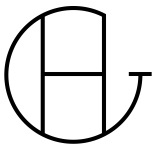

HoGent

Faculteit Bedrijf en Organisatie

Use Cases voor Unikernels

Michiel De Wilde

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

Promotor:
Antonia Pierreux
Co-promotor:
Philippe De Wilde

Instelling: Hogeschool Gent

Academiejaar: 2016-2017

Eerste examenperiode

Faculty of Business and Information Management

Use Cases voor Unikernels

Michiel De Wilde

Thesis submitted in partial fulfillment of the requirements for the degree of professional bachelor of applied computer science

Promotor:
Antonia Pierreux
Co-promotor:
Philippe De Wilde

Institution: Hogeschool Gent

Academic year: 2016-2017

First examination period

Samenvatting

Het efficiënt gebruiken van de middelen die ter beschikking worden gesteld is een belangrijk gegeven binnen informatica. Doorheen de tijd zijn er innovaties gekomen om dit steeds naar een hoger niveau te tillen. Eerst was er sprake van virtuele machines en dan kwamen software containers ter sprake.

Unikernels is een opkomende manier voor het compileren en uitvoeren van applicaties in een productieomgeving. Unikernels vraagt om een andere kijk op programma's in de productieomgeving en hoe er gewerkt wordt met de architectuur en infrastructuur. Deze bachelorproef focust op de use cases voor unikernels. Er wordt nader gekeken naar de mogelijkheden, moeilijkheden en huidige implementaties. Ook de vergelijking tussen software containers en unikernels wordt gemaakt.

Het resultaat van dit onderzoek is dat unikernels kunnen gebruikt worden in één specifieke use case: netwerkapplicaties. Er kan ook nog overwogen worden om unikernels te gebruiken wanneer veiligheid van uitermate belang is.

De belangrijkste conclusie is dat het ecosysteem rond unikernels niet ver genoeg gevorderd is om het te gebruiken in productie en om productiviteit voor de ontwikkelaars te garanderen. Debuggen van unikernels is een groot probleem en dit zou interessant kunnen zijn om te kijken hoe dit zich verder ontwikkeld.

Voorwoord

Deze bachelorproef is tot stand gekomen uit eigen interesse. Computers hebben mij sinds mijn jeugd al gefascineerd. Het heeft een tijd geduurd vooraleer ik begon met programmeren. Ik begon met een simpele applicatie maar al snel werd het meer complex. Het beheren van servers en infrastructuur was een uitdaging die ik zeker aanging. De traditionele servers maakten plaats voor containers. Een paar maanden voor mijn bachelorproef, kwam ik het concept van unikernels tegen. Dit leek mij een goed onderwerp voor mijn bachelorproef en was ook al als onderwerp beschikbaar gesteld. Het was een mooie uitdaging om meer te leren over concepten rond systeembeheer en software ontwikkeling.

Ik wens mevrouw Antonia Pierreux te bedanken voor de begeleiding.

Verder bedank ik de werknemers van mijn stageplaats, Wercker BV. In het bijzonder Toon Verbeek, Micha Hernández van Leuffen en Benno van den Berg. De bachelorproef zou er niet gekomen zonder hun hulp. En als laatste mijn co-promoter Philippe De Wilde.

Contents

1. Inleiding

Bij het opleveren van software worden applicaties eerst lokaal op de eigen machine ontwikkeld. Wanneer de software opgeleverd wordt dan wordt het geplaatst in een test- en/of productieomgeving. Het geschreven programma van de lokale ontwikkelomgeving naar de productieomgeving brengen kan veel moeite met zich mee brengen. De productieomgeving bestaat uit infrastructuur die zich lokaal in het bedrijf bevindt of bij een cloud provider (Amazon Web Services, Google Cloud) te vinden is.

De middelen van de productieomgeving moeten zo goed mogelijk benut worden. In deze bachelorproef zal bekeken worden of unikernels hierbij een rol kunnen spelen. En voor welke use cases unikernels gebruikt kunnen worden.

1.1 Stand van zaken

De voorbije jaren zijn er veel problemen gevonden bij algemene besturingssystemen. Deze besturingssystemen zijn niet alleen te vinden op computers van gebruikers, maar ook op de servers van applicaties. Beide omgevingen vragen om een andere oplossing. Unikernels (hoofdstuk 5) kunnen hierop inspelen omdat er uitgegaan wordt van een minimale basis waarbij er functionaliteit kan worden toegevoegd. Naast unikernels zijn er nog de klassieke oplossingen zoals virtuele machines (hoofdstuk 3) en software containers (hoofdstuk 4). Verder wordt ook de architectuur van applicaties bekeken (hoofdstuk 7) om de veranderingen op het valk van

de applicatie en de omgeving aan te duiden.

1.2 Probleemstelling en Onderzoeksvragen

Het doel van deze bachelorproef is om de use cases voor unikernels aan te duiden en de verhouding tegenover software containers en virtuele machines. Dit zal gebeuren door de volgende onderzoeksvragen te beantwoorden:

- Wat zijn de use cases voor unikernels?
- De verhouding van software containers tegenover unikernels?
- De gevolgen voor de applicatie architectuur wanneer unikernels in gebruik worden genomen?
- Wordt het opstellen en beheren van applicaties eenvoudiger of niet?
- Wat is de impact op beveiliging?

1.3 Opzet van deze bachelorproef

Deze bachelorproef behandelt het onderwerp unikernels (Mao and Humphrey, 2012). Hierbij wordt gekeken naar de veiligheid en efficiëntie van unikernels. Verder wordt er ook gekeken naar de gevolgen die unikernels kunnen hebben op de architectuur van applicaties. De omgeving waarin de ontwikkelaar werkt kan verschillen van project tot project. Daarom is het aantonen van de specifieke situaties waarin unikernels kunnen gebruikt worden van uiterst belang.

De bachelorproef is als volgt opgebouwd:

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

Virtualisatie vormt de basis voor veel van de technologieën die worden aangehaald. Virtualisatie zal belicht worden in hoofdstuk 3.

Op 21 maart 2013 werd op Pycon de eerste demo van Docker gegeven (Hykes, 2013). Het gegeven software containers bestond al langer, maar is pas echt doorgebroken door Docker. In hoofdstuk 4 worden software containers nader bekeken.

Hoofdstuk 5 dat unikernels behandelt, zal zich focussen op de werking, voordelen en implementaties van unikernels.

In hoofdstuk 6 zal gekeken worden naar de verschillende implementaties van unikernels en deze, tot zover dit mogelijk is, met elkaar vergelijken.

Hoofdstuk 7 geeft een beeld van de architectuur van applicaties door de concepten van microservices en monolieten nader te bekijken.

Hoofdstuk 8 zal het proof of concept toelichten. Hierbij wordt er een simpele web server op verschillende omgevingen uitgevoerd en gekeken wat de mogelijkheden en moeilijkheden zijn. De keuze van het soort applicatie wordt gegeven omwille van de beperkte mogelijkheden van unikernels.

In Hoofdstuk 9, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Methodologie

Het begrip unikernel vraagt om een uitgebreide theoretische kennis van besturingssystemen en virtualisatie-technologieën.

Om deze concepten goed te begrijpen werd er eerst een literatuurstudie uitgevoerd. Een belangrijk beginpunt was de volgende website (Unikernel Systems, 2016). De voorgenoemde website heeft een lijst van papers over unikernels en verwijzingen naar implementaties van unikernels. Veel over virtuele machines was te vinden in thesissen van de voorbije jaren van de opleiding toegepaste informatie. De kennis over software containers werd voornamelijk opgedaan tijdens mijn stage bij Wercker. Een resem boeken over software containers, met als centraal onderwerp Docker, gaven meer inzicht in software containers en hun use cases.

De literatuurstudie vormde de basis voor de eerste hoofdstukken over virtuele machines, containers en unikernels. De bachelorproef focust niet alleen op de mogelijkheden en moeilijkheden van unikernels. De gevolgen voor de architectuur van applicaties worden nader bekeken.

Het hoofdstuk na unikernels maakt een vergelijking tussen de implementaties van unikernels op het vlak van eenvoud, inzetbaarheid en behandelt de mogelijkheden van elke implementatie. Hiervoor werd er opzoekwerk verricht om de gegevens van de implementaties te vinden.

3. Virtualisatie

In dit hoofdstuk zal bekeken worden waarom en hoe virtualisatie ontstaan is. Verder zal bekeken worden welke concepten meespelen binnen virtualisatie. Dit hoofdstuk dient als een inleiding om concepten zoals software containers, unikernels en virtuele machines te kunnen begrijpen.

In de jaren 60 was de tijd dat een computer kon gebruikt worden beperkt (Conferences, 1968). Bij de eerste computers, had men problemen om programma's uit te werken. Dit lag vooral aan de tijd dat er kon gewerkt worden aan de computer en de weinige computers die beschikbaar waren. Een voorbeeld van het ontwikkelen van een programma in die tijd was het volgende: "De broncode van het programma werd ingegeven en in een wachtrij geplaatst. Pas een bepaalde tijd later konden de resultaten van het programma bekeken worden. Fouten in het geschreven programma zorgden voor veel tijdsverlies." (Conferences, 1968)

Eén van de grootste bijdrages, tot de ontwikkelingssnelheid van programma's, is de lengte van de feedbackcyclus: hoe snel kan een programma getest worden wanneer er een verandering in de broncode plaatsvindt. Als er lang moet worden gewacht op de feedback, dan kan dit tijdsverlies tot gevolg hebben.

Timesharing (Pyke Jr., 1967) werd uitgevonden om het verlies van tijd, tijdens het ontwikkelen van programma's, te beperken. Bij timesharing kunnen gebruikers inloggen op een console en zo met meerdere tegelijk van de computer gebruik maken. Dit was een technische uitdaging. Elke gebruiker en zijn programma's bevinden zich binnen een afzonderlijke context bij

timesharing. De computer moet van de ene context naar de andere kunnen veranderen. Timesharing zou de basis vormen voor het moderne besturingssysteem. Eén van de moeilijkheden van timesharing is de isolatie van de verschillende processen en gebruikers. De processen moeten zich bevinden binnen een verschillende context om geen invloed op elkaar te hebben.

Doorheen de tijd kregen computers meer middelen ter beschikking. De meeste programma's konden niet langer alle middelen van de computer benutten. Dit zorgde voor de creatie van virtuele middelen of virtual resources (Vakilinia, Ali, and Qiu, 2015). Om deze virtuele middelen te voorzien moeten bepaalde delen van de computer gevirtualiseerd worden. Dit kan voorkomen onder verschillende vormen zoals hardware virtualisatie en virtualisatie van het besturingssysteem.

Het concept van virtualisatie deelt een aantal gelijkenissen met timesharing. De computer wordt opgedeeld in verschillende delen bij virtualisatie. Bij timesharing wordt de computer opgedeeld in afzonderlijke contexten.

Een aantal voordelen van virtualisatie zijn de volgende: financieel voordeel (men kan van één taak naar meerdere taken gaan op één computer), het besparen van energie (Beloglazov and Buyya, 2010) en veiligheid(Mortleman, 2009).

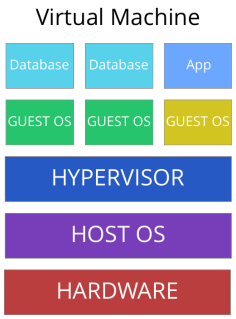

Figure 3.1: structuur van een virtuele machine

Een virtuele machine (Smith and Nair, 2005) bootst een computer na. Een virtuele machine kan de middelen van de fysieke computer, waar het zich op bevindt, gebruiken. Dit zorgt ervoor dat de middelen van de fysieke computer kunnen gebruikt worden als virtuele middelen. Doorheen deze bachelorproef zal naar de fysieke computer verwezen worden als de host machine. Guest is de naam dat wordt gegeven aan virtuele machines die zich bevinden op de host. In het volgende deel zullen we de laag tussen de host machine en virtuele machine bekijken: de hypervisor. Figuur 3.1 toont de structuur van een virtuele machine.

