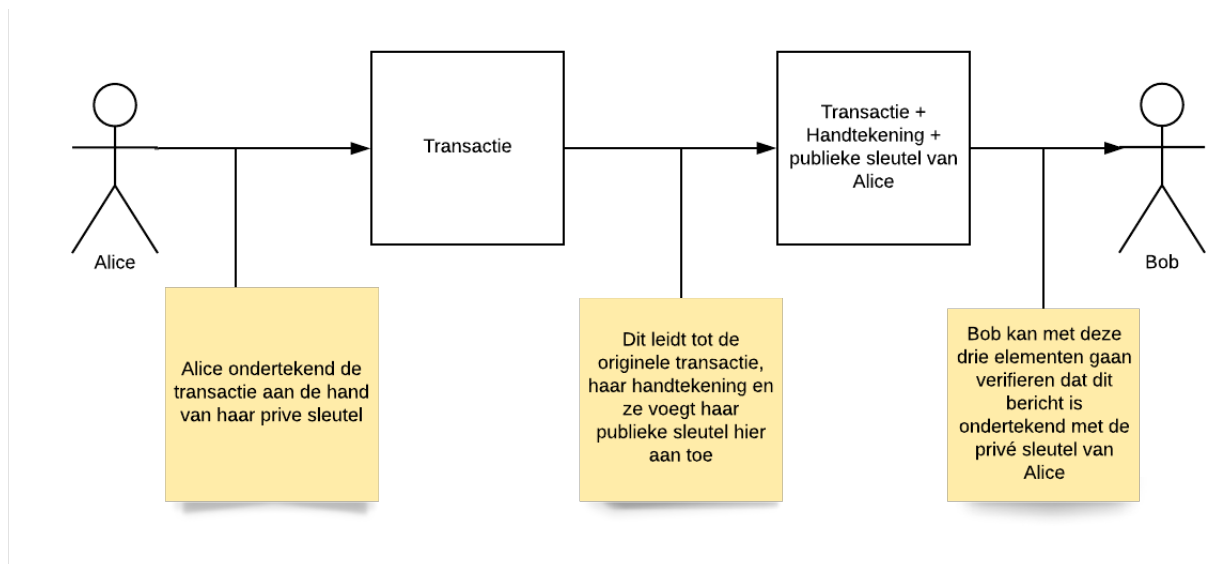
Figuur 2.9: De blockchain bestaat uit onderling verbonden blokken. Een blok bevat een lijst van transacties.

Hoe worden deze transacties gevalideerd? Hoe weten we bijvoorbeeld dat Alice wel degelijk 15 bitcoins aan Bob geeft? Dit gebeurt door **asymmetrische cryptografie**. Farah e.a. (2012) stellen dat asymmetrische cryptografie gebruik maakt van twee sleutels: enerzijds de sleutel die gebruikt wordt bij het encrypteren van een bericht, anderzijds de sleutel die gebruikt wordt bij het decrypteren. Als zowel de encryptiesleutel en het hashingalgoritme



gekend zijn dan is het computermatig onwaarschijnlijk dat de decryptiesleutel kan worden achterhaald. De encryptiesleutel wordt ook wel de **publieke sleutel** genoemd. De decryptiesleutel wordt de **privé sleutel** genoemd. De privé sleutel is enkel in handen van één persoon en mag niet gedeeld worden met andere. De publieke sleutel is toegankelijk voor iedereen. Hoe wordt dit principe gebruikt om te verifiëren dat Alice wel degelijk de verzender is van het bericht?

Alice kan bewijzen dat ze een transactie heeft goedgekeurd door deze transactie te ondertekenen met haar privé sleutel. Dit principe is gelijkaardig aan de manier waarop officiële documenten worden ondertekend. Bij handtekeningen op officiële documenten kan er worden geverifieerd dat deze ondertekend zijn door een bepaald persoon door middel van zijn/haar handschrift. Bij digitale handtekeningen kunnen we verifiëren dat ze zijn ondertekend door een bepaald privé sleutel van een persoon. Dit gebeurt door de corresponderende publieke sleutel, de handtekening en het originele bericht.<sup>12</sup> Dit wordt geïllustreerd in de onderstaande grafiek:



Figuur 2.10: Alice ondertekent transacties aan de hand van haar privé sleutel.

Elke peer heeft zoals reeds eerder vermeld zijn eigen kopie van de ledger. Dit wilt zeggen dat elke deelnemer van het netwerk ten allen tijden een volledige lijst van transacties heeft. Een voordeel van een volledige lijst van transacties bij te houden is dat er kan geverifieerd worden dat Alice wel degelijk over 15 bitcoin beschikt. Alice kan aldus niet meer uitgeven dan ze reeds heeft. Op het moment dat Alice of Bob een transactie uitvoert, zal deze dus over heel het netwerk worden verspreid. Al deze transacties worden gebundeld onder de vorm van blokken -zie grafiek 2.9 -. Door bijvoorbeeld twee keer dezelfde blokken toe te voegen of bepaalde blokken eruit te halen kan een individuele gebruiker blockchain en transacties manipuleren. Om dit te vermijden wordt het concept van een consensus algoritme gebruikt. Om meer specifiek te zijn wordt binnen bitcoin het zogenaamde **Proof**

<sup>12</sup> Asymmetrische encryptie is een uitgebreid concept. Voor meer informatie hoe deze wordt toegepast binnen bitcoin zie: [https://learnmeabitcoin.com/beginners/digital\\_signatures](https://learnmeabitcoin.com/beginners/digital_signatures)

of **Work** algoritme gebruikt.

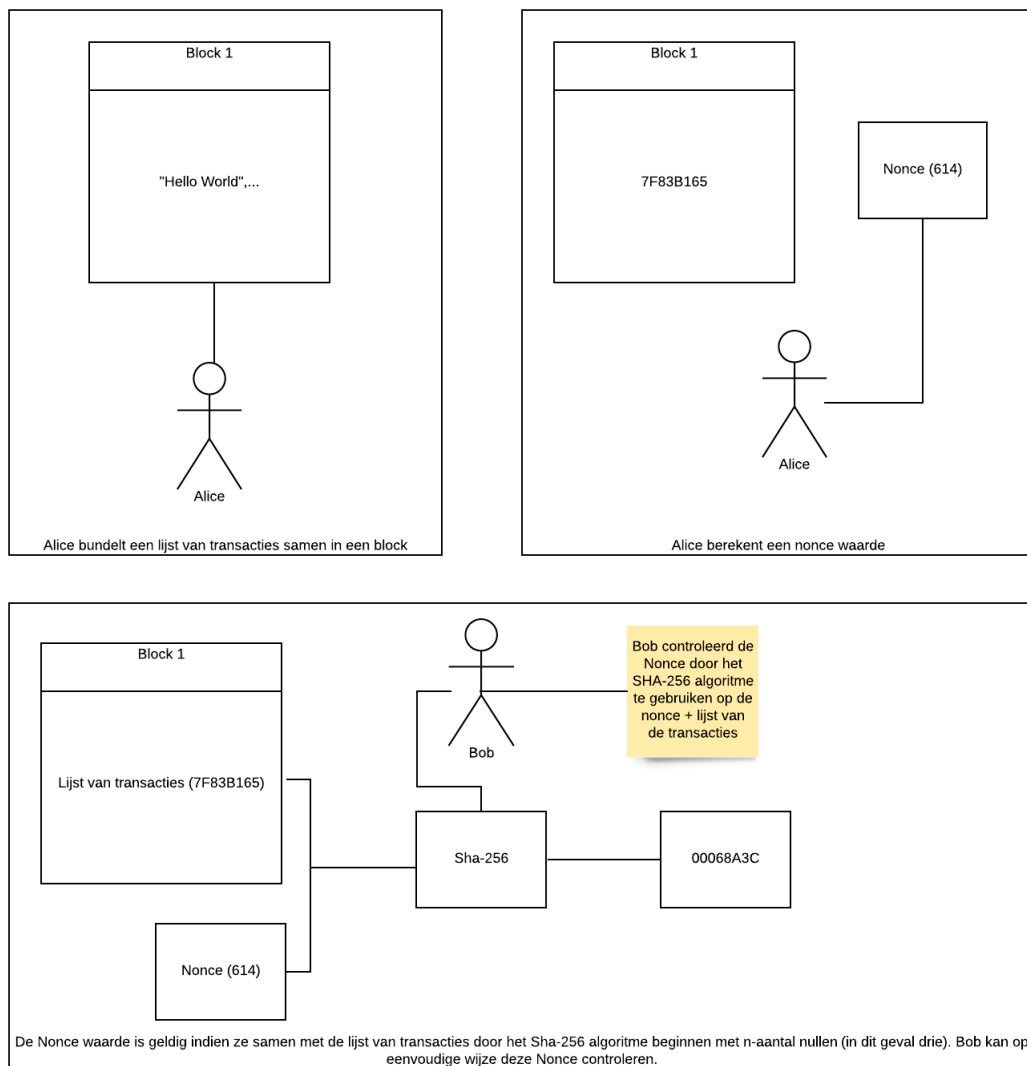
Proof of work is een concept dat is ontwikkeld door Dwork en Naor (g.d.). Proof of work vereist het oplossen van een uniek wiskundig probleem alvorens een bepaalde handeling mag worden uitgevoerd. In het artikel van Dwork en Naor (g.d.) werd dit principe gebruikt om spam te vermijden. Indien een computer voor elke e-mail een wiskundig probleem moest oplossen is het praktisch onmogelijk om duizenden en duizenden e-mails te versturen. Hoe kadert dit principe binnen blockchain?

