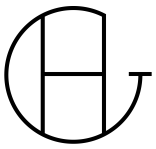


HoGent

Faculteit Bedrijf en Organisatie

Unikernels

Michiel De Wilde

Scriptie voorgedragen tot het bekomen van de graad van
Bachelor in de toegepaste informatica

Promotor:
Bert Van Vreckem
Co-promotor:
Micha Herandez van Leuffen

Instelling: Hogeschool Gent

Academiejaar: 2015-2016

2e examenperiode

Faculteit Bedrijf en Organisatie

Unikernels

Michiel De Wilde

Scriptie voorgedragen tot het bekomen van de graad van
Bachelor in de toegepaste informatica

Promotor:
Bert Van Vreckem
Co-promotor:
Micha Herandez van Leuffen

Instelling: Hogeschool Gent

Academiejaar: 2015-2016

2e examenperiode

Samenvatting

Voorwoord

Toen ik begon met programmeren had ik geen idee wat er gebeurde in de achtergrond van de computer. Ik starte met de simpele todolist-applicaties om alles te leren over programmeren. Na een tijd kwam ik een black box tegen: het besturingssysteem. Vooral om servers sneller te laten werken en de techniek erachter te leren kennen begon ik aan een zoektocht. Linux was de startplek bij uitstek. Package managers en file systems waren de eerste concepten die mij met verstomming lieten staan. Toen ik meer en meer naar infrastructuur keek begon ik termen te leren en alle handige tips om ervoor te zorgen dat je server altijd beschikbaar is. Toen een paar jaar geleden Docker voor het eerst echt vaart maakte met containers was ik verbaasd. Ik dacht eerst dat dit nooit zou werken. Na een tijd heb ik wel het licht gezien en gebruikte ik containers meer en meer. Toen er gevraagd werd om een onderwerp voor mijn thesis dacht ik meteen en wat volgens mij de volgende stap is: unikernels.

Inhoudsopgave

Hoofdstuk 1

Inleiding

Wanneer men een programma maakt dan kan men dit lokaal laten werken. Lokaal kan geen permanente oplossing zijn. Men moet dus het programma opstellen op een server en beheren. Dit is de taak van systeembeheerders in een notendop.

Voor cloud computing (Bala et al. (2000)) op de voorgrond kwam kochten bedrijven servers en werden die servers in het bedrijf zelf opgesteld. Cloud computing zorgde voor een revolutie. Het was niet langer nodig om zelf servers te hebben, ze waren beschikbaar in de cloud om te huren. Om de capaciteit die men gebruikt ten volle te benutten, kwamen er nog andere concepten naar voren, zoals: containers, microservices, unikernels.

Deze bachelorproef behandelt het onderwerp unikernels (Madhavapeddy et al. (2013)) en wat voor gevolgen unikernels kunnen hebben op de taak van de systeembeheerder. De omgeving waarin wordt gewerkt kan verschillen van situatie tot situatie. Daarom is het aantonen van de situaties waarin unikernels kunnen gebruikt worden heel belangrijk.

Deze thesis zal proberen de veranderingen te beschrijven. Dit is enkel mogelijk vanuit de huidige situatie. Sommige concepten kunnen een groot gevolg hebben terwijl die nu nog niet aanwezig zijn. Het is een blik op unikernels en gevolgen voor systeembeheerders met de huidige informatie beschikbaar.

In hoofdstuk 2 wordt bekeken hoe deze bachelorproef is uitgevoerd.

Om deze programma's te laten werken op servers maakt men gebruik van virtualisatie. Virtualisatie vormt de basis voor veel van de concepten die worden aangehaald. Virtualisatie zal belicht worden in hoofdstuk 3.

Op 21 maart 2016 werd op Pycon de eerste demo van Docker gegeven (Hykes (2013)). Docker heeft voor een aantal grote veranderingen geleid voor veel software ontwikkelaars. Dit kwam niet uit het niets. Er waren al veel initiatieven om containers naar het grote publiek te brengen. Docker heeft deze initiatieven kunnen aanwenden om de puzzel compleet te maken. In hoofdstuk 4 worden containers en Docker nader bekeken.

Het hoofdstuk 5 dat unikernels behandelt zal zich focussen op de werking van het

concept, de voordelen en de implementaties van unikernels.

Verder zullen we kijken naar de veranderingen op het vlak van architectuur van programma's en infrastructuur. Dit beperkt zich niet tot unikernels want de meeste concepten kunnen ook mogelijk zijn met containers. Hoofdstuk 6 is bedoeld om concepten aan te halen die veel op de voorgrond zullen treden wanneer containers en unikernels alomtegenwoordig zijn.

Hoofdstuk 7 zullen de experimenten behandelt worden.

1.1 Probleemstelling en Onderzoeksvragen

Unikernels zijn een een nieuwe stroom binnen het landschap van besturingssystemen. We hebben al aangehaald dat containers een populaire werkwijze is om software te maken en programma's op te stellen. Unikernels gaat nog een stap verder. De systeembeheerders zullen een paar veranderingen tegenkomen bij het opstellen en onderhouden van programma's.

De vraag is welke veranderingen er zich zullen voordoen, wanneer unikernels meer gebruikt worden. Zullen de competenties van de systeembeheerder veranderen? Wordt het opzetten van applicaties eenvoudiger of niet? We kunnen wel spreken over de opvolger van containers maar is deze al werkbaar in de toekomst? Wat is de impact op beveiliging, meer bepaald aspecten als beschikbaarheid, autorisatie, integriteit en vertrouwelijkheid van gegevens?

Hoofdstuk 2

Methodologie

Het begrip unikernel vraagt om een uitgebreide theoretische kennis van huidige besturingssystemen en virtualisatietechnologieën.

Voor veel van deze concepten goed te begrijpen werd er eerst een literatuurstudie uitgevoerd. Artikels en andere informatie over unikernels was simpel te vinden door de volgende website (Unikernel Systems (2016)). Veel over virtuele machines was te vinden in een paar thesissen van de vorige jaren. De kennis over containers werd voornamelijk gevonden tijdens mijn stage bij Wercker. Een paar boeken over containers, met centraal onderwerp Docker, gaven meer inzicht in containers en hun use cases.

Door het literatuuronderzoek konden de eerste hoofdstukken over virtuele machines, containers en unikernels geschreven worden. De thesis focust niet alleen op de mogelijkheden die unikernels hebben. Ook de veranderingen voor systeembeheerders moet bekeken worden.

Microservices en Immutable Infrastructure waren twee gegevens die bekend aan het worden zijn binnen de wereld van devOps en software development. Dit leidde tot een hoofdstuk die hun verband aantoonde met unikernels en containers. De revolutie voor software development en devOps door deze soort technologieën zou een duidelijk beeld geven op de verandering voor systeembeheerders.

Daarna was het tijd voor experimenten. Als eerste was het vinden van een implementatie van een unikernel belangrijk vooraleer we konden beginnen met experimenten. Het kiezen van de meest populaire omwille van ongegronde redenen zou niet leiden tot een goed onderzoek. Daarom werden er een aantal implementaties bekeken en met elkaar vergeleken.

Hoofdstuk 3

Virtualisatie

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

Vroeger was de tijd dat je een computer kon gebruiken beperkt. Vooral bij de eerste computers had men problemen om programma's en concepten uit te werken. Dit lag vooral aan de tijd dat je kon werken aan de computer en vele andere mensen wouden de computer ook gebruiken. Een voorbeeld van het ontwikkelen van een programma in die tijd was de volgende: "De broncode van het programma werd ingegeven en in een wachtrij geplaatst. Pas een bepaalde tijd later kon men de resultaten van het programma bekijken. Fouten in het geschreven programma zorgen voor een groot tijdsverlies."