3.1 Hypervisor

Een hypervisor (Popek and Goldberg, 1974) is een voorbeeld van hardware virtualisatie. Het is een stuk software, firmware of hardware, dat de laag vormt tussen de virtuele machine en de host machine. De host machine zorgt voor de middelen zoals CPU, RAM, enzoverder. Elke virtuele machine die zich bevindt op de host machine zal dan gebruik kunnen maken van

een gedeelte van deze middelen. Doordat virtualisatie alomtegenwoordig geworden is in datacenters (Soundararajan and Anderson, 2010) heeft het ervoor gezorgd dat er meer logica komt te liggen bij de hypervisor. De hypervisor neemt verder de rol op zich van het verdelen van de middelen en het beheren van de guests. Er zijn twee soorten hypervisors: type 1 en type 2. Type 1 is de bare-metal hypervisor en type 2 de hosted hypervisor.

3.1.1 Hosted Hypervisors

Als eerste zullen we de hosted hypervisor of type 2 hypervisor behandelen. Figuur 3.2 toont de structuur van een hosted hypervisor. De hosted hypervisor bevindt zich op het besturingssysteem van de host machine en heeft geen directe toegang tot de hardware. De type 2 hypervisor is dus compleet afhankelijk van het besturingssysteem van de host om zijn taken uit te voeren. Als er problemen optreden in het besturingssysteem van de host heeft dit gevolgen voor de hypervisor en guests.

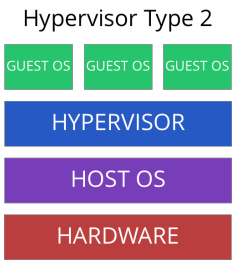

Figure 3.2: structuur van een hosted hypervisor

Voorbeelden van hosted hypervisors zijn: Oracle Virtualbox (Oracle, 2016a) en VMware Workstation (VMware, 2016).

3.1.2 Bare-metal Hypervisors

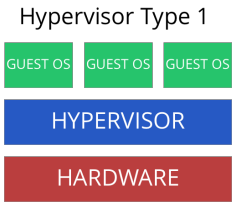

Figure 3.3: structuur van een bare-metal hypervisor

Type 1, bare-metal, embedded of native hypervisors bevinden zich rechtstreeks op de hardware. De voornaamste taak van de hypervisor is het beheren en verdelen van middelen. Dit maakt de hardware hypervisor kleiner in omvang dan de hosted hypervisor. De hosted hypervisor is niet afhankelijk van het besturingssysteem van de host machine want de hosted hypervisor bevindt zich rechtstreeks op de hardware. Figuur 3.3 toont de structuur van een bare-metal hypervisor.

Er is een laag minder in de structuur dus dit betekent dat er minder instructies moeten worden uitgevoerd bij een bepaalde handeling. Dit heeft een betere prestatie tot gevolg. Omdat er geen problemen kunnen zijn met het besturingssysteem van de host kan er aangenomen worden dat deze soort hypervisors stabieler zijn. Wanneer het besturingssysteem van de host faalt bij een hosted hypervisor dan zal dit gevolgen hebben voor de guests.

Een paar voorbeelden van bare-metal hypervisors zijn VMware ESXi en Xen.

3.2 Operating System-level Virtualisation

Naast hardware virtualisatie kan ook een besturingssysteem gevirtualiseerd worden. Bij deze toepassing van virtualisatie worden de mogelijkheden van

de kernel gebruikt. De kernel van bepaalde besturingssystemen laat toe om meerdere geïsoleerde namespaces tegelijkertijd te laten werken. Dit zorgt ervoor dat er maar één besturingssysteem moet zijn om verschillende programma's naast elkaar en geïsoleerd te laten werken.

Elke namespace heeft zijn eigen configuratie en zijn geïsoleerd van elkaar. Dit geeft eveneens de beperking dat de guests over een besturingssysteem of kernel moeten beschikken, die overeenkomt met de host.

Tegenover hardware virtualisatie zal besturingssysteem virtualisatie minder gebruik maken van middelen omdat er maar één besturingssysteem is. Dit is omdat het besturingssysteem wordt gedeeld. Dit geeft voordelen bij de prestatie.

Voorbeelden van besturingssysteem virtualisatie zijn: chroot (Linux, n.d.), Solaris Containers (Oracle, 2016b) en Docker (Docker, 2016).

Besturingssysteem virtualisatie is vooral bekend geworden door Docker vanaf 2013 (Hykes, 2013). In het volgende hoofdstuk wordt verder ingegaan op besturingssysteem virtualisatie onder de vorm van software containers.

4. Software Containers

4.1 Software Containers

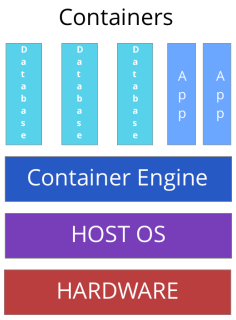

Figure 4.1: structuur van containers

Software containers bestaan al een ruime tijd. In Soltesz, Pötzl, Fiuczynski, Bavier, and Peterson, 2007 werd al gekeken naar de voordelen van software containers ten opzichte van hypervisors. Chroot ("Linux Containers - LXC - Introduction," n.d.) is een concept dat veel deelt met software containers. Chroot zal de root directory van het huidige proces en zijn children veranderen. Er wordt een virtuele kopie gemaakt van het systeem waarin het proces kan werken. Het proces is dus afgesloten van het systeem en dit zorgt voor meer veiligheid bij het uitvoeren van processen.

Linux Containers of LXC (Containers, n.d.) is een implementatie van besturingssysteem virtualisatie. LXC kan meerdere geïsoleerde Linux systemen laten werken op één host. Deze geïsoleerde Linux systemen worden software containers genoemd. Deze containers worden ook getoond in figuur 4.1 waarbij de apps en databases containers zijn.

Zoals al gezien is, is de rol van de hypervisor het delen en beheren van de middelen. Om containers te gebruiken, moet een ander deel de rol van de hypervisor overnemen. Cgroups is een Linux kernel extension die deze rol overneemt. Door middel van cgroups kunnen middelen worden beheerd voor processen. Het toont ook de mogelijkheden om backups te

creëren van processen.

Eerder werd ook al aangehaald dat de processen van elkaar gescheiden moeten worden. Dit wordt bereikt door namespaces. De functies die name spaces voorzien zijn uitgebreid. Elke namespace heeft zijn eigen bestandssysteem structuur, netwerk interfaces en proces ID space. De containers delen de kernel met alle andere processen die op de kernel aan het werken zijn.

De containers zijn van elkaar afgesloten. Wanneer één container aangetast wordt, dan heeft dit geen rechtstreeks gevolg op de andere containers.

Toch zijn er een paar andere problemen zoals aangehaald in Madhavapeddy et al., 2015. Processen die als root werken kunnen niet geïsoleerd worden van elkaar. Verder wordt er ook aangehaald dat strengere isolatie nodig is. (tabel 2, Madhavapeddy et al., 2015)

4.2 Docker

Docker (Docker, 2016) is een open source project om software containers te gebruiken voor programma's gemakkelijk te ontwikkelen en te laten werken op verschillende besturingssystemen.

Er zijn andere formaten zoals rkt CoreOS, n.d. die het ook mogelijk maken om gemakkelijk software containers te gebruiken.

In 2015 werd het Open Container Initiative opgericht. Veel grote spelers op vlak van software containers zoals Docker, CoreOS, Microsoft en Google maken hier deel van uit. Samen willen ze een standaard voor software containers vastleggen.

De problemen die we bij software containers tegenkomen hebben voornamelijk betrekking tot beveiliging. Unikernels kunnen een veiliger alternatief zijn. In het volgende hoofdstuk worden de werking en mogelijkheden met unikernels bekeken.

5. Unikernels

5.1 Inleiding

Dit hoofdstuk zal uitleggen uit welke delen een unikernel bestaat. Tevens wordt er ook beeld gegeven over de voordelen en eigenschappen van unikernels. In het volgende hoofdstuk wordt bekeken welke implementaties van unikernels er al bestaan en wat de verschillen tussen deze implementaties zijn.

Virtuele machines (hoofdstuk 3) zijn er gekomen wanneer men de middelen van computers beter wou gebruiken. Toch werden de middelen niet optimaal benut door de besturingssystemen die werden gebruikt. Bij containers (hoofdstuk 4) wordt het besturingssysteem van de host machine gedeeld tussen de verschillende software containers. De middelen worden efficiënter gebruikt, doordat er maar één besturingssysteem is.

Unikernels (Madhavapeddy et al., 2013) gaat een andere weg op. Bij unikernels wordt er gekeken naar het besturingssysteem. Traditionele besturingssystemen zoals Ubuntu worden onder de loep genomen. De meeste besturingssystemen die worden gebruikt binnen een productieomgeving, kunnen ook gebruikt worden voor algemene doeleinden.

De eerste implementaties van unikernels worden tegengekomen op het einde van de jaren 90. Exokernel (MIT, 1998) werd ontwikkeld door MIT. De software ontwikkelaars kunnen zelf dus keuzes maken op vlak van abstractie. Nemesis (University of Cambridge, 2000) werd vanuit de universiteit van Cambridge ontwikkeld. Deze onderzoekers wilden unikernels gebruiken

voor doeleinden binnen de multimedia.

Unikernels vragen inzicht in een aantal verschillende technologieën. De kernel ligt aan de basis van een besturingssysteem en zal in de volgende sectie worden uitgelegd.

5.2 Kernel

De kernel is het programma dat zich centraal bevindt in het besturingssysteem. Het werkt rechtstreeks met de hardware van de computer. De kernel kan gezien worden als het fundament waar het hele besturingssysteem op steunt. Omdat het een belangrijke rol vervult, in het besturingssysteem, is veel van het geheugen van de kernel beveiligd. Dit is bedoeld zodat bepaalde applicaties geen veranderingen kunnen aanbrengen in de kernel. Als er iets fout zou gaan met de kernel, dan heeft dit rechtstreekse gevolgen op het besturingssysteem. Al de handelingen die de kernel uitvoert bevinden zich in de kernel space. Daartegenover gebeurd alles wat de gebruiker uitvoert zich in de user space. Het is van uiterst belang dat de kernel space en user space strikt van elkaar gescheiden zijn. Als dit niet zo zou zijn, dan zou een besturingssysteem niet veilig zijn. De kernel voert nog andere taken uit zoals geheugenbeheer en system calls. Figuur 5.1 toont de positie van de kernel binnen het systeem.

Als een programma wordt uitgevoerd, dan bevindt het programma zich binnen de user space. Om het programma in werkelijkheid te kunnen uitvoeren, moet er om toestemming gevraagd worden aan de kernel. Deze instructies moeten worden nagegaan voor de veiligheid. Soms wordt er ook gesproken van memory isolation, waarbij de user space en kernel space niet rechtstreeks met elkaar kunnen communiceren. Dit is een veiligheidsmaatregel.

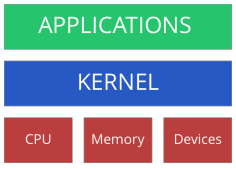

Figure 5.1: positie van kernel tussen de programma's en hardware

Volgend boek heeft meer informatie over de werking van de kernel (Bovet and Cesati, 2005).

De twee volgende delen zullen de belangrijkste eigenschappen van unikernels uitleggen.

