



HoGent

Faculteit Bedrijf en Organisatie

ZFS met RAID-Z als alternatief voor klassieke RAID-oplossingen

Jonas De Moor

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

Promotor:
Antonia Pierreux
Co-promotor:
Karine Van Driessche

Instelling: —

Academiejaar: 2016-2017

Tweede examenperiode

Faculteit Bedrijf en Organisatie

ZFS met RAID-Z als alternatief voor klassieke RAID-oplossingen

Jonas De Moor

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

Promotor:
Antonia Pierreux
Co-promotor:
Karine Van Driessche

Instelling: —

Academiejaar: 2016-2017

Tweede examenperiode

Samenvatting

Voorwoord

Deze bachelorproef duidt het einde aan van mijn opleiding Toegepaste Informatica. Met deze bachelorproef wil ik bewijzen dat ik op een zelfstandige en objectieve manier onderzoek kan voeren over een (nieuwe) technologie in de IT-wereld. Omdat onze branche vrijwel continu onderhevig is aan verandering, vind ik dit een niet-onbelangrijke competentie.

Als onderwerp van deze bachelorproef heb ik besloten om een al wat oudere (maar zeker niet oninteressante) technologie te bespreken: het ZFS bestandssysteem. De redenen waarom ik net dit onderwerp zou willen bespreken, zijn nogal uiteenlopend. Ik ben zelf een enorme Linux- en UNIX-fan en ik hou ervan om mezelf nieuwe dingen aan te leren. Met ZFS was ik nog niet vertrouwd, en daar ik toch van plan ben om zelf een homeserver samen te stellen en als bestandssysteem ZFS te gebruiken, leek deze bachelorproef mij een uitstekende opportuniteit om mij wat meer te verdiepen in de werking en implementatie van deze technologie. Ook hoorde ik hier en daar geruchten vallen over de mogelijke onbetrouwbaarheid van RAID (het zgn. RAID 5 "write hole") en dit was dan ook een reden om onderzoek te voeren naar een mogelijk alternatief.

Deze bachelorproef is het resultaat van vele uren noeste arbeid; het is een werk waar ik enorm trots op ben. Desalniettemin zou dit werk niet mogelijk zijn geweest zonder de hulp van een aantal mensen. Graag neem ik daarom even de tijd om enkele personen te bedanken voor hun steun en toeverlaat gedurende deze periode.

Eerst en vooral zou ik mijn promotor, mevr. Antonia Pierreux, en mijn co-promotor, mevr. Karine Van Driessche, willen bedanken voor het goed laten verlopen van deze periode. Het schrijven van deze scriptie en het voeren van een onderzoek waren geen makkelijke karweien; zonder hun inhoudelijke en technische steun zou deze bachelorproef

nooit mogelijk geweest zijn. Tevens zou ik ook nog graag mijn familie en vrienden willen bedanken voor de onophoudelijke steun en begrip. Ook wil ik graag de mensen van DViT bedanken voor het aanreiken van technische kennis en materiaal voor mijn onderzoek. En *last but not least* wil ik mijn ouders enorm bedanken om mij gedurende deze drie jaar ten volle te steunen: dankzij hen heb ik telkens opnieuw de moed teruggevonden om er met volle teugen tegenaan te gaan in perioden dat het wat moeilijker ging.

Ik hoop dat u evenveel plezier beleeft met het lezen van mijn scriptie als ik had met het schrijven ervan.

*Jonas De Moor;
Academiejaar 2016-2017*

Inhoudsopgave

| | | |
|------------|---------------------------------------|-----------|
| 1 | Inleiding | 11 |
| 1.1 | Probleemstelling en Onderzoeksvragen | 11 |
| 1.2 | Opzet van deze bachelorproef | 12 |
| 2 | Methodologie | 13 |
| 3 | Inleiding tot RAID & ZFS | 15 |
| 3.1 | Basisbeginselen van RAID | 15 |
| 3.1.1 | Geschiedenis | 15 |
| 3.1.2 | Eigenschappen van RAID-systemen | 16 |
| 3.1.3 | RAID-niveaus: een overzicht | 16 |
| 3.2 | Inleiding tot ZFS & RAID-Z | 18 |
| 3.2.1 | Geschiedenis | 18 |
| 3.2.2 | RAID-Z | 18 |

| | | |
|-------|------------------------|----|
| 3.2.3 | Toekomst van ZFS | 19 |
|-------|------------------------|----|

4 Ontwerpprincipes & architectuur van ZFS **21**

4.1 Ontwerpprincipes **21**

| | | |
|-------|--|----|
| 4.1.1 | Eenvoud van beheer & Storage Pools | 21 |
|-------|--|----|

| | | |
|-------|----------------------------------|----|
| 4.1.2 | Consistentie & Integriteit | 22 |
|-------|----------------------------------|----|

| | | |
|-------|----------------------------------|----|
| 4.1.3 | Ingebouwde volumebeheerder | 22 |
|-------|----------------------------------|----|

4.2 De architectuur van ZFS: een overzicht **23**

| | | |
|-------|------------------------------------|----|
| 4.2.1 | Storage Pool Allocator (SPA) | 23 |
|-------|------------------------------------|----|

| | | |
|-------|----------------------------------|----|
| 4.2.2 | Data Management Unit (DMU) | 23 |
|-------|----------------------------------|----|

| | | |
|-------|-----------------------------|----|
| 4.2.3 | ZFS POSIX Layer (ZPL) | 24 |
|-------|-----------------------------|----|

| | | |
|-------|-------------------------------------|----|
| 4.2.4 | ZFS Attribute Processor (ZAP) | 25 |
|-------|-------------------------------------|----|

| | | |
|-------|----------------------------|----|
| 4.2.5 | ZFS Intent Log (ZIL) | 25 |
|-------|----------------------------|----|

| | | |
|-------|-------------------------|----|
| 4.2.6 | ZFS Volume (ZVOL) | 25 |
|-------|-------------------------|----|

5 Het opslagmodel van ZFS **27**

5.1 Structuur van het bestandssysteem **27**

5.2 Checksumming & Redundantie op blokniveau **28**

5.3 Transacties binnen ZFS **28**

6 Opzetten van een testserver voor ZFS **31**

6.1 Gebruikte hardware **31**

6.2 Installatie van Linux **31**

6.3 Installatie van ZFS **32**

6.4 Uittesten van ZFS **34**

7 Zpools & VDEV's 37

7.1 VDEV's: Virtual Devices 37

7.1.1 Concept 37

7.1.2 Speciale VDEV's 37

7.2 Storage pools of zpools 39

7.2.1 Aanmaken en beheren van zpools 39

7.2.2 Aanmaken en wijzigen van VDEV's 42

8 ZFS Datasets 47

8.1 Verschillen met traditionele bestandssystemen 47

8.1.1 Gebruik van de beschikbare opslagcapaciteit 47

8.1.2 Verschillende types van datasets 48

8.1.3 Hiërarchische structuur van ZFS datasets 48

8.2 Aanmaken & beheren van ZFS datasets 49

8.2.1 Bekijken en aanmaken van datasets 49

8.2.2 Verwijderen van datasets 54

8.2.3 Opvragen en wijzigen van properties 55

8.3 Toepassing: opzetten van een NFS-share m.b.v. ZFS datasets 56

8.3.1 Installatie van NFS 56

8.3.2 Aanmaken van een nieuwe ZFS dataset 56

8.3.3 Delen van de ZFS dataset 57

8.3.4 Toegang tot de share vanaf een client-computer 57

9 Performantieverschillen bij verschillende types RAID-opstellingen 59

| | | |
|-----------|---------------------|-----------|
| 10 | Conclusie | 61 |
| | Bibliografie | 64 |
| | Woordenlijst | 65 |

1. Inleiding

Al jarenlang is de meest gebruikte oplossing tegen dataverlies door het falen van schijven RAID, wat staat voor Redundant Array of Independent (of Inexpensive) Disks. RAID is echter niet geheel feilloos: het beschermt bv. niet tegen fouten die gemaakt zijn door gebruikers (denk maar aan het wissen van belangrijke data) en het biedt ook geen oplossing als er in een zelfde tijdsbestek meerdere schijven falen (M. Chen e.a., 1994).

In 2002 begon het toenmalige Sun Microsystems, nu onderdeel van Oracle Corporation, aan de ontwikkeling van ZFS (Zettabyte Filesystem). Dit is een bestandssysteem dat volledig *from scratch* is ontwikkeld om “*de tekortkomingen van huidige bestandssystemen op te lossen*” (Bonwick & Moore, Unknown), vooral met betrekking tot data-integriteit. Volgens de toenmalige hoofdontwikkelaar Bonwick e.a. (2002) biedt ZFS heel wat vernieuwende functionaliteiten zoals een eenvoudiger beheer, automatische foutcorrectie, automatisch schalende bestandssystemen en een softwarematige RAID onder de term *RAID-Z*.

Sinds 2013 is ZFS ook beschikbaar voor Linux, maar de eerste uitgaven leden onder nogal wat stabiliteitsproblemen. Tegenwoordig is ZFS op Linux volwassen genoeg geworden om in te zetten in productie. Deze bachelorproef is dan ook een goede gelegenheid om zelf wat meer onderzoek te doen naar ZFS en of het een volwaardig alternatief zou zijn voor een klassieke, hardwaregebaseerde RAID-oplossing op een server.

1.1 Probleemstelling en Onderzoeksvragen

Aangezien (een deel van) de RAID-functionaliteit van BTRFS nog niet als *production-ready* wordt beschouwd (The BTRFS Project, 2014, maart 4), zal ZFS worden beschouwd en geëvalueerd als volwaardig alternatief voor klassieke RAID-oplossingen op Linux-

systemen.

Deze bachelorproef zal een antwoord vinden op volgende vragen:

- Wat zijn de grootste verschillen tussen een klassieke RAID-oplossing en ZFS RAID-Z?
- Hoe is de architectuur van ZFS opgebouwd en op welke manieren tracht het oplossen te vinden voor de problemen die zich voordoen bij andere bestandssystemen en RAID-opstellingen?
- Hoe staat het met data-integriteit en performantie¹ bij ZFS onder verschillende workloads en toepassingen?

1.2 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te formuleren op de onderzoeksvragen.

In Hoofdstuk 3 wordt er een korte inleiding gegeven op de geschiedenis en de algemene werking van RAID-systemen. Ook ZFS en RAID-Z worden reeds kort toegelicht.

In Hoofdstuk 4 wordt er een globaal overzicht gegeven van de architectuur en ontwerpprincipes van ZFS. In de daaropvolgende hoofdstukken worden de belangrijkste onderdelen en functionaliteiten wat meer uitgediept.

In Hoofdstuk 5 wordt het opslagmodel van het ZFS-bestandssysteem besproken. Onder andere de datastructuur en het transactiemodel van ZFS komen aan bod.

In Hoofdstuk 6 worden de stappen die moeten worden ondernomen om een desktopcomputer om te zetten naar een Linux-server die kan worden gebruikt voor ZFS besproken.

In Hoofdstuk 7 worden zpools en VDEV's wat meer in detail belicht. Tevens wordt er gedemonstreerd hoe men zpools en VDEV's aanmaakt en wijzigt.

In Hoofdstuk 10, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

¹Met 'performantie' wordt het aantal I/O's per seconde en de globale schijf- en CPU-belasting bedoeld.

2. Methodologie

3. Inleiding tot RAID & ZFS

In dit hoofdstuk wordt er een korte inleiding gegeven over de basisprincipes van RAID, aangezien RAID-Z een softwarematige vorm van RAID is. Daarnaast wordt de geschiedenis en globale werking van ZFS reeds kort besproken.

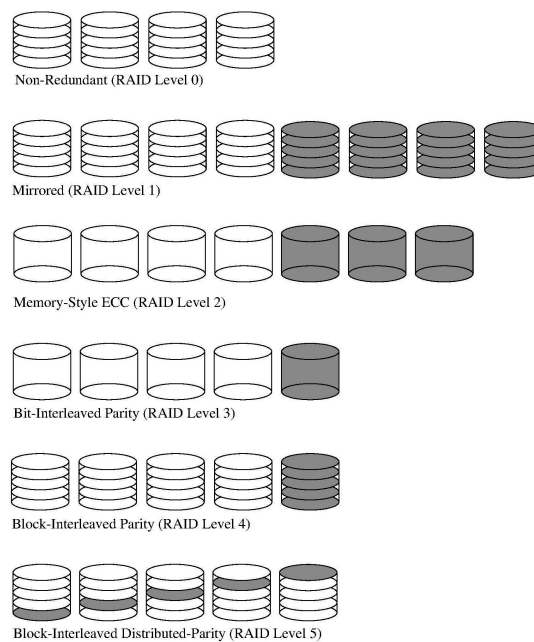
3.1 Basisbeginselen van RAID

3.1.1 Geschiedenis

Al van oudsher worden magnetische harde schijven gebruikt als opslagmedium voor data (Goda & Kitsuregawa, 2012). Maar reeds in de jaren 80 zagen onderzoekers in dat I/O-performance een bottleneck zou vormen voor computersystemen in de toekomst: terwijl geheugenchips en processoren steeds sneller werden, bevonden opslagmedia zich in een impasse (A. Paterson, 1987).

In de paper "*A Case for Redundant Arrays of Inexpensive Disks*" formuleerden A. Paterson (1987) en zijn collega's voor het eerst de term 'RAID', wat een acroniem is voor 'Redundant Arrays of Inexpensive Disks'. De oorspronkelijke idee achter RAID was dat een verzameling van goedkopere schijven performanter zou zijn dan grotere en duurdere mainframeschijven van die tijd.

Naast performantie en kost was betrouwbaarheid (reliability) ook een belangrijke factor. Als bv. één of meerdere schijven van de array falen, dan mag dit geen invloed hebben op de werking van de rest van de verzameling schijven. Daarom introduceerden de onderzoekers de zgn. "RAID levels" (A. Paterson, 1987), die vandaag de dag nog steeds in gebruik zijn. Er bestaan een aantal RAID-niveaus, waarvan de voornaamste zullen besproken worden.



