Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

UHASSELT KU LEUVEN

Universiteit Hasselt | Campus Diepenbeek | Faculteit Industriële Ingenieurswetenschappen | Agoralaan Gebouw H - Gebouw B | BE 3590 Diepenbeek

Universiteit Hasselt | Campus Diepenbeek | Agoralaan Gebouw D | BE 3590 Diepenbeek
Universiteit Hasselt | Campus Hasselt | Martelarenlaan 42 | BE 3500 Hasselt

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Design en implementatie van een schaalbaar logsysteem voor grootschalige Kubernetes-beheerde testbedomgevingen

Arne Papen

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

PROMOTOR :
Prof. dr. Kris AERTS

PROMOTOR :
Dhr. Jan VAN DEN ABEELE

COPROMOTOR :
Dhr. Maikel PUNIE

BEGELEIDER :
Dhr. Erik NEEL

UHASSELT KU LEUVEN

2022
2023

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: elektronica-ICT

Masterthesis

Design en implementatie van een schaalbaar logsysteem voor grootschalige Kubernetes-beheerde testbedomgevingen

Arne Papen

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: elektronica-ICT

PROMOTOR :

Prof. dr. Kris AERTS

PROMOTOR :

Dhr. Jan VAN DEN ABEELE

COPROMOTOR :

Dhr. Maikel PUNIE

BEGELEIDER :

Dhr. Erik NEEL

UHASSELT KU LEUVEN

Woord vooraf

Als finaal eindwerk voor de opleiding industrieel ingenieur Elektronica-ICT zocht ik net datgene waar ik doorheen de opleiding een beetje op mijn honger bleef zitten. Namelijk een onderwerp rond netwerk-gerelateerde oplossingen in combinatie met een vleugje virtualisatie en automatisatie. Bovendien zocht ik een toonaangevend bedrijf dat me binnen deze sector enorm veel kon bijleren op een korte termijn van slechts één semester.

Uiterlijk op maandag 10 oktober 2022 ben ik warm welkom geheten binnen Nokia te Antwerpen, om mee te werken aan de modernisering van hun testlabo binnen Kubernetes in het kader van deze masterproef. Naast mijn enthousiasme om te kunnen meewerken aan een implementatie die effectief gebruikt zou worden, bleek de bestaande omgeving toch net wat complexer dan ik gewoon was. De eerste weken kwamen moeizaam op gang maar waren desondanks een enorm goed leermoment om mijn kennis op allerhande vlakken al doende bij te schaven. Naarmate het geheel steeds duidelijker werd en de probleemstellingen zich vormde tot uitdagingen, begon er schot in de zaak te komen waardoor er steeds meer voldoening groeide. Met diezelfde voldoening in combinatie met de bijhorende passie, lever ik met trots deze masterproef af.

De totstandkoming van deze masterproef inclusief scriptie was nooit mogelijk geweest zonder de hulp van enkele personen die nog een oprecht dankwoord verdienen. Allereerst zou ik graag mijn externe promotor Jan van den Abeele willen bedanken voor zijn geduld en alle antwoorden die hij kon bieden op de technische vragen die ik hem dagdagelijks stelde. Verder zou ik ook heel graag mijn interne promotor Prof. dr. Kris Aerts willen bedanken voor zijn kritische feedback – in de beste zin van het woord – op alle inhoudelijk cruciale documenten, alsook zijn flexibiliteit in het algemeen. Daarnaast wil ik graag externe copromotor Maikel Punie en begeleider Erik Neel bedanken voor hun ontvangst binnen Nokia enerzijds, anderzijds voor het vertrouwen en de vrijheid die ze me gaven om deze masterproef tot een goed einde te leiden.

Tot slot spreek ik ook nog graag een dankbetuiging uit naar mijn vriendin alsook mijn familie en vrienden die me dag in dag uit over de hele lijn van de opleiding hebben gesteund waar nodig.

Dankjulliewel!

Arne Papen
Januari 2023

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Abstract

Het Nokia Network Infrastructure (NI) team in Antwerpen voert een modernisering uit op het in-house datacenter waar software-updates van corporate service-routers worden getest. De klassieke combinaties van virtuele machines maken plaats voor een gecontaineriseerde omgeving beheerd door Kubernetes.

Deze masterproef focust op het aanleveren van logfiles bij elk van de uitgevoerde testen. De hoofddoelstelling is om de logs realtime beschikbaar te stellen tijdens het uitvoeren van een test en na afloop permanent centraal raadpleegbaar te maken, waarbij de aanpak van de huidige implementatie en de schaalvoordelen van een gedistribueerd systeem optimaal benut worden.

Een studie naar de dimensionering van de logfiles, naar de state-of-the-art en naar de concrete behoeften leidde tot het gebruik van de Apache HTTP-server, enerzijds met een vluchtige opstelling tijdens lopende tests per testrun en anderzijds als centraal permanent en geaggregeerd toegangspunt. HAProxy wordt als reverse proxyserver en load balancer gebruikt om aanvragen naar de correcte HTTP-server te routeren. Tot slot is een zelfgeschreven ‘resulthandler’ in Go geïmplementeerd, die de testen traceert en de aansturing tussen de bovengenoemde componenten verzorgt in de Kubernetes-cluster.

Deze implementaties ten gevolge van de designkeuzes bieden een betere opvolging en aggregatie van logs, die efficiënter aan de gebruikers van de testruns aangeleverd worden, waarbij het systeem toelaat om ongecompliceerd extra modules toe te voegen.

Abstract (English)

The Network Infrastructure (NI) team within Nokia is modernizing the in-house datacenter where software-updates of their corporate service-routers are being tested. The traditional methods of using virtual machines are replaced by a containerized environment managed by Kubernetes.

This master's thesis focuses on providing log files of each performed test. The main objective is to make logs accessible while running a test, as well as making logs centrally and permanently available upon completion, taking full advantage of the current methodology and scalability of a distributed system.

A study into the dimensioning of log files using a sample, into the state-of-the-art and into the specific requirements of Nokia resulted in using the Apache HTTP-server, on the one hand as a volatile implementation during an ongoing test and on the other hand as a centrally and permanently aggregated gateway. HAProxy is used as a reverse proxy server and load balancer to route requests to the appropriate HTTP-server. Finally, a self-written Go script has been implemented as a so called 'resulthandler', which tracks the tests and controls the previously mentioned components within the Kubernetes cluster.

These implementations, as a result of the substantiated design selections, offer a better tracking mechanism and the centralization of log files, which are delivered more efficiently to the users of a testrun, allowing to add extra modules more easily in the future.

1 Inleiding

Het inleidende hoofdstuk geeft de situering en probleemstelling van deze masterproef, met daaropvolgend de doelstellingen en de methode die chronologisch de algemeen uit te voeren stappen benadrukt. Tot slot wordt nog kort de architecturale opbouw van de hoofddoelstelling besproken als inleiding naar de literatuurstudie en daaropvolgende hoofdstukken.

1.1 Situering

Nokia is een bedrijf dat bij de gewone consument vaak gelinkt wordt aan consumentenelektronica, meer bepaald aan mobiele telefoons. Hoewel Nokia nog steeds actief is in dit specifiek domein, focussen ze zich algemeen op technologieën in de telecommunicatiewereld [1]. Sites van Nokia zijn wereldwijd verspreid waarbij de Antwerpse vestiging zich voornamelijk focust op R&D van corporate ICT- en netwerkoplossingen [2]. Hieronder valt ook het testen en debuggen van Nokia's service-routers inclusief bijhorende software(updates) [3]. Dergelijke routers zijn in omvang alsook performantie en features, ontworpen om in verschillende datacenters en bij internetserviceproviders dienst te doen.

Voor het testen, debuggen en evalueren van dergelijke machines worden zogenaamde testbedden opgezet in een in-house datacenter. Door middel van een lokaal ontwikkeld CLI-commando kan een gebruiker de volledige testomgeving in één keer opstellen. Deze omgevingen bestaan uit virtuele machines met computer resources en benodigde netwerklinken. Hoewel het starten van deze testbedden een vertrouwde manier van werken is bij de engineers, zijn de achterliggende principes voor het klaarmaken van de infrastructuur verouderd. Zo wordt er binnen het Network Infrastructure team (NI) gewerkt met een paar duizend aparte testbedden die volledig gescheiden zijn van elkaar, waarbij elk testbed zijn eigen resources (geheugen en processoren) en hypervisor met virtuele machines ter beschikking heeft. De hoge performantiekost van virtuele machines, alsmede het complexe beheer van alle afzonderlijke servers werd steeds meer als problematisch ervaren. Bovendien was de security binnen de omgeving nog niet optimaal hoewel alle systemen gescheiden waren van elkaar.

Vandaar is er door Nokia beslist om over te gaan op containerisatie. Hierdoor werd bij aanvang van de modernisering van het testlabo de focus gelegd op een meer lichtgewichtoplossing die centraal te beheren valt met de nodige veiligheidsvoorschriften. Voor het centrale beheer van containers wordt er in het algemeen gebruik gemaakt van orkestratiesoftware. Nokia heeft reeds een studie gedaan over beschikbare orkestratiesoftware en hiervoor Kubernetes gekozen als oplossing die het beste aan hun noden voldoet.

De implementatie van het volledige systeem staat nog in zijn kinderschoenen en moet op verschillende punten nog geoptimaliseerd of zelfs uitgerold worden. Deze masterproef focust zich in het algemeen op het dimensioneren van logfiles en het aanleveren van logfiles aan gebruikers van de testbedden, wat slechts een relatief klein item is binnen het moderniseringsproces van het testlabo. De Kubernetes-cluster is momenteel nog in een testopstelling geplaatst, maar naar de toekomst toe is het echter van belang dat het grootste deel van het in-house datacenter mee in de Kubernetes-cluster deelneemt.

1.2 Probleemstelling

Testbedden worden vandaag de dag nog apart klaargemaakt in een intern ontwikkelde tool van Nokia waarbij gebruik wordt gemaakt van virtuele machines die op een hypervisor draaien. Wat betreft de test- en debugfunctionaliteit, zijn test engineers of gewoonweg gebruikers op zich wel tevreden met het systeem dat intern ontwikkeld is. Vandaar is een belangrijke vereiste dat op zijn minst alle functionaliteiten van de bestaande versie ook beschikbaar zijn in de nieuwe Kubernetes-versie. Verder

is het ook van belang dat de nieuwe versie volgens de state-of-the-art opgebouwd wordt waarbij de voordelen van de actuele technologie zo optimaal mogelijk benut worden.

Binnen de scope van deze masterproef stelt men de volgende problemen vast:

- Logfiles worden al jarenlang geproduceerd en geaggregeerd per testrun. Echter is er niet geweten welke impact de logfiles – in hun geheel of afhankelijk van het soort testbed – hebben op de storage van een machine of het volledige testlabo. Er zijn nagenoeg **geen statistische gegevens** over het aantal bestanden per testrun of de grootte van een testrun bekend.
- Het aanleveren van logfiles gebeurt momenteel binnen de virtuele machines a.d.h.v. een aparte HTTP-server per testbed. Deze logs worden echter automatisch verwijderd zodra de tests ten einde zijn, tenzij de gebruiker bewust actie onderneemt. Bovendien is er **geen centrale plaats** voorzien waar logs worden bewaard. Bijgevolg zijn ze achteraf moeilijk terug te vinden en te raadplegen.
- Hoewel een virtuele machine op zich een geïsoleerde omgeving is, kunnen gebruikers zonder enige vorm van verdere authenticatie alle testbedden benaderen. Veiligheid tussen testbedden is gegarandeerd, maar **veiligheid** van gebruikers naar een testbed niet. Zo zijn in het verleden al wormen geïntroduceerd, met alle gevolgen van dien.
- Per fysieke machine is er slechts één testbed dat draait. Intern onderzoek van Nokia toont aan dat een fysieke machine soms wel 1,5 tot 3 testbedden tegelijk zou kunnen draaien afhankelijk van de load en de beschikbare resources. De **efficiëntie** in het algemeen ligt dus laag.
- Elke Nokia-site die rondom dezelfde onderwerpen testen uitvoert, heeft de dag van vandaag de vrijheid om te kiezen hoe de concrete aanpak per site verloopt. Het **centraal beheer** van alle onderdelen is dus haast onmogelijk door een en dezelfde tool.

Met deze probleemstellingen in het achterhoofd kan er één overkoepelende onderzoeksvraag aan gekoppeld worden, namelijk:

Hoe kunnen logfiles gedimensioneerd worden, alsook hoe kunnen die centraal en gedistribueerd aangeleverd worden aan gebruikers d.m.v. Kubernetes zonder daarbij aan de natuurlijke schaalvoordelen van een volledig gedistribueerd systeem te geraken?

Bovendien kan deze onderzoeksvraag verder opgedeeld worden in kleinere deelvragen die als gevolg effectieve doelstellingen nastreven. Deze verdere deelvragen zijn daarom raadpleegbaar onder de titel Doelstellingen, om zo de samenhang te bevorderen.

1.3 Doelstellingen

Deze masterproef focust zich voornamelijk op het aanleveren van logfiles zoals reeds aangehaald bij de probleemstelling. Hierbij is het van belang om niet in dezelfde valkuilen als de vooropgestelde problemen te vallen. Daarentegen is het wel belangrijk om vooruitstrevende ideeën met focus op efficiëntie, schaalbaarheid en centraal beheer te introduceren.

In de praktijk, zoals die toegepast wordt binnen Nokia, is het duidelijk dat logbestanden in een testomgeving cruciaal zijn op allerhande vlakken. Een eerste vereiste is echter het verkrijgen van statistische gegevens over logfiles per type testbed en globaal over alle testbedden d.m.v. een steekproef.

Nadat algemene informatie over logfiles is verkregen kan er gefocust worden op de aanlevering ervan. Aanvankelijk zijn de logfiles van de cluster tijdens het opbouwen van een testbed van groot belang voor de infrastructure engineers. In latere stadia, zal o.a. de test engineer baat hebben bij de geproduceerde logfiles van de software die in het testbed draait. Het raadplegen van deze files wordt in de huidige opbouw gedaan d.m.v. een HTTP-server per testbed. Deze server visualiseert de gegevens volgens een vastgelegde structuur. De masterproef zal de containerisatie in een Kubernetes-cluster van zo'n HTTP-server per testbed op zich nemen.

Logs worden nadat het testbed wordt gesloten mee verwijderd met alle bijhorende gegevens indien de gebruiker zelf geen actie onderneemt om ze te bewaren. Echter kan het van belang zijn om deze gegevens later op een centrale plaats te raadplegen om de logs van individuele testbedden te kunnen bekijken als een dashboard van geaggregeerde gegevens. Hiervoor is een centrale HTTP-server nodig die bereikbaar is voor elke gebruiker met toegangsrechten afhankelijk van de rol. Desalniettemin dient de communicatie tussen de gedistribueerde log servers en de centrale log server op een gepaste en consistente manier te gebeuren, en dit alles op een zo performant en schaalbaar mogelijke wijze. Afhankelijk van de load dient er een mogelijkheid te zijn om de centrale server op te schalen. Hoe dan ook, moet deze centrale server minimaal 100 aanvragen per seconde kunnen afhandelen. Hierbij wordt er voorkeur gegeven aan een opensourceoplossing boven een ad-hoc en intern te ontwikkelen systeem.

Deze concrete vereisten geven aanleiding tot de volgende deelvragen:

- **Wat is de omvang van logfiles in de huidige opzet, gecategoriseerd per type testbed en voor de volledige steekproef?**
- **Hoe kan een Kubernetes gevirtualiseerd testbed automatisch logs van testbedgerelateerde apparaten combineren, visualiseren en transfereren naar een centrale log server?**
- **Hoe kan een centrale Kubernetes gevirtualiseerde server die gedistribueerde logs ontvangt, gestructureerd en gecombineerd logs weergeven, rekening houdend met de stijgende schaalbaarheid afhankelijk van het aantal aanvragen per tijdsinterval?**
    - **Wat is de meest performante freeware HTTP-server die voldoet aan de schaalbaarheidsnoden?**
    - **Wat is de meest geschikte freeware reverse proxyserver en load balance oplossing voor deze opschaalbare HTTP-server in Kubernetes?**
    - **Welke storage oplossing is het meest geschikt voor logfiles die vanuit verschillende standpunten worden benaderd?**

Over het algemeen wil men tot een punt komen waar een gebruiker de mogelijkheid heeft om van buitenaf de Kubernetes-cluster ononderbroken logfiles van enerzijds gedistribueerde testbedden te raadplegen tijdens het lopen van een testrun. Anderzijds dat na de afloop van een testrun de logfiles centraal raadpleegbaar zijn op één geaggregeerde plaats. Deze implementatie laat toe dat een testbed een vluchtig object wordt met dezelfde levensduur als een testrun.

1.4 Materiaal en methode

Het eerste werkpakket slaat op het dimensioneren van de logfiles tot enkele statistische resultaten. Het daaropvolgende werkpakket focust op het voorzien van een server die de aanlevering van logfiles gedistribueerd per testbed verzorgt. Hierbij is de voornaamste taak om lokale logfiles te visualiseren per testbed tijdens het lopen van testruns. Vervolgens heeft het derde en laatste werkpakket betrekking op het deployen van een schaalbare centrale HTTP-server, die als centraal contactpunt voor alle logfiles van de afgelopen testruns dient. Het is belangrijk om op te merken dat het tweede en derde werkpakket apart worden geïmplementeerd, maar dat ze feilloos met elkaar dienen samen te werken. De implementatie van deze samenwerking wordt verweven in het derde werkpakket.

Basisinzichten zoals virtualisatie, containerisatie en orkestratie zijn fundamenteel om bepaalde begrippen en concepten te kunnen toepassen doorheen de masterproef, bovenop de keuzes van bepaalde software. Daarom diende een brede literatuurstudie over deze onderwerpen uitgevoerd te worden, alvorens kon gestart worden met de effectieve werkpakketten. Daarenboven is een bescheiden kennis van de service-routers van Nokia die getest worden in de testbedden een meerwaarde. Dit laat toe om globaal gezien mee te kunnen denken in het volledige verhaal. De literatuurstudie zal dus ook enigszins

uitweiden over de zogenaamde BNG-routers van Nokia, gevolgd door het effectief onderzoek naar een antwoord op de vermelde onderzoeksvragen.

WP 1: Dimensioneren van logfiles

In het eerste werkpakket dient er een statistisch onderzoek gedaan te worden naar de omvang van de bestaande testruns op de permanente testbedden. Dit omvat als eerste stap een intern onderzoek naar de structuur van de huidige logfiles, hoe logfiles worden gegenereerd en waar ze worden opgeslagen. Zodra genoeg kennis is verworven, dient er een oplossing gevonden te worden voor het al dan niet automatisch verkrijgen van de grootte van de logfiles en het aantal bestanden per testrun. Een steekproef met voldoende resultaten moet leiden tot statistische gegevens die enerzijds over alle categorieën testbedden uitweidt, anderzijds over gegevens per aparte categorie.

WP 2: Gedistribueerd aanleveren van logfiles

Het tweede werkpakket heeft betrekking op het aanleveren van logfiles gedistribueerd per testbed. Aanvankelijk dient er een exploratie uitgevoerd te worden van de huidige HTTP-server en de structuur hoe logfiles worden aangeleverd aan deze server. Vervolgens dient er onderzoek gedaan te worden naar een mogelijkheid om deze HTTP-server te containeriseren per testbed in de Kubernetes-cluster. Hierbij is het van belang om eenzelfde structuur zoals geëxploreerd in de eerste stap aan te houden. Op basis van het onderzoek dient de effectieve implementatie uitgevoerd te worden, rekening houdend met de vluchtigheid van een testrun. Dit tweede werkpakket zal als basis dienen om kennis te maken met de Kubernetes-omgeving en interne informatie die reeds in de cluster werd ondergebracht.

WP 3: Gecentraliseerd aanleveren van logfiles

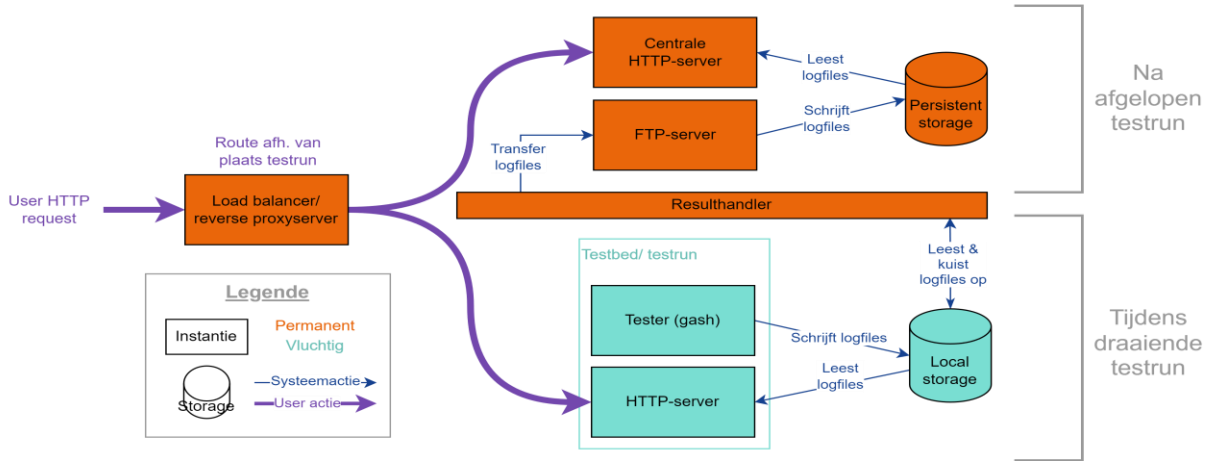
Het derde werkpakket betreft de centrale HTTP-server die de logs van alle testbedden visualiseert en aggregeert, in combinatie met de communicatie tussen de gedistribueerde testbedden. Allereerst is het essentieel om een verantwoorde keuze van HTTP-server alsook van load balancer – die door Nokia noodzakelijk werd geacht – te maken a.d.h.v. een literatuurstudie. Hierbij staan performantie, populariteit en containerisatie-mogelijkheden centraal. Daaropvolgend is de implementatie van deze persistente centrale servers binnen de cluster van belang. Desalniettemin dient de communicatie tussen de centrale en gedistribueerde servers alsook tussen de gebruikers en de HTTP-servers op een consistente en betrouwbare manier voorzien te worden. De natuurlijke schaalvoordelen van een Kubernetes-cluster dienen vooral tijdens dit werkpakket zo optimaal mogelijk benut te worden.

Tot slot is het nog van belang om storageoplossingen voor centrale logfiles die worden getoond via de HTTP-server te onderzoeken. Het onderzoek naar de storageoplossing doorheen de masterproef is puur theoretisch en onafhankelijk van andere componenten. Vandaar dat dit gedeelte van het onderzoek enkel onder de Literatuurstudie is ondergebracht. Bij het in productie stellen van de Kubernetes-cluster in de toekomst, zou mogelijks één van de besproken storageoplossingen in de praktijk geïmplementeerd worden.

1.5 Architectuur

Dit hoofdstuk beschrijft in het algemeen de visie van de architecturale schikking die in het begin van deze masterproef werd bedacht als zijnde een mogelijke oplossing voor werkpakketten 2 en 3 tezamen. Hiermee komen al enkele basisinzichten tot stand vooraleer de literatuurstudie en het effectieve design alsook de implementatie worden toegelicht.

Elk testbed bevat te testen software dat wordt getest door de zogenaamde tester of general access shell (gash). Testbedden bevatten meerdere testruns, en een testrun bestaat echter uit meerdere testen. Zo worden logfiles van één testrun als een pakket van resultaten gebundeld en verwerkt door de gash instantie. De bijhorende PHP-bestanden die automatisch worden gegenereerd, zorgen voor het aanleveren van de samenvattende indexpagina van de uitgevoerde of uit te voeren testen van een testrun. De resultaten van een testrun in de huidige opbouw worden lokaal op gash opgeslagen. Elke gash instantie van een testbed is op zijn beurt bereikbaar door een vast IP-adres voor elke eindgebruiker van de testruns. Hierdoor kan een eindgebruiker zowel tijdens het draaien van een testrun, alsook na de afloop ervan de testresultaten weergeven. Figuur 1 toont een simplistisch blokschema dat de te bekomen high-level architectuur voor het aanleveren van logfiles visueel weergeeft. De volgende paragraaf schept hier meer duidelijk over.



Figuur 1: Blokschema van de architectuur voor het aanleveren van logfiles

Doordat een testbed een vluchtig object wordt binnen de modernisering van het testlabo, dient er een oplossing gevonden te worden voor het aanleveren van de logfiles tijdens en na de afloop van een testrun. In eerste instantie dient er een HTTP-server voorzien te worden naast gash, die per testrun of testbed mee opstart. Zowel gash als de lokale HTTP-server gebruiken dezelfde storage en zijn vluchtige objecten. Verder dient er een persistente centrale HTTP-server voorzien te worden die gebruikmaakt van persistent storage om afgelopen testresultaten te hosten. Om testresultaten te transfereren vanuit de lokale naar de centrale storage dient er enerzijds een instantie voorzien te worden die de communicatie opzet en bestanden verstuurt, anderzijds een instantie die de bestanden ontvangt. De FTP-server staat in voor het ontvangen van de bestanden en deze op de persistent storage op te slaan. De ‘resulthandler’ daarentegen is ook een persistente instantie, die de communicatie tussen het vluchtige en het permanente verzorgt. Deze resulthandler heeft per testrun toegang tot de lokale storage, maar mag niet mee verwijderd worden na afloop van een testrun. De architectuur zoals in Figuur 1 zorgt dat er altijd x+1 HTTP-servers aanwezig zijn waarbij x gelijk is aan het aantal draaiende testruns. Er dient dus ook een manier gevonden te worden om een eindgebruiker naar de correcte HTTP-server door te verwijzen. Een reverse proxyserver of load balancer – indien er in de toekomst meer dan één instantie centraal draait – is hiervoor een mogelijke oplossing. Afhankelijk van de locatie van de testrun dient de HTTP-aanvraag naar de lokale HTTP-server van de desgewenste testrun geforward te worden, of naar de centrale HTTP-server zodra de testrun is afgelopen.

De overlopen architectuur geeft enkele concepten weer die in de Literatuurstudie verder worden onderzocht om effectieve keuzes te vergelijken i.v.m. software die mogelijks te gebruiken is binnen de effectieve implementatie voor het Aanleveren van logfiles. Hierbij dient opgemerkt te worden dat enkel de belangrijkste concepten worden vergeleken zoals de Webservers, de Load Balancers en de mogelijke Storage oplossingen.

2 Literatuurstudie

De literatuurstudie baseert zich afhankelijk van de mate van relevantie in deze masterproef op de technologieën die vandaag de dag gebruikt worden om een testbed op te zetten en samen te stellen. Daarom start dit hoofdstuk met de weg van virtualisatie naar containerisatie gecombineerd met orkestratie. Vervolgens wordt het concept Broadband Network Gateway (BNG) toegelicht, als zijnde een mogelijke applicatie die in het testbed bovenop de gevirtualiseerde service-router draait. Dit zijn allemaal reeds bestaande concepten of concepten die nog geïmplementeerd worden in de digital sandbox door Nokia. De literatuurstudie focust zich in dit deel voornamelijk op de werking en toepassingen van de technologieën. De nodige implementaties die dienen uitgevoerd te worden in opdracht van deze masterproef, worden dan ook sterk beïnvloed door deze technologieën.

Daarnaast focust de literatuurstudie ook op het maken van doordachte keuzes voor de implementaties in opdracht van deze masterproef. Deze implementaties van webservers en load balancers vinden plaats naast de service-routersoftware bovenop de gevirtualiseerde omgeving. Waar de service-routersoftware dient als het object dat getest wordt, zullen de webservers en load balancers dienen als instrumenten die de gebruikers helpen bij het testen van deze software. Dit concreet door logs visueel aan te leveren waarbij o.a. de performantie van het aanleveren van deze logs zo goed mogelijk dient te zijn.

Tot slot wijdt dit hoofdstuk nog uit over mogelijke storage oplossingen die dienen voor het persistent opslaan van logfiles. Echter is dit puur een theoretische benadering waar een vergelijking wordt gemaakt zonder een effectieve beste keuze naar voor te brengen. De implementatie van deze storage oplossing zal door Nokia pas na de testfase van de digital sandbox omgeving worden geïmplementeerd. Vandaar valt de implementatie niet verder onder de scope van deze masterproef.

2.1 Virtualisatie, containerisatie en orkestratie

Een testbed binnen Nokia draait vandaag de dag op een virtuele machine. Voor de digital sandbox omgeving stapt Nokia over op containerisatie in combinatie met orkestratiesoftware. Dit hoofdstuk focust zich op virtualisatie en de weg hiernaartoe. Vervolgens geeft dit deel de voor- en nadelen van virtualisatie die geleid hebben tot de overgang naar containerisatie. Tot slot komt orkestratie aan bod als technologie die het beheer van de gecontaineriseerde omgeving vereenvoudigt.

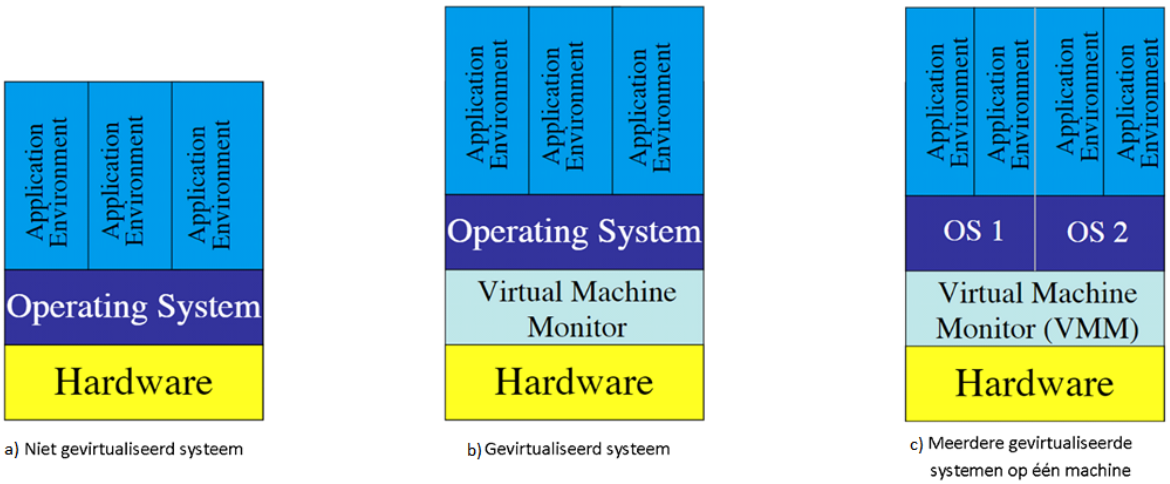
2.1.1 Virtualisatie

Dit hoofdstuk geeft een toelichting over de basisprincipes en opbouw van virtualisatie alsook de opkomst met de nadruk op de belangrijke gebeurtenissen in de geschiedenis van virtualisatie. Tot slot worden de voor- en nadelen verder toegelicht.

Wat is virtualisatie?

Virtualisatie is een paradigma dat leunt op het virtueel aanbieden van resources (alle benodigde computerhardware) aan operating systemen of toepassingen alsof deze resources volledig in het beheer zijn van het OS of de toepassing. Bijkomend, maakt virtualisatie het mogelijk om resources van verschillende al dan niet aparte fysieke apparaten te combineren tot een pool van resources die kunnen gebruikt worden door meerdere operating systemen of toepassingen op eenzelfde moment. Hiervoor is echter wel een extra abstractieniveau nodig die de verschillende virtuele resources coördineert, namelijk een hypervisor of Virtual Machine Monitor. De hypervisor interfereert met zogenaamde virtuele machines (of ook wel VM's genoemd) die bovenop de hypervisor draaien. Bovendien voorziet de hypervisor een blokkade voor acties die de VM niet zou mogen uitvoeren met betrekking tot het in gevaar brengen van naburige VM's of het gehele systeem. Vermits de hypervisor het enige contactpunt

is naar de hardware toe, is het ook mogelijk om verschillende operating systemen met elks een andere kernel te draaien. Dat wil zeggen dat op een enkele machine zowel bv. een Windows 10, Ubuntu 18.04 alsook Windows Server 2012 kan draaien. Dit fenomeen maakt het ook mogelijk om een enkele fysieke machine verantwoordelijk te stellen voor allerlei totaal aparte applicaties die bovenop het OS draaien. Figuur 2 toont een schematische voorstelling van een niet gevirtualiseerd systeem t.o.v. gevirtualiseerde systemen [4]–[6].



Figuur 2: Schematische voorstelling van gevirtualiseerde systemen b) en c), ten opzichte van een niet gevirtualiseerd systeem a) [6, pp. 3, 4]

Er zijn echter verschillende vormen van virtualisatie mogelijk. Gaande van hardwarevirtualisatie waar dit deel voornamelijk op focust, tot OS-virtualisatie. Ook applicatievirtualisatie is nog een voorbeeld van een mogelijk virtualisatieconcept [5]. De volgende paragraaf die over de Geschiedenis van virtualisaties uitweidt, overloopt kort enkele vormen.

Geschiedenis

Hoewel het begrip virtualisatie een vrij recente technologie lijkt te zijn, was het concept al bekend vanaf 1967 waarbij IBM het eerste systeem ontwikkelde. Het systeem kon opgedeeld worden in virtuele omgevingen waarbij elke gebruiker de illusie had op zijn eigen compleet systeem te werken. Dit was noodzakelijk omdat in die tijd mainframe computers veel fysieke ruimte nodig hadden en men bovendien ook geen fysieke toegang wilde verlenen tot deze ruimtes. A.d.h.v. virtualisatie konden geïsoleerde omgevingen worden aangeboden via terminaltoegang. Verder kon het enorme kostenplaatje dat aan dergelijke mainframes vasthing zo ook verdeeld worden over verschillende gebruikers. Computerelektronica evolueerde mee met zijn tijd waardoor componenten sneller en goedkoper werden. De personal computer is één van de bewijzen dat computerelektronica goedkoper alsook beschikbaar werd voor de gewone consument.

Rond 1980-1990 verdween virtualisatie stilaan in zijn oorspronkelijke vorm door de goedkoper wordende hardware en de verschillende operating systemen die werden ontwikkeld. Echter kwam er een nieuwe vorm van virtualisatie waar de Java programmeertaal een van de grondleggers is. Het idee was om een (Java-)applicatie op elke mogelijke systeemarchitectuur te kunnen draaien a.d.h.v. een gevirtualiseerde (Java-)omgeving bovenop een operating systeem. Dit maakte een applicatie overdraagbaar naar een apparaat onafhankelijk van het operating systeem.

Eind 1990, begin 2000 zagen organisaties het aantal servers stijgen afhankelijk van de hoeveelheid applicaties die ze draaiden. Elke server was vaak verantwoordelijk voor één specifieke applicatie om interferenties met andere applicaties te vermijden bij bv. updates en security-inbreuken. Het beheer van

deze fysieke serverparken werd als gevolg steeds onoverzichtelijker, en veel apparaten waren onvoldoende benut voor wat ze effectief aankonden. VMWare legde daarom de grondbeginselen voor een nieuwe vorm van virtualisatie. Die maakte het mogelijk om effectieve hardware zoals CPU, geheugen en opslag te virtualiseren. Eén fysieke machine kon als gevolg meerdere operating systemen draaien bovenop de hypervisor. Dit laatste concept is reeds in de kop 'Wat is virtualisatie?' geïntroduceerd [4]–[9].