5.3 Library besturingssysteem

Elke virtuele machine binnen de architectuur van een productieomgeving heeft meestal één functie. Dit is al getoond in figuur 3.1. Elke guest heeft een gespecialiseerde rol om de middelen, dat het ter beschikking krijgt, optimaal te benutten. Dezelfde architectuur en denkwijze kan teruggevonden worden bij containers. De besturingssystemen, die virtuele machines en containers gebruiken, kan traditioneel worden genoemd. Dit is aangehaald in Madhavapeddy et al., 2013.

Als er dieper wordt ingegaan op deze evolutie, dan kunnen we een patroon vaststellen: er worden steeds kleinere eenheden gebruikt. Eerst was de machine de eenheid en dan de virtuele machine. In het geval van software containers is de eenheid een container. Een unikernel kan ook bekeken worden als een eenheid binnen dit patroon.

Het meest gebruikte besturingssysteem voor servers is het Ubuntu besturingssysteem (Matthias Gelbmann, 2016) met 32%. Veel applicaties van databases tot en met web applicaties gebruiken het. Dit terwijl een database en een web applicatie andere middelen en functionaliteit nodig hebben. Er zijn er ook gespecialiseerde besturingssystemen zoals Mini-OS (Satya Popuri, n.d.). Mini-OS gebruikt veel van de overbodige functies van een traditioneel besturingssysteem niet. Deze gespecialiseerde besturingssystemen zijn in de minderheid en ook niet zeer gekend.

Traditionele besturingssystemen zijn niet de basis die men nodig heeft in een architectuur waar elke eenheid één rol heeft. Alpine (Alpine Linux Development Team, n.d.) is een Linux besturingssysteem dat een zeer minimale basis heeft. Tevens beschikt het over een uitgebreide package repository. Dit maakt het een ideaal besturingssysteem voor de basis van software containers. Er kan gestart worden met een kleine basis en alle onderdelen toevoegen die nodig zijn. Dit zorgt voor een container met een kleinere omvang. Hier wordt verder over uitgeweid in sectie 5.6.

Sommige onderdelen van het besturingssysteem hebben geen nut meer. Dit kan voor slechtere prestaties en lagere veiligheid zorgen.

Het concept van library besturingssystemen (Madhavapeddy et al., 2013) neemt dit nog een stap verder. Er wordt met een minimale basis gestart. Daarna worden alleen de componenten toegevoegd, die nodig zijn voor de functionaliteit die de eenheid uitvoert. Library duidt op de verschillende onderdelen of componenten die kunnen

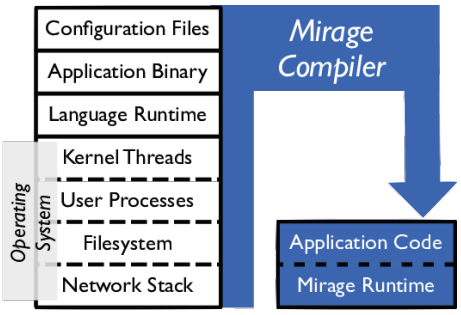

Figure 5.2: Samenstelling algemeen besturingssysteem tegenover unikernel Madhavapeddy et al., 2013

worden toegevoegd.

Web applicaties hebben verschillende functies nodig om te kunnen communiceren. Hiervoor bestaan netwerkprotocollen. TCP is een kritiek protocol om te communiceren met het internet en andere apparaten. Bij algemene besturingssystemen, zoals Ubuntu, is dit al aanwezig. Omdat er gestart wordt met een minimale basis, moeten deze protocollen geïmplementeerd worden. Gelukkig zijn er libraries die hiervoor kunnen gebruikt worden. De libraries, broncode en configuratie worden dan gecompileerd. Als resultaat heb je dan een image. Wanneer deze image wordt uitgevoerd dan hebben we een werkende unikernel. De unikernel wordt gecompileerd voor één omgeving en kan alleen gebruikt worden op deze omgeving. Dit is een verschil met software containers en virtuele machines die meer los staan van de omgeving waarin ze zich bevinden.

Unikernels kunnen enkel werken op de omgeving waarvoor ze gecompileerd zijn. Dit is omdat drivers voor de hardware componenten moeten worden geschreven. Kleine veranderingen in de specificatie of interface van een hardware component zorgt ervoor dat de driver niet meer werkt. Drivers zijn één van de grootste obstakels die worden vastgesteld bij library besturingssystemen. Hypervisors lossen dit probleem op door een standaard interface open te stellen. Zo moet er alleen maar één driver worden geschreven voor de hardware component. het werk wordt uitermate verminderd door dit. De protocollen, waar eerder over gesproken is, vormt dan nog een deel van het werk.

5.4 Single Address Space

In een unikernel bestaat er geen concept van user of kernel space. Alle processen bevinden zich binnen dezelfde space. Dit zou problemen geven bij traditionele besturingssystemen. Maar bij het compileren van de unikernel wordt de broncode, libraries en configuratie gecontroleerd. Dit is om te kijken of er zich geen problemen kunnen voordoen. De afwezigheid van de communicatie tussen de kernel en user spaces zorgt voor betere prestaties. Deze verbetering van de prestatie komt ook doordat de hardware kan worden aangesproken zonder dat er veranderd moet worden van context.

Eén globale address space zorgt voor problemen met de isolatie van processen. Meerdere programma's naast elkaar laten werken op een library besturingssysteem is complex. De hypervisor lost dit probleem gedeeltelijk op. Een mogelijke oplossing voor dit probleem wordt verder besproken in Hoofdstuk 7.

5.5 Veiligheid

De hoge mate van veiligheid van unikernels is een gevolg van de specialisatie van de eenheid. Bij virtuele machines en software containers zijn traditionele besturingssystemen de basis. Deze besturingssystemen hebben zeer veel functionaliteit die niet nodig is voor de specifieke taak dat ze uitvoeren. De overbodige functionaliteit kan zorgen voor een lagere prestatie, maar ook voor meer veiligheidsrisico's. Het is dus gemakkelijker om de veiligheid te garanderen van een kleinere broncode tegenover een grote. Daar komt nog bij dat er unikernel implementaties worden gebruikt die specifiek zijn voor de omgeving. Het vraagt veel meer moeite om een gespecialiseerde implementatie te schrijven tegenover een algemene implementatie. Het marktaandeel van de algemene implementatie zal ook veel groter zijn tegenover de specifieke implementatie.

Verder heeft een unikernel geen shell of een andere mogelijkheid om een unikernel aan te passen terwijl deze aan het werken is. Eén unikernel overnemen heeft geen gevolg op de andere unikernels. Daarbij komt nog dat de hypervisors zelf meer veiligheid garanderen (Colp et al., 2011).

5.6 Andere voordelen

De omvang van een unikernel is kleiner dan een virtuele machine of een container. Zoals er al eerder werd aangehaald kunnen containers ook een kleine omvang hebben wanneer ze een miniem besturingssysteem gebruiken. Een voorbeeld daarvan is Alpine dat start vanaf 5 MB ("gliderlabs/docker-alpine," n.d.). Een voorbeeld van de omvang van een unikernel kan gevonden worden in Madhavapeddy et al., 2015, hoofdstuk 4, p. 10 met 1 MB.

Het systeem optimaliseren kan ook in veel grotere mate gebeuren (Madhavapeddy et al., 2010) dan bij traditionele besturingssystemen.

5.7 Productie

Veel van de commentaar die gegeven wordt op unikernels komt van de moeilijkheden om te kijken wat er fout gaat in productie (Bryan Cantrill, 2016). Aanpassingen doen in productie om problemen te verhelpen is niet de beste manier om iets op te lossen. Het programma moet uitgebreid getest worden vooraleer het in de productieomgeving wordt opgesteld. Wanneer er dan toch iets fout gaat in productie, dan zal men de situatie proberen na te bootsen in een soortgelijke omgeving. Het terugzetten van

een oudere versie van het programma kan helpen om de gebruikers geen ongemak te laten ondervinden.

5.8 Hedendaags gebruik

Unikernels kunnen momenteel gebruikt worden in beperkte situaties. Er was hetzelfde fenomeen te vinden bij software containers een paar jaar geleden, vooraleer Docker op de voorgrond trad. Elke technologie moet een bepaalde tijd ondergaan om matuur te worden.

In het volgende deel zullen we implementaties van verschillende unikernels vergelijken om een beter beeld te krijgen van de huidige mogelijkheden en het huidige landschap.

6. Vergelijking implementaties unikernels

6.1 Inleiding

In het hoofdstuk over unikernels zijn er een aantal voorbeelden van moderne implementaties van unikernels aangehaald. In dit hoofdstuk zullen er een aantal implementaties van unikernels worden vergeleken met elkaar. Het volgende deel zal aanhalen welke criteria, er zullen gebruikt worden om deze vergelijking te realiseren.

6.2 Criteria

Implementatie programmeertaal Op het eerste zicht lijkt de programmeertaal waarin de implementatie geschreven niet belangrijk. Dit is wel belangrijk om te weten of het een programmeertaal is waar gemakkelijk kan mee gewerkt worden. Een meer voor de hand liggende programmeertaal, zoals C++, zal gemakkelijker open source ontwikkelaars aantrekken. En dat is ook een belangrijk gegeven. De meeste implementaties zijn in C/C++ geschreven. Dit komt vooral door het feit dat veel software ontwikkelaars die zich bezig houden met unikernels vanuit een achtergrond met besturingssystemen komen. De meest gebruikte programmeertaal binnen besturingssystemen is C/C++.

Hypervisors Hypervisors zijn een groot deel van de keuze van een implementatie. De meest gebruikte unikernels zijn beschikbaar op een aantal hypervisors. Als er veel omgevingen zijn, waarop de unikernels

kunnen werken, dan is het gemakkelijk om van omgeving te veranderen als bedrijf. Wendbaarheid en inspelen op veranderingen zal gemakkelijker gebeuren, wanneer er een uitgebreid aantal mogelijkheden zijn.

Ondersteunde programmeertalen Sommige unikernels ondersteunen een aantal programmeertalen. Programmeertalen hebben sterke en zwakke kanten en keuze hebben uit een aantal programmeertalen helpt voor een keuze te maken.

GitHub stars GitHub bepaalt de status van een open source project. Open source software wordt meer en meer gebruikt door bedrijven. De keuze is gemakkelijker te maken tussen een project met een hoge populariteit(sterren) en een project met lage populariteit. Hierbij wordt er wel gesproken over projecten met dezelfde functionaliteit. Door GitHub kan gekeken worden of er actief aan het project gewerkt wordt en/of het onderhouden wordt. De community achter een project is ook belangrijk voor de keuze te maken tussen verschillende projecten.

De inhoud van de bovenstaande tabel wordt per implementatie uitgelegd en is te vinden op het einde van dit hoofdstuk.

6.3 Implementaties van moderne unikernels

6.3.1 ClickOS

Implementatie programmeertaal : C/C++

Hypervisors : Xen

Ondersteunde programmeertaal : ondersteund door bindings

GitHub stars : 243

ClickOS (Martins et al., 2014) wordt ontwikkeld door Cloud Networking Performance Lab.

De toepassingen, waarvoor ClickOS voornamelijk wordt gebruikt, zijn middlebox applicaties. Een middlebox is een netwerk applicatie dat netwerktrafiek kan omzetten, filteren, inspecteren of manipuleren. Voorbeelden hiervan zijn firewalls en load balanceres. Een modulaire router vormt de basis van ClickOS. Op deze router kunnen onderdelen worden toegevoegd. Deze unikernel werkt alleen op MiniOS. MiniOS is beschikbaar bij de broncode van de Xen hypervisor.

