HO
GENT

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

Samenvatting

Inhoudsopgave

1 Inleiding 13

1.1 Probleemstelling 13

1.2 Onderzoeksvraag 14

1.3 Onderzoeksdoelstelling 14

1.4 Opzet van deze bachelorproef 14

2 Stand van zaken 15

2.1 Versiebeheer 15

2.1.1 Inleiding 15

2.1.2 RCS 17

2.1.3 branches 21

2.1.4 Conclusie 23

3 Methodologie 25

4 Conclusie 27

A Appendix 29

A.1 Introductie 29

A.2 State-of-the-art 31

A.3 Methodologie 32

A.4 Verwachte resultaten 32

A.5 Verwachte conclusies 33

A.1 Duiding 34

A.2 Opzet van het project 34

A.3 Nieuwe versie van Alice 36

Bibliografie 37

Lijst van figuren

2.1 Overzicht types VCS 17
2.2 Overzicht concepten boomstructuur 18
2.3 Voorbeeld van deltas. 19
2.4 Voorbeeld merge proces. 21
2.5 Voorbeeld flow 22

Lijst van tabellen

2.1 Concepten binnen versiebeheer systemen. 24

1. Inleiding

De inleiding moet de lezer net genoeg informatie verschaffen om het onderwerp te begrijpen en in te zien waarom de onderzoeksvraag de moeite waard is om te onderzoeken. In de inleiding ga je literatuurverwijzingen beperken, zodat de tekst vlot leesbaar blijft. Je kan de inleiding verder onderverdelen in secties als dit de tekst verduidelijkt. Zaken die aan bod kunnen komen in de inleiding (Pollefliet, 2011):

- context, achtergrond
- afbakenen van het onderwerp
- verantwoording van het onderwerp, methodologie
- probleemstelling
- onderzoeksdoelstelling
- onderzoeksvraag
- ...

1.1 Probleemstelling

Uit je probleemstelling moet duidelijk zijn dat je onderzoek een meerwaarde heeft voor een concrete doelgroep. De doelgroep moet goed gedefinieerd en afgelijnd zijn. Doelgroepen als "bedrijven," "KMO's," systeembeheerders, enz. zijn nog te vaag. Als je een lijstje kan maken van de personen/organisaties die een meerwaarde zullen vinden in deze bachelorproef (dit is eigenlijk je steekproefkader), dan is dat een indicatie dat de doelgroep goed gedefinieerd is. Dit kan een enkel bedrijf zijn of zelfs één persoon (je co-promotor/opdrachtgever).

1.2 Onderzoeksvraag

Wees zo concreet mogelijk bij het formuleren van je onderzoeksvraag. Een onderzoeksvraag is trouwens iets waar nog niemand op dit moment een antwoord heeft (voor zover je kan nagaan). Het opzoeken van bestaande informatie (bv. “welke tools bestaan er voor deze toepassing?”) is dus geen onderzoeksvraag. Je kan de onderzoeksvraag verder specifiëren in deelvragen. Bv. als je onderzoek gaat over performantiemetingen, dan

1.3 Onderzoeksdoelstelling

Wat is het beoogde resultaat van je bachelorproef? Wat zijn de criteria voor succes? Beschrijf die zo concreet mogelijk. Gaat het bv. om een proof-of-concept, een prototype, een verslag met aanbevelingen, een vergelijkende studie, enz.

1.4 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.

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

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?

Stel volgende scenario voor: 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 door bijvoorbeeld 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 -.

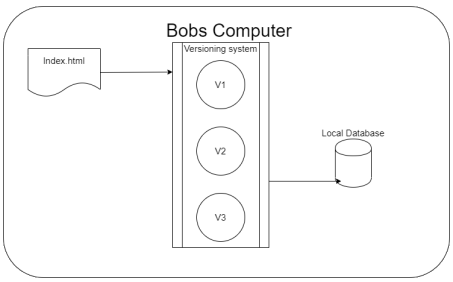
- CVCS: om samen te kunnen werken aan dezelfde bestanden kan een CVCS (Centralised Version Control System) worden gebruikt. In plaats van het archief lokaal bij te houden wordt er gebruik gemaakt van een centrale server. Bestanden worden vervolgens lokaal gekopieerd. Als er veranderingen worden aangebracht 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)

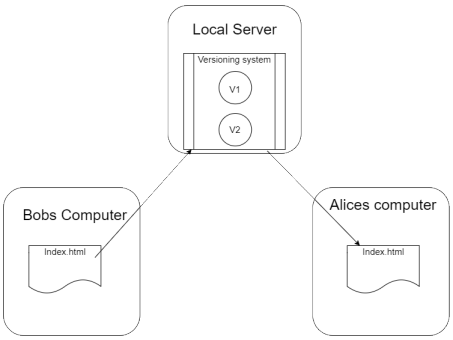
- DVCS: om het SPOF probleem te voorkomen kan men kopieën maken van het centraal archief. Deze kopieën kunnen vervolgens worden verspreid over verschillende computers. Dit is het uitgangspunt van DVCS (Distributed version control System). Elke gebruiker heeft een lokale kopie van de centrale server. 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

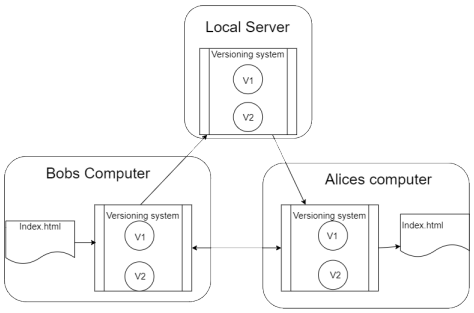
kan de gebruiker zelfs optreden als nieuwe centrale server.