Eén van de grootste bijdrage tot de ontwikkelsnelheid van programma's is de lengte van de feedbackcyclus: hoe snel kan een programma getest worden wanneer er een verandering gebeurd. Als een paar minuten moet worden gewacht op het testen van een kleine verandering, dan is dit niet ideal. Dit leidt tot een verlies van tijd en dus geld.

Timesharing werd uitgevonden om het verlies van tijd te beperken. Bij timesharing konden de gebruikers inloggen op een console en zo de computer tegelijkertijd gebruiken. Dit was een technische uitdaging. Elke gebruiker en zijn programma's bevinden zich binnen een bepaalde context. De computer zou van de ene context naar de andere moeten kunnen veranderen. Timesharing en verschillende gebruikers op één computer zou de basis vormen voor het moderne besturingssystee. Eén van de moeilijkheden van timesharing is de isolatie van twee verschiilende processen. De twee processen moeten zich bevinden binnen een verschillende context. Deze contexten mogen niet met elkaar elkaar in contact komen of niet elkaar beïnvloeden.

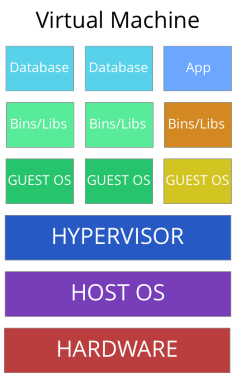
Doorheen de tijd werden computers krachtiger en programma's konden niet langer de middelen van de computer benutten. Dit zorgde voor de creatie van virtuele mid-

delen of virtual resources. Om deze virtuele middelen te voorzien moet men bepaalde delen van de computer gaan virtualiseren. Dit kan voorkomen onder verschillende vormen zoals hardware virtualisatie en virtualisatie van het besturingssysteem.

Het concept van virtualisatie deelt een aantal overeenkomstigheden met timesharing. De computer wordt opgedeeld in verschillende delen bij virtualisatie en bij timesharing gaan we de computer opdelen in contexten. De verschillende delen bij beide concepten moeten ook geïsoleerd zijn van elkaar.

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

Virtuele machines gaan een computer emuleren. Figuur 3.1 toont de structuur van een virtuele machine. Het laat toe om een besturingssysteem te gebruiken, wanneer de hardware van de fysieke computer dit niet toelaat. Een virtuele machine kan ook 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. De fysieke computer zal verder naar verwezen worden als de host machine. Guest is de naam dat we geven 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.



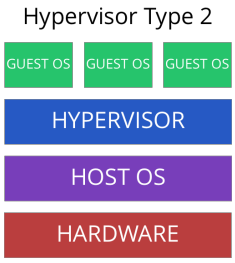
Figuur 3.1: structuur van een virtuele machine

3.1 Hypervisor

De hypervisor 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, ... Elke virtuele machine die zich bevindt op de host machine zal dan gebruik maken van een gedeelte van deze middelen. Doordat virtualisatie alomtegenwoordig geworden is in datacenters (Soundararajan and Anderson (2010)) heeft dit 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. De volgende twee segmenten zullen deze twee verschillende types uitleggen.

3.1.1 Hosted Hypervisors

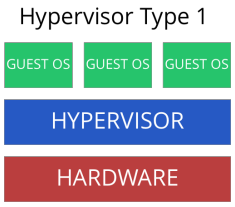
Als eerste zullen we de hosted hypervisor of type 2 hypervisor behandelen. Figuur 3.2 toont de structuur van een hosted hypervisor. De hosted hypervisor bevindt zich op het besturingssysteem van de host machine en heeft geen directe toegang tot de hardware. Het is wel compleet afhankelijk voor het host besturingssysteem om zijn taken uit te voeren. Als er problemen optreden in het besturingssysteem van de host zijn er ook problemen bij de hypervisor en daarop volgend de guests. De hele structuur is enkel zo sterk als de host.



Figuur 3.2: structuur van een hosted hypervisor

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

3.1.2 Bare-metal Hypervisors



Figuur 3.3: structuur van een bare-metal hypervisor

Type 1, bare-metal, embedded of native hypervisor bevindt zich rechtstreeks op de hardware. Figuur 3.3 toont de structuur van een bare-emtal hypervisor. De voornaamste taak van de hypervisor is het beheren en delen van hardware middelen. Dit maakt de hardware hypervisor kleiner in omvang dan de hosted hypervisor. De hypervisor heeft niet het probleem zoals de hosted hypervisor dat er grote problemen kunnen liggen bij het besturingssysteem van de host omdat er geen besturingssysteem is.

Er is een laag minder in de structuur dus dit betekent dat er minder instructies moeten uitgevoerd worden bij een bepaalde handeling en dit geeft een beter performantie. Omdat er geen problemen kunnen zijn met het host besturingssysteem kunnen we aannemen dat het stabieler. Wanneer het host besturingssysteem faalt bij een hosted hypervisor dan zullen de guests ook falen.

Een paar voorbeelden van bare-metal hypervisors zijn VMware ESXi (VMware (2016a)) en Xen (Xen Project (2016)).

3.2 Operating System-level Virtualization

Naast hardware virtualisatie kunnen we ook een besturingssysteem virtualiseren. Bij deze toepassing van virtualisatie worden de mogelijkheden van de kernel van het be-

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

De verschillende user spaces maken gebruik van CPU, geheugen en netwerk van de host. Elke user space heeft zijn eigen configuratie omdat de user spaces op zichzelf staan en geïsoleerd zijn van de andere user spaces. Dit geeft ook eveneens de beperking dat guests geen besturingssysteem kunnen hebben dat niet overeenkomt met het host besturingssysteem.

Tegenover hardware virtualisatie zal besturingssysteem virtualisatie minder gebruik maken van middelen omdat er maar één besturingssysteem is en eveneens het delen van het besturingssysteem. Dit geeft voordelen bij de perfomantie.

Deze user spaces worden ook wel containers genoemd.

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

Besturingssysteem virtualisatie is vooral bekend geworden door Docker vanaf 2013 (Hykes (2013)). In het volgende hoofdstuk gaan we verder in op containers en Docker.

Hoofdstuk 4

Containers

4.1 Containers versus virtuele machines

Het heeft een tijd geduurd vooraleer containers op de voorgrond treden. Het maken van applicaties werd meestal op de eigen machine gedaan omdat het opzetten van een werkomgeving soms te moeilijk was. Er was natuurlijk vagrant die het stukken gemakkelijker maakte maar dan nog voelde het aan als of er nog iets beter aankwam.

Lange tijd werd vagrant aangeraden omdat het opzetten van een werkomgeving veel tijd in beslag nam en de meeste applicaties eerder een monolitische structuur hadden. Hierbij bedoelen we een grote applicatie die alles doet tegenover verschillende applicaties die elk 1 ding doen.

Containers was het antwoord op veel van de moeilijkheden van de virtuele machine: het beter gebruiken maken van resources, eenduidige ontwikkelingsomgeving en vlugger kunnen ontwikkelen van applicaties. Dus we hebben twee mogelijkheden om applicaties te maken en te laten werken. Ofwel gebruiken we virtuele machines als de kleinste eenheid ofwel containers.