Door een evolutie binnen de netwerk laag (García Villalba, Valdivieso Caraguay, Barona López, and López, 2015) wordt veel van de functionaliteit, die vroeger bij de hardware zat, nu in software geïmplementeerd. Dit laat toe om een eigen implementatie te schrijven en zo veel functionaliteit van de hardware over te nemen.

De use cases waarbinnen ClickOS kan gebruikt worden zijn beperkt. Als je geen gebruik wilt maken van ingebouwde netwerk functionaliteit van de hardware, dan is ClickOS de uitgesproken keuze.

Er wordt Swig gebruikt om ondersteuning te bieden voor hogere programmeertalen. Swig maakt een bindings die C/C++ verbindt met een hogere programmeertaal.

ClickOS verwijst naar zijn packages als elements. Die elements voeren een bepaalde actie uit. Dit zijn hele kleine stukken functionaliteit. Er zijn om en bij de 300 elementen beschikbaar. Het is eenvoudig om zelf een eigen element te maken en te distribueren.

Meer informatie kan gevonden worden volgende website: Cloud Networking Performance Lab, n.d.

6.3.2 HaLVM

Implementatie programmeertaal : Haskell

Hypervisors : Xen

Ondersteunde programmeertaal : Haskell

GitHub stars : 665

HaLVM (Galois Inc., n.d.) wordt ontwikkeld door Galios. Galios is een software ontwikkelingsagentschap dat unikernels al een tijd in productie gebruikt. Er zijn niet al te veel bedrijven die unikernels gebruiken in productie.

De programmeertaal waarin de unikernel van HaLVM wordt geschreven is Haskell. Haskell is een functionele programmeertaal met een uitgebreid type system. HaLVM is een implementatie die één supervisor en één programmeertaal ondersteunt.

Het werd ontwikkeld met als doel om componenten voor besturingssystemen snel te maken en te testen. Het is verder geëvolueerd naar andere use cases.

Bij HalVM wordt de Xen hypervisor als omgeving gebruikt. Er is een integratie met de Xen hypervisor waarvan de core library van HaLVM gebruikmaakt. Er bestaat ook een communicatie library, die bestaat uit het Haskell File System en de Haskell Network Stack. Deze library kan gebruikt worden in de meeste gevallen, als er netwerkfunctionaliteit nodig is. Als we meer mogelijkheden nodig hebben voor een programma dan kunnen er modules worden toegevoegd. Er is een ecosysteem uitgebouwd om het gemakkelijker te maken voor software ontwikkelaars om hun eigen modules te bouwen.

De werkwijze is de volgende: eerst wordt er zoveel mogelijk functionaliteit als een normaal Haskell programma geschreven. Daarna moet het programma aangepast worden om het te gebruiken op HaLVM. Dit is niet gemakkelijk bij uitgebreide applicaties, want er zijn maar beperkte debug mogelijkheden op HalVM.

Zoals in de meeste gevallen moet de compiler van Haskell worden aangepast om de unikernel te maken. Het is ook geen probleem om standaard Haskell libraries in de code te gebruiken.

Het wordt gebruikt door Galios in productie en dit maakt het gemakkelijk om vragen te stellen. De GitHub repository, waar de applicatie zich op bevindt, is over het algemeen actief en is populair voor maar één programmeertaal te ondersteunen.

6.3.3 Ling

Implementatie programmeertaal : C/Erlang

Hypervisors : Xen

Ondersteunde programmeertaal : Erlang

GitHub stars : 523

Ling (Erlang on Xen, n.d.) is een Erlang virtuele machine die werkt op de Xen hypervisor. Het bedrijf achter Ling is Cloudozer. Ze hebben al meerdere language runtimes gemaakt die rechtstreeks op Xen werken. Ling is open source maar de andere tools, onder meer het beheren van unikernels, zijn niet open source. Wanneer er problemen met het ecosysteem zijn, moet de ondersteuning van Cloudozer gecontacteerd worden.

Zoals bij HaLVM, moet de applicatie eerst worden geschreven in Erlang. De package manager die gebruikt word met Erlang is Rebar, dit is de standaard Erlang package manager. Na het omzetten van de applicatie naar een Xen afbeelding zou de unikernel moeten werken.

Railing is een tool die meegeleverd is met Ling die je toelaat om, erlang on Xen, afbeeldingen te maken. We gebruiken ook xl utility van Xen om domeinen te beheren. De focus van Erlang on Xen is de Xen hypervisor.

Bij het uitbrengen van een nieuwe versie van LING is het mogelijk geworden om andere omgevingen te ondersteunen. Dit heeft veel nieuwe omgevingen, zoals internet of things en mobiele omgevingen, mogelijk gemaakt. Unikernels kunnen handig zijn op deze omgevingen omwille van de kleine omvang.

Verder opent dit ook de mogelijkheid voor de unikernels van LING op bare-metal te laten werken.

6.3.4 Rumprun

Implementatie programmeertaal : C

Hypervisors : hardware, Xen, KVM

Ondersteunde programmeertaal : onder meer C, C++, Erlang, Go, Javascipt, Python, Ruby

GitHub stars : 469

Rumprun (Rumpkernel, n.d.) gebruikt rump kernels voor hun implementatie. Deze rump kernels worden samengesteld uit componenten afkomstig van NetBSD. NetBSD is een traditioneel besturingssysteem maar is modulair geschreven. Men kan het dus gebruiken om een rump kernel samen te stellen.

Er is een uitgebreide keuze aan hypervisors waaruit kan gekozen worden. De term hw duidt op hardware. Dit betekent dat rumprun één van de enige implementaties is die rechtstreeks kan werken op hardware. De unikernel kan ook werken op besturingssystemen die een POSIX-interface hebben. De POSIX-interface duidt op de meeste Unix systemen.

Er zijn verschillende soorten implementaties van unikernels. Sommige unikernels specialiseren op basis van programmeertaal en andere op basis van omgeving. Sommige doen zelf beide. Rumprun behoort tot de laatste groep. Dit is wel niet zonder gevolg. De prestatie zal niet die van een gespecialiseerde unikernel kunnen evenaren.

De rump-run packages zijn implementaties van drivers, protocollen en libraries die kunnen toegevoegd worden aan de rumprun kernels. Er zijn een groot aantal packages die kunnen gebruikt wordenen de meest bekende zijn aanwezig. Het spijtige is wel dat er nog geen packaging systeem aanwezig is. Dit zou er wel voor zorgen dat er gewerkt kan worden met verschillende dependencies en versies van packages.

Rumprun verziet zelf geen compiler. Er wordt gebruik gemaakt van een compiler die aanwezig is op het systeem. In het geval van Mac OS X moet er een aparte compiler geïnstalleerd worden.

De programmeertalen die ondersteund zijn, zijn de volgende: C, C++, Erlang, Go, Javascript(node.js), Python, Ruby en Rust.

Meer informatie is te vinden in volgende thesis: Kantee, 2012.

6.3.5 MirageOS

Implementatie programmeertaal : OCaml

Hypervisors : Xen, Unix

Ondersteunde programmeertaal : OCaml

GitHub stars : 657

Er kan gezegd worden dat het voor een deel allemaal begon bij MirageOS (Mirage, n.d.). Hun paper (Madhavapeddy et al., 2013) over unikernels en MirageOS wakkerde veel interesse aan. Ervoor was er wel al sprake van unikernels, maar MirageOS zorgde voor veel nieuwe initiatieven.

MirageOS is een cloud besturingssysteem gemaakt om veilige netwerk toepassingen, met een hoge prestatie, te maken op verschillende omgevingen.

De programmeertaal dat gebruikt word om een MirageOS applicatie te maken is OCaml. OCaml is de algemene implementatie van de Caml programmeertaal en voegt object georiënteerd programmeren toe. Het wordt extensief gebruikt door Facebook. Deze taal is niet bekend en dit kan ervoor zorgen dat het niet veel tractie krijgt.

De voornaamste redenen om OCaml te gebruiken zijn static type checking en automatic memory management. De eerste reden is om tegen te gaan, dat er iets fout gaat wanneer een programma aan het werken is. De compiler gaat kijken of het geen onveilige code kan vinden. Als dit het geval is, wordt het programma niet gecompileerd. Memory management is belangrijk voor resource leaks tegen te gaan. Resource leaks kunnen ervoor zorgen dat het programma meer middelen gebruikt dan het nodig heeft. In het extreme geval kan het systeem waarop het programma werkt ook hinder ondervinden van dit probleem.

De applicatie kan geschreven worden op een Linux of Mac OSX besturingssysteem. Deze applicatie kan dan werken op een Xen of Unix omgeving. Dit geeft veel mogelijkheden op het vlak van omgevingen. Er zijn plannen om mobiele omgevingen te ondersteunen.

MirageOS bestaat al een tijd en heeft een groot aantal libraries ter beschikking. Het heeft een uitstekende toolchain voor het compileren van programma's en het debuggen van de resulterende unikernel. Debuggen kan soms tot problemen leiden bij unikernels, want er kan niet in de unikernel gekeken worden welke problemen zich voordoen. Dit komt omdat de unikernel geen shell heeft. De debug optie kan hierbij helpen.

6.3.6 IncludeOS

Implementatie programmeertaal : C/C++

Hypervisors : KVM, VirtualBox

Ondersteunde programmeertaal : C++

GitHub stars : 1341

IncludeOS (Oslo and Akershus University College and of Applied Sciences, n.d.) is gemotiveerd door het paper van Bratterud and Haugerud, 2013. Het onderscheid tussen een minimale virtuele machine tegenover een unikernel is zeer klein. Daarom worden beide termen afwisselend gebruikt. Net zoals ClickOS moeten de applicaties geschreven worden in C++.

IncludeOS zorgt voor een bootloader, standaard libraries, modules voor de drivers te implementeren en een build- en uitrolsysteem. Het is simpel om applicaties te maken voor deze unikernel. Je moet alleen één dependency

toevoegen aan het programma. Dan kan het worden omgezet naar een unikernel. Er veranderd dus niet veel voor de software ontwikkelaars. Dit zorgt voor een vlotte overgang en dit is zeker belangrijk wanneer er gekozen wordt om applicaties te bouwen voor unikernels.

Meerdere processen tegelijk laten werken op een unikernel van includeOS is niet mogelijk. Dit kan sommige software ontwikkelaars afschrikken. Het gebruik van microservices (hoofdstuk 7) is nog niet wijdverspreid en kan een factor zijn bij het selecteren van een unikernel implementatie. Enerzijds gaan bedrijven nooit bij unikernels uitkomen, wanneer hun architectuur niet gebaseerd op microservices. Er zijn ook geen race conditions mogelijk, omdat er maar één proces mogelijk is.

Momenteel ligt de focus van IncludeOS voornamelijk op C++. Dit is een strategie dat kan helpen wanneer software ontwikkelaars zoeken naar een implementatie die een gemeenschap heeft. IncludeOS heeft een grote gemeenschap van C++ software ontwikkelaars. Hun doel is vooral om een soortgelijk Node.js te maken maar dan in efficiënt C++.

Er zijn geen plannen om hogere programmeertalen zoals Javascript te ondersteunen. Ook is IncludeOS niet POSIX compliant en dit kan voor problemen zorgen wanneer er extra functionaliteit moet worden toegevoegd.

Als omgeving focussen ze KVM en Virtualbox. Het is het dus gemakkelijk om een unikernel te testen op de ontwikkelomgeving. Als je services schrijft in C++ dan is IncludeOS een zeer goede keuze. Er kan meer informatie gevonden worden op de GitHub repository: Oslo and Akershus University College and of Applied Sciences, n.d.