Figuur 3.1: Illustratie van verschillende RAID-niveaus (M. Chen e.a., 1994)

3.1.2 Eigenschappen van RAID-systemen

Bij het bouwen van RAID-systemen worden er gebruikelijk drie aspecten in beschouwing genomen: **performantie** (performance), **betrouwbaarheid** (reliability) en **capaciteit** (capacity). Een RAID-niveau is in principe niets anders dan een balans tussen deze verschillende eigenschappen; meestal zullen er dus één of meerdere trade-offs moeten gemaakt worden (M. Chen e.a., 1994).

Begrippen die centraal staan bij RAID zijn *striping* en *parity*. Striping heeft betrekking op de manier waarop een RAID-controller (hetzij hardwarematig, hetzij softwarematig) de blokken data verdeelt over de array van schijven. Bij RAID 0 bijvoorbeeld wordt de data gelijkmatig gedistribueerd volgens het *round-robin*-algoritme (Arpaci-Dusseau & Arpaci-Dusseau, 2015). Datablokken die verdeeld zijn over meerdere schijven en samen één geheel vormen worden een *stripe* genoemd.

3.1.3 RAID-niveaus: een overzicht

RAID 0

Een voordeel bij RAID 0 is dat de gehele capaciteit van de schijven kan gebruikt worden; er gaat geen ruimte verloren, aangezien de data gelijkmatig verdeeld wordt over de array. Een bijkomend voordeel van striping is dat performantie in het algemeen goed is (Arpaci-Dusseau & Arpaci-Dusseau, 2015) : de meeste en reads en writes kunnen parallel worden afgehandeld. Een voorwaarde voor goede performantie is echter wel dat de *chunk size* (de

grootte van de blokken data die worden weggeschreven en/of uitgelezen) ook optimaal wordt gekozen, i.e. afhankelijk van de workload op het systeem (Arpaci-Dusseau & Arpaci-Dusseau, 2015). Toch kent RAID 0 een groot nadeel: betrouwbaarheid is nagenoeg onbestaande. Aangezien data nergens wordt gedupliceerd, betekent dat het falen van eender welke disk leidt tot verlies van data (Arpaci-Dusseau & Arpaci-Dusseau, 2015).

RAID 4

Parity is een mechanisme dat werd geïntroduceerd bij RAID-niveau 4 om betrouwbaarheid af te dwingen. Bij RAID 4 wordt er metadata over de opgeslagen data bijgehouden in parity blocks op een aparte schijf. Deze metadata wordt verkregen d.m.v. het uitvoeren van een mathematische functie op de opgeslagen data. Meestal is dit een XOR-functie (exclusieve OF) (M. Chen e.a., 1994). Aan de hand van deze parity kan bij het verlies van één of meerdere schijven de originele data worden gereconstrueerd door XOR'ing toe te passen op de parity bits en de data bits. Bij een XOR-operatie geven een even aantal enen (1) steeds als resultaat nul (0); omgekeerd geldt ook dat een oneven aantal enen (1) steeds een één (1) als resultaat zullen opleveren. Stel dat één schijf van een array van vier schijven faalt, dan kan nog steeds de originele data worden verkregen. Echter, als er meer dan één schijf verloren gaat, dan is het bij RAID 4 onmogelijk om de originele data te herstellen. Het grote voordeel van RAID 4 is dan weer echter dat er minder wordt ingeboet op capaciteit dan bij bv. RAID 1 en RAID 5 (Arpaci-Dusseau & Arpaci-Dusseau, 2015).

RAID 1

Naast RAID 0 en RAID 4 zijn er nog andere RAID-levels, zoals RAID 1 en RAID 5. RAID 1 staat ook bekend als *mirroring*, omdat het kopieën maakt van de datablokken naar één of meerdere disks afhankelijk van het aantal schijven. Op gebied van capaciteit is RAID 1 niet echt gunstig, aangezien maar de helft van de totale schijfruimte bruikbaar is. Stel dat er vier schijven in een array aanwezig zijn, dan is slechts de opslagcapaciteit van twee schijven bruikbaar. Daarentegen is de betrouwbaarheid van RAID 1 wel vele malen beter dan die van RAID 0: in theorie mogen er bij een reeks van n schijven $\frac{n}{2}$ schijven falen. Maar dan mogen de schijven die elkaars mirror zijn niet falen, want dan is de data op deze disks verloren. Daarom houdt men in de praktijk meestal de maatstaf van één schijf aan (Arpaci-Dusseau & Arpaci-Dusseau, 2015).

RAID 5

Als laatste wordt RAID 5 besproken. RAID 5 is in principe niets anders dan RAID-niveau 4, maar dan uitgebreid met functionaliteit dat de parity blocks roteert over de verschillende schijven. Dit is een groot verschil t.o.v. RAID 4, waarbij de parity blocks zich op één disk bevinden. Read-performantie is nagenoeg gelijk aan RAID 4, maar write-performantie is stukken beter. Dit komt omdat bij RAID 5 de schrijfoperaties parallel kunnen worden afgehandeld; bij RAID 4 vormt de parity-schijf een bottleneck bij het wegschrijven van data (M. Chen e.a., 1994). De reden hiervoor is dat bij het updaten van data ook de parity blocks moeten worden geüpdatet; alle operaties worden dus m.a.w. serieel uitgevoerd

(Arpaci-Dusseau & Arpaci-Dusseau, 2015).

Er bestaan nog andere, niet-standaard RAID levels, zoals RAID 6 en RAID 10, maar deze worden hier niet besproken.

3.2 Inleiding tot ZFS & RAID-Z

3.2.1 Geschiedenis

ZFS is een bestandssysteem dat ontwikkeld is door het toenmalige Sun Microsystems, nu onderdeel van Oracle Corporation. Voormalig Sun-werknemer Jeff Bonwick was de oorspronkelijke hoofdontwikkelaar van het bestandssysteem. De ontwikkeling van ZFS startte aan het begin van de jaren 2000; Sun had reeds met de ontwikkeling van bestandssystemen geëxperimenteerd, maar deze pogingen mislukten telkens (Bonwick e.a., 2015). ZFS is een *copy-on-write* (COW) bestandssysteem (Hickmann & Shook, 2007): bij elke aanpassing van een datablok, wordt het datablok in kwestie niet aangepast, maar wordt het gekopieerd naar een nieuwe locatie en aangepast (Lucas & Jude, 2015).

In die tijd gebruikte het bedrijf haar UNIX-besturingssysteem Solaris intern voor verschillende soorten toepassingen, waaronder file servers (Bonwick e.a., 2015). De schijven van deze servers waren opgedeeld in volumes, beheerd door de Solaris Volume Manager (SVM) en geformatteerd in UFS (UNIX Filesystem) (Bonwick e.a., 2015). Toen een verkeerd ingevoerd SVM-commando erin slaagde om het systeem te doen crashen en voor een enorme *downtime* zorgde bij Sun, was dit voor Bonwick een aanleiding om volledig *from scratch* een bestandssysteem op te bouwen dat makkelijk in gebruik én beheer was.

De hoofdrede om een volledig nieuw bestandssysteem te ontwikkelen was volgens Bonwick en Moore (Unknown) dat de toenmalige oplossingen voor bestandssystemen totaal achterhaald waren. Bestandssystemen waren nog ontwikkeld voor opslagnoden uit de jaren '80 en '90, welke niet te vergelijken zijn met de huidige noden van zowel bedrijven als particulieren. Naarmate de nood aan meer opslagruimte steeg, moesten er oplossingen worden bedacht, en dit m.b.v. bestandssystemen die hier helemaal niet op voorzien waren. Een 'tussenoplossing', aldus volgens Bonwick en Moore (Unknown), die ook vandaag de dag nog gebruikt wordt, is de combinatie van volumes en bestandssystemen. Volumes zijn abstracties voor (delen van) fysieke schijven.

3.2.2 RAID-Z

Het ZFS-bestandssysteem omvat ook een softwarematige RAID, RAID-Z genaamd. Volgens Bonwick (2005) lost RAID-Z in combinatie met ZFS een aantal problemen op die inherent aanwezig zijn bij andere RAID-implementaties. Zo claimen de ontwikkelaars dat RAID-Z het zogenaamde RAID 5 *write hole* probleem volledig oplost. Bij klassieke RAID-oplossingen worden stripes weggeschreven op een niet-atomaire manier, i.e. de bewerkingen worden niet als één geheel uitgevoerd. Een bijkomend probleem hierbij is

dat bij RAID-levels met parity (zoals RAID 5) ook nog eens de parity-blocks moeten herberekend worden. Indien het systeem tussen deze bewerkingen crasht of uitvalt door bv. elektriciteitsproblemen, dan bevindt de data op de array van schijven zich in een inconsistente toestand: stripes en parity blocks kunnen corrupt raken (Bonwick, 2005).

3.2.3 Toekomst van ZFS

Ondertussen werd Sun Microsystems overgenomen door Oracle in 2010, na het faillissement van deze eerstgenoemde (Oracle Corporation, Onbekend). In 2010 richtten enkele ex-ontwikkelaars van Solaris het illumos-project op met de bedoeling om een volledig open source variant van OpenSolaris te ontwikkelen; ook ZFS werd verder ontwikkeld als onderdeel van illumos (illumos, 2012, augustus 20). De reden hiervoor was dat Oracle geen nieuwe uitgaven meer uitbracht voor OpenSolaris. In 2013 werd het OpenZFS-project opgericht, dat zichzelf als de *"ware en open opvolger van het oorspronkelijke ZFS-project"* ziet (The OpenZFS Project, 2014).

Mede dankzij dit project is ZFS ondertussen beschikbaar op verschillende platformen, waaronder FreeBSD, Linux en macOS.

4. Ontwerpprincipes & architectuur van ZFS

In dit hoofdstuk worden enkele principes besproken waarop de ontwikkelaars zich hebben gebaseerd bij het ontwerp en de ontwikkeling van ZFS. Tevens wordt de architectuur van ZFS globaal beschreven, om zo de ontwerpbeslissingen van de ontwikkelaars wat meer toe te lichten.

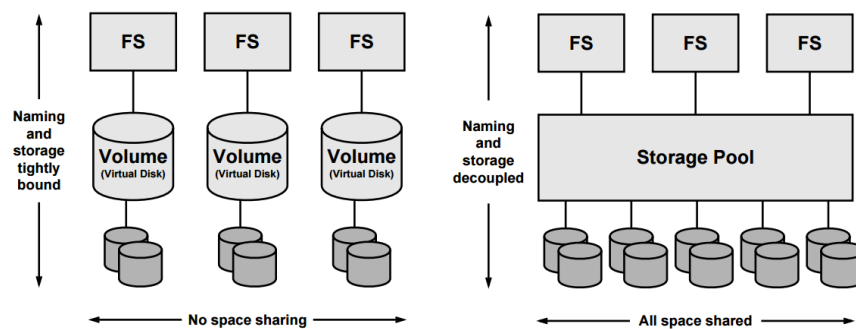
4.1 Ontwerpprincipes

De principes die aan de basis van ZFS liggen vloeiden meestal voort uit de problemen die de ontwikkelaars zelf ervaarden bij het gebruik van andere bestandssystemen.

4.1.1 Eenvoud van beheer & Storage Pools

Volgens Bonwick e.a. (2002) kan en moet het aanmaken en beheren van bestandssystemen een stuk makkelijker gemaakt worden. Hierbij speelt automatisatie van verschillende taken een belangrijke rol. Ook moet het mogelijk zijn om beheerderstaken (zoals bestandssystemen aanmaken en verwijderen) uit te voeren zonder de werking van het gehele systeem te ondermijnen (Bonwick e.a., 2002).

Een niet onbelangrijke feature hierbij zijn storage pools. Storage pools hebben als doel om opslagruimte zoveel mogelijk los te koppelen van de fysieke schijven: alle schijven bevinden zich in een pool van disks. Uit deze pool kunnen bestandssystemen worden aangemaakt, zonder rekening te moeten houden met de limitaties van bv. partities. Tevens kunnen bestandssystemen op een flexibele manier gebruik maken van deze pooled storage door automatisch in te krimpen en uit te breiden wanneer nodig (Bonwick e.a., 2002).



Figuur 4.1: Illustratie van ZFS pooled storage (rechts) t.o.v. volume-based storage (links) (Bonwick e.a., 2002)

4.1.2 Consistentie & Integriteit

Eén van de taken van een bestandssysteem is om ervoor te zorgen dat het in een consistente toestand blijft, i.e. het moet mogelijk zijn om van inconsistente toestanden te herstellen d.m.v. bijvoorbeeld een journal of door een filesystem check te draaien (Arpaci-Dusseau & Arpaci-Dusseau, 2015). Een nadeel aan deze technieken volgens Bonwick e.a. (2002) is dat deze niet makkelijk zijn om te implementeren, omdat bijvoorbeeld in het geval van journaling filesystems een roll back of roll forward moet worden uitgevoerd. Hierbij moet ook nog de volgorde van de bewerkingen in acht worden genomen (Arpaci-Dusseau & Arpaci-Dusseau, 2015).

De oplossing volgens Bonwick e.a. (2002) is om het bestandssysteem te allen tijde consistent te houden; er mag m.a.w. geen enkel moment zijn waarop het systeem in een inconsistente toestand kan terecht komen. Dit wordt verwezenlijkt door de Copy-On-Write eigenschappen van ZFS, waardoor bewerkingen atomair gebeuren (Li, 2009).

Een ander aspect waar ZFS een antwoord op tracht te vinden, is het voorkomen en minimaliseren van datacorruptie. Volgens Bonwick e.a. (2002) is het niet verstandig om volledig te vertrouwen op hardware, omdat er steeds een kans is dat deze bugs bevat. Dit kan leiden tot *silent data corruption*: dit is een fenomeen waarbij de schijf corrupte datablokken niet kan detecteren (Arpaci-Dusseau & Arpaci-Dusseau, 2015). Om deze reden bevat ZFS een uitgebreid checksumming-mechanisme dat werkt op blokniveau. In Hoofdstuk 5 wordt er dieper ingegaan op deze technieken.