De container gebruikt resources van het besturingssysteem van de host. Dit zorgt ervoor dat een container vlugger kan gestart worden want het besturingssysteem moet niet opgestart worden. Bij virtuele machine moet men wachten tot het besturingssysteem opgestart is.

De omvang van virtuele machines tegenover containers is al besproken hiervoor. Een gevolg hiervan is dat men meerdere containers naast elkaar kan laten werken. De virtuele machine waarop de docker container werken kunnen dan beslissen hoeveel resources er aan wie worden gegeven.

Voor containers was het opzetten van een complexe applicatie met tientallen dependencies een heel moeilijk gegeven. Bij een container moet je gewoon de image downloaden en gebruiken. In deze container is alle configuratie al aanwezig en kan men gewoon starten zonder veel tijd te verspelen aan het configureren en installeren

van de applicatie en het besturingssysteem.

Omgekeerd kan men ook denken dat de developers niet meer moeten nadenken over het ondersteunen van meerdere systemen. 1 Container image kan gebruikt worden voor iedereen en als men veranderingen wil aanbrengen dan kan men verder op de container bouwen.

Als men wilt schalen met virtuele machines dan moet je meerdere besturingssystemen opzetten die worden beheert door een hypervisor. Elk besturingssysteem heeft een applicatie en zijn dependencies.

Bij containers delen ze het besturingssysteem en worden beheert door een container engine die zich bevindt op het niveau van het besturingssysteem. Ze delen ook een de kernel van de host en daarbij kan er ook voor gezorgd worden dat gemeenschappelijke dependencies gedeeld kunnen worden over meerdere applicaties. De container engine vervuld dezelfde functie als de hypervisor bij virtuele machines.

Veiligheid kan een probleem zijn bij containers omdat ze nog niet lang in productie worden gebruikt. Hypervisor zorgen voor extra veiligheid bij virtuele machines omdat ze battle-tested zijn. Ze worden al gebruikt voor lange tijd door grote bedrijven die beveiliging hoog in het vandaag dragen.

4.2 Docker

Onderdelen van het begrip container bestonden al onder diverse vormen. FreeBSD had jail(1998). Google was begonnen met het ontwikkelen van cGroups voor de linux kernel. Het Linux Containers projects bracht veel van deze technologieën samen om te zorgen dat containers een realiteit konden worden.

Docker was het bedrijf dat de al de delen samenbracht onder een mooie interface en uitgebreid ecosysteem. Het werd veel gemakkelijker om containers te maken en distribueren. Docker zorgde ervoor dat veel van moeilijkheden van containers verdwenen of veel kleiner werden. Door veel van de componenten van het ecosysteem open te stellen konden ze rekenen op de steun van vele developers. Deze steun kon zijn in de vorm van het melden van bugs of het uitbreiden of verbeteren van huidige componenten.

Er waren andere formaten dan Docker zoals Rocket die het mogelijk maakte voor containers te maken. Maar door hun ecosysteem en de hulp van de open source gemeenschap hebben zijn ze de standaard geworden voor om containers te maken en te verspreiden.

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

Hoofdstuk 5

Unikernels

5.1 Inleiding

Unikernels bestaan al lang onder verschillende vormen. Je kan dit zeggen van de meeste technologieën die zorgen voor grote veranderingen: eerst waren een paar projecten die uiteenlopende standaarden gebruikten en na verloop van tijd kwamen die samen. Het experimenteren met nieuwe software of in het algemeen met ideeën zorgt voor innovatie.

De eerste implementaties komen we tegen op het einde van de jaren 1990. Exokernel werd ontwikkelt door MIT en wou ervoor zorgen dat er zo weinig mogelijk abstractie de developers op te leggen en dat ze zelf over de abstractie moeten beslissen. Nemesis werd dan weer vanuit University of Cambridge ontwikkeld. Zij hadden eerder multimedia als het doel in hun achterhoofd.

5.2 Kernel

De kernel is het programma dat zich centraal bevindt in de computer. 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 zo een belangrijke rol vervult in de computer is veel van het geheugen van de kernel beveiligd zodat andere applicaties geen veranderingen kunnen aanbrengen. Als er iets fout zou gaan met de kernel dan heeft dit rechtstreeks gevolgen op het besturingssysteem. Al de handelingen die de kernel uitvoert bevinden zich in de kernel space. Daartegenover hebben we alles wat de gebruiker uitvoert gebeurd 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 en tevens de computer onstabiel en niet veilig zijn. De kernel voert nog andere taken uit zoals memory management en system calls.

Als een applicatie wordt uitgevoerd dan bevind die zich binnen de user space. Om het programma in werkelijkheid te kunnen uitvoeren moet men toestemming vragen

aan de kernel om deze instructies van het programma realiseren. Deze instructies moeten worden nagegaan of ze wel veilig zijn. Soms spreken we ook van memory isolation waarbij de user space en kernel space niet rechtstreeks met elkaar kunnen communiceren. Dit is voor nog veiliger te zijn. Om een applicatie uit te voeren moet er veel communicatie gebeuren tussen de onderdelen. Deze onderdelen kunnen zo laag als de hardware gaan tot de applicatie zelf.

5.3 Library besturingssystemen

Het meest gebruikte besturingssysteem waarop een web applicatie zich bevindt is een Linux besturingssysteem. Het meest bekende besturingssysteem voor servers is Ubuntu. Daartegenover hoor je ook dat Ubuntu zich wil begeven naar het grote publiek. Een gratis alternatief voor Windows zowaar. De editie dat je gebruikt voor je web applicaties is niet helemaal dezelfde als het windows-alternatief.

Maar als we er toch bij stilstaan dan zijn er grote delen van een besturingssysteem die we niet nodig hebben voor grote applicaties. Doorheen de geschiedenis zijn er veel verschillende onderdelen nodig geweest om goed gebruik te kunnen maken van een besturingssysteem. Na een tijd zijn sommige onderdelen niet meer nodig. Het verwijderen van deze onderdelen brengt moeilijkheden mee. Sommige gebruikers hebben nog juist dat onderdeel nodig terwijl andere het niet meer nodig hebben. We kunnen ook niet zomaar delen verwijderen. Dit kan ervoor zorgen dat andere delen niet meer werken.

Er is een trend binnen containers dat je een container moet gebruiken die alles heeft wat je nodig hebt maar niets meer. Hoe kleiner de container hoe beter. Alpine is een Linux besturingssysteem dat zeer klein is (5 MB) maar beschikt over een package repository die zeer uitgebreid is. Dit maakt het het ideale besturingssysteem voor containers. Je kan starten met een kleine basis en alle onderdelen toevoegen die je nodig hebt. Dit zorgt voor betere performantie en een kleinere container.

Time-sharing is een onderdeel dat alleen nog maar nodig is in uitzonderlijke gevallen. Time-sharing zorgt ervoor dat de systeem en hardware componenten van de gebruiker worden afgeschermd. Een gebruiker die een programma liet werken zou een andere programma gestart door een andere gebruiker kunnen laten stoppen door een bepaalde instructie uit te voeren. Dit probleem is bijna niet meer voorkomend omdat hardware zo goedkoop is geworden en iedereen zijn eigen computer heeft.

Niet alleen hebben sommige delen geen nut meer maar sommige delen zullen ervoor zorgen dat de performantie van een besturingssysteem gehinderd wordt voor bepaalde taken. Het is dezelfde analogie als het gebruiken van een hamer voor alle klusjes die je moet doen. Het is niet omdat een hamer goed is om spijkers in hout te slaan dat je het moet gebruiken voor een boom om te zagen.