6.3.7 OSv

Implementatie programmeertaal : C/C++

Hypervisors : VMWare, VirtualBox, KVM, Xen

Ondersteunde programmeertaal : Java

GitHub stars : 2121

OSv (System, n.d.) is een implementatie die een grote naam heeft binnen de unikernel wereld. Er wordt een hoog aantal programmeertalen ondersteund. Waaronder Java, Ruby, Javascript, Scala en vele anderen. Hierbij moeten er wel vermeld worden dat de implementaties van Ruby en Javascript in Java zijn geschreven. Rhino en JRuby zijn de namen van deze implemenaties. Het is simpel om deze programmeertalen toe te voegen, wanneer Java als programmeertaal wordt ondersteund.

Verder kunnen de resulterende unikernels werken op veel omgevingen: VMware, VirtualBox, KVM en Xen.

Zoals IncludeOS voorheen is OSv geschreven in C++.

Voor het beheren van een OSv instance kan gebruik worden gemaakt van de GUI. Bij de meerderheid van unikernels is informatie te vinden door middel van een GUI onmogelijk. Extensies voor de hypervisor kunnen hierbij helpen, maar dan nog komt de user experience tekort. De GUI is gebouwd op een REST API die de componenten van OSv openstellen. Dit heeft overstemming met de manier, hoe de architectuur van het Docker ecosysteem in elkaar zit. Deze componenten stellen een API open waar de tools verder op gebouwd kunnen worden. Er is een API-specificatie die kan bekeken worden om te zien hoe deze componenten met elkaar werken.

OSv ondersteund Amazon Web Services en Google Container Engine als cloud providers. Het is uitzonderlijk dat een unikernel zoveel informatie heeft over hoe het moet gebruikt worden. Er is documentatie over cloud providers, hypervisors, hoe het gebruikt moet worden en hoe programma's moeten omgezet worden naar de implementatie.

Het is de meest populaire implemenatie van unikernels van alle implementaties die we hebben overlopen tijdens deze vergelijking. Ook de activiteit op de Github repository is het hoogste van alle bekeken unikernels.

Meer informatie is te vinden in volgende paper: Kivity et al., 2014.

6.4 Conclusie

Er zijn veel verschillende soorten implementaties van unikernels op dit moment. Er is MirageOS die als één van de eerste opkwam en ook de meest extreme weg opgaat met het starten van een minieme basis. HalVM bevindt zich in aan de dezelfde kant als MirageOS. Terwijl OSv en Rumprun zich bevinden aan de overkant. Ze ondersteunen een groot aantal programmeertalen en omgevingen. Dit wordt mogelijk gemaakt door een tussenlaag te gebruiken die zorgt voor compatibiliteit.

Hetzelfde fenomeen kunnen we vinden met de toepassingen waar de unikernels kunnen voor gebruikt worden. ClickOS heeft vooral middlebox applicaties als doel en andere unikernels kunnen voor uiteenlopende situaties kunnen gebruikt worden.

Table 6.1: Implementaties unikernels

Naam	Taal implementatie	Hypervisor	Ondersteunde talen	GitHub sterren
ClickOS	C/C++	Xen	bindings	243
HaLVM	Haskell	Xen	Haskell	665
LING	C/Erlang	Xen	Erlang	523
Rumprun	C	hw, Xen, POSIX	C, C++, Erlang, Go, ...	469
MirageOS	OCaml	Xen	OCaml	657
IncludeOS	C++	KVM, VirtualBox	C++	1341
OSv	C/C++	KVM, Xen, ...	JVM	2121

7. Microservices en monolieten

De meeste programma's worden gemaakt door een monolothische architectuur te hanteren (Villamizar et al., 2015). Alle functionaliteit en verantwoordelijkheden worden gestopt in één programma. Dit soort van programma wordt een monoliet of monolith in het Engels genoemd. De architectuur waarin deze soort programma's voorkomen noemen we een monolithische architectuur. Software ontwikkelaars structureren door middel van patronen (Tichy, 1997) om functionaliteit en verantwoordelijkheid van elkaar te scheiden. Modulariteit is hierbij een belangrijk gegeven.

Een probleem dat veel voorkomt bij een monolithische architectuur is dat het programma zeer complex wordt na verloop van tijd (Villamizar et al., 2015). Functionaliteit toevoegen is niet vanzelfsprekend bij een complex programma, want er moet rekening worden gehouden met de andere delen van het programma. Nieuwe software ontwikkelaars die het programma niet kennen moeten eerst een paar weken het programma verkennen. Dan pas kan er begonnen worden met nieuwe functionaliteit te schrijven.

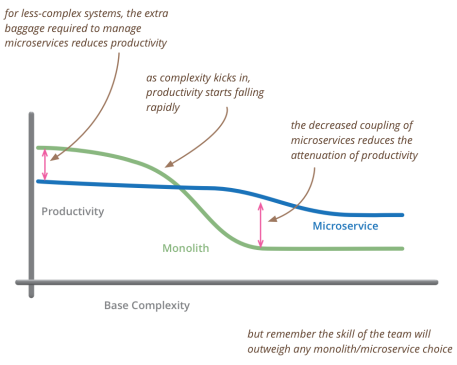

Figure 7.1: Productiviteit en complexiteit van microservices architectuur tegenover een monolithische architectuur (Martin Fowler, 2015)

Schaalbaarheid is een probleem waar ook tegen gelopen wordt na verloop van tijd (Villamizar et al., 2015). Sommige onderde-

len van een programma moeten meer trafiek kunnen verwerken dan andere delen. Zoals al aangehaald werd in hoofdstuk 3, hebben verschillende soorten programma's andere middelen nodig. Dit is hetzelfde bij de interne delen van programma. Het schalen van deze componenten is alleen mogelijk door een nieuwe instantie toe te voegen van de hele applicatie of de implementatie te verbeteren. Eén groot programma is ook niet handig voor grote teams te laten samenwerken. Het uitbrengen van een versie moet dan nagegaan worden bij alle teams die aan dat programma werken.

Al langer bestaat het idee om een groot programma op te splitsen in kleinere programma's. Telecommunicatie is een industrie waarbinnen microservices al werden gebruikt Griffin and Pesch, 2007. De opkomst van software containers heeft dit deze werkwijze meer verspreid. Dit komt omdat het gemakkelijker is geworden om kleinere programma's met elkaar te verbinden, zelf als ze zich niet op dezelfde omgeving bevinden. De architectuur waarbinnen dit idee wordt gebruikt, wordt microservices architectuur genoemd. De microservices kunnen we bekijken als deelapplicaties die één verantwoordelijkheid hebben.

Een programma opsplitsen in componenten, met één verantwoordelijkheid, zorgt ervoor dat het meer schaalbaar is. Containers en unikernels helpen hierbij. Het opstarten van een nieuwe instantie van een microservice neemt minder tijd in beslag dan het opstarten van een nieuwe instantie van een monoliet.

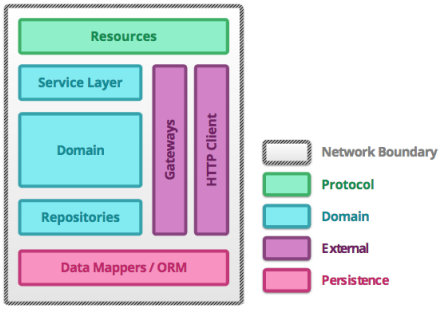

Figure 7.2: Structuur van een architectuur met microservices (Toby Clemson, 2014)

Bij microservices zal de topologie van het probleemdomein goed gekend moeten zijn. Starten met het gebruiken van microservices architecture wanneer men het domein niet goed kent, vraagt om problemen. Het ontwerpen van een microservices architectuur moet goed gebeuren. Anders kunnen er problemen voorkomen met impact op de gehele architectuur, bijgevolg moeten er grote delen worden herschreven. Een monolitische architecture zal beter kunnen reageren op dit probleem. Als men later het domein kent kan een microservices architecture worden gebruikt. Hiervoor moet de monoliet modulair geschreven worden. Modulariteit is een vanzelfsprekende bouwsteen binnen programmeren.

De complexiteit van een monoliet wordt overgebracht naar het communiceren en het behouden van de consistentie tussen de microservices. Verder wordt ook het opstellen van de architectuur moeilijker.

Het voordeel van een microservices architectuur is dat de microservices van elkaar gescheiden zijn. Het gebruik van een nieuwe technologie of framework is niet langer een groot probleem. Dit komt omdat de microservices los van elkaar staan. Er kan dus een andere programmeertaal gebruikt worden zolang de communicatie tussen de microservices consistent blijft.

De communicatie van de microservices gebeurd via het netwerk. Dit maakt het mogelijk om microservices op verschillende virtuele machines of software containers te laten werken. Software defined network (García Villalba et al., 2015) neemt veel complexiteit weg van de communicatie.

Een monolitische architectuur opstellen in productie is relatief simpel tegenover een microservices architectuur. Wanneer er een nieuwe versie moet worden opgesteld in productie kan gebruik worden gemaakt van blue-green deployment (Martin Fowler, 2016). Waarbij we een oude en nieuwe versie hebben van de architectuur en de router het verkeer verlegt naar de nieuwe architectuur. Dit zorgt voor een gemakkelijke overgang.

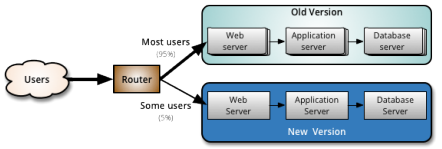

Figure 7.3: Voorbeeld van een canary release (Danilo Sato, 2014)

Bij microservices kan gebruik gemaakt worden van een canary release strategie (Danilo Sato, 2014). Hier wordt weer een nieuwe versie opgesteld met de laatste veranderingen van de architectuur. De router gaat een deel van het verkeer naar de nieuwe architectuur versturen. Naarmate de tijd vordert, wordt het vertrouwen in de nieuwe versie nagegaan. Als het vertrouwen is toegenomen dan zal meer verkeer naar de nieuwe versie worden gestuurd. Zo kunnen er problemen worden nagegaan en er gepast op reageren. Uiteindelijk krijgt de oude versie geen verkeer meer en wordt alleen de nieuwe versie gebruikt.

De systeembeheerder krijgt meer werk omdat er nu tientallen microservices moeten beheerd worden in plaats van één grote applicatie. Het beheren van deze microservices en hun logs wordt een belangrijke bron van informatie binnen de architectuur. Er kan gezegd worden dat deze microservices simpeler zijn om te verstaan, omdat de functionaliteit per microservices beperkt is. Sommige halen aan dat het de complexiteit die we niet meer tegenkomen in de applicatie nu terechtkomt bij het samen te laten werken van de verschillende onderdelen.

8. Proof of concept: unikernels in de praktijk

8.1 Inleiding

Dit hoofdstuk zal de toepassingen voor unikernels toelichten. De huidige implementaties van unikernels geven een zekere vrijheid wat voor soort applicaties er kunnen opgezet worden.

Er kan gesproken worden over twee types van applicaties: dynamische en statische. Hiernaar wordt ook verwezen in het Engels als stateless en stateful. Blogs en websites zonder al te veel functionaliteit zijn stateless. Het resultaat van de applicatie qua functionaliteit zal hetzelfde zijn voor elke gebruiker. Stateful applicaties zijn web applicaties zoals Wordpress en de meeste applicaties waar er informatie wordt bijgehouden en kan worden aangepast. Het overgrote deel van de applicaties die werken met unikernels zijn stateless.

