Jussi Piipponen

Container-järjestelmät

Tradenomi

Tietojenkäsittely

Kevät 2019

KAMK • University of Applied Sciences

KAMK • University of Applied Sciences

TIIVISTELMÄ

Tekijä(t): Piipponen Jussi

Työn nimi: Container-järjestelmät

Tutkintonimike: Tradenomi, tietojenkäsittely

Asiasanat: container, docker, rkt, lxd, Kubernetes

Container-järjestelmät ovat paketoitua sovelluksen suorittamista suojatussa ympäristössä. Yleisimpiä ovat Docker, rkt ja LXD. Opinnäytetyössä käytiin läpi eroja eri container-järjestelmien välillä ja käsiteltiin, kuinka nämä eroavaisuudet vaikuttavat käytettävyyteen. Virtualisointiin verrattuna erona on tilantarve ja käynnistämisen nopeus. Paketissa ei suoriteta normaalin käyttöjärjestelmän komponentteja kuten kerneliä. Useampaa samanlaista palvelua voidaan ajaa rinnakkain samalla alustalla.

Containerin ja virtuaalikoneen suorituksen nopeudessa ei ole huomattavaa eroa. Opinnäytetyössä tehtin container-järjestelmien nopeustestausta. Vertailukohteena käytettiin virtuaalikonetta. Testiohjelmana käytettiin sysbench ohjelmaa.

Opinnäytetyössä esiteltiin peruskomentoja, joita järjestelmiä käytettäessä tarvitaan. Näissä on hiukan eroja eri container-järjestelmien välillä.

Lisäksi esiteltiin containereiden ja klustereiden orkestrointiin luotu alustamoottori Kubernetes ja sitä käyttäviä järjestelmiä. Rancher on avoimessa kehityksessä oleva containereiden ja klustereiden hallintajärjestelmä. IBM Cloud Private Community Edition on raskaampi vaihtoehto, mutta hyvän yritystuen omaava hallintajärjestelmä.

KAMK • University of Applied Sciences

ABSTRACT

Author(s): Piipponen Jussi

Title of the Publication: Container-platforms

Degree Title: Bachelor of Business Administration, Information Technology

Keywords: container, docker, rkt, lxd, kubernetes

Container platforms are containerized applications running in isolated environments, the most most common being Docker, rkt and LXD. In this Bachelor's thesis, differences between container platforms and in their usability were examined. The difference to virtualization is in storage requirements and launch speed. In containers, only libraries are needed for the program to run. Host operating systems kernel is used in container. Multiple similar services can be run side-by-side without interference.

Furthermore, differences between platforms and their significance for users are described in this thesis.

Stress tested speed difference between virtual machines and containerized applications was negligible. In this thesis, speed tests in different container platforms were conducted. A virtual machine was used as a reference. Speed testing was conducted by using the sysbench program.

The basic commands for using different container platforms are described in this thesis. There are slight differences between the container platforms.

In the thesis, the cluster and container orchestration platform Kubernetes was introduced along with a few programs utilizing it. Rancher is an open source management platform for clusters and containers. IBM Cloud Private Community Edition is more requirement heavy but contains good backing for businesses.

SISÄLLYS

SYMBOLILUETTELO

API	Rajapinta, jonka kautta ohjelmat voivat välittää tietoja
ARM	Acorn Computersin kehittämä suoritinarkkitehtuuri, joka on alkanut yleistymään viime vuosina kevyissä laitteissa ja palvelimissa
Container	Paketoitu sovellus, joka on suojattu ympäröivästä ympäristöstä
Cow	Copy-On-Write on optimointitekniikka, jossa kirjoitustilanteessa vain viitataan vanhaan, näin säästetään tilaa
CPU	On tietokoneen osa, joka suorittaa laskennan
Käyttöjärjestelmä-container	Sisältää melkein täydellisen levykuvan käyttöjärjestelmästä
LXC	Ensimmäinen container-järjestelmä, johon myöhemmin julkaistut pohjautuvat
POWER	Suoritinarkkitehtuuri, jonka kehittäjänä toimii IBM
Sovellus-container	Container, jossa on vain todella kriittisimmät tiedostot mukana sovelluksen suorittamiseen
X86	Vuonna 1972 Intelin markkinoille tuoma suoritinarkkitehtuuri

1 Johdanto

Tässä opinnäytetyössä tulen tutkimaan erilaisia container-järjestelmiä. Otan lähempään esittelyyn suurimman tällä hetkellä olevan ratkaisun, Dockerin. Lisäksi tutkin myös paria muuta pienempää järjestelmää: rkt:n ja LXD:n. Jälkimmäisessä on suurimpana taustatukijana Canonical, joka antaa rahoitusta ja kehitystukea järjestelmälle. Koska LXD on tuotu Ubuntu-jakeluun, sille löytyy käyttäjäkuntaa testaamaan järjestelmää.

Valitsin aiheen, koska olen hyvin kiinnostunut container-järjestelmistä ja teknologiasta taustalla. Koska tästä ei ollut koulutusta, haluan tuoda tätä esille ja kertoa enemmän aiheesta. Lisäksi koen, että tämä tulee auttamaan minua työurallani eteenpäin.

Teen vertailuja container-järjestelmien ja virtuaalikoneen välisistä nopeudellisista eroista. Lisäksi esittelen muutaman Kuberneteksen ympärille rakennetun hallintajärjestelmän, joissa hyödynnetään Dockeria tai rkt:tä. Ensimmäisenä esittelen avoimeen lähdekoodiin pohjautuva Rancher, koska se on edullinen valinta klustereiden hallintaan yksityiskäytössä tai pienyrityksissä. Toisena esittelen IBM Cloud Private alustan, johon olen perehtynyt omassa työssäni.

Lopussa pohdin eri ratkaisujen tulevaisuutta.

2 Container-teknologian historia

Virtualisointi kehitettiin, jotta pystyttäisiin hyödyntämään täydellisesti saatavilla olevat laskenta-resurssit. Virtualisoinnilla on mahdollista ajaa useampaa virtuaalikonetta samalla palvelinalustalla antaen jokaiselle suojatun oman ympäristön. Tällainen saavutetaan käyttämällä hypervisoria, joka asettuu raudan ja virtuaalikoneen väliin. Containerit ovat looginen seuraava askel virtualisoinnista, antaen mahdollisuuden virtualisointiin niin käyttöjärjestelmätason kuin sovellustason containereille. [1.]

Container-teknologia on ollut olemassa jo pitkään erilaisissa muodoissa, mutta on saanut vasta viime aikoina suurta suosiota osakseen, kun Linux kerneliin lisättiin natiivi tuki containereille. Taulukossa 1 on listattu erilaisia tekniikoita, jotka ovat johtaneet tämän hetkiseen tilanteeseen. [1.]

Osalla taulukossa mainituilla teknologioilla on hyvin erityinen käyttötarkoitus, kuten chroot, joka antaa tiedostojärjestelmätasolla eristyksen muuttamalla juurihakemiston suoritettavalle prosessille tai sen lapsiprosesseille. Toiset teknologiat listalla antavat käyttöjärjestelmätason virtualisointia, kuten Solaris containers tai LXC. [1.]

Kaikki nykypäivänä olevat container-järjestelmät pohjautuvat LXC:hen, joka julkaistiin vuonna 2008. LXC:n kehityksen mahdollisti, kun Linux kerneliin versiosta 2.6.24 eteenpäin lisättiin muutamia tärkeitä ominaisuuksia. [1.]

Taulukko 1. Container-teknologian historia [1]

Vuosi	Teknologia	Ensimmäisen kerran käyttöjärjestelmissä
1982	Chroot	Unix pohjaiset käyttöjärjestelmät
2000	Jail	FreeBSD
2000	Virtuozzo containers	Linux, Windows (Parallels Inc. versio)
2001	Linux VServer	Linux, Windows
2004	Solaris containers (zones)	Sun Solaris, Open Solaris
2005	OpenVZ	Linux (avoimen lähdekoodin versio Virtuozzo:sta)
2008	LXC	Linux
2013	Docker	Linux, FreeBSD, Windows

3 Docker

Dockerin tarkoituksena on nopeuttaa kehittämistä, koska alusta ei ole rajoituksena koodin optimoinnille. Virtualisointiin verrattuna erona on keveys. Samalla alustalla voi myös toimia paketoituja sovelluksia ja virtualisoituja koneita. Koska jokainen palvelu pyörii omassa paketissa, ei tämä voi millään tavalla muokata alla toimivaa isäntää. Lisäksi palveluiden ylös nostaminen on paljon nopeampaa. [2.]

Linux Containers, johon Docker alun perin pohjautui, julkaistiin 2008. Vuonna 2013 pilvipalvelutarjoaja dotCloud julkaisi avoimena lähdekoodina oman järjestelmänsä Dockerin. [3.]