(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

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 [1]- waar het als vervanging voor het CSSC Systeem (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

[1] 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/

bachelorproef.

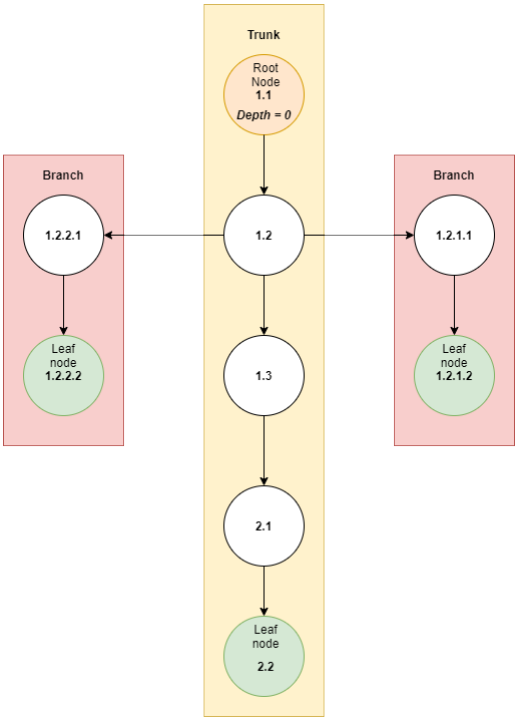
De manier waarop versies worden bijgehouden in RCS -zoals door Tichy (1985) beschreven- is gebaseerd op een boomstructuur -denk aan een stamboom-. Volgens Lievens (2019), is een boom een collectie van **toppen** *(in het Engels ook wel Nodes genoemd)*. Deze toppen hebben een hiërarchische verband.Zo bestaat er bijvoorbeeld een kind-ouder verband. Er zijn twee bijzondere toppen in een boom:

- de wortel(*root*): Deze top ligt helemaal aan het begin van de boom. Alle andere toppen zijn afstammelingen van deze top. Het heeft aldus geen ouders.
- een blad(*leaf*): Deze top heeft geen kinderen. In tegenstelling tot een wortel kunnen er meerdere bladeren aanwezig zijn.

Alle andere toppen worden intermediair genoemd. Elke top heeft mogelijk een aantal kinderen. De diepte van de wortel is nul (d=1) en elk kind heeft als diepte: Elk van deze kinderen is op zijn beurt de wortel voor een nieuwe deelboom. Het concept van **diepte** is belangrijk.

$$d_{kind} = d_{ouder} + 1 \tag{2.1}$$

Al deze concepten worden ook nog eens grafisch verduidelijkt in de figuur 2.2.



Figuur 2.2: Een overzicht van alle concepten binnen een boomstructuur waar RCS van gebruik maakt.

Er kan gebruik gemaakt worden van een boomstructuur om de onderlinge relaties tussen

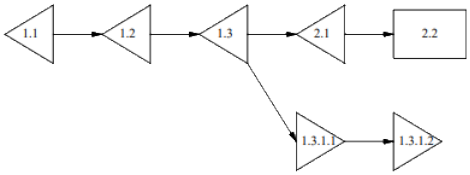
de versies weer te geven. Stel: Bob en Alice zijn bezig aan de hoofdpagina. Bob maakt een initiële versie van de pagina. Vervolgens wilt hij die graag delen met Alice. Hiervoor gebruikt hij het commando om een nieuwe versie aan te maken[2]. Dit commando wordt **inchecken** genoemd. Binnen GIT is dit 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 [3]. Bij branches is er ook nog spraken 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 [4] . 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 [5] . 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



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

[2] `ci homepage.html`
[3] `ci -r2 homepage.html`
[4] (`co homepage.html`)
[5] (`co -r1.1 homepage.html`)

deltas. Een delta houdt bij welke regels veranderd zijn ten opzichte van de vorige versie. Doordat de delta enkel de relevante lijnen bijhoudt wordt de opslag beperkt [6]. Er zijn twee types van deltas: **voorwaardse deltas** en **achterwaardse deltas**. Bij het inchecken van een nieuwe versie zal de vorige versie worden vervangen door een achterwaardse delta. Zit men momenteel op versie 1.3 en vraagt men versie 1.2 dan zal de achterwaardse delta van versie 1.2 worden toegepast op versie 1.3. (Voorwaardse 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[7]. 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 tezelfdertijd kunnen gebruiken door middel van een terminal connectie. Hierdoor kan men onderling de bestanden delen ondanks dat men niet gaat werken in een CVCS.