4.1.3 Ingebouwde volumebeheerder

Indien bijvoorbeeld LVM (Logical Volume Manager) wordt gebruikt op een Linux-systeem voor het aanmaken van volumes, dan staan de volumes relatief "los" van de bestandssystemen die worden aangemaakt op deze volumes: eerst maakt men volumes aan, nadien pas bestandssystemen (Lewis, 2006).

Bonwick e.a. (2002) is van mening dat een volledige integratie van de volumebeheerder

in het bestandssysteem een aantal interessante voordelen met zich meebrengt, zoals betere optimalisaties en betere consistentie, omdat de volume manager nu "weet" hoe de bestandssystemen in de storage pool(s) zijn opgebouwd.

4.2 De architectuur van ZFS: een overzicht

De architectuur van ZFS is opgebouwd uit een zestal componenten (Li, 2009): de Storage Pool Allocator (SPA), de Data Management Unit (DMU), een ZFS POSIX Layer (ZPL), de ZFS Attribute Processor (ZAP), de ZFS Intent Log (ZIL) en ZFS Volume (ZVOL). Elk onderdeel biedt een bepaalde functionaliteit aan en levert diensten aan boven- en onderliggende lagen.

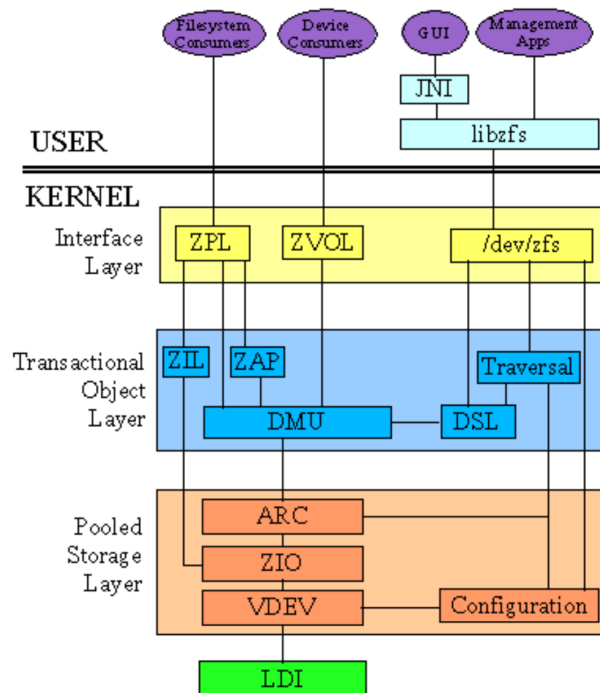
4.2.1 Storage Pool Allocator (SPA)

De taak van de SPA is hoofdzakelijk om datablokken van verschillende apparaten in één pool te verzamelen; de functionaliteit van de SPA komt m.a.w. in grote mate overeen met die van een volume manager. Het grote verschil tussen een volume manager en de SPA is dat de SPA louter een *interface* is voor virtuele datablokken; dit in tegenstelling tot een volume manager, dat volumes doorgeeft aan het besturingssysteem (Bonwick e.a., 2002).

De Storage Pool Allocator biedt een interface aan tot **data virtual addresses (DVA's)**: dit zijn de adressen van de virtuele datablocks die zich in een storage pool bevinden. De datablokken die zich fysiek op een schijf bevinden worden dus niet rechtstreeks aangesproken; alle communicatie van bovenliggende lagen gericht tot de fysieke disks moet eerst via de SPA. Hierdoor worden de ontwerpprincipes van '*Eenvoud van beheer*' en '*Storage Pools*' gerealiseerd: schijven kunnen dynamisch worden toegevoegd zonder de werking van de rest van het systeem te ondermijnen. Ook kan de systeembeheerder op een flexibele manier bestandssystemen aanmaken, zonder rekenschap te moeten geven aan de onderliggende fysieke structuur (Bonwick e.a., 2002).

4.2.2 Data Management Unit (DMU)

De Data Management Unit is de laag boven de SPA en is a.h.w. de 'lijm' tussen de nogal *low-level* (virtuele) datablokken en de bovenliggende lagen. Deze component vertaalt de blokken naar ZFS objecten; dit is nodig omdat ZFS alles voorstelt als objecten. Zo stelt een ZFS-object van het type DMU_OT_ACL een access control list voor (Sun Microsystems, 2006). Objecten behoren steeds tot een bepaalde naamruimte (*namespace*) of dataset; dit verhoogt de flexibiliteit m.b.t. het beheer van bestandssystemen aanzienlijk. Zo wordt een bestandssysteem binnen ZFS voorgesteld door objecten uit een bepaalde dataset. Aangezien objecten van elkaar gescheiden zijn door private namespaces, leven bestandssystemen (die uit een reeks van ZFS-objecten bestaan) ook naast en onafhankelijk van elkaar. Dit maakt taken zoals aanmaken en verwijderen van filesystems een stuk makkelijker voor zowel de gebruiker als de DMU (Bonwick e.a., 2002).



Figuur 4.2: Een overzicht van de verschillende componenten van ZFS (Kendi, Onbekend)

Daarnaast staat de DMU ook garant voor de consistentie van alle data. Om deze reden worden transacties binnen ZFS geïmplementeerd door de Data Management Unit als COW-transacties (Copy-On-Write). Hierbij wordt gebruik gemaakt van een 'alles of niets'-aanpak: ofwel is de transactie gelukt, ofwel is de transactie mislukt en moeten er back-upblokken worden aangewend (Bonwick e.a., 2002).

4.2.3 ZFS POSIX Layer (ZPL)

Deze laag vertaalt objecten komende van de Data Management Unit naar met POSIX compatibele bestandssystemen (Bonwick e.a., 2002). POSIX (acroniem voor Portable Operating System Interface) is een reeks van standaarden waaraan ontwikkelaars zich moeten houden indien ze willen dat hun programma's *portable* (overdraagbaar) zijn. Dit betekent dat het weinig tot geen moeite zou moeten kosten voor een ontwikkelaar om een programma draaiende te krijgen op bv. verschillende UNIX-systemen (IEEE and The Open Group, 2016). Solaris en macOS zijn voorbeelden van 'POSIX-compliant' besturingssystemen (The Open Group, Onbekend).

Voor het uitvoeren van bewerkingen maakt de ZPL gebruik van de onderliggende Data Management Unit. Het aanmaken van een bestandssysteem gebeurt bijvoorbeeld door de ZFS POSIX Layer: de ZPL maakt enkele DMU-objecten met metadata e.d. aan en maakt gebruik van het transactiemodel van de DMU om deze objecten weg te schrijven. Dankzij deze aanpak verlopen deze bewerkingen ook atomair (Bonwick e.a., 2002).

4.2.4 ZFS Attribute Processor (ZAP)

De ZFS Attribute Processor is een component die samenwerkt met de DMU-laag. De ZAP manipuleert ZAP-objecten: dit zijn speciale DMU-objecten die metadata over allerlei dingen bijhouden in de vorm van key-value pairs. Deze informatie handelt over verschillende andere objecten, zoals datasets en filesystem objects. Voor een grote hoeveelheid attributen (en dus informatie) wordt er gebruikgemaakt van zgn. fatzaps; indien de hoeveelheid informatie nogal gering is, dan kiest ZFS voor microzaps (Sun Microsystems, 2006).

4.2.5 ZFS Intent Log (ZIL)

Deze component houdt logbestanden bij van alle transacties voor het geval dat het bestandssysteem toch in een inconsistente toestand zou terecht komen, bijvoorbeeld bij een stroompanne. Op deze manier kunnen transacties worden gereplayed indien consistentie van uiterst belang is. Deze records worden bijgehouden in het RAM-geheugen tot ze worden gepersisteerd door een DMU-transactie of door een fsync (Bonwick e.a., 2002).

4.2.6 ZFS Volume (ZVOL)

ZFS Volumes of ZVOL's zijn objecten die binnen ZFS worden voorgesteld als een koppel van een property object en een dataobject. Deze logische volumes worden dan aan het OS doorgegeven als ordinaire UNIX block devices en kunnen ook op deze manier benaderd worden (Sun Microsystems, 2006).

5. Het opslagmodel van ZFS

In dit hoofdstuk worden de interne bestandssysteemoperaties van ZFS wat meer toegelicht. Hierbij wordt vooral de werking van de Storage Pool Allocator en de Data Management Unit dieper uitgespit, aangezien deze componenten het beheer van data voor zich nemen.

5.1 Structuur van het bestandssysteem

Datablokken worden in ZFS voorgesteld als een boomstructuur. Vele andere bestandssystemen, zoals BTRFS op Linux, gebruiken ook boomstructuren voor het bijhouden van data (The BTRFS Project, 2017). De algemene werking van ZFS en andere, boomgebaseerde bestandssystemen verschillen niet zo heel veel. Echter zijn er ook eigenschappen die uniek zijn aan de manier waarop ZFS de data opslaat.

De belangrijkste elementen van een boom in de informatica zijn de volgende: de wortel (Eng.: *root*), de knopen (Eng.: *nodes*) en de bladeren (Eng.: *leaves*) (Cohen, g.d.). Indien men bij de wortel van een ZFS-boom start, dan komt men als eerste de *überblock* tegen: dit is simpelweg een andere benaming voor de wortel van de boom. Dit datablok bevat een checksum van zichzelf en verwijst naar de andere datablokken in de boom. Afhankelijk van de sectorgrootte van een schijf, bevat een ZFS pool een zeker aantal *überblocks* (een schijf waarvan de sectoren 512 bytes groot zijn geeft 128 *überblocks* als resultaat) (Lucas & Jude, 2015).

Zoals reeds gezegd in Hoofdstuk 4, worden alle blokken gechecksummed om corruptie van data tegen te gaan. Deze checksums worden bewaard in het datablok zelf en in de ouder van dit blok: op deze manier is de hele ketting van datablokken gechecksummed en kan er makkelijk schade worden vastgesteld en kan deze schade eventueel worden

gerepareerd. De *überblock* is het enige datablok die geen ouder heeft, en dus bewaart deze zijn checksum bij zichzelf (Bonwick e.a., 2002).

5.2 Checksumming & Redundantie op blokniveau

Checksumming is een goede oplossing om corruptie te detecteren. Echter moeten er nog steeds back-upblokken aanwezig zijn om van datacorruptie te herstellen. Indien men bijvoorbeeld mirroring toepast binnen een pool, dan kan ZFS de corrupte datablokken zelf herstellen door een goede kopie van de andere schijf van de mirror te halen. Als een applicatie een aanvraag doet voor een reeks datablokken waarvan er één of meerdere beschadigd zijn, dan repareert ZFS deze datablokken automatisch; nadien worden de herstelde blokken doorgegeven aan de applicatie, zonder dat deze laatstgenoemde hier iets van merkt (Bonwick e.a., 2002).

Maar zelfs zonder gebruik te maken van mirror VDEV's kan ZFS in sommige gevallen van corruptie herstellen. Hiervoor maakt ZFS gebruik van zogenaamde *ditto blocks*: dit zijn simpelweg kopieën van belangrijke datablokken, zoals van metadata of pool data. Deze blokken worden zo ver mogelijk uit elkaar op de schijf geplaatst, om zo de impact van datacorruptie te minimaliseren (Lucas & Jude, 2015).

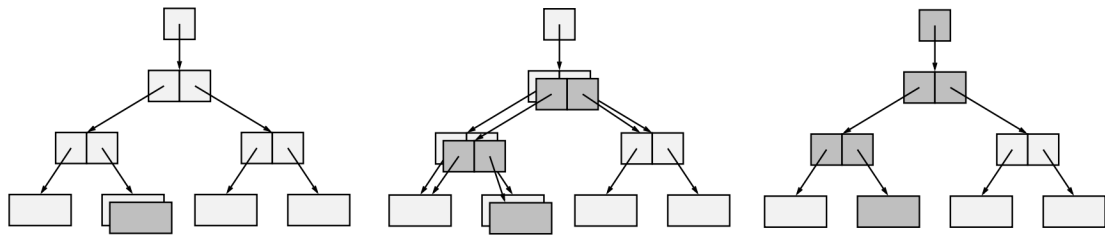
5.3 Transacties binnen ZFS

ZFS is een transactioneel bestandssysteem (Bonwick e.a., 2002). Transacties worden binnen ZFS afgehandeld door de DMU¹ en verlopen volgens een Copy-On-Write (COW) manier. Bij Copy-On-Write worden blokken nooit direct aangepast of overschreven; in plaats daarvan kopieert ZFS deze blokken eerst naar een andere locatie, om deze dan vervolgens aan te passen. Deze manier van werken garandeert dat data op de schijf of schijven steeds consistent blijft: als er zich bijvoorbeeld een stroompanne voordoet midden in een transactie, dan kan ZFS makkelijk terugvallen op de oude boomstructuur, aangezien deze niet wordt overschreven (Lucas & Jude, 2015).

Uit de paper van Bonwick e.a. (2002) kan men de volgende stappen opmaken die worden ondernomen bij het uitvoeren van een transactie:

1. De ZFS POSIX Layer (ZPL) krijgt de opdracht om een reeks van blokken aan te passen, te creëren of te verwijderen. Stel dat er in dit geval één enkel datablok moet worden aangepast en dat dit blok zich in een blad van de boom bevindt. De ZPL groepeerde alle wijzigingen in een transaction group (txg); hierbij moet de ZPL de wijzigingen goed groeperen opdat het bestandssysteem aan het einde van de transactie in een consistente toestand wordt achtergelaten. Voor het uitvoeren van de transactie doet de ZPL beroep op de DMU.

¹De DMU behandelt eigenlijk enkel objecten, maar voor de eenvoud wordt er in dit hoofdstuk meestal gesproken over blokken.



