Palvelinarkkitehtuuri konteilla ja konttienhallintajärjestelmillä

Teemu Koivisto

Helsinki 2.6.2018

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen osasto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen osasto

Tekijä — Författare — Author
Teemu Koivisto

Työn nimi — Arbetets titel — Title

Palvelinarkkitehtuuri konteilla ja konttienhallintajärjestelmillä

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	2.6.2018	20 sivua + liitesivua

Tiivistelmä — Referat — Abstract

Palvelinmäärien kasvaessa on syntynyt tarve tehostaa entisestään niiden käyttöastetta. Kontit ovat uudentyyppinen käyttöjärjestelmätason virtuaalisaation muoto joka tarjoaa virtuaalikoneisiin nähden kevyemmän virtualisaation, yksinkertaisen käyttötotetuksen ja paremman skaalautuvuuden. Sen sijaan että kehittäjät pystyttäisivät ohjelmansa suoraan virtuaalikoneiden päälle voidaan luoda kontti, joka tarjoaa standardin tavan palvelimen luomiseen. Se silloin nopeuttaa kehittäjien työtä sekä parantaa ohjelmistolaatua pilkottaessa isompia kokonaisuuksia mikropalveluihin. Kontit ovat muodostumassa samanlaiseksi standardiksi yksiköksi kuin luokat ovat ohjelmoinnissa.

Konttienhallintajärjestelmällä voidaan järjestellä ja hallinnoida näitä resursseja tehokkaalla ja edistyneellä tavalla, jolloin vuorottajat maksimoivat resurssien käytön automaattisesti klusterissa. Monet pilvipalvelunyhtiöt kuten Amazon Web Services (AWS) tai Google Cloud Platform (GCP) tarjoavat valmiita konttienhallintajärjestelmä-ratkaisuja. Käyn täyssä tutkielmassa läpi konttien kehityksen historiaa ja niiden toteutuksista yleisimmän, Dockerin, toimintapaa sekä konttienhallinnan kehittymistä Googlen sisäisistä järjestelmistä aina vapaan lähdekoodin Kubernetekseen.

ACM Computing Classification System (CCS):
C.2.1 [Computer systems organization]: Architectures

Avainsanat — Nyckelord — Keywords
hajautetut järjestelmät, kontti, konttihallintajärjestelmä, Docker, Kubernetes

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Sisältö

1 Johdanto virtuaalisaatioon

Tietokoneiden hallinnassa on niiden syntymästä asti ollut haasteita. 1960-luvulla työt annettiin yhä reikäkortti-muodossa eikä niiden jakamiseen aika-viipaleina käyttäjilleen ollut helppoa tapaa. IBM ryhtyi tällöin kehittämään käyttöjärjestelmää keskustietokoneille (main frame), joka olisi virtualisoitu muusta käyttöjärjestelmästä. Tätä virtualistoitua käyttöjärjestelmää kutsuttiin virtuaalikoneeksi (virtual machine, VM) jonka hyödyt tulivat nopeasti selviksi; kun kehittäjien ei tarvinnut enää huolehtia sekä laitteistosta että muista käyttöjärjestelmään littyvistä ohjelmistoista tehosti se heti heidän ajankäyttöään. IBM havaitsi kuinka hyödyllinen ja tärkeä virtualisaatio oli ja pian se olikin keskeisessä asemassa tietojärjestelmien kehittyessä. Myöhemmin yritykset kuten VMware jatkoivat virtuaalisaation kehittämistä [10].

Virtuaalikoneet siis voidaan ajatella käyttöjärjestelmänä käyttöjärjestelmän päällä ja ne jaetaan kahteen eri luokkaan: täysi virtualisaatio (full virtualization) ja paravirtualisaatio (paravirtualization). Täydessä virtualisaatiossa vieraskäyttöjärjestelmä toimii isäntäkäyttöjärjestelmän päällä kaikkien laitteistopyyntöjen kääntyessä virtuaalisiksi. Näin ollen järjestelmäkutsut käännetään aina kuvitteelliselle laitteistotasolle asti. Paravirtualisaatiossa vieraskäyttöjärjestelmä on tietoinen virtualisaatiosta ja voi siten kutsua sille annettuja ajureita laitteistokäskyjen sijaan [17].

Useimmiten virtuaalikoneisiin liittyy myös hyperviisori (hypervisor) joka on varta vasten luotu järjestelmä tai ohjelma virtuaalikoneiden hallinnointiin ja ajamiseen. Tämä hyperviisori voi olla myös kahta eri tyyppiä: tyyppiä 1, joka toimii suoraan laitteiston päällä kuten Xen tai tyyppiä 2, joka toimii toisen käyttöjärjestelmän päällä ohjelmana kuten VirtualBox [9].