Bitcoin gebruikt het proof of work (=PoW) principe voor het valideren van blokken. Pas als het volgende wiskundig probleem is opgelost, mag een blok worden toegevoegd aan de blockchain. Binnen cryptografie speelt het concept van hashing een grote rol - zie 2.2.2 -. Een eigenschap van deze hashes is dat het wiskundig onwaarschijnlijk is om gegeven een bepaalde hashwaarde de originele input waarde te vinden. Het is echter relatief eenvoudig om een bepaalde input waarde te hashen. Het PoW algoritme gebruikt deze eigenschappen door volgend wiskundig probleem op te stellen:

Bereken de hashwaarde gebruik makend van alle transacties + een bepaald getal. De resulterende hash moet x-aantal nullen bevatten in het begin van de hashcode. Het probleem bestaat er aldus uit om het getal te vinden dat ervoor zorgt dat de resulterende hash met x-aantal nullen begint. Om dit principe nog eens te verduidelijken stel onderstaand scenario:

Alice ontvangt 200 transacties. Ze bundelt deze samen tot een blok. De wiskundige opgave bestaat er aldus uit om een getal  $x$  te vinden (ook wel de **nonce** genoemd) dat samen met de lijst van transacties door het SHA-256 een hash code oplevert met n-aantal nullen in het begin.

De volgende grafiek verduidelijkt dit scenario:



Figuur 2.11: Voorstelling van een hash-puzzel.

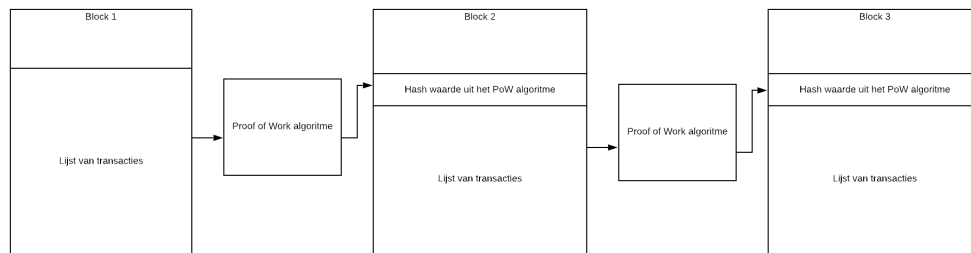
Het proof of work algoritme heeft twee verschillende eigenschappen:

1. De nonce kan enkel worden gevonden door alle verschillende mogelijkheden te berekenen. Zoals vermeld in 2.2.2 kan men niet via de gewenste uitkomst de inputwaarde berekenen. Men kan dus niet simpelweg een getal met het aantal vereiste 0-bits omzetten naar een corresponderende inputwaarde.
2. De moeilijkheidsgraad voor het algoritme is rechtstreeks verbonden met het aantal nullen dat vereist is in het begin van de resulterende hashwaarde. Dit is immers ook logisch. Stel bijvoorbeeld dat de resulterende hashwaarde 10 bit lang is en slechts één 0-bit vereist vooraan dan zijn er  $2^9 = 512$  toegestane waarden. Wordt deze vereiste op vijf 0-bits gelegd dan zijn er slechts  $2^5 = 32$  mogelijke uitkomsten.

Er kan dus gesteld worden dat indien een peer een bepaalde nonce waarden heeft berekend deze een significant rekenkundige taak heeft verricht. Hoe zorgt het Proof Of Work algo-

ritme nu echter voor data integriteit binnen een blockchain netwerk? Anders gesteld hoe zorgt dit algoritme ervoor dat fraude onwaarschijnlijk is?

De echte kracht van blockchain ligt in het feit dat de blokken onderling zijn verbonden. Concreet wilt dit zeggen dat de oplossing uit het PoW algoritme wordt opgenomen in het volgende blok. Volgende grafiek illustreert dit principe:



Figuur 2.12: De blockchain is een verzameling van blokken die onderling verbonden zijn.

Dit maakt het vervalsen van transacties uitermate moeilijk om de volgende reden:

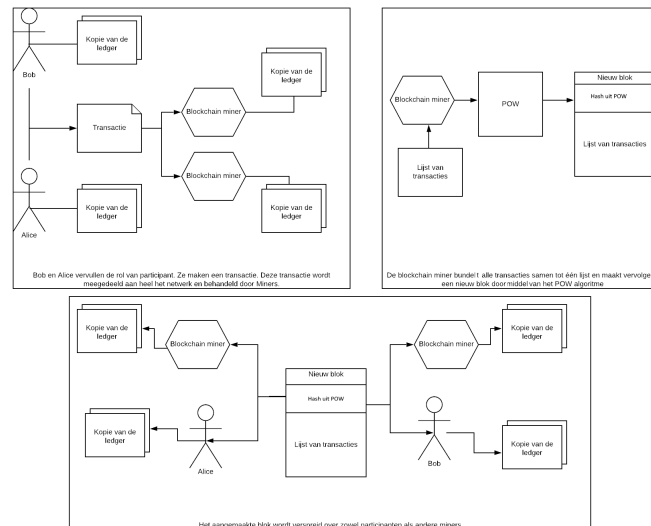
Stel dat er momenteel 85 blokken aanwezig zijn op de Blockchain. Bob wil blok 42 veranderen. Hiervoor berekent hij opnieuw de nonce waarde. Bob stuurt dit nieuwe blok naar alle andere peers. Deze peers zitten momenteel op blok nummer 85 en zullen het nieuw blok aldus niet opnemen. **De langste keten is immers correct.** Het blok dat Bob heeft gestuurd, bevat de PoW hash van blok 41 en dus niet de meest recente PoW hash waarde. Bob zal dus de overige 44 blokken ook moeten berekenen. Tegen de tijd dat Bob al deze blokken heeft berekend, is het netwerk hoogstwaarschijnlijk al veel verder. Tenzij Bob sneller dan alle andere peers deze nonce waarde kan berekenen is het vervalsen haast onmogelijk.

Alle bovenstaande principes vormen de achterliggende technologie van blockchain netwerken. Een korte definitie luidt dus als volgt: blockchain is een manier om in een gedecentraliseerd systeem aan dataopslag te doen. Het is een aaneenschakeling van lijsten van transacties. Hierbij wordt een Proof of Work algoritme gebruikt om ervoor te zorgen dat frauduleuze transacties onwaarschijnlijk zijn.

### 2.3.3 Blockchain mining

Om deel te nemen aan een blockchain netwerk moet men ten alle tijden een recente kopie hebben van de ledger. Vervolgens kan men transacties maken en deze gaan valideren. Het is hierbij belangrijk om te onthouden dat **transacties valideren geen vereiste** is om transacties aan te maken. Peers die actief transacties valideren aan de hand van het Proof-of-work algoritme worden **miners** genoemd. Peers die deelnemen aan het netwerk worden doorheen deze tekst aangeduid als **participanten**. Het is de taak van miners om

transacties door andere peers te bundelen in blokken en vervolgens het PoW algoritme toe te passen. Eenmaal een blok is gevonden, wordt deze verspreid over het netwerk (over zowel participanten als andere miners). De volgende grafiek verduidelijkt het verschil tussen een participant en een miner:



Figuur 2.13: Een demonstratie van het verschil tussen miners en participanten.

Zoals hierboven vermeld is het verifiëren van blokken niet verplicht om deel te nemen aan het netwerk. Waarom zouden miners hun tijd en energie in dit proces steken? Ze worden beloond voor het verifiëren aan de hand van een **block reward**. Wanneer ze een transactielijst samenstellen voegt het bitcoin protocol een speciale transactie eraan toe. Deze transactie wijst een aantal bitcoin toe aan de miner van dit blok. Slaagt de miner er aldus in om dit blok te verifiëren krijgt hij het aantal bitcoin van deze transactie.

Zoals hierboven vermeld worden er dus nieuwe munten in circulatie gebracht door het PoW algoritme. Dit is zeker belangrijk als men realiseert dat er geen centrale autoriteit is die deze munt in circulatie brengt. In België bijvoorbeeld wordt de hoeveelheid geld in circulatie beheerd door de Nationale Bank. Hierdoor is er echter een probleem. Mocht het verificatie proces te snel gaan, worden er teveel munten in omloop gebracht wat leidt tot het principe van inflatie. Hoe verloopt dit bij Bitcoin?

Er zijn drie manieren waarop bitcoin en andere varianten hun geldstroom beheren:

1. Zoals eerder vermeld is de moeilijkheidsgraad en tijd die het duurt voor het verifiëren rechtstreeks afhankelijk van het aantal 0-bits vereist in het PoW algoritme. Het bitcoin algoritme past dit aantal 0-bits automatisch aan zodat er gemiddeld één blok wordt aangemaakt elke tien minuten. Er worden aldus niet teveel nieuwe munten in omloop gebracht.
2. De grootte van een blok is beperkt tot 1 megabyte. De reden hierachter is dat iedere peer een kopie moet bijhouden van de ledger. Opslagruimte is namelijk beperkt. Daarom kan een blok enkel om en bij de 2400 transacties bevatten. Als men weet dat er één blok elke 10 minuten wordt aangemaakt is het aantal transacties aldus min of meer constant. Het bitcoin netwerk is in staat om ongeveer 14.400 transacties te verwerken per uur.
3. Het aantal munten dat een miner krijgt voor het verifiëren van een transactie wordt ook wel de **block reward** genoemd. Deze block reward wordt automatisch gehalveerd elke 210.000 blokken (ongeveer 4 jaar). Hierdoor wordt het aantal nieuwe munten in circulatie verminderd.

Zoals hierboven vermeld is het aantal transacties gelimiteerd tot ongeveer 14.400 transactie per uur. Hierdoor is het dus niet duidelijk dat wanneer een participant een transactie verstuurt deze ook effectief snel zal verwerkt worden. Om miners aan te sporen hun transacties te versnellen kunnen participanten ook een **transactiekost** aan hun transactie toevoegen. Dit is een kleine optionele kost die participanten toevoegen en die rechtstreeks naar de miner gaat indien deze transactie verwerkt wordt.