Voor- en nadelen

Het virtualiseren van een omgeving brengt een heleboel voordelen met zich mee. Zoals reeds vermeld in het gedeelte over de Geschiedenis, is een cruciaal voordeel de kostenbesparing. Zowel de ontwikkelingskosten bij de opstart (ook wel Capital Expenditures = CaPex genoemd), alsook de operationele kosten (ook wel Operational Expenditures = OpEx genoemd) dalen significant volgens [10] en [11]. In tegenstelling tot voor elke applicatie een afzonderlijk systeem te hebben, is een systeem nu in staat om verschillende applicaties die draaien op verschillende operating systemen te hosten. Dit komt de efficiëntie en de flexibiliteit van het gebruik van de hardware ten goede. Daar bovenop is security ook een voordeel, elk operating systeem is volledig gescheiden van een ander OS op dezelfde machine. De hypervisor zorgt voor de afhandeling naar de hardware en er is geen rechtstreekse interactie tussen twee operating systemen mogelijk. Doordat hardware in een grote pool zit, kan er dynamisch hardware aan een systeem worden toegevoegd en verwijderd met als gevolg dat de schaalbaarheid zal stijgen. De beschikbaarheid van de services zullen als gevolg ook stijgen doordat fysieke hardware kan vervangen worden indien op voorhand kleine reserves aan resources worden voorzien. Tot slot is het belangrijk om weten dat het relatief eenvoudig is om snapshots van het hele systeem te maken om achteraf te gebruiken als back-up of als een nieuwe virtuele machine bovenop een hypervisor [5]–[7], [10].

Anderzijds zijn er ook enkele nadelen die parten spelen. Zo zal een abstractieniveau als een hypervisor overhead veroorzaken met als gevolg dat er rekening moet gehouden worden bij het deployen van tijdkritische processen [5], [12]. Afhankelijk van het type hypervisor is deze overhead echter minimaal in vergelijking met een fysieke machine [5]. Aansluitend, wordt met een virtuele omgeving een of meerdere single point(s) of failure gecreëerd. Elk fysiek systeem is verantwoordelijk voor meerdere virtuele machines. Indien er in de hardware zich een probleem voordoet, zal dit vermoedelijk gevolgen hebben voor alle virtuele machines bovenop deze hardware. Dit nadeel kan daarentegen wel opgelost worden door redundante hardware in te bouwen in de machines en door redundante machines te voorzien op verschillende fysieke locaties. Tot slot blijft een virtuele machine een omvangrijk object, dat een volledig operating systeem omvat met alle bijhorende bibliotheken. Het operating systeem zelf is in vele gevallen zelfs groter in omvang dan de applicatie die op het OS draait. Een hypervisor die als gevolg meerdere VM's van hetzelfde of gelijkaardig operating systeem draait, bevat heel wat dubbele of ongebruikte informatie die bij elke VM apart wordt verwerkt en opgeslagen [5], [12]. De oplossing tot dit nadeel wordt verder besproken in 2.1.2.

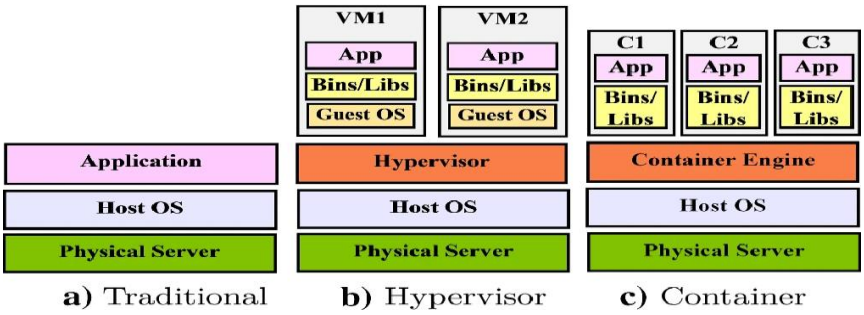
2.1.2 Containerisatie

Containerisatie of container-gebaseerde virtualisatie is een concept dat zich dicht aansluit tegen de term 'virtualisatie' [5], [12]. In tegenstelling tot kop 2.1.1 dat voornamelijk focust op hardwarevirtualisatie, focust dit hoofdstuk zich op de verschillen tussen virtualisatie en containerisatie. Daarnaast wordt doorheen dit hoofdstuk de aanleiding gegeven waarom containerisatie de dag van vandaag steeds populairder wordt, en stilaan de klassieke manier van virtualisatie – alleszins binnen de testomgeving van Nokia – meer op de achtergrond brengt.

Wat is containerisatie?

Containerisatie is een methodiek die zijn naam niet gestolen heeft op basis van zijn eigenschappen. Deze naam is afgeleid van containers zoals op cargoschepen, vrachtwagens en meer. Een container op zich is een autonome omgeving met vooraf vastgelegde eigenschappen zoals bv. afmetingen, dewelke in zekere mate los van zijn omgeving kan functioneren. Bovendien is een container voorzien om eenvoudig getransporteerd te worden waarbij alle nodige instrumenten om de container operationeel te houden, in de container zelf zitten. Het effectieve voordeel komt pas echt tot zijn recht eenmaal het proces van de container gestart wordt, namelijk de container plaatsen op de daarvoor voorziene omgeving [13].

In tegenstelling tot de klassieke virtuele machines, zijn containers een meer lichtgewicht-oplossing waarbij in de container veelal een gestripte versie van een operating systeem is terug te vinden. De container omvat enkel de nodige libraries en bestanden om de applicatie(s) te kunnen draaien met zo weinig mogelijk overhead. Docker is een voorbeeld van een containertechnologie die het mogelijk maakt om zogenaamde 'images' te creëren die vervolgens makkelijk te transporteren zijn naar alle andere Docker omgevingen. Vanuit deze images kunnen dan containers worden opgestart. Het grote verschil ten opzichte van een virtuele machine is de afwezigheid van een hypervisor, alsook de aanwezigheid van maar een enkele kernel. Een containertechnologie (zoals Docker) zal enkel het beheer van de containers op zich nemen en dus niet de rol van een hypervisor overnemen. Figuur 3 toont een schematische voorstelling van een omgeving met containers. Belangrijk zijn hier de gestripte 'Guest Operating Systems' die rechtstreeks de kernel van de effectieve host machine gebruiken. De verbinding tussen een applicatie in een container en de effectieve hardware is met minder omwegen dan een applicatie in een virtuele machine. Een container is met andere woorden meer in het systeem geïntegreerd. Logischerwijze kunnen er (zonder uitzonderingen) enkel gast besturingssystemen gebruikt worden die eenzelfde soort kernel delen met het host-systeem. Binnen verschillende Linux distributies zoals CentOS, Ubuntu, RedHat, … vormt dit dus geen problemen. Echter is het bv. niet zonder omweg mogelijk om Linux containers op een Windows systeem te draaien [5], [12]–[14]. Deze literatuurstudie focust zich enkel op de containerisatie van Unix-achtige systemen.



Figuur 3: Schematische voorstelling van een gecontaineriseerde omgeving c) t.o.v. een hardware gevirtualiseerde omgeving b) en een traditionele setup a) [15, p. 8587]

Geschiedenis

Container-gebaseerde virtualisatie is geleidelijk aan opgenomen door communities in de beginjaren 2000. FreeBSD – wat een Unix-like operating systeem is – introduceerde hier de zogenaamde 'FreeBSD Jails'. Deze technologie maakte gebruik van het 'chroot' (= change root) systeem dat eind jaren 70 werd geïntroduceerd in de Unix kernel. Dit systeem zorgt vandaag de dag nog steeds voor het isoleren van processen binnen een bepaalde map in het file systeem.

Kleine verbeteringen aan de toen bestaande containertechnologie werden mondjesmaat toegepast tot de introductie van zogenaamde 'Control Groups' in 2006. Control Groups of cgroups is een toevoeging in de Linux kernel die toelaat om systeem resources zoals processortijd of geheugen te limiteren en of te monitoren per proces of procesgroep. Hierop werden in 2008 'LXC' of ook wel de 'Linux containers'

geïntroduceerd. Linux containers waren op dat moment de meest betrouwbare en stabiele container-gebaseerde virtualisaties. Echter werd de technologie nog niet veelvuldig geïmplementeerd.

Bovenop de betrouwbare LXC technologie is in 2011 Cloud Foundry Warden begonnen met het uitrollen van een container-gebaseerde virtualisatietechnologie. Deze technologie wordt vandaag de dag niet meer verder onderhouden. Belangrijker is in 2013 de introductie van Docker. Docker heeft de markt rondom containerisatie compleet opengetrokken door zich o.a. te baseren op LXC, en daarbovenop een makkelijk te gebruiken interface te bouwen voor de communicatie tussen de containers en de gebruikers. Bovendien biedt Docker de mogelijkheid om meerdere applicaties met verschillende vereisten tegelijkertijd op één OS te draaien, alsook een applicatie te encapsuleren in een container image.

De containertechnologie bleef uiteindelijk verder evolueren tot in 2017 de introductie van Kubernetes kwam. Kubernetes is een container-orkestrator die vandaag de dag feilloos samenwerkt in combinatie met Docker. Kubernetes krijgt een diepgaandere focus in 2.1.3. De integratie van voornamelijk Docker in meerdere (cloud)platformen, heeft ervoor gezorgd dat vandaag de dag container-gebaseerde virtualisaties veelal gebruikt worden in productieomgevingen [5], [13], [14], [16].

Voor- en nadelen

Het merendeel aan voordelen van containers, vullen de nadelen van virtuele machines op. Zo is het voornaamste voordeel van een container ten opzichte van een virtuele machine de omvang. Een virtuele machine omvat een volledig operating systeem met alle libraries van het OS inclusief de applicatie. Een container daarentegen bevat enkel de benodigde libraries voor de applicatie, waarbij de kernel wordt gedeeld met het host OS. De performantie zal hier stijgen omdat de overhead van een hypervisor alsmede de overhead van de onnodige libraries zal dalen. Door zijn lichtgewicht maakt een container het ook mogelijk om een veel snellere opstarttijd te bekomen in vergelijking met een VM. Bovendien kan een container onafhankelijk herstart worden zonder dat het host OS een herstart moet ondergaan. De combinatie van de bovengenoemde voordelen komt de schaalbaarheid ten goede. Containers zullen als gevolg veel sneller op verschillende apparaten gedeployd kunnen worden afhankelijk van bv. de vraag naar de applicatie die draait in de containers [5], [13], [15].

Desalniettemin hebben containers ook nadelen. Wat betreft security wordt in de literatuur voornamelijk aangehaald dat virtuele machines een betere security hebben tegenover containers door de hypervisor die dient als tussenlaag. Daarentegen wordt in het algemeen wel vermeld dat verschillende eigenschappen van de Linux kernel zoals bv. cgroups kunnen helpen om de security te garanderen. Een omgeving met containers is veelal minder flexibel dan een hardware gevirtualiseerde omgeving. Doordat containers een enkele kernel delen met hun host, kunnen enkel Unix-achtige containers op een Unix-achtig systeem draaien. In tegenstelling tot containers, ondervinden virtuele machines hier geen problemen door de tussenliggende hypervisor waarlangs aanvragen naar de hardware passeren [5], [13], [15]. Tot slot hebben containertechnologieën (zoals bv. Docker) in het algemeen niet de mogelijkheid om aan centraal of overkoepelend beheer te gaan doen van verschillende gecontaineriseerde omgevingen. Containertechnologieën beperken zich voornamelijk tot hun eigen omgeving of machine die slechts minimale functionaliteiten voor opschaling voorziet. De laatstgenoemde tekortkomingen worden weliswaar opgevangen door een container-orkestrator zoals verder toegelicht in 2.1.3.

Combinatie van virtualisatie en containerisatie

In de praktijk worden regelmatig combinaties van hardwarevirtualisaties samen met container technologieën geïmplementeerd. Voornamelijk voor grote datacenters en (public) cloudproviders kan dit interessant zijn. Als voorbeeld heeft een public cloudprovider een aanzienlijk aantal fysieke machines. Indien een klant een door de provider beheerde Docker omgeving wil gebruiken, zal de

cloudprovider achterliggend virtuele machines hosten. Op deze virtuele machines zal vervolgens een containertechnologie zoals Docker worden gedeployd. De klant heeft in dit geval geen beheer van virtuele machines noch van de fysieke hardware. Echter is achterliggend wel hardwarevirtualisatie van toepassing zonder de klant hiervan de aanwezigheid opmerkt. Virtuele machines worden zo in de praktijk gebruikt om beschikbare resources (beter) te isoleren en de algemene toepasbaarheid te verhogen [5], [13], [17].

[17] toont a.d.h.v. uitgevoerde benchmarks een vergelijking tussen de performantie van een niet gevirtualiseerde machine t.o.v. een gevirtualiseerde machine met verscheidene besturingssystemen die draaien op de KVM-hypervisor. De resultaten tonen aan dat een extra virtualisatielaag waarop containers draaien wel degelijk een minimale overhead met zich meebrengt. Zo kan geconcludeerd worden dat er een minimaal performantieverlies van 4,89% optreedt bij de processorkracht voor de benchmark die de meest stabiele resultaten bekwam. Daarnaast is er voor het werkgeheugen slechts een afwijking in performantie van 3,74% op te merken voor KVM in vergelijking met een OS zonder hypervisor. De performantie van het opslaggeheugen is daarentegen zeer afhankelijk van het besturingssysteem dat op de hypervisor draait. Deze benchmarks geven in het algemeen een iets hoger performantieverlies dan CPU en memory. Op het vlak van netwerk gerelateerde benchmarks blijkt KVM bijna geen verlies in performantie te hebben. Het dient tot slot opgemerkt te worden dat in deze paragraaf KVM als enige hypervisor besproken werd. [17] toont benchmarks voor meerdere soorten hypervisors die elks hun eigen specialiteiten hebben. Als conclusie kan worden vastgesteld dat de resultaten een verwaarloosbaar verlies geven bij het gebruik van een virtuele machine afhankelijk van de specifieke toepassing en het gebruikte operating systeem. Het minimale performantieverlies van containers bovenop hardwarevirtualisaties weegt niet op tegen de voordelen. Daarom is de combinatie van virtualisaties en containerisaties voor datacenters vaak een meerwaarde.

2.1.3 Orkestratie

Orkestratie of ‘container orchestration’ zoals meestal in literatuur vermeld, is niet meer weg te denken in hedendaagse clusters die microservices aanbieden in de vorm van containers. Dit hoofdstuk focust zich op wat en waarom er een container-orkestrator is, welke soorten er zijn, alsook wat het kiezen van een container-orkestrator beïnvloedt. Tot slot wordt Kubernetes als container-orkestrator iets verder toegelicht aangezien die al dient binnen de huidige omgeving.

Wat is orkestratie?

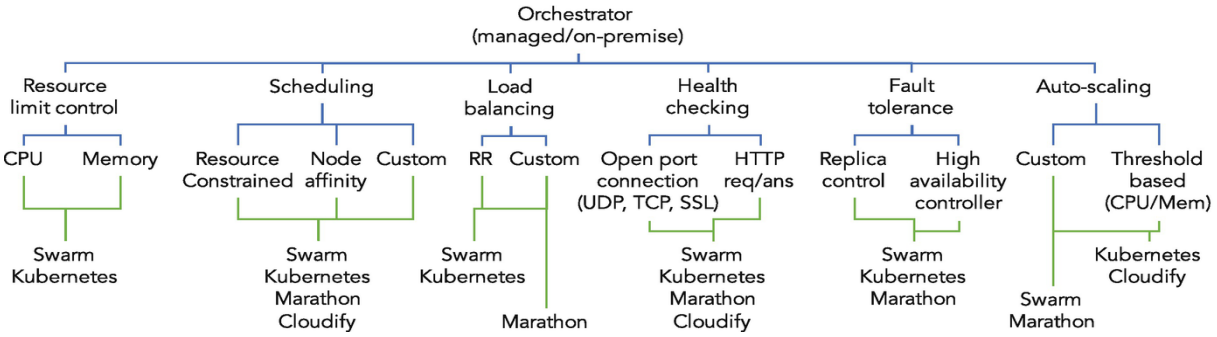
Gecontaineriseerde omgevingen hebben een heleboel voordelen zoals reeds in 2.1.2 vermeld. Er zijn echter ook enkele tekortkomingen die kunnen opgevuld worden door gebruik te maken van een container-orkestrator. Containers worden met regelmaat gebruikt in applicaties die zijn opgesplitst in microservices. Zulke microservices hebben doorgaans als functionaliteit dat ze makkelijk te schalen zijn afhankelijk van de vraag naar een bepaalde service. Bovendien moeten deze verschillende services als een coherent en overkoepelend geheel kunnen samenwerken onafhankelijk van hun fysieke locatie en het aantal replica’s op eender welk moment. De schaalbaarheid, beschikbaarheid, flexibiliteit en betrouwbaarheid vormen bij dergelijke omgevingen een cruciale rol.

Dat is het moment waar een container-orkestrator ingezet kan worden. In tegenstelling tot een containertechnologie (zoals bv. Docker) die het functioneren en isoleren van de container op zich neemt, zal een orkestrator de bredere scope van een zogenaamde ‘cluster’ op zich nemen [18]. Het dynamisch managen van containers, voorzien van communicatie tussen de containers en het internet, schalen afhankelijk van de behoefte, load balancen, … zijn de voornaamste taken van een orkestrator. De orkestrator zal bepalen welke container op welke machine op welk tijdstip wordt gedeployd of vervangen, naargelang de status van de applicatie in de cluster [19], [20]. Dergelijke clusters zijn vaak

ook voorzien om meerdere applicaties simultaan en gescheiden van elkaar te draaien, met één of meerdere bestuursorganen dewelke de orkestrator is. De orkestrator begeleidt het volledige proces dat een container ondergaat van begin tot einde [20]. Daarentegen is hij geen vervanger van de containertechnologie, maar wel een instrument dat samenwerkt met de containertechnologie.

Soorten

Container-orkestrators hebben over het algemeen bepaalde eigenschappen die kenmerkend zijn om in de literatuur te voldoen aan een container-orkestrator. [20] toont aan de hand van een schematisch overzicht de voornaamste eigenschappen voor een container orchestrator. Figuur 4 toont dit schematisch overzicht met de belangrijkste eigenschappen. Onderaan de figuur zijn de meest voorkomende container-orkestrators zichtbaar. Enerzijds zijn Kubernetes, (Docker) Swarm, Marathon (uitbreiding op Apache Mesos) en Cloudify container-orkestrator frameworks die volledig in eigen beheer zijn. Anderzijds bestaan er ook 'container orchestrators as a service', dewelke door een cloudprovider achterliggend onderhouden worden, maar ingezet en geconfigureerd kunnen worden door de beheerder. Zo heeft Google de zogenaamde GKE, Amazon de Amazon ECS en Microsoft de zogenaamde Containerservices [20].



Figuur 4: Schematisch overzicht van eigenschappen van een container-orkestrator bovenaan, en de bijhorende technologieën die deze eigenschappen implementeren onderaan [20, p. 225]

Volgens de literatuur zijn er drie freeware container-orkestrators die de meeste aandacht krijgen, met als gevolg in productieomgevingen het meest gebruikt worden. Namelijk Docker Swarm, Marathon van Apache Mesos en Kubernetes, die ook vermeld staan in Figuur 4. Docker Swarm wordt aanzien als de meest eenvoudige implementatie die naadloos samenwerkt met een bestaande Docker omgeving. Daarenboven lijkt dit de perfecte oplossing voor kleinere omgevingen waar de eenvoudigheid van implementatie primeert.

Marathon is een container-orkestrator die in combinatie met een bestaande Apache Mesos cluster kan werken. Deze orkestrator is voornamelijk geschikt voor zeer grote clusters die in productieomgevingen operationeel moeten zijn.

Tot slot is Kubernetes de container-orkestrator met de meeste functionaliteiten die ook default op Docker berust als containertechnologie. Daarnaast is het opzetten van die orkestrator de meest complexe en tijdrovende in vergelijking met Docker Swarm en Marathon. Verder is Kubernetes ook de traagste als het gaat over de failovertijd van een host machine. M.a.w. de tijd die nodig is om een machine opnieuw op te starten. Afgezien daarvan is gebleken dat Kubernetes veruit de snelste container-orkestrator is voor het opnieuw opstarten bij een falende container. Daarnaast is ook de performantie in vergelijking met Docker Swarm in vele gevallen significant beter [19], [21]–[25].

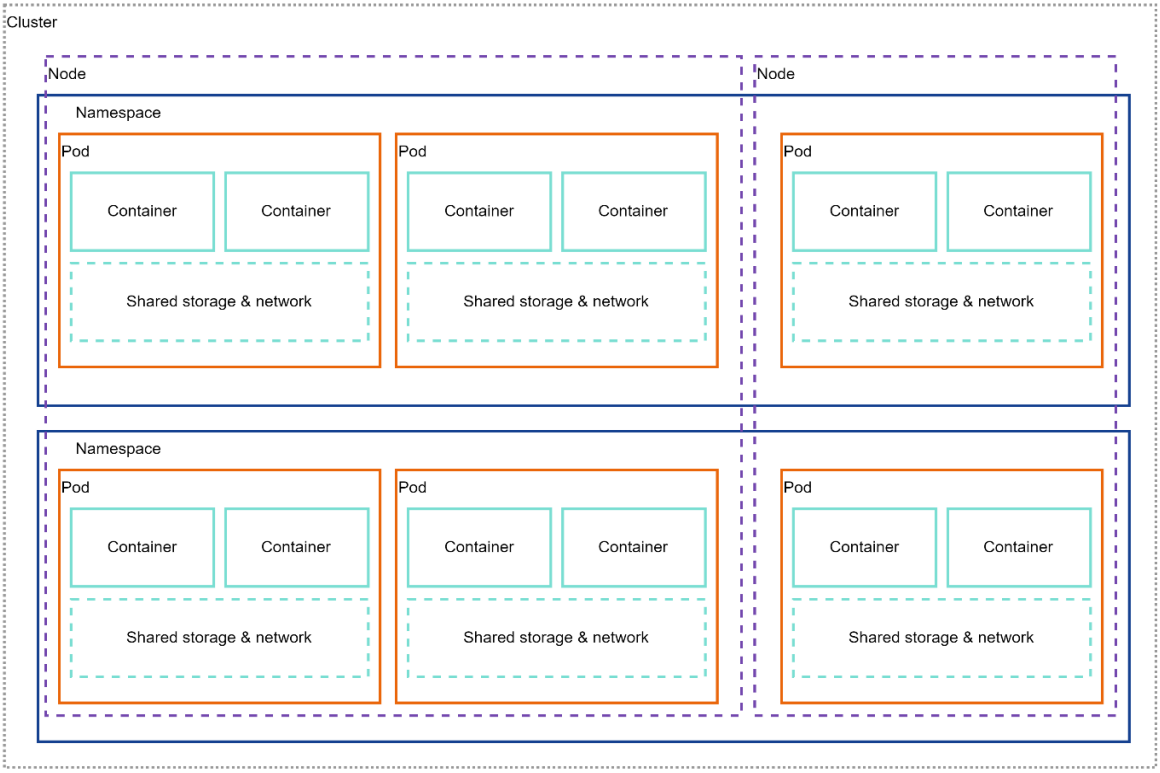
Deze literatuurstudie, alsook volledige masterproef focust zich op Kubernetes als container-orkestrator. Nokia had al voor de start van deze masterproef gekozen om Kubernetes als container-orkestrator te gebruiken vanwege zijn functionaliteiten, performantie en het vrij en gratis gebruik ervan voor o.a. commerciële doeleinden. Bovendien is Kubernetes ook de standaard als container-orkestrator in de industrie [19].

Kubernetes

Aangezien Kubernetes een belangrijk deel uitmaakt van de gehele masterproef wordt hier beknopt de structuur en opbouw van een Kubernetes-omgeving benadrukt, alsook de nodige componenten om dit te realiseren.

Structuur

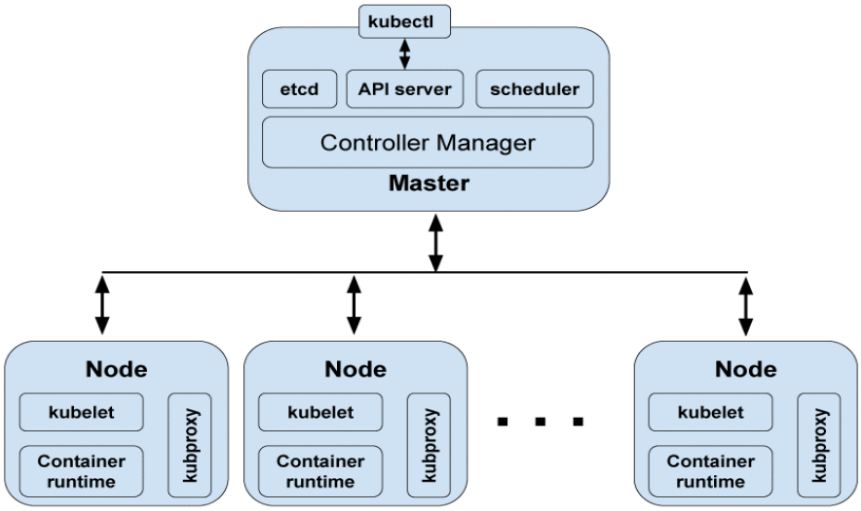
Waar containers kunnen vergeleken worden met containers op een vrachtschip zoals vermeld in paragraaf 2.1.2, kan Kubernetes vergeleken worden met de kapitein van dat vrachtschip waarbinnen containers een plaats hebben toegekend gekregen [19]. Figuur 5 toont een structuur van een mogelijke Kubernetes-omgeving. Kubernetes draait zogenaamde ‘pods’ die onderling geconnecteerd zijn, waarbinnen een of meerdere soortgelijke containers verpakt zitten. Pods zijn als gevolg logische units waarbinnen containers alsook gedeelde resources zoals opslag en netwerkelementen zitten. Een combinatie van pods zijn vervolgens logisch gebundeld als een zogenaamde ‘namespace’. Het grootste logische element is de zogenaamde cluster, waarbinnen meerdere namespaces aangemaakt worden. Pods worden uiteindelijk op ‘nodes’ gezet, dewelke fysieke of virtuele machines voorstellen die deel uitmaken van de cluster [26]. Bovendien heeft Kubernetes de mogelijkheid om allerlei containertechnologieën aan te spreken die op hun beurt de effectieve afhandeling van de containers uitvoeren. Docker is hierbij veruit de meest populaire containertechnologie die wordt gebruikt in combinatie met Kubernetes [18]–[20]. De opbouw die Kubernetes gebruikt om componenten in de cluster zoals zichtbaar in Figuur 5 aan te sturen, wordt in de volgende paragraaf meer verduidelijkt.



Figuur 5: Structuur van een voorbeeld Kubernetes-omgeving

Opbouw

Een cluster bestaat uit een ‘control plane’ dat worker nodes alsook pods op worker nodes gaat beheren. Het control plane op zich is ook een pod die binnen dezelfde cluster draait op een gereserveerde namespace ‘kube-system’. De pod kan binnen een gedeelde worker node actief zijn, maar wordt afgeraden indien deze cluster als productieomgeving wordt gebruikt. Daarom wordt voor performante omgevingen geopteerd om een of meerdere aparte zogenaamde master node(s) te voorzien, die de verantwoordelijkheden van het control plane op zich nemen [26]. Figuur 6 toont de schematische weergave van de opbouw van een Kubernetes-cluster met zijn componenten. Bovenaan de figuur staat de master node – ook het control plane genoemd – en onderaan de worker nodes. Het volledige schema kan aanzien worden als één cluster.



Figuur 6: Schematische weergave van de opbouw van componenten in Kubernetes [18, p. 13]

De master node(s) is verantwoordelijk voor het beheer van de worker nodes. Deze master nodes bestaan default uit vier componenten die terug te vinden zijn in de volgende opsomming [18], [26]:

1. API-server
   De API-Server verzorgt de communicatie tussen de beheerders alsook tussen de verschillende geconnecteerde nodes. Het is met andere woorden de front-end communicatietool van de cluster. Kubectl is een CLI (Command Line Interface) tool die het mogelijk maakt om als beheerder gestructureerd te kunnen communiceren met de cluster.
2. Scheduler
   De scheduler is vervolgens een component die het toekennen van pods aan bepaalde nodes verzorgt. Afhankelijk van vooraf vastgelegde voorschriften zoals bv. vereisten voor resources of applicaties die op eenzelfde node moeten staan, beslist de scheduler de plaats van de pods.
3. Etcd
   Etcd is een beveilgde gedistribueerde opslag waarin cruciale clusterinformatie is opgeslagen. Alle informatie i.v.m. de cluster is dus beschermd tegen ‘single points of failures’ indien meerdere master nodes in de cluster aanwezig zijn.
4. Controller manager
   De controller manager is verantwoordelijk voor het opvolgen van de status en het nastreven van de gewenste toestand van de cluster a.d.h.v. de API-server. Deze component grijpt bijvoorbeeld in als een bepaalde node uitvalt.

Naast de master nodes, zijn er ook worker nodes die onder leiding van de master nodes (of het control plane) staan. Worker nodes zijn verantwoordelijk voor het draaien van containers die de beheerder in zijn cluster wil. Onderstaande opsomming overloopt de belangrijkste componenten van een worker node met telkens zijn functies. Deze elementen zijn gebaseerd op [18] en [26]:

1. Kubelet
   De kubelet is verantwoordelijk voor containers die in zijn node moeten draaien. De component moet de gezondheid van de containers continu monitoren en bijsturen indien nodig.
2. Kube-proxy
   De kube-proxy is een component dat het concept van 'services' in Kubernetes verzorgt per node. Een service is verantwoordelijk voor het netwerkmatige geheel binnenin de cluster. Kube-proxy zal dus de communicatie tussen pods, de cluster en eventueel het internet verzorgen afhankelijk van vooraf vastgelegde instellingen.
3. Container runtime
   De container runtime is de effectieve omgeving waar containers worden opgestart en zullen draaien. Zoals eerder vermeld, zal Docker als containertechnologie het vaakst voorkomen in de literatuur. Alsook, zal Kubernetes in combinatie met Docker in deze masterproef de focus krijgen.

2.2 BNG

De digital sandbox omgeving van Nokia draait om het testen van (software van) service-routers. Een mogelijke en veelgebruikte functie of complete productoplossing van deze service-routers is de 'Broadband Network Gateway' (BNG). De huidige implementatie maakt gebruik van virtuele machines om de software te testen, en gaat stilaan over naar containers. Aangezien de BNG als productoplossing regelmatig in de testomgevingen wordt geïmplementeerd, is het een wezenlijke meerwaarde om minimaal de basisfuncties alsook de reden van implementatie van een BNG te onderzoeken. Dit hoofdstuk focust zich op de basisfunctionaliteit van een BNG, alsook beknopt over de nieuwste implementatiemogelijkheden met betrekking tot het opsplitsen van zijn componenten.

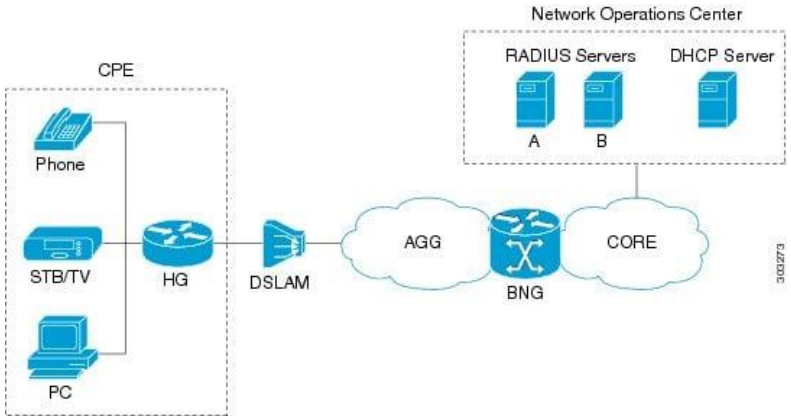
2.2.1 BNG

Breedband netwerk of in de volksmond ook 'internet' genoemd is heden ten dage niet meer weg te denken uit de maatschappij. Dit netwerk heeft doorheen de jaren ook steeds meer verantwoordelijkheden toegekend gekregen. Gaande van het klassieke internetten of gamen, tot streaming, telefoonverkeer en nog veel meer. Dergelijke faciliteiten worden in het algemeen aangeboden door een zogenaamde internetserviceprovider (ISP). Daarenboven kan een klant via verschillende kanalen deze ISP bereiken, gaande van (V)DSL-lijnen en coaxkabels tot zelfs glasvezel, 4G en 5G [27]. Echter moeten deze kanalen op een bepaald punt samenkomen om vervolgens gestructureerd gegevens van de ISP of van het internet te verkrijgen.

Wat is een BNG?

De locatie net voor trafiek het ISP-netwerk binnenkomt, dat is het punt waar de BNG of 'Broadband Network Gateway' van cruciaal belang is. De BNG dient als een toegangspunt voor klanten – ook wel subscribers genoemd – tot het netwerk van de ISP. Voor de ISP dient de BNG als een subscriber management tool die de juiste service aan de juiste klant aanlevert met de juiste prioriteiten. De BNG kan in een zeer simplistische vorm aanzien worden als een router met enkele fundamentele extra functionaliteiten. Figuur 7 toont een eenvoudige weergave van een topologie waarin een BNG centraal staat. De linkerkant van de afbeelding voorgesteld door 'Customer Premise Equipment' (CPE), stelt een

klantennetwerk voor waarin bv. apparaten zoals computers, laptops en telefoons d.m.v. een router/modem met de ISP verbonden zijn. Tussen de BNG en de router/modem van de klant kunnen er apparaten staan die verantwoordelijk zijn voor de fysieke aggregatie van kanalen zoals bv. het DSLAM-apparaat in dit geval. Hierop wordt verder niet gefocust. De rechterkant van de afbeelding daarentegen stelt het ISP-netwerk en of het internet voor, dat bepaalde services aan de klant aanbiedt afhankelijk van bv. het abonnement van de klant [28], [29]. Het gedeelte over de Functies van de BNG gaat hier dieper op in.



Figuur 7: Simplistische architectuur van een BNG [28, p. 18]

Functies

In Figuur 7 zijn enkele services getekend in het zogenaamde Network Operations Center. Deze services zijn voorzien om als functie binnen een BNG te functioneren, of als aparte service te kunnen samenwerken met de BNG, afhankelijk van het type BNG. Dit deel overloopt enkele belangrijke functies van een algemene Broadband Network Gateway.

Alvorens de BNG een of meerdere subscribers toegang geeft tot het ISP-netwerk of het internet, moet er een vorm van authenticatie gebeuren. Hiervoor kan bijvoorbeeld RADIUS gebruikt worden als een service. RADIUS voorziet de AAA-functies, wat voor Authenticatie, Autorisatie en Accounting staat. Naast authenticatie, zal het autorisatiegedeelte de subscriber toegang geven tot de benodigde resources. M.a.w. afhankelijk van welk abonnement de subscriber bij zijn ISP heeft, komen bepaalde diensten ter beschikking. Overigens betekent dit ook dat de ISP d.m.v. de BNG de bandbreedte per subscriber kan beheren. De accounting tot slot, stelt de ISP in staat om het gebruik van bepaalde services afhankelijk van vooraf ingestelde parameters te factureren naar de subscriber toe.