Tässä tutkielmassa tutkitaan kirjallisuuskatsauksen keinoin uudenlaisen virtuaalisaation muodon, konttien, kehittymistä sekä niiden toimintapaa. Käyn myös lävitse eri toteutuksia konteista keskittyen kuitenkin niistä suosituimpaan, Dockeriin. Selvitän myös miten konttienhallintajärjestelmillä voidaan parantaa palvelinarkkitehtuuria sekä hyötyä Googlen vuosien työstä niiden optimoimisessa. Lopuksi vielä selvitän eri vaihtoehtoja konttien ajamiseen GCP ja AWS pilvipalveluntarjoajilla.

2 Kontit

2.1 Johdanto

Konttien voidaan katsoa olevan uusi käyttöjärjestelmätason virtualisaation muoto joka toimii ilman omaa käyttöjärjestelmää kutsuen suoraan isäntäkäyttöjärjestelmän ydintä. Kontti voi olla kahdessa eri tilassa: joko levossa tai ajossa.

Levossa kontti on vain kokoelma tiedostoja levyllä joita kutsutaan konttikuvaksi (container image) ja se sisältää kaiken kontin ajamiseen tarvittavan tiedon. Kun kontti käynnistetään konttikuva puretaan ja suoritetaan prosessina, joka on hyvin samankaltainen normaaliin käyttäjäprosessiin verrattuna. Erona tavallisiin proses-

seihin esimerkiksi Linuxissa on, että prosessi luodaan `clone()`-komennolla `fork()`-komennon sijaan jonka lisäksi prosessi ja sen lapsiprosessit ovat eristetty muusta käyttöjärjestelmästä. Muutoin kontti on kuin mikä tahansa muukin prosessi ja se saa tehdää käyttöjärjestelmäkutsuja suoraan käyttöjärjestelmän ytimeen [32].

Ideana kontti eli käyttöjärjestelmätason virtualisaatio on suhteellisen vanha ja sen alkuperä voidaan nähdä Unixin `chroot`-komennossa vuodelta 1979. Myöhemmin siitä kehitettiin versiot luotiin useisiin eri käyttöjärjestelmiin kuten FreeBSD `jail`:it (1998) ja Solaris `zone`:t (2004). Linuxin muodostuessa dominoivaksi vapaan lähdekoodin käyttöjärjestelmäksi konttien kehitys keskittyi sen ympärille. Kuitenkin vasta kun Linuxin ytimeen lisättiin parempia eristämisen keinoja kuten `cgroup` vuonna 2007 ja nimiavaruudet vuonna 2013 kontit saivat nykyisen muotonsa [8].

Konteista on monia eri toteutuksia kuten Docker, CRI-O, Railcar, RKT ja LXC/LXD. Suosituin toteutus kuitenkin lienee Docker. Kontteja yritetään standardisoida Open Container Initiave (OCI) järjestön kautta, jonka tavoitteena on että kaikki kontit tarjoaisivat hyvin samantapaisen käyttörajapinnan (API) [31].

2.2 Dockerin toimintatapa

Docker syntyi avoimen lähdekoodin projektina dotCloud-nimisessä PaaS (Platform as a Service) pilvipalveluyhtiössä vuonna 2013 yhtiön pilvipalveluiden jatkeena. Yhdeksän kuukautta sen luomisen jälkeen yhtiö liittyi Linux Foundation järjestöön, vaihtoi fokuksensa Dockeriin ja muutti nimensä Docker Inc:ksi [32].

Docker koostuu monesta eri osasta joista itse kontti on vain yksi osa: Docker Engine on konttienhallintaohjelma jolla kontteja suoritetaan, muita vastaavia ohjelmia ovat esimerkiksi RKT, CRI-O tai LXD. Ne osaavat ajaa tietyn tyyppisiä konttikuvia kuten Docker Image joista ne sitten luovat kontteja, jotka ovat vain tavallisia prosesseja. Kuvien luomiseksi käytetään monenlaisia formaatteja mutta joista Dockerfile lienee suosituin ja näitä kuvia säilytetään ja ylläpidetään jossain kuvarekisterissä joka voi olla kaikille avoin kuten DockerHub tai suljettu. Docker Engine ei itse suorita kontteja vaan kutsuu jotain konttienajo-ohjelmaa kuten RunC [31].