### 2.3.4 Ethereum

Volgens Foundation, 2020 is Ethereum een open-source platform voor het schrijven van gedecentraliseerde applicaties. Op dit platform wordt gebruik gemaakt van blockchains. Het idee komt van Vitalik Buterin en werd geformaliseerd in een Witboek gepubliceerd in 2013. De volgende uitleg is dan ook gebaseerd op dit Witboek Buterin, 2015.

Het bitcoin protocol is gebouwd bovenop blockchain technologie. Het biedt -zoals uitgelegd in de inleiding- een manier om aan gedecentraliseerde dataopslag te doen. Hoe maakt men gebruik van deze nieuwe technologie in gedecentraliseerde applicaties? Stel dat men bijvoorbeeld berichten wil opslaan in de blockchain. Hoe maakt men dan gebruik van blockchain? Zoals Buterin vermeld zijn er twee mogelijkheden:

1. De applicatie voorziet zijn eigen vorm van blockchain technologie. Het is in essentie zijn eigen netwerk. Dit is echter geen eenvoudige taak. Men moet namelijk niet alleen de volledige code van de blockchain zelf implementeren maar men moet ook een eigen netwerk maken met participanten en miners. Dan zit men ook nog met de logistieke taak van een voldoende groot netwerk te hebben en de miners effectief aan te sporen om te valideren. Dit is dus hoogst onpraktisch.
2. De applicatie maakt gebruik van de Bitcoin blockchain. Bitcoin is hier echter niet inherent voor ontworpen. De doelstelling van Bitcoin is namelijk om een p2p betalingssysteem te voorzien. Het is niet gemaakt voor de ontwikkeling van gedecentraliseerde applicaties.

Ethereum is dus ontstaan uit de noodzaak voor een zogenaamde programmeerbare blockchain. Dit wilt zeggen een blockchain die bedoeld is om het schrijven van gedecentraliseerde applicaties te bevorderen. Ethereum steunt hierbij op hetzelfde algoritme en basisprincipes van bitcoin. Dat wilt zeggen dat er nog steeds sprake is van miners, een digitale munteenheid genaamd ether en een PoW algoritme. Ethereum bereikt deze programmeerbare blockchain door het concept van **Smart contracts**.

### 2.3.5 Smart contracts

Smart contracts kunnen het best worden vergeleken met objecten met wie ze veel gelijkenissen vertonen. Zo kent een smart contract volgende concepten:

1. Een smart contract bevat variabelen en arrays. Dit zijn in essentie opslag plaatsen voor bepaalde waarden. Het datatype van deze variabelen wordt omgezet in bytes. Deze omzetting is gelijkaardig aan hoe zogenaamde *high-level* programmeertalen zoals C# en java werken.
2. Een smart contract bevat functies. Deze functies kunnen de variabelen manipuleren of uitlezen. Zoals in klassieke programmeertalen kunnen functies ook parameters

mee krijgen. Ze kunnen eveneens waarde teruggeven. Het is ook belangrijk om op te merken dat smart contracts functies kunnen uitvoeren op andere smart contracts

3. Smart contracts bevatten **control flow**. Daarmee wordt bedoeld dat ze concepten bevatten zoals if statements en for loops.

Er bestaan verschillende talen voor het schrijven van smart-contracts. De taal die gebruikt wordt binnen de POC wordt solidity genoemd. Het volledig overlopen van de syntax van deze taal ligt buiten de scope van deze bachelorproef. Voor enkele concrete voorbeelden van deze taal en een beknopte uitleg zie Appendix 3.

Ethereum is in essentie een gedistribueerde computer. Het netwerk bestaat uit een verzameling van miners en participanten. Miners passen data in de blockchain aan op basis van code onder de vorm van smart contracts. Dit stelt software ontwikkelaars in staat om gedecentraliseerde applicaties te schrijven .

Elke transactie op de blockchain moet verwerkt worden door een miner. Hierbij voeren miners de code van smart contracts uit. Dit introduceert echter een probleem: wat als het contract vast komt te zitten of het veel tijd vergt? Om dit probleem tegen te gaan wordt er voor elke stap in de transactie een transactiekost vereist. Deze transactie kost wordt ook wel GAS genoemd. Doordat elke stap een bepaalde kost met zich mee draagt is het voor software ontwikkelaars cruciaal om deze transacties zo kort mogelijk te houden. Het is aldus ook niet mogelijk om een oneindige lus te schrijven. Deze GAS transactie kost gaat integraal naar de miners als een beloning voor het verrichte werk. Hierdoor worden miners ook aangezet om transactie te verwerken op het Ethereum netwerk.

Smart contracts kunnen niet meer worden gewijzigd. Indien een smart contract op de blockchain is geplaatst, dan is deze permanent. Het is dus cruciaal dat een software ontwikkelaar zijn smart contracts goed test alvorens ze op het netwerk te plaatsen.



## 3. Methodologie

### 3.1 Hoe de onderzoeksvraag tot stand is gekomen.

Dit onderzoek ontstond uit een persoonlijke interesse. Internet is een zeer veranderlijk geheel met tal van nieuwe technologieën die vervolgens ingeburgerd raken. Software ontwikkeling is dan ook geen vast gegeven. Toch zijn er een aantal basisfundamenten waarop software ontwikkeling gebaseerd is. Een van deze fundamenteen is het klassieke server-client model. In dit model voorziet één centrale computer bestanden of diensten voor tal van andere computers. Een gekend voorbeeld hiervan is de manier waarop het internet werkt ook wel bekend onder het HTTP protocol.

Bestaat er dan geen alternatief voor het Client-Server model? Distributed Computing tracht deze vraag te beantwoorden. Eén van de oplossingen is een zogenaamd P2P netwerk. Binnen dit type netwerk voorzien computers onderling bestanden en diensten aan elkaar. Dit type netwerk werd vooral populair onder de vorm van File-Sharing Protocollen. Gekende voorbeelden zijn hierbij Pirate Bay en Napster. Hoewel deze protocollen een eenvoudig en elegant alternatief boden, stonden ze nog niet op punt. Zo was er geen manier om op een veilig wijze aan dataopslag te doen.

Blockchain bracht een oplossing voor het probleem van dataopslag. Blockchain technologie is vooral bekend van Bitcoin en de financiële impact die deze met zich mee bracht. Toch zijn de implicaties veel diepgaander. Blockchain biedt een manier om op een veilige wijze aan dataopslag te doen in een gedistribueerd netwerk. Doordat data integriteit kan gewaarborgd worden kunnen ontwikkelaars veiligere en robuustere oplossingen ontwikkelen voor een P2P netwerk.

Toch bleef de blockchain technologie niet beperkt tot Bitcoin en P2P netwerken en werd ze veel verder ontwikkeld. Zo kwam Ethereum met het idee van smart contracts die software ontwikkelaars in staat stellen om transacties op Blockchain netwerken verder te controleren. Hierdoor kunnen applicaties worden ontwikkeld die op een volledig gedistribueerde manier werken. Het is een zeer fascinerende en revolutionaire manier van denken die grote implicaties met zich mee kan brengen.

Deze bachelorproef wil deze technologie verkennen en gebruiken in één van de meest fundamentele stappen van moderne software ontwikkeling: versiebeheer.

Uiteraard is deze bachelorproef niet het enige onderzoek over het decentraliseren van versiebeheer. In hun onderzoek decentraliseren Nizamuddin e.a. (2019) versiebeheer aan de hand van de Ethereum blockchain en IPFS. Dit is een zeer interessante benadering geweest maar te beperkt. Veel van de achterliggende concepten werden niet uitgelegd en hun oplossing was eerder beperkt in grootte en complexiteit. Deze bachelorproef wilt hierop een aanvulling bieden en uitleggen wat blockchain, versiebeheer en IPFS exact zijn en hoe deze kunnen worden gecombineerd.

## 3.2 Het ontleden van de onderzoeksvraag.

Zoals besproken in 3.1 is de uiteindelijke onderzoeksvraag waarop een antwoord wordt gezocht: **Hoe kan men door middel van Blockchain principes en IPFS een werkbaar gedecentraliseerd versiebeheersysteem ontwikkelen?** Deze vraag wordt onderverdeeld in verschillende deelvragen die sequentieel zijn beantwoord in de literatuurstudie. Hieronder volgt een kort overzicht en antwoord op de verschillende deelvragen. Voor meer informatie zie 2.

Deelvraag	Antwoord
Wat zijn de problemen die versiebeheersystemen oplossen?	Versiebeheersystemen bieden een manier om verschillende softwareversies aan te maken. Hierdoor kunnen softwareontwikkelaars ten allen tijden toegang krijgen tot verschillende versies van hun software. Versiebeheersystemen stellen ontwikkelaars in staat om hun code op een eenvoudige manier onderling met elkaar te delen en samen te werken.
Waaruit bestaat een versiebeheersysteem?	Een versiebeheersysteem bestaat uit twee delen: een manier om bestanden op te slaan en te delen met elkaar. Ten tweede een wijze om bestanden te archiveren. Dit wilt zeggen dat ze verschillende versies van de bestanden kunnen opslaan en ook toegang bieden tot deze versies.
Waarom zouden we van gecentraliseerde (server-client architectuur) versiebeheersystemen overstappen naar een gedecentraliseerde variant?	Moderne versiebeheersystemen zijn vaak het eigendom van grote tech-giganten. Binnen software ontwikkeling streven open-source developers en bedrijven naar democratische softwareontwikkeling. Gedecentraliseerde systemen kaderen goed binnen deze ideologie. Gedecentraliseerde systemen hebben ook het voordeel dat ze geen centrale server architectuur vereisen. Servers zijn immers duur en mocht de server kapot gaan is men alle gegevens kwijt ( <i>het zogenaamde SPOF probleem</i> ).
Wat zijn de eigenschappen en valkuilen van gedecentraliseerde(P2P) netwerken?	P2P netwerken bieden een manier om op een gedecentraliseerde wijze diensten en software aan te bieden. Elke peer zal hierbij de rol van zowel server als client op zich nemen om dit te bereiken. Het grootste nadeel is echter dat deze netwerken een groot aantal deelnemers vereisen om functioneel werkbaar te zijn.