De verklaring hiervan is te vinden bij de eigenschappen van unikernels. Unikernels worden geoptimaliseerd voor de applicaties waarvoor ze gecompileerd worden. Het theoretische hoofdstuk over unikernels heeft hiervoor al de redenen gegeven. Wanneer de applicatie en een database zich op een unikernel bevinden dan zou de optimalisatie bijna geen voordelen geven. Daarbij komt er nog het extra gegeven van het opsplitsen van de verantwoordelijkheden van de componenten van een omgeving. Databanken vragen om een ander soort machine en/of besturingssysteem dan een webapplicatie. Hierbij is er nog het gegeven dat het bestaan van een relationele database nog niet bestaat voor unikernels onder een stabiele vorm. Irmin (verwijzing) kan wel een oplossing zijn, maar dit is moeilijk een

relationele database te noemen. Ook kan Xenstore van de Xen hypervisor worden aangehaald. Dit levert dan weer het probleem op dat het niet op de verschillende soorten hypervisors kan gebruikt worden. Daarom kunnen databases zich beter bevinden op een virtuele machine met een algemeen besturingssysteem. Het probleem van databases en de correcte omgeving van databases kan ook teruggevonden worden bij software containers. Tot op het moment wordt aangeraden om databases te laten werken op virtuele machines omwille van de problemen met de levenstijd van software containers.

Veel van toepassingen die werken met unikernels bevinden zich meestal in een bepaald segment. Dit is het geval met technologieën die nog moeten toegroeien naar algemene of meerdere use cases. Toepassingen die moeten gebruik maken van unikernels zullen zich grotendeels bevinden binnen de netwerklaag. Rumprun heeft al pogingen gedaan om een soort van RAMP (RumpRun, Apache, MySQL, PHP) opstellingen te maken, maar dit staat nog in vroeg stadium. De functionaliteit laat hierbij te wensen over. Het maken van unikernels voor statische applicaties is dus een betere keuze voor de verschillende soorten unikernels te vergelijken.

8.1.1 Keuze Programmeertaal

Er worden een aantal programmeertalen ondersteund door unikernels. PHP, java, Node.JS, python, C++ en Go. Deze programmeertalen kunnen worden opgesplitst in twee groepen. Enerzijds zijn er de gecompileerde programmeertalen, zoals Go en C++, waarbij de broncode wordt omgezet in een uitvoerbaar bestand. Anderzijds kan er gesproken worden over geïnterpreteerde programmeertalen zoals PHP, java, Node.JS en python. Hierbij wordt de broncode door een interpreter vertaalt naar instructies die de processor kan begrijpen en de interpreter voert de instructies meteen uit. Het verschil is dus dat de compiler de instructies opslaat tegenover de interpreter die deze onmiddellijk uitvoert. Dit heeft een gevolg op de omgeving waarin de opstelling zich bevindt. Omwille van de aard van unikernels (beperkte levenstijd) kan een interpreter extra tijd innemen om de unikernel te starten. Enerzijds zou er gedacht kunnen worden dat dit een voordeel geeft aan de unikernels tegenover de software containers en virtuele machines. Maar de use cases waarbinnen unikernels kunnen worden gebruikt bevinden zich eerder in andere situaties. Voorbeelden hiervan zijn wanneer er uitzonderlijk geschaald moet worden of wanneer de unikernel de rol van een load balancer moet vervullen.

Go zal gebruikt worden als programmeertaal om de applicaties te realiseren. Hieronder vind je een link naar deze programmeertaal.

Go - https://golang.org

8.2 Opstelling Virtuele Machine

Deze opstelling maakt gebruik van Virtualbox als hypervisor met Vagrant, als configuratie tool, om de ontwikkelomgeving gemakkelijk op te stellen. De vereisten voor deze opstelling zijn:

Virtualbox - https://www.virtualbox.org/

Vagrant - https://www.vagrantup.com/

Repository Go Virtuele machine omgeving - https://github.com/michieldewilde/vagrant-golang-bp

Git - https://git-scm.com/

Het besturingssysteem dat wordt gebruikt is Debian. De opstelling is simpel te noemen, omdat er enkel wordt uitgegaan van een applicatie die zich bevindt op de virtuele machine zonder andere componenten. In een productie omgeving zullen deze twee componenten zich uiteraard niet op dezelfde virtuele machine bevinden. Go wordt gebruikt als programmeertaal om de applicatie op de virtuele machine te demonstreren.

Allereerst is git nodig om de repository lokaal te downloaden. Git is een versie controle systeem om gemakkelijk grote en kleine projecten te beheren en te ontwikkelen wanneer er in team gewerkt wordt.

Het volgende commando zal de repository met de configuratie van de virtuele machine ophalen:

```
$ git clone https://github.com/michieldewilde/vagrant-golang-bp
```

De Vagrantfile in deze directory bevat de configuratie voor deze machine en zal ook zorgen dat de huidige directory van de host synchroniseert met de toegewezen directory in de virtuele machine.

Het volgende commando stelt de virtuele machine op en configureert deze:

```
$ vagrant up
```

Om deze virtuele machine te betreden wordt het volgende commando uitgevoerd:

```
$ vagrant ssh
```

Alle inhoud van de directory die we binnen hebben gehaald, bevindt zich op het /vagrant path. De programmeertaal Go heeft een structuur voor de projecten waarmee er wordt gewerkt. Er kan worden vastgesteld dat er

in de directory die binnen is gehaald dat er een src directory aanwezig is. Binnen de src directory bevindt zich een main.go bestand.

Om dit bestand uit te voeren:

```
$ go run main.go
```

Nu staat er een web server op en deze kan bereikt worden door in uw browser het IP van de virtuele machine in te geven, 192.168.10.10, en de poort 8080 met als prefix : er aan toe te voegen.

Bij deze is er een ontwikkelomgeving voor de programmeertaal Go opgesteld, met daarbij een web server. Het opstellen van een hele virtuele machine voor één applicatie kan overdreven lijken, maar daarbij zijn de vereisten voor de Go ontwikkelomgeving gescheiden van de host machine. Het toevoegen van extra onderdelen kan met gemak gebeuren door andere pakketten te installeren en te configureren. De functie van de virtuele machine die momenteel is opgezet is algemeen en kan gemakkelijk veranderen door onderdelen toe te voegen. Daarbij kan de virtuele machine met een algemeen besturingssysteem, Debian, beschouwd worden als niet gespecialiseerd. Dit is een uiteindelijk doel van elke component in de ontwikkelomgeving.

8.3 Opstelling Software Containers

Deze opstelling maakt gebruik van Virtualbox als hypervisor met daarop Docker om de software containers te beheren en configureren. De vereisten voor deze opstelling zijn:

Docker Toolbox - https://www.docker.com/products/docker-toolbox

Repository Go Software Container Omgeving - https://github.com/michieldewilde/docke
golang-bp

Simpele web server - https://github.com/michieldewilde/go-web-example

Git - https://git-scm.com/

Er zijn al nieuwere versies van Docker uitgebracht voor specifieke besturingssystemen zoals Docker for Mac and Windows. Deze vertonen echter problemen bij het opstellen van unikerneles in het volgende hoofdstuk. Docker Toolbox laat toe om beide omgevingen te ondersteunen. Docker maakt het gemakkelijk om applicaties op te zetten in software containers.

Het volgende commando zal de repository met de configuratie van de software containers ophalen:

```
$ git clone https://github.com/michieldewilde/docker-golang-bp
```

Ga naar de opgehaalde directory en daarbinnen vindt uw een bestand genaamd Dockerfile, met alle configuratie die nodig is om een web server op te stellen binnen een software container.

Listing 8.1: Dockerfile

```
FROM golang:1.6

# Haal de repository met het voorbeeld op
RUN go get github.com/michieldewilde/go-web-example

# Compileer het project
RUN go install github.com/michieldewilde/go-web-example

# Wanneer de software container wordt gestart, voer het
   gecompileerde project uit
ENTRYPOINT /go/bin/go-web-example

# Open poort 8080 om de web server te kunnen bereiken van
    op de host
EXPOSE 8080
```

Als Docker geïnstalleerd wordt dan wordt nog een extra tool mee geïnstalleerd: docker-machine. Deze tool laat toe om Docker hosts te creëren. Deze laat dan toe om met de Docker client die zich bevindt op de host te communiceren.

Er zijn nog manieren om een meer uitgebreide en/of geavanceerde opstelling te bereiken dit kan bereikt worden door gebruik te maken van het docker-compose commando.

Listing 8.2: docker-compose.yaml

```
# gebruik versie twee van de docker-compose syntax
version: '2'
# de verschillende componenten die moeten nodig zijn voor
    de opstelling
services:
  # onze simpele web server van de vorige voorbeelden
  go-web-example:
    # wanneer de service gecompileerd moet worden,
       gebruik de Dockerfile in de huidige directory (
       Dockerfile van het vorige voorbeeld)
    build: .
    # stel de pport 8080 open
    ports:
     - "8080:8080"
```

```
# extra component voor database functionaliteit
redis:
  image: "redis:alpine"
```

Door gebruik te maken van een docker-compose bestand wordt de opstelling van de verschillende componenten simpeler gemaakt. Ook het schalen van de componenten, wanneer er extra last wordt ondervonden, is ook gemakkelijker. Dit komt omdat de componenten gescheiden zijn van elkaar. Het toevoegen van extra componenten aan een opstelling kan dus gebeuren door een service toe te voegen aan het docker-compose bestand. Op de virtuele machine zou deze helemaal moeten worden geïnstalleerd en geconfigureerd worden.

8.4 Opstelling Unikernel

Deze opstelling wordt uitgevoerd op een virtuele machine omdat er extensieve veranderingen worden uitgevoerd. Om deze opstelling lokaal te gebruiken zijn er drie vereisten nodig. Ten eerste de Virtualbox hypervisor. Verder Vagrant om de ontwikkelingomgeving op te stellen. Vagrant en Virtualbox gaan hand in hand om een start te maken wanneer er een bepaalde ontwikkelingomgeving nodig is. Vagrant werkt door middel van een bestand (VagrantFile). In dit bestand is alle configuratie van opstelling beschreven. Het haalt eerst een afbeelding op waarop de configuratie wordt toegepast. Packer is de laatste vereiste. Het zal gebruikt worden om op de virtuele machine de configuratie toe te passen. Dit resultaat wordt dan opgeslagen als een afbeelding.

Virtualbox - https://www.virtualbox.org/

Vagrant - https://www.vagrantup.com/

Packer - https://www.packer.io/

Allereest moeten alle voorgenoemde vereisten zijn geïnstalleerd op uw machine voor er verder kan worden gegaan. MirageOS heeft een repository met alle benodigdheden om deze virtuele machine op te stellen. Deze kan gevonden worden op de volgende link: https://github.com/michieldewilde/mirage-vagrant-vms. Haal deze repository lokaal op de machine en navigeer naar de locatie van deze opgehaalde folder. Er is keuze uit ubuntu 14.04, ubuntu 14.10, debian 7.8.0 en xenserver 6.5.0. Er is gekozen voor Ubuntu 14.04 omdat er het minste problemen bij dit besturingssysteem zijn vastgesteld voor deze opstelling. De Makefile bevat gemakkelijkheidshalve commando's om de virtuele machine op te stellen zonder een lijst van commando's in te geven.

Het volgende commando zal een nieuwe box aanmaken door het gebruik van Packer:

```
$ make ubuntu-14.04-box
```

Het volgende commando neemt de box, die is aangemaakt door het vorige commando, en past de configuratie erop toe:

```
$ make ubuntu-14.04-box
```

Navigeer naar de folder van het gekozen besturingssysteem en ssh in de virtuele machine:

```
$ cd ubuntu-14.04 && vagrant ssh
```

8.4.1 MirageOS

MirageOS is gestart vanaf nul met een schone lei. Dit betekende dat veel van de bestaande tools die nu worden gebruikt worden, zoals webservers (Apache, Nginx) en Databases (MySQL), herschreven zouden moeten worden. Het duurt een tijd voordat deze tervoorschijn komen en dit is één van de grootste redenen dat er gekozen is voor een statische applicatie.