Vervolgens is de DHCP-service binnen dit concept ook een cruciale functie om internet aan te leveren aan klanten. Elke subscriber krijgt doorgaans een publiek IP-adres van de DHCP-server van de ISP waardoor hij bereikbaar is vanop het internet. De BNG zal meestal als DHCP Proxy dienen om aanvragen door te sturen naar de subscriber.

Tot slot is de BNG ook een middel dat de Quality Of Service (QoS) van trafiek behandelt. Zo zal bijvoorbeeld een televisiesignaal of telefoonsignaal doorgaans voorrang krijgen op een gewoon internetsignaal. Hierdoor kan bv. een telefoongesprek gevoerd worden met zo min mogelijk vertraging [28], [29].

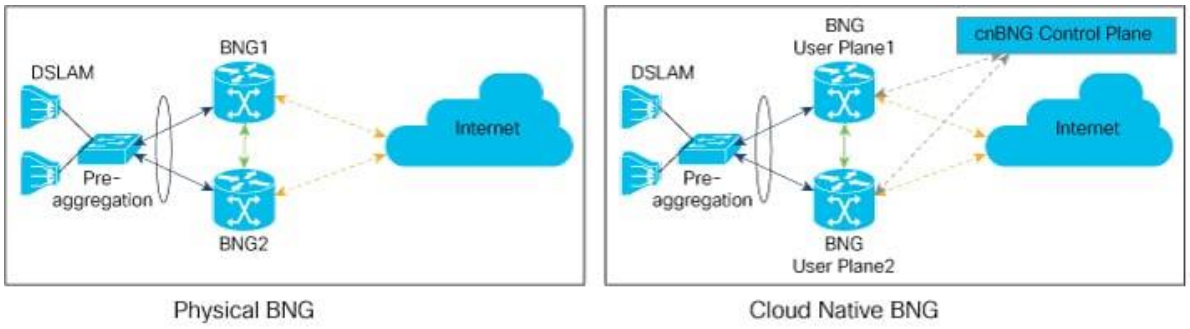
De specifieke functies van een BNG verschillen echter van verkoper tot verkoper. Daarnaast kunnen de functionaliteiten ook verschillen afhankelijk van het model of versies van de firmware. Bij het gebruik van dergelijke apparatuur is het aangeraden om telkens de corresponderende datasheet te raadplegen.

2.2.2 Disaggregating BNG

Dit hoofdstuk beschrijft kort hoe de BNG is geëvolueerd doorheen de jaren, en hoe in de toekomst deze nieuwe vorm kan worden geïmplementeerd a.d.h.v. containerisatie.

BNG heeft afgelopen decennia heel wat progressie ondergaan. Startend van de steeds hogere vraag naar netwerk gerelateerde toepassingen waardoor 'Ethernet Based Broadband Aggregation' de voorkeur kreeg zoals beschreven in [30] door het Broadband forum. Vervolgens werd door dezelfde organisatie een nieuwe update uitgebracht die de zogenaamde 'hybrid access' beschrijft. Deze hybride oplossing is een combinatie van een bedraad (bv. DSL) en een draadloos (bv. 3G) netwerk waardoor betrouwbaarheid en snelheid toenemen [31]. Daaropvolgend werd een nieuwe architectuur gereleased doorheen de evolutie van de BNG met het oog op cloudoplossingen en 5G. Oorspronkelijk kwam deze invloed uit de mobiele (5G Core architecture) wereld, gedefinieerd door 3GPP. Deze architectuur focust zich voornamelijk op het ontkoppelen van de functies van een BNG, in het Engels ook 'disaggregating' genoemd. Het ontkoppelen moet zorgen voor een betere flexibiliteit en schaalbaarheid. Een Disaggregated BNG krijgt daarom als afkorting DBNG [32], [33]. Dit is waar anno 2022-2023 ook Nokia nog onderzoek naar uitvoert.

CUPS of ook 'Control User Plane Seperation' wordt binnen Nokia gebruikt om het ontkoppelen van de functies van een BNG te organiseren tot een DBNG zoals zichtbaar in Figuur 8. Dit maakt het mogelijk om user plane en control plane apart te gaan schalen afhankelijk van de vraag. Waar de user plane voornamelijk moet schalen afhankelijk van de benodigde bandbreedte per subscriber, moet de control plane schalen afhankelijk van het aantal geconnecteerde users en het aantal sessies die de BNG moet opzetten. Deze architectuur laat het ook toe om een centraal control plane te hebben in een datacenter en meerdere user planes gedistribueerd over een geografisch gebied dichter bij de eindgebruikers. Voorheen werden beide delen op een enkel apparaat gevestigd (zoals zichtbaar in het linkerdeel van Figuur 8) waardoor de opschaling duur was en dus ook resources niet optimaal werden benut. CUPS introduceert hier een meer schaalbaar alsook meer kostenefficiënte oplossing [32]–[34]. Ook dit is net hetgeen waar container-gebaseerde virtualisatie in combinatie met een orkestrator op focust en voor gemaakt is. Met andere woorden zal de digital sandbox in de toekomst nog efficiënter kunnen omgaan met DBNG's die worden getest in een testbed.



Figuur 8: Verschil tussen BNG (linkse deel) en DBNG (rechtse deel) gebruikmakend van CUPS [35, p. 11]

2.3 Webservers

Webservers zijn vandaag de dag niet meer weg te denken in de huidige maatschappij. Ze worden veelal gebruikt om allerlei applicaties a.d.h.v. webpagina's ter beschikking te stellen naar de eindgebruiker toe. Deze webpagina's worden aangeleverd door een HTTP-server. In deze masterproef zullen HTTP-servers enerzijds gebruikt worden om logs van een apart testbed te tonen (werkpakket 2), anderzijds om geaggregeerde logs voor eindgebruikers op een centrale plaats te visualiseren (werkpakket 3).

2.3.1 Keuze

Doordat twee van de drie werkpakketten een webserver als basis gebruiken, is een doordachte keuze cruciaal. Deze doordachte keuze wordt gemaakt door veelvoorkomende webservers te vergelijken op basis van de volgende criteria:

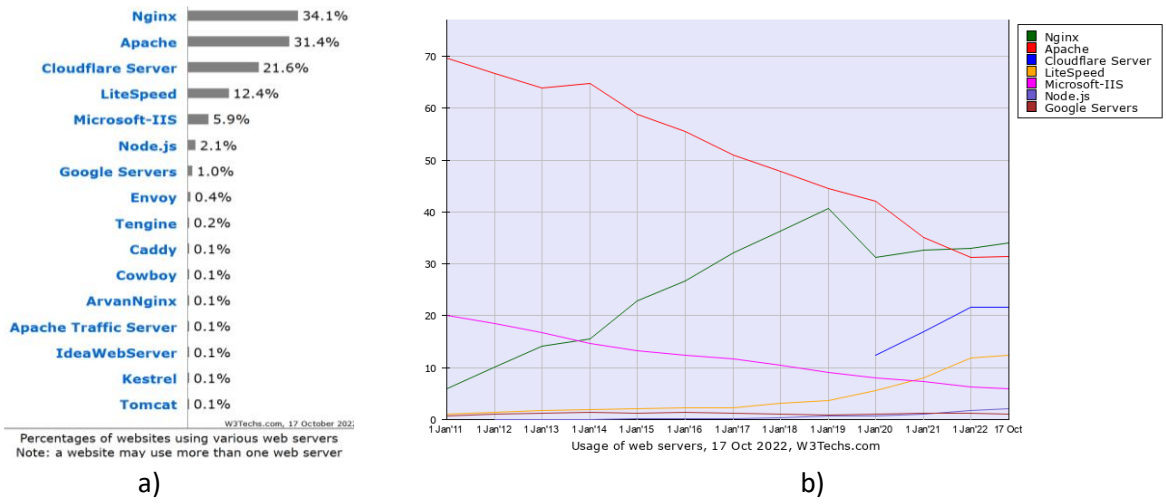
- Populariteit
- Performantie
- Containerisatie-mogelijkheden

De populariteit of het vertrouwen van ontwikkelaars om een webserver te gebruiken is een eerste punt dat bepaald welke webservers verder worden vergeleken doorheen dit hoofdstuk. In combinatie met de populariteit dient er ook rekening gehouden te worden met de betaalbaarheid van de service. Hierbij wordt er door Nokia voorkeur gegeven aan een freeware opensourceoplossing. De snelheid of performantie is daarenboven ook een belangrijk item dat er voor zorgt dat engineers die de webpagina raadplegen efficiënt te werk kunnen gaan en dus niet te lang moeten wachten per pagina die moet ingeladen worden. Echter zal – zoals aangehaald in het derde werkpakket – een load balancer worden toegevoegd om de aanvragen proportioneel te verdelen over verschillende instanties. Kop 2.4 zal de keuze en vergelijking van load balancers verder uitdiepen. Tot slot is de mogelijkheid tot containerisatie van een webserver fundamenteel in een opbouw waar containers een centrale rol spelen. Een webserver die reeds een containerimage ter beschikking heeft zal op dit punt beter scoren dan een webserver waar nog zelf een containerimage dient voor ontwikkeld te worden.

Populariteit

Populariteit slaat enerzijds op hoe ontwikkelaars webservers vertrouwen als service voor het hosten van hun content. Dit vertrouwen kan o.a. uitgedrukt worden in het totale marktaandeel van die bepaalde webserver. Anderzijds, slaat dit op hoe de community van deze webserver wordt onderhouden.

Volgens W3Techs zijn er twee webservers (oktober 2022) die het overgrote aandeel van de markt domineren zoals zichtbaar in Figuur 9a en Figuur 9b. Namelijk Nginx met een marktaandeel van 34,1%, waarna Apache met een marktaandeel van 31,4%. Evenwel heeft Cloudflare Server ook een aanzienlijk aandeel op de markt, maar dit is een cloud serviceprovider met als gevolg dat die niet binnen de scope van deze masterproef valt. LiteSpeed daarentegen vertegenwoordigt 21,6% van de markt en is ook gedeeltelijk een cloud serviceprovider, maar biedt anderzijds ook een freeware alternatief OpenLiteSpeed aan. Vervolgens komt Microsoft-IIS aan bod met een aandeel van 5,9% dewelke enkel op Windows Server kan draaien. Dit heeft als gevolg dat de integratie in de Kubernetes-cluster onnodig complex zou worden [36]. Eveneens zijn Microsoft IIS alsook Apache sinds 2011 sterk gedaald in populariteit, waardoor Nginx momenteel het grootste aandeel op de markt vertegenwoordigd zoals zichtbaar in Figuur 9b. Daarnaast is sinds 2019 ook een opmars zichtbaar van de Cloudflare en LiteSpeed cloudproviders [37], [38]. Echter zijn er nog tal van andere minder populaire mogelijkheden zoals bv. Caddy welke een lichtgewicht secure webserver is die in Go is geschreven, Tomcat dewelke een webserver is voor Java-applicaties, of H2O dewelke een webserver is speciaal voor gebruikers met een minder performant systeem [39], [40].



Figuur 9: a) Het gebruik van webservers over het internet op 17 oktober 2022
b) Procentuele weergave van het marktaandeel van de meest gebruikte webservers over het internet doorheen de jaren vanaf januari 2011 tot en met oktober 2022 [36], [37]

Zowel Apache, Nginx als LiteSpeed vertegenwoordigen beide een aanzienlijk aandeel op de markt als het gaat over mogelijks lokaal te hosten webservers. Daarnaast is Apache volledig gratis en vrij te gebruiken en bestaat er een grote community rond deze software. Nginx daarentegen is grotendeels gratis en vrij te gebruiken, exclusief enkele features die betalend zijn. Het freeware gedeelte van Nginx omsluit genoeg eigenschappen om een volwaardige webserver op te zetten. Evenwel heeft Nginx een community dat kan gebruikt worden door niet-betalende gebruikers. OpenLiteSpeed is tot slot ook een volwaardig softwarepakket dat gratis te gebruiken is, en eventueel is uit breiden naar de enterprise variant om meerdere features in te schakelen. De gratis versie heeft een actieve en degelijke community rond zich, waarnaast de betalende versie professionele support biedt.

De populariteit en kosten van de webservers hebben bepaald dat dit hoofdstuk zich verder focust op enkel Apache, Nginx en OpenLiteSpeed. De uiteindelijke keuze wordt verder beslist door de performantie, schaalbaarheid en containerisatie-mogelijkheden.

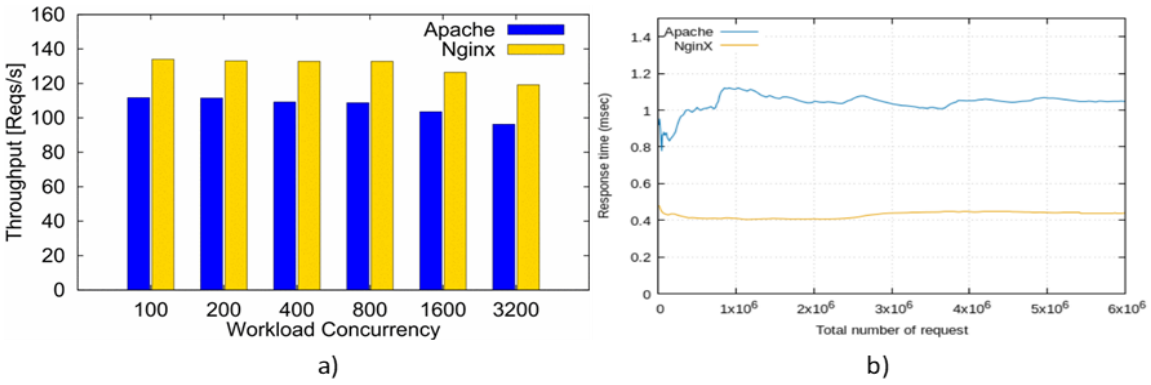
Performantie

Om het meest nauwkeurige resultaat te bekomen over de performantie van een webserver in een omgeving, zou elke webserver moeten getest worden met telkens dezelfde parameters in de huidige context. Echter is dit qua tijdsgebrek onmogelijk, maar wordt dit gecompenseerd door gebruik te maken van algemene performantietesten die in de literatuur zijn uitgevoerd.

Allereerst, focust dit deel zich op de performantie vergeleken tussen de twee bekendste HTTP-servers, namelijk Apache en Nginx. [41] en [42] tonen aan dat Nginx de meest performante webserver is op vlak van het aantal aanvragen per seconde die kunnen afgehandeld worden met dezelfde hardware. Een mogelijke verklaring is dat Nginx gebruikmaakt van asynchrone events t.o.v. threads die Apache gebruikt voor elke connectie af te handelen. Meerdere aanvragen kunnen als gevolg binnen eenzelfde thread worden verwerkt, waardoor Nginx een grotere buffer heeft zonder dat er meer systeemresources verbruikt worden. Als gevolg zal de schaalbaarheid van een Nginx webserver beter zijn dan die van Apache [41], [43], [44].

Anderzijds toont [45] dat in een gecontaineriseerde omgeving er bepaalde scenario's zijn waar het gemiddeld aantal aanvragen per seconde die kunnen afgehandeld worden bij een Nginx server minder zijn dan bij een Apache server. Echter over het algemeen heeft Nginx in de meeste literatuur een hogere

performantie op vlak van reactietijd, alsook op vlak van het totaal aantal aanvragen dat per seconde kan worden afgehandeld [41]–[44], [46], [47]. Figuur 10a toont een diagram waar zichtbaar is dat Nginx meer aanvragen tegelijkertijd kan afhandelen vergeleken met Apache. Bovendien toont Figuur 10b de reactietijd van de webservers afhankelijk van het totaal aantal verzonden aanvragen. Nginx is hier gemiddeld gezien meer dan de helft sneller in vergelijking met Apache [44], [46], [47].



Figuur 10: a) Het aantal gelijktijdige aanvragen afgehandeld door Apache en Nginx afhankelijk van de belasting b) De reactietijd van Apache en Nginx webservers afhankelijk van het aantal uitgevoerde aanvragen [46, p. 32], [47, p. 41]

Op het gebied van performantie is Nginx een betere keuze dan Apache indien er een vergelijking tussen de twee meest populaire webservers wordt gemaakt. Echter is er een derde opkomende partij die ook het overwegen waard is. Namelijk de opensource versie van LiteSpeed namelijk OpenLiteSpeed (OLS). Helaas is er bijzonder weinig wetenschappelijke literatuur die zowel Nginx als OLS bespreken. Daarom zal in de volgende paragraaf gefocust worden op enkele blogs en hun bijhorende documentatie om een verantwoorde keuze te maken.

Volgens de website van OLS zelf, zou hun webserver vijfmaal meer aanvragen per seconde kunnen afhandelen dan Nginx als het gaat over cached websites [48]. Volgens [49] zou OLS zonder caching ruwweg 1,2 keer meer aanvragen succesvol kunnen afhandelen gedurende één minuut waar 1000 gebruikers elke seconde gelijktijdig connecteren. De gemiddelde reactietijd is weliswaar vier keer trager dan Nginx. Verder toont [50] een resultaat waar 10 gebruikers 100 gelijktijdige aanvragen naar de server sturen, met de resultaten zichtbaar in Tabel 1. Hieruit kan worden geconcludeerd dat Nginx dubbel zoveel uncached aanvragen per seconde kan verwerken ten opzichte van OLS. Met als gevolg dat het voltooien van de 1000 aanvragen bijna dubbel zo snel is.

Tabel 1: Benchmark van Nginx en OpenLiteSpeed als 10 gebruikers gelijktijdig 100 aanvragen sturen naar de webserver waarbij niet wordt gebruik gemaakt van caching [50].

Metrics (10 uncached clients)	Nginx	OpenLiteSpeed
Completion	24.95 seconds	43.54 seconds
Requests	40.09 req/s	22.97 req/s
Throughput	1.92 MB/s	1001 KB/s

OLS lijkt een waardige kanshebber indien caching een nuttige bijdrage kan leveren. Nginx daarentegen is performanter dan Apache en dan OpenLiteSpeed indien caching in minder mate een bijdrage kan leveren [50], [51]. In deze masterproef worden bestanden dynamisch aangeboden waarbij telkens het volledige bestand opnieuw vanuit de server wordt gedownload. Oorspronkelijk met als bedoeling dat het bestand eventueel kan gewijzigd zijn doorheen een testrun. Dit is echter een overblijfsel van verouderde principes die in deze masterproef dienen overgenomen te worden.

Containerisatie-mogelijkheden

Kubernetes heeft als verantwoordelijkheid containers te beheren binnen een cluster. Kortom zijn containers dan ook cruciaal om de omgeving optimaal te laten draaien. Dit hoofdstuk focust kort op de beschikbaarheid van containerimages van webservers die in vorige paragrafen als mogelijkheden werden bekeken.

Zowel Apache als Nginx hebben een officiële Docker image ter beschikking op de Docker Hub. OpenLiteSpeed heeft echter ook een image op Docker Hub beschikbaar, maar wordt binnen Docker nog niet aanzien als een officiële image zelf.

Daarnaast kan Helm ook gebruikt worden als package manager voor Kubernetes. Helm gebruikt zogenaamde 'charts' om applicaties te gaan deployen op de cluster. Zowel Apache als Nginx hebben geverifieerde en gepubliceerde charts beschikbaar. OpenLiteSpeed daarentegen heeft helemaal geen charts ter beschikking.

Conclusie

Op basis van onderzoek over populariteit, performantie en containerisatie-mogelijkheden kan worden besloten dat Nginx voor deze masterproef de beste keuze is. Nginx is de dag van vandaag de meest globaal gebruikte webserver die gratis ter beschikking is voor corporate omgevingen waar maandelijks updates voor worden uitgebracht. Bovendien is de performantie en schaalbaarheid van deze webserver beter dan zijn freeware alternatieven op de markt. Tot slot heeft Nginx veel mogelijkheden om te containeriseren, wat de implementatie vereenvoudigd.

2.4 Load Balancers

Applicaties zoals webservers hebben een maximumaantal aanvragen die ze kunnen verwerken per tijdsinterval afhankelijk van de beperkingen van hardware en software. Meerdere instanties van applicaties kunnen bijgevolg meerdere aanvragen gelijktijdig verwerken. Daarenboven zijn load balancers verantwoordelijk voor het verdelen van de binnenkomende aanvragen over de onderliggende (applicatie)servers. Dit hoofdstuk focust zich enerzijds op een korte introductie van load balancers. Anderzijds focust dit hoofdstuk op het maken van een doordachte keuze van een load balancer of reverse proxyserver die geschikt is om HTTP-trafiek van eindgebruikers optimaal te verdelen over webservers.

2.4.1 Wat zijn load balancers?

Een load balancer is verantwoordelijk voor het doorverwijzen en verdelen van trafiek naar onderliggende applicatieservers. In deze masterproef betekent dit concreet dat aanvragen van eindgebruikers naar een server eerst op de load balancer toekomen. Vervolgens zal de load balancer afhankelijk van enkele parameters bepalen welke HTTP-server de aanvraag zal ontvangen. De load balancer staat als single point of contact beschikbaar naar de eindgebruikers toe. Dat wil zeggen dat eindgebruikers nooit rechtstreeks toegang krijgen tot een enkele HTTP-server.

Dit maakt het mogelijk om in een gecontaineriseerde omgeving achterliggend extra containers op te starten om o.a. meer gelijktijdige aanvragen te kunnen behandelen. De schaalbaarheid, flexibiliteit en efficiëntie van de applicatie in het algemeen zal hierdoor stijgen. De redundantie wordt pas opgetrokken als ook de load balancers in hun meervoud worden gedeployd.

Load balancers hebben in het algemeen twee manieren van werken, namelijk op laag 4 of laag 7 van het OSI-model. Laag 4 is de transport laag die TCP- en UDP-connecties afhandelt. In deze modus kan een load balancer alle soorten binnenkomende TCP-trafiek gaan verdelen naar onderliggende servers. Hierbij worden pakketten niet – of enkel de eerste frames van een TCP-stream – geïnspecteerd voordat

de load balancer een beslissing maakt. Daartegenover is er ook de laag 7 modus van een load balancer. Deze modus is gericht op de applicatie laag in het OSI-model, en kan als gevolg meer diepgaandere pakketinspecties uitvoeren. De load balancer kan a.d.h.v. de volledige inhoud van het pakket een 'slimme' beslissing maken over de verdere weg van het inkomend HTTP-bericht [52], [53]. Laag 7 load balancers worden in deze masterproef gebruikt voor het afhandelen van binnenkomende HTTP-trafiek.

De uiteindelijk beslissing die een load balancer maakt om een pakket uit te sturen naar een bepaalde server is afhankelijk van de ingestelde eigenschappen in combinatie met het geselecteerde algoritme. Het algoritme kiest welke (web)server de inkomende aanvraag zal ontvangen. De volgende lijst geeft enkele veelvoorkomende load balancing algoritmes of methodes weer met een verklarende uitleg.

- **Round robin** is een algoritme dat sequentieel elke onderliggende server een binnenkomende aanvraag toestuurt. Standaard krijgt elke server evenveel aanvragen te verwerken. Dit is een statisch algoritme dat geen rekening houdt met de belasting van de achterliggende servers.
- De server met het minst **aantal actieve connecties** zal de eerstvolgende aanvraag binnenkrijgen. Dit is een dynamische methode die afhankelijk van de belasting een andere server kiest.
- **IP-gebaseerde** connecties maken gebruik van de hash van het source IP-adres om te bepalen welke server de binnenkomende aanvraag verwerkt. Dit is een statisch algoritme dat het mogelijk maakt om vanuit het algoritme eenzelfde gebruiker telkens naar dezelfde server te sturen.
- De server met de **minste verwerkingstijd** zal de eerstvolgende aanvraag behandelen. Dit is een dynamisch algoritme dat o.b.v. de snelste responstijd zal beslissen welke aanvraag naar welke achterliggende server wordt gestuurd.

In sommige gevallen bestaan er ook mogelijkheden om een wegingsfactor aan servers toe te kennen. Dit maakt het mogelijk om het algoritme of de methode lichtelijk te beïnvloeden o.b.v. een vooraf toegekende parameter. Als voorbeeld kan een server met meer computer resources een hogere wegingsfactor krijgen dan een server met minder resources.

2.4.2 Keuze

Nokia heeft verschillende vereisten vastgelegd waaraan de logserver moet voldoen. De logserver bestaat uit drie delen, namelijk de HTTP-server, PHP-module en de load balancer. Deze vereisten zijn opgesteld als algemene doelstellingen voor de logserver. Aangezien de load balancer het eerste contactpunt is vanuit de gebruikers, moet de load balancer minimaal 100 aanvragen per seconde kunnen afhandelen. Er is dus vraag naar een zo performant mogelijk systeem dat mee kan schalen naar de toekomst. Daarnaast is een opensource en freewareproduct van cruciaal belang om de totale kosten van de omgeving te drukken. Tot slot moet er een container image beschikbaar zijn om de containerisatie in de Kubernetes-cluster zo vlot mogelijk te laten verlopen.

Vier laag 7 (L7) freeware load balancers blijken in de literatuur de meeste aandacht te krijgen. De volgende opsomming bespreekt kort deze load balancers voordat ze in dit hoofdstuk verder worden vergeleken:

- **Nginx** kan niet alleen als een webserver maar ook als load balancer worden geconfigureerd. De functionaliteit van load balancing wordt afgehandeld door dezelfde service als eerder besproken in 2.4. Naast de opensource beschikbare features, biedt Nginx ook zogenaamde 'Plus features' dewelke betalende eigenschappen zijn. Deze extra features hebben voornamelijk betrekking tot monitoring en extra configuratiemogelijkheden [54].
- **HAProxy** is een tweede mogelijke L7 load balancer. HAProxy noemt zichzelf een zeer snelle en betrouwbare load balance oplossing speciaal voor websites met veel trafiek. Op de referentielijst van organisaties die HAProxy gebruiken staan o.a. Airbnb, GitHub, Reddit, …

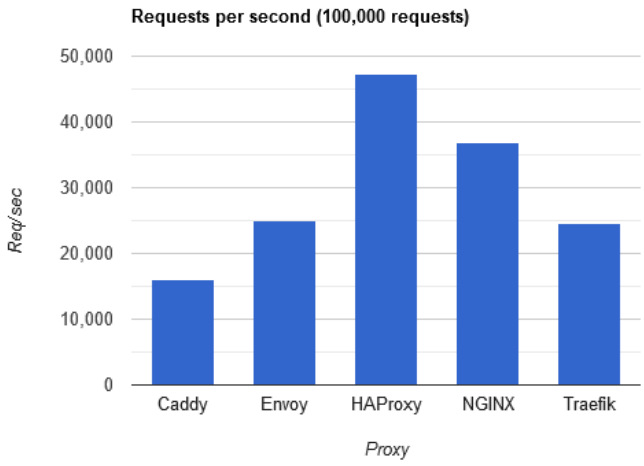
Naast de gratis community versie bestaat er ook een betalende enterprise versie die o.a. professionele support bevat inclusief meer geavanceerde configuratiemogelijkheden [55].

- **Traefik** is een derde mogelijke opensourceoplossing die als load balancer kan fungeren. In tegenstelling tot Nginx en HAProxy die al jaren ervaring hebben, is Traefik een relatief nieuwe speler op de markt sinds 2015. Traefik focust zich voornamelijk op simpele configuraties voor dynamische microservices. Ook Traefik heeft een betalende versie die meer features alsook professionele support aanbiedt [56].
- **Envoy** is de laatst besproken load balancer doorheen dit hoofdstuk. Envoy is net zoals Traefik een nieuwere speler op de markt die zich focust op microservices. Naast load balance functionaliteiten biedt Envoy ook een heleboel mogelijkheden i.v.m. proxyservices. In tegenstelling tot voorgaande alternatieven is Envoy volledig freeware zonder betalende features [57].

Performantie

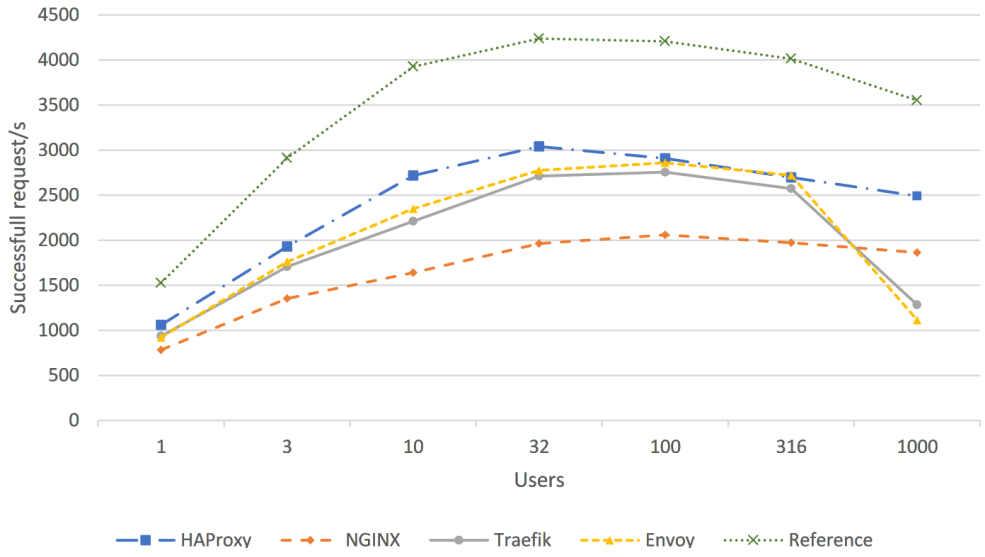
De performantie moet als eerste parameter optimaal zijn om aanvragen van eindgebruikers zonder merkbare vertragingen te behandelen. Daarnaast heeft Nokia een ondergrens vastgelegd voor het minimumaantal aanvragen die per seconde moeten behandeld worden. Deze ondergrens ligt op 100 aanvragen per seconde. De volgende paragrafen geven testresultaten van load balancers weer om de performantie te kunnen vergelijken.

[58] toont een benchmark waarin de vier genoemde load balancers voorkomen. Hier wordt aangetoond dat HAProxy bij het HTTP-protocol de minste vertraging (=latency) heeft. Echter is bij de secure HTTPS-variant de vertraging in het algemeen hoger, waarbij Envoy noemenswaardig dicht in de buurt komt van HAProxy. Verder toont [58] dat Envoy vijfmaal meer aanvragen per seconde kan verwerken t.o.v. de daaropvolgende load balancers HAProxy en Nginx. Het dient opgemerkt te worden dat [58] dateert van 2018 en dat er reeds een update is uitgebracht die hierop is gebaseerd. Zo toont [59] – enkel d.m.v. het HTTP-protocol – dat HAProxy de minste vertraging heeft. Echter bij het 99ste percentiel is Envoy nipt sneller dan HAProxy. Bijkomend toont dezelfde bron dat HAProxy bijna 50000 aanvragen per seconde kan verwerken, waarbij Envoy er maar net 25000 haalt zoals zichtbaar in Figuur 11. Traefik komt er in dit geval uit als slechtste van de vier. Overigens toont [60] dat Traefik op de hoogte is dat er minder aanvragen per seconde kunnen afgehandeld worden t.o.v. Nginx. Dit wordt gerechtvaardigd doordat het Traefik-project nog in zijn kinderschoenen staat in vergelijking met HAProxy en Nginx.



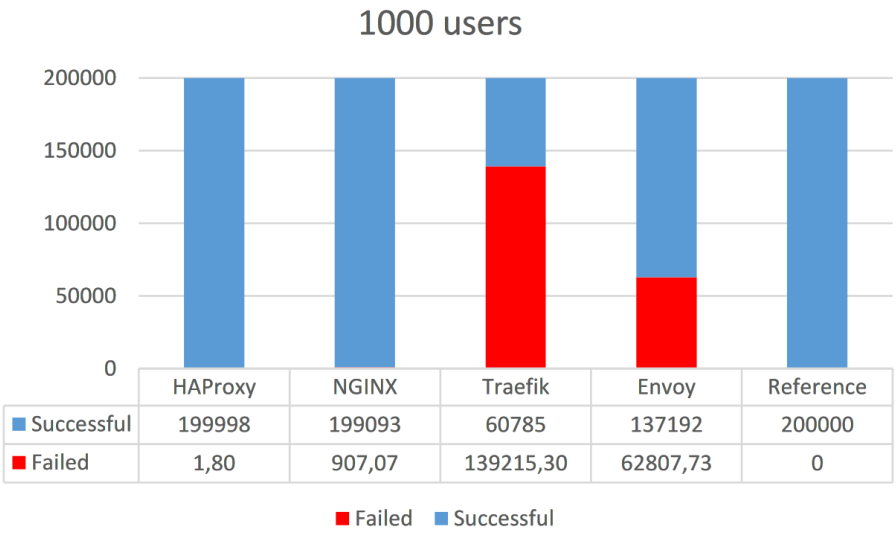
Figuur 11: Staafdiagram met het aantal aanvragen per seconde per load balancer [59]

Bovendien toont [61] enkele resultaten over de vier load balancers die in dit hoofdstuk worden toegelicht. Het dient opgemerkt te worden dat alle testresultaten in deze bron zijn bekomen d.m.v. het HTTPS-protocol. Figuur 12 toont een scenario waarin 30 keer op rij het aantal succesvolle aanvragen per seconde wordt gemeten, indien 200000 aanvragen verdeeld over x aantal users worden verstuurd. Van deze 30 steekproeven is een gemiddelde genomen en vervolgens geplot in Figuur 12. Hieruit kan worden geconcludeerd dat HAProxy de meeste aanvragen per seconde kan verwerken ongeacht het aantal users. Desalniettemin zijn er minder aanvragen die per seconde kunnen worden verwerkt in vergelijking met een webserver zonder load balancer. Nginx komt er in alle gevallen als slechtste uit, waarbij er een verschil is van 34% t.o.v. HAProxy bij 1000 users.



Figuur 12: Lijndiagram met het aantal succesvolle aanvragen per seconde, verwerkt door een load balancer afhankelijk van het aantal gelijktijdige aanvragen(=users) [61, p. 20]

Figuur 13 toont als aanvulling op Figuur 12 een weergave van het aantal aanvragen die al dan niet succesvol zijn beantwoord door de load balancers bij 1000 users. Uit het staafdiagram blijkt dat HAProxy veruit de meest betrouwbare load balancer is, doordat amper 0,0009% van het aantal aanvragen mislukt. Bij Traefik faalt bijna 70% van de aanvragen terwijl 31% van het aantal aanvragen bij Envoy falen [61].



Figuur 13: Staafdiagram met het gemiddelde aantal aanvragen die al dan niet succesvol zijn beantwoord per load balancer [61, p. 21]

Op het vlak van performantie blijkt HAProxy de beste keuze. Traefik en Envoy zijn eerder startersprojecten die gemiddelde resultaten behalen voor applicaties op kleinere schaal. Op grotere schaal met meerdere gelijktijdige aanvragen zullen Traefik alsook Envoy minder goede resultaten behalen.