Als we kijken naar het voorbeeld van Alpine binnen containers kunnen we het

dezelfde weg opgaan. Library besturingssystemen neemt dit nog een stap verder. We starten dus van een nog lagere basis en we voegen alleen de delen toe die we nodig hebben. Library duidt op functionaliteit die je kan gebruiken in verschillende programma's of contexten.

Als we een web applicatie gaan bouwen dan moeten we kunnen communiceren met internet. Hiervoor bestaan verschillende netwerkprotocollen die je kan gebruiken. Wij hebben TCP nodig om te communiceren met internet. Bij alledaagse besturingssystemen zoals Ubuntu is dit al aanwezig. Omdat we starten vanaf nul moeten we deze zelf implementeren. Gelukkig zijn er libraries die we hiervoor kunnen gebruiken. Het verschil zit hem in de grote van het uiteindelijke besturingssysteem en zijn grotere varianten. De libraries en de applicatie worden dan gecompiled met de configuratie. Als resultaat heb je dan 1 binary die rechtstreeks op een hypervisor of de hardware kan werken. Je compiled de applicatie voor de omgeving waar hij zal werken en enkel daar zal hij werken. Dit is een verschil tegenover een container die minder afhankelijk is van de omgeving.

Doordat we zelf een applicatie opbouwen met enkel wat de applicatie nodig heeft zorgt dit voor een kleinere attack surface. De code dit uitenindelijk terecht komt op de server is veel minder groot.

5.4 Single Address Space

Wat een unikernel nog uniek maakt is dat de kernel geen concept heeft van een user space en kernel space. Alle processen bevinden zich dus in dezelfde omgeving. Dit zou problemen geven bij traditionele besturingssystemen. Maar we compileren de applicatie en zijn libraries en controleren of dat er zich geen problemen kunnen voordoen. Een voordeel dat we kunnen aanhalen is dat er geen communicatie moet gebeuren tussen de user space en de kernel space omdat ze niet bestaan. We zullen sowieso in problemen lopen wanneer we meerdere applicaties naast elkaar laten lopen of applicaties met veel functionaliteit. Dit is ook niet de manier waarop men unikernels moet gebruiken. In de laatste secties van dit hoofdstuk zullen we dit concept nog verder aanhalen.

5.5 Veiligheid

Het grootste verkooppunt van unikernels is veiligheid. Vulnerabilities en veiligheidsrisico's kunnen zorgen voor grote schade. De gebruikers vertrouwen hun informatie aan ons en wanneer dit vertrouwen geschaad word is dit niet goed. Web applicaties moeten veilig zijn maar andere sectoren zoals de banksector maken een obsessie van veligheid.

Doordat cloud computing zo hard op de voorgrond treedt, komen er veel meer applicaties op het internet. Meestal zijn deze appliacties niet zeer goed beveiligd. Een

bijdrage daarbij is zeker ook dat de drempel veel lager wordt.

Veiligheid is soms zeer moeilijk te vinden wanneer we grote besturingssystemen gebruiken als basis voor onze applicaties. Deze besturingssystemen hebben zodanig veel onderdelen dat het heel moeilijk wordt om deze allemaal te controleren. Daar komen ook nog de combinaties van deze onderdelen bij.

Containers gebruiken een besturingssystemen als basis en door dit gebruiken ze ook de Linux kernel. De Linux kernel is niet resistent tegen bepaalde onveiligheden. Daartegenover starten unikernels met een heel minimaal gegeven en voegen toe wat de applicatie nodig heeft. Het schrijven van een exploit is ook veel moeilijker voor een unikernel omdat elke unikernel anders in zit.

Wanneer je bijvoorbeeld toegang hebt tot een container kan je gemakkelijk toegang krijgen tot de shell van de container en vanalle kattenkwaad uithalen.

Wanneer een hacker binnendringt op een virtuele machine die een traditionele stack heeft dan kan hij tools installeren door de package manager te gebruiken en andere commando's om meer informatie te verzamelen. De server kan misschien geinfecteerd worden met een virus. Dit scenario is veel moeilijker bij unikernels omdat veel van commando's weggenomen zijn en er geen package manager aanwezig is. De hacker zou al heel veel werk moeten leveren om de unikernel te infecteren omdat hij zelf de vulnerability zou moeten schrijven. Dan blijft er enkel nog de hypervisor over om te misbruiken. De onveiligheden van hypervisors zijn veel moeilijker uit te buiten omdat ze al een paar decennia in gebruik zijn van veel grote bedrijven.

Uitbreiden over TLS?

5.6 Just in Time

Network latency is een groot probleem wanneer de meeste services die u aanbied zich in de cloud bevinden. Dit maakt de applicatie zeer afhankelijk is van de server. Wanneer de connectie met de server wegvalt kan de applicatie in uitzonderlijke gevallen nog werken of helemaal niet meer. De applicatie is rechtstreeks afhankelijk van de connectiviteit met de server. Zelf een slechte connectie kan zorgen voor problemen. De oorzaak van een slechte connectie kan een oorzaak van de server of de applicatie. We gaan er vanuit dat de applicatie en server goed geschreven zijn en dat er geen grote problemen liggen bij de infrastructuur.

De locatie van je servers tegenover de locatie van je gebruiker kan zorgen voor problemen. Je infrastructuur verspreiden over verschillende datacenters en nadenken over hun locatie kan hierbij een goede actie zijn. Maar wanneer je resources verbruikt in een datacenter dat weinig gebruikt wordt kan dit duur uitkomen. Het verbruik van de resources aanpassen aan de load van de server is een stap die al genomen wordt door vele cloud providers. Dan nog moeten er nieuwe instanties van jouw applicaties of servers worden opgestart. Dit alles kan meer dan een halve minuut duren. Sommige

soorten applicaties, zoals IOT en anderen, hebben constante connectiviteit nodig. Dit kan van zeer groot belang zijn. Unikernels kunnen hierbij helpen.

Het opstarten van een virtuele machine en alles in orde brengen om te werken kan 1 minuut of langer duren. Bij containers wordt dit veel minder omdat er geen besturingssysteem van nul moet worden opgestart. Unikernels neemt dit nog een paar stappen verder omdat men gewoon een binary moet uitvoeren op de hypervisor of hardware. De opstarttijd zal meestal minder dan een seconde innemen. Dit geeft ons een betere manier om de capaciteit aan te passen aan de load. Het opstarten van een unikernel kan sneller gebeuren dan de request naar de server toe. Kosten zullen hierdoor lager liggen.

5.7 Andere voordelen

De binary die gemaakt wordt wanneer de unikernel gecompileerd word zal zeer klein zijn. Omdat verschillende drivers zelf geïmplementeerd worden. Er moet ook niet in dezelfde mate als het besturingssysteem rekening gehouden worden met andere drivers. De hypervisor zorgt ook voor een stabiele interface en deze zorgt ervoor dat we geen uitgebreide drivers moeten schrijven om te zorgen voor compatibiliteit voor de verschillende hardware apparaten.

5.8 Productie

Veel van de commentaar op unikernels komt van de onmogelijkheid om in productie te bekijken wat er fout aan het gaan is bij een unikernel. Voor de developers is het soms gemakkelijker om een applicatie aan te passen in productie om te zien of een bepaalde bug wordt verholpen. Sowieso is dit geen best practice. Er moet uitgebreid getest worden in development cycle om fouten/bugs tegen te gaan. Wanneer er dan toch iets fout gaat in productie dan zal men de situatie proberen na te bootsen en dan applicaties aan te passen om zo de fout/bug op te lossen. Het terugzetten van een oudere versie van de applicatie kan helpen om de gebruikers geen ongemak te voorzien en tevens meer tijd te hebben om bepaalde bugs op te lossen.