Konttien alkuperän voidaan nähdä olevan Unixin `chroot`-komennossa (change root) joka julkaistiin osana Unixin versiota 7 vuonna 1979. `Chroot`:in avulla prosessin ja sen lapsien juurihakemisto (root directory) oli mahdollista muuttaa näin ollen rajaten sen pääsyä muihin järjestelmän osiin [19]. Prosessi tarvitsee tällöin omat binäärinsä sekä vaaditut jaetut kirjastot esimerkiksi `/bin/bash` -komentotulkin tai normaalien komentojen kuten `ls`:n suorittamiseen [37]. Jo alusta asti `chroot`:issa on ollut paljon turvallisuusongelmia koska esimerkiksi kahden `chroot`:in tekeminen saattaa vapauttaa prosessin alkuperäisestä `chroot`:ista [28]. Prossesien resursseja ei myöskään mitenkään rajata ja ne voivat kuluttaa kaiken käyttöjärjestelmässä olevan muistin ja prossesointitehon [44].

Konttien kehityskulku sittemmin on kulkenut monen eri käyttöjärjestelmän kautta. Esimerkiksi FreeBSD:n `jail`:it tulivat vuonna 1998 (tästä syntyi termi jailbreak)

ja Solariksen `zone`:t vuonna 2004. Lopulta kun Linuxista oli tullut dominoiva vapaan lähdekoodin käyttöjärjestelmä `chroot` yhdistettynä muihin uusiin eristyksen keinoihin oli luomassa kontiksi miellettävää kokonaisuutta [8].

Toinen konttien kannalta tärkeä uusi Linuxin järjestelmäkutsu oli `cgroup` (control groups). Se mahdollisti prossesin ja sen lapsien luonnin varaten niille tietyn määrän resursseja (muisti, CPU, I/O, tietoliikennekaista) joiden rajoissa niiden oli toimittava. `Cgroup` myös antaa paljon tietoa näiden prosessien resurssien kulutuksesta, mitä esimerkiksi Docker käyttää omassa monitoroinnissaan [32].

Linuxin nimiavaruuksien (name spaces) avulla käyttöjärjestelmän resurssit voidaan pilkkoa eri tavalla jaettuihin osiin ja rajata ne vain nimiavaruuden sisälle oleville prosesseille. Nimiavaruudet voivat olla eri tasoisia kuten PID- tai verkkonimiavaruuksia. PID-nimiavaruus esimerkiksi mahdollistaa täysin eristetyn prosessi-ID nimiavaruuden jonka avulla prosessit voidaan piilottaa muilta käyttöjärjestelmän prosessseilta. Verkkonimiavaruus mahdollistaa verkkoon liittyvien resurssien jaon tai piilottamisen ja esimerkiksi Dockerissa kaikki kontit jakavat verkkonimiavaruuden jotta IP-osoitteet olisi jaettuja saman palvelimen konttien kesken [26].

Näiden kolmen tärkeimmän Linuxin ytimen ominaisuuden yhdistelmää käyttöjärjestelmätason-virtualisaationa voidaan kutsua *kontiksi*. Nykyisen muotonsa ne saavuttivat vasta nimiavaruuksien saapuessa Linuxin ytimen päähaaraan (mainline) vuonna 2013. Niiden käyttämiseksi on luotu erillisiä kirjastoja jotka tarjoaisivat helpomman käyttöliittymän konttien käyttöön joista ensimmäinen oli LinuX Containers (LXC)[1] [32]. LXC:lle on olemasssa myös oma konttienhallintaohjelma nimeltä LXD[2] joka on samantapainen kuten Docker mutta pelkästään LXC-konttien ajamiselle. Se on Dockeria virtuaalikonemaisempi ja siten soveltuu eri käyttötarkoitukseen [36].

Docker alunperin käytti LXC:ää mutta vaihtoi sen sittemmin RunC-kirjastoon[3] [24]. RunC on, kuten LXC, pelkkä työkalu konttien ajamiselle jonka tarkoituksena on toteuttaa OCI:n mukainen standardi rajapinta [25].