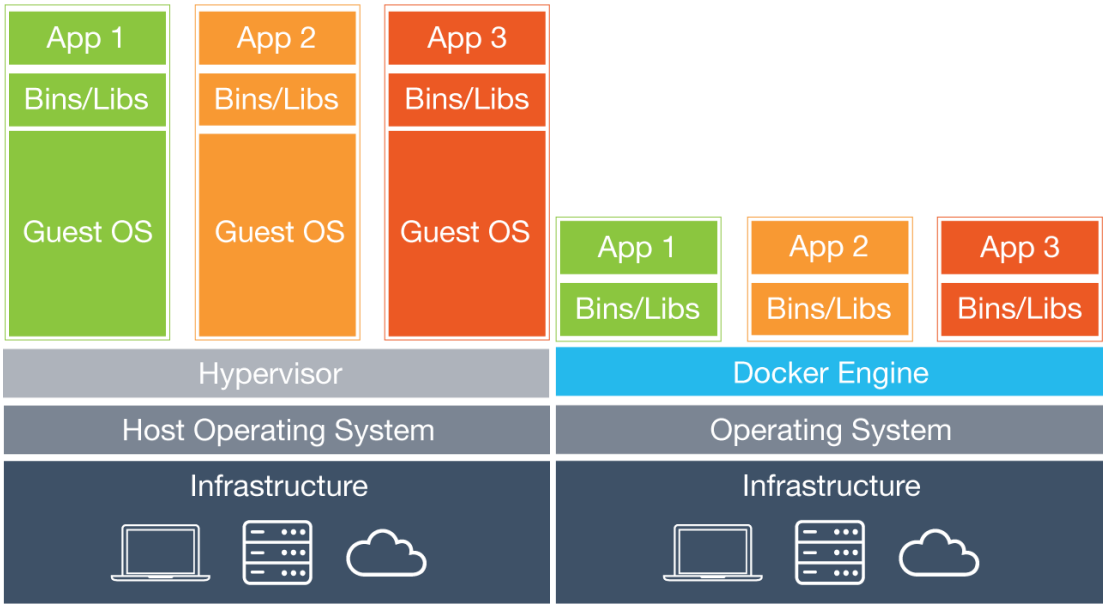
DotCloudin toimitusjohtaja Solomon Hykes esitteli Dockerin kevyenä uudelleentoistettavana virtualisointiratkaisuna. Hän totesi, että Docker on eristetty prosessitasolla, jolla on oma tiedostojärjestelmä. API antaa mahdollisuuden pääkäyttäjälle ajaa komentoja paketoituna. [4.]

Maaliskuussa 2014 Docker luopui Linux Containersin LXC-tekniikasta siirtyessä omaan libcontainer-järjestelmään. Siirtymisen jälkeen Dockeria pystyi ajamaan muillakin alustoilla kuin Linux, kuten FreeBSD tai Solaris. [5.]

Dockerin kehityksessä käytettiin Go kieltä, joka oli tässä vaiheessa hyvin nuori kieli. Ottaen huomioon Dockerin toiminnan tarvittiin nopeutta, toinen hyvä vaihtoehto olisi ollut käyttää perinteisempää C-kieltä. Kuitenkin nähtiin viisaammaksi vaihtoehdoksi valita toinen kieli, joka oli yksinkertainen ja tehokas ottaen huomioon päähuomiona olleen käyttöjärjestelmän, Linuxin. [6.]

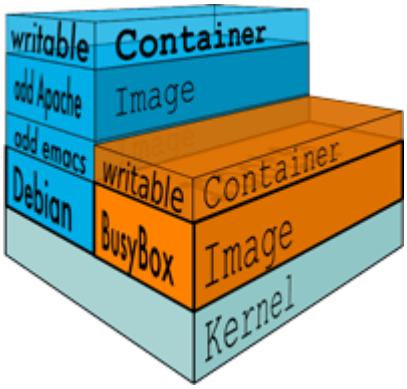
Registry on paikallisesti suoritettava kirjasto, johon voi tallentaa omia paketteja myöhempää suoritusta varten talteen. Tämä ohjelmoitiin uusiksi Go-kielellä 2015 alkupuolelle noudattamaan Dockerin koodia. [7.]

Virtualisoinnissa virtualisoidaan tietokone täydellisesti. Virtuaalikone ei näe, että se pyörii hypervisorin päällä. Kuten kuvassa 1 on esitetty, containeri poistaa käyttöjärjestelmän ja jättää jäljelle pelkän suoritettavan sovelluksen. Paketissa on vain tarvittavat kirjastot ja suoritettava sovellus. Kerneliä ei suoriteta paketissa, vaan luotetaan isäntätietokoneen omaan. Paketoitu sovellus tulee tästä syystä olla samaa prosessoriarkkitehtuuria kuin isäntä. Yleisimpinä arkkitehtuureina ovat x86 ja ARM. IBM Cloud Private on saatavilla myös POWER arkkitehtuurille.



Kuva 1. Virtualikoneen ja containerin ero [8]

Kuvassa 2 pakettiin laitetaan yleensä vain tarvittavat kirjastot, joita suoritettava sovellus tarvitsee toimiakseen. Oikeanpuoleisessa on vain hyvin yksinkertainen komentokehote alustana, jonka päällä voi suorittaa erilaisia skriptejä. Vasemmanpuoleisessa containerissa on jo monimutkaisempi paketti, jossa on muun muassa emacs-tekstieditori ja Apache-palvelinohjelma. Alustana tässä on sisällä täydellinen Debian-jakelu.



Kuva 2. Kaksi erilaista Docker pakettia [8]

4 Rkt

Dockerin lähin kilpailija sovellus-container puolella on rkt kehittäjänään CoreOS, jonka päätarkoituksena oli keskittyminen tietoturvaan. Ajan saatossa Docker on sulkenut useita tietoturvaongelmia; kuitenkin Docker suoritetaan pääkäyttäjänä (root), kun taas rkt suoritetaan rajoitetuilla käyttäjäoikeuksilla. Jos joku onnistuisi murtautumaan ulos containerin nimiavaruudesta, niin Dockerissa murtautuja saisi täydet oikeudet automaattisesti isäntäkoneeseen. Rkt:ssa tämä ei ole mahdollista, koska murtautujan pitäisi pystyä vielä eskaloimaan käyttäjän oikeudet. Vaikka tämä ei anna hyvää kuvaa Dockerista tietoturvan näkökulmasta, niin on mahdollista kehityksen näkökulmasta, että nämä ongelmat voidaan lieventää tai korjata tulevaisuudessa. [9.]

Toinen eroavaisuus on, että Docker suorittaa yhtä prosessia, kun taas rkt pystyy suorittamaan useampaa. Tämä antaa mahdollisuuden suorittaa useamman palvelun samassa containerissa. On mahdollista myös rakentaa Dockerilla container, jossa suoritetaan useampaa prosessia, mutta tämä vaatii työtä saada toimimaan kunnolla. Kuitenkin tämä myös muokkaa kehittäjän ajattelua micropalveluja kohtaan, ilman että miettisi containeria virtuaalikoneena. [9.]

Vaikka on eroavaisuuksia olemassa rkt:n ja Dockerin välillä, miksi valita toinen yli toisen? Isoin ero on kuitenkin adoptointinopeus. Vaikkakin rkt on hiukan nuorempi, niin Docker on jo hyväksytty suurimpien toimijoiden toimesta. Eikä tälle näy hidastumisen merkkejä. [9.]

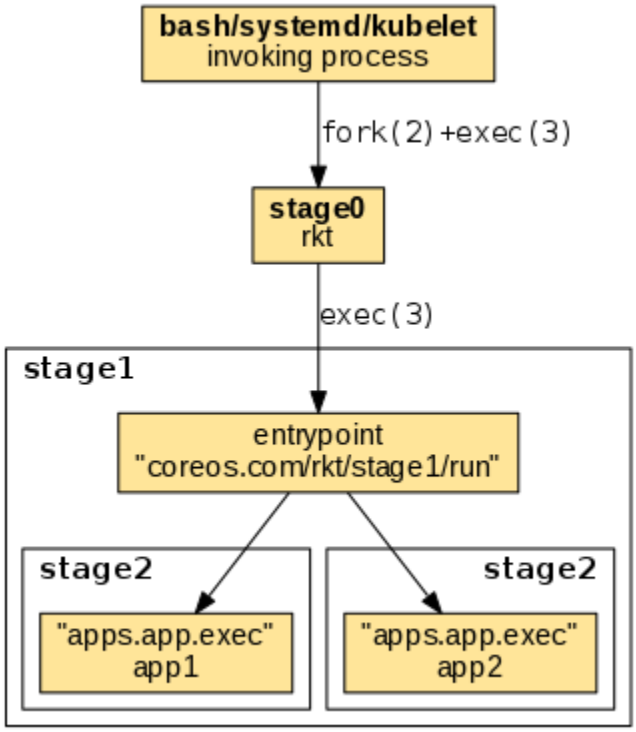
Rkt:ssa suoritus on pilkottu useaan selkeään vaiheeseen (stage). Lisäksi rkt:ssa käytetään nimitystä pod containerin sijaan. Tätä juuri siitä syystä, että rkt pystyy suorittamaan useamman prosessin samassa containerissa. Kuten kuvassa 3 on esitetty, useampi vaihe tapahtuu, kun kutsutaan komentoa rkt. [10.]

Stage0 on vaihe, joka tapahtuu, kun rkt-komentoa käytetään. Esimerkissä suorittajana on kubelet-palvelu, mutta tämä voi olla käyttäjäkin komentokehotteesta. Vaikka rkt kutsutaan pääkäyttäjänä, sen prosessi viedään lapsiprosessiksi pääprosessille, käyttäen fork- ja exec-käskyjä. Stage0:n tarkoituksena on ladata tarvittavat levykuvat, luoda UUID podille, luoda manifesti, tiedostojärjestelmän luonti ja muita pienempiä suoritukseen tarvittavia ylläpidollisia toimia. Lisäksi valmistellaan levykuvasta Stage1- ja Stage2-tiedostot oikeisiin kansioihin. [10.]