5.9 Hedendaags gebruik

Unikernels kunnen gebruikt worden in uiterst uitzonderlijke use cases momenteel. Hetzelfde zal je vinden bij containers vooraleer zij op de voorgrond treden. Elke technologie zal dit ondergaan omdat iets gemakkelijker maken en veralgemenen tijd kost. Maar we hebben gezien uit de geschiedenis van besturingssystemen dat we sommige concepten

niet moeten blijven gebruiken. Als unikernels een groter publiek kunnen dienen zonder hun voordelen te verliezen dan zou het een groot succes kunnen worden.

In het volgende hoofdstuk zullen we bekijken welke concepten en tools de systeembeheerder zal tegenkomen in een toekomst met unikernels. Veel van deze concepten komen we ook nu tegen bij containers.

Enkele voorbeelden van moderne implementaties van unikernels zijn: MirageOS, OSv, HaLVM, LING, ClickOS, Rumprun en IncludeOS.

In het volgende deel zullen bekijken op wat voor manieren we het best unikernels kunnen gebruiken en wat voor gevolgen dit heeft op de architectuur van onze systemen.

Hoofdstuk 6

Architectuur in een wereld van unikernels

6.1 Microservices

De laatste 20 jaar werd er meestal 1 grote applicatie gemaakt die alle functionaliteit op zich nam. Deze soort van applicaties zijn simpel te maken tot een bepaalde punt. Een applicatie die alle zeer groot is en veel functionaliteit heeft noemen we een monoliet of monolith in het Engels. De architectuur waarin deze soort applicaties voorkomen noemen we monolithic architecture.

Het probleem bij monolithic architecture is dat het in begin simpel is om functionaliteit toe te voegen. Maar na een tijd wordt de applicatie zo groot dat de complexiteit snel stijgt. Nieuwe developers die de applicatie niet kennen moeten een paar weken de applicatie verkennen. Dan pas kunnen ze beginnen met nieuwe functionaliteit te schrijven. Schaalbaarheid is een moeilijkheid waar men ook tegenloopt na een tijd. Sommige componenten van een applicatie moeten meer kunnen verwerken dan andere of hebben meer nog nodig van een bepaalde resource. Het schalen van deze componenten is enkel mogelijk door een nieuwe instantie toe te voegen van de hele applicatie.

Wanneer een bepaald deel van een applicatie een bottleneck wordt dan bestaat de mogelijkheid om deze te herschrijven en te optimaliseren. Maar soms moet een deel van een applicatie gewoon veel meer kunnen verwerken dan een ander deel.

Iets waar veel tegen gezondigd wordt bij het programmeren is het opofferen van de kwaliteit van code om functionaliteit vlugger klaar te hebben. Een monoliet heeft een grotere kans voor om dit tegen komen dan andere soorten architectuur.

Door de opkomst van containers is een ander soort architectuur in het oog gesprongen: microservices architecture. Hierbij gaan we conventionele applicaties opsplitsen in verschillende services. Deze microservices kunnen we bekijken als deelapplicaties die

één verantwoordelijkheid hebben.

Microservices is een concept dat al langer bestaat maar dit is de uitgesproken oplossing voor dit fenomeen. De applicatie opsplitsen in componenten met 1 verantwoordelijke zorgt ervoor dat het veel beter te schalen is. Als je een applicatie hebt met al de functionaliteit in dan zou je een nieuwe virtuele machine moeten opzetten. Dit is niet schaalbaar. Bij een microservices architecture zal je een nieuwe microservice opstarten. Dit vraagt natuurlijk wel om een nieuwe manier van werken.

Bij microservices zal men eerder de topologie moeten kennen van het probleem. Starten met het gebruiken van microservices architecture wanneer men het domein niet goed kent vraagt om problemen. Een monolithische architecture zal beter kunnen reageren op dit probleem. Als men later het domein kent kunnen we een microservices architecture gebruiken. Hiervoor moet men de monoliet modulair schrijven. Modulairiteit is een vanzelfsprekende bouwsteen binnen programmeren dus we kunnen hiervan uitgaan.

De microservices moeten goed met elkaar werken en er moet consistentie bewaard worden over alle microservices.

Het voordeel van microservices architecture is dat de microservices apart van elkaar gescheiden zijn. Het gebruik van een nieuwe technologie of framework is opeens niet zo een groot probleem meer.

Deployment is een relatief groot probleem bij monolithic architecture. Bij deze soort architecture moet men de applicatie vervangen op een moment dat ze niet veel gebruikt word. Tijdens de nacht of het weekend is een voorbeeld van zo'n project. De moeilijkheid van dit soort deployements zorgt ervoor dat grote releases veel meer voorkomen. Het testen van een applicatie kan maar zoveel doen en bijna altijd zijn er nog grote ongemakken die voorkomen in productie. Het vinden van deze bugs kan eem moeilijk gegeven zijn. Microservices architecture laat toe om frequenter deployments te doen want men verandert maar een klein deel van het geheel. Ongemakken zijn veel vlugger te vinden omdat het probleem beperkt is tot het deel dat gedeployed is.

Het devOps team krijgt ook meer werk omdat ze nu tientallen microservices moeten beheren in plaats van één grote applicatie. Het beheren van deze microservices en hun logs wordt een belangrijk doel voor het devOps team. Men kan zeggen dat deze microservices veel simpeler zijn om te verstaan omdat hun functionaliteit beperkt is. Sommige halen aan dat het de complexiteit die we niet meer tegenkomen in de applicatie nu terechtkomt bij het samen laten werken van de microservices.

Een devOps cultuur waarbij iedereen betrokken is met het maken en opzetten van software wordt bijna verplicht. Veranderen van cultuur kan heel moeilijk zijn in gevestigde bedrijven. Microservices architecture is een puzzel is die in elkaar moet passen. Je moet goed doordacht te werk gaan en je visie naar de toekomst toe ook in het achterhoofd houden.

6.2 Immutable Infrastructure

Unikernels vraagt ook een verschuiving van de manier dat we over infrastructuur denken. Immutable infrastructure is een opkomende gedachtengang. We gaan geen enkele verandering aanbrengen aan applicaties die zich in productie bevinden. Als er een probleem is dan gaan we een nieuwe unikernel maken en deze in productie laten treden terwijl de andere wordt neergehaald. Je kan dit al zien bij sommige cloud providers die werken door middel van een git push om de voormalige versie te vervangen. Dit zorgt voor een betere veiligheid en de hele cyclus van development naar productie wordt veel kleiner.

We gaan dan ook meer naar het model van rolling updates. Rolling updates wordt al gebruikt bij sommige Linux distributies. Hierbij gaan we niet spreken van grote updates waarbij er een lijst van functionaliteit wordt uitgebracht op 1 moment. We gaan updates op gelijk welke tijd uitbrengen zonder de distributie te breken. Er gaan dus meer kleinere updates zijn en veiligheidsmaatregelingen kunnen veel vlugger getroffen worden.

We moeten opletten dat we niet zonder nadenken nieuwe systemen opzetten dit kan leiden tot een hoop die heel moeilijk op te ruimen.