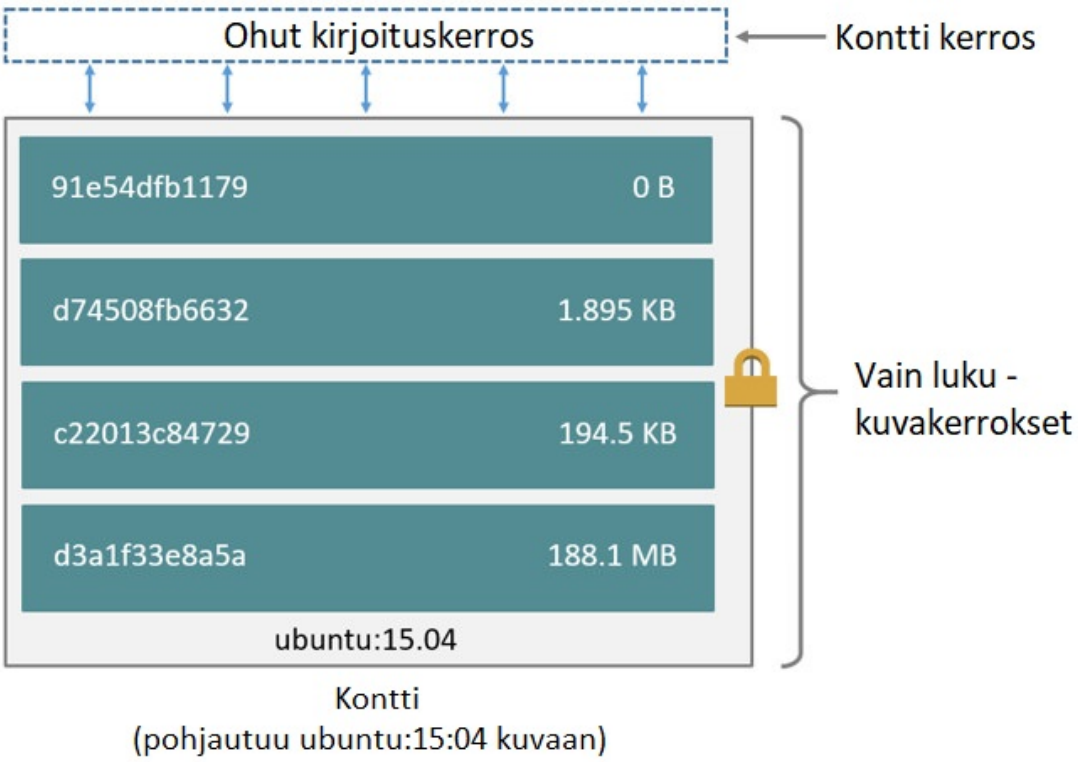
Tämän lisäksi Docker myös käyttää Advanced Multi-Layered Unification Filesystem (AuFS) kerrostettua tiedostojärjestelmää joka voi näkymättömästi kerrostaa useita tiedostojärjestelmiä päällekkäin. Sen avulla Docker voi jakaa samaa monen kontin käyttämää tiedostokerrosta vain luku -muodossa kaikille sitä tarvitseville konteille. Kontille itselle tarjotaan vain ohutta omaa kirjoitettavaa kerrosta joka on nopea luoda. Kontit kuitenkin suositellaan luotavan ilman kirjoituskerrosta eli tilattomana ja mahdolliset levylle kirjoittamiset tehtävän vain kontille varta vasten tarjottuun levy-asemaan. Tilattomia kontteja on moninverroin helpompi kehittää ja korjata. AuFS:in copy-on-write tiedostojärjestelmä tarjoaa myös helpon tavan seurata tiedostomuutoksia samaan tapaan kuin Git:in `diff` [32].

Ongelmia AuFS:n käytössä tuo sen hitaus verrattuna tavalliseen tiedostojärjestelmään. Myös kerroksittainen kirjoittaminen kasvattaa kontteja ajan kanssa jolloin on tarpeen niiden ajoittainen uudelleenluonti kerroksien yhdistämiseksi [19].

[1] `https://linuxcontainers.org/`

[2] `https://github.com/lxc/lxd`

[3] `https://github.com/opencontainers/runc`



Kuva 1: Dockerin kerroksista koostuva kuva. Huomaa vain yksi ohut kirjoituskerros (Docker 2018)