Stage1-vaiheessa myös käytetään exec-käskyä, jotta suoritus tehdään stage0-lapsiprosessina. Tämän vaiheen tehtävänä on suorituksen ylläpidolliset tehtävät. Vaihe myös hallitsee suoritettavien containereiden nimiavaruuksia, verkotuksia ja tiedostojärjestelmäliitoksia, joita tarvitaan podin

suorittamiseksi. Stage1 käyttää Stage0:n luomaa manifestia määrittääkseen asetukset näille asioille. Manifestilla voidaan myös luoda prosessijonoja missä järjestyksessä mikäkin suoritetaan. [10.]

Stage2 on sitten itse ympäristöt, joissa Stage1 määrittämät prosessit tai sovellukset suoritetaan [10].



Kuva 3. Rkt sovellus-container vaiheet [10]

5 LXD

Suurin ero LXD:n ja Dockerin välillä on suhtautuminen tiedostojärjestelmään. Docker käyttää union filesystem -tekniikkaa, kun taas LXD käyttää rootfs-juurihakemistoa.

LXD on laajennus LXC:hen, jolla pyrittiin parantamaan käyttäjäkokemusta. LXD käyttää liblxc-kirjastoa ja käyttää Go-kieltä containereiden hallintaan. LXD muodostuu kolmesta komponentista, daemonista (lxd), komentorivi-asiakasohjelmasta (lxc) ja OpenStack Nova-liitännäisestä (nova-compute-lxd). Ubuntuun pohjautuvat variantit tukevat hyvin LXD:tä, mutta tätä ei ole vielä paketoitu Debianiin ja siihen pohjautuviin. [1.]

LXD:ssä käytetään valmiiksi luotuja levykuvia, joita on saatavilla laajasta osasta eri Linux-jakeluita. Projektin luoja, rahoittaja ja tämän hetkinen suurin kehittäjä on Canonical Ltd. mukana on kuitenkin eri yhtiöitä ja itsenäisiä kehittäjiä. Kehitys tapahtuu käyttäen Apache 2 lisenssiä ja kehittäjien ei tarvitse allekirjoittaa minkäänlaisia sopimuksia kehittääkseen LXD:tä. [11.]

LXD:stä on olemassa kaksi erilaista julkaisuversiota, Long time support (LTS) ja Feature. LTS on tällä hetkellä versiossa 3.0 ja on tuettu vuoteen 2023 asti. Siihen tulee päivityksiä ja vikakorjauksia, mutta uusia ominaisuuksia ei lisätä. Feature-polkua päivitetään joka kuukausi, ja siihen tulee uusimmat ominaisuudet. Feature-julkaisussa on tuki olemassa vain uusimpaan versioon. Poikkeuksena kuitenkin, jos julkaisu kuuluu jonkin distribuution jakelulistoille. Tässä tapauksessa tuen vastuu on distribuution julkaisijalla. [11.]

6 Tekniikka

Seuraavassa kappaleessa esittelen ominaisuuksia, joita tarvitaan eristetyn ympäristön luomiseksi alustana olevalla koneella.

6.1 Cgroups

Jotta ymmärtäisimme control groupsin (cgroups) tärkeyden, otetaan huomioon normaali skenaario: Prosessi yrittää tehdä pyynnön erästä resurssia varten järjestelmässä, mutta resurssia ei ole saatavilla, joten järjestelmä päättää hylätä pyynnön niin pitkäksi aikaa, kun järjestelmässä tulee kyseisiä resursseja vapaaksi. Pyydettyjä resursseja voi vapautua jonkin toisen prosessin niitä vapauttaessa. Tämä myöhästyttää prosessin suorittamista; monikaan ohjelma ei hyväksy tällaista käytöstä. Tällainen resurssien puute voi aiheutua jonkin haittasovelluksen toimesta, joka kaappaa kaikki resurssit haltuunsa ja estää järjestelmää toimimasta. [1.]

2007 Google esitteli uuden tavan hallita resursseja geneerisellä tavalla käyttäen cgroups projektia. Control groups antaa mahdollisuuden hallita ja pitää tiliä resursseista perustuen prosessiryhmiin. Mainline Linux kerneli implementoi cgroupsin vuonna 2008, ja tämä raivasi tietä LXC:lle. [1.]

Cgroupsilla pystytään järjestemään taskit, prosessit ja niiden lapsiprosessit omiin hierarkkisiin prosessiryhmiin. Näille ryhmille voidaan antaa asetuksia ja määrityksiä tarpeen mukaan. [1.]

Prosessori- ja muistiresurssit ovat yleisimpiä, joita hallinnoidaan cgroupsilla, vaikkakin sillä pystyy myös tekemään myös paljon monimutkaisempia resurssirajoituksia [1]. Containeria käynnistettäessä tai suorituksen aikana voidaan rajoittaa CPU-ytimet kahteen ja muisti rajoittaa yhteen gigatavuun.

Järjestelmissä on hiukan eri syntaksit olemassa näille rajoituksille. Seuraavaksi esimerkkejä Dockerissa ja LXD:ssä, jos halutaan muuttaa rajoituksia käynnissä oleville containareille. Rkt ei tue suorituksen aikaista containerin rajoitusten muuttamista suoraan komentoriviltä. Määritykset pitää siis antaa containeria käynnistäessä.

Docker

```
docker update -m 1g --cpus 2 my-web-server
```

`-m 1g`	Asetetaan muistirajoitus 1 GB
`--cpus 2`	Määritetään containerille kahden CPU ytimen rajoitus
`my-web-server`	Mille containerille päivitetyt rajoitukset annetaan

LXD

```
lxc config set my-web-server limits.cpu 2
lxc config set my-web-server limits.memory 1GB
```

`limits.cpu 2`	Määritetään containerille kahden CPU ytimen rajoitus
`limits.memory 1GB`	Asetetaan muistirajoitus 1 GB
`my-web-server`	Mille containerille rajoitukset annetaan

Rkt

```
rkt run --insecure-options=image --cpu=2 --memory=1G my-web-server.aci
```

`--insecure-options=image`	Tällä ohitetaan containerin allekirjoituksen tarkistus
`--cpu=2`	Määritetään containerille kahden CPU ytimen rajoitus
`--memory=1G`	Asetetaan muistirajoitus 1 GB
`my-web-server.aci`	Mikä image suoritetaan containeriksi

6.2 Namespaces

Nimiavaruudella (Namespaces) pystytään antamaan yleisiä järjestelmäresursseja suojatusti prosessille käyttöön, sen vaikuttamatta itse järjestelmäresurssiin. Nimiavaruuksilla luodaan containereita; niillä saadaan toteutettua tarvittavat eristämiset containerin ja isäntäkoneen välillä. [1.]

Ajan saatossa erilaisia nimiavaruuksia on lisätty Linuxin kerneliin. Tällä hetkellä on lisätty seitsemän eri käyttötarkoituksen nimiavaruutta kerneliin. Näitä on esitelty taulukossa 2. [1.]

Taulukko 2. Olemassa olevat nimiavaruudet [1]

Nimiavaruus	Muuttuja	Eristää
Cgroup	CLONE_NEWCGROUP	Cgroup juurihakemisto
IPC	CLONE_NEWIPC	System V IPC, POSIX viestijonot
Network	CLONE_NEWNET	Verkkolaitteet, verkkopinot, portit, jns.
Mount	CLONE_NEWNS	Tiedostojärjestelmäliitokset
PID	CLONE_NEWPID	Prosessi ID:t
User	CLONE_NEWUSER	Käyttäjä ja ryhmä ID:t
UTS	CLONE_NEWUTS	Isäntänimi ja NIS juurinimi

PID nimiavaruus mahdollistaa jokaisella containerilla oman prosessinumeroinnin (PID). Jokainen PID muodostaa oman prosessihierarkian, joten lapsinimiavaruudessa olevalla prosessilla voi olla eri PID kuin isäntänimiavaruudessa. Isäntänimiavaruus voi nähdä lapsinimiavaruuden ja vaikuttaa siihen, mutta lapsinimiavaruus ei voi nähdä eikä vaikuttaa isännän nimiavaruuteen. Jos suoritamme prosessin `sh container.sh` containerissa, niin se saa PID-numeron 8, kuten kuvassa 4 on esitetty, mutta isäntäkone on antanut samalle prosessille PID-numeron 29778, kuten kuvassa 5 on esitetty. [12.]

```
bash-4.3# ps aux | grep container
root         8  0.0  0.0  11664  2656 ?        S    07:37   0:00 sh container.sh
root        80  0.0  0.0   9084   840 ?        S+   07:43   0:00 grep container
bash-4.3#
```

Kuva 4. Prosessin PID containerissa [12]

```
[root@dockerhost ~]# ps aux | grep container
root     29778  0.0  0.0  11664  2660 pts/3    S    07:37   0:00 sh container.sh
root     29912  0.0  0.0 113004  2160 pts/4    S+   07:45   0:00 grep --color=auto container
[root@dockerhost ~]#
```

Kuva 5. Prosessin PID isäntäkoneella [12]

Network-nimiavaruudella voimme suorittaa esimerkiksi Apache web-palvelinta useamman kerran samassa isäntäkoneessa. Ilman net-nimiavaruutta ei pystyisi kuuntelemaan porttia 80 jokaisessa containerissa. Jokaisella containerilla voi olla omat verkkolaitteet hyödyntäen net nimiavaruutta. Myös loopback adapteri voi olla eri jokaisella containerilla. [12.]