Figuur 5.1: Boomstructuur bij aanpassing van één datablok binnen een ZFS-transactie. Van links naar rechts: (1) Aanpassen van het datablok; (2) Aanpassen van de indirecte blokken; (3) Overschrijven van de überblock. Alle bewerkingen gebeuren op een COW-manier. (Bonwick e.a., 2002)

2. De DMU start de transactie en doorzoekt de boom. De DMU heeft het blok in kwestie gevonden, kopieert het naar een nieuwe locatie en past het vervolgens aan (Copy-On-Write). Na deze aanpassing moet ook de checksum van het datablok worden herberekend.
3. Aangezien andere blokken in de boom direct of indirect gekoppeld zijn aan het aangepaste datablok, moeten deze ook worden aangepast, en dit op dezelfde manier als het reeds aangepaste datablok. Ook moeten de checksums van deze blokken herberekend worden. Dit is een recursief gebeuren, en stopt bij de wortel van de boom, nl. de überblock.
4. Om de COW-eigenschappen van ZFS te garanderen, wordt het überblock niet direct aangepast, maar wordt er geroteerd naar de volgende beschikbare überblock. De oude en nieuwe boomstructuren worden dus a.h.w. met elkaar omgewisseld, en dit op een atomaire, transactionele manier.

Indien het systeem zou crashen tijdens deze transactie, dan kan de überblock corrupt raken. Aangezien corruptie kan worden gedetecteerd met behulp van checksums, kan er eenvoudig worden teruggevallen op een oudere versie van een überblock (Bonwick e.a., 2002).

6. Opzetten van een testserver voor ZFS

In dit hoofdstuk wordt de procedure besproken die werd ondernomen bij het omvormen van een desktopcomputer tot een volwaardige Linux-server die kan gebruikt worden voor ZFS. Nadien worden de stappen besproken voor de installatie van ZFS op deze server.

6.1 Gebruikte hardware

Voor deze scriptie wordt er een HP Pavilion Elite HPE-310be desktopcomputer gebruikt voor te experimenteren met ZFS en het uitvoeren van de testen. Bij het kiezen van een systeem werd er zoveel mogelijk rekening gehouden met de aanbevelingen van de OpenZFS-ontwikkelaars waar mogelijk¹. Wat verder volgt er een overzicht van de specificaties van het gekozen systeem.

6.2 Installatie van Linux

De Linux-distributie die wordt gebruikt doorheen deze scriptie is Fedora 25 Server Edition. De belangrijkste redenen om voor Fedora te kiezen, zijn de volgende:

- Het beschikt over een relatief recente Linux kernel en recente packages;
- Het is relatief eenvoudig om OpenZFS te installeren op Fedora;
- De distributie is eenvoudig te installeren

¹Het gebruikte geheugen beschikt niet over ECC-errorcorrectie; ECC-functionaliteit wordt door de OpenZFS-ontwikkelaars aangeraden om datacorruptie te voorkomen (The OpenZFS Project, 2017).

| Specificaties | |
|------------------|--|
| Fabrikant | HP |
| Model | HP Pavilion Elite HPE-310be |
| CPU | Intel Core i5 650 @ 3.2 GHz (2 Cores; 4 Threads) |
| Geheugen | 10GB DDR3 @ 1333MHz |
| GPU | AMD Radeon HD 5570 |
| Interne schijven | SAMSUNG HD103SJ (1TB) |
| | WDC WD1002FAEX-0 (1TB) |
| | WDC WD5000AZRX-0 (500GB) |
| Externe schijf | WD Elements 1078 (1TB) |
| RAID Controller | Intel Corporation SATA RAID Controller |

Tabel 6.1: Specificaties van het systeem dat gebruikt wordt doorheen deze bachelorproef (data verkregen via `lshw`)

Fedora wordt geïnstalleerd op een externe harde schijf via USB: dit om de interne schijven zoveel mogelijk vrij te houden voor ZFS en het uitvoeren van testen. Op de volgende pagina bevindt zich een overzicht van de schijfindeling die gehanteerd wordt.

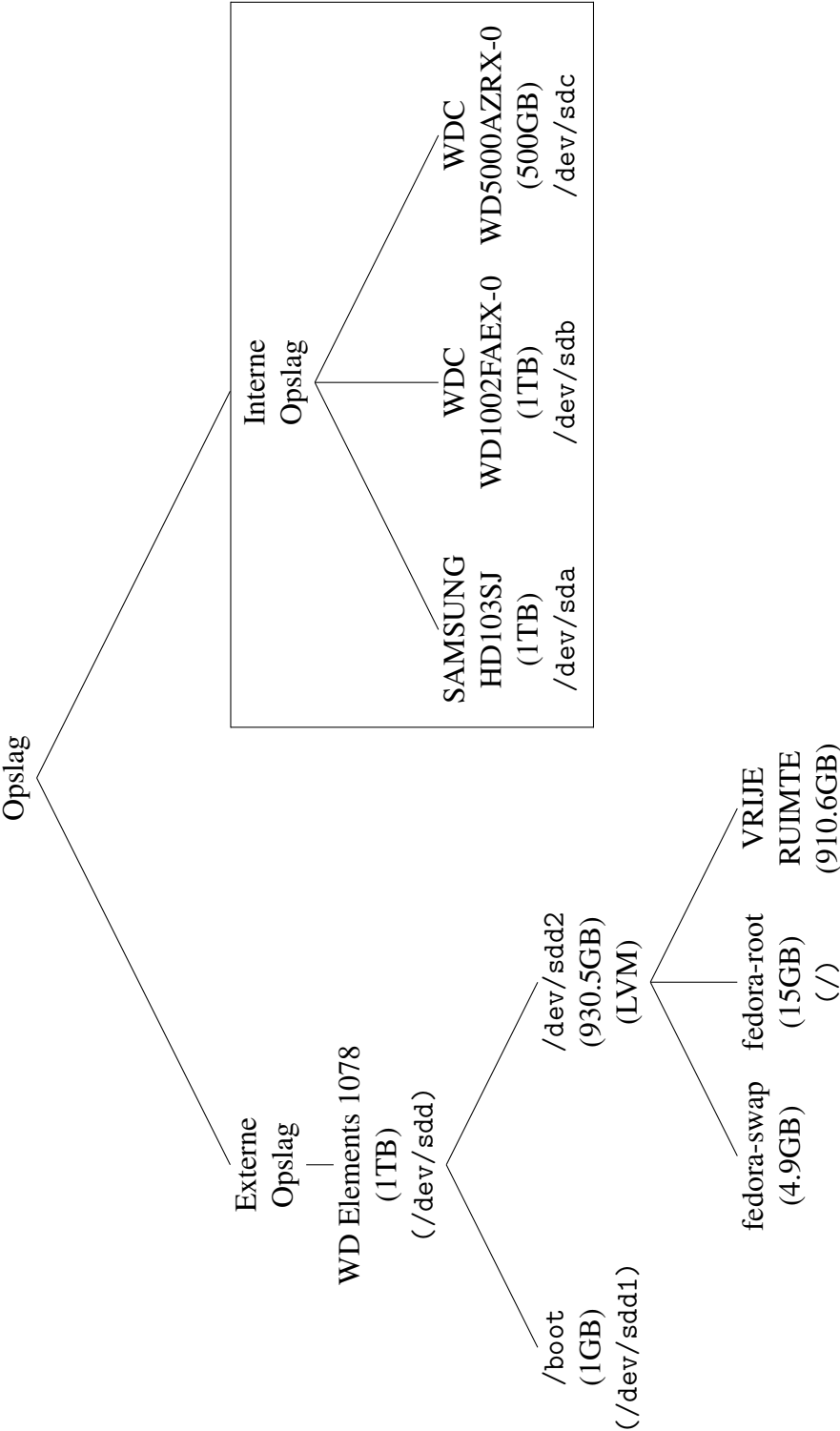
Na de installatie worden er nog enkele kleine dingen aangepast, zoals het uitzetten van de Cockpit-service en het aanpassen van de SSH-daemon (PermitRootLogin wordt op 'No' gezet). De firewall is reeds goed ingesteld: er moeten geen aanpassingen worden gemaakt.

Na het herstarten van de SSH-daemon is de server klaar voor gebruik en kan er worden ingelogd via SSH.

6.3 Installatie van ZFS

Voor de installatie van OpenZFS wordt er gebruik gemaakt van de documentatie en repositories van ZFS On Linux, een zusterproject van OpenZFS dat de installatie en het gebruik van OpenZFS op Linux vergemakkelijkt (ZFS On Linux, 2017). Het installeren van ZFS is vrij eenvoudig: eerst voegt men de nodige repositories toe, nadien installeert men de nodige packages; tenslotte moeten nog de nodige kernel modules worden ingeladen. Men kan de kernel modules inladen via `modprobe` of men kan simpelweg de server herstarten.

Bij het installeren van ZFS werden nauwgezet de stappen gevolgd uit de GitHub wiki van ZFS On Linux (2016). Alle commando's worden uitgevoerd als root, tenzij anders aangegeven.



Figuur 6.1: Illustratie van de gehanteerde disk-layout van het systeem. De ingekaderde schijven zullen gebruikt worden door ZFS.

Eerst en vooral worden de nodige packages geïnstalleerd. Deze packages voegen een .repo-bestand toe aan /etc/yum.repos.d/: dit maakt het voor DNF (de package manager van Fedora) mogelijk om deze repository te gebruiken als installatiebron.

```
$ dnf install http://download.zfsonlinux.org/fedora/zfs-release$(
    rpm -E %dist).noarch.rpm
```

Nadien moet de GPG key van ZFS On Linux worden gecontroleerd. Indien nodig moet eerst GnuPG worden geïnstalleerd. De vingerafdruk van de geïnstalleerde sleutel zou moeten overeenkomen met C93A FFFD 9F3F 7B03 C310 CEB6 A9D5 A1C0 F14A B620.

```
$ dnf install gnupg
$ gpg --quiet --with-fingerprint /etc/pki/rpm-gpg/RPM-GPG-KEY-
    zfsonlinux
```

Nadat de repository is opgezet, worden de benodigde packages voor ZFS geïnstalleerd. De kernel modules voor ZFS worden gecompileerd via DKMS.

```
$ dnf install kernel-devel zfs
```

Na installatie moet de ZFS kernel module in de draaiende kernel worden ingeladen en moeten de benodigde services worden geactiveerd.

```
$ modprobe zfs
$ systemctl enable zfs.target && systemctl start zfs.target
$ systemctl enable zfs-import-cache && systemctl start zfs-import-
    cache
$ systemctl enable zfs-mount && systemctl start zfs-mount
```

6.4 Uittesten van ZFS

Op dit moment is ZFS geïnstalleerd en zou het naar behoren moeten werken. In deze sectie worden enkele manieren aangereikt om na te gaan of ZFS correct geïnstalleerd is.

Een eerste vereiste voor ZFS is dat de kernel module in de huidige kernel moet ingeladen zijn. Dit kan eenvoudig worden nagegaan met het volgende commando:

```
$ lsmod | grep "zfs"
zfs                2699264    0
zunicode           331776     1 zfs
zavl               16384      1 zfs
zcommon            49152      1 zfs
znvpair            77824      2 zcommon,zfs
spl                98304      3 znvpair,zcommon,zfs
```

lsmod geeft een lijst van alle modules die op dit moment in de kernel zijn geladen. De output van lsmod wordt doorgestuurd ("gepipet") naar het commando grep; dit commando filtert de lijnen waarin het woord 'zfs' voorkomt en print deze af op het scherm. De output zou er ongeveer zoals hierboven moeten uitzien.

Een tweede vereiste voor het correct functioneren van ZFS is dat de nodige services moeten geactiveerd zijn. Op een Linux-systeem dat systemd gebruikt als init-systeem en service supervisor kan dit worden nagegaan met het commando `systemctl status`. De targets en unit files die zeker moeten geactiveerd zijn, zijn `zfs.target`, `zfs-import-cache.service` en `zfs-mount.service`. Hieronder bevindt zich een voorbeeld.

```
$ systemctl status zfs-mount.service
• zfs-mount.service - Mount ZFS filesystems
  Loaded: loaded (/usr/lib/systemd/system/zfs-mount.service;
         enabled; vendor
  Active: active (exited) since Mon 2017-05-01 12:35:43 CEST; 2h
         43min ago
  Process: 1731 ExecStart=/sbin/zfs mount -a (code=exited, status
         =0/SUCCESS)
  Main PID: 1731 (code=exited, status=0/SUCCESS)
  Tasks: 0 (limit: 4915)
  CGroup: /system.slice/zfs-mount.service

May 01 12:35:43 SRV-ZFS systemd[1]: Starting Mount ZFS filesystems
...
May 01 12:35:43 SRV-ZFS systemd[1]: Started Mount ZFS filesystems.
```

Als laatste kan er worden nagekeken of twee beheerscommando's van ZFS, `zpool` en `zfs`, naar behoren werken.

```
$ zpool status
no pools available

$ zfs list
no datasets available
```

De uitvoer van deze commando's is normaal, aangezien er nog geen pools en bijgevolg dus ook geen datasets zijn aangemaakt.

7. Zpools & VDEV's

In dit hoofdstuk worden zpools en diens bouwstenen, VDEV's, wat meer toegelicht. Er wordt ook een demonstratie gegeven over hoe men zpools en VDEV's aanmaakt en wijzigt.