Een nieuw gegeven is ook dat we infrastructuur meer zullen benaderen zoals we programmeren. Dit doen we door de infrastructure uitermate te testen en herhaalbare patronen te gebruiken. Ook zullen we de configuratie van servers bijhouden in version control. Door dit te doen kunnen we een beeld krijgen wat er gebeurd met de infrastructuur door de tijd heen. Dit geeft de devOps de mogelijkheid om problemen terug te leiden naar 1 verandering. Het is belangrijk dat er steeds kleine veranderingen gebeuren zodat we problemen gemakkelijker kunnen herleiden naar 1 oorzaak. Gebruik maken van version control zal niets uitmaken wanneer je tientallen veranderingen bundelt in 1 verandering.

Een trend dat we zeker zien en dat zich zal blijven doorzetten is het lenen van programmeer principes in de wereld van infrastructuur. Dit komt doordat het programmeren het meest geëvolueerd is door de jaren en ook met grotere problemen meestal heeft moeten werken.

Hoofdstuk 7

Experimenten

7.1 Vergelijking Implementaties unikernels

7.1.1 Inleiding

In het hoofdstuk over unikernels zijn er een aantal voorbeelde van moderne implemenaties van unikernels aangehaald. Uit deze implementaties zullen er twee worden gekozen om het ecosysteem en mogelijkheden van deze implementatie te bekijken. Hierbij zijn er veel verschillende criteria die we kunnen aanhalen. In volgende sectie zullen we de criteria beschrijven en hoe we ze een score geven. Eén specifieke implementatie die zich richt op één taal en andere die meerdere omgevingen en talen ter beschikking heeft. Hierdoor kunnen we het verschil tussen een algemene en specifieke implementatie zien.

7.1.2 Criteria

Implementatie programmeertaal In welke programmeertaal is de implementatie geschreven? De meeste implementaties zijn in C/C++ geschreven. Dit komt vooral door het feit dat veel developers die bezig zijn met unikernels vanuit een achtergrond met besturingssystemen komen. De de facto programmeertaal voor een besturingssysteem is C/C++.

Hypervisors Hypervisors zijn een groot deel waarom unikernels gekozen worden. De meest gebruikte unikernels zijn beschikbaar op een aantal hypervisors. Sommige unikernels specializeren zich in één hypervisor. Dit is om gebruik te maken van de specifieke mogelijkheden van de hypervisor en zo de sterktes daarvan ui te buiten.

Ondersteunde programmeertaal Misschien is dat wel de grootste factor bij het kiezen van een implementaite. Het is gemakkelijker om een bepaalde implementatie

te gebruiken wanneer je kan kiezen uit een aantal programmeertalen. Als je een microservice wilt schrijven in een andere programmeertaal, zou het wel handig zij om bij dezelfde implementatie te blijven.

Documentatie Wanneer een developer vlug van start kan gaan met een implementatie, zorgt dit voor een betere ervaring voor de developer. Niet alleen de basis is van belang maar ook hoe je meer geadvanceerde situaties moeten worden aangepakt. Documentatie is volgens velen het meest belangrijke onderdeel van een open-source project.

Tools Bestaat er een ecosysteem tools dat we kunnen gebruiken voor de implementatie. Platformen en ondersteuning van cloud providers kunnen hierbij passen. De mogelijkheid om de applicatie te debuggen is een zeer handig gegeven.

Packages Dit is tweeledig. Enerzijds willen we de normale package manager voor een programmeertaal blijven gebruiken. Als we hiervan afwijken dan kan het moeilijk wanneer we bugs en andere tegenkomen. Ook aan de unikernels zijde kunnen we spreken van packages. Packages zullen we dan eerder bekijken als drivers om bepaalde functionaliteit aan de implementatie te geven. Bijvoorbeeld een bepaald protocol zoals TCP.

GitHub stars Hieruit kunnen we opmaken hoeveel mensen het in de gaten houden en hoe populair is.

Naam	Taal implementatie	Hypervisor	Ondersteunde talen	Docs	Tools	GitHub
ClickOS	C/C++	Xen	bindings	5	7	243
HaLVM	Haskell	Xen	Haskell	8	6	665
LING	C/Erlang	Xen	Erlang	4	6	523
Rumprun	C	hw, Xen, POSIX	C, C++, Erlang, Go, ...	9	8	469
MirageOS	OCaml	Xen	OCaml	9	8	657
IncludeOS	C++	KVM, VirtualBox	C++	6	6	1341
OSv	C/C++	KVM, Xen, ...	JVM	10	9	2121

De inhoud van de bovenstaande tabel wordt verder uitlegd in de volgende sectie.

7.2 Implementaties van moderne unikernels

7.2.1 ClickOS

Implementatie programmeertaal : C/C++

Hypervisors : Xen

Ondersteunde programmeertaal : ondersteund door bindings

Documentatie : 5

Tools : 7

GitHub stars : 243

Deze implementatie van een unikernel komt van Cloud Networking Performance Lab.

De applicaties waarvoor ClickOS wordt voor gebruikt zijn middleboxes. Een middlebox is een netwerk applicatie dat netwerktrafiek kan omzetten, filteren, inspecteren of manipuleren. Voorbeelden hiervan zijn firewalls en load balanceres. We starten met een modulaire software router waarop je onderdelen kan toevoegen. Deze werkt op enkel op MiniOS. MiniOS is beschikbaar bij de source van de Xen hypervisor.

Vroeger zat de laag die alles regelde met de netwerk trafiek eerder bij de hardware, maar je kan dus ook je eigen implementatie schrijven om veel van functionaliteit van de hardware over te nemen. Dit zorgt voor een implementatie die veel efficiënter is. De resulteerde unikernel is rond de 6MB groot, starten op rond de 30 milliseconden en kunnen snel gebruikt worden 45 microseconden. Bij snel gebruikt bedoel ik dat ze worden toegevoegd om netwerk trafiek te verwerken.

De use cases waarbinnen ClickOS kan gebruikt worden zijn redelijk beperkt. Dit is geen slecht gegeven, maar het is beter als we een implementatie gebruiken die meer algemeen te gebruiken is. Als je geen gebruik wilt maken van ingebouwde netwerk functionaliteit van de hardware dan is ClickOS de uitgesproken keuze.

Hun documentatie over ClickOS is verwarrend. Het was een tijd zoeken welk programmeertalen ze ondersteunen. Uiteindelijk werd gevonden dat ze swig gebruiken voor hogere talen te integreren met C. Swig maakt een wrapper die de C/C++ kant verbind met een hogere taal.

Cosmos is de toolchain dat gebruikt word. Het maakt het simpeler om te developen en interactie met de gebruiker te verbeteren. Vooraleer bestond cosmos, zou je zelf de bindings moeten definiëren. Dit is wel wat werk.

ClickOS verwijst naar zijn packages als elements. Die elements voeren in bepaalde actie uit. Dit zijn hele kleine stukken functionaliteit. Er zijn om en bij 300 elementen aanwezig. Het is ook niet moeilijk om zelf elementen te schrijven.

7.2.2 HaLVM

Implementatie programmeertaal : Haskell

Hypervisors : Xen

Ondersteunde programmeertaal : Haskell

Documentatie : 8

Toolchain : 6

GitHub stars : 665

HaLVM wordt ontwikkeld door Galios. Galios is software development agency dat unikernels al een tijd in productie gebruikt. Er zijn niet veel bedrijven die unikernels al gebruiken in productie dus ze hebben veel ervaring met de mogelijkheden en moeilijkheden van unikernels.

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

Het werd ontwikkeld met als doel voor besturingssysteem componenten snel te maken en te testen. Maar na een tijd is het geëvolueerd naar andere use cases.