Jotta container pystyy käyttämään verkkoa, pitää lisätä uusi pari erityisiä verkkolaitteita eri nimiavaruuksissa ja antaa näille oikeus keskustella keskenään. Yksi erityinen verkkolaite sijaitsee containerissa ja toinen sijaitsee isäntäkoneella. Yleensä laite, joka sijaitsee containerissa, saa nimeksi eth0 ja isäntäkoneella oleva saa satunnaisen nimen, kuten veth516cc56. Nämä laitteet linkitetään käyttäen siltaa (docker0) isäntäkoneella mahdollistaakseen yhteydenpidon containerien väillä ja välittääkseen paketteja. [12.]

UTS-nimiavaruudella voidaan toteuttaa niin, että containerilla ja isäntäkoneella on eri isäntänimi [12].

User-nimiavaruudella pystytään mahdollistamaan käyttäjälle ei-pääkäyttäjä-ID isäntäkoneelle, mutta pääkäyttäjäoikeudet containerin sisällä. Tämä on mahdollista, kun pystytään määrittämään käyttäjille ja ryhmille ID-numerot jokaiselle nimiavaruudelle erikseen. [12.]

IPC eli inter-process communication antaa mahdollisuuden eri prosessien välittää viestejä tai jakaa muistia keskenään. Jotkin ohjelmat luottavat tähän toimintaan ja voi tulla tilanne, jossa samanlaiset ohjelmat voisivat varata saman IPC-resurssin käyttöönsä, jolloin tulisi päällekkäisyyksiä ja molemmat ohjelmat kaatuisivat. Kun on mahdollista eriyttää IPC-resurssit container kohtaisesti, niin ei tule päällekkäisyyksiä. [12.]

Mount-nimiavaruudella voidaan antaa containerille käyttöön omia liitettyjä tiedostojärjestelmiä ja toiset containerit eivät voi lukea niitä [12].

6.3 Union filesystem ja rootfs

LXD ja Docker eroavat hiukan toisistaan, kuinka tiedostojärjestelmää käsitellään. LXD luottaa oletuksena rootfs-lähestymiseen, jossa koko juurihakemisto on yksi levykuva. Docker ja siitä olevat versiot käyttävät union filesystem -tekniikkaa, jossa niputetaan useampi erilainen tiedostojärjestelmä päällekkäin kuin sipuli konsanaan. Tällä tavalla voidaan ensin lisätä esimerkiksi Ubuntu levykuva staattisena vain lukukerroksena. Tämän jälkeen lisätään kerros, jossa on muutokset tiedostojärjestelmään Apache web-palvelimelle. Tällä säästetään tilaa huomattavasti, kun Ubuntu levykuvaa voidaan hyödyntää useammassa containerissa.

LXD:ssä tarvitaan containerille levykuva, missä on containerin juurihakemisto (rootfs). Juurihakemisto sisältää joukon tiedostoja, sisältäen samanlaisen tiedostorakenteen kuin normaalissa GNU/Linux-pohjaisessa käyttöjärjestelmässä. Juurihakemiston koko on kuitenkin pienempi kuin normaalissa käyttöjärjestelmässä, koska sen ei tarvitse sisältää kerneliä. Container jakaa saman kernelin isäntäkoneen kanssa. [1]

Juurihakemiston kokoa voidaan pienentää entisestään, jos se sisältää pelkästään vain tarpeelliset tiedostot sovellukselle ja se jakaa tiedostojärjestelmän isäntäkoneen kanssa. Käyttäessä copy-on-write (COW) tekniikkaa voidaan käyttää yhtä vain lukulevykuvaa useamman containerin kesken. [1.]

Dockerissa union filesystem mahdollistaa erillisten tiedostojärjestelmien tiedostojen ja kansioiden niputtamisen kerroksiksi, jolla voidaan luoda läpinäkymättömästi uusi tiedostojärjestelmä. Kun container käynnistetään, Docker niputtaa kaikki kerrokset ja luo vain lukutiedostojärjestelmän. Tämän päälle lisätään luku-/kirjoitusoikeudellinen kerros containerin suorituksen aikaiselle tiedostojärjestelmälle. Dockerissa voidaan käyttää useampaa erilaista variaatiota tästä teknologiasta, kuten AUFS, Btrfs, zfs, overlay, overlay2 ja DeviceMapper. [12.]

7 Ohjeita

Seuraavissa alikappaleissa on listattuna muutamia perustekniikoita containereiden tai alustan hallintaan eri järjestelmissä. Käyttöjärjestelminä seuraavissa kappaleissa on käytetty Ubuntua.

7.1 Docker

Dokumentaatio

https://docs.docker.com/

Asentaminen

Alla komennot, joilla voidaan asentaa Docker-ce Ubuntu käyttöjärjestelmään.

```
sudo apt remove docker docker-engine docker.io
sudo apt install -y apt-transport-https ca-certificates curl software-proper-
ties-common python
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo apt-key fingerprint 0EBFCD88
sudo add-apt-repository "deb [arch=amd64] https://down-
load.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt update && sudo apt install -y docker-ce
```

Run

Tällä komennolla pystyy luomaan ja suorittamaan uuden containerin. Alla olevassa komennossa suoritetaan täydellinen Ubuntu-jakelu. Kun annetaan muuttujat `-i`, `-t` ja komennon loppuun `/bin/bash`, suoritetaan containerissa konsoli, joka avataan käyttäjälle. Tämän jälkeen voidaan suorittaa komentoja normaalisti. Jos kuitenkin tulee containerista ulos komennolla exit, niin containeri tässä esimerkissä pysähtyy.

```
docker run -i -t -name ubuntu-kone ubuntu /bin/bash
```

Attach

Voit irrottautua suorituksessa olevasta containerista antamalla näppäinkomennon CTRL+PQ. Tämän jälkeen palaat edelliseen komentokehotteeseen. Voit avata containerin komentokehotteen uudelleen antamalla alla olevan käskyn.

```
docker attach ubuntu-kone
```

Start, stop ja restart

Näillä kolmella komennolla voi ohjata containerin tilaa. Start komennolla voidaan käynnistää sammuksissa oleva containeri. Stop pysäyttää ja restart käynnistää containerin uudelleen. Jos on tehty muutoksia, jotka pitää saada voimaan, näitä komentoja tarvitaan.

```
docker start ubuntu-kone
docker stop ubuntu-kone
docker restart ubuntu-kone
```

Ps

Komennolla näkee kaikki sillä hetkellä käynnissä olevat containerit.

```
docker ps
```

Jos kuitenkin haluaa nähdä kaikki myös pysäytetyt tai kuolleet containerit, tulee lisätä muuttuja `-a`.

```
docker ps -a
```

Rm

Rm-komennolla voidaan poistaa containeri.

```
docker rm ubuntu-kone
```

Yllä oleva poistaa sammuneen tai kuolleen containerin. Jos kuitenkin halutaan poistaa käynnissä oleva niin pitää lisätä muuttuja `-f`.

```
docker rm -f ubuntu-kone
```

Logs

Tällä komennolla voit lukea containerin tulostaman login. Tämän voi tehdä niin pitkään kuin containeri on olemassa, vaikka se olisi sammutettuna. Jos haluat seurata reaaliaikaisesti logia, niin anna myös muuttuja `-f` seuraavan komennon kanssa. Voit katkaista reaaliaikaisen login seurannan näppäinkomennolla CTRL+C. Logiin tulostuvat containerin prosessin tulosteet jonoista stdout ja stderr.

```
docker logs ubuntu-kone
```

7.2 Rkt

Dokumentaatio

https://coreos.com/rkt/docs/latest/

Asentaminen

Alla olevia komentoja hyödynnetään asennettaessa versio 1.30.0. Suosittelen tarkistamaan rkt:n GitHub-tililtä uusimman version numeron. Osoitekorjaukset rajoittuvat versionumeron vaihtamiseen seuraavissa riveissä. GitHub-tili löytyy osoitteesta https://github.com/rkt/rkt/releases/latest, joka ohjaa automaattisesti uusimpaan versioon.

```
gpg --recv-key 18AD5014C99EF7E3BA5F6CE950BDD3E0FC8A365E
wget https://github.com/rkt/rkt/releases/download/v1.30.0/rkt_1.30.0-
1_amd64.deb
wget https://github.com/rkt/rkt/releases/download/v1.30.0/rkt_1.30.0-
1_amd64.deb.asc
gpg --verify rkt_1.30.0-1_amd64.deb.asc
sudo dpkg -i rkt_1.30.0-1_amd64.deb
```

Run

Voit suorittaa tarkistetun levykuvan tässä esimerkin tapauksessa Alpine distribuution. Tämä komento suorittaa automaattisesti sh-komentokehotteen ja liittää sinut siihen. Jos lisää myös muuttujan --dns=8.8.8.8, määritetään podille myös nimipalvelinasetus. Tämä helpottaa, jos on tarpeellista käyttää verkkoja.

```
rkt run --interactive --dns=8.8.8.8 quay.io/coreos/alpine-sh
```

Tällä hetkellä rkt ei tue samanlaista irrottautumista ja takaisin liittymistä containerista, kuin Docker tukee.