Containerisatie-mogelijkheden

Op vlak van containerisatie-mogelijkheden is de eerste en meest voor de hand liggende bron Docker Hub [62]. Zowel Nginx, HAProxy alsook Traefik zijn gepubliceerd op Docker Hub als officiële images. Envoy heeft geen officiële Docker image maar is wel beschikbaar gemaakt door verscheidene geverifieerde gebruikers.

Daarnaast zijn er voor HAProxy, Traefik alsook Nginx zogenaamde Helm charts beschikbaar gemaakt door geverifieerde gebruikers. Voor Envoy zijn er ook Helm charts beschikbaar gemaakt, maar niet door geverifieerde gebruikers.

Qua containerisatie-mogelijkheden blijken er in het algemeen geen restricties, en zou elke load balancer makkelijk te containeriseren zijn. Er dient enkel opgemerkt te worden dat er voor Envoy geen Helm chart beschikbaar is die is geverifieerd door de Helm community.

Conclusie

Uit de literatuurstudie over load balancers blijkt dat HAProxy de meest performante load balancer is i.v.m. het aantal aanvragen dat per seconde kan worden verwerkt. Bovendien is HAProxy in staat om in bijna elk geval de minste vertraging te veroorzaken en is er zowel een officiële Docker image alsook geverifieerde Helm chart beschikbaar. HAProxy lijkt daarom een load balancer die geschikt is voor snelgroeiende applicaties die veel aanvragen gelijktijdig moeten verwerken zoals in deze masterproef het geval.

2.5 Storage oplossingen

Naast CPU- en geheugenperformantie in een cluster van servers, is opslag ook een belangrijk item dat mee voor de performantie zorgt. Dit hoofdstuk focust zich op het vergelijken van gedistribueerde (of distributed) filesystems. Binnen deze masterproef is dit een puur theoretische benadering waarbinnen geen specifieke beste keuze naar voor wordt gebracht. Dit hoofdstuk focust op de vergelijking van gedistribueerde filesystems waarmee Nokia in de toekomst verder aan de slag kan na afloop van deze masterproef.

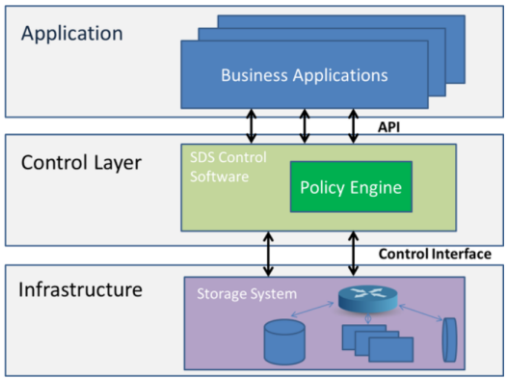
2.5.1 Wat zijn storage oplossingen?

Storage zorgt in het algemeen voor een persistente opslag van data. In deze masterproef vertegenwoordigen logfiles de bovengenoemde data. Hoewel in een PC ook een bepaalde hoeveelheid storage zit, wordt dit begrip op het niveau van een datacenter enigszins abstracter. Bijkomend heeft een datacenter ook andere voorwaarden op vlak van beschikbaarheid, schaalbaarheid, snelheid en budgetten. Zo is het bijvoorbeeld moeilijk om bij de opstart van een bepaald project dadelijk genoeg redundante storage te voorzien voor de hele levensduur van het project. De return on investment (ROI) van de storage bij de start zou veel te laag liggen. Er is met andere woorden behoefte aan flexibiliteit om storage te laten groeien.

Een gedistribueerd filesystem zorgt voor een softwarematige abstractielaag tussen de storage en de services die deze storage raadplegen. De software dient als een samenhangend geheel alsook centrale beheerplaats van alle onderliggende storageservers. Dit maakt het mogelijk om virtueel opslag aan te

bieden over het netwerk afhankelijk van de vereisten. Daarnaast laat het centrale beheer toe om opslag dynamisch toe te voegen aan een virtuele opslaglocatie, waardoor het aanbod mee kan groeien afhankelijk van de benodigdheden van eindgebruikers. Omdat storage wordt geabstraheerd zijn er als gevolg ook mogelijkheden om authenticatie alsook redundantie mee in te bouwen of pas achteraf toe te voegen. Afhankelijk van budgetten kunnen hiervoor al dan niet freewareoplossingen gekozen worden, waarbij commerciële producten doorgaans support en extra features bevatten.

Figuur 14 toont een simplistische weergave van een gedistribueerd filesystem. Hierbij zijn onderaan de harde schijven of infrastructuur die al dan niet rechtstreeks gekoppeld zijn aan de control layer zichtbaar. De control layer is de abstractielaag die dient als API tussen de effectieve infrastructuur en de applicatie of service.



Figuur 14: Simplistische architectuur van een gedistribueerd filesystem [63, p. 2]

In het algemeen biedt een gedistribueerd filesystem meer vrijheid aan de storage administrator om bepaalde zaken achteraf pas toe te voegen, zonder een volledige herinstallatie.

2.5.2 Keuze

Gedistribueerde filesystems zijn er in alle vormen en maten. Echter zijn er enkele voorwaarden die moeten voldoen om binnen deze masterproef een nuttige bijdrage te kunnen leveren. Allereerst wordt er verwacht dat de totale kosten van de oplossing zo laag mogelijk zijn. Deze kosten richten zich enerzijds op de prijs van de oplossing zelf, anderzijds op de prijs van de opslagmedia die nodig zijn om de oplossing te implementeren. Daardoor wordt er door Nokia voorkeur gegeven aan een opensource gedistribueerd filesystem dat een breed aanbod aan makkelijk te verkrijgen hardware ondersteunt, ook wel commodity hardware genoemd. Vervolgens wordt er in dit hoofdstuk gefocust op de algemene bestaansreden van het gedistribueerd filesystem en wat deze oplossing uniek maakt. Daarna is het belangrijk dat de performantie naar behoren is, waarbij voornamelijk de leessnelheid van bestanden doorslaggevend is. De leessnelheid is essentieel doordat verschillende gebruikers gelijktijdig met zo weinig mogelijk vertraging logfiles moeten kunnen raadplegen. Aansluitend bij de performantie is ook de marktpositie in mindere mate van belang.

Op basis van een literatuurstudie is er gekozen voor vier verschillende freeware opensourceoplossingen die het meest voorkomend zijn. Daarenboven is er ook nog één minder populair gedistribueerd filesystem toegevoegd aan deze lijst, die in de toekomst mogelijks gaat domineren. De volgende opsomming bespreekt kort de vier gedistribueerde filesystems die doorheen dit hoofdstuk worden vergeleken, alsmede de minder populaire oplossing:

- **HDFS** staat voor Hadoop Distributed File System en is een product van de Apache Foundation. De organisatie beschrijft het product als een uiterst fouttolerante software die kan draaien op goedkopere hardware die ruimschoots verkrijgbaar is [64]. HDFS maakt gebruik van een gecentraliseerde architectuur waarbij één master node verschillende slave nodes aanspreekt [64], [65].
- **Ceph** is een distributed file system dat alle soorten data storage kan behandelen, gaande van block storage en object storage tot file system storage, dit onder één groot softwarepakket dat een volledig gedistribueerde architectuur heeft met de mogelijkheid om te draaien op commodity hardware [65], [66].
- **GlusterFS** is een vrij te gebruiken distributed file system dat in tegenstelling tot HDFS of Ceph, een gedecentraliseerde architectuur hanteert. Een gedecentraliseerde architectuur slaat op meerdere master nodes die verschillende slave nodes coördineren, dewelke meestal fysiek op een andere locatie staan. Ook dit gedistribueerd filesystem ondersteunt commodity hardware [65], [67].
- **Swift** is een uitbreiding van de OpenStack community en biedt een oplossing voor het opslaan en verkrijgen van veel data zoals ze zelf op hun website vermelden [68]. Swift is dan wel een uitbreiding op OpenStack, maar kan volledig autonoom draaien zonder dat er een OpenStack cluster aanwezig is. Daarenboven maakt Swift gebruik van een gedistribueerde architectuur die toelaat om commodity hardware te gebruiken [69].
- **Seaweedfs** is een veelbelovend maar minder populair filesystem [70]. [71] toont aan dat Seaweedfs zowel Ceph als Swift blijkt te overtreffen op alle uitgevoerde experimenten. Echter is Seaweedfs nog maar sinds 2020 operationeel en is er nagenoeg geen wetenschappelijke informatie over terug te vinden. Daarom wordt dit gedistribueerd file system buiten de scope van deze masterproef gehouden. Naar de toekomst toe dient er rekening gehouden te worden met deze oplossing doordat de performantie veelbelovend is gezien de jeugdigheid van het project.

Elk van de vier relevante gedistribueerde filesystems heeft een eigen architectuur wat telkens voordelen alsook nadelen met zich meebrengt. Zo heeft enerzijds een gecentraliseerde architectuur als nadeel dat er een single point of failure ontstaat op de master node wat ook de schaalbaarheid bemoeilijkt. Echter als voordeel is de configuratie eenvoudiger. Anderzijds kunnen gedistribueerde systemen extra complexiteit met zich meebrengen, wat doorgaans resulteert in een betere beschikbaarheid en schaalbaarheid [65], [72].

Algemeen

Globaal bekeken is HDFS het meest gebruiksvriendelijke gedistribueerd file system voor doeleinden die een ongecompliceerde oplossing vereisen met een goede performantie, goede fouttolerantie en met weinig overhead. Tevens zal HDFS het snelst kunnen herstellen van een uitgevallen node in een cluster [65].

Ceph daarentegen is complexere software met heel wat configuratiewerk. Qua performantie scoort Ceph in het algemeen net iets minder dan HDFS. Echter is Ceph het paradepaardje als het gaat over de mogelijke features en de toepasbaarheid. Dit maakt Ceph de ideale oplossing voor complexe en veeleisende omgevingen [65], [73].

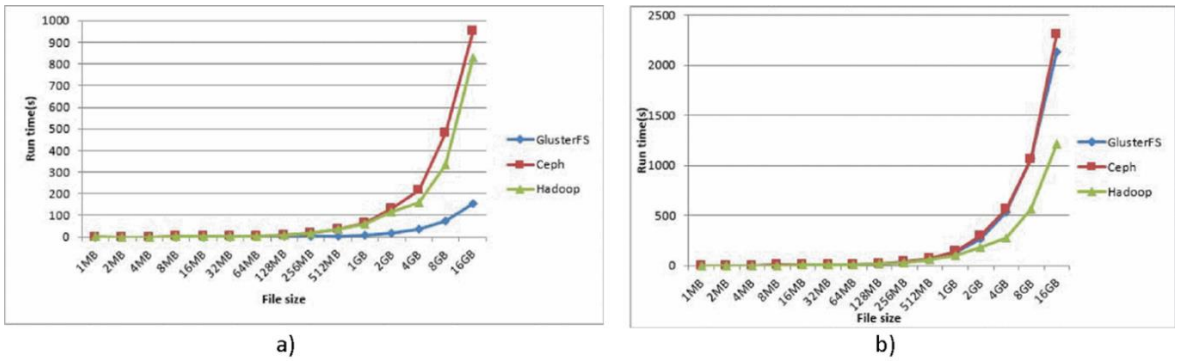
GlusterFS is een gedistribueerd filesystem dat meer focust op de integratie in applicaties, en minder gedetailleerd op de effectieve hardware waarvoor het verantwoordelijk is [73]. Volgens [65] slaagde GlusterFS er niet in om een falende machine in een testomgeving terug te herstellen, met als gevolg dat de fouttolerantie een slechte beoordeling krijgt. Nochtans tonen [65], [74] en [75] dat de performantie

over het algemeen geen grote afwijkingen bevat in de negatieve zin. GlusterFS behaalt in enkele zeer specifieke omstandigheden – met een beperkt aantal cores en threads – zelfs de beste performantie [75].

Tot slot is Swift het minst beschreven gedistribueerd file system in de literatuur. Naast de weinige wetenschappelijk resultaten die terug te vinden zijn, toont [71] aan dat Swift bij twee op de drie experimentele resultaten een slechtere performantie heeft dan Ceph. Door de slechtere performantie in het algemeen alsook de weinige wetenschappelijke resultaten, zal ook Swift niet verder vergeleken worden in de volgende hoofdstukken.

Performantie

Indien er specifieker ingegaan wordt op de performantie van de drie meest mature opensource gedistribueerde filesystems, zijn er enkele opmerkelijke bevindingen. Figuur 15a toont de tijd die nodig is om een bestand van bepaalde omvang te lezen vanuit het gedistribueerd file system. Figuur 15b toont de tijd die nodig is om een bestand te schrijven naar het gedistribueerd file system afhankelijk van de grootte van het bestand. Als conclusie blijkt dat bij zowel het lezen als het schrijven van een bestand tot 512 MB, de performantie van zowel GlusterFS, Ceph en HDFS gelijkaardig blijft. Echter bij een stijgende omvang van bestandsgrootte, zal de leessnelheid van GlusterFS significant beter zijn, terwijl de schrijfsnelheid van HDFS beter is. Bovendien hebben Ceph en GlusterFS gelijkaardige schrijfsnelheden, maar qua leessnelheid is Ceph net iets trager voor bestanden boven de 512 MB [74]. [65] toont evenzeer aan dat GlusterFS betere schrijfsnelheden behaalt dan Ceph, maar dat HDFS nog betere snelheden behaalt dan GlusterFS voor kleinere bestanden.



Figuur 15: a) De grootte van een bestand in functie van de verstreken tijd totdat het bestand is gelezen vanuit het gedistribueerd filesystem
b) De grootte van een bestand in functie van de verstreken tijd totdat het bestand is geschreven naar het gedistribueerd filesystem [74, p. 105].

Tabel 2 daarentegen toont tegenstrijdige resultaten ten opzichte van Figuur 15. Op vlak van de leessnelheid van grote bestanden komt Ceph als beste oplossing naar voor. Terwijl GlusterFS de snelste blijkt te zijn indien het gaat over het schrijven van grote bestanden. Daarentegen is HDFS de snelste oplossing voor het lezen en GlusterFS voor het schrijven van meerdere kleine bestanden [72]. HDFS komt er in dit geval uit als beste oplossing voor kleine bestanden.

Tabel 2: Performantie van gedistribueerde filesystems uitgedrukt in het aantal verstreken seconden voor het schrijven en lezen van een of meerdere bestanden [72, p. 16].

	HDFS		Ceph		GlusterFS	
Schrijven/ Lezen	Schrijven	Lezen	Schrijven	Lezen	Schrijven	Lezen
1 X 20 GB	407 s	401 s	419 s	382 s	341 s	403 s
1000 X 1 MB	72 s	17 s	76 s	21 s	59 s	18 s

Marktpositie

Naast de performantie van een product, is de positie op de markt van dat product ook belangrijk om een doordachte keuze te maken. A.d.h.v. de marktpositie wordt gepeild hoe een bepaald product erin slaagt om hun visie ten opzichte van de concurrentie waar te maken naar de klant toe. Hiervoor is het Gartner Magic Quadrant een doeltreffend middel. Figuur 16 toont het Gartner Magic Quadrant voor gedistribueerde filesystems. In dit kwadrant staan voornamelijk merken van producten die noch freeware noch opensource zijn. Echter is er één organisatie in dit kwadrant, namelijk Red Hat die wel opensource freewareoplossingen aanbiedt. Red Hat heeft zowel GlusterFS alsook Ceph onder zijn bewind, met als gevolg dat beide technologieën invloed hebben op de plaats van de organisatie in het kwadrant. Red Hat staat hier in het vak van de visionairs met een middelmatige volledigheid van visie (x-as), en een eerder bescheiden mogelijkheid tot het effectief kunnen waarmaken van deze visie (y-as). Gartner beschrijft dit als een speler die de markt begrijpt en op kan anticiperen, maar nog niet volledig kan waarmaken naar de klant toe. Alsmede is Red Hat de enige organisatie die binnen het Gartner Magic Quadrant is opgenomen met betrekking tot de onderzochte gedistribueerde filesystems in dit hoofdstuk [76]–[78].

Figuur 16: Gartner Magic Quadrant voor gedistribueerde filesystems [77]

Globaal kan worden besloten dat betalende software een hogere positie op de markt aanneemt in tegenstelling tot freeware. Zo is bv. zichtbaar dat Dell een aanzienlijk hogere mogelijkheid heeft om hun visie naar de klant toe waar te maken t.o.v. Red Hat. Dell is daarmee ook de marktleider als het gaat over gedistribueerde filesystems, maar wordt buiten de scope van dit onderzoek gehouden door de kosten die aan deze services verbonden zijn.

Conclusie

Voor de keuze van een betrouwbaar opensource freeware gedistribueerd file system kan worden besloten dat HDFS, Ceph, GlusterFS en Swift commodity hardware ondersteunen. Daarnaast zijn ze alle vier opensource alsook vrij te gebruiken. Als eindoordeel is HDFS een uitstekende oplossing voor algemene omgevingen waar snelheid van kleine bestanden primeert, maar schaalbaarheid eerder beperkt is door de gecentraliseerde architectuur. Ceph daarentegen is een gepaste oplossing indien de betrouwbaarheid en functionaliteiten van groot belang zijn. Echter moet de snelheid van Ceph hier wel op inboeten bij kleine bestanden. GlusterFS neemt van beide gedistribueerde filesystems enkele

voordelen mee. Zo heeft GlusterFS voor kleinere bestanden een betere performantie dan Ceph, alsook zou de betrouwbaarheid van nature hoger moeten liggen dan HDFS door zijn gedecentraliseerde architectuur. Daarenboven zijn zowel Ceph als GlusterFS deel van de Red Hat organisatie wat het mogelijk maakt om professionele en betalende support te krijgen. In tegenstelling tot Red Hat moeten HDFS-gebruikers voornamelijk steunen op de opensource community en documentatie. Tot slot kan Swift een goede oplossing zijn indien OpenStack al reeds gebruikt wordt in een organisatie en waarbij een feilloze integratie primeert. Op vlak van performantie en functionaliteiten zijn HDFS, Ceph en GlusterFS een betere oplossing.

3 Dimensioneren van logfiles

Door het overzetten van testomgevingen naar een modernere Kubernetes-cluster, dient er een beeld gevormd te worden van de vereiste resources die moeten voorzien worden bij deze overgang. Deze masterproef focust in dit deel op het verkrijgen van statistische gegevens i.v.m. de omvang van logfiles – uitgedrukt in de grootte en het aantal files per testrun per categorie testbedden – op de huidige implementatie. Dit kan in latere stadia bijdragen tot het voorzien van de nodige opslagcapaciteit voor (centraal) opgeslagen logfiles, of te voorziene netwerkcapaciteiten voor het transfereren van deze logfiles. Als doelstelling moesten er minimaal twee aparte testbedden per categorie onderzocht worden.

Dit hoofdstuk geeft de werkingsmethode weer van het onderzoek naar waar en hoe logfiles zijn opgeslagen op een testbed, alsook hoe deze logfiles raadpleegbaar worden gemaakt. Daarnaast wijdt dit hoofdstuk verder uit over de implementatie voor het automatiseren van het verkrijgen van de omvang van logfiles en het bespreken van de resultaten.

3.1 Huidige structuur van logfiles

Alvorens de automatisatie naar het dimensioneren van logfiles kan worden uitgevoerd, dient er een beeld gevormd te worden van de structuur die achterliggend gebruikt wordt om logfiles te bewaren.

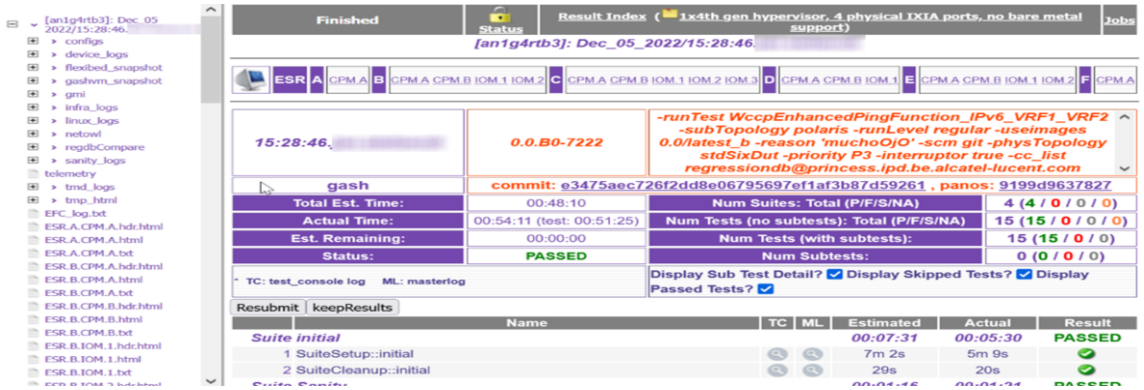
Elk testbed maakt gebruik van een gash VM om toegang te verlenen tot een omgeving. Hierop kan een gebruiker a.d.h.v. een centraal LDAP-authenticatiesysteem inloggen. In tegenstelling tot een gewone Linux-machine, heeft in dit geval elke gebruiker dezelfde home map waarin de gegevens staan van het testbed waarop de gebruiker inlogt. Het grootste deel van de mappenstructuur is vrijwel identiek voor elk testbed met uiteraard andere gegevens. De logfiles van elk apparaat in de omgeving worden opgevangen in de ~/logs/ map, die aan de hand van een complex Perl-script tijdens het lopen van een testrun worden omgevormd tot consistente logfiles die zijn geaggregeerd per uitgevoerde test. Een gebruiker kan vervolgens per uitgevoerde testrun deze logs raadplegen in de vorm van een mappenstructuur die is opgebouwd volgens een datum- en tijdnotatie gevolgd door de gebruiker die de testrun startte. Figuur 17 toont de mappenstructuur in een meer visuele vorm, waarbij de laatste map de zogenaamde startmap van verwerkte testresultaten is, die meerdere logfiles en andere onderliggende mappen bevat.

~/results/<jaar>/<maand>/<dag>/<HH:MM:SS.testEngineer>/

Figuur 17: Mappenstructuur van testresultaten op een testbed

Elke startmap van een test bevat naast de variabele logfiles ook HTML-bestanden, waarvan één van deze bestanden als indexpagina of startpagina van die testrun dient. Figuur 18 toont een voorbeeld van een startpagina van een testrun. Merk hierbij op dat deze HTML-pagina is opgebouwd uit twee frames en dat bij het laden ongeveer 20 achterliggende HTTP-aanvragen worden uitgevoerd. Een frame binnen HTML zorgt voor het insluiten van een andere webpagina – meestal in de vorm van een URL – in de huidige pagina. Het eerste frame zorgt voor het genereren van een lijst van bestanden van die testrun in de vorm van een boomstructuur, zichtbaar aan de linkerkant van de figuur. Het tweede frame daarentegen toont de algemene informatie van de geopende testrun, inclusief de verwijzingen om naar de gegenereerde logfiles te navigeren. Beide frames verwijzen bij elke testrun naar dezelfde PHP-bestanden die buiten de startmap van deze testrun zijn opgeslagen, namelijk binnen de ~/status/ map. De PHP-bestanden voeren vervolgens enkele handelingen uit afhankelijk vanuit welk pad en welk testbed ze worden opgeroepen, om voornamelijk JavaScript dynamisch te genereren. De PHP-bestanden zijn op hun beurt afhankelijk van het /usr/global/bin/regress.params bestand, dat via NFS (Network File System) centraal op elk testbed beschikbaar wordt gesteld. Tot slot zorgt de gegenereerde JavaScript die

door de browser wordt geïnterpreteerd, voor het opbouwen van de pagina zoals zichtbaar in Figuur 18. Test engineers zijn volledig vertrouwd met de opbouw en locatie van elke logfile binnen een testrun. Daarom dient de structuur van de webpagina bij het Aanleveren van logfiles in de Kubernetes-cluster gelijkaardig te zijn.



Figuur 18: Startpagina van één specifieke reeds afgelopen testrun

De reeds vermelde paden en bestanden zijn cruciaal om een correcte vertoning van de startpagina van een testrun in een browser te kunnen doen. Daarenboven bleken er na uitvoerig onderzoek ook nog configuraties te bestaan binnen de webserver, die de URL's binnen de frames herinterpreteren en doorverwijzen naar andere files binnen de ~/status/ map. Vermoedelijk is dit een manier om een geüpdatete versie van de PHP-bestanden door te voeren, zonder dat mogelijkse oudere configuraties zouden breken.

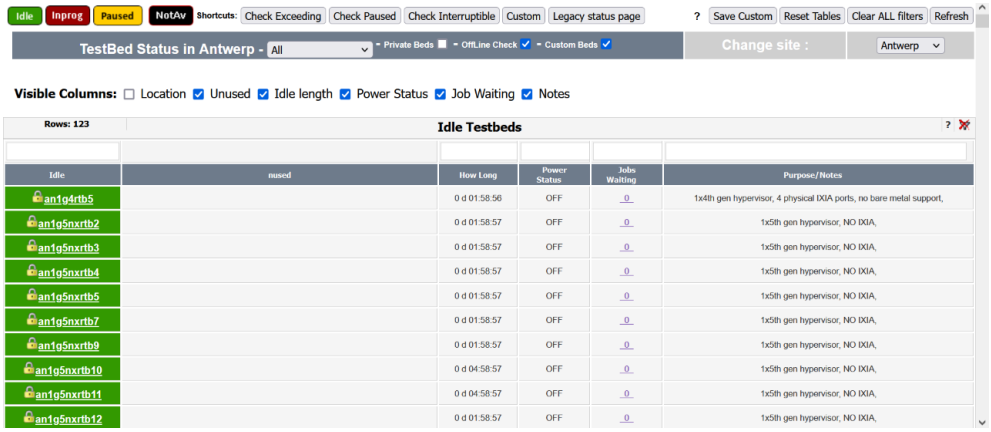
3.2 Automatiseren voor het dimensioneren van logfiles

Doordat het manueel dimensioneren van logfiles eentonig en tijdrovend is, werd doorheen deze masterproef een manier bedacht om bijna volledig automatisch zonder interactie met een gebruiker logfiles te dimensioneren. Dit hoofdstuk focust op het uitvoeren van het dimensioneren van de logfiles in de vorm van het ophalen en opslaan van statistische gegevens per testbed, alsook op het in bulk verwerken van de resultaten.

3.2.1 Verkrijgen van inactieve testbedden

Om te voorkomen dat lopende testruns beïnvloed worden door de automatisatiescripts die verder besproken worden in dit hoofdstuk, dienen enkel de IP-adressen van de testbedden die op dat moment geen testrun aan het uitvoeren zijn te verkregen worden. Het verkrijgen van deze IP-adressen diende op een zo snel en efficiënt mogelijke manier te gebeuren. De verkregen IP-adressen zouden dan vervolgens gebruikt kunnen worden door een ander script dat verantwoordelijk is voor het dimensioneren zelf. Dit deel focust op het script en het proces voor het verkrijgen van IP-adressen van inactieve testbedden.

Binnen Nokia wordt er gebruik gemaakt van een webpagina die de status van alle testbedden van een gespecifieerde geografische locatie weergeeft. Figuur 19 toont een voorbeeld van zo'n statuspagina van de site in Antwerpen. Een testbed is als gevolg inactief of beschikbaar indien er geen testrun op dat moment aan het lopen is. Figuur 19 toont in dit geval 123 inactieve testbedden. Indien er een testrun loopt, is een testbed actief en wordt die weergegeven als zijnde bezet. Tot slot kan een testbed ook in pauze staan, wat duidt op bv. een gebruiker die aan het uitzoeken is waarom zijn testrun op een bepaald punt heeft gefaald. Ook de laatst vernoemde status duidt op het feit dat een testbed bezet is. Het is vanzelfsprekend cruciaal dat lopende testruns of testbedden die in pauze staan niet gestoord worden.



Figuur 19: Statuspagina van alle testbedden binnen één geografische locatie

Doordat de webpagina up-to-date informatie bevat over de status van de testbedden, bleek dit een perfecte informatiebron voor het verkrijgen van de IP-adressen van de beschikbare inactieve testbedden. Het IP-adres staat immers in het title-attribuut in de eerste kolom van elk testbed van de HTML-code verwerkt. Daarnaast kon deze informatie geraadpleegd worden zonder beroep te moeten doen op een aanvraag voor meer permissies. In de context van webpagina's en client-gebaseerde scripttalen bleek JavaScript vervolgens een geschikte taal om snel en eenvoudig een lijst van IP-adressen van inactieve testbedden te verkrijgen.

Figuur 20 toont het script dat via de Web Developer Tools in een webbrowser via de Console kan worden geïnjecteerd in de webpagina. Als resultaat komt er een lijst met IP-adressen van inactieve testbedden tevoorschijn.

```
//Inject jquery scripts in order to have easy selectors
var script = document.createElement('script');script.src =
"https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js";document.getElementsByTagN
ame('head')[0].appendChild(script);
//Get each link of an idle testbed
var eachtb = $("#idleTable tbody tr td:nth-child(1) a.tb");
//Get each IP which is encapsulated in the title of the link
var ips = "";
eachtb.each(function() {
    var titleOfObject = $(this).attr("title");
    titleOfObject = titleOfObject.split(',');
    ips = ips.concat(titleOfObject[0], "\n");
});
console.log(ips)
```

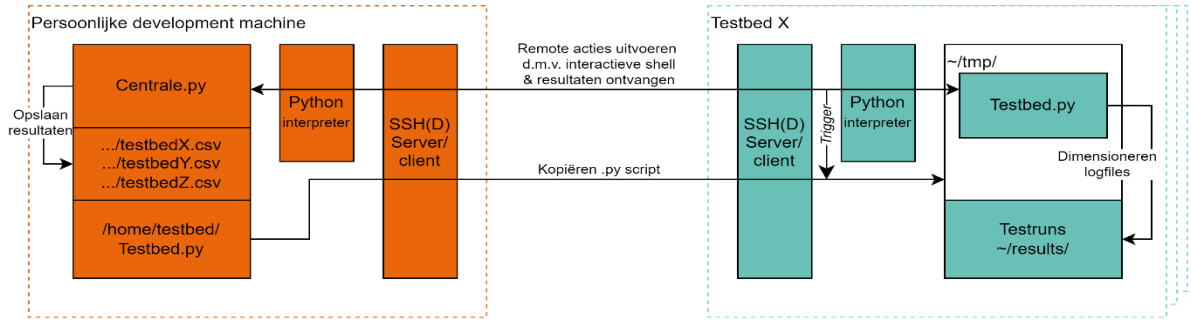
Figuur 20: JavaScript voor het verkrijgen van IP-adressen van inactieve testbedden

Het script van Figuur 20 toont alle IP-adressen in de console. In de huidige staat van het script moeten we deze nog manueel selecteren en er dan mee aan de slag gaan. In principe zou het mogelijk zijn om het script verder uit breiden, inclusief authenticatie, om zo een volautomatisch proces te bekomen. Doordat het script slechts enkele keren diende uitgevoerd te worden, zijn deze toevoegingen uiteindelijk niet geïmplementeerd geweest omdat de tijdsinvestering voor het automatiseren niet zou renderen.

3.2.2 Ophalen en opslaan van gegevens over logfiles

Voor het ophalen en opslaan van statistische gegevens van logfiles per testrun per testbed, zijn er twee aparte programma's uitgewerkt. Beide zijn geschreven in Python 2.7, enerzijds door de aanwezige voorkennis van deze programmeertaal, anderzijds door de aanwezigheid van deze (oudere) Python-omgeving die op elk testbed zonder roottoegang te gebruiken was.

Figuur 21 toont een simplistische weergave van de interacties tussen de machine waarop het startsignaal wordt gegeven en de testbedden die het dimensioneren van de resultaten uitvoert. Het linkse deel van de figuur toont de Linux development machine waaruit de coördinatie gebeurt. Het rechtse gedeelte stelt alle testbedden voor, die commando's uitvoeren op bevel van de centrale development machine. Deze opdeling is noodzakelijk vanwege de beperkte permissies die aanwezig waren op de testbedden enerzijds, en de volledige vrijheid op de development machine anderzijds.



Figuur 21: Blokdiagram van de werking van het programma voor het dimensioneren van logfiles

Starten van het proces a.d.h.v. het centrale Python-programma

Het centrale Python-programma staat allereerst in voor het leggen van een SSH-tunnel tussen de centrale development server en de IP-adressen verkregen uit het script van 3.2.1. Hierbij is het van cruciaal belang om een interactieve BASH (Bourne Again SHell) shell op te starten, doordat de juiste permissies pas na het uitvoeren van het gekende Linux .bashrc loginscript worden toegekend aan de gebruiker op het testbed. Dit heeft als gevolg dat het centrale Python-programma enkel a.d.h.v. de stdout (standard output) van het testbed kan vergelijken of de output van een bepaald commando overeenkomt met een op voorhand vastgelegde output. Nadat een interactieve shell is opgestart, beveelt het centrale Python-script aan het testbed, om via SFTP (SSH File Transfer Protocol) het testbed specifiek Python-programma van de centrale development machine te downloaden en te starten.

Starten van het dimensioneren a.d.h.v. het testbed Python-programma

Dit is het punt waar het dimensioneren van de logfiles start door het testbed Python-programma. De cursor begint met zoeken in de eerste map zoals zichtbaar in Figuur 17, en zoekt steeds dieper met een maximum van vier niveaus waar de testruns zijn gesitueerd. Vanaf dat er een map met een bepaalde RegEx (Regular Expression) overeenkomstig met het laatste deel van Figuur 17 wordt gevonden, start een methode die de omvang en het aantal bestanden van die map (=een testrun) verwerkt. Dit proces blijft draaien totdat alle mappen die maximaal vier niveaus diep zitten zijn overlopen. Vervolgens worden alle resultaten die zijn gecombineerd doorheen het proces, geprint naar de stdout.

Verzamelen van de resultaten a.d.h.v. het centrale Python-programma

Door de interactieve shell krijgt de centrale server de volledige output in een string binnen, die door wat nabewerking kan worden omgevormd tot een Numpy array en een CSV-file voor elk onderzocht testbed waarin de volgende elementen zitten per rij:

- Het nummer van de testrun
- Hostname van het testbed
- Absolute pad naar de testrun
- Aantal files exclusief mappen binnen de testrun
- Grootte in bytes van de testrun

3.3 Resultaten

A.d.h.v. de Python-programma's in 3.2.2 werden 37 testbedden uit acht verschillende categorieën onderzocht. Om resultaten per testbed en per categorie te combineren werd gebruik gemaakt van een apart Python-programma dat o.b.v. gekende libraries uit de literatuur (Pandas, Numpy en Matplotlib) de gegenereerde CSV-bestanden uitleest en hier vervolgens statistische gegevens uit opmaakt. Dit hoofdstuk wijdt verder uit over de bespreking van de totale resultaten alsook van de resultaten per categorie testbedden. Bijlage A geeft de overige boxplots weer die niet binnen dit hoofdstuk zijn aangehaald. Tabel 3 geeft een kort overzicht van het aantal onderzochte testbedden per categorie. Dit aantal werd bepaald afhankelijk van de beschikbaarheid en de snelheid van verwerken van deze categorie testbedden over een bepaalde tijdspanne.