Bovenstaande deelvragen bieden een goed overzicht van wat er exact nodig is om een Proof Of Concept op te stellen. De eigenlijke doelstelling van de onderzoeksvraag is het bekomen van een gedecentraliseerd versiebeheer systeem. Welke eigenschappen moet de POC voorzien om te voldoen aan deze vereisten?:

1. De oplossing die wordt ontwikkeld mag geen centrale component hebben en moet op een volledig gedistribueerde manier in staat zijn om te functioneren.
2. De oplossing moet een gebruiker in staat stellen om documenten met anderen te

Deelvraag	Antwoord
Wat is IPFS en hoe kadert het binnen versiebeheer?	IPFS is een P2P file-sharing protocol. Door IPFS kan men op een gedecentraliseerde wijze bestanden opslaan en opvragen. Aangezien versiebeheersystemen toegang willen bieden tot het delen van software versies zijn daar uiteraard ook bestanden bij betrokken. Een softwareproject is immers niets meer of minder dan verschillende bestanden. Hiervoor is IPFS dus een geschikte kandidaat.
Wat zijn smartcontracts en hoe bieden ze een meerwaarde aan Blockchain applicaties?	Smartcontracts zijn een manier om op een gedecentraliseerde wijze code uit te voeren en de data in de blockchain aan te passen. Zo kan men voorwaarden opleggen aan wijzigingen in de data. Door smartcontracts kan men bijvoorbeeld enkel de eigenaar van bepaalde data toestaan om deze data te wijzigen.

kunnen delen.

3. De oplossing moet een manier aanbieden om wijzigingen aan deze documenten op te slaan onder de vorm van verschillende versies. Deze verschillende versies moeten kunnen geraadpleegd worden.

### 3.3 Tools

Het delen van bestanden wordt mogelijk gemaakt door middel van IPFS. Dit protocol stelt de POC in staat om bestanden te verspreiden op een gedistribueerde manier en deze vervolgens opnieuw op te vragen. Hiervoor gebruikt IPFS het principe van een CID. Dit is in essentie een zelfbeschrijvende hash code die vervolgens kan worden gebruikt om bestanden op te halen -zie 2.2.2-. IPFS stelt ons ook in staat om meerdere versies van dezelfde bestand(en) op te slaan en ter beschikking te stellen. Neem bijvoorbeeld onderstaand scenario:

Bob en Alice willen een website ontwikkelen. Bob plaatst de homepage op het IPFS netwerk en krijgt een CID om dit bestand opnieuw op te vragen. Hij deelt deze CID met Alice die vervolgens het bestand lokaal binnenhaalt. Alice brengt enkele wijzigingen aan en zet het bestand opnieuw op het netwerk. Alice zal een nieuwe CID worden toegewezen. Er zijn nu in feite twee verschillende versies van dit bestand op het netwerk, een versie van Bob die kan worden opgehaald met de CID die Bob heeft en een versie van Alice met haar eigen CID.

Toch is IPFS op zich niet voldoende. Er is immers geen eenvoudige manier voorzien om deze CIDS met elkaar te delen. Hier biedt blockchain een oplossing. Blockchain functioneert immers als een databank. De CIDS kunnen daar dus worden bewaard en opgevraagd. Om dit proces van opslaan en opvragen te vereenvoudigen wordt gebruik gemaakt van smart contracts. Niet iedere blockchain ondersteunt deze smartcontracts. Ethereum is een gekend voorbeeld van een netwerk dat dit wel ondersteunt.

Deze smartcontracts worden geschreven in de programmeertaal Solidity. Veel netwerken vragen transactiekosten voor het werken met smartcontracts. Om deze transactiekosten te vermijden kan er gebruik gemaakt worden van een lokaal Ethereum netwerk. Dit lokaal netwerk voorziet dezelfde functies maar heeft geen transactiekosten. De software die hiervoor gebruik wordt is Truffle Suite (2020a).

Het ontwikkelen en testen van smartcontracts is ook niet evident. Om dit te vergemakkelijken wordt gebruik gemaakt van de Truffle Suite (2020b). Deze suite biedt de mogelijkheid om smartcontracts te testen. Hiervoor wordt gebruik gemaakt van het javascript framework Mocha.js. Truffle wordt ook gebruikt om de smartcontracts die worden ontwikkeld op de lokale blockchain uit te rollen.

Tot slot wordt er gebruik gemaakt van een console applicatie geschreven in Dotnet-core 3.1 om de verschillende elementen samen te brengen. Deze console applicatie zal onder andere de verschillende bestanden op het IPFS netwerk plaatsen en de smartcontracts op de blockchain aanspreken. Om smartcontracts te kunnen aanspreken vanuit onze console applicatie zal gebruik gemaakt worden van Juan Blanco (2020). Om IPFS aan te wenden wordt gebruik gemaakt van Richard Schneider (2019).

### 3.4 Meetbare criteria

Zoals hierboven vermeld wordt -zie 3.2- zijn er drie eigenschappen waaraan het ontwikkelde prototype moet voldoen:

1. De oplossing die wordt ontwikkeld mag geen centrale component hebben en moet op een volledig gedistribueerde manier in staat zijn om te functioneren.
2. De oplossing moet een gebruiker in staat stellen om documenten met anderen te kunnen delen.
3. De oplossing moet een manier aanbieden om wijzigingen aan deze documenten op te slaan onder de vorm van verschillende versies. Deze verschillende versies moeten kunnen geraadpleegd worden.

Hoe kunnen we deze drie eigenschappen nu concreet aftoetsen aan de ontwikkelde oplossing?

De oplossing voldoet aan de eerste vereiste als er strikt gebruik wordt gemaakt van gedecentraliseerde P2P protocollen. Deze protocollen werken namelijk per definitie zonder centraal element. Doorheen de POC wordt gebruik gemaakt van IPFS en de Ethereum blockchain. Beide zijn P2P protocollen. Zolang de console applicatie die deze twee protocollen combineert geen gebruik maakt van een centrale server wordt aldus aan deze vereiste voldaan.

Om een functioneel versiebeheersysteem te zijn moet er een manier zijn om documenten op te slaan en op te vragen. Het moet ook mogelijk zijn om deze op te vragen en de te delen met iemand anders. Kortom, er is aan deze eis voldaan als volgende drie functionaliteiten zijn geïmplementeerd:

1. Een manier om bestanden op te slaan binnen de oplossing.
2. Een manier om opgeslagen bestanden opnieuw te downloaden.
3. Een manier om deze opgeslagen bestanden te delen met iemand anders.

De laatste vereiste stelt dat er een manier moet zijn om wijzigingen aan te brengen aan de bestanden. Deze wijzigingen worden opgeslagen onder de vorm van verschillende versies. Het moet tevens mogelijk zijn om terug te gaan naar een eerdere versie van een bestand. Dat wilt concreet zeggen dat het mogelijk moet zijn om terug te grijpen naar een eerdere versie van het bestand voor een bepaalde wijziging is aangebracht. Er wordt dus aan de vereiste voldaan indien volgende functionaliteit aanwezig is:

1. Er kunnen wijzigingen worden aangebracht aan de bestanden binnen de oplossing.
2. Deze wijzigingen moeten publiekelijk toegankelijk zijn. Dat wilt concreet zeggen dat andere deze wijzigingen moeten kunnen raadplegen.
3. Er moet een mogelijkheid voorzien worden om terug te grijpen naar een eerdere versie van de bestanden.

**Indien aan al deze vereisten wordt voldaan kan men spreken van een werkbaar gedecentraliseerd versiebeheersysteem.**

Al deze vereisten kunnen worden samengevat in onderstaande tabel:

Succescriteria van de POC
De applicatie mag enkel gebruik maken van P2P protocollen. Deze zijn per definitie gedecentraliseerd. Er mag dus geen centrale component gebruikt worden.
Men moet in staat zijn om bestanden op te slaan binnen de applicatie op een gedecentraliseerde manier.
De opgeslagen bestanden moeten opnieuw gedownload kunnen worden.
Deze bestanden moeten eveneens kunnen worden gedeeld met anderen.
Er moeten wijzigingen kunnen worden aangebracht binnen deze bestanden. Deze moeten kunnen worden opgeslagen door het systeem.
Anderen die dit bestand lokaal hebben gedownload moeten in staat zijn om deze wijzigingen binnen te halen en hun eigen wijzigingen aan te brengen.
Er moet een manier voorzien worden waarop men terug kan grijpen naar een eerder versie van het bestand.

Tabel 3.1: Succescriteria van de POC.

## 3.5 Werkwijze

Dit gedeelte beschrijft het proces in grote lijnen. Om de implementatie van de verschillende werkwijzes te zien die hier worden beschreven kan men het project bekijken via volgende url: <https://github.com/michiel-schoofs/Bachelorproef/>.