List

List komennolla näkee listauksen olemissa olevista podeista ja niiden tilan

```
rkt list
```

Rm

Rkt käyttää ajastettua puhdistusta vanhoille podeille, jotka ovat Exit-tilassa. Ensimmäisellä kierroksella podit, jotka ovat sammuneet, merkataan roskaksi ja toisella kierroksella ne poistetaan. Näin saadaan resursseja vapautettua takaisin käyttöön. Rm komennolla voit poistaa podeja tilanteessa, jossa tarvitset resursseja nopeasti takaisin käyttöön. List komennolla saat podin UUID:n tietoon, jota tarvitset komennossa.

```
rkt rm c138310f
```

Gc

Tällä komennolla voit suorittaa podien merkkauksen pakotetusti, jotta seuraavalla kierroksella ne jo poistetaan.

```
rkt gc
```

7.3 LXD

Dokumentaatio

https://lxd.readthedocs.io/en/latest/

Asentaminen

LXD pitäisi tulla esiasennettuna Ubuntu 18.04 LTS-jakelussa. Mutta jos ei ole esiasennettu niin sen voi asentaa ajamalla alla olevan komennon.

```
snap install lxd
```

Init

LXD:n asennuksen jälkeen perusasetukset voi määrittää komennolla init.

```
lxd init
```

Perusasetukset voi määrittää painamalla enter kaikkiin kysymyksiin.

Launch

Voit suorittaa ensimmäisen containerin komennolla launch. Tämä lataa rootfs-levykuvan ja käynnistää containerin.

```
lxc launch ubuntu ubuntu-kone
```

Exec

Tällä komennolla voi suorittaa containeriin prosesseja. Alla olevassa avataan bash komentokehote.

```
lxc exec ubuntu-kone -- /bin/bash
```

Console

Jos haluat käyttää consolea, pitää ubuntu-käyttäjälle määrittää salasana. Tämän voi tehdä yllä olevalla exec komennolla ja suorittamalla komennon `passwd ubuntu`. Tällä annetaan containerissa olevalle peruskäyttäjälle salasana, jolla voidaan sitten kirjautua. Kun tämä on suoritettu, voidaan antaa alla oleva komento, jolla saadaan konsoliyhteys containeriin auki. Toiminnallisuutta voisi verrata SSH:on.

```
lxc console ubuntu-kone
```

Tästä konsolista pääsee väliaikaisesti pois antamalla näppäinyhdistelmä CTRL+a q.

Start, stop ja restart

LXD:ssä on vastaavanlaiset komennot kuin Dockerissa containerin tilan hallintaan.

```
lxc start ubuntu-kone
lxc stop ubuntu-kone
lxc restart ubuntu-kone
```

Delete

Tällä komennolla voidaan poistaa containeri, jos sitä ei enää tarvita.

```
lxc delete ubuntu-kone
```

8 Nopeustestaus

Testaukset aloitettiin tekemällä neljä kappaletta virtuaalikoneita VMWare-ympäristöön. Näihin kaikkiin asennettiin Ubuntu 18.04 LTS käyttöjärjestelmä. Tämä siitä syystä, että kaikille kolmelle vertailussa olleelle järjestelmälle löytyy asennusohjeet kyseiselle käyttöjärjestelmälle. Lisäksi löytyy myös valmis containeri, joka perustuu kyseiseen käyttöjärjestelmään. Näin pystyttiin minimoimaan containerista johtuvat erot.

Kun container-järjestelmä oli saatu asennettu kaikille koneille, suoritettiin jokaisella komennot. Näin saatiin luotua containeri, jolla testit suoritettiin. Rasitustestaukseen käytettiin sysbench-ohjelmaa.

Containerin pohjustamiseen käytettiin seuraavia komentoja.

Docker

```
sudo docker run -it --name ubuntu ubuntu:18.04 /bin/bash
```

rkt

```
sudo rkt run --interactive --net=host --dns 8.8.8.8 docker://ubuntu:18.04 --
insecure-options=image
```

LXD

```
lxc launch ubuntu:18.04 ubuntu
lxc exec ubuntu -- /bin/bash
```

Tämän jälkeen suoritettiin samat komennot kaikissa kolmessa containerissa ja puhtaassa virtuaalikoneessa. Ainoana erona, että containereissa ollaan sisässä jo pääkäyttäjänä (root), joten niissä ei tarvitse käyttää sudo käskyä.

```
apt update && apt upgrade -y
curl -s https://packagecloud.io/install/repositories/akopytov/sys-
bench/script.deb.sh | bash
apt -y install sysbench
```

Tämän jälkeen vietiin jokaiselle koneelle pieni skriptin, jolla suoritettiin testit sarjassa.

test.sh

```
#!/bin/bash
sysbench fileio prepare
sysbench --threads=16 --time=10 --file-test-mode=seqwr --validate=on fileio
run
sysbench --threads=16 --time=10 --file-test-mode=seqrewr --validate=on fileio
run
sysbench --threads=16 --time=10 --file-test-mode=seqrd --validate=on fileio
run
sysbench --threads=16 --time=10 --file-test-mode=rndrd --validate=on fileio
run
sysbench --threads=16 --time=10 --file-test-mode=rndwr --validate=on fileio
run
sysbench --threads=16 --time=10 --file-test-mode=rndrw --validate=on fileio
run
sysbench --threads=16 --time=30 --validate=on cpu run
sysbench --threads=16 --time=30 --validate=on memory run
```

Kun tiedosto oli kopioitu containereihin, pystyttiin suorittamaan skripti alla olevalla komennolla. Skripti tallensi tuloksen result.txt-tiedostoon talteen. Alla oleva syntaksi tulostaa samaan aikaan tuloksen konsoliin ja tallentaa tiedostoon. Ensin kuitenkin annettiin suoritusoikeus skriptitiedostolle.

```
chmod +x test.sh
./test.sh |& tee result.txt
```

Koska koneissa oltiin kiinni yhteydessä SSH:lla, pystyttiin kopioimaan tuloste suoraan konsolista omalle koneelle. Suositeltavaa on aina tehdä tuloste, jotta vikatilanteessa on olemassa jotain tallessa.

Testiohjelma loi kiintolevylle yhden gigatavun testitiedoston, jota vasten tiedostojärjestelmätestit suoritettiin. Ensin mitattiin sequental nopeutta. Tässä testi kirjoitti ja luki satunnaisia merkkejä sarjassa testitiedoston sisällä. Random testit tekivät kirjoitus- ja lukutapahtumia eri kohtiin testitiedostoa. Lopuksi random testi suoritti rinnakkaisia luku tai kirjoitus pyyntöjä satunnaisiin kohtiin testidataa. Näin luotiin todellista tiedostojärjestelmän kuormaa simuloiva tilanne.

Taulukon 3 vihreä väri tarkoittaa testin nopeinta tulosta. Punainen väri kertoo hitaimmasta testituloksesta. Virtuaalikoneen testin tulokset jätettiin värjäyksen ulkopuolelle.

Ensimmäinen huomio taulukon 3 tuloksista, oli LXD:n hitaus kirjoitukseen. Johtuiko ongelma siitä, testit tehtiin virtuaalikoneessa. Testit toistettiin useaan kertaan, mutta tilanne ei muuttunut miksikään. Kokeilun vuoksi LXD asennettiin niin snap paketoituna kuin repository jakeluna. Kuitenkin asennuksesta riippumatta nopeudet olivat samaa luokkaa. CPU testin tulos oli vertailun nopein.

Rkt oli tasapainoisen nopea. Seq kirjoitus sekä ylikirjoitus olivat testin nopeimmat. Myös rnd luku/kirjoitus olivat testin nopeimmat. Rkt:n toimintatavaltaan erilainen ympäristö vaati totuttelua. Rkt tukee Docker containereita suoraan ja niitä voidaan suorittaa, kunhan antaa insecure parametrin, jolloin ei tehdä tarkistuksia containerin oikeellisuudesta.

Docker hävisi niin CPU nopeustestissä kuin muistin nopeustestissä.

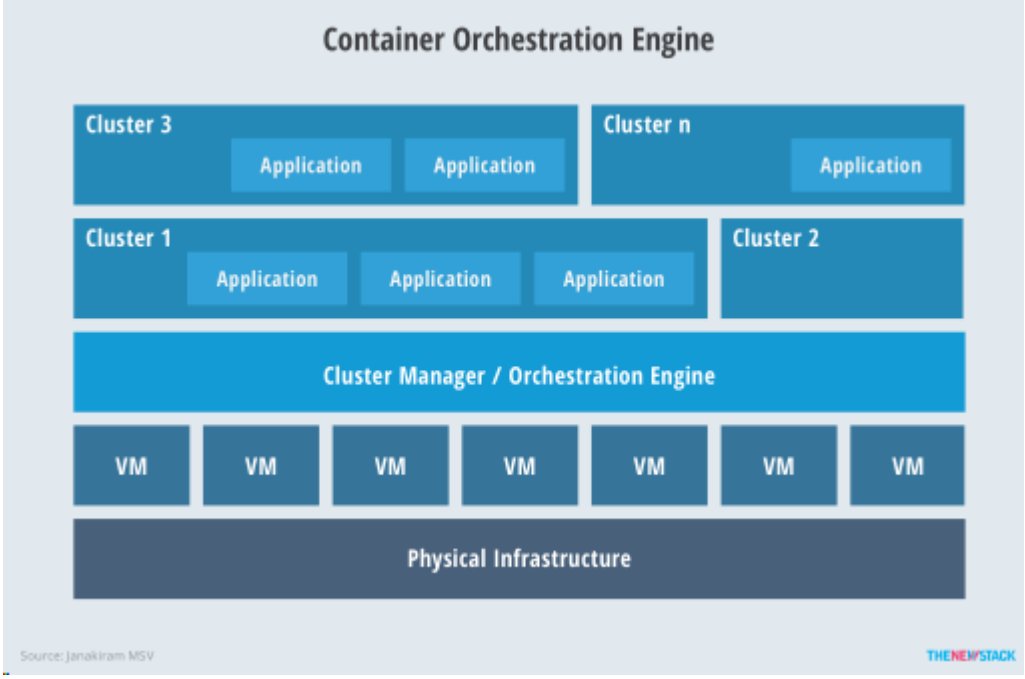
Tulokset olivat kautta linjan virtuaalikoneella paremmat kuin containereilla. Käytössä eroja ei kuitenkaan huomaa, ja näissä testeissä otettiin huomioon vain laskentateho.