Tabel 3: Aantal onderzochte testbedden per categorie

ang3nxrtb	ang4rtb	ang5nxrtb	ang6rtb	anhw	anrtb	anrtb2vm	anrtbvm
4	3	3	4	7	4	4	8

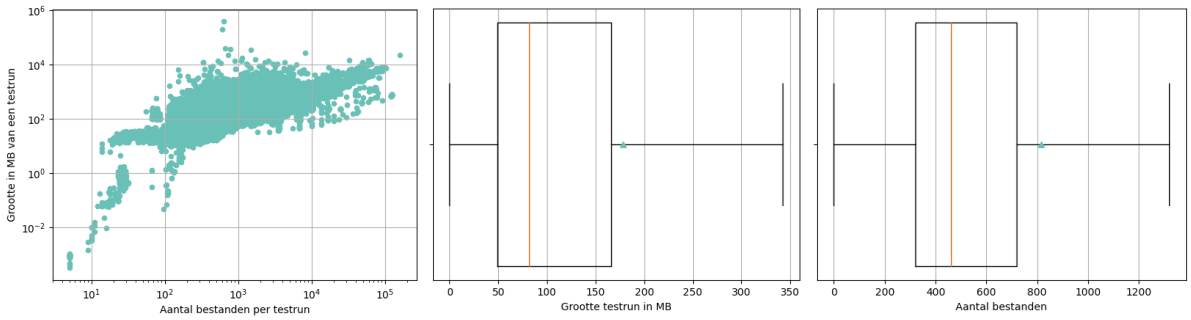
3.3.1 Totaal

Dit deel geeft de statistische gegevens weer van de totale steekproef van 37 testbedden. Tabel 4 toont het aantal onderzochte testruns, het gemiddelde ($\bar{x}$), de standaardafwijking (σ), de minimumwaarde, het eerste kwartiel (Q1), de mediaan, het derde kwartiel (Q3) en de maximumwaarde van zowel het aantal bestanden per testrun alsook de grootte van een testrun in MB.

Tabel 4: Algemeen statistische gegevens over testruns van het totaal aantal onderzochte testbedden

	Aantal	$\bar{x}$	σ	Min	Q1	Mediaan	Q3	Max
Omvang testrun	136505	178,35 MB	1261,38 MB	0 MB	48,88 MB	82,00 MB	166,32 MB	388273,48 MB
Aantal bestanden	136505	816,47	3115,48	0	320	460	720	160997

A.d.h.v. de opgehaalde gegevens in combinatie met Tabel 4 kan vervolgens Figuur 22 gegenereerd worden. Hierin staat een scatterplot van alle gegevens inclusief de boxplots van het aantal bestanden en de omvang van de bestanden in een testrun. Er zijn in totaal 136505 testruns onderzocht. Hieruit blijkt dat er gemiddeld 816,47 bestanden met een gemiddelde van 178,35 MB per testrun aanwezig zijn. Echter is de mediaan een pak lager met 460 bestanden met een omvang van 82,00 MB per testrun, door de hoge standaardafwijking en de uitschieters die dicht bij het maximum liggen. Zo bestaat er enerzijds een testrun in de steekproef die meer dan 388 GB aan bestanden bevat en anderzijds ook testruns die helemaal geen bestanden bevatten. Dit minimum van nul bestanden zijn testruns die vlak na de start zijn geannuleerd. Ondanks de hoge maximum kan hieruit geconcludeerd worden dat 75% van de testruns maximaal 720 bestanden bevatten die in totaal maximum 166,32 MB groot zijn. Daarenboven is het 99^{ste} percentiel (niet zichtbaar in de tabellen) van de omvang van de testruns gelijk aan 1535,80 MB.



a) b) c)

Figuur 22: a) Scatterplot van de volledige steekproef van het aantal bestanden per testrun in functie van de grootte van een testrun
b) Boxplot van de grootte per testrun van de volledige steekproef, excl. uitschieters
c) Boxplot van het aantal bestanden per testrun van de volledige steekproef, excl. Uitschieters

3.3.2 Per categorie

Naast de totale statistische gegevens van de 37 testbedden in zijn geheel, focust dit gedeelte zich op de statistische gegevens over de omvang en het aantal bestanden van de testbedden gegroepeerd per categorie, zoals zichtbaar in Tabel 5 en Tabel 6. Er zijn acht categorieën testbedden, waarbij elke categorie specifieke eigenschappen heeft waarop niet verder wordt gefocust doorheen deze masterproef.

Op basis van het aantal testruns dat is gevonden per categorie, is zichtbaar dat anrtbvm de meeste testruns bevat doordat in deze categorie de meeste testbedden zijn opgenomen zoals zichtbaar in Tabel 3. De anrtb2vm categorie daarentegen, heeft per testbed de meeste testruns doordat er 32267 testruns op slechts vier testbedden van deze categorie zijn gevonden. Desalniettemin heeft die categorie de op een na minste bestanden per testrun gelet op de mediaan, en de op een na laagste omvang per testrun gelet op het gemiddelde, de mediaan en de standaardafwijking. De categorie met veruit het minst aantal bestanden alsook de kleinste omvang per testrun voor zowel de mediaan als het gemiddelde is anhw. Slechts 25% van de testruns van deze categorie heeft meer dan 318 bestanden en is groter dan 97,40 MB. De categorieën ang4rtb, ang5nxrtb, anhw, anrtb en anrtb2vm hebben een mediaan van minder dan 100 MB per testrun en een gemiddelde van minder dan 200 MB per testrun. De categorie ang6rtb testbedden produceren de meest omvangrijke testruns doordat ze een gemiddelde van 693,67 MB en 6586,27 bestanden en een mediaan van 192,16 MB en 1607 bestanden per testrun hebben. De standaardafwijking en interkwartielafstand van zowel de omvang als het aantal bestanden is hier veruit de grootste – indien er geen rekening wordt gehouden met de standaardafwijking van de omvang van anrtbvm door de extreme maximum – terwijl de maximumwaarde slechts de op twee na grootste is. Ang6rtb testbedden hebben de meest omvangrijke alsook variërende testruns. Aan de hand van de maximumwaardes kan besloten worden dat anrtbvm de grootste testrun bevat van 388,273 GB, met slechts een maximumwaarde van 47969 bestanden per testrun. De maximumwaarde voor het aantal bestanden per testrun is 160997, dewelke is gevestigd in de ang5nxrtb categorie. Tot slot is de laagste waarde binnen de kolom van de maxima van de omvang van een testrun 4772,83 MB of ruim 4,5 GB. De percentielscore van deze exacte waarde binnen de totale gegevens is 99,89%. Dat wil zeggen dat 99,89% van alle verkregen testruns een omvang hebben die kleiner is dan 4772,83 MB.

Om in de toekomst opslag te voorzien voor logfiles, dient er rekening gehouden te worden met het aantal testruns men wil opslaan inclusief de periode voor hoelang men deze testruns wil bijhouden. 1 TB aan storage zou gemiddeld gezien logfiles van grofweg 5600 testruns kunnen opslaan door het algemeen gemiddelde van 178,35 MB per testrun. Daarenboven dient bij het toekennen van storage ook dieper ingegaan te worden op de omvang per categorie. Zo is de omvang van een testrun bij zeven van de acht categorieën gemiddeld onder de 300 MB, terwijl bij de ang6rtb categorie de gemiddelde omvang

opmerkelijk hoger is tot bijna 700 MB. Naast de gemiddeldes dienen tot slot ook de beperkte maar aanwezige uitschieters in rekening gebracht te worden voor het voorzien van opslag- en netwerkcapaciteiten. Reserveopslag voor dergelijke uitzonderingen zijn cruciaal om de persistentie van de ‘gewone’ testruns te garanderen.

Tabel 5: Algemeen statistische gegevens over de omvang van een testrun per categorie testbedden

Categorie	Aantal	$\bar{x}$ [MB]	σ [MB]	Min [MB]	Q1 [MB]	Mediaan [MB]	Q3 [MB]	Max [MB]
ang3nxrtb	13993	219,35	566,20	0,08	108,25	151,27	243,62	38980,19
ang4rtb	12767	183,06	513,24	0	51,55	83,88	155,29	17363,29
ang5nxrtb	7648	133,69	384,41	0	52,74	99,71	131,54	23099,02
ang6rtb	2544	693,67	1603,66	0,01	54,84	192,16	556,83	34557,16
anhw	17153	105,48	210,06	0,07	41,27	58,24	97,40	8234,71
anrtb	4295	184,14	356,29	0,19	49,20	71,96	147,24	4772,83
anrtb2vm	32267	120,27	260,77	0	42,97	62,31	104,78	7454,26
anrtbvm	45838	210,98	2074,23	0	54,94	93,11	222,15	388273,48

Tabel 6: Algemeen statistische gegevens over het aantal bestanden van een testrun per categorie testbedden

Categorie	Aantal	$\bar{x}$	σ	Min	Q1	Mediaan	Q3	Max
ang3nxrtb	13993	863,56	580,16	14	678	799	917	12583
ang4rtb	12767	822,29	2242,72	0	351	440	576	61619
ang5nxrtb	7648	565,35	2192,90	0	363	435	549	160997
ang6rtb	2544	6586,27	13124,77	11	456	1607	5090	101992
anhw	17153	260,86	142,71	14	155	239	318	2632
anrtb	4295	923,04	2051,06	23	317	442	694	20598
anrtb2vm	32267	683,28	4408,06	0	270	348	471	123887
anrtbvm	45838	813,83	987,40	0	385	644	898	47969

4 Aanleveren van logfiles

De Kubernetes-cluster heeft geen kennis meer van een persistent testbed op zich, enkel van een vluchtige en flexibele testrun. Het aanleveren van logfiles aan o.a. test engineers brengt als gevolg extra nieuwe uitdagingen met zich mee. Dit hoofdstuk focust op het proces dat doorlopen is om het aanleveren van logfiles te implementeren binnen de Kubernetes-testomgeving. Hierbij wordt eerst de gebruikte testopstelling benaderd, waarna het geïmplementeerde concept globaal wordt verduidelijkt. Vervolgens wordt de tweeledige implementatie – enerzijds per testrun en anderzijds als centrale implementatie – grondiger verklaard. De resultaten geven tot slot de chronologische weergave van het totaal bekomen proces met bijhorende figuren.

4.1 Testopstelling

Voor de aanvang van deze masterproef was al reeds een brede basis gelegd om de Kubernetes-testomgeving op te zetten alsook te beheren. Dit deel geeft de gebruikte methodes en technieken weer om de testopstelling die gebruikt werd algemeen toe te lichten. Dit vormt echter de basis waarop bepaalde zaken zijn opgebouwd zoals verder besproken in deze masterproef. Belangrijk om weten is dat tijdens de gehele periode er nog geen centrale cluster aanwezig was, enkel een geëmuleerde cluster die op een eigen development machine draaide.

4.1.1 Development machine

Binnen het datacenter werd een 32-core met 2,35 GHz klokfrequentie en 126 GB RAM Ubuntu server 20.04.5 development machine voorzien. Hierop is Docker als containertechnologie geïnstalleerd waarin vervolgens containers worden gecreëerd a.d.h.v. Kind. Kind is een tool die het mogelijk maakt om een volledige Kubernetes-cluster te emuleren op één machine a.d.h.v. Docker containers die zogenaamde nodes binnen een cluster voorstellen. Daarenboven kan een cluster gedefinieerd worden o.b.v. files, waardoor er telkens een reproduceerbare Kubernetes-omgeving wordt opgezet bij het initialiseren. Kind is voorzien om testomgevingen snel en reproduceerbaar op te zetten zoals in dit geval van toepassing [79]. Kind creëert in de testomgeving drie nodes, waarvan één node als master optreedt en de verantwoordelijkheden van het control plane op zich neemt en twee andere nodes die als workers fungeren. Deze rolverdelingen werden reeds toegelicht in het hoofdstuk rond Kubernetes. Om manueel te interageren met de cluster om bijvoorbeeld te debuggen of logs na te kijken, wordt gebruik gemaakt van Kubectl en k9s als cli tools geïnstalleerd op de host machine die rechtstreeks de API-server binnen Kubernetes aanspreken.

4.1.2 Netwerkconnectiviteit

Binnen de cluster worden applicaties bereikbaar gemaakt a.d.h.v. services zoals reeds vermeld in het hoofdstuk rond Kubernetes. Het voordeel is dat een service een blijvend object is binnen de cluster, dat onafhankelijk van de pods die al dan niet bestaan een vastgelegde DNS-naam heeft. Pods kunnen gekoppeld zijn aan één of meerdere services en zijn daarbij vrij om al dan niet te vermenigvuldigen of uit te vallen zonder het netwerkmatige te doen falen. Er zijn verschillende types services waarbij in de testopstelling er voornamelijk twee gebruikt worden. Eerst het type ClusterIp waarbij pods enkel bereikbaar zijn door andere pods die in de cluster aanwezig zijn. Daaropvolgend is er het NodePort type dat bepaalde TCP/UDP poorten aan een service vasthangt, waarna deze poorten op elke node beschikbaar worden. In de testopstelling is het mogelijk om via het IP-adres van één van de Docker containers die lid zijn van de Kind cluster, connectie te maken met een vastgelegde poort. De node zal op zijn beurt een binnenkomende aanvraag op die poort doorsturen naar de nodige service. In standaardomstandigheden laat Kubernetes enkel poorten toe tussen de 30 000 en 32 767.

Doordat de Development machine van het type Ubuntu server is en met als gevolg geen grafische interface bevat, is het toch wenselijk om af en toe bv. een grafische browser te kunnen gebruiken. Daarom wordt doorheen deze masterproef regelmatig gebruik gemaakt van SSH portforwarding. Hiermee kan een poort van de Development machine geforward worden tot op de client machine die met de SSH-server connectie maakt. Hierdoor kan bv. vanop een Windows-machine connectie gemaakt worden met een webserver die draait in de Kubernetes-cluster, d.m.v. de poort die ook op de huidige localhost beschikbaar gemaakt wordt.

Om connectie te maken tussen pods binnenin de cluster kan gebruik gemaakt worden van vastgelegde DNS-namen die bij de aanmaak van een service intern worden geregistreerd. Figuur 23 toont de opbouw van een DNS-naam van een service binnen de cluster. Indien het object dat de DNS-aanvraag uitvoert binnen dezelfde namespace ligt als het object waarnaar het op zoek is, dan hoeft de namespace in de DNS-naam niet vermeld te worden. Om consistentie en zekerheid te garanderen is het wel aangeraden om telkens de volledige naam te gebruiken.

<serviceNaam>.<namespace>.svc.<clusterNaam>.local

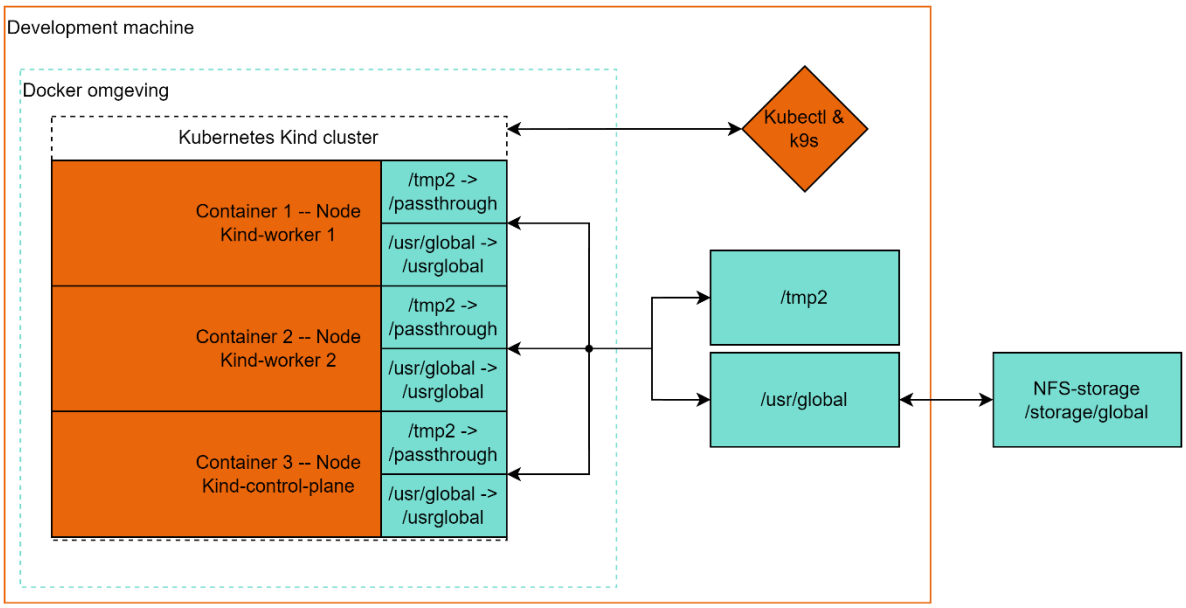
Figuur 23: Opbouw DNS-naam van een service binnen Kubernetes

4.1.3 Volumes & images

Zowel images voor containers die in pods draaien alsook allerlei andere bestanden dienen aangeleverd te worden aan de nodes die draaien in Kind. Dit hoofdstuk geeft kort de methodes weer over hoe en welke volumes voor de start van de masterproef reeds gebruikt werden enerzijds, anderzijds hoe images worden aangeleverd en opgeslagen tijdens development.

Volumes

Binnen de testopstelling zijn er enkele volumes die van buitenaf de cluster dienen binnengebracht te worden om een gewenste werking te garanderen. Allereerst is er de /tmp2 map die lokaal op elke development machine aanwezig is en die informatie bevat voor elke aanwezige node. Aangezien in de testfase alle nodes op eenzelfde machine draaien en er niet dadelijk een verschil hoeft te zijn, werd er één gezamenlijke /tmp2 map voorzien die binnen elke node wordt gemount. Dit heeft ook als gevolg dat data rechtstreeks vanuit elke pod – geconfigureerd met hostPath volumes in Kubernetes – kan worden geraadpleegd. De /tmp2 map bevat o.a. alle data die is gemaakt door een lokaal draaiende testrun. Verder wordt er ook gebruik gemaakt van een globaal storage medium dat via NFS is gemount in /usr/global op de host machine. Zoals reeds in het hoofdstuk Huidige structuur van logfiles vermeld, wordt o.a. het regress.params bestand hieruit gebruikt om de opbouw van een eerder bestaand testbed op te slaan. Ook dit volume is verder binnen de nodes gemount. Figuur 24 toont een schematische voorstelling van de opstelling van volumes die op de development machine werden geconfigureerd.



Figuur 24: Blokdiagram van de testopstelling op de development machine

Doordat volumes worden gedeeld over verschillende (gecontaineriseerde) instanties die allen toegang hebben tot de files, is er een vorm van afspraak nodig om files beschikbaar te maken met behulp van de correcte toegangspermissies. Daarom is er een algemene afspraak die zegt dat binnen elke pod (die naar een gedeeld volume schrijft of er files uit leest) een gebruiker met UID:GID (UserId:GroupId) 1000:1000 dient aangemaakt te worden. Deze gebruiker kan vervolgens ongelimiteerd de bestanden lezen en schrijven naar de desgewenste volumes. De naam van de gebruiker heeft geen invloed, maar is meestal iets in de vorm van ‘testbed’.

Images

Om containers binnen pods op te starten zijn er images nodig waarop de container zich baseert. Om te garanderen dat images te allen tijde kunnen gereproduceerd worden, zijn binnen de Repository Dockerfiles voor elke pod voorzien die de bijhorende image volledig kunnen opbouwen. Meestal baseert een image zich echter op een andere basisimage. Zowel de basisimage alsook de geproduceerde image dienen beschikbaar te zijn binnen de Nokia omgeving onafhankelijk van externe bronnen zoals bv. de Docker hub. Daarom beschikt Nokia over een eigen interne image registry die alle basis- alsook geproduceerde images bevat, die vervolgens door de pods in de cluster kunnen binnengehaald worden.

4.1.4 Repository

Opdat de vooruitgang van elke developer consistent wordt opgeslagen en uitgewisseld, wordt gebruik gemaakt van een Gitlab repository. Hierin zit alle broncode die nodig is om de volledige omgeving van nul af aan op te bouwen. De repository bevat in eerste instantie een globale Makefile die het mogelijk maakt om a.d.h.v. enkele make commando’s de volledige omgeving op te bouwen, te updaten of te verwijderen. Achterliggend worden er verscheidene BASH-scripts aangeroepen die de effectieve handelingen uitvoeren. Bijvoorbeeld het ‘make allimages’ commando voert een BASH-script uit dat op zijn beurt alle Docker images waarvan de namen gedefinieerd in een bepaalde file, gaat bouwen volgens een vastgelegd stramien. Verder bestaat de repository uit mappen met een specifieke naam waarin vervolgens files zitten die elks een cruciale functie binnen de cluster vervullen.

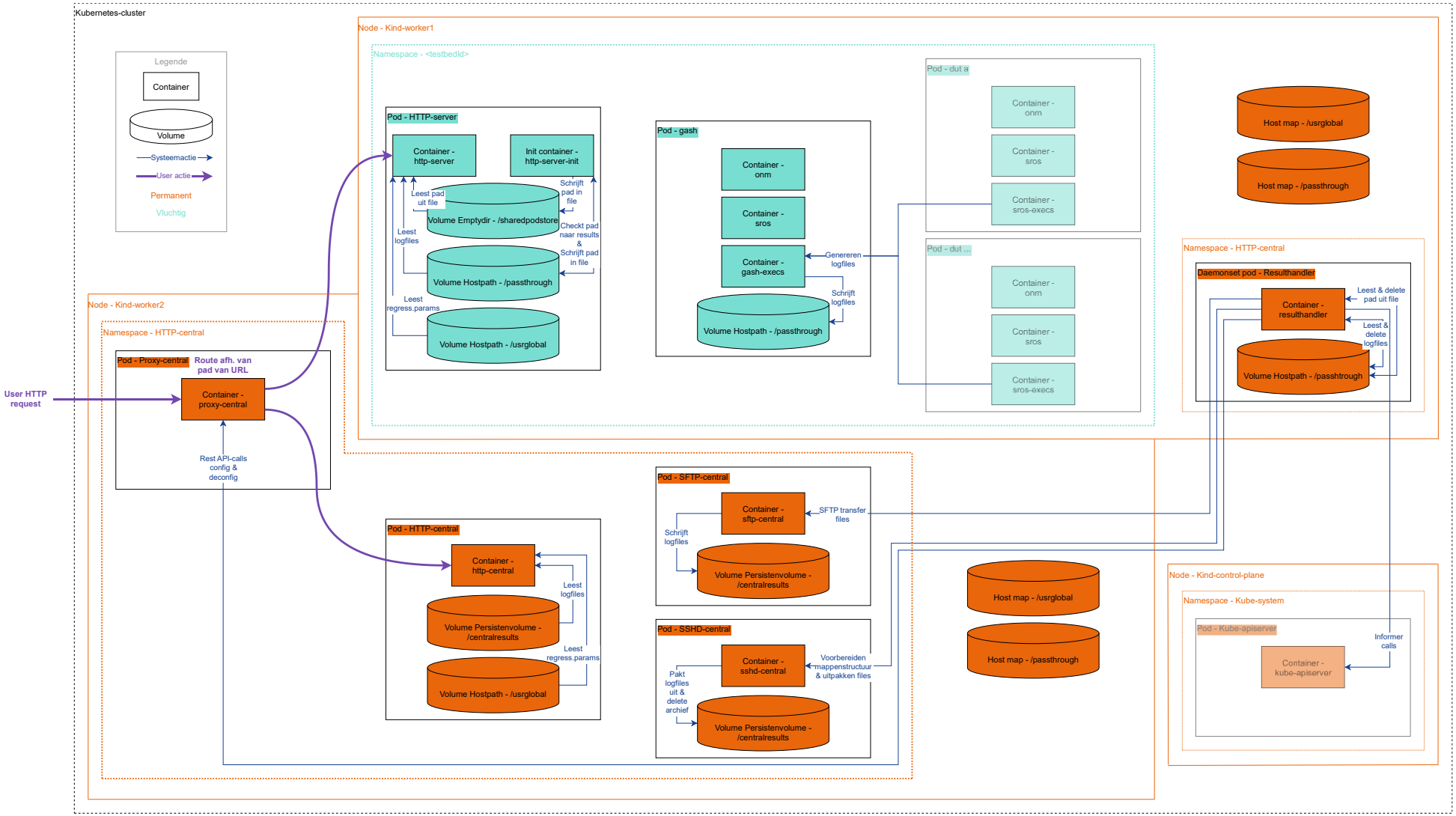
Over het algemeen kunnen de functies verdeeld worden in twee categorieën. Enerzijds de functies die permanent dienen beschikbaar te zijn doorheen de hele levensloop van de cluster, anderzijds functies die enkel gedurende een testrun beschikbaar zijn. Hierdoor zijn er ook twee manieren van implementeren. Aan de ene kant bestaat de statische manier voor permanente functies die aan de hand van Helm charts worden geïmplementeerd. Hierbij wordt gebruik gemaakt van op maat gemaakte YAML-files waarin variabelen worden gedefinieerd, die aanzien worden als templates binnen Helm. Aan de andere kant zijn er ook dynamische functies die enkel gedurende een testrun beschikbaar zijn. In dit geval wordt er gebruik gemaakt van de Golang client API voor Kubernetes. Hierbij is het mogelijk om dynamisch vanuit bv. een pod die draait in de cluster, met de cluster zelf te interageren. De interactie kan opgezet worden a.d.h.v. REST API-calls die vanuit een Go-programma met bijhorende libraries de cluster bereiken.

4.2 Concept

Dit deel geeft een beknopte werking van het systeem a.d.h.v. een zo compleet mogelijk schema met bijhorende uitleg. Verder kan het aanzien worden als basis voor het begrijpen van de volgende hoofdstukken.

Figuur 25 toont een schema waarin alle objecten en acties staan vermeld die noodzakelijk zijn om logfiles aan te leveren aan de eindgebruiker. Hierin wordt een onderscheid gemaakt tussen enerzijds vluchtige objecten die betrekking hebben op het aanleveren van logfiles tijdens een testrun, anderzijds op permanente objecten voor het aanleveren van logfiles nadat een testrun is afgelopen. Objecten die deels doorzichtig werden gemaakt zijn cruciaal in het proces, maar hierop werden verder geen aanpassingen doorgevoerd tijdens de uitvoering van de masterproef. Het dient opgemerkt te worden dat niet alle objecten binnen de figuur zijn opgenomen, enkel de noodzakelijke componenten om het concept dat binnen deze masterproef is onderzocht en geïmplementeerd te begrijpen.

Zodra een testrun gestart wordt en een tijdelijke namespace voor de testrun wordt aangemaakt, bereidt de gash pod de omgeving voor door o.a. een juiste mappenstructuur voor de logfiles te voorzien. Vervolgens onderschept de init container van de HTTP-server deze structuur, waarna dit wordt gecommuniceerd met de vluchtige HTTP-applicatiecontainer en de resulthandler a.d.h.v. een gedeeld volume. De lokale HTTP-server start op, de reverse proxyserver wordt geconfigureerd voor de huidige testrun op de DNS-naam van de lokale HTTP-server en de mappenstructuur van de testrun wordt op de centrale storage voorbereidt vanuit de resulthandler die coördineert. De software die getest wordt – die draait in de doorzichtige objecten in Figuur 25 – produceert logfiles die door de gash pod verwerkt worden en op een gedeeld lokaal volume van de node worden geplaatst. Een gebruiker kan a.d.h.v. een HTTP-aanvraag naar de reverse proxyserver de lopende testrun terugvinden en daardoor deze logfiles raadplegen. De HTTP-server van de testrun leest logfiles vanuit het gedeelde volume, inclusief andere benodigdheden voor de PHP-scripts die de pagina genereren. Nadat de testrun is afgelopen wordt de testrun verwijderd, wat de resulthandler opvangt als een event vanuit Kubernetes. De lokale logfiles worden gecomprimeerd, getransfereerd en tot slot uitgepakt op de centrale opslag, waarna ze lokaal verwijderd worden. De resulthandler past de configuratie in de reverse proxyserver aan om het pad van de testrun naar de centrale HTTP-server te laten verwijzen. Gebruikers kunnen als gevolg a.d.h.v. dezelfde URL de logfiles centraal raadplegen.



Figuur 25: Compleet schema van de werking voor de aanlevering van logfiles

4.3 Implementatie

Dit hoofdstuk beschrijft (de weg naar) de implementatie voor het aanleveren van logfiles a.d.h.v. HTTP-servers; enerzijds voor logfiles tijdens het uitvoeren van een testrun gedistribueerd over de verschillende nodes, anderzijds voor logfiles gecentraliseerd nadat een testrun is afgelopen.

4.3.1 Per testrun

Dit gedeelte focust op de implementatie en gedachtegang voor het aanleveren van logfiles via een HTTP-server tijdens het uitvoeren van een testrun. In Figuur 25 zijn dit voornamelijk de vluchtige blauwe pods die dit gedeelte vertegenwoordigen, waarop de verdere subkoppen van dit hoofdstuk zijn gebaseerd. Bovendien leunt dit hoofdstuk het dichtste aan tegen de reeds bestaande werking zonder Kubernetes waarbij testruns altijd binnen een testbed blijven.

Starten van een testrun

Het opstarten, updaten en verwijderen van een testrun (=deployment, niet te verwarren met Kubernetes deployment) gebeurt aan de hand van het intern ontwikkelde dsctl (digital sandbox control) create, update of delete deployment commando. Dit commando maakt API-calls naar de persistente pod 'ds-apiserver' beschikbaar in de cluster. Deze API-server is op zijn beurt verantwoordelijk voor het opvangen van de commando's en het uitvoeren van de gewenste acties binnen de cluster. Bij het aanmaken van een testrun wordt in eerste instantie een '<deploymentId>-create' pod aangemaakt. Deze pod gaat de zogenaamde deployservice starten die vervolgens d.m.v. de Kubernetes Golang API-client connectie maakt met de huidige Kubernetes-cluster. Daaropvolgend wordt een tijdelijke Kubernetes namespace aangemaakt waarin de nodige pods voor die specifieke testrun worden geplaatst. De namespace inclusief pods zijn zichtbaar in Figuur 25. Zodra de pods beschikbaar zijn, wordt de '<deploymentId>-create' pod verwijderd en wacht de 'ds-apiserver' op het update of delete commando om de testrun – overeenkomstig met een namespace inclusief pods – te updaten of terug te verwijderen.

Gash

Zoals eerder vermeld in Huidige structuur van logfiles, wordt gash in de bestaande omgeving o.a. gebruikt voor het uitvoeren van een testrun en het verwerken van de logfiles tot een consistent geheel. Het is echter belangrijk om enkele basisinzichten van deze instantie aan te halen door zijn cruciale werking binnen het genereren van logfiles. Gash bestaat binnen de cluster uit een pod met drie containers waarover geen verdere informatie wordt gegeven. Figuur 25 toont deze bewering inclusief het feit dat 'device(s) under test' (dut) pods logfiles genereren voordat gash ze behandelt en in het volume opslaat. Een dut kan aanzien worden als een instantie dat de te testen software van een service-router draait.

Schrijven van logfiles

In tegenstelling tot 3.1 waar een testbed permanent bewaard bleef en logfiles lokaal werden opgeslagen, is een testrun nu een vluchtig object waardoor de persistentie van de storage gegarandeerd dient te worden. Om dit op te lossen maakt gash gebruik van het hostPath volume in Kubernetes, wat zorgt voor lokale storagevoorzieningen op de node waarop de pod draait. Gegevens van een testrun worden met andere woorden tijdelijk op het lokale /passthrough volume van de node opgeslagen volgens het pad dat in Figuur 26 zichtbaar is. Dat wil ook zeggen dat pods die niet op dezelfde node draaien, in geen enkel geval aan de gegenereerde bestanden van de gash pod geraken.

/passthrough/testenvs/<deploymentId>/results/<jaar>/<maand>/<dag>/<HH:MM:SS.testEngineer>/

Figuur 26: Mappenstructuur van een testrun naar de logfiles binnen een node in Kubernetes

Het dient opgemerkt te worden dat het pad gelijkaardig is aan het pad in Figuur 17. Het verschil is dat er in dit geval een extra variabele wordt meegegeven dewelke de testrun identificeert. Hierdoor kan er op een node meer dan één testrun gelijktijdig draaien en opgeslagen worden. Het gedeelte na de <deploymentId> is echter exact hetzelfde als Figuur 17, door de PHP-scripts die dit soort structuur afdwingen voor een gewenste werking.

HTTP-server

Dit hoofdstuk beschrijft enerzijds waarom en op welke manier de HTTP-server als aparte pod werd geïmplementeerd per testrun, anderzijds het werkingsprincipe van deze pod dat gewaarborgd wordt door de configuraties in Kubernetes met de op maat gemaakte Docker images die in de containers worden opgestart. De HTTP-server is opgebouwd uit twee aparte containers die ook als een aparte kop behandeld worden.

Implementatie

Zoals reeds in het hoofdstuk Starten van een testrun vermeld, wordt de gash pod mee opgestart bij het starten een testrun. Hierdoor is het mogelijk om de HTTP-server waarmee logfiles worden aangeboden, mee te integreren in de gash containers net zoals in de huidige situatie zonder Kubernetes-cluster. Dit zou makkelijker te integreren zijn doordat hetzelfde systeem dan zowel de HTML-pagina's aanlevert alsook toont aan de gebruiker, zonder rekening te moeten houden met files die op meerdere locaties tezelfdertijd aanwezig dienen te zijn. Daartegenover zijn beide systemen afhankelijk van elkaar en zou eventuele schaalbaarheid naar de toekomst toe moeilijker te verwezenlijken zijn. Verder kan er performantieverlies optreden bij het uitvoeren van de testruns, indien er nog meer verantwoordelijken – die rechtstreeks beïnvloedbaar zijn door externe gebruikers – aan de gash pod worden toegekend. De nadelen van de HTTP-server integratie binnenin de gash pod wegen zwaarder door dan de voordelen, waardoor er gekozen is om de HTTP-server als een aparte pod per testrun of namespace te integreren.

Om de HTTP-server mee op te starten bij de aanvang van een testrun, dient er in de deployservice een extra functie voorzien te worden. De deployservice werd reeds vermeld bij het Starten van een testrun. Deze functie gaat dynamisch bij de aanmaak van een testrun a.d.h.v. in Go geschreven code (met behulp van enkele voorgemaakte functies in de Repository), API-calls maken door middel van de Kubernetes Golang client. Zodra de connectie is geïnitialiseerd, kan de juiste pod met bijhorende Kubernetes-configuraties worden opgestart in de cluster.

HTTP-server-init

Een init container is een container die als eerste in een pod opstart en succesvol dient af te sluiten voordat de gewone applicatiecontainer(s) binnen de pod opstart(en). Het doel is om de omgeving voor te bereiden op de applicatiecontainer die binnen de pod gaat draaien.

In deze context wordt een init container gebruikt om het pad naar de logfiles van de huidige testrun te bepalen zoals zichtbaar in Figuur 26 en vervolgens dit te communiceren met de effectieve HTTP-applicatieserver. Bij het opstarten van een testrun geeft Kubernetes de omgevingsvariabele DS_DEPLOYMENT_ID mee aan alle containers die draaien binnen die testrun. Deze ID is gelijk binnen de testrun, maar uniek binnen de cluster. Hierdoor kan de gash pod o.a. een map initialiseren a.d.h.v. de ID (zoals zichtbaar in het derde niveau van Figuur 26), waardoor de init container diezelfde map op hetzelfde volume kan raadplegen door dezelfde ID in de omgevingsvariabele. Er dient rekening gehouden te worden met het feit dat deze map enkel op die specifieke node beschikbaar is door het hostPath volume. Daardoor is er binnen Kubernetes een affinity op de HTTP-server pod voorzien, die verplicht om deze pod op exact dezelfde node als de gash pod van die testrun te plaatsen. Hierdoor

communiceren de HTTP-server en gash pod over exact dezelfde logfiles binnen het gedeelde /passthrough volume.