De opstelling die gemaakt is in de vorige sectie heeft de toolset van MirageOS ook geïnstalleerd. Verder moet er worden gekeken of de juiste versie van Ocaml en OPAM (package manager) is geïnstalleerd:

OPAM versie:

```
$ opam --version
# De versie moet minstens 1.2.2 zijn.
1.2.2
```

Ocaml versie:

```
$ ocaml -version
# Deze moet 4.02.3 of hoger zijn.
$ opam switch 4.02.3
```

In de login shell moet de omgeving van opam ingeladen wanneer er ingelogd wordt. Dit kan gedaan worden door de volgende lijn toe te voegen aan het /.bashrc bestand:

```
$ eval `opam config env`
```

Verder moeten er ook gekeken of de versie van mirage niet moet aangepast moet worden:

```
$ opam install mirage
$ mirage --help
# Versienummer moet hoger zijn 2.9.0
```

Het mirage commando kan gebruikt worden om applicaties te maken en uit te rollen.

Als statische website zullen we de website van MirageOS zelf gebruiken. De repository van de website bevindt zich op de volgende link: https://github.com/mirage/mir www/. Eerst moet de repository worden opgehaald.

Dit doen we door het git commando te gebruiken:

```
$ git clone https://github.com/mirage/mirage-www/
```

Vooraleer we de omgeving van de applicatie gaan configureren moet het mogelijk worden gemaakt om de Xen hypervisor te laten communiceren met de virtuele machine. Deze twee zijn afgeschermd van elkaar en daarom moet er een TUN/TAP constructie worden gemaakt.

sudo modprobe tun Dit doen we als volgt:

```
$ sudo apt-get install tunctl
# laden van de tuntap kernel module
$ sudo modprobe tun
# maak een tap0 interface
$ sudo tunctl

$ sudo ifconfig tap0 10.0.0.1 up
```

Navigeer naar de opgehaalde folder van de MirageOS website. Daarna moet de omgeving voor de applicatie worden ingesteld:

```
$ cd mirage-www
$ make prepare
$ cd src
# configureren voor de Xen hypervisor
# direct MirageOS network stack
# maak gebruik DHCP
$ mirage configure --xen \
    -vv --net direct \
    --dhcp true  \
    --tls false --network=0
```

Hierdoor is de omgeving van de applicatie goed ingesteld. Het compileren van de unikernel gebeurd door middel van het volgende commando:

```
$ make
```

Eerst dienen er nog file blocks aangemaakt te worden om de afbeeldingen en de stylesheets te kunnen gebruiken: Aanmaken file blocks

```
$ ./make-fat_block1-image.sh
```

Verder moeten er nog een paar wijzigingen worden aangebracht in het www.xl bestand. Dit bestand wordt doorgegeven naar de Xen launcher om de unikernel te starten. Het disk gedeelte moeten worden aangepast naar het volgende:

```
disk = ['format=raw,
         vdev=xvde,
         access=rw,
         target=/home/vagrant/mirage-www/src/fat_block2.img',
        'format=raw,
         vdev=xvdc,
         access=rw,
         target=/home/vagrant/mirage-www/src/fat_block1.img
      ']
```

Uiteindelijk kunnen we de unikernel starten:

```
$ sudo xl -v create -c www.xl
```

Doorheen de output kan gezien worden op welk IP de statische website staat te luisteren. Als er genavigeerd wordt naar dat adres dan kan de MirageOS website worden bekeken.

8.4.2 Rumprun

Bij de rumprun unikernel zal er gebruik gemaakt worden van Nginx webserver om een statische website weer te geven. Allereerst moeten de build tools van Rumprun worden geïnstallereerd. Dit wordt bereikt als volgt:

We halen de Rumprun repository op en alle dependencies

```
$ git clone http://repo.rumpkernel.org/rumprun
$ cd rumprun
$ git submodule update --init
```

Verder moeten er een toolchain gemaakt worden voor de unikernels te compileren. In dit geval worden de unikernels gecompileerd voor ze te uit te voeren op de Xen hypervisor.

```
$ ./build-rr.sh xen
```

Dit maakt een folder aan genaamd rumprun met daarin een bin directory die toegevoegt zal worden aan het PATH. Dit wordt gedaan zodat de

rumprun toolchain overal in het systeem kan gebruikt worden. Toevoegen Rumprun toolchain aan PATH:

```
$ export PATH=${PATH}:$(pwd)/rumprun/bin
```

De toolchain voor Rumprun unikernels te maken is geïnstalleerd. Nu moet er een nginx Rumprun unikernel worden opgehaald. Ophalen Nginx unikernel:

```
$ git clone http://repo.rumpkernel.org/rumprun-packages
$ cd rumprun-packages/
$ cd nginx/
```

Er moet ook een package worden geïnstalleerd om een ISO bestand te maken. Dit zal gebruikt worden voor bestanden te laden in de unikernel. Installeren genisoimage:

```
$ sudo apt-get install genisoimage
```

De volgende stap is om Nginx te compileren. Compileren Nginx:

```
$ make
```

De unikernel werkt nu al maar moet omgezet worden naar een unikernel die werkt op de Xen hypervisor. unikernel voor Xen maken:

```
$ rumprun-bake xen_pv ./nginx.bin bin/nginx
```

Hierbij is nginx.bin de unikernel die moet worden gestart.

Uitvoeren van de Nginx unikernel:

```
# data van www folder laden (inhoud van statische website)
# dhcp instellen
# laden van unikernel en configuratie
$ rumprun -T tmp xen -M 64 -i \
-b images/data.iso,/data \
-I mynet,xenif,bridge=br0 -W mynet,inet,dhcp \
-- nginx.bin -c /data/conf/nginx.conf
```

Het adres van de nginx webserver is te vinden in de output in het gedeelte van met de DHCP informatie.

De bridge br0 ,die in MirageOS en Rumprun unikernel wordt gebruikt, is een host-only bridge tussen de host en de virtuele machine om netwerkverkeer tussen beide te laten werken.

8.5 Conclussie

Tijdens dit hoofdstuk is er gekeken naar de mogelijkheden en limitaties die er zijn wanneer er gebruik wordt gemaakt van unikernels. De verschillende opstellingen (virtuele machine, software container en unikernel) tonen duidelijk aan dat er een gebrek is aan maturiteit bij unikernels. Het ecosysteem is gefragmenteerd door de verschillende implementaties. Verder zijn er moeilijkheden om de unikernels te debuggen in een productieomgeving. Dit kan leiden tot frustaties als er naar een andere omgeving wordt veranderd. Er zijn verschillende pogingen om al deze build tools achter één interface te brengen en ze zo simpel te kunnen gebruiken. Spijtig genoeg staat dit alles nog in een vroeg stadium. Statische applicaties en specifieke netwerktoepassingen zijn één van de enige mogelijkheden momenteel om unikernels in productie te gebruiken. Er werd ook geprobeerd om een Wordpress blog op te stellen met behulp van Rumprun maar er werd op veel moeilijkheden gestoten. Ten eerste het compileren van de gehele applicatie met de omgeving zorgde voor problemen wanneer bepaalde dependencies de juiste versie niet hadden. Deze versies waren dan weer niet te vinden in de Rumprun package directory. Verder was het debuggen van een unikernel uiterst moeilijk omwille van de afwezigheid van tools hiervoor. Dit zorgde voor veel aanpassingen. Daarbij kwam nog dat het hele gegeven zeer traag is om veranderingen te kunnen maken.

9. Conclusie

Het efficiënt gebruiken van de middelen van de productieomgeving is een belangrijk gegeven. Innoveren om deze omgeving beter te gebruiken is broodnodig. Virtuele machines, containers en unikernels zijn maar enkele innovaties die dit proberen te bereiken. Software containers hebben al gezorgd voor grotere veranderingen binnen de meeste bedrijven. Doorheen de bachelorproef werd gekeken naar nieuwe innovaties, vooral unikernels, en hun mogelijkheden tegenover andere oplossingen.

Op de volgende onderzoekvragen werd een antwoord gezocht:

- Wat zijn de use cases voor unikernels?
- De verhouding van software containers tegenover unikernels?
- De gevolgen voor de applicatie architectuur wanneer unikernels in gebruik worden genomen?
- Wordt het opstellen en beheren van applicaties eenvoudiger of niet?
- Wat is de impact op beveiliging?

De eerste onderzoeksvraag heeft betrekking tot de use cases waarvoor unikernels kunnen gebruikt worden. De use cases voor unikernels liggen niet bij dynamische web applicaties, dit probleem ligt bij het gebrek aan ondersteuning voor databases. Statische web applicaties en netwerk applicaties dienen zich meer als use cases. Verder kan er naar de toekomst

gekeken worden en kan er gezegd worden dat Internet of Things toepassingen ook mogelijk zijn. De veiligheid die deze toepassing vraagt is te vinden bij unikernels.

Het antwoord op de tweede onderzoeksvraag, De verhouding van software containers tegenover unikernels, is het volgenden: software containers hebben al een heel ecosysteem beschikbaar met Docker. Het opzetten van de software container in het proof of concept toont ook aan dat het gemakkelijker is om de omgeving op te stellen. Eenvoud zorgt ervoor dat de gebruikers meer vertrouwen zullen hebben om te starten. Unikernels hinken achterop op het vlak van ecosysteem en tools omwille van de verschillende implementaties met elk hun eigen focus en het gebrek aan maturiteit. Op dit moment zijn software containers de keuze.

De derde onderzoeksvraag kijkt naar de architectuur van de applicatie. Hierbij werd ondervonden dat monolieten problemen krijgen met de hoeveelheid van dependencies wanneer deze gecompileerd moeten worden. Daarom zullen applicaties die in een unikernel compileerd worden, eerder bestaan uit microservices. De mogelijkheid om elk onderdeel van een applicatie de optimale middelen ter beschikking te stellen is een niet te onderschatten voordeel. Verder kan de logica van de applicatie gedeeltelijk verhuizen van de gehele applicatie naar hoe de verschillende onderdelen van de applicatie.

De vierde onderzoeksvraag valt terug op de tweede onderzoeksvraag. Voor het opstellen en beheren van applicaties moet er gekeken worden naar het ecosysteem rond de technologie. Momenteel levert dit gedeelte de meeste moeilijkheden op. Ook de verschillen van implementaties op het vlak van unikernels, daarbij in acht genomen: de ondersteunde hypervisor en programmeertalen, kunnen zorgen voor moeilijkheden. Een omgeving opstellen en een applicatie compileren en uitvoeren is geen gemakkelijke opgave zoals kan afgeleid worden uit het proof of concept.

Als laatste vraag wordt er gekeken naar de veiligheid. Veiligheid is één van de grote voordelen van unikernels tegenover de andere opstellingen. Kwetsbaarheden zijn minder aanwezig door de kleinere aanvalsruimte en de specialisatie van de unikernel. Absolute veiligheid is niet mogelijk, maar unikernels zorgen toch voor verbetering. Het schrijven of vinden van een zwakheid van een applicatie en/of omgeving is niet uiterst zinvol, als het bekeken wordt vanuit een groter geheel wanneer deze enkel werkt bij een gelimiteerd aantal omgevingen/applicaties. Hierbij zorgt unikernels dus voor veel vooruitgang. Verder als er gekeken wordt naar de efficiëntie, kan er verwezen worden naar de architectuur van de applicatie waarbij elk onderdeel in een optimale afzonderlijke omgeving terecht komt.