Taulukko 3. Sysbench tulokset

Testit	**Yksikkö**	**Ubuntu**	**Docker**	**rkt**	**LXD**
Seq kirjoitus	MiB/s	392,81	403,20	**406,58**	**279,51**
Seq ylikirjoitus	MiB/s	437,60	374,55	**442,93**	**259,57**
Seq luku	MiB/s	14602,27	14135,79	**14015,45**	**14201,64**
Rnd luku	MiB/s	13708,67	**13182,58**	**12972,14**	13163,57
Rnd kirjoitus	MiB/s	69,71	**71,00**	69,62	**21,89**
Rnd luku/kirjoitus					
- **Read**	MiB/s	64,10	65,52	**66,25**	**17,00**
- **Write**	MiB/s	42,74	43,68	**44,18**	**11,34**
Keskiarvo	MiB/s	29317,90	**28276,32**	28017,15	**27954,52**
CPU nopeus	Events/s	16385,26	**16307,85**	16320,30	**16380,49**
Muistin nopeus	MiB/s	8983,82	**8919,70**	**8963,42**	8949,80

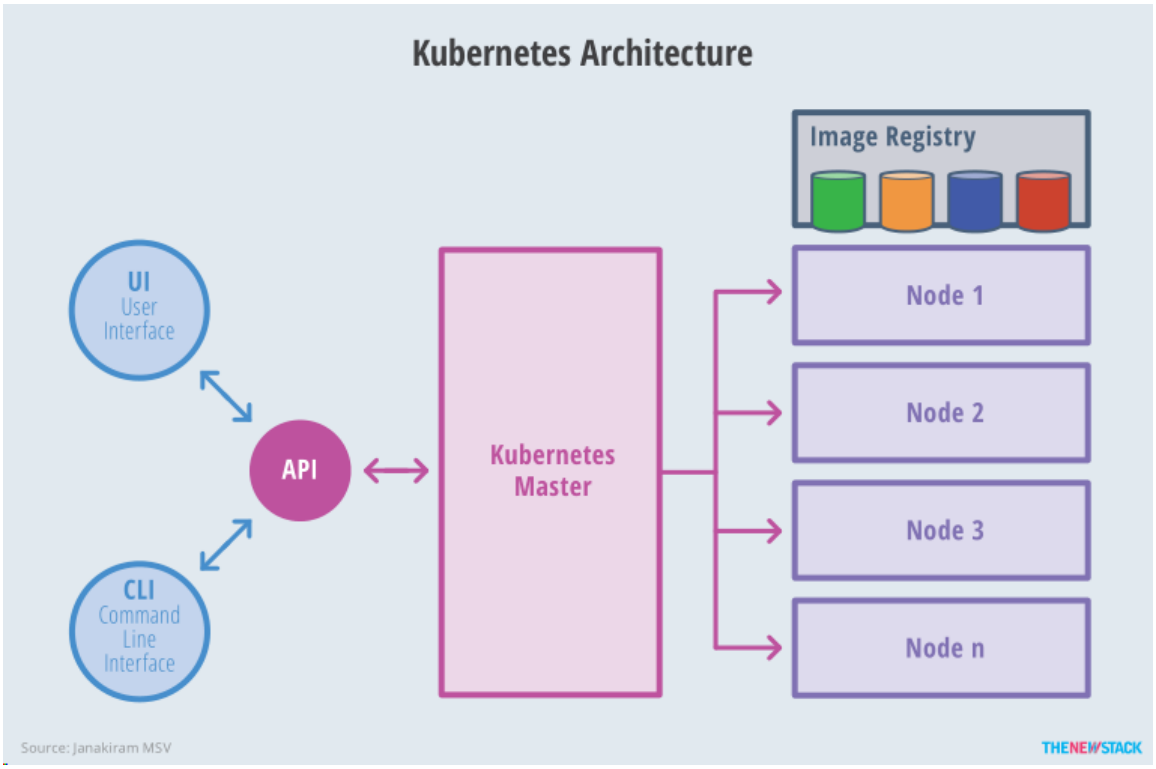
9 Kubernetes

Kubernetes on containereiden ja klustereiden orkestrointiin luotu alustamoottori, jolla on suunniteltu ajettavaksi yritystason cloud mahdollistettua ja pilvi laajennettavaa työkuormaa. Kuvassa 6 esitettynä Kuberneteksen rakenne. Se on rakennettu Googlen 15 vuoden kokemukselle ajaa containerissa sovelluksia. Kehittäjät ja IT-operaattorit ovat kääntäneet katseensa containereihin suorittaakseen paketoitua koodia ja liitännäisiä. Tästä on tullut kriittinen osa automatisoitua koodin koontia CI/CD kehitysputkissa. [13.]

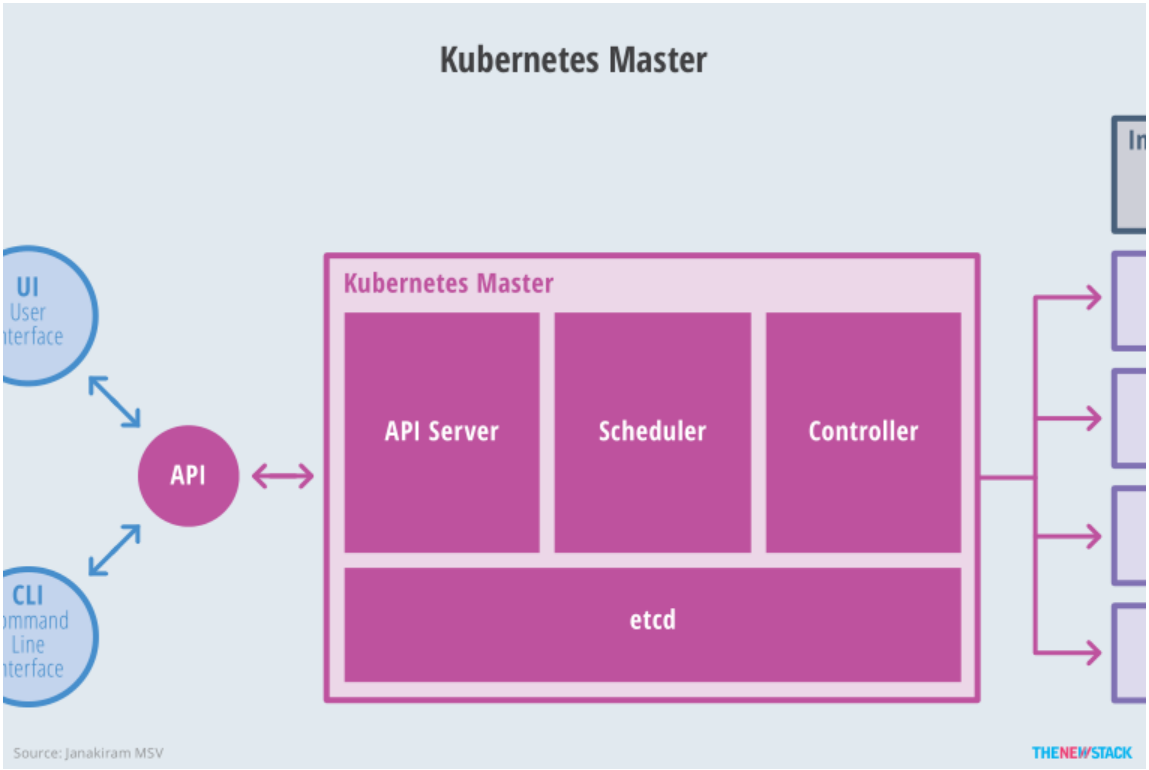


Kuva 6. Klusterin hallinta ja orkestrointi moottori [13]