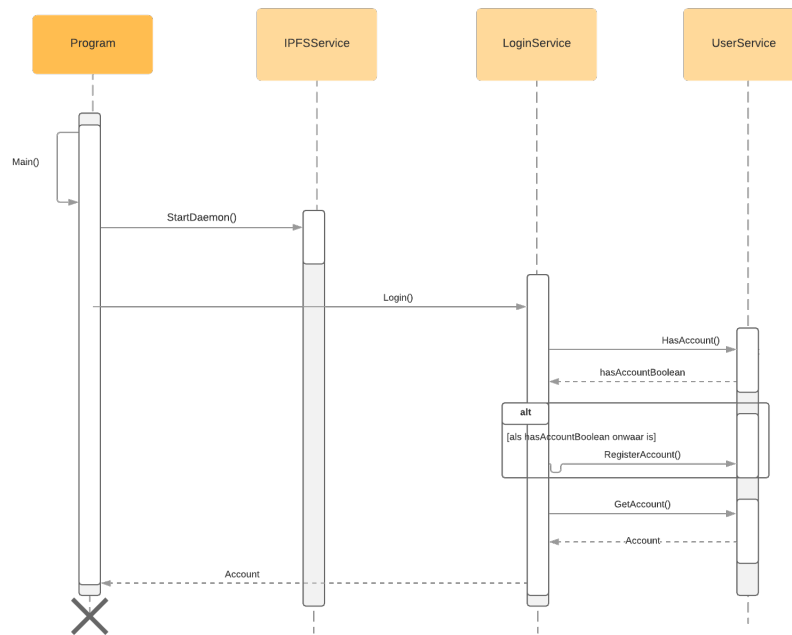
### 3.5.1 Opzet

Om gebruik te kunnen maken van IPFS en de Ethereum blockchain moet men eerst een aantal stappen doorlopen.

Voor men IPFS kan gebruiken moet men de IPFS CLI downloaden. Dit programma is beschikbaar via volgende url: <https://github.com/ipfs/go-ipfs/releases/>. Dit programma bevat de zogenaamde **IPFS daemon**. Dit is een onderdeel van IPFS dat toegang verleent tot het netwerk. De eerste stap die de Console Applicatie aldus onderneemt is om deze daemon op te starten.

Zoals ook vermeld in 3.3 wordt er gebruik gemaakt van het Ganache netwerk. Dit is een lokale Ethereum blockchain die de gebruiker in staat stelt om transacties te verrichten zonder daarbij echte transactie kosten te hoeven betalen. Dit netwerk bevat een aantal gesimuleerde test accounts met in elk test account 100 ether. Deze ether wordt gebruikt om transactiekosten te simuleren. Elk van deze accounts heeft een privé sleutel en een adres. Het zijn deze twee waarden die noodzakelijk zijn voor de interactie met het Ethereum netwerk. De console applicatie zal kijken of deze twee waarden aanwezig zijn binnen het systeem zo niet worden deze waarden opgevraagd bij de gebruiker.

Deze opzet fase kan men voorstellen door onderstaand sequentie diagram:



Figuur 3.1: Een sequentie diagram die de noodzakelijke opstartstappen overloopt.

### 3.5.2 Repositories

Binnen deze proof of concept wordt gewerkt met het concept van een Repository. Dit concept is analoog aan Repositories binnen Git. Een Repository bevat volgende elementen:

1. Een unieke naam. Deze naam wordt gebruikt om de repository opnieuw te kunnen opvragen en te identificeren.
2. De meest recente CID van de bestanden. Deze CID wordt gebruikt om de bestanden te identificeren op het IPFS netwerk. Het is dus in feite een verwijzing naar de meest recente versie van onze bestanden.
3. Een lijst van alle voorgaande CIDS. Dit is in feite een lijst van alle voorgaande versies van bestanden. Als een nieuwe versie wordt toegevoegd dan zal de vorige meest recente CID worden toegevoegd aan de lijst.
4. Een functie om nieuwe versies toe te voegen.
5. Een functie om een specifieke versie op te vragen.
6. Een functie om de meest recente versie op te vragen.

Om dit concept van repositories nog eens te verduidelijken kan gebruik worden gemaakt van volgende klassediagram:

### 3.5.3 Het opslaan van de bestanden op een gedecentraliseerde manier

Bestanden binnen dit prototype worden opgeslagen op de volgende manier:





Figuur 3.2: Een grafische voorstelling van een repository.

1. De gebruiker maakt een nieuwe repository met een opgegeven naam.
2. De gebruiker selecteert een map van bestanden.
3. Deze bestanden worden op het IPFS netwerk geplaatst. Hierdoor krijgen deze bestanden een unieke CID<sup>1</sup>. Deze CID kan gebruikt worden om later deze bestanden terug op te vragen aan het IPFS netwerk.
4. Deze CID zal vervolgens op het blockchain netwerk worden geplaatst.
5. De CID wordt vervolgens toegevoegd als meest recente versie aan de repository.

### 3.5.4 Het opvragen van bestanden op een gedecentraliseerde manier

Om bestanden op te halen wordt de volgende procedure gebruikt:

1. De gebruiker geeft de naam van het repository op.
2. Er wordt gecontroleerd of deze repository wel effectief bestaat.
3. Er wordt een referentie naar de repository opgehaald.
4. Gebruikmakend van deze referentie wordt de meest recente CID opgehaald.
5. De gebruiker geeft aan waar hij/zij wilt dat de bestanden worden gedownload.
6. De bestanden worden van het IPFS netwerk gehaald. Hiervoor wordt gebruik gemaakt van de CID. Deze bestanden worden op de aangegeven locatie gedownload.

Het delen van deze bestanden met andere gebruikers is gemakkelijk. Het volstaat immers om de naam van de Repository mee te delen aan iemand anders die de bestanden dan kan downloaden.

### 3.5.5 Het aanbrengen van wijzigingen

Het aanbrengen van wijzigingen verloopt op volgende manier:

1. De gebruiker brengt de wijzigingen aan binnen het pad dat hij/zij heeft aangegeven bij het opvragen.
2. De gebruiker geeft aan dat er wijzigingen zijn gebeurd.
3. De nieuwe bestanden worden op het IPFS netwerk geplaatst. Hierdoor wordt een nieuwe CID berekend.

<sup>1</sup> Het concept van een CID wordt besproken in 2.2.4

4. De repository waarbinnen deze bestanden zijn gewijzigd wordt op de hoogte gesteld van deze nieuwe CID. Deze zal de nieuwe CID als meest recente versie beschouwen. Hierbij wordt de vorige CID geplaatst op de lijst van voorgaande versies.

### 3.5.6 Het binnenhalen van wijzigingen

Het binnenhalen van wijzigingen kan dus via het volgend principe:

1. De gebruiker geeft aan welke repository hij/zij de wijzigingen van wilt binnen halen. Dit doet hij/zij door een naam op te geven.
2. Er wordt gecontroleerd dat deze repository al lokaal bestaat.
3. Er wordt een verwijzing opgehaald naar de repository.
4. De meest recente CID wordt opgehaald, gebruikmakende van deze verwijzing.
5. De lokale bestanden worden verwijderd.
6. De meest recente bestanden worden opgehaald op het IPFS netwerk en gedownload op dezelfde locatie waar bestanden verwijderd zijn.

### 3.5.7 Het teruggaan naar een eerdere versie

Het teruggaan naar een eerdere versie gebeurt volgens onderstaande werkwijze:

1. De gebruiker geeft aan welke repository hij/zij wilt terugzetten naar een eerder versie. Dit doet hij/zij door een naam op te geven.
2. Er wordt gecontroleerd dat deze repository al lokaal bestaat.
3. Er wordt een referentie opgehaald naar deze repository.
4. Er wordt een lijst gegeven van alle eerdere versies.
5. De gebruiker selecteert de versie naar waar men wilt terugkeren.
6. Op basis van de opgehaalde referentie wordt de corresponderende CID opgehaald.
7. De lokale bestanden worden verwijderd.
8. De bestanden van de bovenstaande CID worden opgehaald en gedownload op dezelfde locatie waar bestanden verwijderd zijn.

## 4. Conclusie

De hoofdonderzoeksvraag die deze bachelorproef behandelde, luidt als volgt:

**Hoe kan men door middel van Blockchain principes en IPFS een werkbaar gedecentraliseerd versiebeheersysteem ontwikkelen?**

Om deze vraag te beantwoorden moet men definiëren wat een werkbaar gedecentraliseerd versiebeheersysteem is. Dit werd steunend op de literatuurstudie afgebakend door middel van concrete succesfactoren. Een overzicht van deze factoren vindt men in de tabel 3.1: Succescriteria van de POC. Vertrekkende vanuit deze factoren werd vervolgens een prototype ontwikkeld door middel van IPFS en blockchain. Het ontwikkelde prototype voldeed daarbij aan elk van deze factoren.

De conclusie luidt dus als volgt: **door de werkwijze te volgen zoals uitgelijnd in de methodologie -zie 3- kan men een werkbaar versiebeheersysteem bekomen waarbij gebruik wordt gemaakt van IPFS en blockchain principes.**

Zoals ook vermeld in de inleiding was het de bedoeling dat deze oplossing een gangbaar alternatief bood voor open source bedrijven. In dat aspect is deze bachelorproef minder geslaagd. Veel van de huidige oplossingen zoals GitHub zijn namelijk gratis of vereisen een vaste kost. Doordat het prototype gebruik maakt van de Ethereum blockchain spelen transactiekosten een rol. Elke handeling die data zal schrijven op onze blockchain zoals het opslaan van nieuwe bestanden vereist aldus een betaling. Daarom lijkt deze oplossing niet commercieel haalbaar.

## 4.1 Verder onderzoek

Er zijn veel concepten en vragen die niet realistisch waren om in de gegeven tijdspanne te behandelen. Er zijn dan ook enkele interessante vervolgstudies die uit dit onderzoek kunnen voortvloeien:

1. Het Ethereum netwerk vereist transactiekosten om data aan te passen met behulp van smartcontracts. Er zijn ook netwerken die geen transactiekosten vereisen voor gebruikers van gedecentraliseerde applicaties waaronder Loom. Zijn deze netwerken een gangbaar alternatief voor Ethereum?
2. Smartcontracts hebben een sterke focus op zo klein, veilig en efficiënt mogelijk te zijn. Daarom wordt er sterk gesteund op reeds bestaande implementaties zoals Open Zeppelin. Is er een manier om zoals in klassiek programmeren verschillende Ontwerp Patronen te formaliseren die als leiddraad kunnen dienen voor de ontwikkeling van smartcontracts?