7.1 VDEV's: Virtual Devices

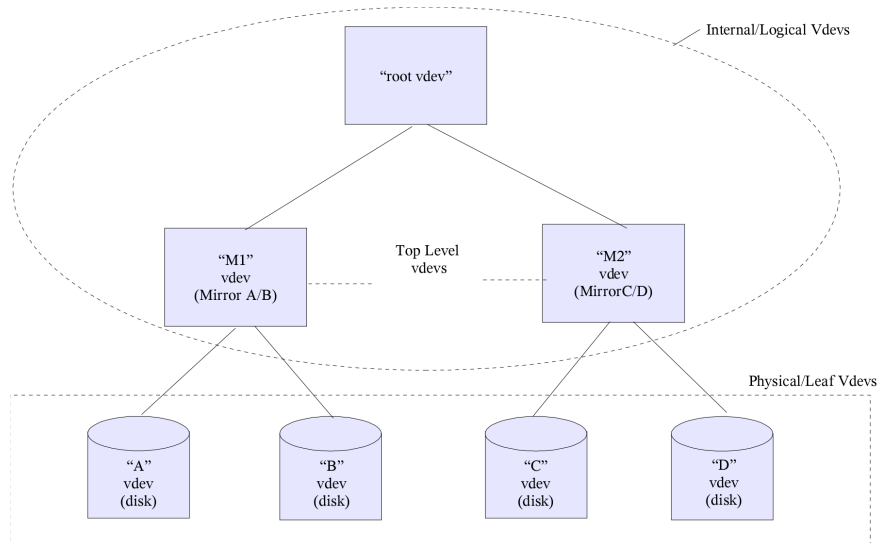
7.1.1 Concept

VDEV's (of voluit Virtual Devices) zijn de bouwstenen van storage pools; het zijn een soort van device drivers die elk een bepaalde functionaliteit aanbieden. Er bestaan verschillende soorten VDEV's: zo zijn er bijvoorbeeld striping VDEV's en mirror VDEV's. Ook worden de verschillende RAID-Z-vormen geïmplementeerd met behulp van één of meerdere VDEV's (Bonwick e.a., 2002).

Conceptueel worden virtual devices voorgesteld in een boomstructuur, waarvan de bladeren de fysieke VDEV's voorstellen; deze VDEV's komen overeen met fysieke apparaten zoals harde schijven. De andere knopen in de boom worden logische VDEV's genoemd omdat ze een groepering vormen van fysieke VDEV's. Zoals in elke boom bestaat er ook in deze structuur van virtual devices een wortel of *root*. De kinderen van deze root VDEV worden de top-level VDEV's genoemd (Sun Microsystems, 2006).

7.1.2 Speciale VDEV's

Naast VDEV's voor redundantie en striping bestaan er ook nog andere virtual devices die een specifieke rol vervullen. Deze VDEV's worden ingezet met als hoofddoel de



Figuur 7.1: Conceptuele voorstelling van VDEV's in een boomstructuur: M1 en M2 zijn mirrors VDEV's; apparaten A t.e.m. D zijn kinderen van deze VDEV's (Sun Microsystems, 2006).

performantie van ZFS naar omhoog te trekken (Lucas & Jude, 2015).

SLOG: Seperate Intent Log

Onder normale omstandigheden bevindt de ZFS Intent Log (ZIL) zich in een pool en worden de bewerkingen die op dit moment bezig zijn naar deze log geschreven. Echter kan de systeembeheerder deze log naar een apart, snel apparaat (zoals een SSD) buiten een pool verplaatsen om zo de performantie omhoog te trekken. ZFS kan logdata efficiënter groeperen in batches in plaats van deze weg te schrijven in de volgorde dat de opslagbewerkingen gebeuren; daarna worden deze weggeschreven naar de pool. Dit verhoogt de gehele efficiëntie en snelheid van bewerkingen aanzienlijk. Vele databanksystemen wachten bijvoorbeeld totdat data gepersisteerd is naar de schijf alvorens verder te gaan met een volgende operatie. ZFS kan deze soort operaties loggen en achteraf uitvoeren; het bestandssysteem meldt terwijl aan de databank dat deze operatie gepersisteerd is (Lucas & Jude, 2015).

L2ARC: Level 2 Adaptive Replacement Cache

Traditioneel gebruiken schijven en bestandssystemen buffers of caches om veelgebruikte data tijdelijk bij te houden; bestandssystemen gebruiken het RAM geheugen om blokken data tijdelijk bij te houden (Arpaci-Dusseau & Arpaci-Dusseau, 2015). ZFS is hierop geen uitzondering en cachet ook data in het werkgeheugen. Naast deze vorm van caching beschikt ZFS ook over de Adaptive Replacement Cache (ARC). Dit geeft de mogelijkheid om een snel apparaat (zoals een SSD) te gebruiken als extra cache om veelgebruikte bestanden in op te slaan. Het grote verschil met deze manier van bufferen en het gebruik

van caches in het RAM-geheugen, is dat in de L2ARC (Level 2 ARC) enkel bestanden worden bijgehouden die frequent gebruikt worden, maar dan weer niet frequent genoeg om in het werkgeheugen te worden bijgehouden (Lucas & Jude, 2015).

7.2 Storage pools of zpools

Zoals reeds gezegd in Hoofdstuk 4 vormen storage pools (of zpools in ZFS-termen) de eerste vorm van abstractie binnen de gehele ZFS-stack: zpools bieden namelijk een interface aan tot de onderliggende fysieke schijven. Het zijn echter VDEV's die ervoor zorgen dat data kan weggeschreven worden van en naar de schijven. Ter verduidelijking kan er een analogie worden gemaakt met een traditionele RAID-controller en de schijven in een RAID-array: zpools vervullen de rol van RAID-controller, terwijl VDEV's de schijven van de 'array' voorstellen. De zpool ('RAID-controller') verdeelt of stripet data over één of meerdere VDEV's ('schijven') (Lucas & Jude, 2015).

7.2.1 Aanmaken en beheren van zpools

Het testsysteem waarover we beschikken bevat drie interne harde schijven, elk direct aangesloten op het moederbord met een SATA-kabel. Dit aantal is voldoende om een RAID-Z1 opstelling mee te maken. Vooraleer er echter VDEV's kunnen worden toegevoegd of gewijzigd, moet er eerst een ZFS storage pool worden aangemaakt; er kunnen meerdere storage pools worden aangemaakt, maar in ons geval is één storage pool meer dan voldoende.

Bekijken van aanwezige pools

Voor het beheer van pools wordt er gebruik gemaakt van het commando `zpool`. Om bijvoorbeeld de huidige zpools te bekijken, geeft men het volgende commando in:

```
$ zpool list
no pools available
```

Aanmaken van een nieuwe pool

Op dit moment zijn er nog geen pools aangemaakt, dus de uitvoer van bovenstaand commando is normaal. Om een zpool aan te maken met de drie schijven waarover we beschikken, gebruik je het volgende commando:

```
$ zpool create storage /dev/sda /dev/sdb /dev/sdc
```

Deze instructie maakt een zpool aan met de naam 'storage'. Vervolgens kan je een lijst opvragen van alle pools die op het systeem aanwezig zijn:

```
$ zpool list
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH
storage  2.27T  154K   2.27T        -       0%    0%   1.00x  ONLINE
```

(deel van de uitvoer is weggelaten)

De uitvoer van dit commando geeft reeds enkele eigenschappen van de pool weer, zoals de naam, de totale grootte van de pool, de gebruikte ruimte van de pool, de vrije ruimte en de hoeveelheid fragmentatie. Een andere eigenschap die interessant kan zijn, is DEDUP of deduplicatie: indien er verschillende kopieën zijn van een stuk data, dan houdt ZFS deze maar één keer bij.

Gezondheid van pools

Om de gezondheid en structuur van een pool na te kijken, gebruik je het commando `zpool status`:

```
$ zpool status
pool: storage
state: ONLINE
scan: none requested
config:

      NAME      STATE      READ WRITE CKSUM
storage  ONLINE          0     0     0
   sda      ONLINE          0     0     0
   sdb      ONLINE          0     0     0
   sdc      ONLINE          0     0     0

errors: No known data errors
```

Dit commando geeft een overzicht van de interne structuur en gezondheid van elke pool op het systeem, samen met de aanwezige VDEV's. Het merendeel van de uitvoer spreekt voor zich: 'pool' geeft de naam van de pool weer en 'state' geeft de algemene toestand van een pool weer. De eigenschap 'scan' geeft aan of er een zogenaamde scrub wordt uitgevoerd of uitgevoerd is geweest. Een scrub is een scan die kan worden uitgevoerd door ZFS om de consistentie en integriteit van de pool na te gaan; indien mogelijk worden fouten automatisch gerepareerd. Een scrub is dus in principe de equivalent voor een `fsck` binnen ZFS. De vijf kolommen onder de eigenschap 'config' geven informatie over de VDEV's van de pool weer; de drie kolommen aan de rechterzijde geven het aantal fouten aan dat door een bepaald VDEV werd gedetecteerd.

Eigenschappen van pools

Zoals reeds gezegd in Hoofdstuk 4 is ZFS grotendeels een objectgeoriënteerd bestandssysteem. Elk object binnen ZFS heeft bepaalde eigenschappen (properties); deze kunnen dan

ook worden opgehaald en gewijzigd. Het is dan ook niet verwonderlijk dat zpools tevens objecten zijn, met elk bepaalde eigenschappen.

Om bijvoorbeeld alle eigenschappen van een pool op te halen, gebruikt men het commando `zpool get all <naam van de pool>`:

```
$ zpool get all storage
NAME      PROPERTY      VALUE      SOURCE
storage   size          2.27T      -
storage   capacity      0%         -
storage   altroot       -          default
storage   health        ONLINE     -
storage   guid          2498162094782357460 default
storage   version       -          default
storage   bootfs        -          default
storage   delegation    on         default
storage   autoreplace   off        default
storage   cachefile     -          default
storage   failmode      wait       default
storage   listsnapshots off        default
```

(deel van de uitvoer is weggelaten)

Indien men een eigenschap van een pool wilt wijzigen, gebruikt men het commando `zpool set <eigenschap>=<waarde> <naam van de pool>`:

```
$ zpool set comment="Testpool" storage
$ zpool get comment storage
NAME      PROPERTY  VALUE      SOURCE
storage   comment   Testpool   local
```

In bovenstaand voorbeeld werd de property 'comment' aangepast; vervolgens werd de nieuwe waarde opgehaald.

Verwijderen van een pool

Om een zpool en diens VDEV's te verwijderen, gebruik je het commando `zpool destroy <naam van de pool>`:

```
$ zpool destroy storage
$ zpool list
no pools available
```

7.2.2 Aanmaken en wijzigen van VDEV's

In het begin van dit hoofdstuk werd er reeds kort gesproken over VDEV's en hun rol bij zpools: alle redundantie bij RAID-Z zit in feite in deze VDEV's. Het zijn de virtual devices (en dus niet de storage pools) die het maken van een RAID-opstelling binnen ZFS mogelijk maken (Lucas & Jude, 2015).

De eigenschappen en principes van de verschillende soorten redundantie VDEV's komen grotendeels overeen met de overeenkomstige standaard RAID-niveaus; deze werden reeds uitvoerig besproken in Hoofdstuk 3.

Vooraleer er overgegaan wordt tot het aanmaken van VDEV's, moet er worden opgemerkt dat de structuur van zpools en van de meeste VDEV's na creatie vast ligt. Zpools kunnen uitgebreid worden met meer schijven en/of partities en VDEV's, maar aan een VDEV kan men in de meeste gevallen geen apparaten toevoegen (The FreeBSD Documentation Project, 2017).

Striping VDEV's

Striping VDEV's komen overeen met RAID-niveau 0; bij zpools bestaande uit striping VDEV's wordt data gelijkmatig verdeeld over de verschillende VDEV's.

In de voorgaande sectie over zpools werden er bij het aanmaken van een nieuwe zpool impliciet striping VDEV's gebruikt:

```
$ zpool create storage /dev/sda /dev/sdb /dev/sdc
$ zpool status
pool: storage
state: ONLINE
scan: none requested
config:

    NAME        STATE        READ  WRITE  CKSUM
    storage     ONLINE       0     0     0
      sda       ONLINE       0     0     0
      sdb       ONLINE       0     0     0
      sdc       ONLINE       0     0     0

errors: No known data errors
```

In dit geval wordt elke fysieke harde schijf in een aparte VDEV gestopt en wordt de data gelijkmatig verdeeld over de drie VDEV's (sda, sdb en sdc).

Mirror VDEV's

Bij mirror VDEV's zijn de kinderen van de VDEV's kopieën van elkaar: dit is hetzelfde principe dat bij RAID 1 wordt toegepast.

Mirrors zijn overigens de enige VDEV's waaraan die men na creatie nog kan wijzigen, sa-

men met stripes. Aan een mirror VDEV kunnen achteraf nog schijven worden toegevoegd; een striping VDEV kan worden geüpgradet tot een mirror VDEV door een extra schijf of schijven aan de VDEV toe te voegen (The FreeBSD Documentation Project, 2017).

We beschikken bijvoorbeeld over de volgende storage pool met één striping VDEV, bestaande uit één fysieke schijf:

```
$ zpool create storage /dev/sda
$ zpool status
pool: storage
state: ONLINE
scan: none requested
config:

    NAME      STATE    READ  WRITE CKSUM
    storage   ONLINE      0     0     0
      sda     ONLINE      0     0     0

errors: No known data errors
```

Deze striping VDEV kan makkelijk worden geüpgradet naar een mirror VDEV m.b.v. het commando `zpool attach <naam pool> <naam VDEV> <schijf>`:

```
$ zpool attach storage sda sdb
$ zpool status
pool: storage
state: ONLINE
scan: resilvered 59.5K in 0h0m with 0 errors on Tue May 9
      17:16:23 2017
config:

    NAME      STATE    READ  WRITE CKSUM
    storage   ONLINE      0     0     0
  mirror-0   ONLINE      0     0     0
      sda     ONLINE      0     0     0
      sdb     ONLINE      0     0     0

errors: No known data errors
```

Bij het weergeven van de structuur van de pool, kan men zien dat de stripe werd geüpgradet naar een mirror, met sda en sdb als kinderen. Ook werd er een scrub uitgevoerd na het toevoegen van de nieuwe schijf.

RAID-Z1 VDEV's

Aangezien het testsysteem over slechts drie fysieke schijven beschikt, kunnen we niet verder gaan dan RAID-niveau 5¹. Het equivalent van RAID 5 bij ZFS is RAID-Z1.