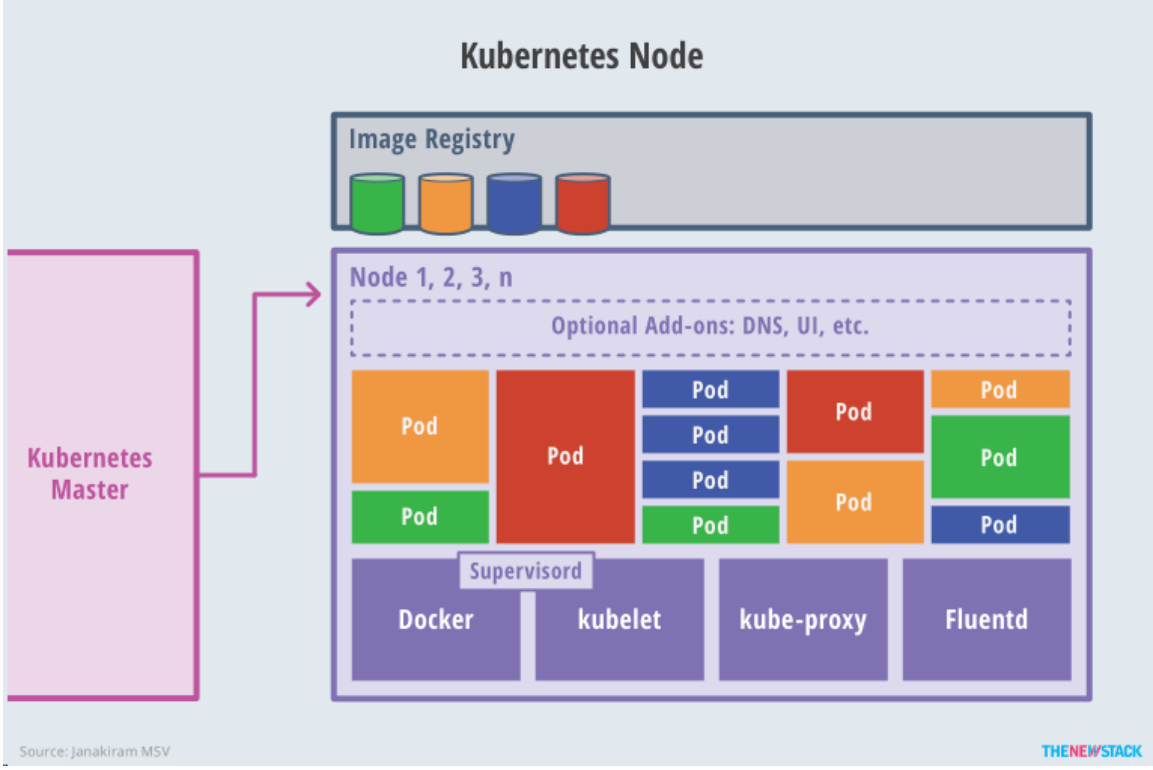
Kubernetes muodostuu kuvan 7 kaltaisesti hallintasolmusta (master node) ja laskentasolmuista (worker node), kuten muutkin jaetun laskennan alustat. Hallintasolmun tehtävänä on julkaista sovelluksen ohjelmointirajapinta (API), ajastaa julkaisuja (deployments) ja hallita klusteria. Hallintasolmun rakenne esitettynä kuvassa 8. Kubernetes tukee alustana Dockeria tai rkt:n. Laskentasolmun tehtävänä on jakaa sovellukselle laskenta-, verkko- ja tiedontallennusresurssit. Laskentasolmun rakenne on esitettynä kuvassa 9. [13.]



Kuva 7. Kubernetes arkkitehtuurin yleiskuva [13]



Kuva 8. Hallintasolmun rakenne [13]



Kuva 9. Laskentasolmun rakenne [13]

Useampi containeri rakentaa yhden podin. Pod toimii Kubernetesin sisäiselle hallinnalle ydinmääritteenä. Pod toimii loogisena rajana containereilla jakaa sama nimiavaruus ja resurssit. Ryhmittelymekanismit podissa kaventavat kuilua virtualisoinnin ja containerin välillä suorittaa prosesseja, jotka riippuvat toisistaan. Suorituksen aikana podeja voi skaalata luomalla niin kutsuttuja replica settejä, jolla voidaan määritellä, että suoritetaan aina tarvittava määrä podeja. [13.]

Replica setillä voidaan toimittaa tarvittava määrä ja saatavuus määrätyllä määrällä podeja. Yksittäinen pod tai replica set voidaan julkaista kuluttajalle käyttämällä serviceä. Käyttämällä nimittäjiä (labels) ja valitsijoita (selectors) voidaan joukko podeja merkitä servicen automaattisesti löydettäväksi. [13.]

Määritteet Kubernetesissa objekteille kuten podit, replica sets ja service jaetaan hallintasolmulle. Riippuen saatavilla olevista resursseista voi hallintasolmu ajastaa käynnistymään määrätyllä laskentasolmulla. Laskentasolmu lataa sitten tarvittavan levykuvan rekisteristä ja hallinnoi alla olevan alustan kanssa containerin käynnistämisestä. [13.]

Etcd on avoimeen lähdekoodiin pohjautuva key-value-tietokanta CoreOS:ltä, joka toimii ainoana lähteenä totuudelle (SSOT) kaikille komponenteille Kubernetes-klusterissa. Hallintasolmu kyselee

etcd:ltä erilaisia arvoja muiden solmujen, podien tai containereiden tilaan liittyen. Tämä arkkitehtuuri tekee Kubernetesistä modulaarisen ja skaalattavan tehden läpinäkymättömän pinnan sovelluksen ja alla olevan infrastruktuurin välille. [13.]

9.1 Rancher

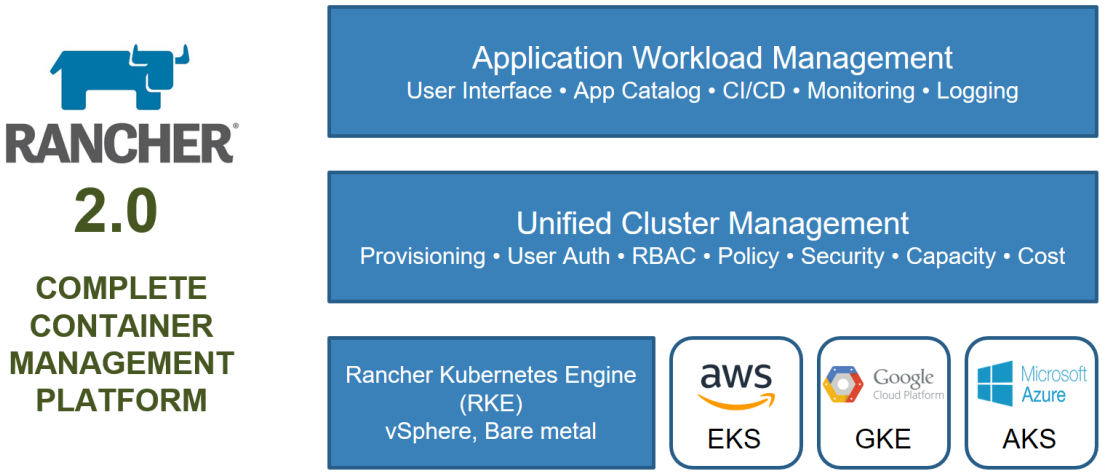
Rancher on containereiden ja klustereiden hallintaan luotu järjestelmä. Se on käynyt läpi useamman erilaisen iteraation. Ensimmäinen versio 1.0 julkaistiin 2016, ja se pohjautui sisäisesti kehitettyyn containereiden hallintajärjestelmään nimeltään Cattle. Tällä pystyi hallitsemaan niin Swarmin, Mesos:n kuin Kubernetesin päällä olevia containereita. Kuitenkin viime vuosina Kubernetes on noussut suosituimmaksi ympäristöksi Dockerin päällä. Tästä syystä Rancher siirtyi täysin käyttämään ja tukemaan Kubernetes alustoja versiossa 2.0. Rancher on nykyään rakennettu täysin Kubernetesin päälle. [14.]

Rancher koostuu kolmesta osasta (Kuva 10). Ensimmäisenä Rancher Kubernetes Engine (RKE), jota voidaan ajaa joko raakaraudalla tai virtuaalisena. Myös alustoissa, joissa ei ole Kubernetestukea, voidaan ajaa virtuaalikone, johon laitetaan RKE. [14.]

Rancherilla voidaan hallita palveluntarjoajien Kubernetes alustoja suoraan yhdestä verkkopalvelusta. Näihin kuuluu Amazonin Elastic Container Service for Kubernetes (EKS), Google Kubernetes Engine (GKE) ja Microsoftin Azure Kubernetes Service (AKS). Unified Cluster Management luo yhtenäisen käyttäjien hallinnan näihin järjestelmiin, jolloin oikeuksien hallinta eri ympäristöjen välillä helpottuu. Pystytään luomaan RBAC- eli roolipohjaisia käyttäjien hallintoja, joka on olemassa natiivisti Kubernetesissa. Rancherissa on myös Projects, jossa voidaan koota eri projekteille omat nimiavaruusrajoitukset, RBAC-säännöt ja olemassa olevia resurssivarauksia. Käyttäjät voivat luoda projekteja ja lisätä käyttäjiä tai ryhmiä niihin. Myöhemmässä vaiheessa on suunnitelmissa lisätä kapasiteettinäkymät ja mahdolliset kululaskennat, mitä Kubernetes klusterit vievät. [14.]

Kolmantena tasona on Application Workload Management. Tällä voidaan hallita helpossa graafisessa ympäristössä Kubernetes palveluita. Käyttäjän ei tarvitse ymmärtää täysin koko Kubernetesin toimintaperiaatetta. Rancherin app catalog-versiosta 1.0 on muokattu tukemaan Kubernetesin Helm chartsia. Tällä pystytään todella tehokkaasti tekemään monimutkaisia palveluita yksinkertaisella template tiedostolla. Rancher yksinkertaistaa tätä prosessia kysymällä vain tarpeellisia muuttujia, jotta palvelu voidaan nostaa pystyyn. Continous integration/continuous delivery (CI/CD) kehitysputket ovat täysin tuettuja Rancherissa. Näistä suosituimpina Jenkins, Drone ja

GitLab. Rancher 2.0 on myös sisäänrakennettuna Jenkinsiin pohjautuva kehitysputkijärjestelmä, jolloin ulkoista ei tarvitse erikseen asentaa. Myös yleisimmät valvontajärjestelmät on tuettu, joita voi käyttää Kubernetesinkin kanssa. [14.]