# A. Appendix

## Appendix 1: Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

### A.1 Introductie

De onderzoeksvraag waaruit wordt vertrokken luidt als volgt: "**Hoe kunnen we door middel van IPFS en Blockchain technologie versiebeheer van een Client-Server architectuur naar een gedecentraliseerde Peer-to-peer model overzetten?**" Binnen deze onderzoeksvraag zijn er vier begrippen:

- **Versiebeheer:** Git -een grote speler op het gebied van Versiebeheer- hanteert volgende definitie van het begrip "Versiebeheer is het systeem waarin veranderingen in een bestand of groep van bestanden over de tijd wordt bijgehouden, zodat je later specifieke versies kan opvragen." Deze definitie werd gepubliceerd in het boek *Pro git* (Chacon & Straub, 2014).
- **IPFS:** Interplanetary File System of (IPFS) werd in de paper "IPFS - Content Addressed, Versioned, P2P File System" geïntroduceerd door Benet (2014). Hierin stelde hij zijn technologie voor als een peer-to-peer gedistribueerd bestandssysteem waarin alle computers werken met dezelfde bestandsindeling. Hiermee wordt

bedoeld dat bestanden kunnen worden opgedeeld in verschillende delen (*ook wel 'Shards' genoemd*) en vervolgens worden opgeslagen op verschillende computers op een gezamenlijk netwerk. Vervolgens kunnen bestanden worden opgevraagd door middel van dit gezamenlijk netwerk aan te spreken. Er is dus geen gecentraliseerd aanspreekpunt.

- **Blockchain:** Blockchain is een manier om data op te slaan aan de hand van blokken. Deze blokken bevatten verschillende gegevens. Deze gegevens worden omgezet door middel van wiskundige functies die ook wel hashfuncties worden genoemd. Door de blokken onderling aan elkaar te koppelen en wiskundige functies te gebruiken bij het verifiëren van de integriteit van de blokken ontstaat er een veilige en volledig gedecentraliseerde manier van dataopslag.
- **P2P:** Tot slot is er ook het concept van Peer-to-peer (courant afgekort als P2P). Voor een gangbare definitie kan gebruik gemaakt worden van de werken van Schollmeier (2001) en IAB en Gonzalo (2009). Hierbij wordt gesteld dat P2P bestaat uit verschillende computers (nodes) die onderling met elkaar verbonden zijn. Deze nodes vervullen daarbij de rol van zowel server als client (zogenaamde Servents). Hierdoor kunnen de verschillende nodes diensten en data aan elkaar opvragen en delen zonder een centraal aanspreekpunt (client-server architectuur). Dit is dan ook het achterliggende principe van IPFS. Blockchain is een manier om gegevens en gedragsintegriteit (als gedefinieerd in (Drescher, 2017)) te waarborgen zonder centrale autoriteit binnen P2P netwerken.

Versiebeheer wordt vaak uitbesteed aan derden of zelf gedaan aan de hand van een centrale server. Het nadeel hiervan is dat er één centrale plek is waar het kan mislopen. Stel bijvoorbeeld dat de centrale server gegevens verliest is men alles kwijt. Een ander probleem is dat er binnen versiebeheer een aantal monopolies ontstaan waaronder Microsoft die GitHub kocht in 2018. Dit staat haaks op de open source beweging die streeft naar een transparante en democratische manier van software ontwikkeling. Door het introduceren van de bovengenoemde technologieën kunnen zowel de bestanden als de nodige informatie voor versiebeheer worden verspreid waardoor er geen centraal punt is en er ook geen commercieel bedrijf bij betrokken is.

De doelstelling van dit onderzoek is om versiebeheer te decentraliseren. Daaronder wordt verstaan overstappen van de klassieke server-client architectuur zoals github (of een lokaal gehost versiebeheer systeem) naar een P2P netwerk. Voor de bestanden wordt gebruik gemaakt van de reeds bestaande IPFS technologie. Hierbij zal een blockchain oplossing worden ontwikkeld om het geheel te ondersteunen (metadata, manifest bestanden,...). -Zie ook A.3 Methodologie.-

Om de onderzoeksvraag volledig te beantwoorden kan deze nog verder opgesplitst worden

in verschillende deelvragen. Deze vragen komen dan ook chronologisch aan bod om uiteindelijk tot een werkend systeem te bekomen. De verschillende deelvragen die worden behandeld zijn:

- Wat zijn de problemen van versiebeheer binnen softwareontwikkeling en hoe worden deze aangepakt door versiebeheer systemen zoals GIT?
- Waarom zouden we van gecentraliseerde (server-client architectuur) versiebeheer systemen overstappen naar een gedecentraliseerde variant?
- Wat zijn de eigenschappen en valkuilen van gedecentraliseerde (P2P) netwerken?
- Hoe gaan protocollen zoals Gnutella en BitTorrent te werk voor Peer-to-peer filescharing?
- Wat is IPFS en hoe vergelijkt het met andere gedecentraliseerde filesharing protocollen?
- Op welke manier biedt IPFS een meerwaarde ten opzichte van klassieke versiebeheer systemen?
- Wat zijn de basisprincipes van Blockchain?
- Hoe kunnen data veilig en integer worden bewaard op een Blockchain netwerk?
- Welke meerwaarde kan blockchain bieden binnen de context van Peer-to-peer netwerken?
- Wat zijn smartcontracts en hoe kunnen ze een meerwaarde bieden binnen blockchain oplossingen?
- Op welke wijze kunnen we IPFS en blockchain combineren tot een werkzaam versiebeheer systeem?

## A.2 State-of-the-art

De manier van werken is gebaseerd op het artikel “Decentralized document version control using ethereum blockchain and IPFS.” (Nizamuddin e.a., 2019) In het onderzoek wordt er gebruik gemaakt van smart contracts. Dit is in essentie code die zal uitgevoerd worden als aan bepaalde voorwaarden wordt voldaan. Deze smart contracts worden gebruikt om de verschillende aspecten van versiebeheer en data vast te leggen en uit te voeren. Voor de bestanden binnen het project wordt gekozen voor IPFS om op een gedecentraliseerde manier deze te kunnen opslaan.

De paper vormt een zeer goede aanzet en ook de werkmethode is uitvoerig beschreven. Toch blijft het zeer abstract. Belangrijke aspecten van versiebeheer worden kort of niet aangehaald waaronder “cloning”, “merging” of “branching”. In de bovengenoemde paper wordt een sterke focus op Ethereum gelegd, ontwikkeling bovenop deze blockchain interpretatie brengt echter significante overhead met zich mee. Zo is de snelheid van het systeem afhankelijk van de capaciteit en belasting van het netwerk op het gegeven moment.

Deze bachelorproef legt de focus op het ontwikkelen van een concrete toepassing. Ook de meer complexe en technische problemen zullen worden behandeld. De algemene principes van blockchain zullen vrijer worden geïmplementeerd en op een lokaal netwerk van enkele computer worden verspreid. In plaats van een grotere architectuur en implementatie te

gebruiken

### A.3 Methodologie

Er wordt vertrokken vanuit een literatuurstudie om de verschillende elementen van versiebeheer en de reeds bestaande technologieën te verkennen. Vervolgens komen de aspecten van blockchain en IPFS aan bod door middel van een demo waarin wordt gebruik gemaakt van een Word document met verschillende versies.

Tot slot worden de verschillende aspecten van versiebeheer aan de hand van een demo-applicatie geïllustreerd. Hiervoor zijn er drie hypothetische gebruikers: Alice, Bob en Carol die samen een T-shirt webshop ontwikkelen. Ze zullen hiervoor gebruiken maken van ASP.Net en Visual Studio. Binnen hun ontwikkelingsproces zullen ze een aantal gekende problemen tegenkomen waaronder “merge conflicten” en verschillende “branches”.

Bij elk van die problemen wordt er gekeken naar hoe Git -een klassiek versiebeheer systeem- dit oplost en hoe er een oplossing kan voorzien worden vanuit de voorgestelde gedistribueerde blockchain benadering. Voor het opstellen van de blockchain wordt gebruik gemaakt van C# en Nethereum (Juan Blanco, 2020). Aangezien er wordt gesteund op de IPFS API wordt er gebruik gemaakt van de open source bibliotheek net-ipfs-client-http geschreven door Richard Schneider (2019). De bedoeling is om op het einde van de bachelorproef tot een werkend prototype te komen dat gebruikt kan worden voor verschillende doeleinden.

### A.4 Verwachte resultaten

Het eindresultaat van de Bachelorproef is om op een onderbouwde manier een prototype aan te reiken om op gedecentraliseerde wijze aan versie beheer te gaan doen. De voorgestelde werkwijze wordt grondig vergeleken met Git op de volgende twee punten:

- Snelheid van een transactie: hoe lang duurt het om bewerkingen zoals pull requests en branching toe te passen op een project en/of branch?
- Performantie qua geheugengebruik: hoe efficiënt wordt er binnen de algoritmen van de oplossing omgesprongen met geheugengebruik? Hiermee wordt zowel het extern geheugen (Hardschijf, SSD) als het werkgeheugen bedoelt.

De verwachting is dat de implementatiesnelheid lager zal zijn dan met de klassieke Git-systemen, omdat blockchain van nature vrij omslachtig en intensief is. De performantie van het geheugensysteem zal eveneens slechter scoren ten opzichte van Git. De blockchain



implementatie waarborgt echter een hogere mate van data integriteit .