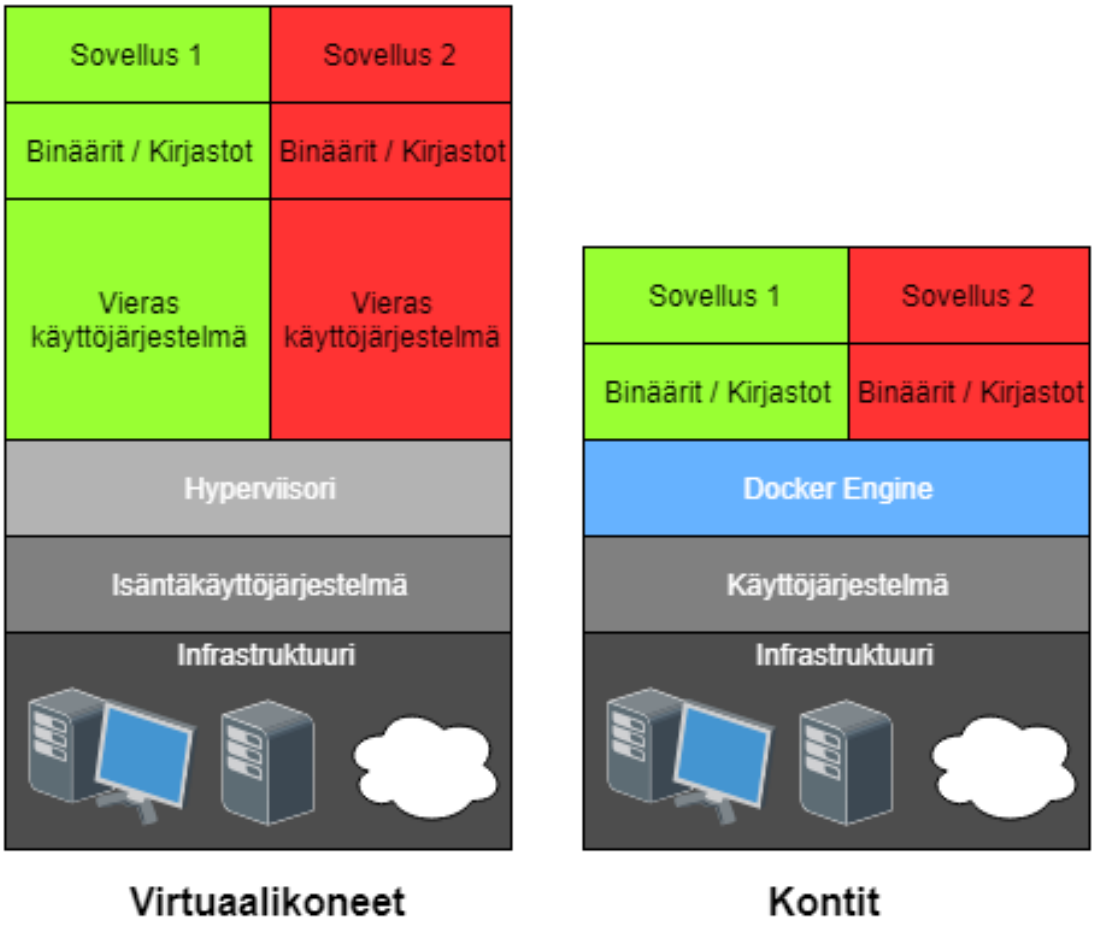
Docker ja muut kontit käyttävät myös monia eri kirjastoja paremman eristyksen ja turvallisuuden takaamiseksi kuten SELinux tai AppArmor [31].

2.3 Kontit virtuaalikoneisiin verrattuna

Kontit tarjoavat virtuaalikoneisiin verrattuna kevyemmän virtuaalisaation ja nopeamman tavan luoda ja käynnistää uusia instansseja. Ehkä tärkeimpänä ominaisuutena kuitenkin voidaan pitää konttikuvia jotka mahdollistavat helpon tavan luoda ja muokata kontteja esimerkiksi juuri Dockerin ylläpitämän rekisterin DockerHubin kautta [32]. Tämä modulaarisuus hajautetuissa järjestelmissä on verrattavissa olio-ohjelmoinnin yleistymiseen 90-luvulla. Standardiyksikön kuten kontin käyttö mahdollistaa tehokkaamman systeemien hallinnan ja yleisten suunnittelumallien käytön [12].

Kontit myös mahdollistavat helpomman tavan luoda mikropalveluarkkitehtuureja jossa ison monoliittisen sovelluksen sijaan logiikka on hajautettu omiksi erillisiksi palveluiksi [31, 12]. Tämä tuo etuja sovelluksen ylläpidossa ja kehittämisessä kun eri osia voi vaihtaa sekä kehittää toisista riippumatta mikä myös tekee siitä skaalautuvamman [33].

Virtuaalikoneilla on kontteihin verrattuna etuna parempi käyttöjärjestelmätason eris-



Kuva 2: Virtuaalikoneiden ja konttien virtuaalisaation ero (Tech Glimpse 2017)

tys ja ne pystyvät käyttämään kehittyneempiä eristyksen muotoja kuten Intelin VT tai AMD-V [14]. Intelin VT teknologia mahdollistaa erilaisia tehokkuusparannuksia virtualisointiin. VT-x esimerkiksi antaa prosessorille 10 uutta konekäskyä [42] ja moodin jolla vieraskäyttöjärjestelmä voi suorittaa ytimen tasolla 0 isäntäkäyttöjärjestelmän ollen yhä samalla suojattu [23]. AMD-V tuo samanlaisia lisäominaisuuksia prosessorille joka tehostaa virtualisoitujen komentojen suorittamista [29].