¹In theorie zou er kunnen gebruikt gemaakt worden van binary files als storage-backend; ZFS kan namelijk bestanden gebruiken als provider. Maar aangezien deze bestanden op de externe harde schijf zouden moeten worden opgeslagen, zou dit de performantie te sterk naar beneden halen (de schijf is aangesloten via USB 2.0).

Om een RAID-Z1 VDEV te kunnen creëren, dienen eerst de bestaande VDEV's worden verwijderd. Het makkelijkste is om de bestaande pool volledig te verwijderen en een nieuwe pool met een RAID-Z1 VDEV aan te maken:

```
$ zpool destroy storage
$ zpool create storage raidz1 /dev/sda /dev/sdb /dev/sdc
$ zpool status
pool: storage
state: ONLINE
scan: none requested
config:

    NAME            STATE        READ  WRITE CKSUM
    storage          ONLINE       0     0     0
      raidz1-0       ONLINE       0     0     0
        sda          ONLINE       0     0     0
        sdb          ONLINE       0     0     0
        sdc          ONLINE       0     0     0

errors: No known data errors
```

Er zijn nog andere RAID-Z VDEV's mogelijk, zoals RAID-Z2 (equivalent van RAID 6) en RAID-Z3. Deze vereisen echter vier schijven of meer en zijn dus niet mogelijk in combinatie met het testsysteem.

SLOG & L2ARC VDEV's

Typisch wordt er een klein en snel opslagapparaat - zoals een SSD - gebruikt voor de SLOG en L2ARC. Om te demonstreren hoe de indeling van een pool met één van deze VDEV's er zou uitzien, wordt er telkens een mirror VDEV aangemaakt samen met respectievelijk een SLOG VDEV en een L2ARC VDEV. Een SLOG VDEV kan worden gemirrored; een L2ARC VDEV niet (The FreeBSD Documentation Project, 2017).

Eerst en vooral moet de bestaande pool worden verwijderd; nadien kan een nieuwe pool met de nodige VDEV's worden aangemaakt:

```
$ zpool destroy storage
$ zpool create storage mirror /dev/sda /dev/sdb log /dev/sdc
$ zpool status
pool: storage
state: ONLINE
scan: none requested
config:

    NAME            STATE        READ  WRITE CKSUM
    storage          ONLINE       0     0     0
      mirror-0       ONLINE       0     0     0
        sda          ONLINE       0     0     0
        sdb          ONLINE       0     0     0
    logs
      sdc            ONLINE       0     0     0

errors: No known data errors
```

In bovenstaand voorbeeld ziet men mooi dat SLOG VDEVs zich buiten een pool bevinden. Ook is er in dit geval een duidelijk onderscheid tussen logische en fysieke VDEV's: mirror-0 en logs zijn logische VDEV's, terwijl sda, sdb en sdc fysieke VDEV's zijn. Men ziet ook dat het mogelijk is om bij de creatie van een pool onmiddellijk alle VDEV-definities mee te geven.

Het aanmaken van een caching VDEV verloopt gelijkaardig:

```
$ zpool destroy storage
$ zpool create storage mirror /dev/sda /dev/sdb cache /dev/sdc
$ zpool status
  pool: storage
  state: ONLINE
    scan: none requested
config:

      NAME          STATE      READ  WRITE CKSUM
  storage          ONLINE         0     0     0
    mirror-0       ONLINE         0     0     0
      sda           ONLINE         0     0     0
      sdb           ONLINE         0     0     0
  cache
    sdc             ONLINE         0     0     0

errors: No known data errors
```

Ook hier bevindt de L2ARC VDEV zich buiten de pool en kan opnieuw de indeling tussen fysieke en logische VDEV's duidelijk worden opgemerkt.

8. ZFS Datasets

In dit hoofdstuk worden ZFS datasets of bestandssystemen besproken. Er wordt onder andere getoond welke soorten datasets er binnen ZFS bestaan, hoe men een dataset aanmaakt en hoe eigenschappen en gedrag van een dataset kan worden gewijzigd m.b.v. properties. Daarnaast wordt er ook getoond hoe datasets kunnen worden gedeeld met andere computers via NFS.

8.1 Verschillen met traditionele bestandssystemen

In essentie verschillen datasets (of bestandssystemen) binnen ZFS niet zo heel veel van andere, "traditionele" bestandssystemen. Een dataset is in principe een verzameling van data dat een bepaalde naam heeft; deze dataset vormt een logische eenheid van beheer. Dit komt grotendeels overeen met partities bij andere bestandssystemen, waarbij een (deel van) een schijf wordt gereserveerd voor een bepaald type gebruik (zoals gebruikersmappen) (Lucas & Jude, 2015).

Toch zijn er een aantal grote verschillen, zoals bijvoorbeeld de manier waarop er omgegaan wordt met beschikbare schijfruimte.

8.1.1 Gebruik van de beschikbare opslagcapaciteit

Een groot verschil tussen bestandssystemen bij ZFS en vele andere bestandssystemen, is dat ZFS datasets gebruik maken van de voordelen van storage pools. ZFS legt bijna geen limieten op aan de grootte van bestandssystemen; de enige limiet die aan de grootte van bestandssystemen wordt opgelegd, is de grootte van de storage pool. Dit is een

groot verschil met traditionele bestandssystemen: alvorens een partitie aan te maken en te formatteren, moet de systeembeheerder eerst nadenken over de grootte en lay-out van deze partitie en van de rest van de schijf of schijven. Na deze beslissingen genomen te hebben, maakt de gebruiker deze partitie aan. (Lucas & Jude, 2015).

Standaard neemt een ZFS dataset de ruimte van een pool in die het nodig heeft; de systeembeheerder moet zich dus niet bezighouden met het vooraf instellen van de grootte van een dataset. Het is echter wel mogelijk om de groei van een dataset tegen te gaan d.m.v. quota's en reservaties (The FreeBSD Documentation Project, 2017); het gebruik van ZFS properties op datasets komt later in dit hoofdstuk nog aan bod.

8.1.2 Verschillende types van datasets

Een dataset kan binnen ZFS een aantal vormen aannemen. Eén van de meest voorkomende vormen van een dataset is een **file system** of bestandssysteem: dit is het equivalent van een klassiek bestandssysteem, met bestanden, mappen, permissies enzovoorts. Daarnaast bestaan er nog andere datasettypes, zoals snapshots, clones, volumes (of zvols) en bookmarks (Lucas & Jude, 2015). Deze verschillende types zullen iets uitgebreider worden besproken in de volgende sectie.

8.1.3 Hiërarchische structuur van ZFS datasets

Datasets worden - net zoals zpools - binnen ZFS voorgesteld als objecten, met elk bepaalde eigenschappen (of properties). Traditionele bestandssystemen hebben ook bepaalde eigenschappen die kunnen veranderd worden, maar ZFS gaat hier nog een stap verder in. Net zoals bij VDEV's worden datasets voorgesteld in een boomstructuur: elke dataset heeft een ouder en eventueel kinderen. Dit maakt het mogelijk om overerving toe te passen op datasets; hierbij erven de kinderen van een bepaalde dataset de properties over van hun ouder (The FreeBSD Documentation Project, 2017).

Datasets maken het voor de systeembeheerder makkelijk om verschillende soorten data van elkaar te scheiden; op deze datasets kunnen dan bepaalde properties worden ingesteld m.b.t. bijvoorbeeld gebruikersrechten. Met behulp van ouder-kind relaties tussen datasets kunnen zowel algemene eigenschappen als meer specifieke eigenschappen worden ingesteld. Er kan bijvoorbeeld een dataset worden aangemaakt voor een project; deze beschikt dan over algemene eigenschappen, zoals toegangsrechten voor de verschillende teams die aan het project werken. Binnen deze ouder-dataset kunnen kind-datasets worden aangemaakt met meer specifieke eigenschappen; elke dataset behoort bijvoorbeeld toe aan een specifiek team en heeft dan ook eigenschappen die uniek zijn voor dat bepaald team, bovenop de reeds overgeërfdde eigenschappen van de ouder-dataset (voorbeeld naar Lucas en Jude (2015)).

8.2 Aanmaken & beheren van ZFS datasets

Voor het beheer van datasets wordt het commando `zfs` gebruikt; de syntax komt grotendeels overeen met die van `zpool`, het commando om ZFS pools te beheren.

Als vertrekpunt wordt er een `zpool` gebruikt bestaande uit één RAID-Z1 VDEV en drie fysieke VDEV's:

```
$ zpool create storage raidz1 /dev/sda /dev/sdb /dev/sdc
$ zpool status
pool: storage
state: ONLINE
scan: none requested
config:
```

| NAME | STATE | READ | WRITE | CKSUM |
|----------|--------|------|-------|-------|
| storage | ONLINE | 0 | 0 | 0 |
| raidz1-0 | ONLINE | | 0 | 0 |
| sda | ONLINE | | 0 | 0 |
| sdb | ONLINE | | 0 | 0 |
| sdc | ONLINE | | 0 | 0 |

```
errors: No known data errors
```

8.2.1 Bekijken en aanmaken van datasets

Om de huidige aanwezige datasets te bekijken, gebruik je het commando `zfs list`:

```
$ zfs list
NAME      USED  AVAIL  REFER  MOUNTPOINT
storage  464K   898G   128K   /storage
```

In dit geval is er reeds automatisch een dataset aangemaakt met de naam 'storage' en als mountpoint '/storage'. Deze heeft dezelfde naam als de `zpool`: deze is dus m.a.w. de parent dataset van alle andere datasets die binnen deze pool zullen worden aangemaakt.

Er kan ook een lijst worden opgevraagd van alle aangekoppelde (*mounted*) bestandssystemen m.b.v. het UNIX-commando `df`:

```
$ df -h
Filesystem                Size      Used Avail Use% Mounted on
devtmpfs                  4.9G         0   4.9G   0% /dev
tmpfs                     4.9G         0   4.9G   0% /dev/shm
tmpfs                     4.9G    920K   4.9G   1% /run
tmpfs                     4.9G         0   4.9G   0% /sys/fs/cgroup
/dev/mapper/fedora-root    15G    1.8G    14G  12% /
tmpfs                     4.9G    4.0K   4.9G   1% /tmp
/dev/sdd1                 976M    138M   772M  16% /boot
tmpfs                     993M         0   993M   0% /run/user/1000
storage                   899G    128K   899G   1% /storage
```

Zoals kan worden afgeleid uit de uitvoer van bovenstaand commando, is de ZFS dataset inderdaad gekoppeld aan de map `/storage`.

Om binnen de dataset `storage` een nieuwe dataset aan te maken, gebruik je het commando `zfs create`. Met behulp van dit commando kunnen er verschillende soorten ZFS datasets worden aangemaakt. Hieronder volgt een overzicht van de meest voorkomende datasets en hoe deze kunnen worden ingezet.

Filesystems

Filesystems (of bestandssystemen) zijn de meest voorkomende datasettypes. Qua eigenschappen en werking komen ze overeen met traditionele bestandssystemen: beide dienen om bestanden en mappen op te slaan, bevatten een POSIX-achtig permissiemodel en inodes (Lucas & Jude, 2015).

Om het voorgaande voorbeeld van de verschillende teams met verschillende projecten wat te illustreren, maken we onder de root dataset `/storage` de dataset `/storage/projects` aan. Daarna worden de andere datasets aangemaakt onder de parent `/storage/projects`.

```
$ zfs create storage/projects
$ zfs create storage/projects/www
$ zfs create storage/projects/dev
$ zfs list
```

| NAME | USED | AVAIL | REFER | MOUNTPPOINT |
|----------------------|------|-------|-------|-----------------------|
| storage | 767K | 898G | 133K | /storage |
| storage/projects | 394K | 898G | 139K | /storage/projects |
| storage/projects/dev | 128K | 898G | 128K | /storage/projects/dev |
| storage/projects/www | 128K | 898G | 128K | /storage/projects/www |

Uit de uitvoer van `zfs list` kan men de makkelijk de ouder-kindrelaties opmaken: `storage` is de ouder van `projects` en `projects` is op zijn beurt opnieuw ouder van `dev` en `www`.

Volumes

ZFS volumes (of zvols) zijn het equivalent van UNIX block devices en worden dan ook als block devices voorgesteld; zvols bevinden zich onder de map `/dev/zvol`. In deze situatie doet ZFS zich voor als een soort van volume manager: de zvols worden door gebruikers en applicaties gezien als een normale schijf. Bovenop deze ZFS volumes kunnen dan andere bestandssystemen (zoals EXT4) worden aangemaakt; het grote voordeel hierbij is dat er bovenop de functionaliteiten van het gebruikte bestandssysteem ook eigenschappen van ZFS kunnen worden toegepast. Zo kan er bijvoorbeeld m.b.v. ZFS compressie worden toegepast op bestandssystemen die dit normaal niet ondersteunen (zoals FAT) (The FreeBSD Documentation Project, 2017).

Om een ZFS volume aan te maken met een grootte van 20GB en met compressie, gebruik je het volgende commando:

```
$ zfs create -V 20G -o compression=on storage/projects/vol_test
$ zfs list
```

| NAME | USED | AVAIL | REFER | MOUNTPOINT |
|---------------------------|-------|-------|-------|---------------------------|
| storage | 20.6G | 877G | 133K | /storage |
| storage/projects | 20.6G | 877G | 139K | /storage/projects |
| storage/projects/dev | 128K | 877G | 128K | /storage/projects/ dev |
| storage/projects/vol_test | 20.6G | 898G | 85.2K | - |
| storage/projects/www | 128K | 877G | 128K | /storage/projects/ www |

Een zvol heeft geen mount point, aangezien het om een block device gaat. Om bijvoorbeeld een EXT4 bestandssysteem aan te maken bovenop het volume en het vervolgens aan te koppelen, kan men de volgende procedure volgen:

```
$ mkfs.ext4 /dev/zvol/storage/projects/vol_test
$ mkdir /mnt/zvol_test
$ mount /dev/zvol/storage/projects/vol_test /mnt/zvol_test/
$ df -h /mnt/zvol_test/
```

| Filesystem | Size | Used | Avail | Use% | Mounted on |
|------------|------|------|-------|------|----------------|
| /dev/zd0 | 20G | 45M | 19G | 1% | /mnt/zvol_test |

In bovenstaand voorbeeld werd er eerst een EXT4-bestandssysteem aangemaakt op het volume; daarna werd het nodige *mount point* aangemaakt. Vervolgens werd het bestandssysteem aan het aankoppelpunt gekoppeld.