## A.5 Verwachte conclusies

Gedecentraliseerd versiebeheer door middel van Blockchain en IPFS is zeker technisch mogelijk. De voordelen die het biedt zijn niet alleen zuiver ideologisch. De inherente garantie op data integriteit en het weghalen van een centraal “*point of failure*” is interessant voor grote bedrijven met een groot aantal aan verschillende projecten. Er zijn echter een aantal nadelen verbonden waaronder de omslachtige procedure en het intensief gebruik van computertechnische middelen. Dit maakt het voor kleine bedrijven minder interessant.

## Appendix 2: Voorbeeld RCS

### A.1 Duiding

Dit is een voorbeeld van RCS. De meest gebruikte commando's worden gedemonstreerd (`ci`, `co`, *branching*,...). De inhoud van het centraal archief bestand `homepage.html`, v wordt ook besproken. In dit voorbeeld werken Alice en Bob samen aan twee bestanden:

- `homepage.html`: Een simpele hoofdpagina met test-tekst.
- `main.css`: Een stijlblad met daarin de stijlen van de hoofdpagina.

### A.2 Opzet van het project

Alice en Bob hebben besproken dat ze RCS zullen gebruiken. Ze maken een gezamenlijke map aan. Bob gaat van start en maakt beide bestanden aan. Hiervoor schreef hij volgende code:

#### **homepage.html:**

```
1 <!doctype html>
2 <html>
3   <head>
4     <title>My awesome cool shop</title>
5   </head>
6   <body>
7     <p>delete me</p>
8     <p>move me</p>
9     <p>Fix the typoooo</p>
10    <p> <!--A couple of paragraphs of lorem ipsum.--></p>
11  </body>
12 </html>
```

#### **main.css:**

```
1 body{
2   height:100%;
3   width:100%;
4   background-color:black;
5 }
```

Bob gaat een volledig project uitwerken met meerdere bestanden. Er zijn er op dit moment al twee aanwezig. Daarom is het interessant om deze bestanden te gaan bundelen. RCS bundelt automatisch archief bestanden samen, als er een map met de naam *RCS* bestaat. Bob maakt deze map aan door het commando `mkdir RCS`. Vervolgens zal hij de initiële versie aanmaken van beide bestanden. Hiervoor wordt er gebruik gemaakt van volgende commando's:

```
1 ci homepage.html
2 ci -r1.1 -i -m "main stylesheet for the homepage" main.css
```

<sup>1</sup>.Binnen de map zijn twee bestanden aangemaakt main.css,v en homepage.html,v. Beide bestanden zijn analoog aan elkaar. In tegenstelling tot git waar er per project een archief is heeft **elk bestand zijn eigen archief**. Wat zit er in zo een archief bestand?

#### main.css,v:

```
1 head 1.1;
2 access;
3 symbols;
4 locks; strict;
5 comment @# @;
6
7
8 1.1
9 date 2020.03.07.15.13.18; author bob; state Exp;
10 branches;
11 next ;
12
13
14 desc
15 @Main website for our shop
16 @
17
18
19 1.1
20 log
21 @Initial revision
22 @
23 text
24 @<!doctype html>
25 <html>
26   <head>
27     <title>My awesome cool shop</title>
28   </head>
29   <body>
30     <p>delete me</p>
31     <p>move me</p>
32     <p>Fix the typoooo</p>
33     <p> <!--A couple of paragraphs of lorem ipsum.--></p>
34   </body>
35 </html>
36 @
```

De verschillende eigenschappen zoals gedefinieerd door Loeliger (2012) -zie 2.1.1- zijn aanwezig in dit bestand. Op het einde van het archief bestand staat de originele code. Er is ook een logboek met zowel een versie als globale beschrijving. Tot slot is er ook metadata aanwezig waaronder de datum en auteur.

---

<sup>1</sup>De -i optie wordt gebruikt om duidelijk te maken dat het over een initiële eerste check in gaat. De -r optie wordt gebruikt om het versienummer te specificeren.

### A.3 Nieuwe versie van Alice

Alice wilt graag enkele aanpassingen maken in de bestanden van Bob. Hiervoor heeft ze de meest recente versie nodig. Het opvragen van de meest recente bestanden gebeurt bij het uitchecken. Zoals vermeld in sectie 2.1.2 speelt het concept van locks een belangrijke rol. Alice moet namelijk het bestand versleutelen als ze wijzigingen wilt aanbrengen. Hierdoor kan niemand anders het bestand aanpassen terwijl Alice haar wijzigingen nog niet heeft doorgevoerd. Het bestand versleutelen kan door de optie `-l` mee te geven bij het uitchecken. Alice opent een shell en navigeert naar de locatie van het project. Vervolgens vraagt ze een lokale kopie van de bestanden op via volgende commando's:

```
1 co -l homepage.html
2 co -l -r1.1 main.css
```

In tegenstelling tot het inchecken worden de archief bestanden niet verwijderd.

## Appendix 3: Voorbeeld Contracten

### A.1 Basis syntax van de taal

Onderstaand contracten tonen enkele van de concepten en werkwijzen in Solidity. De onderstaande code voorbeelden zijn voorzien van enkele commentaar regels ter aanvulling. Aangezien de focus in deze Bachelorproef eerder ligt op de implementatie wordt er niet uitgebreid ingegaan op Solidity. Deze voorbeeld contracten dienen dus enkel als een ondersteuning bij codevoorbeelden. Voor meer informatie raadpleeg de officiële documentatie Project, 2020. Tot slot de verschillende contracten zijn getest op de versies van Solidity zoals gedefinieerd in de **pragma** tags.

```
1 //specify the version of solidity to use
2 pragma solidity >=0.4.24;
3
4 contract Concepts {
5     //A getter is automatically created with a public state
        variable
6     string public name;
7     //The way we define a constant value, can not be changed
8     bool public constant alwaysTrue=true;
9     //Only make this visible to our contract, people can't view the
        owner but we can reference it in our contract
10    address internal owner;
11
12    //Epoch time
13    uint internal openingtime;
14
15    //Modifiers are a way of limiting when a function can execute
16    //Note overloaded Modifiers is not a thing that exists
        currently in solidity
17    modifier onlyOwner(){
18        //If the expression evaluates to false then throw exception
19        //msg is metadata that gets added to every call, so we
            verify that the person calling this function is the
            owner
20        require(msg.sender == owner, "Only the owner of this
            contract can run this command");
21        //If normal execution don't do anything
22        _;
23    }
24
25    //Allow the owner to change control
26    //Note: You cannot add optional parameters however there is PR
        to add it.
27    function changeOwner(address _newOwner) public onlyOwner {
28        owner = _newOwner;
29    }
30
31    //Make a way that our owner can change the opening time
```

```

32     function setOpeniningtime(uint _openingTime) public onlyOwner {
33         openingtime = _openingTime;
34     }
35
36     //The view modifier makes it clear we only intend this function
37     //to read data and not actually modify the state variables
38     function getOpeningtime() public view returns (uint) {
39         return openingtime;
40     }
41
42     //Enumerations also exist
43     enum ContractStatus{
44         Open, Closed
45     }
46
47     //We can use enumerations in much the same way
48     function getStatus() public view returns (ContractStatus){
49         if(block.timestamp >= openingtime) {
50             return ContractStatus.Open;
51         }
52         return ContractStatus.Closed;
53     }
54
55     modifier ContractIsOpen() {
56         //We can evaluate enumerations in this way
57         require(getStatus() == ContractStatus.Open, "the contract is
58             not opened");
59     }
60
61     // A way to represent custom datastructures
62     struct People{
63         string _firstname;
64         string _lastname;
65         uint8 _age;
66     }
67
68     //Comparable to a dictionary so a key value store
69     mapping(uint8 => People) PeopleAr;
70     //There's no way to query a mapping for the keys or count so
71     //we have to manually add those variables
72     uint8[] public ids = new uint8[](0);
73     uint8 public size = 0;
74
75     //This is how we set a value in our mapping and also make an
76     //instance of a struct
77     function addPeople(string memory _fn, string memory _ln, uint8
78         _age) public returns (uint8) {
79         //There's no lists but the concept of dynamic arrays
80         //function in much the same way
81         ids.push(size);
82         //Set a new person object with the id == size and
83         //instantiate a new Person object
84         PeopleAr[size] = People(_fn,_ln,_age);
85         //Short hand operator of size = size+1

```

```

81         size++;
82     }
83
84     //Gets called when the contract is first deployed
85     //Prevent using times because of an exploit
86     constructor(string memory _name) public {
87         //set state variable
88         name = _name;
89         owner = msg.sender;
90         openingtime = now;
91     }
92
93     //Used to demo that function overloading is a thing that exists
94     function functionToOverload(uint8 blabla) public pure returns(
95         uint8) {
96         return blabla;
97     }
98
99     //Parameters just need to be different types, uint8 and uint256
100    //is in fact another type cool huh :)
101    function functionToOverload(uint256 blabla) public pure returns
102    (uint256) {
103        return blabla;
104    }
105 }

```

## A.2 Overerving

```

1  pragma solidity >=0.5.1;
2
3  //Contract to show how inheritance works
4  contract ERC20Token{
5      string public name;
6      mapping(address=> uint256) balance;
7
8      constructor(string memory _name) public{
9          name = _name;
10     }
11
12     //Make sure we can override the method with the virtual
13     //modifier
14     function mint() public virtual {
15         //The sender is a contract so we can't use msg.sender
16         //We can use tx.origin to get the person initiaing a
17         //transaction
18         balance[msg.sender]++;
19     }
20 }
21
22 contract InheritanceContract is ERC20Token {
23     //You can't override state variabls
24     //string override name = "MyToken";

```

```

24 //You can have your own state variables inside of subclass
25 string public symbol;
26 address[] public owners;
27 uint256 public ownerCount;
28
29 //Super constructor call
30 constructor(string memory _name, string memory _symbol) public
    ERC20Token(_name) {
31     //Set property from super class
32     symbol = _symbol;
33 }
34
35 //override function mint
36 function mint() public override {
37     super.mint();
38     ownerCount++;
39     owners.push(msg.sender);
40 }
41 }

```

### A.3 Library voorbeeld

```

1 pragma solidity >=0.5.0;
2
3 //Library
4 //Promote code reuse
5 library Math {
6     //A library just stores a set of functions that we can reuse so
        we only have to maintain the code in one place
7     //A pure function basically just tells we're only using the
        variables passed into the function and no state variables
8     //Good practice to make this public
9     function devide(uint _val1, uint _val2) public pure returns (
        uint){
10         require(_val2 > 0,"");
11         return _val1/_val2;
12     }
13 }

```

```

1 contract UsingMath {
2     uint256 public value;
3
4     function calculate(uint _val1, uint _val2) public pure returns
        (uint) {
5         //call the library
6         return Math.devide(_val1,_val2);
7     }
8 }

```

### A.4 geavanceerde concepten

```

1 pragma solidity >=0.5.1;
2
3 //Contract to show how we can talk to another contract.