Kuva 10. Rancher alustan rakenne [14]

Rancherin käyttöönotto ja mukautuvuus on omaa luokkaansa. Hallintajärjestelmän voi nostaa pystyyn minuuteissa, ja minimivaatimuksetkaan ei ole kuin 1 CPU ja 4 GB muistia. Rancherin voi ottaa käyttöön Docker-alustalla käyttäen alla olevaa komentoa. Tämän jälkeen käyttöliittymästä voi käydä noutamassa komennon, jolla voi saman noden ottaa käyttöön laskentasolmuksi tai muita Kubernetes järjestelmiä käyttöön.

```
sudo docker run -d --restart=unless-stopped -p 80:80 -p 443:443
rancher/rancher:latest
```

Tämän jälkeen voi avata hallintapaneelin osoitteessa https://<palvelimen ip>.

9.2 IBM Cloud Private Community Edition

IBM Cloud Private Community edition on samaan käyttötarkoitukseen luotu kuin Rancher. Sillä voi hallita containereita ja klustereita.

IBM Cloud Private (ICP) rakentuu Kuberneteksen ympärille. ICP voidaan nostaa pystyyn omille master-, worker-, proxy- ja vaihtoehtoisille hallinta ja Vulnerability Advisor -nodeille käyttäen Ansiblea käyttävää asennusohjelmaa. Testiympäristö on mahdollista asentaa yhdellekin nodelle; kuitenkin järjestelmävaatimukset nousevat tällaisessa tilanteessa. [15.]

ICP sisältää myös laajan logien hallintaan ja valvontaan olevat järjestelmät esiasennettuna. Suoraan käyttöliittymästä voi päästä käsiksi ELK (Elasticsearch, Logstash ja Kibana) -nipun tuottamiin logeihin. Valvontaa varten asentuu myös uusimmissa versioissa Grafanakin. Näitä voi käyttää, kun tarvitsee valvoa oman sovelluksen toimintaa.

Identiteetti ja oikeuksien hallinta mahdollistaa pääsyn koko alustan järjestelmiin. ICP:ssä on myös mahdollista tehdä tiimejä normaalin Kubernetesin käyttäjien- ja ryhmienhallinnan lisäksi. Tiimi mahdollistaa joukon resursseja niin Kubernetesin sisällä kuin ulkopuolellakin joukolle käyttäjiä perustuen rooleihin. Tämä perustuu samoihin malleihin, joita käytetään IBM UrbanCode Deploossa. [15.]

Haavoittuvuuksien tarkistuksella (Vulnerability Advisor) pystytään valvomaan aktiivisesti mahdollisia ongelmia. Tällä pystytään valvomaan levykuvien prosesseja ja oikeuksiin perustuvia haavoittuvuuksia. Lisäksi pystytään etsimään parannuksia hyviin tapoihin pohjautuen. [15.]

ICP:ssä on sisäisesti rakennettuna yksityinen Docker-levykuvarekisteri. Tämä toimii samalla tavalla kuin Docker Hub, mutta paikallisena. Tälle voidaan antaa rajoituksia, mitkä käyttäjät voivat ladata minkäkin levykuvan. [15.]

Koska ICP on rakennettu Kubernetesin ympärille, niin samoja Helm templateja voidaan käyttää ongelmitta. IBM on tuonut joukon valmiiksi testattuja saataville, mutta Kubernetesin oletus-chartin pystyy lisäämään ICP:hen. Tämä mahdollistaa ICP:n käyttämisen Kubernetes-alustana. [15.]

ICP:n järjestelmävaatimukset ovat huikeat verrattuna Rancheriin. Minimivaatimus yhden noden järjestelmässä on 8 CPU ja 64 GB muistia. Lisäksi tarvitaan vähintään 200 GB kiintolevytilaa. Pitää muistaa, että jos aiotaan käyttää yrityskäytössä, niin on helppo askel siirtyä ilmaisesta Community Editionista maksulliseen Enterprise Editioniin. Tällöin saa IBM:n käyttäjätuen käyttöönsä. Rancherin sisältämät komponentit CI/CD kehitykseen pitää erikseen asentaa ICP:hen.

10 Pohdinta

Container-järjestelmät ovat tehty korjaamaan ongelmia virtuaalikoneiden kanssa. Container on nopeampi ottaa käyttöön kuin virtuaalikone. Resursseja container käyttää vähemmän kuin virtuaalikone.

Työni puolesta olen saanut käyttää useita eri containeri järjestelmiä ja toivottavasti tulevaisuudessakin voin työskennellä näiden ympäristöjen kanssa.

Dockeria olen käyttänyt neljä vuotta, ja olen nähnyt sen kehittyvän todella suureksi tekijäksi ei pelkästään containeri-maailmassa, vaan myös sovelluspuolella. Lisäksi se on muuttunut näkymättömäksi tekijäksi suuremmissa mittakaavoissa, kuten alustana Kubernetekselle. Siinä hyödynnetään Dockerin nimiavaruuksia ja kaikki muu resurssien hallinta toteutetaan toisilla avoimen lähdekoodin komponenteilla.

Rkt:n tulevaisuus on turvattuna, kun Red Hat osti CoreOS:n tammikuussa 2018. Tällä hetkellä rkt:n kehityksen puolella on ollut vielä hiljaista. Rkt tulee vielä nostamaan suosiotaan Kubernetesin tukiessa rkt:ia alustanaan.

LXD:n kehittäjä Canonical voi saada melko isojakin toimijoita taustalleen. LXD:n teknologia ja oletuksena oleva rootfs rajoittaa käyttötarkoituksen virtuaalikoneen korvaajaksi. Tällaiselle järjestelmälle on käyttöä, koska samalla alustalle voidaan pakata virtuaalikonetta tehokkaammin palveluita.

Vertailuissa tuli selväksi, että containeri ei jää jälkeen puhtaalle virtuaalikoneelle. Myös yhdellä virtuaalikoneella ei pysty suorittamaan rinnakkain samanlaisia palveluita ilman että ne sotkisivat toisiaan. Lisäksi containerin luoma tietoturva ja laajennettavuus klusteroiduksi ilman, että lähdekoodia tarvitsee muokata, antaa paljon uusia mahdollisuuksia.

Kubernetesin kehitys viimeisen kahden vuoden aikana on ollut edistyksellistä. Kun aloitin tämän opinnäytetyön, minulla oli olemassa kuva, että Kubernetes on hankala ottaa käyttöön ja olinkin varautunut kirjoittamaan tästä ICP kappaleeseen. Mutta tutustuminen Rancher 2.0:aan pisti minut muuttamaan suhtautumiseni. Voin siis suositella Rancheria pienemmille yrityksille, joilla ei ole antaa ICP:lle sen vaatimia minimivaatimuksia.

Lähteet

[1] K. S. Senthil, Practical LXC and LXD: Linux Container for Virtualization and Orchestration, Apress, 2017.

[2] Docker, "Introduction to Container Security," 18 3. 2015. [Online]. Available: https://www.docker.com/sites/default/files/WP_Intro%20to%20container%20security_03.20.2015%20(1).pdf.

[3] N. Martin, ”A brief history of Docker Containers' overnight success.,” 11 5. 2015. [Online]. Available: http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success.

[4] A. Avram, ”Docker: Automated and Consistent Software Deployments,” 27 3. 2013. [Online]. Available: http://www.infoq.com/news/2013/03/Docker.

[5] C. Swan, "Docker drops LXC as default execution environment," 13 3. 2014. [Online]. Available: http://www.infoq.com/news/2014/03/docker_0_9.

[6] A. Williams, The Docker and Container Ecosystem, The New Stack, 2015.

[7] B. Fihrsman, "Faster and Better Image Distribution with Registry 2.0 and Engine 1.6," 16 4. 2015. [Online]. Available: https://blog.docker.com/2015/04/faster-and-better-image-distribution-with-registry-2-0-and-engine-1-6/.

[8] "What is Docker?," 17 12. 2015. [Online]. Available: https://www.docker.com/what-docker.

[9] S. Grubor, Deployment with Docker, Packt Publishing, 2017.

[10] Red Hat, Inc., "rkt architecture," 2019. [Online]. Available: https://coreos.com/rkt/docs/latest/devel/architecture.html. [Accessed 1 5. 2019].

[11] Canonical Ltd., "LXD - Introduction," 2019. [Online]. Available: https://linuxcontainers.org/lxd/introduction/. [Accessed 30 4. 2019].

[12] N. K. Khare, K. Cochrane and J. S. Chelladhurai, Docker Cookbook, 2nd ed., Packt Publishing, 2018.

[13] J. MSV, Use Cases for Kubernetes, The New Stack, 2016.

[14] Rancher Labs, Inc., Rancher: Technical Architecture, 2018.

[15] IBM Corporation, "IBM Cloud Private v3.1.2 documentation," 2019. [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSBS6K_3.1.2. [Accessed 1 5. 2019].