Snapshots

Snapshots zijn *read-only* kopieën van een dataset; deze kunnen gemaakt worden van zowel bestandssystemen als volumes. Dankzij de COW-eigenschappen van ZFS is het triviaal om een snapshot te maken: bij het aanmaken van een snapshot wordt er eerst een kopie gemaakt van de data; deze zal worden blijven gebruikt door het systeem. Bij COW wordt de oude data niet overschreven: het is deze data dat in een snapshot terecht zal komen (Lucas & Jude, 2015).

Voordelen van snapshots zijn o.a. dat ze snel kunnen worden aangemaakt en worden teruggezet en dat ze weinig beslag leggen op de beschikbare schijfruimte. Bij creatie worden alle data en metadata van de dataset gedupliceerd d.m.v. referenties te leggen naar de originele dataset. Vanaf het ogenblik dat de brondataset verandert, moeten deze wijzigingen in de snapshot worden bijgehouden. In het begin is dus de grootte van een snapshot gelijk aan die van de dataset waarvan er een kopie gemaakt werd (The FreeBSD Documentation Project, 2017).

Een snapshot kan bijvoorbeeld worden gebruikt om een back-up te hebben van de productieomgeving binnen het bedrijf. Als er iets fout is gelopen, kan er makkelijk worden teruggerold naar de vorige versie m.b.v. een snapshot.

Om een snapshot te maken van de dataset `storage/projects/www`, kan men het volgende commando gebruiken:

```
$ zfs snapshot storage/projects/www@www_$(date +%d-%m-%Y)
$ zfs list -t snapshot
```

| NAME | USED | AVAIL | REFER |
|-------------------------------------|------|-------|--------|
| MOUNTPOINT | | | |
| storage/projects/www@www_16-05-2017 | 0 | - | 128K - |

De '@' is een delimiter; na deze karakter kan een naam worden ingegeven voor de dataset. In bovenstaand voorbeeld wordt de naam van de dataset gekozen gevolgd, door de huidige datum. De optie '-t' bij het commando `zfs list` wordt gebruikt om enkel bepaalde types van datasets weer te geven.

Er kunnen ook recursieve snapshots worden gemaakt: hierbij worden kind-datasets ook mee opgenomen in een snapshot. Om aan te geven dat er een recursieve snapshot moet worden gecreëerd, wordt de optie '-r' gebruikt.

```
$ zfs snapshot -r storage@storage_rec_$(date +%d-%m-%Y)
$ zfs list -t snapshot
```

| NAME | REFER | MOUNTPOINT | USED | AVAIL |
|--|-------|------------|------|-------|
| storage@storage_rec_16-05-2017 | | | 0 | - |
| 133K - | | | | |
| storage/projects@storage_rec_16-05-2017 | | | 0 | - |
| 139K - | | | | |
| storage/projects/dev@storage_rec_16-05-2017 | | | 0 | - |
| 128K - | | | | |
| storage/projects/vol_test@storage_rec_16-05-2017 | | | 0 | - |
| 3.43M - | | | | |
| storage/projects/www@www_16-05-2017 | | | 0 | - |
| 128K - | | | | |
| storage/projects/www@storage_rec_16-05-2017 | | | 0 | - |
| 128K - | | | | |

Hier ziet men inderdaad dat het aanmaken van de snapshot `storage_rec_16-05-2017` recursief gebeurde.

Het gebruik van snapshots komt nog uitgebreid aan bod in Hoofdstuk ??: *Data-integriteit en Consistentie bij ZFS*.

Clones

Een clone is een kopie van een snapshot dat qua werking overeenkomt met een volwaardig bestandssysteem. De clone en de originele snapshot leven dus naast elkaar; zo is bijvoorbeeld de snapshot de productieomgeving en de clone een kopie van deze omgeving. Dit maakt het mogelijk om aanpassingen door te voeren en te testen op de clone zonder dat deze de productieomgeving beïnvloeden. Bij creatie nemen clones geen extra ruimte in, aangezien ze gelijk zijn aan de snapshot waarop ze gebaseerd zijn. Als er aanpassingen aan de clone worden gedaan, dan wordt er echter wel extra opslagruimte aangewend (Lucas & Jude, 2015).

Het concept van snapshots en clones valt het best te vergelijken met die van branches in bijvoorbeeld een versiebeheersysteem zoals git: de snapshot is de master branch, de clone is een aftakking van deze master branch. Er moet wel opgemerkt worden dat clones geen updates ontvangen van het bestandssysteem waarvan de snapshot werd genomen; indien nodig moeten er dus een nieuwe snapshot en dus ook een nieuwe clone worden aangemaakt (Lucas & Jude, 2015).

Om een clone van een snapshot aan te maken, gebruikt men het commando `zfs clone <snapshot> <nieuwe_dataset>`:

```
$ zfs clone storage/projects/www@www_16-05-2017 storage/projects/
  www_new
$ zfs list
```

| NAME | USED | AVAIL | REFER | MOUNTPOINT |
|---------------------------|-------|-------|-------|---------------------------|
| storage | 20.6G | 877G | 133K | /storage |
| storage/projects | 20.6G | 877G | 144K | /storage/projects |
| storage/projects/dev | 128K | 877G | 128K | /storage/projects/dev |
| storage/projects/vol_test | 20.6G | 898G | 3.43M | - |
| storage/projects/www | 128K | 877G | 128K | /storage/projects/www |
| storage/projects/www_new | 10.7K | 877G | 128K | /storage/projects/www_new |

Uit de output van `zfs list` kan worden afgeleid dat er inderdaad een nieuwe dataset werd aangemaakt met de naam `www_new`. Deze nieuwe dataset en de oorspronkelijke snapshot zijn aan elkaar gekoppeld.

Om de relatie tussen een snapshot en zijn kloon of klonen te bekijken, kunnen de betreffende properties van de clone worden opgehaald. Uit onderstaande uitvoer kan worden afgeleid dat de clone gebaseerd is op de snapshot `storage/projects/www@www_16-05-2017`.

```
$ zfs get origin /storage/projects/www_new
```

| NAME | PROPERTY | VALUE |
|--------------------------|----------|-------------------------------------|
| storage/projects/www_new | origin | storage/projects/www@www_16-05-2017 |

Om een clone te 'upgraden' of te promoveren tot een volwaardige dataset die volledig op zichzelf staat, gebruik je het commando `zfs promote <naam clone>`:

```
$ zfs promote storage/projects/www_new
$ zfs get origin /storage/projects/www_new
```

| NAME | PROPERTY | VALUE | SOURCE |
|--------------------------|----------|-------|--------|
| storage/projects/www_new | origin | - | - |

De waarde van de property 'origin' is leeg, wat aanduidt dat de dataset niet meer afhangt van de snapshot `storage/projects/www@www_16-05-2017`.

8.2.2 Verwijderen van datasets

Een dataset verwijderen is relatief eenvoudig; dit kan verwezenlijkt worden met behulp van het commando `zfs destroy`. Echter moet er ook rekening worden gehouden met de relaties tussen de datasets, zoals ouder-kindrelaties en de relaties tussen snapshots en clones.

Op dit moment beschikken we over de volgende reeks van datasets:

```
$ zfs list
NAME                                USED    AVAIL    REFER    MOUNTPOINT
storage                            20.6G   877G    133K    /storage
storage/projects                   20.6G   877G    139K    /storage/projects
storage/projects/dev               128K    877G    128K    /storage/projects/
dev                                128K    877G    128K    /storage/projects/
storage/projects/vol_test          20.6G   898G    3.43M    -
storage/projects/www_new           139K    877G    128K    /storage/projects/
www_new                            139K    877G    128K    /storage/projects/

$ zfs list -t snapshot
NAME                                USED    AVAIL
storage@storage_rec_16-05-2017     85.2K    -
storage/projects@storage_rec_16-05-2017 90.6K    -
storage/projects/dev@storage_rec_16-05-2017 0        -
storage/projects/vol_test@storage_rec_16-05-2017 160K    -
storage/projects/www_new@www_16-05-2017 10.7K    -
```

Ter demonstratie wordt er een nieuwe dataset aangemaakt, die na afloop terug verwijderd wordt:

```
$ zfs create storage/test_ds
$ zfs list storage/test_ds
NAME                                USED    AVAIL    REFER    MOUNTPOINT
storage/test_ds                    128K    877G    128K    /storage/test_ds
$ zfs destroy storage/test_ds
$ zfs list storage/test_ds
cannot open 'storage/test_ds': dataset does not exist
```

Als een dataset kinderen bevat (zoals bijvoorbeeld snapshots), dan geeft het commando `zfs destroy` de volgende fout:

```
$ zfs destroy storage/projects/dev
cannot destroy 'storage/projects/dev': filesystem has children
use '-r' to destroy the following datasets:
storage/projects/dev@storage_rec_16-05-2017
```


Om een dataset en diens kinderen te verwijderen, gebruikt men de optie '-r':

```
$ zfs destroy -r storage/projects/dev
$ zfs list -t snapshot
```

| NAME | REFER | MOUNTPOINT | USED | AVAIL |
|--|-------|------------|-------|-------|
| storage@storage_rec_16-05-2017 | | | 85.2K | - |
| 133K | - | | | |
| storage/projects@storage_rec_16-05-2017 | | | 95.9K | - |
| 139K | - | | | |
| storage/projects/vol_test@storage_rec_16-05-2017 | | | 160K | - |
| 3.43M | - | | | |
| storage/projects/www_new@www_16-05-2017 | | | 10.7K | - |
| 128K | - | | | |

Na uitvoering kan inderdaad worden vastgesteld dat de kind-dataset storage/projects/dev@storage_rec_16-05-2017 verwijderd is.

8.2.3 Opvragen en wijzigen van properties

Datasets beschikken - net zoals storage pools - over een reeks properties die de eigenschappen en het gedrag van een dataset beïnvloeden. Reeds bij de voorbeelden over clones en snapshots werden er enkele voorbeelden van properties gegeven.

Om een lijst van alle properties op te halen wordt volgende instructie gebruikt:

```
$ zfs get all storage/projects
```

| NAME | PROPERTY | VALUE | SOURCE |
|------------------|---------------|-----------------------|---------|
| storage/projects | type | filesystem | - |
| storage/projects | creation | Mon May 15 21:19 2017 | - |
| storage/projects | used | 20.6G | - |
| storage/projects | available | 877G | - |
| storage/projects | referenced | 133K | - |
| storage/projects | compressratio | 2.15x | - |
| storage/projects | mounted | yes | - |
| storage/projects | quota | none | default |
| storage/projects | reservation | none | default |

(deel van de uitvoer is weggelaten)

De kolom SOURCE duidt aan of deze property al dan niet door de systeembeheerder werd gewijzigd; de waarde 'default' duidt aan dat deze property niet manueel werd aangepast.

In onderstaand voorbeeld wordt de waarde van de eigenschap quota eerst opgehaald, vervolgens gewijzigd en tenslotte een laatste keer opgehaald:

```
$ zfs get quota storage/projects
NAME                PROPERTY  VALUE   SOURCE
storage/projects    quota     none    default
$ zfs set quota=40GB storage/projects
$ zfs get quota storage/projects
NAME                PROPERTY  VALUE   SOURCE
storage/projects    quota     40G     local
```

Een quota dient om de groei van een dataset in te perken; in dit voorbeeld kan de dataset een maximale grootte aannemen van 40GB. Men kan ook zien dat de waarde van de kolom SOURCE veranderd is naar 'local', wat aanduidt dat de property manueel werd aangepast.

8.3 Toepassing: opzetten van een NFS-share m.b.v. ZFS datasets

In deze sectie wordt een ZFS dataset gebruikt om een NFS-share op te zetten die benaderd kan worden vanop het lokale netwerk. NFS (Network File System) is een netwerbestands-systeem dat gebruikt wordt om bestanden te delen met andere computers. Een grote troef van NFS is de relatieve eenvoud waarmee shares kunnen worden aangemaakt (The FreeBSD Documentation Project, 2017).

8.3.1 Installatie van NFS

Op de meeste Linux-distributies is NFS standaard geïnstalleerd. Wel moeten nog de nodige services worden geactiveerd.

```
$ dnf install nfs-utils
$ systemctl start nfs-server.service && systemctl enable nfs-server
.service
```

8.3.2 Aanmaken van een nieuwe ZFS dataset

Voor dit voorbeeld werden eerst alle andere datasets verwijderd. Daarna werd er een nieuwe dataset met de naam share aangemaakt onder de dataset storage.

```
$ zfs destroy -r storage
$ zfs create storage/share
$ zfs list
NAME                USED  AVAIL  REFER  MOUNTPOINT
storage             607K  898G   128K   /storage
storage/share       128K  898G   128K   /storage/share
```

Nu is het slechts nog een kwestie van de juiste properties in te stellen op de dataset en deze te delen met behulp van NFS.