De pod is opgebouwd uit twee containers waarbij elke container is gegenereerd vanuit een Docker image. Deze image wordt voorbereid a.d.h.v. de Dockerfile die zichtbaar is in Figuur 27. Doordat de init container slechts enkele basiscommando's dient uit te voeren om de omgeving voor te bereiden, wordt gebruikgemaakt van een lichtgewicht Alpine image. Hierin wordt vervolgens een opstartscript gekopieerd met de juiste permissies.

```
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/alpine:latest
COPY http-testbedinit/initscript.sh /docker-entrypoint/initscript.sh
RUN chmod 775 /docker-entrypoint/initscript.sh
CMD [ "/bin/sh", "-c" , "/docker-entrypoint/initscript.sh" ]
```

Figuur 27: Dockerfile HTTP-server-init

Zodra de pod binnen de cluster wordt ingepland, start de init container waarbinnen het zelfgeschreven BASH-opstartscript draait. Het script checkt eerst de aanwezigheid van een map in de vorm van de deploymentId, vervolgens de aanwezigheid van de results map en tot slot een mappenstructuur die voldoet aan een structuur zichtbaar in Figuur 26 vanaf niveau vier. Dit laatste wordt gecheckt aan de hand van enkele Linux commando's die gebruikmaken van een RegEx string om het verkregen pad te evalueren. Vanaf dat er een overeenkomstige map is gevonden, wordt het pad naar de testrun alsook het pad naar de logfiles geschreven in een file op twee aparte volumes. Enerzijds naar het vluchtige emptyDir volume om het pad naar de applicatiecontainer te communiceren, anderzijds naar het hostPath volume om het pad naar de resulthandler (meer informatie in 4.3.2) te communiceren. Zodra de files zijn geschreven en de permissies correct zijn toegewezen, sluit het initialisatiescript en beëindigt de container succesvol. Figuur 28 toont het BASH-script dat wacht op de mappenstructuur.

```
#!/bin/sh
dspath="/passthrough/testenvs/$DS_DEPLOYMENT_ID"
found=0
echo "Start watching FS path till Gash pod creates complete results dir in: $dspath"
while [ $found -eq 0 ]; do
    #First check if DS_DEPLOYMENT_ID map is available
    if [ -d "$dspath" ]; then
        #Check if results map is available
        if [ -d "$dspath/results" ]; then
            testpath=$(find $dspath -type d | grep -m 1 -oh -E '(\/results\/)(20[0-9][0-9])\/(Month_[0-1][0-9])\/([A-Z]
[a-z]{2}_[0-3][0-9])\/([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9]\.([a-zA-Z0-9_]|\.|\-)*')
            #Check if path is available and testpath is not empty
            if [ -d "$dspath$testpath" ] && [ -n "$testpath" ]; then
                echo "$dspath$testpath directory found"
                echo "short path: $testpath"
                echo "$testpath" > /sharedpodstore/testpath
                #Give the right files for resulthandler in CSV format deployment_id,deployment_path,testrun_path
                echo "$DS_DEPLOYMENT_ID,$dspath,$testpath" > "/passthrough/resulthandler/$DS_DEPLOYMENT_ID"
                chown 1000:1000 "/passthrough/resulthandler/$DS_DEPLOYMENT_ID"
                chmod 775 "/passthrough/resulthandler/$DS_DEPLOYMENT_ID"
                echo "HTTP-server ready to start"
                found=1
            fi
        fi
    fi
done
```

Figuur 28: BASH-script bij het opstarten van de init HTTP-server

HTTP-server

Eens de init container succesvol wordt beëindigd, start de applicatiecontainer die de effectieve HTTP-server bevat. Deze container heeft als doel de HTTP-aanvragen van gebruikers te behandelen en bijhorende PHP-scripts uit te voeren. Het dient opgemerkt te worden dat de PHP-scripts onveranderd zijn gebleven bij de overgang naar Kubernetes.

In de Literatuurstudie rond Webservers werd geconcludeerd dat Nginx de beste keuze was voor de huidige implementatie indien de snelheid en populariteit primeren. Echter tijdens de implementatie werd duidelijk dat de huidige omgeving zonder Kubernetes-cluster gebruikmaakt van Apache webservers met bijhorende specifieke configuraties. Het omvormen van Apache-configuratiefiles naar Nginx-configuratiefiles zou enerzijds tijdrovend zijn, terwijl anderzijds de performantie van de Apache servers in de huidige implementatie nooit als problematisch werd ervaren. Om die redenen is er besloten om over te stappen naar de Apache webserver, waardoor er zo sneller tot een resultaat kon gekomen worden. Naar de toekomst toe kan er indien nodig onderzoek gedaan worden om de Apache-configuratiebestanden om te zetten naar Nginx-configuratiebestanden, om alsnog Nginx als webserver te implementeren.

De HTTP-server container is opgebouwd uit de basis PHP-image met Apache vanuit de Docker Hub zoals zichtbaar in Figuur 29 [62]. Daarenboven zijn de Apache-configuratiefiles inclusief opstartscripts toegevoegd aan de image. Vervolgens wordt in de Dockerfile van Figuur 29 een groep en een user aangemaakt met de nodige UID en GID om het lezen en schrijven naar het gedeelde volume mogelijk te maken. Ook de Apache webserver wordt ingesteld om te draaien volgens de aangemaakte gebruiker en de nodige rewrite module wordt ingeschakeld opdat specifieke paden worden doorverwezen. Tot slot start het opstartscript – hetgeen in de volgende alinea is uitgelegd –, waarna de effectieve HTTP-service wordt gestart.

```
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/php:7.4-apache
COPY http-testbed/apacheConfigFiles/ /etc/apache2/
COPY http-testbed/apacheScripts/ /DockerStartup/
RUN chmod 777 /DockerStartup/*
#Ensure the right user and group is assigned in order to let apache access the files
RUN groupadd -g 1000 www-data-testbed
RUN useradd -u 1000 -g 1000 -s /usr/sbin/nologin -M -d /var/www www-data-testbed
ENV APACHE_RUN_USER=www-data-testbed
ENV APACHE_RUN_GROUP=www-data-testbed
#Enable mods
RUN a2enmod rewrite
CMD [ "sh", "-c", "/DockerStartup/startup.sh ; apache2-foreground" ]
```

Figuur 29: Dockerfile HTTP-server

Zodra de init container is afgelopen start de HTTP-server met zijn opstartscript zoals zichtbaar in Figuur 30. Dit script maakt in eerste instantie de nodige mappen en linken aan, zodat de PHP-scripts de juiste files op de juiste locatie aangeleverd krijgen. Vervolgens wordt het pad uit het emptyDir volume gelezen dat door de init container is voorbereid, om daarna het juiste pad om op te starten (=DocumentRoot map) in de Apache-configuratiefiles weg te schrijven. Het pad verwijst naar de lokale node storage waarop gash de files van een testrun heeft weggeschreven. Nadat het opstartscript succesvol het pad in de configuratiefiles heeft weggeschreven, start de webserver en is deze klaar om aanvragen te verwerken.

```
#!/bin/sh
ln -s /passthrough/testenvs/$DS_DEPLOYMENT_ID /$HOSTNAME
mkdir -p /usr/global/bin
#Change url depending on path that init container recognizes in /passhtrough once results dir is available
shorttestrunpath=$(cat /sharedpodstore/testpath)
testrunpath="/passthrough/testenvs/$DS_DEPLOYMENT_ID$shorttestrunpath"
sed -i --expression "s|\${TESTRUNPATH}|$testrunpath|" /etc/apache2/sites-enabled/000-default.conf
sed -i --expression "s|\${SHORTTESTRUNPATH}|$shorttestrunpath|" /etc/apache2/sites-enabled/000-default.conf
```

Figuur 30: BASH-script bij het opstarten van de HTTP-server

Wat betreft de netwerkconnectiviteit dient er geen rechtstreekse externe connectie gemaakt te worden door gebruikers. Een reverse proxyserver zal zoals vermeld in de Proxy-central, de aanvraag intern in de cluster doorsturen tot bij de juiste HTTP-server die verantwoordelijk is voor die testrun. Daarom bevat deze pod enkel een service van het type ClusterIp.

4.3.2 Centraal

Dit hoofdstuk focust op de implementatie en gedachtegang voor het aanleveren van logfiles via een HTTP-server met bijhorende componenten, nadat een testrun is afgelopen. Figuur 25 geeft dit weer aan de hand van de oranje permanente componenten. Dit wordt genoemd naar de centrale implementatie omdat logfiles niet langer zijn gekoppeld aan een testrun, maar centraal zijn opgeslagen. Alle componenten binnen dit hoofdstuk zijn geaggregeerd binnen de Kubernetes namespace HTTP-central.

Implementatie

Om persistentie alsook centralisatie van loggegevens te bereiken, diende er een oplossing gevonden te worden om logfiles blijvend te kunnen raadplegen nadat een testrun is afgelopen. In tegenstelling tot de implementatie Per testrun waar dynamisch a.d.h.v. in Go geschreven code een nieuwe pod werd ingepland, dient er hier een statische en blijvende methode gebruikt te worden. Daarom is er gekozen om Helm charts te gebruiken met onderliggend YAML-files, net zoals andere gelijksoortige applicaties die reeds binnen de bestaande Repository werden geïmplementeerd. Het grote voordeel is dat er slechts eenmaal bij de opbouw van de cluster een commando dient uitgevoerd te worden, waardoor Kubernetes op elk moment de situatie zoals in de YAML-files probeert na te streven. Daarenboven kunnen alle objecten zoals bv. namespaces, deployments, daemonsets of services op die manier binnen de cluster uitgedrukt worden op een overzichtelijke manier.

HTTP-central

De centrale HTTP-server als pod binnen de HTTP-central namespace is de hoofdfunctionaliteit binnen het concept voor het aanleveren van logfiles. Aanvragen van gebruikers voor het verkrijgen logfiles, zullen in eerste instantie deze webserver passeren afhankelijk of het pad naar de gezochte testrun al dan niet is gekend. Resultaten worden nadat een testrun is afgelopen opgeslagen in een centraal persistent volume, waardoor de HTTP-server de resultaten kan raadplegen. Naast het persistent volume wordt ook het /usr/global volume gemount, dat op zijn beurt het regress.params bestand aanlevert zoals ook in Huidige structuur van logfiles werd aangehaald.

Persistent volume

Een persistent volume (PV) wordt aanzien als een beschikbaar resource binnen Kubernetes. Om dit resource te laten gebruiken door bijvoorbeeld een pod, dient er een persistent volume claim (PVC) aan de pod gelinkt te zijn. Een PVC is een aanvraag aan het PV om storage te mogen gebruiken.

Binnen de cluster is er een permanente NFS-client-provisioner pod aanwezig die rechtstreekse connectie heeft met een geëmuleerde externe NFS-server. Hiermee kan er persistent storage aangevraagd worden, waarna deze storage aan pods kan worden toegekend o.b.v. enkel en alleen een persistent volume claim. In de statische YAML-file van de HTTP-central deployment, wordt de PVC gelinkt aan de pod waarna er storage beschikbaar komt in de container van de centrale HTTP-server. Deze persistent storage wordt gebruikt om logfiles centraal en permanent op te slaan buiten de Kubernetes-cluster. Dezelfde PVC wordt ook gelinkt aan de SFTP-central en SSHD-central.

Het volume bevat alle nodige bestanden om een testrun aan te leveren via een webserver die gebruik maakt van PHP-scripts om de inhoud op een vertrouwde manier weer te geven. Logfiles moeten gecentraliseerd beschikbaar zijn op een consistente manier zonder de logica van de PHP-bestanden te breken. Dit laatste cruciaal vereiste heeft ervoor gezorgd dat dezelfde mappenstructuur werd gekozen zoals in Figuur 17, waarbij enkel de root directory is aangepast naar de centrale storage zoals zichtbaar in Figuur 31. Testruns zijn nog steeds uniek o.b.v. de tijd en de persoon die test start.

/centralresults/results/<jaar>/<maand>/<dag>/<HH:MM:SS.testEngineer>/

Figuur 31: Mappenstructuur naar de logfiles van een testrun op de centrale persistente storage

Docker image

De Docker image is net zoals de HTTP-server per testrun en om dezelfde reden opgebouwd uit een basis PHP-image waarbij Apache is geïntegreerd als webserver. Op maat gemaakte configuraties worden mee in de image verwerkt in combinatie met de nodige default files van bv. de status map zoals vermeld in Huidige structuur van logfiles. Om de Repository alle Dockerfiles te laten bevatten maar ook vrij te houden van grote bestanden, werd een container gemaakt die op de private image registry werd gezet met daarin de gecomprimeerde default bestanden. Deze bestanden worden vanuit de aparte image in de HTTP-server gekopieerd en uitgepakt. Figuur 32 toont de Dockerfile die een image maakt voor de HTTP-central pod. Het dient opgemerkt te worden dat de pod opstart door middel van een zelfgeschreven Opstartscript waarna de effectieve Apache HTTP-service opstart.

```
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/generalwebfiles:latest as generalWebfiles
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/php:7.4-apache
COPY http-central/apacheConfigFiles/ /etc/apache2/
COPY http-central/apacheScripts/ /DockerStartup/
#Ensure startup scripts have the right permissions
RUN chmod 777 /DockerStartup/*
#Ensure the right user and group is consistently assigned
RUN groupadd -g 1000 www-data-testbed
RUN useradd -u 1000 -g 1000 -s /usr/sbin/nologin -M -d /var/www www-data-testbed
ENV APACHE_RUN_USER=www-data-testbed
ENV APACHE_RUN_GROUP=www-data-testbed
#Enable mods
RUN a2enmod rewrite
#Ensure status/ and regress_data/ is centrally available using the alpine image that contains tar.gz
COPY --from=generalWebfiles /generalWebfiles/generalWebfiles.tar.gz /tmp
RUN tar -xvf /tmp/generalWebfiles.tar.gz -C /tmp/
RUN rm /tmp/generalWebfiles.tar.gz
CMD [ "sh", "-c", "/DockerStartup/startup.sh ; apache2-foreground" ]
```

Figuur 32: Dockerfile HTTP-central

Opstartscript

Zodra de pod in de cluster is opgestart, start het script zoals zichtbaar in Figuur 33. Dit BASH-script zorgt dat het persistent volume wordt opgevuld of geüpdatet met correcte files die de PHP-scripts nodig hebben. Daarnaast wordt ook een testfile gemaakt en worden de permissies indien nodig nogmaals correct gezet. Vanaf dit script is afgelopen, start de Apache webserver.

```
#!/bin/sh
ln -s /centralresults /$HOSTNAME
# If results dir not exists, make it and ensure base files are present
mkdir -p /centralresults/results
cp -r -u -v /tmp/status /centralresults/
cp -r -u -v /tmp/regress_data /centralresults/
rm -R /tmp/status
rm -R /tmp/regress_data
echo "<h1>Central HTTP server</h1>" > /centralresults/results/testindex.html
chown 1000:1000 /centralresults/results
chown -R 1000:1000 /centralresults/status
chown -R 1000:1000 /centralresults/regress_data
chown 1000:1000 /centralresults/results/testindex.html
```

Figuur 33: Opstartscript HTTP-central

SFTP-central

Om het netwerk zo min mogelijk te belasten alsook om zo weinig mogelijk fouten te verkrijgen gedurende meerdere testruns, is er besloten om gebruik te maken van een eenmalige transfer van logfiles over het netwerk naar de persistent store. Gedurende een testrun worden files lokaal op een node opgeslagen zoals in het hoofdstuk HTTP-server vermeld. Nadat een testrun is afgelopen worden deze files gecomprimeerd en overgezet naar de centrale storage a.d.h.v. een SFTP-server (SSH File Transfer Protocol). Deze centrale SFTP-server is een pod die nauw samenwerkt met de HTTP-central. De enige functie van deze pod is om logfiles na een testrun te ontvangen en vervolgens op dezelfde persistent storage te plaatsen als de plaats waarop de HTTP-server deze raadpleegt.

Docker image

Ook deze Docker image is gebaseerd op een image die reeds bestond in de Docker hub online. Figuur 34 toont de Dockerfile die de image voor de SFTP-server genereert. Hierin wordt eerst en vooral de correcte gebruiker aangemaakt om op de centrale storage met de juiste permissies te werken. Vervolgens wordt chrooting uitgezet in de configuratiefile om te voorkomen dat de permissies van de bovenliggende mappen in het pad enkel door de root user mogen beheerd worden. Zodra de pod in de cluster staat, wordt de SFTP-server gestart.

```
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/sftp-server:debian
# Ensure FTP users are corresponding to the users to access the shared volume
ENV SFTP_USERS=www-data-central:                    :1000:1000
RUN groupadd -g 1000 www-data-central
RUN useradd -u 1000 -g 1000 -s /usr/sbin/nologin -p '
' -M -d /centralresults www-data-central
#Disable chrooting to access shared volume without changing dir permissions
RUN sed -i --expression "s|ChrootDirectory %h||" /etc/ssh/sshd_config
```

Figuur 34: Dockerfile SFTP-central

SSHD-central

De SSHD (Secure SHell Daemon) is naast de SFTP-server medeverantwoordelijk voor het compleet afleveren van de logfiles op de centrale storage die gelinkt is aan de voorgaande twee pods. De SSHD pod biedt de mogelijkheid om remote vanuit andere pods acties uit te voeren op de centrale storage. Dit wordt o.a. binnen het proces voor het afleveren van logfiles gebruikt om de mappenstructuur zoals zichtbaar in Figuur 31 voor te bereiden. Deze functionaliteit is noodzakelijk om eindgebruikers de mogelijkheid te geven naar hun gestarte testrun te browsen. Bovendien wordt de SSHD gebruikt om de

gecomprimeerde .tar.gz file – die via de SFTP-central op de centrale storage werd gezet – uit te pakken op de correcte locatie, zodat de HTTP-central de files kan raadplegen.

Docker image

De Dockerfile van de SSHD-central pod is zichtbaar in Figuur 35. Hier worden enkele instellingen meegegeven die default a.d.h.v. omgevingsvariabelen in de image worden toegepast. De mogelijkheid tot SFTP wordt uitgeschakeld omdat er reeds een andere pod deze functionaliteit heeft meegekregen. Zo is er een duidelijke scheiding tussen beide acties en moet de actie dat het voorbereiden van de mappenstructuur op de centrale storage doet, nooit wachten op het overzetten van een testrun. Het overzetten kan echter enige tijd duren afhankelijk van de grootte van de testrun waarover meer info in het hoofdstuk rond het Dimensioneren van logfiles is terug te vinden. Verder wordt bij het creëren van de Docker image ook een script uitgevoerd dat het wachtwoord van de gebruiker instelt volgens de principes die in de basisimage werden toegelicht.

```
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/sshd-server:latest
ENV SSH_USERS=www-data-central:1000:1000
ENV SSH_ENABLE_PASSWORD_AUTH=true
ENV DISABLE_SFTP=true
COPY sshd-central/setpassword.sh /etc/entrypoint.d/
RUN chmod 775 /etc/entrypoint.d/setpassword.sh
```

Figuur 35: Dockerfile SSHD-central

Proxy-central

Reeds twee soorten HTTP-servers werden geïntroduceerd in de cluster. Enerzijds de HTTP-server per testrun en anderzijds de HTTP-central. Om eindgebruikers geen verwarring te laten ervaren, is er één toegangspunt gecreëerd – namelijk de proxy-central – waarop HTTP-aanvragen voor testruns binnenkomen. Doordat dit het enige toegangspunt is van buitenaf de cluster, is dit ook de enige pod die in deze masterproef voor HTTP-aanvragen is geconfigureerd als een NodePort zoals uitgelegd in het hoofdstuk Netwerkconnectiviteit.

Bij het implementeren werden oorspronkelijk twee mogelijkheden vergeleken. Enerzijds de mogelijkheid om een centrale HTTP-server te voorzien die HTTP-aanvragen redirect naar de testbed-specifieke webservers. Anderzijds een reverse proxyserver die ook als load balancer en enig contactpunt dient voor alle binnenkomende aanvragen. Het grote voordeel bij het gebruik van een reverse proxyserver is dat er slechts één service naar buiten gebracht wordt, waarbij alle aanvragen verder intern in de cluster verwerkt worden. Verder was er reeds een uitgevoerde Literatuurstudie naar Load Balancers, die HAProxy als meest performante keuze naar voren bracht waarbij containeriseren perfect mogelijk is. Daarentegen zou het redirecten van HTTP-aanvragen steeds afhankelijk zijn van de centrale server, maar zou het wel makkelijker te implementeren zijn. Samengevat is het gebruik van HAProxy als reverse proxyserver de beste oplossing om de huidige functionaliteit te bekomen.

Werkingsprincipe

De structuur van de laatste vier mappen van testruns die lokaal zijn opgeslagen zoals in Figuur 26, komt overeen met de structuur van de laatste vier mappen van centraal opgeslagen testruns zoals in Figuur 31. Daarenboven is de eerste map telkens de results map die in een URL achter het IP-adres of DNS-adres van beide HTTP-servers komt. Hierdoor kan HAProxy enkel al op basis van de eerste vijf mappen van het pad van een binnenkomende HTTP-aanvraag beslissen naar waar de aanvraag dient te gaan. Indien het pad aanwezig is in de configuratie, forwardt HAProxy de aanvraag naar desgewenste HTTP-server per testrun. Indien het pad niet aanwezig is, wordt de aanvraag naar de centrale server geforward.

Echter dient er rekening gehouden te worden met het feit dat de PHP-pagina's worden hergebruikt vanuit de bestaande implementatie. De indexpagina van elke testrun bestaat met andere woorden uit meerdere frames zoals eerder vermeld in de Huidige structuur van logfiles. Deze frames roepen op hun beurt bepaalde bestanden op die buiten de map van de testrun zijn gevestigd (bv. in de /status/ map). Daarom wordt als tweede parameter ook de 'referer' HTTP-header nagekeken door HAProxy. Deze HTTP-header bevat informatie over de locatie van waar de aanvraag naar het huidige object komt. Hiermee kan dus onrechtstreeks het pad naar de testrun bepaald worden, om te voorkomen dat een gedeelte van de aanvragen telkens naar de centrale HTTP-server worden gestuurd.

Figuur 36 toont een gedeeltelijke momentopname van een voorbeeldconfiguratie van de reverse proxyserver. In deze voorbeeldconfiguratie staat ingesteld hoe HTTP-aanvragen worden geforward. De front-end is de kant die naar buiten wordt gebracht op poort 80, waarop de aanvragen van eindgebruikers binnenkomen. Als een aanvraag binnenkomt wordt dit sequentieel gematcht met mogelijke back-end switching rules waarbij de eerste overeenkomende configuratie wordt gebruikt. Vandaar dat als laatste back-end switching rule de centrale HTTP-server staat vermeld. Daarenboven wordt elke back-end gecheckt op beschikbaarheid a.d.h.v. het 'check' trefwoord in de configuratie. Indien een server niet online is, zal de aanvraag naar de volgende overeenkomende back-end worden gestuurd. Het dient opgemerkt te worden dat hier wordt gebruikgemaakt van DNS-namen die enkel intern in de cluster gelden.

```
frontend public
     bind :80

     use_backend star if { path /results/2022/Month_12/Dec_16/15:27:45.arne.papen } || { path_beg /results/2022/Month_12/Dec_16/15:27:45.arne.papen/ }
     || { hdr_dir(Referer) -i /results/2022/Month_12/Dec_16/15:27:45.arne.papen } || { hdr_dir(Referer) -i /results/2022/Month_12/Dec_
     16/15:27:45.arne.papen/ }

     #Backend central if no specified testrun matched
     use_backend central

backend central
     server s1 http-central.http-central.svc.cluster.local:80 check
backend star
     server Server1 star-http-server.ns-star.svc.cluster.local:80 check
```

Figuur 36: Momentopname van een voorbeeldconfiguratie van HAProxy voor het forwarden van HTTP-aanvragen

Data Plane API

Doordat testruns afhankelijk van de vraag worden opgestart, alsook dat testruns na de afloop transfereren naar de centrale server, dient de configuratie van de reverse proxy dynamisch worden aangepast. HAProxy heeft hiervoor de Data Plane API add-on ontwikkeld. Deze add-on maakt het mogelijk om de configuraties dynamisch toe te voegen, aan te passen of te verwijderen aan de hand van REST API-calls. Bovendien kan dit verwezenlijkt worden zonder enige downtime van de server.

Door de Data Plane API is het mogelijk om dynamisch bij het opstarten van een nieuwe testrun, een nieuwe back-end alsook back-end switching rule toe te voegen. Gebruikers die naar een specifiek pad surfen, worden als gevolg doorverwezen naar de testbed-specifieke HTTP-server. Omdat bij het starten van een testrun het pad in de URL nooit geweten is, bereidt de SSHD-central de mappenstructuur steeds voor op de centrale storage. Een gebruiker kan als het ware browsen naar zijn test op de centrale server, waarna HAProxy de gebruiker doorstuurt naar de specifieke HTTP-server bij het effectief openen van de map van de testrun. Dit concept wordt aangetoond aan de hand van een voorbeeld onder het hoofdstuk Resultaten Na de opstart van een testrun. Zodra een testrun is afgelopen en de logfiles zijn getransfereerd, dienen de toegevoegde configuraties in HAProxy opnieuw ongedaan gemaakt te worden.

Docker image

Het genereren van de op maat gemaakte Docker image gebeurt aan de hand van de HAProxy Debian image als basis. Figuur 37 toont de Dockerfile waarin de default configuratiebestanden worden meegegeven. Deze configuratiebestanden voor HAProxy bevatten enerzijds de configuratie voor de centrale HTTP-server die permanent online is, anderzijds de configuratie voor de Data Plane API die mee in de image als een add-on zit verwerkt. Een voorbeeld van de functionele configuratie werd reeds in Figuur 36 getoond.

```
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/haproxy-debian:2.6.6
COPY proxy-central/haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
COPY proxy-central/dataplaneapi/dataplaneapi.yaml /var/lib/haproxy/dataplaneapi/
RUN chown -R haproxy:haproxy /var/lib/haproxy/dataplaneapi/
EXPOSE 80
EXPOSE 443
EXPOSE 5555
```

Figuur 37: Dockerfile Proxy-central server

Resulthandler

Naast de aparte pods in de cluster met elks hun eigen functionaliteiten is er ook een zogenaamde resulthandler die per testrun de coördinatie verzorgt. De resulthandler is een in Go geschreven daemonset die de status van een testrun op de voet volgt. De voornaamste functionaliteiten zijn het transfereren van de testruns naar de centrale storage indien nodig en het up-to-date houden van de reverse proxyserver met testruns die lokaal aan het draaien zijn.

Een daemonset binnen Kubernetes zorgt default dat op elke worker node een desgewenste kopie van de pod wordt ingepland. Binnen de testomgeving zullen er dus twee dezelfde pods worden ingepland, beide op een andere node. Elke pod binnen de daemonset is verantwoordelijk voor de testruns die op zijn specifieke node draaien. Het voordeel om op elke node één resulthandler pod te hebben is in eerste instantie om een single point of failure te voorkomen in de volledige cluster. Daarenboven is het een interessant gegeven dat er per node hoe dan ook een pod moest zijn die de lokale storage van elke node moest opkuisen. Het opkuisen van de lokale storage is nu medeverantwoordelijkheid van de resulthandler wat perfect in het takenpakket past.

Werkingsprincipe

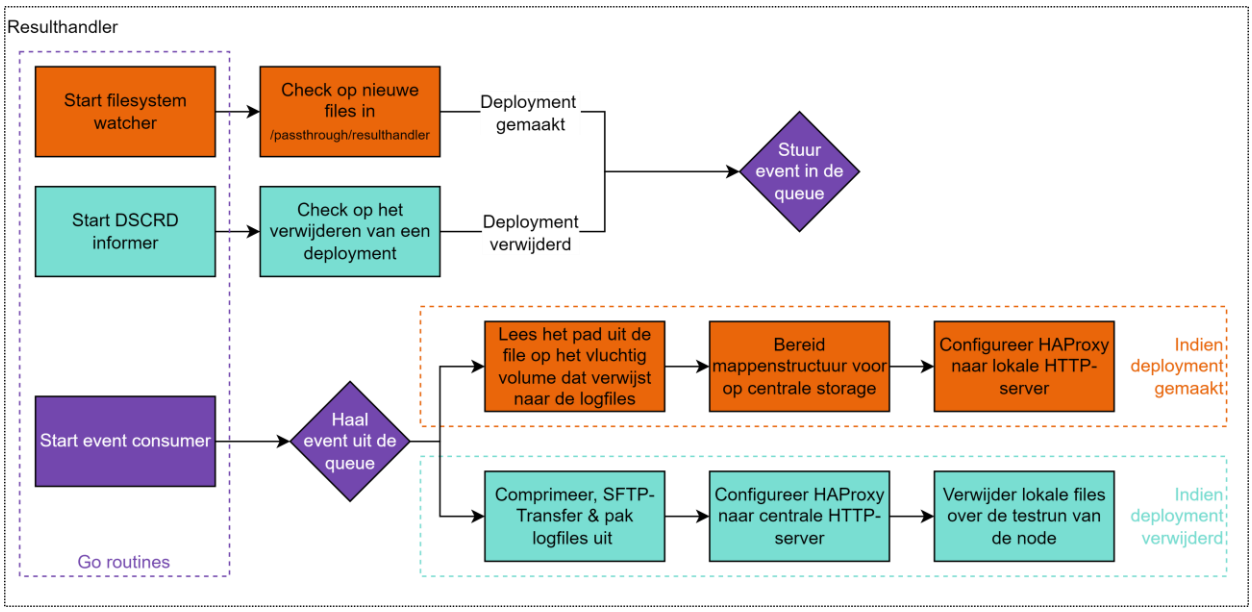
Figuur 38 toont de inwendige werking van de resulthandler a.d.h.v. een blokschema. Bij de start van het programma worden er drie routines gestart die parallel t.o.v. elkaar draaien. De communicatie tussen de drie routines gebeurt aan de hand van een queue waar events kunnen ingezet en uitgehaald worden d.m.v. het First In First Out (FIFO) principe.

De eerste routine, namelijk de filesystem watcher, is verantwoordelijk voor het opvolgen van files die in het lokale node volume met pad /passthrough/resulthandler terechtkomen. Op deze locatie worden files geschreven met daarin het pad naar de nieuwe testrun en de bijhorende logfiles die door de init containers van de HTTP-servers per testrun zijn aangemaakt. Zodra een nieuwe file in dit pad is opgemerkt (bij de aanmaak van een nieuwe testrun) door de resulthandler, wordt er een nieuw event in de queue geplaatst waarna de event consumer dit event behandelt. Het pad naar de testrun en het pad naar de bijhorende logfiles worden uit de desbetreffende file gelezen, waarna de resulthandler de mappenstructuur voor de logfiles aanmaakt op de centrale storage d.m.v. de SSHD-central zoals ook in Figuur 25 zichtbaar. De resulthandler stelt zich als SSH-client op richting de SSH-server om de mappenstructuur aan te maken. Vervolgens wordt d.m.v. REST API-calls de reverse proxyserver

geconfigureerd zodanig dat HTTP-aanvragen richting de testrun worden doorgestuurd naar de lokale HTTP-server van die testrun. Een voorbeeldconfiguratie is zichtbaar in Figuur 36.

De tweede routine is verantwoordelijk voor het opvangen van de trigger bij het verwijderen van een deployment of testrun. Hiervoor is een zogenaamde Kubernetes-informer vereist dewelke een cruciale functie vervult binnen de resulthandler. Een informer maakt het mogelijk om een client-programma voortdurend updates te laten ontvangen op acties zoals het aanmaken, updaten of verwijderen van (custom) resources zoals pods, namespaces, … in de Kubernetes-cluster. Dit is ook de voornaamste reden dat Go als programmeertaal is gekozen voor de resulthandler, vermits de Golang Kubernetes-client de enige programmeertaal is die het concept van informers volledig ondersteunt. In het geval van de resulthandler wordt een informer gestart op de custom resource definition van de deployment service (DSCRD). Vanaf het ogenblik dat een deployment verwijderd wordt (d.m.v. het 'dsctl delete deployment' commando), krijgt elke resulthandler een event binnen dat vervolgens in elke queue van elke resulthandler terechtkomt. Zodra de consumer het event ophaalt, start het proces voor het transfereren en verwijderen van de lokale logfiles op enkel de daarvoor bevoegde resulthandler. Deze trigger comprimeert in eerste instantie alle logfiles naar een .tar.gz file op het lokale volume van de node (/passthrough/testenvs/<deploymentId>/results). In tegenstelling tot het on-the-fly transfereren van logfiles tijdens een testrun, wordt slechts eenmaal op het einde van een testrun connectie gemaakt met de SFTP-central om alle files als een gecomprimeerd archief te transfereren. Dit heeft als voordeel dat netwerktrafiek alsook de kans op fouten drastisch vermindert, aangezien dit op voorhand de grootste bezorgdheid van Nokia was. Aansluitend, wordt er nogmaals connectie gemaakt met de SSHD-central om eerst en vooral het archief uit te pakken en daarna het archief te verwijderen. Alle logfiles zijn nu centraal beschikbaar waardoor a.d.h.v. REST API-calls de configuratie van de HAProxy voor die specifieke testrun ongedaan gemaakt wordt. Dit heeft als gevolg dat een HTTP-aanvraag naar die testrun de default configuratie volgt en dus wordt doorverwezen naar de centrale HTTP-server. Tot slot worden de lokale bestanden van de afgelopen testrun verwijderd van de storage van de node.

De horizontale rechthoeken in stippenlijnen uit Figuur 38, bevatten cruciale acties die sterk beïnvloed kunnen worden door parameters buiten het effectief programma. Zo kan er bv. in het netwerk iets mislopen tijdens de API-calls naar de HAProxy, waardoor de resulthandler een onverwacht resultaat binnenkrijgt. Dit wordt gedeeltelijk opgevangen door een ingebouwd retry-mechanisme dat maximaal vijf keer – per actie in de horizontale rechthoeken – met een random wachttijd opnieuw diezelfde actie gaat proberen. Indien er na vijf keer nog steeds een fout optreedt, crasht de resulthandler met een bijhorend logbericht. Dit bericht wordt vervolgens opgevangen door een centrale pod die niet binnen de scope van deze masterproef wordt behandeld. Kubernetes zal in dit geval de pod opnieuw opstarten, maar er dient wel actie ondernomen te worden om de fout die werd gelogd na te gaan.