```



```
4 contract ERC20Token{
5     string public name;
6     mapping(address=> uint256) balance;
7
8     function mint() public {
9         //The sender is a contract so we can't use msg.sender
10        //We can use tx.origin to get the person initiating a
11        transaction
12        balance[tx.origin]++;
13    }
14 }
15
16 contract AdvancedConcepts {
17     //Payable makes sure that the wallet is valid and can receive
18     ether
19     address payable wallet;
20     address public tokenContract;
21
22     //Build in subscriber pattern, indexed fields can be filtered
23     on when subscribing
24     event SomeonePaid(
25         address indexed _buyer,
26         uint amount
27     );
28
29     constructor(address payable _owner, address _token) public {
30         wallet = _owner;
31         tokenContract = _token;
32     }
33
34     //default function that gets executed when calling this command
35     also referred to as a fallback function
36     //When the fallback function is payable then use the receive
37     keyword if not you can use the fallback keyword
38     receive() external payable {
39         sendEther();
40     }
41
42     //A way to send ether to the wallet, payable makes sure we can
43     actually attach ether to this function
44     function sendEther() public payable {
45         //transfer the attached amount of ether to the wallet
46         wallet.transfer(msg.value);
47         //Gets pushed on the log, transactions are async and we can
48         use this to listen effectively
49         // We can also use this to know when a contract function
50         got executed
51         emit SomeonePaid(msg.sender, msg.value);
52         //Get a reference to the contract and tell what it is
53         ERC20Token _token = ERC20Token(address(tokenContract));
54         //Call a function
55         _token.mint();
56     }
57 }
```

## **A.5** Nieuwe concepten vanaf 0.6.0

## Bibliografie

- 3Blue1Brown. (2017). But how does bitcoin actually work? <https://www.youtube.com/watch?v=bBC-nXj3Ng4>
- Anderson, R. (1993). The Classification of Hash Functions.
- Attiya, W. (2004, maart 11). *Distributed Computing 2e*. John Wiley & Sons. [https://www.ebook.de/de/product/3611769/attiya\\_welch\\_distributed\\_computing\\_2e.html](https://www.ebook.de/de/product/3611769/attiya_welch_distributed_computing_2e.html)
- Benet, J. (2014). IPFS - Content Addressed, Versioned, P2P File System.
- Bird, C. & Zimmermann, T. (2012). Assessing the Value of Branches with What-If Analysis, In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, Cary, North Carolina, Association for Computing Machinery. <https://doi.org/10.1145/2393596.2393648>
- Buterin, V. (2015). A Next-Generation Smart Contract and Decentralized Application Platform.
- Chacon, S. & Straub, B. (2014). *Pro Git* (2nd). Berkely, CA, USA, Apress.
- Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N. & Shenker, S. (2003). Making Gnutella-like P2P Systems Scalable, In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Karlsruhe, Germany, Association for Computing Machinery. <https://doi.org/10.1145/863955.864000>
- Desktop, N. (2018, september 21). <https://www.nobledesktop.com/blog/what-is-git-and-why-should-you-use-it>. <https://www.nobledesktop.com/blog/what-is-git-and-why-should-you-use-it>
- Drescher, D. (2017). *Blockchain Basics : A Non-Technical Introduction in 25 Steps*. New York, APRESS. <http://blockchain-basics.com/>
- Dwork, C. & Naor, M. (g.d.). Pricing via Processing or Combatting Junk Mail, In *Advances in Cryptology — CRYPTO' 92*. Springer Berlin Heidelberg. [https://doi.org/10.1007/3-540-48071-4\\_10](https://doi.org/10.1007/3-540-48071-4_10)

- Explained, S. (2018). IPFS: Interplanetary file storage! <https://www.youtube.com/watch?v=5Uj6uR3fp-U>
- Farah, S., Javed, Y., Shamim, A. & Nawaz, T. (2012). An experimental study on performance evaluation of asymmetric encryption algorithms, In *Recent Advances in Information Science, Proceeding of the 3rd European Conf. of Computer Science, (EECS-12)*.
- Fonseca, J., Reza, B. & Fjeldsted, L. (2005, april). *BitTorrent Protocol – BTP/1.0*. <http://jonas.nitro.dk/bittorrent/bittorrent-rfc.html>
- Foundation, E. (2020, mei 27). *About Ethereum.org | Ethereum.org* (E. Foundation, Red.). Ethereum Foundation. <https://ethereum.org/about>
- IAB & Gonzalo, C. (2009). Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability. RFC Editor. <https://doi.org/10.17487/RFC5694>
- IPFS. (2020a). How IPFS Deals With Files - IPFS Camp Workshop. [https://www.youtube.com/watch?v=Y\\_-TWTmF\\_1I](https://www.youtube.com/watch?v=Y_-TWTmF_1I)
- IPFS. (2020b, mei 21). *ipfs/docs* (Versie 1.0.0). <https://github.com/ipfs/docs>
- Juan Blanco. (2020, februari 13). *Nethereum* (Versie 3.7.0). <https://github.com/Nethereum/Nethereum/>
- Lievens, S. (2019, februari 5). *Probleemoplossend Denken II Lesnota's*.
- Loeliger, J. (2012, september 1). *Version Control with Git*. O'Reilly UK Ltd. <https://books.google.be/books?hl=nl&lr=&id=aM7-Oxo3qdQC&oi=fnd&pg=PR3&dq=Version+control&ots=39BeLFUfqd&sig=V5WFl33nbxhbMH1r97EwxMfmdqs#v=onepage&q&f=false>
- Microsystems, S. (2007, maart). *Sun Java System Directory Server Enterprise Edition 6.0 Deployment Planning Guide* (S. Microsystems, Red.). Verkregen 29 februari 2020, van <https://docs.oracle.com/cd/E19693-01/819-0992/fjdch/index.html>
- Nakamoto, S. (2009). Bitcoin: A Peer-to-Peer Electronic Cash System. *Cryptography Mailing list at https://metzdowd.com*.
- Nizamuddin, N., Salah, K., Azad, M. A., Arshad, J. & Rehman, M. (2019). Decentralized document version control using ethereum blockchain and IPFS. *Computers & Electrical Engineering*, 76, 183–197. <https://doi.org/10.1016/j.compeleceng.2019.03.014>
- Project, E. (2020, mei 14). *Solidity*. <https://solidity.readthedocs.io/en/v0.6.8/>
- Richard Schneider. (2019, augustus 30). *net-ipfs-http-client* (Versie 0.33.0). <https://github.com/richardschneider/net-ipfs-http-client>
- Rochkind, M. J. (1975). The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4), 364–370. <https://doi.org/10.1109/tse.1975.6312866>
- Schollmeier, R. (2001). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications, In *Proceedings First International Conference on Peer-to-Peer Computing*, IEEE Comput. Soc. <https://doi.org/10.1109/p2p.2001.990434>
- Slagter, P. (2018, mei 12). *Consensus bereiken door te stemmen over het volgende block* (P. Slagter, Red.). <https://lekkercryptisch.nl/kennisbank/artikelen/2018/05/12/Consensus-bereiken-door-te-stemmen-over-het-volgende-block>
- team, T. D. I. (2020, januari 12). *Debian GNU/Linux Installation Guide* (T. D. I. team, Red.). Verkregen 1 maart 2020, van <https://www.debian.org/releases/stable/i386/install.pdf.en>

- Thanekar, S. A., Patel, R. B. & Singh, B. P. (2010). Issues to be resolved in Torrents—Future Revolutionised File Sharing, AIP. <https://doi.org/10.1063/1.3526224>
- Tichy, W. F. (1985). RCS – A System for Version Control. *Software: Practice and Experience*, 15(7), 637–654. <https://doi.org/10.1002/spe.4380150703>
- Truffle Suite. (2020a, april 28). *Ganache | Truffle Suite* (Versie 2.4.0). <https://www.trufflesuite.com/ganache>
- Truffle Suite. (2020b, mei 23). *Truffle Suite* (Versie 5.1.27). <https://www.trufflesuite.com/truffle>
- Wang, L. & Kangasharju, J. (2013). Measuring large-scale distributed systems: case of BitTorrent Mainline DHT, In *IEEE P2P 2013 Proceedings*, IEEE. <https://doi.org/10.1109/p2p.2013.6688697>
- Warner, J. (2018, oktober 8). *Thank you for 100 million repositories* (J. Warner, Red.). <https://github.blog/2018-11-08-100m-repos/>
- Youngman, J. (2016, juni 11). *GNU CSSC* (J. Youngman, Red.). <https://www.gnu.org/software/cssc/>