8.3.3 Delen van de ZFS dataset

Om NFS als 'provider' te gebruiken, wordt de property `sharenfs` op de betreffende dataset aangezet en wordt de dataset gedeeld met het netwerk. Daarnaast wordt een firewall-regel toegevoegd om NFS-verkeer door te laten.

```
$ zfs set sharenfs="rw=@192.168.0.0/24" storage/share
$ zfs share storage/share
$ firewall-cmd --add-service=nfs && firewall-cmd --add-service=nfs
  --permanent
```

Met bepaalde properties kunnen er instellingen worden meegegeven; in dit geval kan men m.b.v. de property `sharenfs` instellingen meegeven die door NFS worden gebruikt: enkel computers in het 192.168.0.0/24 subnet hebben lees- en schrijftoegang tot de share. Indien men de mappen ophaalt die door NFS worden gedeeld, dan ziet men inderdaad dat de map `/storage/share` gedeeld wordt:

```
$ showmount -e
Export list for SRV-ZFS:
/storage/share 192.168.0.0/24
```

Tenslotte wordt de gebruiker 'jonas' eigenaar van de map gemaakt. Het is belangrijk dat de gebruikersnaam en user ID van zowel de gebruiker op de server als de gebruiker op de client overeenkomen. Er kan gebruik worden gemaakt van idmapping, maar dit voorbeeld is dermate eenvoudig dat dit enkel zou leiden tot complexiteit.

```
$ chown -R jonas:jonas chown -R jonas:jonas /storage/share/
```

8.3.4 Toegang tot de share vanaf een client-computer

Op dit moment is de share toegankelijk vanop de client; er moet enkel nog een mount point worden aangemaakt op de client waaraan de NFS-share kan worden gekoppeld. Vervolgens kan de share worden aangekoppeld. Om dit proces in de toekomst wat te stroomlijnen, wordt er een nieuwe lijn toegevoegd in `/etc/fstab`.

```
$ mkdir /mnt/nfsmount
$ echo "192.168.0.10:/storage/share /mnt/nfsmount nfs rw,
  defaults 0 2" >> /etc/fstab
$ mount -a
$ df -h
```

| Filesystem | Size | Used | Avail | Use% | Mounted on |
|---|------|------|-------|------|------------|
| (deel van de uitvoer is weggelaten) | | | | | |
| 192.168.0.10:/storage/share nfsmount | 899G | 756M | 898G | 1% | /mnt/ |

De gebruiker kan nu vanop de client bestanden aanmaken, verwijderen en wijzigen op de server via een NFS-share.

9. Performantieverschillen bij verschillende ty

10. Conclusie

Bibliografie

- A. Paterson, D. e. a. (1987). *A Case for Redundant Arrays of Inexpensive Disks*. University of California, Berkeley. Verkregen van <https://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-391.pdf>
- Arpaci-Dusseau, R. H. & Arpaci-Dusseau, A. C. (2015). *Operating Systems: Three Easy Pieces* (0.91). Arpaci-Dusseau Books.
- Bonwick, J. e.a. (2002). *The Zettabyte Filesystem*. Verkregen van <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.184.3704&rep=rep1&type=pdf>
- Bonwick, J. (2005). RAID-Z. Verkregen 2 april 2017, van https://blogs.oracle.com/bonwick/entry/raid_z
- Bonwick, J. e.a. (2015). The Birth of ZFS by Jeff Bonwick. OpenZFS Project. Verkregen 30 maart 2017, van <https://www.youtube.com/watch?v=dcV2PaMTAJ4&feature=youtu.be>
- Bonwick, J. & Moore, B. (Unknown). ZFS The Last Word In File Systems. Sun Microsystems. Verkregen van https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf
- Cohen, J. (g.d.). Trees. Verkregen van <https://www.cs.jhu.edu/~cohen/CS226/Lectures/Trees.pdf>
- Goda, K. & Kitsuregawa, M. (2012). *The History of Storage Systems*. Institute of Electrical and Electronics Engineers (IEEE). Verkregen van <http://ieeexplore.ieee.org/document/6182574/>
- Hickmann, B. & Shook, K. (2007). *ZFS and RAID-Z: The Uber-FS?* Verkregen van <http://pages.cs.wisc.edu/~remzi/Courses/736/Fall2007/Projects/BrianKynan/paper.pdf>
- IEEE and The Open Group. (2016). The Open Group Base Specifications Issue 7. Verkregen van <http://pubs.opengroup.org/onlinepubs/9699919799/>

- illumos. (2012, augustus 20). illumos Project Announcement - August 3, 2010. Verkregen van <https://wiki.illumos.org/display/illumos/illumos+Project+Announcement+-+August+3%2C+2010>
- Kendi, C. (Onbekend). ZFS: Enhancing the Open Source Storage System (and the Kernel). Verkregen van https://www.blackhat.com/presentations/bh-dc-10/Kendi_Christian/Blackhat-DC-2010-Kendi-Enhancing-ZFS-slides.pdf
- Lewis, A. (2006). LVM HOWTO: Setting up LVM on three SCSI disks. Verkregen van <http://www.tldp.org/HOWTO/LVM-HOWTO/recipe3threescsi.html#AEN1078>
- Li, A. (2009). *Digital Crime Scene Investigation for the Zettabyte File System* (proefschrift, Macquarie University). Verkregen van http://web.science.mq.edu.au/~rdale/teaching/itec810/2009H1/FinalReports/Li_Andrew_FinalReport.pdf
- Lucas, M. W. & Jude, A. (2015). *FreeBSD Mastery: ZFS*.
- M. Chen, P. e.a. (1994). *RAID: High-Performance, Reliable Secondary Storage*. University of Michigan , Carnegie Mellon University , University of California (Berkeley). Verkregen van <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=59E48A55C27748208046D52C5730C98F?doi=10.1.1.41.3889&rep=rep1&type=pdf>
- Oracle Corporation. (Onbekend). Oracle and Sun Microsystems. Verkregen van <https://www.oracle.com/sun/index.html>
- Sun Microsystems. (2006). *ZFS on-disk specification*. Verkregen van http://www.giis.co.in/Zfs_ondiskformat.pdf
- The BTRFS Project. (2014, maart 4). Status. Verkregen van <https://btrfs.wiki.kernel.org/index.php/Status>
- The BTRFS Project. (2017). On-disk Format. Verkregen van https://btrfs.wiki.kernel.org/index.php/On-disk_Format#Basic_Structures
- The FreeBSD Documentation Project. (2017). *FreeBSD Handbook*.
- The Open Group. (Onbekend). Open Brand. Verkregen van <https://www.opengroup.org/openbrand/register/xy.htm>
- The OpenZFS Project. (2014). History - OpenZFS. Verkregen van <http://open-zfs.org/wiki/History>
- The OpenZFS Project. (2017). Hardware - OpenZFS. Verkregen van <http://open-zfs.org/wiki/Hardware>
- ZFS On Linux. (2016). Fedora. Verkregen van <https://github.com/zfsonlinux/zfs/wiki/Fedora>
- ZFS On Linux. (2017). FAQ. Verkregen van <https://github.com/zfsonlinux/zfs/wiki/FAQ>

Woordenlijst

atomair In de informatica betekent atomiciteit (Engels: atomicity) hetzelfde als ondeelbaarheid; vaak wordt dit begrip ook omschreven als een "alles-of-niets" aanpak. In de context van bijvoorbeeld databanksystemen heeft atomiciteit betrekking op o.a. transacties: ofwel verloopt een transactie succesvol, ofwel mislukt deze (door bv. een onderbreking) en moet de databank kunnen teruggebracht worden naar een consistente toestand. Meestal betekent dit dan dat er een rollback moet worden uitgevoerd. 22, 24

betrouwbaarheid Betrouwbaarheid (Engels: reliability) heeft betrekking op de betrouwbaarheid van een computersysteem, i.e. of deze al dan niet zijn taken op een juiste manier uitvoert. Men kan bijvoorbeeld stellen dat er zich bij een computersysteem een X aantal fouten mogen voordoen; als deze grens overschreden wordt, dan wordt het systeem als onbetrouwbaar verklaard. Daarnaast heeft betrouwbaarheid ook te maken met de handelswijze van een systeem indien er zich een fout voordoet: men kan zich bijvoorbeeld afvragen hoe er zou moeten gereageerd worden op een kritieke fout en hiervoor test cases opstellen om de betrouwbaarheid van het systeem te testen. 15, 16, 17

bottleneck Een bottleneck (letterlijk vertaald 'flessenhals') is een begrip dat gebruikt wordt om een onderdeel van een systeem aan te duiden dat andere onderdelen hindert en/of vertraagt. Dit begrip wordt ook vaak gebruikt in de zelfbouw PC community om een computeronderdeel aan te duiden dat de computer ervan verhindert om optimaal te kunnen presteren: zo kan het samen gebruiken van een oude CPU en een nieuwe GPU ertoe leiden dat de computer niet optimaal presteert bij bijvoorbeeld games. In dit geval wordt de CPU de bottleneck van het systeem genoemd. 15, 17

capaciteit In de context van schijven of RAID-systemen kan capaciteit (Engels: capacity) een aantal betekenissen hebben. In deze scriptie wordt er vooral gesproken over de

bruikbare capaciteit van een schijf of RAID-array: dit is de hoeveelheid schijfruimte die kan gebruikt worden door gebruikers om data op te slaan. Daarnaast kan men ook spreken over de totale capaciteit: dit is de som van de gebruikte schijfruimte en de vrije schijfruimte. In theorie is de totale capaciteit van een schijf gelijk aan de werkelijke grootte van een schijf. 15, 16, 17

checksum Een checksum is een (meestal) klein gegeven dat dient om de dataintegriteit van een ander bestand te garanderen. Indien er tijdens het verzenden van een bestand bijvoorbeeld fouten optreden, dan kan dit met behulp van deze checksum worden gedetecteerd: bij aankomst wordt de checksum herberekend en vergeleken met de originele tekenreeks. Deze tekenreeks wordt berekend met behulp van een cryptografische functie dat het bestand in kwestie als invoer neemt. Zelfs bij een kleine wijziging aan het bestand, zal een herberekening van de checksum leiden tot een andere tekenreeks dan de originele. 27, 28

fsck fsck (acroniem voor 'filesystem consistency check') is een programma op UNIX en UNIX-achtige systemen dat bestandssystemen controleert op consistentie. Dit programma kan manueel worden gedraaid door de systeembeheerder, maar in de meeste gevallen wordt dit programma automatisch opgeroepen bij het opstarten van het systeem. fsck werkt op bestandssysteemniveau en dus heeft elk bestandssysteem een eigen, specifieke implementatie van dit programma. 40

fsync Op UNIX en UNIX-achtige systemen is sync of fsync een system call dat gebufereerde data dat zich in het RAM-geheugen bevindt flusht naar opslagapparaten. Door deze call uit te voeren wordt data die zich in het werkgeheugen bevindt - en zich dus in een vluchtige toestand bevindt - gepersisteerd en niet-vluchtig gemaakt. 25

journal Een journal is een soort logboek dat door journaling bestandssystemen (zoals EXT4) wordt bijgehouden. In een journal worden alle transacties die op een bepaald moment aan het lopen zijn gelogd; bij een crash of andere onderbreking kan het bestandssysteem worden hersteld naar een consistente toestand door het journal na te kijken en transacties terug te rollen. 21

parity Parity is een foutdetectiemechanisme dat gebruikt wordt om in geval van bijvoorbeeld datacorruptie de originele data te reconstrueren. Bij RAID gebruikt men parityblokken: dit zijn blokken metadata die worden berekend door een mathematische functie (meestal een XOR-functie) uit te voeren op de opgeslagen data. Zo kan een RAID-controller de originele data reconstrueren indien er een schijf wordt vervangen in een RAID-array. 16, 17, 18

performantie Performantie (Engels: performance) is een vrij uitgebreid begrip in de informatica. In het algemeen bedoelt men met performantie meestal hoe goed een computersysteem vooropgestelde taken kan uitvoeren in een bepaalde situatie. Bij het meten van performantie van een systeem kan men één of meerdere aspecten beschouwen: bij schijven kan bijvoorbeeld het aantal I/O's (Invoer- en uitvoerbewerkingen) worden genomen als maatstaf; bij CPU's kan een maatstaf bijvoorbeeld het aantal instructies per seconde zijn dat deze kan verwerken. 12, 15, 16, 37, 38, 43

round-robin In de informatica is round-robin een scheduling-algoritme dat o.a. gebruikt wordt in besturingssystemen. Bij round-robin krijgt elk proces om de beurt een gelijk stukje procestijd. Hierdoor treedt er bij deze vorm van scheduling geen

starvation op. Starvation is een fenomeen waarbij een proces te weinig of helemaal geen processortijd krijgt om zijn taken uit te voeren; het proces wordt a.h.w. "uitgehongerd" doordat het geen resources krijgt van het OS. 16

striping Striping heeft betrekking op de manier waarop blokken data verdeeld worden over een reeks van schijven door bijvoorbeeld een RAID-controller (hetzij hardwarematig, hetzij softwarematig). Datablokken die verdeeld zijn over meerdere schijven en samen één geheel vormen worden een stripe genoemd. 16, 37, 42, 43

Lijst van figuren

| | | |
|-----|---|----|
| 3.1 | Illustratie van verschillende RAID-niveaus (M. Chen e.a., 1994) . . . | 16 |
| 4.1 | Illustratie van ZFS pooled storage (rechts) t.o.v. volume-based storage (links) (Bonwick e.a., 2002) | 22 |
| 4.2 | Een overzicht van de verschillende componenten van ZFS (Kendi, Onbekend) | 24 |
| 5.1 | Boomstructuur bij aanpassing van één datablok binnen een ZFS-transactie. Van links naar rechts: (1) Aanpassen van het datablok; (2) Aanpassen van de indirecte blokken; (3) Overschrijven van de überblock. Alle bewerkingen gebeuren op een COW-manier. (Bonwick e.a., 2002) | 29 |
| 6.1 | Illustratie van de gehanteerde disk-layout van het systeem. De ingekaderde schijven zullen gebruikt worden door ZFS. | 33 |
| 7.1 | Conceptuele voorstelling van VDEV's in een boomstructuur: M1 en M2 zijn mirrors VDEV's; apparaten A t.e.m. D zijn kinderen van deze VDEV's (Sun Microsystems, 2006). | 38 |

Lijst van tabellen

| | |
|--|----|
| 6.1 Specificaties van het systeem dat gebruikt wordt doorheen deze bachelorproef (data verkregen via lshw) | 32 |
|--|----|