Bij HalVM gaan we de Xen hypervisor als omgeving gebruiken. Er is een integratie met de Xen hypervisor waarop de core library van HaLVM op rust. Er bestaat ook een communications library die bestaat uit Haskell File System en Haskell Network Stack. Deze library kan gebruikt in de meeste gevallen als je een netwerklaag nodig hebt. Als we meer mogelijkheden nodig hebben voor de applicatie dan gaan we modules toevoegen. Ze hebben verder een ecosysteem uitgebouwd om het gemakkelijker te maken voor developers om hun eigen modules te bouwen.

De werkwijze is de volgende: eerst schrijf je best zoveel mogelijk functionaliteit als een normaal Haskell programma. Daarna moet je het programma beginnen refactoren om te gebruiken op HaLVM. Dit is niet gemakkelijk bij uitgebreide applicaties zijn want er zijn maar beperkte mogelijkheden om te debuggen op HalVM.

Er wordt gebruik gemaakt van de standaard Cabal toolset van Haskell zelf. Cabal neemt het maken en ophalen van packages op zich.

HaLVM heeft zelf uiterst goede documentatie met een groot aantal voorbeelden. Die voorbeelden zijn heel gemakkelijke applicaties tot beginnende complexe applicaties.

Zoals in de meeste gevallen moet de compiler van Haskell worden aangepast om rechtstreeks te kunnen werken op de Xen hypervisor. Het is ook geen probleem om andere 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 redelijk actief.

7.2.3 Ling

Implementatie programmeertaal : C/Erlang

Hypervisors : Xen

Ondersteunde programmeertaal : Erlang

Documentatie : 4

Toolchain : 6

GitHub stars : 523

Ling is een Erlang virtuele machine die werkt op Xen. 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, die meer het beheren doen, zijn niet open source. Wanneer je problemen met het ecosysteem moet je support contacteren van Cloudozer.

Zoals bij HaLVM moet eerst de applicatie geschreven worden 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 image zou de applicatie moeten werken.

Railing is een tool die meegeleverd is met Ling die je toelaat om erlang on Xen images te maken. We gebruiken ook xl utility van Xen om domeinen te beheren. De focus van Erlang on Xen was xen in het begin. Met het uitbrengen van LING is het mogelijk geworden om ports te maken voor andere omgevingen. Dat heeft het mogelijk gemaakt om het te laten werken op ARM. De omgeving ARM is vooral te vinden in IOT en mobiele applicaties. Unikernel kunnen handig zijn op deze servers omdat ze soms te klein zijn voor een groot besturingssysteem. De ruimte die je wint bij een unikernel kan hierbij helpen. Verder opent dit ook de mogelijkheid voor de unikernels van LING op bare-metal te laten werken.

De documentatie laat zeker de wensen over. De voorbeelden zijn niet uitgewerkt en het heeft een tijd geduurd voor ik andere tutorials vond. Het was zeer frustrerend om uitgebreide informatie te vinden over Ling en Erlang on Xen. Deze tutorials waren ook niet uitgebreid. Bij het bijkijken van de GitHub repository kunnen we zien dat er niet veel aan gewerkt wordt.

7.2.4 Rumprun

Implementatie programmeertaal : C

Hypervisors : hw, Xen, KVM

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

Documentatie : 9

Toolchain : 8

GitHub stars : 469

Rumprun gebruikt rump kernels voor hun implementatie. Deze rump kernel wordt samengesteld uit componenten afkomstig van NetBSD. NetBSD is een groter besturingssysteem maar zit modulair in elkaar. Men kan het dus gebruiken om een rump kernel te samen te stellen. De rump kernel wordt dan samen met de applicatie verpakt om gebruikt te kunnen worden in verschillende omgevingen.

Er is een uitgebreide collectie van hypervisors waaruit je kan kiezen wanneer je een rumprun applicatie hebt. De term hw staat voor 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.

Er zijn verschillende soorten unikernels. Sommige unikernels specialiseren op basis van programmeertaal en andere op basis van omgeving. Sommige doen zelf beide. Rumprun doet beide. Dit is wel niet zonder gevolg. De performantie zal niet een gespecializeerde unikernel kunnen evenaren.

Er wordt ook aangehaald in hun wiki, dat wanneer je een applicatie hebt die specifiek geschreven is om op een programmeertaal-gebaseerde unikernel en jouw omgeving ondersteunt, dat je best die oplossing gebruikt. Als dit niet het geval is dan kan een rumprun kernel een goede oplossing zijn.

De rump-run packages zijn implementaties van drivers en technologieën die je kan toevoegen aan rumprun kernel. Er zijn een groot aantal packages die je kan gebruiken en de meest bekende zijn zeker 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 dependencies en versies van packages. Een packaging systeem maken is natuurlijk geen simpele opdracht en moet eerst goed uitgedacht worden voor het uitgebracht word.

Rumprun verziet zelf geen compiler ze gebruiken de compiler die aanwezig is op het systeem. In het geval van Mac OS X moet je een aparte compiler installeren. Het is wel goed dat je native op Mac OS X de builds tools kan gebruiken meestal moet je een virtuele machine of container opzetten.

Er is documentatie beschikbaar voor alle hypervisors waarop rumprun kan werken en ook alle informatie die je zou willen als team om te beginnen werken met unikernels.

De programmeertalen die ondersteund zijn, zijn de volgende: C, C++, Erlang, Go, Javascript(node.js), Python, Ruby en Rust. De keuze tussen de programmeertalen is wel uitgebreid te noemen.

7.2.5 MirageOS

Implementatie programmeertaal : OCaml

Hypervisors : Xen, Unix

Ondersteunde programmeertaal : OCaml

Documentatie : 9

Toolchain : 8

GitHub stars : 657

We kunnen zeggen dat het voor een deel allemaal begon bij MirageOS. Hun paper over unikernels en MirageOS wakkerde veel interesse aan rond unikernels. Ervoor was er wel al sprake van unikernels maar MirageOS zorgde voor veel nieuwe initiatieven.

Mirage is een cloud besturingssysteem gemaakt voor veilige en netwerk applicaties met een hoge performantie te maken op verschillende platformen.

De programmeertaal dat je moet gebruiken voor een MirageOS applicatie te maken is OCaml. OCaml is de algemene implementatie van de Caml programmeertaal en voegd object georiënteerd programmeren toe. Het wordt extensief gebruikt door facebook. Deze taal is niet heel erg bekend en dit kan ervoor zorgen dat het niet veel tractie heeft. 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 applicatie aan het werken is. De compiler gaat kijken of hij geen onveilige code kan vinden. Als dit het geval is wordt er niet gecompileerd. Memory management is belangrijk voor resource leaks tegen te gaan. Resource leaks kunnen ervoor zorgen dat de applicatie meer resources gebruikt dan nodig is of zelf de applicatie/systeem laten stoppen met werken.

De toolchain is zeer uitgebreid van het lokaal maken van de applicatie tot het beheren van de applicaties in productie.

De applicatie kan geschreven worden op een Linux of Mac OSX machine. Deze applicatie kan dan werken op een Xen of Unix omgeving. Dit geeft veel mogelijkheden voor deployment.

MirageOS bestaat al een tijd en heeft een groot aantal libraries ter beschikking. Het heeft een uitstekende toolchain voor het compileren van applicaties en het debuggen

van de resulterende unikernel. Debuggen kan soms tot problemen leiden bij unikernels want men kan niet zelf in de unikernel kijken. Dit komt omdat de unikernel geen shell heeft. De debug optie kan hierbij helpen. De resulterende unikernel kan ook werken op ARM-based apparaten. Dit betekent dat we het kunnen gebruiken voor mobiele en IOT applicaties.