10. Bijlagen

10.1 Bachelorproef voorstel

Unikernels: Monoliet of microservices
Bachelorproef

Michiel De Wilde

Samenvatting
Unikernels laten toe om veiliger en efficiënter te werken dan een software container of virtuele machine waarbij een algemeen besturingssysteem wordt gebruikt. Het opstellen van applicaties met unikernels kan moeilijkheden geven omdat er niet uitgegaan wordt van de beperkte functionaliteit van een unikernel. Veel bedrijven stappen over op software containers en unikernels wordt aangeduid als ge volgende stap. Door middel van een experiment waarbij verschillende soorten applicaties (monoliet, microservices) op verschillende omgevingen(virtuele machine, software container, unikernel) worden opgesteld. De resultaten kunnen worden vergelijkt door de mogelijkheden en moeilijkheden van elke opstelling te bekijken. Dit onderzoek zal aantonen dat verschillende omgevingen zich meer lenen tot het behuizen van een specifieke soort van applicatie. In het heden en de toekomst is efficiëntie en veiligheid van applicatie heel belangrijk deze studie zal tonen welke soorten applicaties op welke omgevingen beter en veiliger werken.

Sleutelwoorden
Systeembeheer, Applicatieontwikkeling

Contact: [1] michiel.dewilde.u9305@student.hogent.be

Inhoudsopgave

1. Introductie

De voorbije jaren zijn er veel opmerkingen en problemen gevonden bij de de huidige (algemene) besturingssystemen. Deze besturingssystemen zijn niet alleen te vinden op computers van gewone gebruikers, maar ook op de servers van belangrijke applicaties. Beide omgevingen vragen om een ander soort oplossing. Unikernels kunnen hierop inspelen omdat er uitgaan van een kleinere fundatie waarbij er functionaliteit kan worden toegevoegd. Het werken en samenstellen van een unikernel staat nog een vroeg stadium en moet daarom verder verkend worden. De doelstelling van dit onderzoek is om te bekijken of de applicaties die momenteel worden gemaakt kunnen werken in een unikernel. Het grootste deel van de huidige applicaties bestaan uit één grote applicatie. Een andere stroming is waarbij de applicaties worden opgesplitst in kleinere applicaties die elk een specifiek probleemdomein en -verantwoordelijkheid hebben: microservices.

2. Literatuurstudie

Deze literatuur zal eerst hypervisors en virtuele machines bekijken om een beeld te geven van de huidige oplossingen. Ook wordt er verder ingegaan waarom hierbij applicaties als monolieten of één grote applicatie wordt gebouwd. Verder wordt er gekeken naar software containers en de bekendste implementatie ervan: Docker. Het volgende hoofdstuk zal unikernels behandelen waarbij er gekeken wordt naar de soorten besturingssystemen hierbij gebruikt worden. Als voorlaatste hoofdstuk van de literatuurstudie zal er vergelijking tussen de huidige implementaties van unikernels en waar ze worden gebruikt. Als laatste hoofdstuk wordt er gekeken naar de manier hoe de applicaties worden gemaakt als geheel.

3. Methodologie

Er zal een proof-of-concept worden gemaakt waarbij er verschillende soorten applicaties(microservices en monolieten) worden opgezet op een virtuele machine, software containers en unikernels. De hoeveelheid van opzet op het vlak van scripting en benodigdheiden van tools. Unikernels staan nog in hun kinderschoenen dus we zullen bekijken hoe we van de ene opstelling naar de andere gaan en hoeveel extra werk er nodig is om deze opstelling te laten werken.

4. Verwachte resultaten

Microservices zullen beter werken binnen een opstelling van software containers en unikernels. Unikernels zullen dan weer meer problemen hebben met het opstellen van de omgevingen (verbinden van de verschillende gehelen). Monolieten

zullen gemakkelijk kunnen worden opgesteld binnen virtuele machines en software containers.

5. Verwachte conclusies

Als er meer wordt gegaan van een opstelling waarbij de componenten van een applicatie niet hecht verbonden zijn dan zal de opstelling gemakkelijker te realiseren als de opstelling bestaat uit meerdere losse componenten. De applicatie weerspiegelt als het ware de opstelling of omgeving.

Bibliografie

Alpine Linux Development Team. (n.d.). Alpine Linux | Alpine Linux. Opgehaald op 18/05/2016 van http://alpinelinux.org/.

Beloglazov, A. & Buyya, R. (2010). Energy Efficient Resource Management in Virtualized Cloud Data Centers. *2010 10th IEEE/ACM Int. Conf. Clust. Cloud Grid Comput.* 826–831.

Bovet, D. P. & Cesati, M. (2005, November). *Understanding the Linux Kernel* (3 edition). Beijing ; Sebastopol, CA: O'Reilly Media.

Bratterud, A. & Haugerud, H. (2013, December). Maximizing Hypervisor Scalability Using Minimal Virtual Machines. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)* (Vol. 1, pp. 218–223).

Bryan Cantrill. (2016, January). Unikernels are unfit for production - Blog - Joyent. Opgehaald op 18/05/2016 van https://www.joyent.com/blog/unikernels-are-unfit-for-production.

Cloud Networking Performance Lab. (n.d.). Cloud Networking Performance Lab | ClickOS | Modular VALE | Xen. Opgehaald op 18/05/2016 van http://cnp.neclab.eu/getting-started/#clickos.

Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., ... Warfield, A. (2011). Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (pp. 189–202). SOSP '11. New York, NY, USA: ACM.

Conferences, U. o. M. E. S. (1968). *Modern Methods for Solving Engineering Problems: Numerical Methods, Optimization Techniques and Simulation*.

Containers, L. (n.d.). Linux Containers (LXC). Opgehaald op 17/05/2016 van https://linuxcontainers.org/lxc/.

CoreOS. (n.d.). Rkt CoreOS. Opgehaald op 23/05/2016 van https://coreos.com/rkt/.

Danilo Sato. (2014, June). CanaryRelease. Opgehaald op 25/06/2014 van http://martinfowler.com/bliki/CanaryRelease.html.

Docker. (2016). Docker. Opgehaald op 17/05/2015 van https://www.docker.com/.

Erlang on Xen. (n.d.). Cloudozer/ling. Opgehaald op 18/05/2016 van https://github.com/cl

Galois Inc. (n.d.). The Haskell Lightweight Virtual Machine (HaLVM) source archive. Opgehaald op 18/05/2016 van https://github.com/GaloisInc/HaLVM.

García Villalba, L. J., Valdivieso Caraguay, Á. L., Barona López, L. I., & López, D. (2015). Trends on virtualisation with software defined networking and network function virtualisation. *IET Networks*, *4*(5), 255–263.

Griffin, D. & Pesch, D. (2007, July). A Survey on Web Services in Telecommunications. *IEEE Communications Magazine*, *45*(7), 28–35.

Hykes, S. (2013). *The future of Linux Containers*.

Kantee, A. (2012). *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels*. Aalto University.

Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., & Zolotarov, V. (2014). OSv—Optimizing the Operating System for Virtual Machines. (pp. 61–72).

Linux. (n.d.). Chroot(2) - Linux manual page. Opgehaald op 17/05/2016 van http://man7.org/linux/man-pages/man2/chroot.2.html.

Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D., ... Leslie, I. (2015). Jitsu: Just-in-time summoning of unikernels. *USENIX Symp. Networked Syst. Des. Implement.* 559–573.

Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., ... Crowcroft, J. (2013). Unikernels. *ACM SIGPLAN Not. 48*(4), 461.

Madhavapeddy, A., Mortier, R., Sohan, R., Gazagnaire, T., Hand, S., Deegan, T., ... Crowcroft, J. (2010). Turning Down the LAMP: Software Specialisation for the Cloud. *HotCloud*, *10*, 11–11.

Mao, M. & Humphrey, M. (2012). A performance study on the VM startup time in the cloud. In *Proc. - 2012 IEEE 5th Int. Conf. Cloud Comput. CLOUD 2012* (pp. 423–430).

Martin Fowler. (2015, May). MicroservicePremium. Opgehaald op 19/05/2016 van http://martinfowler.com/bliki/MicroservicePremium.html.

Martin Fowler. (2016, January). BlueGreenDeployment. Opgehaald op 19/05/2016 van http://martinfowler.com/bliki/BlueGreenDeployment.html.

Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., & Huici, F. (2014). ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (pp. 459–473). NSDI'14. Berkeley, CA, USA: USENIX Association.

Matthias Gelbmann. (2016, February). Ubuntu became the most popular Linux distribution for web servers. Opgehaald op 18/05/2016 van http://w3techs.com/blog/entry/ubuntu_became_the_most_popular_linux_distributio

Mirage. (n.d.). Mirage/mirage source code. bibtex: mirage/mirage_0000.

MIT. (1998). MIT Exokernel Operating System. Opgehaald op 17/05/2016 van https://pdos.csail.mit.edu/exo.html.

Mortleman, J. (2009). Security Advantages of Virtualisation. *Comput. Wkly.* 23.

Oracle. (2016a). Oracle Virtualbox. Opgehaald op 21/05/2016 van https://www.virtualbox.o

Oracle. (2016b). Solaris. Opgehaald op 17/05/2016 van https://www.oracle.com/solaris.

gliderlabs/docker-alpine. (n.d.). Opgehaald op 18/05/2016 van https://github.com/gliderla alpine.

Linux Containers - LXC - Introduction. (n.d.).

Oslo and Akershus University College & of Applied Sciences. (n.d.). Hioa-cs/IncludeOS. Opgehaald op 18/05/2016 van https://github.com/hioa-cs/IncludeOS.

Popek, G. J. & Goldberg, R. P. (1974, July). Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM, 17*(7), 412–421.

Pyke Jr., T. N. (1967). Time-Shared Computer Systems. In F. L. A. a. M. Rubinoff (Ed.), *Advances in Computers* (Vol. 8, pp. 1–45). Elsevier.

Rumpkernel. (n.d.). Rumpkernel/rumprun source code. bibtex: rumpkernel/rumprun_0000.

Satya Popuri. (n.d.). A Tour of Mini-OS Kernel. Opgehaald op 18/05/2016 van https://www.cs.uic.edu/ spopuri/minios.html.

Smith, J. E. & Nair, R. (2005, May). The architecture of virtual machines. *Computer*, *38*(5), 32–38.

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., & Peterson, L. (2007). Container-based operating system virtualization. *ACM SIGOPS Oper. Syst. Rev. 41*(3), 275.

Soundararajan, V. & Anderson, J. M. (2010). The impact of management operations on the virtualized datacenter. *ACM SIGARCH Comput. Archit. News*, *38*(3), 326.

System, C. (n.d.). Cloudius-systems/osv. bibtex: cloudius-systems/osv_0000.

Tichy, W. F. (1997, July). A catalogue of general-purpose software design patterns. In *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 23. Proceedings* (pp. 330–339).

Toby Clemson. (2014, November). Testing Strategies in a Microservice Architecture. Opgehaald op 19/05/2016 van http://martinfowler.com/articles/microservice testing/.

Unikernel Systems. (2016). Unikernels. Opgehaald op 17/05/2016 van http://unikernel.org/.

University of Cambridge. (2000). Nemesis. Opgehaald op 17/05/2016 van http://www.cl.cam.ac.uk/research/srg/netos/projects/archive/nemesis/.

Vakilinia, S., Ali, M. M., & Qiu, D. (2015, November). Modeling of the resource allocation in cloud computing centers. *Computer Networks*, *91*, 453–470.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015, September). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10ccc), 2015 10th* (pp. 583–590).

VMware. (2016). VMware Workstation. Opgehaald op 17/05/2016 van http://www.vmware.com/products/workstation/.

List of Figures

List of Tables