Figuur 38: Blokschema van de in Go geschreven software van de resulthandler

Docker image

Doordat meer dan 80% van de Repository is geschreven in Go, is er een methode ontwikkeld die a.d.h.v. een tijdelijke Golang compiler alle aanwezige Go-programma's compileert tot machinetaal, die vervolgens in aparte applicatiecontainers gaan draaien. Figuur 39 toont de Dockerfile van de resulthandler. Hierin worden eerst de nodige packages geïnstalleerd die het eventueel debuggen in latere stadia versimpelen, alsook de mogelijkheid tot het gebruik van sudo. In standaardomstandigheden draait de applicatie in de container onder gebruiker root met ID nul. Aangezien de resulthandler het lokale node volume moet kunnen lezen alsook naar kunnen schrijven, dient er een nieuwe gebruiker (met ID 1000:1000) aangemaakt te worden die het programma gaat draaien. Daarenboven is het noodzakelijk dat de nieuwe gebruiker ook root permissies krijgt mits gebruik van de sudo prefix. Tot slot dient ook opgemerkt te worden dat het gecompileerde resulthandler programma uit de gobuilder container wordt gekopieerd naar de lichtgewicht Alpine applicatiecontainer die de code effectief gaat uitvoeren.

```
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/gobuilder:latest AS builder
FROM registry.gitlabsr.nuq.ion.nokia.net/sr/linux/gatenet/tools/alpine:latest
RUN apk update && apk add git bash sqlite openssh curl busybox-extras sudo
# Add testbed user & grant sudo access (wheel = sudo as in debian)
RUN adduser  --disabled-password --home /testbed --uid 1000 testbed -G wheel
RUN echo '%wheel ALL=(ALL:ALL) NOPASSWD: ALL' > /etc/sudoers.d/wheel
# Copy builded application to the container
COPY --from=builder /app/resulthandler /app
VOLUME [ "/var/resulthandler" ]
# Run it as user testbed
USER testbed
ENTRYPOINT ["/app/resulthandler"]
```

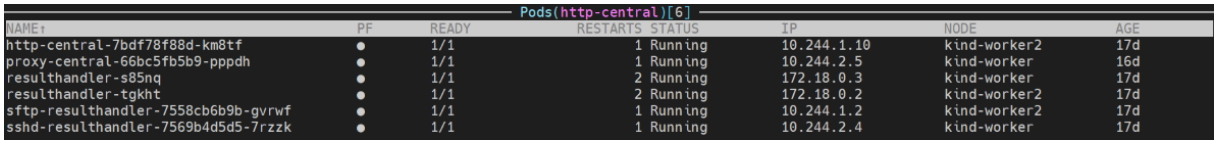
Figuur 39: Dockerfile resulthandler

4.4 Resultaten

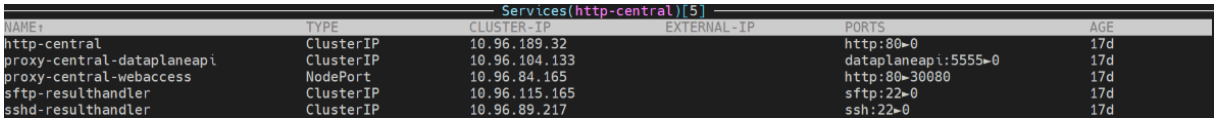
Voorgaande hoofdstukken focusten voornamelijk op het design en de individuele implementatie binnen de cluster. Dit hoofdstuk daarentegen focust op de bekomen resultaten als één chronologisch geheel door elk van de voorgaande componenten te implementeren volgens het uitgedachte Concept binnen de testomgeving. De screenshots van dit hoofdstuk zijn voornamelijk genomen uit de k9s client tool voor Kubernetes, aangezien deze tool een geaggregeerde bron van informatie over de Kubernetes-cluster bevat.

4.4.1 Na de opstart van de cluster

Vanaf het ogenblik dat de cluster in gebruik genomen wordt, starten alle objecten die persistent binnen de cluster aanwezig dienen te zijn. Binnen deze masterproef zijn dat enkel objecten binnen de namespace HTTP-central. Kubernetes deployments worden vertaald in pods zoals zichtbaar in Figuur 40, en de bijhorende services om netwerktrafiek mogelijk te maken zijn zichtbaar in Figuur 41. Figuur 42 toont de beschikbare resulthandler als een daemonset in de HTTP-central namespace, die op zijn beurt twee pods op de worker nodes plaatst zoals zichtbaar in Figuur 40.



Figuur 40: Persistente pods voor het aanleveren van logfiles binnen de Kubernetes-cluster

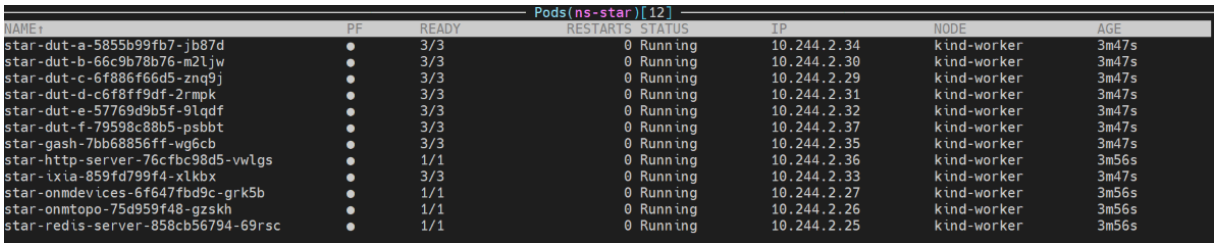


Figuur 41: Services die bij de persistente pods voor het aanleveren van logfiles binnen de Kubernetes-cluster horen

Figuur 42: Daemonsets voor het aanleveren van logfiles binnen de Kubernetes-cluster

4.4.2 Na de opstart van een testrun

Een testrun of deployment start a.d.h.v. het ‘dsctl create deployment’ commando zoals aangehaald binnen het hoofdstuk Starten van een testrun. Elke testrun krijgt op zijn beurt bij de opstart een bepaalde DS_DEPLOYMENT_ID. Deze ID wordt buiten de HTTP-server-init ook gebruikt als naam voor de namespace en als prefix voor alle pods binnen diezelfde namespace. De ID die voor deze test gebruik wordt is ‘star’. Figuur 43 toont alle pods van een opgestarte testrun waarbij de ‘star-http-server’ de belangrijkste is voor deze masterproef. Verder dient er ook opgemerkt te worden dat zowel de ID alsook de naam en het pad voor de testrun nog manueel dienen ingevoerd te worden voor het starten van een testrun.



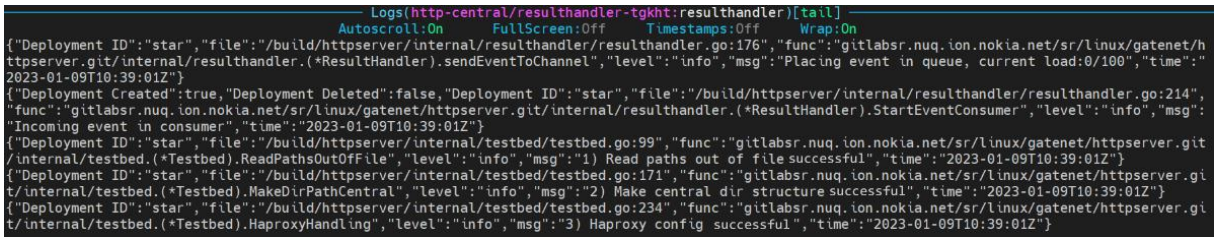
Figuur 43: Vluchtige pods van een testrun binnen de Kubernetes-cluster

De vluchtige HTTP-server bevat twee containers zoals zichtbaar in Figuur 44. De init container dient eerst succesvol af te lopen vooraleer de applicatiecontainer start. Figuur 45 toont de logs die de init container heeft geschreven. Hieruit blijkt dat het benodigde pad naar de logfiles – dat is voorzien door de gash pod – uit het gedeelde volume is gevonden. Dit pad wordt als gevolg gebruikt binnen de HTTP-server om te verwijzen naar de correcte map op het lokale node volume, alsook om de resulthandler op hetzelfde pad af te stemmen.

Figuur 44: Containers van de HTTP-server pod binnen de Kubernetes-cluster

Figuur 45: Logs van de afgelopen HTTP-server-init container binnen de Kubernetes-cluster

Zodra de filesystem watcher van de resulthandler de file met daarin het pad waarneemt, wordt een event getriggerd en start het proces voor het aanbieden van logfiles via de lokale HTTP-server. Figuur 46 toont de logs van de resulthandler nadat de trigger in de software is binnengekomen. Het dient opgemerkt te worden dat deze logs een JSON-structuur volgen zodat deze gestructureerd door een externe applicatie kunnen worden verwerkt, waarbij het message-attribuut de effectieve boodschap van de log weergeeft. Met dit in het achterhoofd, zijn er ook duidelijk drie verschillende stappen terug te vinden in de logs van Figuur 46 die reeds aangehaald werden in het blokschema van Figuur 38. In dit geval zijn alle acties succesvol verlopen.



Figuur 46: Logs van de resulthandler bij het voorbereiden van de omgeving om logfiles aan te bieden via de lokale HTTP-server binnen de Kubernetes-cluster

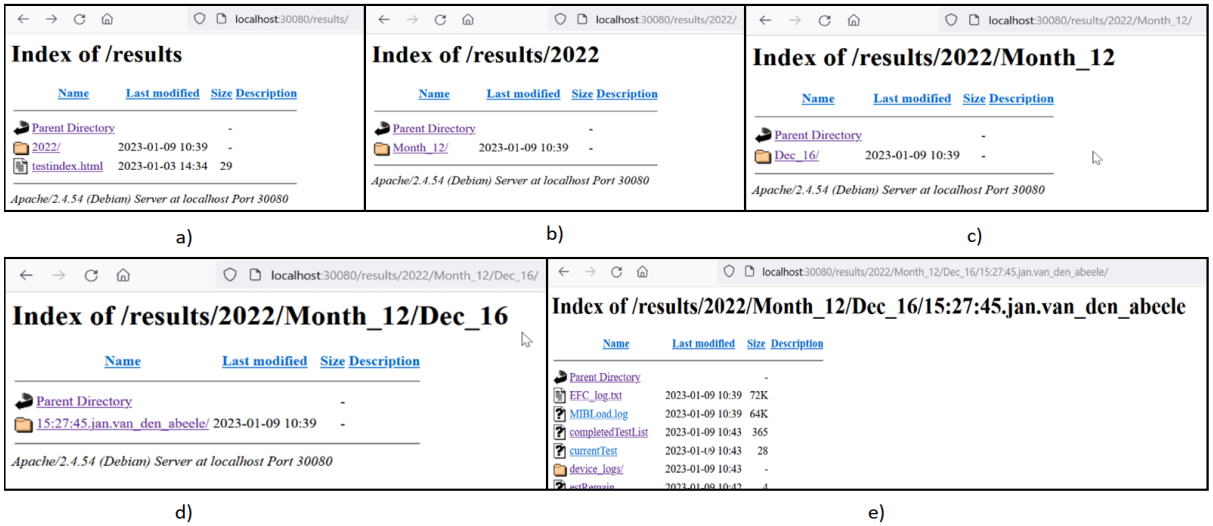
Nu de mappenstructuur is voorzien op de centrale storage en de reverse proxyserver is ingesteld om de testrun te forwarden naar de lokale HTTP-server van de testrun, kan een gebruiker browsen naar de gestarte testrun via het IP-adres (of in de toekomst DNS-naam) van de Proxy-central zoals zichtbaar in Figuur 47.

Het dient opgemerkt te worden dat de URL ‘localhost’ bevat als DNS-naam. Dit komt door het feit dat via een SSH-tunnel de NodePort van de reverse proxyserver van de Development machine werd geforward naar de lokale machine met een webbrowser.

Verder is het ook belangrijk om op te merken dat op het moment van testen, het volledige pad inclusief de naam van de tester nog een statische parameter was in één van de configuratiefiles binnen gash. Elke testrun had op dat moment dezelfde mappenstructuur van december 2022, terwijl de testrun effectief is uitgevoerd op 9 januari 2023.

Daarenboven komt ook het feit dat er geen duidelijke HTML-pagina getoond wordt zoals in Figuur 18. Dit komt echter door het feit dat in de gash pod er een bepaald Perl-script nog niet gestart was, wat verantwoordelijk is voor het genereren van de PHP-bestanden. Als gevolg dat ook het aanleveren van de logfiles iets minder aantrekkelijk was op het moment van testen, zoals zichtbaar in Figuur 47e.

Zodra echter het Perl-script dat de PHP-bestanden aanlevert (buiten de scope van deze masterproef) gestart wordt, is de cluster hier op voorbereid waardoor Figuur 47e er gelijkaardig zou uitzien als Figuur 18.



Figuur 47: Webpagina's die de mappenstructuur van een testrun tonen waarbij in de uiterste map de logfiles van één testrun worden getoond in de Kubernetes-cluster

Indien eindgebruikers rechtstreeks zonder een pad toe te voegen aan de URL op de reverse proxyserver connecteren, komt deze gebruiker op de pagina zoals in Figuur 47a. Van hieruit kan een gebruiker browsen door de aangeleverde mappenstructuur van de centrale HTTP-server. Zowel de mappenstructuur van afgelopen testruns als van draaiende testruns zijn hierop aanwezig. De effectieve resultaten (zoals mogelijks in Figuur 47e) binnen de mappenstructuur van een afgelopen testrun zijn ook centraal aanwezig. Daarentegen zijn de effectieve resultaten (zoals in Figuur 47e) binnen de mappenstructuur van een draaiende testrun enkel aanwezig op de lokale HTTP-server van de node die de testrun draait. Desalniettemin heeft een eindgebruiker het gevoel dat die op eenzelfde HTTP-server aan het browsen is, door het gebruik van de reverse proxyserver.

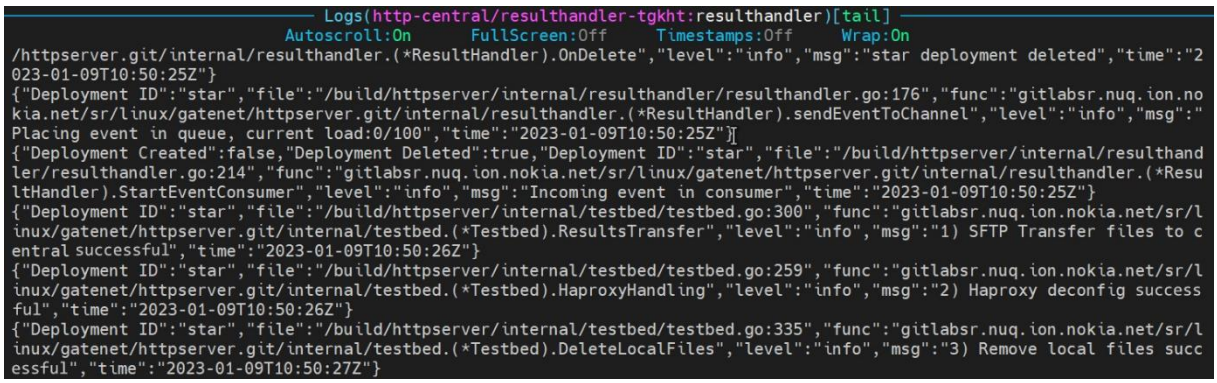
Figuur 48 toont de logs van HAProxy tijdens het browsen tussen de mappenstructuur richting een draaiende testrun. Het ‘public’ trefwoord is de naam van de front-end waarop elke HTTP-aanvraag binnenkomt. Na het trefwoord van de front-end komt de back-end met zijn naam, gevolgd door de server die de aanvraag naar deze back-end groep heeft behandeld. Als voorlaatste parameter per log, wordt telkens het pad getoond dat in de HTTP-aanvraag zit. Zo kan geconcludeerd worden dat in dit geval elke aanvraag van de gebruiker tot en met de map /results/2022/Month_12/Dec_16/ wordt behandeld door de centrale HTTP-server. Zodra een gebruiker in de map van de effectieve testrun zit, wordt de aanvraag gerouteerd naar de lokale HTTP-server van die testrun zoals zichtbaar in de voorlaatste log van Figuur 48.

```
Logs(http-central/proxy-central-66bc5fb5b9-pppdh:proxy-central)[tail]
Autoscroll:On  FullScreen:Off  Timestamps:Off  Wrap:On
<134>Jan  9 10:46:05 haproxy[296]: 10.244.2.1:44085 [09/Jan/2023:10:46:05.764] public central/s1 0/0/0/2/2 200 669 - - ---- 1/1/0/0/0 0/0 "GET /results/ HTTP/1.1"
<134>Jan  9 10:46:05 haproxy[296]: 10.244.2.1:44085 [09/Jan/2023:10:46:05.872] public central/s1 0/0/0/0/0 200 440 - - ---- 1/1/0/0/0 0/0 "GET /icons/folder.gif HTTP/1.1"
<134>Jan  9 10:46:38 haproxy[296]: 10.244.2.1:25695 [09/Jan/2023:10:46:38.880] public central/s1 0/0/0/2/2 200 641 - - ---- 1/1/0/0/0 0/0 "GET /results/2022/ HTTP/1.1"
<134>Jan  9 10:46:39 haproxy[296]: 10.244.2.1:25695 [09/Jan/2023:10:46:39.016] public central/s1 0/0/0/0/0 404 434 - - ---- 1/1/0/0/0 0/0 "GET /favicon.ico HTTP/1.1"
<134>Jan  9 10:46:56 haproxy[296]: 10.244.2.1:63838 [09/Jan/2023:10:46:56.341] public central/s1 0/0/0/2/2 200 646 - - ---- 1/1/0/0/0 0/0 "GET /results/2022/Month_12/ HTTP/1.1"
<134>Jan  9 10:46:56 haproxy[296]: 10.244.2.1:63838 [09/Jan/2023:10:46:56.448] public central/s1 0/0/0/0/0 404 434 - - ---- 1/1/0/0/0 0/0 "GET /favicon.ico HTTP/1.1"
<134>Jan  9 10:47:14 haproxy[296]: 10.244.2.1:8596 [09/Jan/2023:10:47:14.387] public central/s1 0/0/1/1/2 200 669 - - ---- 1/1/0/0/0 0/0 "GET /results/2022/Month_12/Dec_16/ HTTP/1.1"
<134>Jan  9 10:47:14 haproxy[296]: 10.244.2.1:8596 [09/Jan/2023:10:47:14.485] public central/s1 0/0/0/0/0 404 434 - - ---- 1/1/0/0/0 0/0 "GET /favicon.ico HTTP/1.1"
<134>Jan  9 10:47:34 haproxy[296]: 10.244.2.1:34781 [09/Jan/2023:10:47:34.564] public star/Server1 0/0/1/1/2 200 1014 - - ---- 1/1/0/0/0 0/0 "GET /results/2022/Month_12/Dec_16/15:27:45.jan.van_den_abeele/ HTTP/1.1"
<134>Jan  9 10:47:34 haproxy[296]: 10.244.2.1:34781 [09/Jan/2023:10:47:34.669] public star/Server1 0/0/0/1/1 200 460 - - ---- 1/1/0/0/0 0/0 "GET /icons/unknown.gif HTTP/1.1"
```

Figuur 48: Logs van HAProxy bij het connecteren naar een testrun die nog op een lokale node aan het draaien is in de Kubernetes-cluster

4.4.3 Na de afloop van een testrun

Zodra een testrun verwijderd wordt komt er een trigger binnen op de Resulthandlers. De bevoegde resulthandler gaat vervolgens de afhandeling van de logfiles op zich nemen en de reverse proxyserver configureren zodanig dat HTTP-aanvragen voor de testrun worden gerouteerd naar de centrale HTTP-server. Figuur 49 toont de logfiles van de resulthandler nadat een deployment verwijderd is. De JSON-structuur kom ook hier terug alsook de drie stappen zoals zichtbaar in Figuur 38. Ook hier zijn alle acties succesvol verlopen.



Figuur 49: Logs van de resulthandler die de nabehandeling van de omgeving verzorgt om logfiles centraal aan te bieden binnen de Kubernetes-cluster

Vanaf dat de resulthandler de testrun volledig heeft afgehandeld is de testrun centraal beschikbaar. Echter blijft de eindgebruiker exact dezelfde weergave hebben als in Figuur 47 met dezelfde URL met als enige verschil dat de HTTP-aanvraag nu is afgehandeld door een andere HTTP-server zoals zichtbaar in de logs van de HAProxy in Figuur 50.

Figuur 50: Logs van HAProxy bij het connecteren naar een testrun in de Kubernetes-cluster die centraal is opgeslagen

5 Conclusie

Binnen de huidige masterproef werd het design alsmede de implementatie van een schaalbaar systeem ontwikkeld dat logfiles – gegenereerd door testruns op de software van service-routers van Nokia – aanlevert aan eindgebruikers op een gebruiksvriendelijke manier.

Als eerste werd een Literatuurstudie gedaan naar het concept en de opkomst van containerisatie dat wordt toegepast binnen deze masterproef, vanwege de efficiëntere en meer schaalbare aanpak dan virtuele machines. Daarenboven werd ook het concept van orkestrators onderzocht, dewelke waken over de bredere scoop van een volledige cluster met betrekking tot het voorzien van de schaalbaarheid en de communicatie tussen de microservices. Kubernetes werd door Nokia gekozen als orkestrator door zijn performantie, functionaliteiten en het vrij en gratis gebruik van de software.
In het tweede deel van de literatuurstudie werd onderzoek gedaan naar de beste keuzes voor software die op voorhand noodzakelijk werden geacht voor het aanleveren van logfiles in de Kubernetes-cluster. Voor de webservers werd geconcludeerd dat Nginx de beste keuze is o.b.v. de performantie, populariteit alsook containerisatie-mogelijkheden. Daarentegen is Apache uiteindelijk effectief geïmplementeerd door de reeds aanwezige configuratiefiles voor de omgeving van deze webserver. Voor de keuze van een load balancer of reverse proxyserver werd HAProxy als beste naar voor gebracht omwille van de performantie uitgedrukt in het aantal aanvragen per seconde en de minste vertraging. Daarenboven zijn er ook voldoende containerisatie-mogelijkheden ter beschikking. Tot slot werden vier mogelijke storage oplossingen vergeleken als zijnde een puur theoretische benadering. HDFS, Ceph en GlusterFS kwamen als beste oplossingen naar boven, waarbij HDFS focust op kleinere clusters met een betere performantie bij kleinere files, Ceph op grotere betrouwbare clusters met veel functionaliteiten maar iets minder snel, en tot slot GlusterFS dewelke de belangrijkste voordelen van beide systemen combineert.

Hierna werd onderzoek gedaan naar het bepalen van de dimensie van de logfiles, uitgedrukt in de grootte en het aantal files van een testrun. A.d.h.v. Python-scripts werden automatisch 37 testbedden als een steekproef uit acht categorieën onderzocht. Hieruit bleek dat er gemiddeld 816 bestanden per testrun aanwezig zijn met een gemiddelde grootte van 178 MB, waarbij de mediaan 460 bestanden per testrun met een omvang van 82 MB bedraagt. Daarenboven bedraagt de minimumwaarde van de maxima van de grootte van een testrun per categorie al 4,77283 GB, maar procentueel gezien is slechts 0,11% van alle testruns groter dan deze 4,77283 GB. Door de beperkte maar aanwezige uitschieters dient er naar de toekomst toe rekening gehouden te worden met de te voorziene opslag voor logfiles enerzijds, en de netwerkcapaciteiten voor het transfereren van deze logfiles anderzijds. Desalniettemin worden deze files voor transfer gecomprimeerd, waardoor de werkelijke grootte die over het netwerk gaat gedeeltelijk wordt gereduceerd.

Binnen het laatste gedeelte van deze masterproef werd een tweeledig basisdesign ontworpen en geïmplementeerd om logfiles aan eindgebruikers aan te leveren d.m.v. Apache HTTP-servers met PHP-integratie. Enerzijds a.d.h.v. een vluchtige lokale HTTP-server die per testbed de lopende testrun aanlevert, anderzijds a.d.h.v. een persistente centrale HTTP-server die alle afgelopen testruns aanlevert. HAProxy doet dienst als reverse proxyserver en enig contactpunt om gebruikers buiten de cluster door te verwijzen naar de correcte interne HTTP-server op basis van het pad van de testrun in de URL van de HTTP-aanvraag. De in Go geschreven resulthandler die per node in Kubernetes aanwezig is, is op zijn beurt verantwoordelijk voor de dynamische configuratie van HAProxy en de transfer van lokale logfiles naar de centrale storage. De persistente SFTP- en SSHD-server zijn de instanties die gelinkt zijn aan deze centrale storage, die de communicatie met de resulthandler afhandelen. Door het voorziene design zijn grotendeels dezelfde functionaliteiten beschikbaar dan de bestaande implementatie, inclusief de voordelen van een gedistribueerd systeem. Bovendien kunnen alle logfiles vanop één centrale locatie geraadpleegd worden zonder een verschil te merken waar ze fysiek zijn gesitueerd.

Deze masterproef heeft in het algemeen bijgedragen aan het bekomen van statistische gegevens over logfiles per categorie testbedden. Verder heeft deze masterproef ook bijgedragen aan de modernisering van het testlabo binnen Nokia. Hierbij is de solide basis gevormd voor het aanleveren van logfiles binnen de Kubernetes-cluster waardoor in de toekomst ongecompliceerd extra bouwstenen kunnen worden toegevoegd. Het hoofdstuk Future work geeft een meer diepgaande blik over enkele verbeterpunten en of bouwstenen die naar de toekomst kunnen verbeterd of geïmplementeerd worden.

6 Future work

De implementatie voor het aanleveren van logfiles binnen deze masterproef werd als een werkend geheel binnen de digital sandbox omgeving geïmplementeerd. Echter zijn er nog enkele verbetermogelijkheden die naar de toekomst toe interessant zijn om mee te integreren. Dit hoofdstuk geeft in de mate van belangrijkheid de mogelijke toekomstige verbeterpunten of bijkomende implementaties met een bijhorende verklaring.

Opstarten van het Perl-script op de gash pod

Het effectief laten draaien van testruns binnen Kubernetes was een takenpakket dat simultaan met, maar buiten de scope van deze masterproef liep. Dit zorgde ervoor dat een eerste testrun pas tegen het einde van de periode deze masterproef kon uitgevoerd worden, waardoor 'live' resultaten ook pas vanaf het einde in realtime getest konden worden. Op dat moment werd ontdekt dat binnen gash er een Perl-script draait dat verantwoordelijk is voor het genereren van de index PHP-pagina zoals in Figuur 18. In de handmatige testen die voorafgaand werden uitgevoerd, werden steeds de PHP-files aangeleverd aan de HTTP-server zodat de doelstellingen van deze masterproef konden bereikt worden, namelijk het aanleveren van de logfiles aan de eindgebruikers. Door de onverwachte wending van het aparte script dat de PHP-files moet genereren, werd er besloten om voorlopig enkel de onbewerkte logfiles aan te bieden a.d.h.v. de HTTP-server zoals ook zichtbaar in Figuur 47e. Daarenboven kon het script ook niet klakkeloos gestart worden door mogelijke incompatibiliteitsproblemen met de cluster. Een goede werking voor het aanleveren van logfiles indien de PHP-bestanden aanwezig zijn is reeds bewezen doorheen voorafgaande testen. Echter dient er in de toekomst nog een manier gevonden te worden om het Perl-script binnenin de cluster van elke gash pod op te starten, zodanig dat eindgebruikers de vertrouwde weergave zoals in Figuur 18 kunnen raadplegen voor elke testrun.

Voorzien van automatisch herstel na de crash van een resulthandler

De resulthandler is een complex Go-programma dat reeds voorzien is van een retry-mechanisme. Desalniettemin bestaat de kans dat het programma al dan niet crasht na vijf mislukte pogingen van een bepaalde stap. Kubernetes zorgt voor de opstart van een nieuwe pod zonder te weten waar de vorige pod is gebleven met uitvoeren. Daarom dient er een mechanisme voorzien te worden dat bij de opstart van de resulthandler er een check gebeurt met welke testruns er aanwezig zijn op de huidige node, alsook waar de gecrashte resulthandler is gebleven. Voor het checken van de aanwezige testruns op de node kan gebruik gemaakt worden van de files die elke HTTP-server-init container heeft aangemaakt in de /passthrough/resulthandler map. Daarenboven kan ook gebruik gemaakt worden van de Golang client-API om Kubernetes te bevragen i.v.m. welke testruns er nog aan het draaien zijn. Om te kijken waar de gecrashte resulthandler is gebleven kan bv. gebruik gemaakt worden van een SQLite database die op het lokale volume van de node wordt bijgehouden als een file. Hierin kunnen gegevens staan van elke testrun op de huidige node, met de status van elke stap – zoals zichtbaar in Figuur 38 – opgeslagen in de database. Vooraleer de resulthandler start dient dan kort even de SQLite database geraadpleegd te worden. Er is reeds een implementatie voorzien die booleans bijhoudt van de doorlopen stappen voor elke aanwezige testrun doorheen het Go-programma. Het schrijven naar en lezen van de database dient nog geïmplementeerd te worden, samen met het checken van de aanwezige testruns bij de start van de resulthandler.

Voorzien van persistente HAProxy configuratiefiles

HAProxy heeft doorheen deze masterproef nog geen persistent volume gekregen om zijn configuratiefiles in op te slaan. Bij elke API-call die binnenkomt past HAProxy zijn configuratiefile aan, dewelke is opgeslagen in /usr/local/etc/haproxy/haproxy.cfg. Als de proxy-central pod zou crashen, start

deze terug van de standaardconfiguratie zoals meegegeven in de Docker image. Echter is het noodzakelijk dat vanaf het ogenblik dat de cluster in productie gaat, de configuratie terug was zoals voordien. Gelukkig kan dit op een simpele manier voorzien worden door een persistent volume claim voor de proxy-central pod aan te vragen waarin de haproxy.cfg file wordt opgeslagen. Zodra de pod terug zou opstarten is exact dezelfde configuratie van toepassing.

Implementeren van LDAP-authenticatie op de (centrale) HTTP-server

Omdat de bestaande implementatie authenticatie voorziet op elk van de webservers van de testbedden, dient ook in de toekomstige Kubernetes-cluster diezelfde authenticatie voorzien te worden. Hierop is verder nog geen onderzoek uitgevoerd, maar er zijn echter mogelijkheden om authenticatie in HAProxy in te bouwen. Daarbij is het van belang om na te gaan of dit mogelijk is d.m.v. LDAP, aangezien dit binnen Nokia gebruikt wordt als centraal authenticatieprotocol. Daartegenover kan er in het slechtste geval ook authenticatie voorzien worden op al dan niet elke HTTP-server in de cluster, of enkel op de centrale HTTP-server.

Vervangen van wachtwoorden naar SSH-keys

Binnen de cluster wordt gecommuniceerd a.d.h.v. het SSH-protocol tussen bv. de Resulthandler en de SSHD-central alsook tussen de Resulthandler en de SFTP-central. Deze communicaties worden op dit moment geauthentiseerd d.m.v. username en wachtwoord. In de toekomst zou er geopteerd moeten worden om gebruik te maken van cryptografische SSH-keys die de veiligheid en uniformiteit binnen de cluster bevorderen.

Implementeren van extra schaalbaarheid voor de centrale HTTP-server

De reverse proxyserver vangt in eerste instantie alle aanvragen op voor testruns die (nog) niet behoren tot de centrale HTTP-server. Maar vanaf het ogenblik dat de centrale HTTP-server te veel aanvragen te verwerken krijgt, dient deze gemultipliceerd te worden. Ook deze implementatie zou niet al te veel om handen mogen hebben door de gekozen architectuur doorheen deze masterproef. Kubernetes kan een HorizontalPodAutoscaler inschakelen op een bestaande deployment, in dit geval de HTTP-central. Deze autoscaler heeft de mogelijkheid om extra pods op te starten vanaf het ogenblik dat bv. het gemiddeld CPU-gebruik boven een bepaalde grens komt. Aangezien de extra pods dezelfde service ClusterIp gebruiken, zal Kubernetes automatisch aanvragen naar de HTTP-central – komende vanuit de proxyserver – forwarden naar één van de beschikbare pods. Daartegenover is er ook een iets complexere methode mogelijk waarbij dynamisch in HAProxy extra servers worden toegekend aan de back-end central, waarbij het load balancen wordt overgedragen aan HAProxy zelf. Hierbij dient gebruik gemaakt te worden van de DNS-naam van de pod i.p.v. die van de service.

Vervangen van NodePort voor de reverse proxyserver naar Ingress Controller

Doordat de NodePort slechts een tijdelijke oplossing was voor de Testopstelling, kan deze nog vervangen worden naar een ingress-controller binnen Kubernetes indien gewenst. Echter kan ook de NodePort behouden blijven als er rekening gehouden wordt met het feit dat de service op die manier beschikbaar blijft via een minder gebruikelijk poortnummer tussen de 30 000 en 32 767.

Voorzien van een DNS-naam in combinatie met de nodige certificaten

Om de secure versie van het HTTP-protocol te kunnen gebruiken, dient er in de toekomst een DNS-naam toegekend te worden aan de reverse proxyserver. Hierdoor kan al dan niet een geverifieerd

certificaat worden toegekend aan de DNS-naam om vervolgens HTTPS-connecties mogelijk te maken. Ook hier komt het voordeel van de reverse proxyserver naar boven door het feit dat er enkel een HTTPS-verbinding moet voorzien worden tussen de eindgebruiker en de reverse proxyserver.

Overwegen van een FTPS-server i.p.v. een SFTP-server

Een SFTP-server zoals gebruikt in deze masterproef baseert zich op het SSH-protocol. Echter werd SSH oorspronkelijk nooit ontwikkeld om bestanden te transfereren waardoor dit eerder een extra feature was. FTP daarentegen is wel een protocol dat speciaal ontwikkeld is om bestanden over het netwerk door te sluizen. Omdat SSH reeds gebruikt werd in de resulthandler en omwille van de tijdsbeperkingen, werd SFTP gekozen om de bestanden van een testrun te transfereren. Indien doorheen de verdere ontwikkelingen van het testlabo een significant tragere transfer wordt opgemerkt dan men zou verwachten, dan kan geopteerd worden om FTPS als protocol te gebruiken i.p.v. SFTP. Volgens [80] en [81] zijn er enkele kleine verschillen merkbaar waarbij FTPS net iets performanter is dan de SFTP-variant.

Implementeren van een lokale zoekmachine

Tot slot zou het voor eindgebruikers ook een enorme meerwaarde bieden om op de centrale server een soort van zoekmachine te hebben waarmee logfiles van afgelopen testruns kunnen gezocht worden o.b.v. veelgebruikte parameters. Hierdoor kan een eindgebruiker vaak veel sneller en efficiënter tussen de duizenden testruns gaan filteren. De zoekmachine is een zeer gegeerd feature door eindgebruikers, maar hiernaar dient nog heel wat onderzoek gedaan te worden in de toekomst vooraleer dit zorgvuldig kan geïmplementeerd worden.

Bibliografie

[1] Nokia, ‘Our history’. https://www.nokia.com/about-us/company/our-history/ (geraadpleegd okt. 13, 2022).

[2] Nokia, ‘Nokia Bell Labs Antwerp’. https://www.bell-labs.com/about/locations/antwerp-belgium/ (geraadpleegd okt. 13, 2022).

[3] Nokia, ‘IP networks’. https://www.nokia.com/networks/ip-networks/ (geraadpleegd okt. 13, 2022).

[4] J. Sweeney, ‘Virtualization’, in *Encyclopedia of Cloud Computing*, Chichester, UK: John Wiley & Sons, Ltd, 2016, pp. 89–101. doi: 10.1002/9781118821930.ch8.