2.4 Konttien ja virtuaalikoneiden tehokkuusvertailu

Konttien ja virtuaalikoneiden välisiä tehokkuuseroja on tutkittu jonkin verran mutta tuloksia on ollut vaikea tulkita virtuaalisaatioiden erilaisten ominaisuuksien takia. *Measuring Docker Performance: What a mess!!!* (2017) -tutkimuspaperin mukaan lopullista päätelmää on vaikea antaa koska esimerkiksi ei ole olemassa tarpeeksi laajoja tai vakaita työkaluja konttien suorituskyvyn testaukseen [13].

Aiheesta hivenen vanhemmassa tutkimuspaperissa, *Containers and Virtual Machines at Scale: A Comparative Study* (2016), tullaan johtopäätökseen että virtuaalikoneiden

ja konttien välillä on tehokkuuseroja, mutta nuo erot selittyvät enemmän virtuaalisointien erilaisuudesta kuin toisen virtualisaation paremmuudesta. Kontit hyötyvät suorista kutsuista suoraan käyttöjärjestelmän ytimeen sekä pienemmästä koosta mutta samalla myös kärsivät heikommasta eristyksestä. Virtuaalikoneilla on tarkat resurssirajat kun taas kontit voivat hyödyntää muiden konttien vapaana olevia resursseja löysemmillä rajoituksilla. Tämän vuoksi yksi suosittu tapa ajaa kontteja on virtuaalikoneiden sisällä näin hyödyntäen virtuaalikoneen turvallisuutta ja samalla käyttäen konttien avulla kaikkia resursseja tehokkaasti hyväksi. Kehitteillä on myös uusi virtuaalikonetyyppi, kevyt virtuaalikone (lightweight VM), jonka tarkoituksena olisi toteuttaa konttien ominaisuudet virtuaalikoneiden eristyksellä [39].

3 Konttienhallintajärjestelmät

3.1 Johdanto

Internetin yleistyessä ja teknologian kehityksen kiihtyessä ovat resurssit palvelimien käytössä kasvaneet huomattavasti vuosikymmenten aikana. Esimerkiksi yksi Googlen uusi palvelinsali koostuu neljästä jalkapallokentällisestä ja kahdensadan miljoonan euron edestä palvelimia [45]. Yksi suurista ongelmista valtavan palvelinmäärän hallinnassa on niiden resurssien optimointi mahdollisimman tehokkaasti. Google on ollut edelläkävijä tässä optimoinnissa ja he ovat kehittäneet järjestelmiä, joiden avulla he ovat huomattavasti parantaneet palvelimiensa käyttöastetta [11].

Näistä järjestelmistä ensimmäinen oli Borg joka luotiin vuosina 2003-2004 korvaamaan joitain Googlen sisäisiä palveluita kuten Babysitter ja Global Work Queue, jotka olivat luotu hoitamaan pitkäaikaisia palveluita ja sarja-ajoja. Borgin tarkoituksena oli tehostaa resurssien käyttöä jonka mahdollisti konttimaisten ominaisuuksien kuten `cgroup`:in tulo Linuxin ytimeen, johon Google oli isolta osin vaikuttamassa. Sen käytön kasvaessa sitä kehitettiin palvelemaan monta eri Googlen sisäistä tiimiä ja sille luotiin laaja määrä työkaluja sekä palveluita helpottamaan sen käyttöä [11].

Borg toi kustannushyötyjä muun muassa käyttämällä jo varattuja mutta käyttämättömiä palvelinresursseja muihin lyhytaikaisiin sarja-ajoihin. Siinä ei myöskään haluttu maksaa virtuaalikoneiden aiheuttamia tehokkuusmenetyksiä joten useimmiten ohjelmat ajettiin Borgissa laitteistotasolla käyttäen kuitenkin konttimaisia eristyskeinoja prosessien eristämiseen. Tämä ei ole kuitenkin verrattavissa täyteen konttitotetukseen kuten Dockeriin [43].

Keskitetty hallintajärjestelmä myös mahdollisti kehittäjien keskittymisen pelkästään sovellusten kehittämiseen jättäen laitteistoriippuvuuksien ja aikatauluttamisen järjestelmän hoidettavaksi. Tämä muutti kehityksen laite-orientoituneesta sovellusorientoituneeksi [11].