[6] De delta wordt opgebouwd aan de hand van het GNU commando diff https://www.gnu.org/software/diffutils/

[7] In sommige gevallen kan het slot ook worden 'geforceerd' mocht Bob bijvoorbeeld ziek vallen

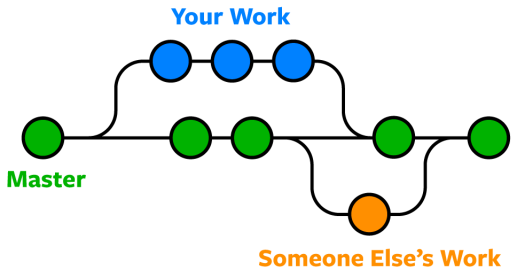
2.1.3 branches

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.

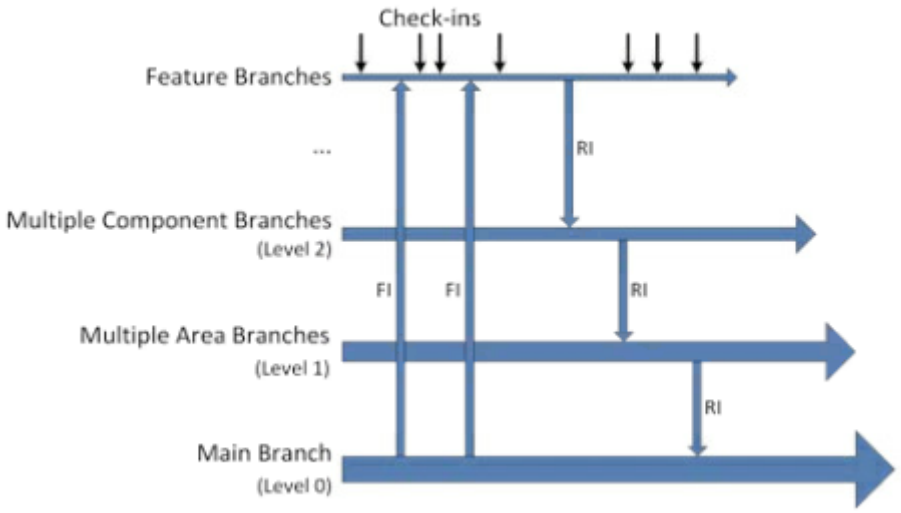


Figuur 2.4: Een voorbeeld van een merge proces. Eerst wordt er een branch aangemaakt die na drie versies terug in de master wordt gemerged. Grafiek gepubliceerd door Desktop (2018)

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 noemen we **mergen**. Dit principe wordt geïllustreerd door de figuur 2.4.

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 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.

Buiten de methodiek voorgesteld door Bird en Zimmermann (2012) bestaat er ook GIT

Flow en Trunk based development.

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 gaan 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**
Archief	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.
Versies	Elke wijziging binnen het archief leid 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 terug gedraaid naar een eerdere versie.
Logboek	Een bestand waarin alle wijzigingen worden bijgehouden. Het logboek is ook onderdeel van het archief.
Pushing	Het principe waarbij we wijzigingen die we lokaal aanbrengen synchroniseren 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 versiebeheer systemen.

3. Methodologie

4. Conclusie

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 doormiddel 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 filesharing?
- 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 (Hardeschijf, 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 weken 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:

```
<!doctype html>
<html>
        <head>
                <title>My awesome cool shop</title>
        </head>
        <body>
                <p>delete me</p>
                <p>move me</p>
                <p>Fix the typoooo</p>
                <p> <!--A couple of paragraphs of lorem ipsum.--></p>
        </body>
</html>
```

main.css:

```
body{
        height:100%;
        width:100%;
        background-color:black;
}
```

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:

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

[1].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:

```
head     1.1;
access;
symbols;
locks; strict;
comment @# @;


1.1
date     2020.03.07.15.13.18;     author bob;     state Exp;
branches;
next     ;


desc
@Main website for our shop
@


1.1
log
@Initial revision
@
text
@<!doctype html>
<html>
        <head>
                <title>My awesome cool shop</title>
        </head>
        <body>
                <p>delete me</p>
                <p>move me</p>
                <p>Fix the typoooo</p>
```

[1]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.

```
                <p> <!--A couple of paragraphs of lorem ipsum.--></p>
        </body>
</html>
@
```

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.

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 gebeurd 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:

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

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

Bibliografie

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

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/

IAB & Gonzalo, C. (2009). Peer-to-Peer (P2P) Architecture: Definition, Taxonomies, Examples, and Applicability. RFC Editor. https://doi.org/10.17487/RFC5694

Juan Blanco. (2020, januari 27). *Nethereum* (Versie 3.6.0). 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

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

Pollefliet, L. (2011). *Schrijven van verslag tot eindwerk: do's en don'ts*. Gent, Academia Press.

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. https://doi.org/10.1109/P2P.2001.990434

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

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

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/