De documentatie heeft tutorials en een technische uitleg hoe MirageOS en zijn onderdelen werken. Doordat MirageOS al volwassen is geworden is er al veel te vinden over MirageOS en ook zijn er een grote collectie van packages die je kan gebruiken.

Verder zijn er ook tools dat we werkende MirageOS unikernels kunnen beheren. Dit is onder de vorm van extensies op Xen. Het nagaan van de veiligheid van een extensie die men toevoegt aan Xen moet wel altijd worden gedaan.

7.2.6 IncludeOS

Implementatie programmeertaal : C/C++

Hypervisors : KVM, VirtualBox

Ondersteunde programmeertaal : C++

Documentatie : 6

Toolchain : 6

GitHub stars : 1341

IncludeOS is gemotiveerd door het paper "Maximizing Hypervisor Scalability using Minimal Virtual Machines". 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++.

Ze zorgen voor een bootloader, standaard libraries, modules voor drivers te implementeren en een build- en deploysysteem. Het is simpel om applicaties te maken voor deze unikernel. Je moet enkel een dependency toevoegen aan je applicatie. Dan kan je je applicatie worden omgezet naar een unikernel. Er veranderd dus niet veel voor de developers zelf. Dit zorgt voor een vlotte overgang en is zeker belangrijk wanneer men kiest voor minimale applicaties te maken.

Het laten van meerdere processen op een unikernel van includeOS is niet mogelijk. Dit kan sommige developers afschrikken. Het gebruik van microservices is nog niet wijdverspreid en kan een factor zijn in het selecteren van een unikernel. Enerzijds gaan bedrijven nooit bij unikernels komen wanneer de architectuur geen microservices bevat. Er zijn ook geen race conditions mogelijk omdat er maar 1 proces mogelijk is.

Momenteel ligt de focus van IncludeOS voornamelijk op C++. Dit is een strategie dat kan helpen wanneer developers zoeken naar een implementatie die een community

heeft. IncludeOS heeft een grote community van C++ developers. Hun doel is vooral om een soortegelijk 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 ze extra functionaliteit willen toevoegen.

Hun documentatie is niet heel uitgebreid maar er wordt actief aan gewerkt om dit in orde te krijgen. Ook de tools lopen wat achter.

Als omgeving focussen ze KVM en virtualbox. Hier is het dus gemakkelijk om een unikernel te testen op je eigen machine. Als je services schrijft in C++ dan is IncludeOS een zeer goede keuze. Je kan veel informatie vinden op hun GitHub repository. Pas wel op met alle mogelijkheden die hebt omdat er geen bescherming is van jezelf.

7.2.7 OSv

Implementatie programmeertaal : C/C++

Hypervisors : VMWare, VirtualBox, KVM, Xen

Ondersteunde programmeertaal : 10

Documentatie : JVM

Toolchain : 8

GitHub stars : 2121

De meest uitgebreide unikernel vanuit mijn oogpunt is OSv. Ze ondersteunen een groot aantal programmeertalen. Waaronder Java, Ruby, Javascript, Scala en vele anderen. Hierbij moeten wel wel vermelden dat de implementaties van Ruby en Javascript in Java zijn geschreven. Rhino en JRuby zijn de namen hiervan. Het is simpel om deze programmeertalen toe te voegen wanneer je Java als taal al ondersteunt. Er is wel werk aan de gang voor de native ondersteuning te voorzien voor deze talen.

Verder kan je deze unikernels laten werken op veel omgevingen: VMware, VirtualBox, KVM en Xen. Het is een indrukwekkende lijst van hypervisors de je kan gebruiken. Dit kan ook ervoor zorgen dat je niet vast ziet op 1 omgeving. Er wordt gewerkt om ondersteuning te bieden voor ARM-based apparaten.

Zoals IncludeOS voorheen is OSv geschreven in C++.

Voor het beheren van een OSv instance kan men gebruik maken van de GUI. Bij de meerderheid van unikernels is informatie krijgen door middel van een GUI onmogelijk. Extensies met de hypervisor kunnen hierbij helpen maar dan nog laat de UX de wensen over. De GUI is gebouwd op een REST API die de componenten van OSv openstellen.

Dit komt overheen met de manier hoe docker hun componenten werken. Deze componenten stellen een API open waar de tools verder opgebouwd kunnen worden. Er is zelfs een API specificatie die je kan bekijken op je eigen machine.

OSv ondersteund Amazon Web Services en Google Container Engine als cloud providers. Het is uitzonderlijk dat een unikernel zoveel informatie heeft over hoe je het moet gebruiken. Er is documentatie over cloud providers, hypervisors, hoe je OSv moet gebruiken, hoe je applicaties moet omzetten naar OSv en hoe je aan development voor OSv kan doen. Dit is een zegen voor veel developers die hun bestaande applicaties willen omzetten naar unikernels.

Het is de meest populair implemenatie van unikernels van alle implementaties die we hebben overlopen op GitHub. Ook de activiteit is het hoogste.

7.3 Gekozen implementaties

Zoals al eerder is aangehaald gaan we twee implementaties van unikernels kiezen voor het volgende experiment. Eén ervan moet algemeen zijn dus meerdere hypervisors/-programmeertalen ondersteunen. De andere mag hoogstens één programmeertaal ondersteunen. De keuze voor de algemene implementatie is OSv. De populariteit en hun documentatie sprongen direct in het oog. Ook de mogelijkheden qua hypervisors en programmeertaal is indrukwekkend. OSv ondersteunt deze programmeertalen door middel van JVM. Daardoor kunnen we ook het verschil zien tussen een native implementatie en een tussenliggende laag zoals JVM.

Voor de specifieke implementatie zullen we IncludeOS nemen. Eén van de redenen dat IncludeOS gekozen is de grote community die achter de implementatie staat. De andere implementaties gebruiken ofwel een functionele programmeertaal of OCaml. Het is ook de perfecte keuze voor microservices want er kan maar één proces tegelijk werken.

7.4 Gebruikmaken van Implementatie

7.4.1 Uitleg

In dit experiment zullen we een applicatie opzetten in de gekozen implementatie en ze deployen op een hypervisor. De bedoeling van het experiment is te zien welke vaardigheden je nodig hebt en welke moeilijkheden die je tegenkomt.

7.4.2 Gebruikmaken van OSv

Hoofdstuk 8

Conclusie

Bibliografie

Bala, V., Duesterwald, E., and Banerjia, S. (2000). Dynamo. *ACM SIGPLAN Notices*, 35(5):1–12.

Beloglazov, A. and Buyya, R. (2010). Energy Efficient Resource Management in Virtualized Cloud Data Centers. *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 826–831.

Docker (2016). Docker.

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

Linux. Chroot.

Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. (2013). Unikernels. *ACM SIGPLAN Notices*, 48(4):461.

Mortleman, J. (2009). Security Advantages of Virtualisation. *Computer Weekly*, page 23.

Oracle (2016a). Oracle Virtualbox.

Oracle (2016b). Solaris.

Soundararajan, V. and Anderson, J. M. (2010). The impact of management operations on the virtualized datacenter. *ACM SIGARCH Computer Architecture News*, 38(3):326.

Unikernel Systems (2016). Unikernels.

VMware (2016a). VMware ESXi.

VMware (2016b). VMware Workstation.

Xen Project (2016). Xen Project.

Lijst van figuren

Lijst van tabellen