Kuitenkin Borgin käytössä havaittiin pian ongelmia, joiden vuoksi ryhdyttiin luomaan uutta konttienhallintajärjestelmää näiden puutteiden korjaamiseksi. Syntyi Omega, jonka tarkoituksena oli edistää resurssien käyttöä tehokkaammalla aikatauluttajalla

(scheduler). Borgin monoliittisen aikatauluttajan sijaan Omegassa se oli hajautettu moneksi erilliseksi aikatauluttajaksi joilla kaikilla oli pääsy klusterin Paxos-pohjaiseen tilaan. Tämä mahdollisti monipuolisempien aikataulutus-algoritmien käytön ja näin ollen tehokkaamman resurssien hyödyntämisen [38].

Omega ei kuitenkaan lopulta korvannut Borgia mutta monet sen innovoimista ratkaisuista lisättiin takaisin Borgiin ja käytettiin perustana Googlen seuraavan järjestelmän, Kuberneteksen, luomiseen.

Kubernetes luotiin aikana jolloin ulkopuolinen kiinnostus kontteja ja konttienhallinta-järjestelmiä kohtaan oli kiihtymässä ja Google tarjosi jo omaa kasvavaa pilvipalvelua Google Cloud Platform:ia (GCP). Edellisistä järjestelmistä poiketen Google päätti luoda Kuberneteksen alusta alkaen avoimen lähdekoodin projektiksi joka olisi Borgia kehittäjäystävällisempi sekä korjaisi joitain sen puutteita kuten yhtä IP-osoitetta per laite sekä töiden jakamista joustavammalla tavalla [11].

Kubernetes on jatkuvassa määrin muodostumassa samanlaiseksi avoimen lähdekoodin standardiksi konttienhallintajärjestelmissä kuten Linux käyttöjärjestelmille. Cloud Native Computing Foundation (CNCF), johon suurimmat teknologiayhtiöt kaikki kuuluvat, hyväksyi standardoidun Kuberneteksen API:n jonka jokainen jäsen on velvoitettu toteuttamaan omassa Kubernetes-pohjaisessa konttienhallinta-palvelussa [16].

Näiden lisäksi on monia muita konttienhallintajärjestelmiä kuten DockerSwarm, Mesos ja OpenStack mutta joista Docker ja MesosSphere ovat jo luvanneet tukensa Kubernetekselle kuten moni muukin pilvipalveluntarjoaja [31].

Jotta järjestelmien kehityskulku selventyisi käyn läpi näiden järjestelmien kehittymisen ja toiminnallisuuden vertailun.