[5] Stijn Schildermans en Kris Aerts, *Technische Concepten Cloud Computing [cursus]*. Diepenbeek, 2020.

[6] D. A. Menascé, ‘Virtualization: Concepts, Applications, and Performance Modeling’, 2005. Geraadpleegd: sep. 29, 2022. [Online]. Available: www.cs.gmu.edu/faculty/menasce.html

[7] Sean Campbell en Michael Jeronimo, ‘An introduction to virtualization’, 2006.

[8] C. Pahl en B. Lee, ‘Containers and Clusters for Edge Cloud Architectures -- A Technology Review’, in *2015 3rd International Conference on Future Internet of Things and Cloud*, aug. 2015, pp. 379–386. doi: 10.1109/FiCloud.2015.35.

[9] S. M. Jain, ‘Virtualization Basics’, in *Linux Containers and Virtualization*, Berkeley, CA: Apress, 2020, pp. 1–14. doi: 10.1007/978-1-4842-6283-2_1.

[10] J. S. J. Rajasingh en J. R. Wesley, ‘Step into the Cloud or Stop with Virtualization – The Project Manager’s Dialectic Dilemma’, *Procedia Comput Sci*, vol. 172, pp. 1077–1083, jan. 2020, doi: 10.1016/j.procs.2020.05.157.

[11] Enterprise Management Associates, ‘Reducing Operational Expense (OpEx) with Virtualization and Virtual Systems Management’, nov. 2009. Geraadpleegd: nov. 09, 2022. [Online]. Available: https://www.vmware.com/files/pdf/vmware-solution-opex-reducing-opex-wp-en.pdf

[12] J. Sahoo, S. Mohapatra, en R. Lath, ‘Virtualization: A survey on concepts, taxonomy and associated security issues’, *2nd International Conference on Computer and Network Technology, ICCNT 2010*, pp. 222–226, 2010, doi: 10.1109/ICCNT.2010.49.

[13] R. Dua, A. R. Raja, en D. Kakadia, ‘Virtualization vs containerization to support PaaS’, *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, pp. 610–614, sep. 2014, doi: 10.1109/IC2E.2014.41.

[14] C. Pahl en B. Lee, ‘Containers and clusters for edge cloud architectures-A technology review’, *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*, pp. 379–386, okt. 2015, doi: 10.1109/FICLOUD.2015.35.

[15] A. Bhardwaj en C. R. Krishna, ‘Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey’, *Arab J Sci Eng*, vol. 46, nr. 9, pp. 8585–8601, sep. 2021, doi: 10.1007/S13369-021-05553-3/TABLES/7.

[16] B. Basyildiz, ‘A Brief History of Container Technology’, aug. 19, 2019. https://www.section.io/engineering-education/history-of-container-technology/ (geraadpleegd nov. 09, 2022).

[17] I. Mavridis en H. Karatza, ‘Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing’, *Future Generation Computer Systems*, vol. 94, pp. 674–696, mei 2019, doi: 10.1016/j.future.2018.12.035.

[18] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, en R. E. Grant, ‘Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms’, in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, nov. 2019, pp. 11–20. doi: 10.1109/CANOPIE-HPC49598.2019.00007.

[19] J. Shah en D. Dubaria, ‘Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform’, in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, jan. 2019, pp. 0184–0189. doi: 10.1109/CCWC.2019.8666479.

[20] E. Casalicchio, ‘Container Orchestration: A Survey’, *EAI/Springer Innovations in Communication and Computing*, pp. 221–235, 2019, doi: 10.1007/978-3-319-92378-9_14/FIGURES/4.

[21] Y. Mao, Y. Fu, S. Gu, S. Vhaduri, L. Cheng, en Q. Liu, ‘Resource Management Schemes for Cloud-Native Platforms with Computing Containers of Docker and Kubernetes’, okt. 2020, Geraadpleegd: nov. 10, 2022. [Online]. Available: http://arxiv.org/abs/2010.10350

[22] A. Modak, S. D. Chaudhary, P. S. Paygude, en S. R. Ldate, ‘Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes?’, in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, apr. 2018, pp. 7–12. doi: 10.1109/ICICCT.2018.8473104.

[23] I. M. al Jawarneh *e.a.*, ‘Container Orchestration Engines: A Thorough Functional and Performance Comparison’, in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, mei 2019, vol. 2019-May, pp. 1–6. doi: 10.1109/ICC.2019.8762053.

[24] A. Grillet, ‘Hot Topics in ISE-Topic 5 Comparison of Container Schedulers’, Berlin.

[25] M. J. Santhosh Kumar, ‘A Comprehensive Comparison Study on Container Orchestration Frameworks’, *International Journal of Research in Engineering and Science (IJRES)*, vol. 10, nr. 5, pp. 72–77, 2022, Geraadpleegd: nov. 10, 2022. [Online]. Available: www.ijres.org

[26] The Kubernetes Authors, ‘Kubernetes Documentation’. https://kubernetes.io/docs/home/ (geraadpleegd okt. 05, 2022).

[27] A. Jansen, ‘Connecting every home with converged fixed and wireless broadband access’, jun. 25, 2021. https://www.nokia.com/blog/connecting-every-home-with-converged-fixed-and-wireless-broadband-access/ (geraadpleegd okt. 08, 2022).

[28] Cisco Systems Inc., ‘Cisco ASR 9000 Series Aggregation Services Router Broadband Network Gateway Configuration Guide, Release 5.2.x’, 2014. Geraadpleegd: okt. 07, 2022. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k_r5-2/bng/configuration/guide/b-bng-cg52xasr9k.pdf

[29] Nokia, ‘Multi-access broadband network gateway (BNG)’. https://www.nokia.com/networks/ip-networks/multi-access-broadband-network-gateway/ (geraadpleegd okt. 07, 2022).

[30] T. Anschutz, D. Allan, D. Thorne, S. Ooghe, en M. Hanrahan, ‘Migration to Ethernet-Based Broadband Aggregation’, jul. 2011. Geraadpleegd: okt. 08, 2022. [Online]. Available: https://www.broadband-forum.org/download/TR-101_Issue-2.pdf

[31] G. Fabregas, D. Allan, en H. Li, ‘Hybrid Access Broadband Network Architecture’, jul. 2016. Geraadpleegd: okt. 08, 2022. [Online]. Available: https://www.broadband-forum.org/download/TR-348.pdf

[32] K. Wan en D. Sinicrope, ‘Disaggregated BNG’, okt. 2019. Geraadpleegd: okt. 08, 2022. [Online]. Available: https://www.broadband-forum.org/marketing/download/MR-459.pdf

[33] S. Wadhwa, ‘Disaggregating the Broadband Network Gateway’, okt. 20, 2019. https://www.nokia.com/blog/disaggregating-broadband-network-gateway/ (geraadpleegd okt. 08, 2022).

[34] M. Wang, J. Chen, en R. Gu, ‘Information Model of Control-Plane and User-Plane separation BNG’, Chicago, mrt. 2017. Geraadpleegd: nov. 10, 2022. [Online]. Available: https://datatracker.ietf.org/meeting/98/materials/slides-98-i2rs-next-generation-bng-control-plane-data-plane-models-02

[35] Cisco Systems Inc., ‘Cloud Native BNG User Plane Configuration Guide for Cisco ASR 9000 Series Routers, IOS XR Release 7.4.x’, 2022. Geraadpleegd: nov. 10, 2022. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k-r7-4/cloud-native-bng/configuration/guide/b-cnbng-user-plane-cg-asr9000-74x.pdf

[36] W3Techs, ‘Usage statistics of web servers’, okt. 17, 2022. https://w3techs.com/technologies/overview/web_server (geraadpleegd okt. 17, 2022).

[37] W3Techs, ‘Historical yearly trends in the usage statistics of web servers’, okt. 17, 2022. https://w3techs.com/technologies/history_overview/web_server/ms/y (geraadpleegd okt. 17, 2022).

[38] Netcraft, ‘September 2022 Web Server Survey’, sep. 22, 2022. https://news.netcraft.com/archives/2022/09/22/september-2022-web-server-survey.html (geraadpleegd okt. 17, 2022).

[39] W. Rash, ‘The Ultimate Web Servers List: 11 Popular Web Servers To Use Today’, sep. 03, 2021. https://www.linode.com/docs/guides/web-servers-list/ (geraadpleegd okt. 17, 2022).

[40] J. Kiarie, ‘The 8 Best Open Source Web Servers’, jul. 29, 2020. https://www.tecmint.com/best-open-source-web-servers/ (geraadpleegd okt. 17, 2022).

[41] A. Isaiah, ‘Comparing the best web servers: Caddy, Apache, and Nginx’, okt. 05, 2021. https://blog.logrocket.com/comparing-best-web-servers-caddy-apache-nginx/ (geraadpleegd okt. 18, 2022).

[42] G. Liu, J. Xu, C. Wang, en J. Zhang, ‘A Performance Comparison of HTTP Servers in a 10G/40G Network’, in *Proceedings of the 2018 International Conference on Big Data and Computing*, apr. 2018, pp. 115–118. doi: 10.1145/3220199.3220216.

[43] D. Kunda, S. Chihana, en S. Muwanei, ‘Web Server Performance of Apache and Nginx: A Systematic Literature Review’, *Computer Engineering and Intelligent Systems*, vol. 8, pp. 43–52, okt. 2017.

[44] P. Prakash, R. Biju, en M. Kamath, ‘Performance analysis of process driven and event driven web servers’, *Proceedings of 2015 IEEE 9th International Conference on Intelligent Systems and Control, ISCO 2015*, sep. 2015, doi: 10.1109/ISCO.2015.7282230.

[45] W. M. C. J. T. Kithulwatta, K. P. N. Jayasena, B. T. G. S. Kumara, en R. M. K. T. Rathnayaka, ‘Performance Evaluation of Docker-based Apache and Nginx Web Server’, in *2022 3rd*

International Conference for Emerging Technology (INCET), mei 2022, pp. 1–6. doi: 10.1109/INCET54531.2022.9824303.

[46] V. N. Nguyen, 'Comparative Performance Evaluation of Web Servers', *VNU Journal of Science: Comp. Science & Com. Eng*, vol. 31, nr. 3, pp. 28–34, 2017.

[47] Q. Fan en Q. Wang, 'Performance Comparison of Web Servers with Different Architectures: A Case Study Using High Concurrency Workload', in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, nov. 2015, pp. 37–42. doi: 10.1109/HotWeb.2015.11.

[48] OpenLiteSpeed, 'Benchmarks'. https://openlitespeed.org/benchmarks/ (geraadpleegd okt. 18, 2022).

[49] O. Mustafa, A. Waqar, en M. Hussain, 'Comparative Run-Time Analysis of Web Servers on The Basis of Resource Deviation Finding the best performing web servers for web 2.0 applications View project', 2019, Geraadpleegd: okt. 19, 2022. [Online]. Available: https://www.researchgate.net/publication/341778909

[50] M. Tramper, 'LiteSpeed Vs Nginx (Benchmarks, WordPress & OpenLiteSpeed)', mrt. 02, 2022. https://makeitwork.press/litespeed-vs-nginx-benchmark-wordpress/ (geraadpleegd okt. 19, 2022).

[51] Johnny, 'NGINX vs OpenLiteSpeed (OLS) honest speed comparison 2022', okt. 16, 2021. https://wpjohnny.com/nginx-vs-openlitespeed-speed-comparison/ (geraadpleegd okt. 18, 2022).

[52] Nginx, 'Benefits of Layer 7 Load Balancing'. https://www.nginx.com/resources/glossary/layer-7-load-balancing/ (geraadpleegd nov. 07, 2022).

[53] N. Ramirez, 'Layer 4 and Layer 7 Proxy Mode', nov. 13, 2020. https://www.haproxy.com/blog/layer-4-and-layer-7-proxy-mode/ (geraadpleegd nov. 07, 2022).

[54] Nginx, 'Nginx as HTTP load balancer'. https://nginx.org/en/docs/http/load_balancing.html (geraadpleegd nov. 07, 2022).

[55] HAProxy, 'HAProxy'. https://www.haproxy.org/ (geraadpleegd nov. 07, 2022).

[56] Traefik, 'Traefik Proxy'. https://doc.traefik.io/traefik/ (geraadpleegd nov. 07, 2022).

[57] Envoy, 'Envoy Proxy'. https://www.envoyproxy.io/ (geraadpleegd nov. 07, 2022).

[58] G. Dillon, 'Benchmarking 5 Popular Load Balancers: Nginx, HAProxy, Envoy, Traefik, and ALB', dec. 09, 2018. https://www.loggly.com/blog/benchmarking-5-popular-load-balancers-nginx-haproxy-envoy-traefik-and-alb/ (geraadpleegd nov. 07, 2022).

[59] N. Ramirez M, F. Lavoie, en D. Corbett, 'NickMRamirez/Proxy-Benchmarks: Benchmarks for several proxies', feb. 21, 2021. https://github.com/NickMRamirez/Proxy-Benchmarks (geraadpleegd nov. 07, 2022).

[60] Traefik, 'Benchmarks - Træfik'. https://doc.traefik.io/traefik/v1.4/benchmarks/ (geraadpleegd nov. 07, 2022).

[61] A. Johansson, J. Zaxmy, en T. Fischer, 'HTTP Load Balancing Performance Evaluation of HAProxy, NGINX, Traefik and Envoy with the Round-Robin Algorithm', Skövde, 2022. Geraadpleegd: nov. 07, 2022. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:his:diva-21475

[62] Docker, ‘Docker Hub Container Image Library’. https://hub.docker.com/ (geraadpleegd nov. 08, 2022).

[63] F. Wu en G. Sun, ‘Software-Defined Storage’, dec. 2013. Geraadpleegd: okt. 24, 2022. [Online]. Available: http://fgwu.me/publications/techrepo/WS2013.pdf

[64] ‘HDFS Architecture Guide’. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html (geraadpleegd okt. 30, 2022).

[65] L. Acquaviva, P. Bellavista, A. Corradi, L. Foschini, L. Gioia, en P. C. M. Picone, ‘Cloud Distributed File Systems: A Benchmark of HDFS, Ceph, GlusterFS, and XtremeFS’, *2018 IEEE Global Communications Conference, GLOBECOM 2018 - Proceedings*, 2018, doi: 10.1109/GLOCOM.2018.8647218.

[66] ‘Ceph.io’. https://ceph.io/en/discover/ (geraadpleegd okt. 30, 2022).

[67] ‘Gluster Docs’. https://docs.gluster.org/en/latest/ (geraadpleegd okt. 30, 2022).

[68] ‘Swift - OpenStack’. https://wiki.openstack.org/wiki/Swift (geraadpleegd okt. 30, 2022).

[69] J. Arnold en SwiftStack team, *OpenStack Swift*. Sebastopol: O’Reilly Media, Inc., 2014. Geraadpleegd: okt. 31, 2022. [Online]. Available: https://cdn.codefine.site:5443/wp-content/uploads/2014/11/OpenStack_Swift_Using_Administering_and_Developing_for_Swift_Object_Storage.pdf

[70] Seaweedfs, ‘GitHub - seaweedfs/seaweedfs’. https://github.com/seaweedfs/seaweedfs (geraadpleegd jan. 19, 2023).

[71] G. Kang, D. Kong, L. Wang, en J. Zhan, ‘OStoreBench: Benchmarking Distributed Object Storage Systems Using Real-World Application Scenarios’, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12614 LNCS, pp. 90–105, 2021, doi: 10.1007/978-3-030-71058-3_6/FIGURES/8.

[72] B. Depardon, C. Séguin, en G. le Mahec, ‘Analysis of Six Distributed File Systems’, feb. 2013. Geraadpleegd: nov. 01, 2022. [Online]. Available: https://hal.inria.fr/hal-00789086/document

[73] Q. M. Nguyen, T. M. Doan, en T. B. Dinh, ‘A Scalable - High Performance Lightweight Distributed File System’, *Proceedings - 2020 7th NAFOSTED Conference on Information and Computer Science, NICS 2020*, pp. 66–71, nov. 2020, doi: 10.1109/NICS51282.2020.9335887.

[74] C. T. Yang, W. H. Lien, Y. C. Shen, en F. Y. Leu, ‘Implementation of a Software-Defined Storage Service with Heterogeneous Storage Technologies’, *Proceedings - IEEE 29th International Conference on Advanced Information Networking and Applications Workshops, WAINA 2015*, pp. 102–107, apr. 2015, doi: 10.1109/WAINA.2015.50.

[75] J. Y. Lee, M. H. Kim, S. A. R. Shah, S. U. Ahn, H. Yoon, en S. Y. Noh, ‘Performance Evaluations of Distributed File Systems for Scientific Big Data in FUSE Environment’, *Electronics 2021, Vol. 10, Page 1471*, vol. 10, nr. 12, p. 1471, jun. 2021, doi: 10.3390/ELECTRONICS10121471.

[76] Gartner, ‘Gartner Magic Quadrant Research Methodology’. https://www.gartner.com/en/research/methodologies/magic-quadrants-research (geraadpleegd nov. 02, 2022).

[77] B. Henderson, ‘Leader Recognition in Distributed File Systems and Object Storage’, okt. 08, 2021. https://www.dell.com/en-us/blog/leader-recognition-in-distributed-file-systems-and-object-storage/ (geraadpleegd nov. 01, 2022).

[78] Gartner, ‘Gartner Distributed File Systems and Object Storage Reviews 2022’. https://www.gartner.com/reviews/market/distributed-file-systems-and-object-storage (geraadpleegd nov. 02, 2022).

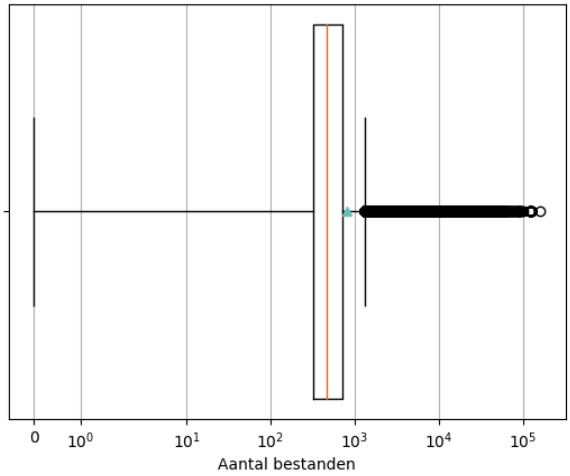
[79] The Kubernetes Authors, ‘kind’. https://kind.sigs.k8s.io/ (geraadpleegd dec. 28, 2022).

[80] U. Gupta, ‘Survey on Security Issues in File Management in Cloud Computing Environment’, *Int J Comput Appl*, vol. 120, nr. 5, pp. 22–24, jun. 2015, doi: 10.5120/21224-3948.

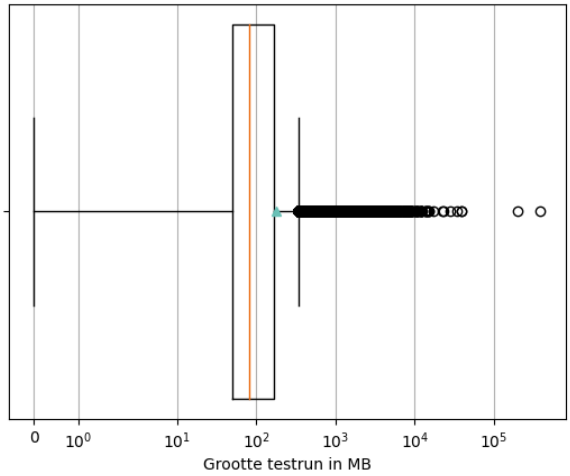
[81] H. Kath, ‘Which is Faster: FTPS or SFTP?’, sep. 16, 2019. https://www.goanywhere.com/blog/which-is-faster-ftps-or-sftp (geraadpleegd jan. 12, 2023).

Bijlagen

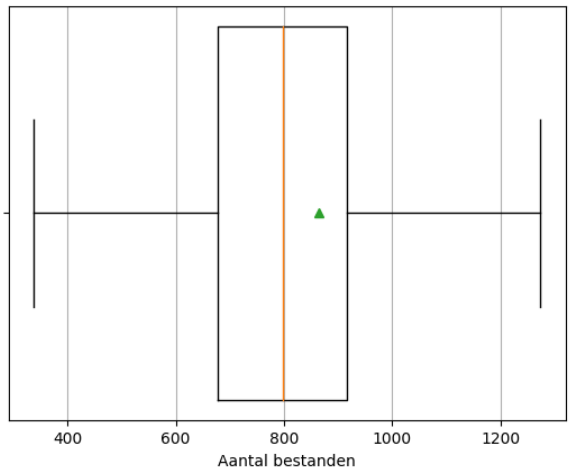
Bijlage A: Overige boxplots dimensioneren logfiles



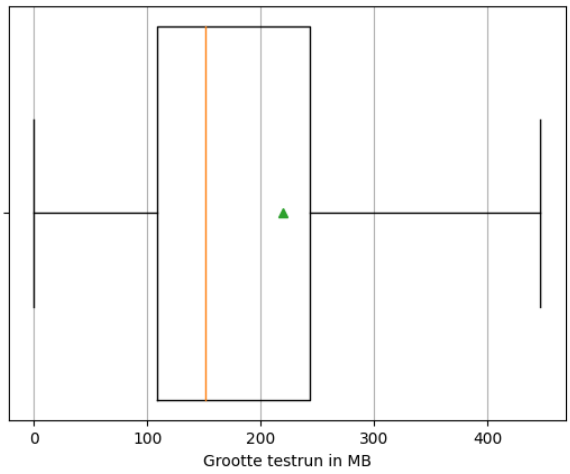
Boxplot van het totaal aantal bestanden per testrun
van de volledige steekproef, incl. uitschieters



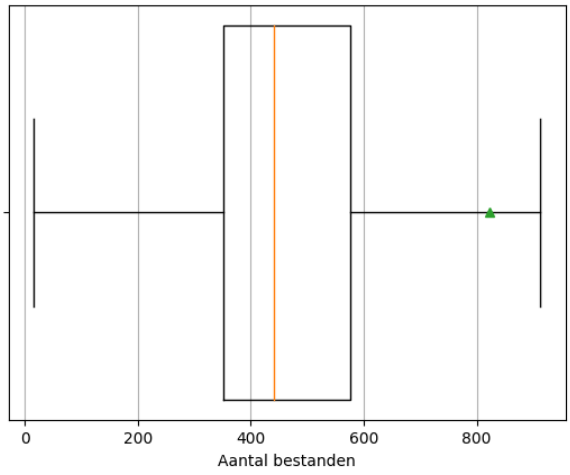
Boxplot van de grootte per testrun
van de volledige steekproef, incl. uitschieters



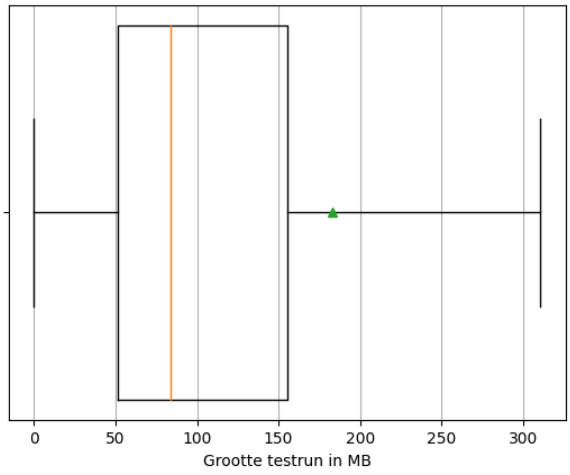
Boxplot van het aantal bestanden per testrun
van categorie ang3nxrtb, excl. uitschieters



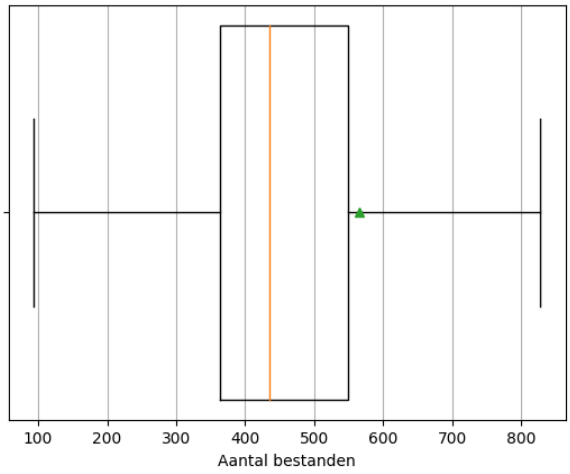
Boxplot van de grootte per testrun
van categorie ang3nxrtb, excl. uitschieters



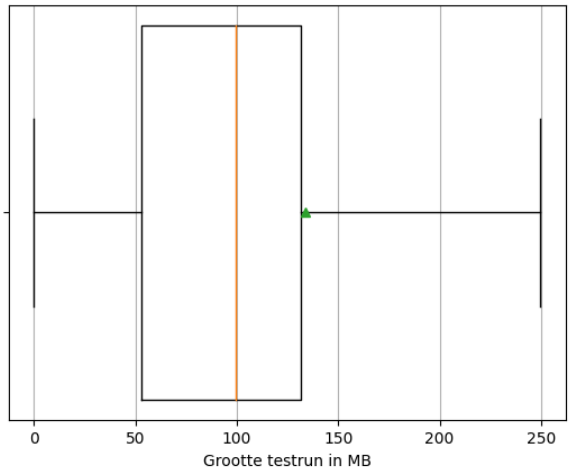
Boxplot van het aantal bestanden per testrun
van categorie ang4rtb, excl. uitschieters



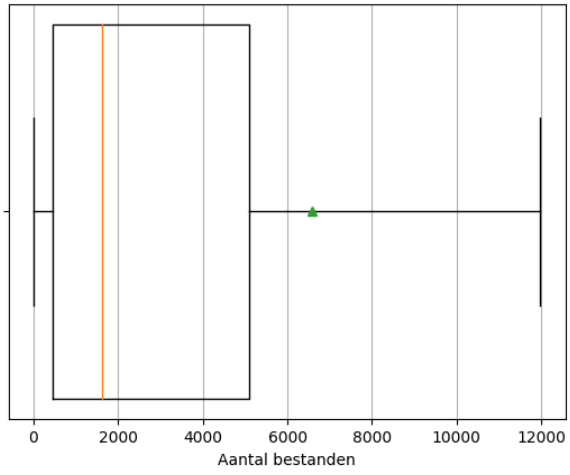
Boxplot van de grootte per testrun
van categorie ang4rtb, excl. uitschieters



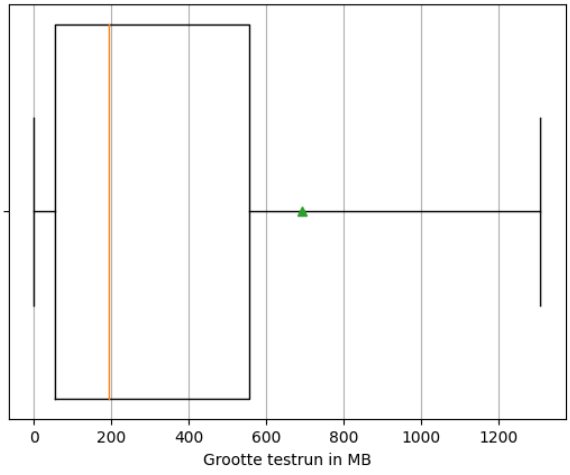
Boxplot van het aantal bestanden per testrun
van categorie ang5nxrtb, excl. uitschieters



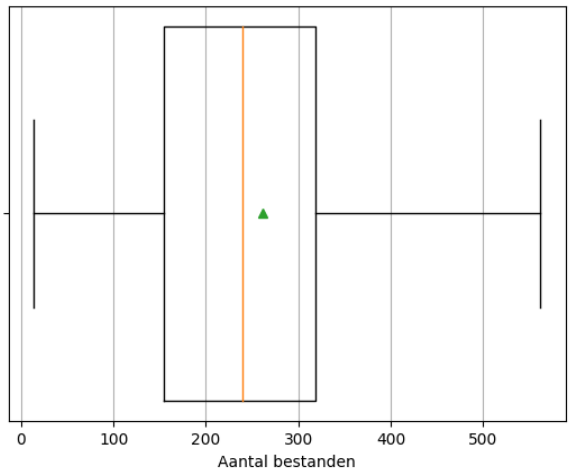
Boxplot van de grootte per testrun
van categorie ang5nxrtb, excl. uitschieters



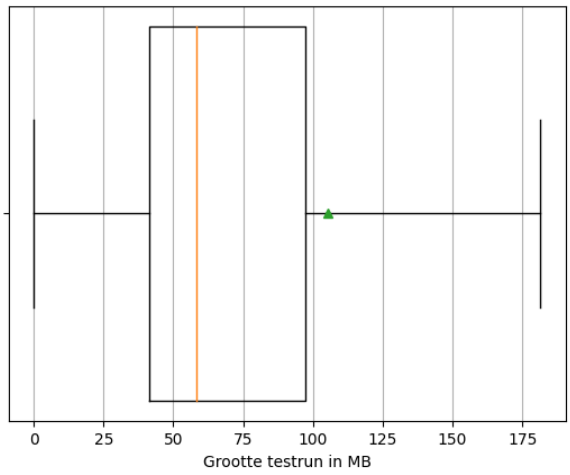
Boxplot van het aantal bestanden per testrun
van categorie ang6rtb, excl. uitschieters



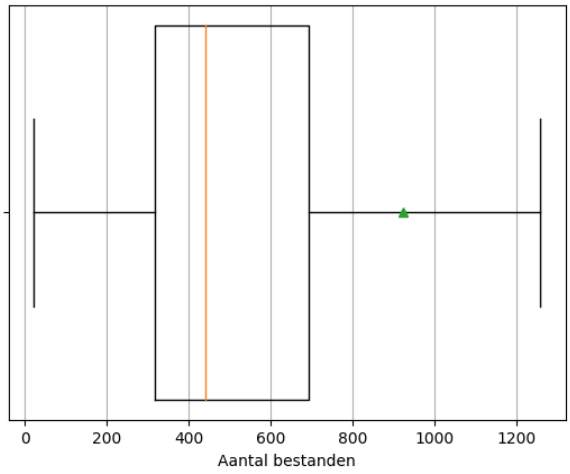
Boxplot van de grootte per testrun
van categorie ang6rtb, excl. uitschieters



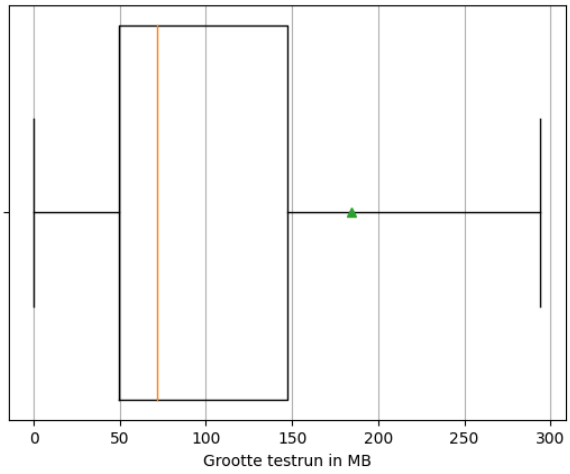
Boxplot van het aantal bestanden per testrun
van categorie anhw, excl. uitschieters



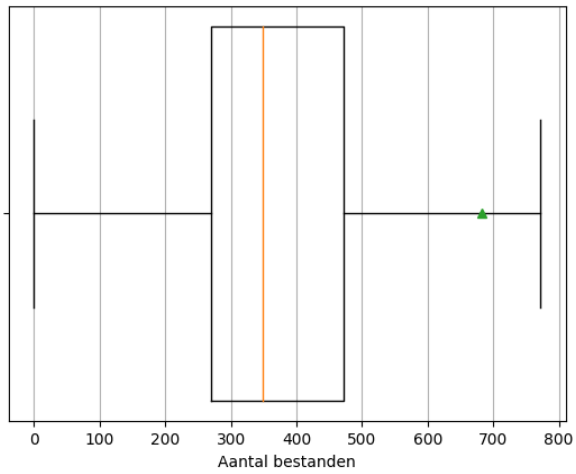
Boxplot van de grootte per testrun
van categorie anhw, excl. uitschieters



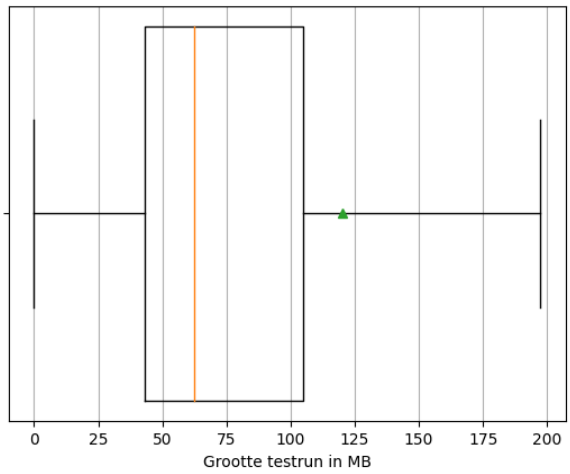
Boxplot van het aantal bestanden per testrun
van categorie anrtb, excl. uitschieters



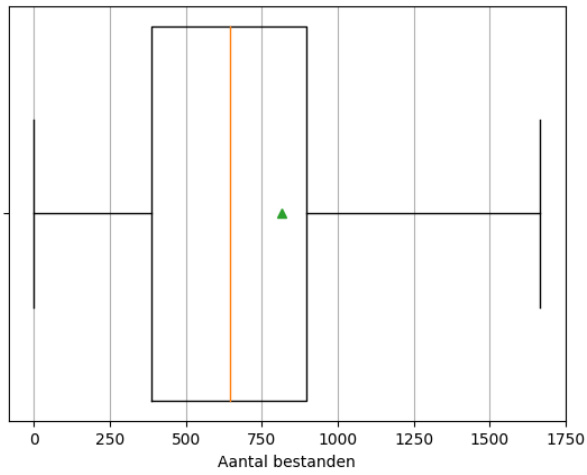
Boxplot van de grootte per testrun
van categorie anrtb, excl. uitschieters



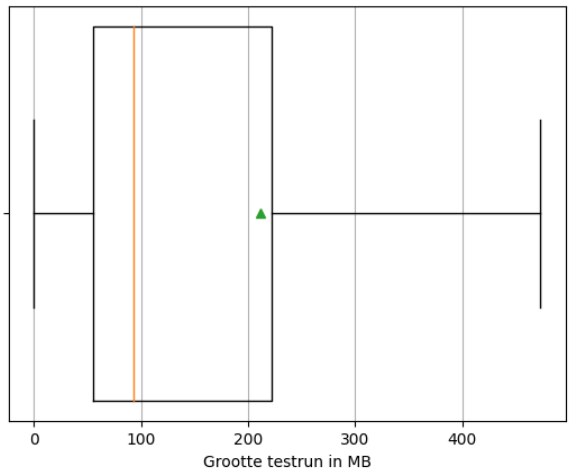
Boxplot van het aantal bestanden per testrun
van categorie anrtb2vm, excl. uitschieters



Boxplot van de grootte per testrun
van categorie anrtb2vm, excl. uitschieters



Boxplot van het aantal bestanden per testrun
van categorie anrtbvm, excl. uitschieters



Boxplot van de grootte per testrun
van categorie anrtbvm, excl. uitschieters