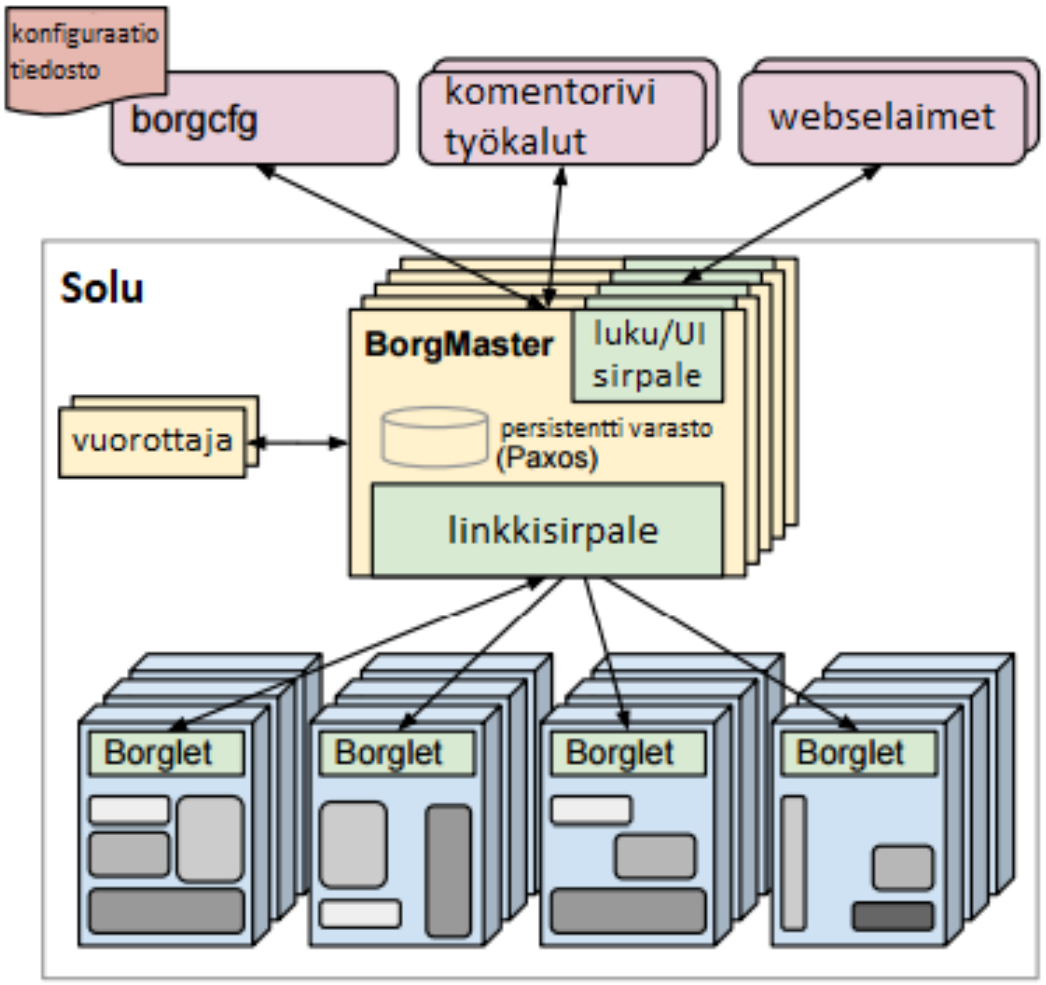
3.2 Borg

Borg syntyi Googlen sisällä korvaamaan sisäisiä sovelluksia kuten BabySitter ja Global Work Queue, joilla hallittiin pitkäaikaisia prosesseja ja sarja-ajoja. Sen tarkoituksena oli yhtenäistää ja tehostaa resurssien käyttöä Googlen palvelimilla pakkaamalla suoritettavia prosesseja paremmin ja käyttämällä kehittyneitä aikataulutus-metodeja.

Alussa Borg toimi suoraan palvelimien päällä ilman virtualisointia tehokkuuden maksimoimiseksi. Kuitenkin käyttöjärjestelmäriippuvuudet ja eri ohjelmistoversioiden hallinta teki kehittämisestä ajoittain vaikeaa kuten IBM huomasi jo aikoinaan.

Borgin keskiössä on BorgMaster-palvelin joka hallinnoi koko klusteria. Yksittäinen klusteri joita kutsutaan myös soluiksi (cell) koostuu Googlella keskimäärin 10 000 palvelimesta. BorgMaster on replikoitu viidelle kopiolle jotka pitävät yllä viimeisintä kopiota klusterin tilasta ja joista valitaan uusi pääsolmu nykyisen kaatuessa. Kiireinen BorgMaster käyttää 10-14 prosessoriydintä ja 50 GiB RAM-muistia.

Eräs esitys Borgin arkkitehtuurista on kuvattu kuvassa 3. Klusteri koostuu solmuista (node) jotka vastaavat yksittäistä fyysistä tai virtuaalista palvelinta. Solmun sisällä



Kuva 3: Borg kaavio Googlen julkaisemasta paperista. Linkkisirpale huolehtii kommunikaatiosta Borglettien välillä (Google 2015)

palvelimen resurssit ovat varattu allokaatioiksi (alloc, lyhenne sanasta allocation) ja sillä suoritetaan vähintäänkin Borglet-prosessia. Borglet-prosessi suorittaa sille annettuja tehtäviä allokaatioissa ja raportoi tilaansa BorgMasterille.

Ensimmäiset Borgin versiot käyttivät yksinkertaista synkronista silmukkaa joka hyväksyi pyynnöt, aikataulutti tehtävät ja kommunikoi Borglettien kanssa. Solujen kasvaessa tämä pilkottiin omaksi palveluksi joka on replikoitu kuten pääsolmu ja joka koko ajan hakee uusinta tilaa pääsolmusta, päivittää kopionsa, aikatauluttaa tehtäviä ja raportoi pääsolmua noista muutoksista. Pääsolmu sitten hyväksyy tai hylkää muutokset jos ne perustuvat vanhentuneeseen tilaan. Tämä muistuttaa hyvin paljon Omegan optimistista samanaikaisuudenhallintaa johon se osaltaan perustuu.

Tehtävä (task) on yksittäinen suoritettava ohjelma joka annetaan samanlaisena kaikille sille varatuille solmuille. Borg yrittää maksimoida juurikin tehtävien ajon

mahdollisimman tehokkaalla tavalla. Suoritettavat tehtävät ovat Borgissa eristetty toisistaan konttimaisilla keinoilla kuten nimiavaruuksilla vaikka ne ei välttämättä käyttäisikään varsinaisia kontteja kuten Dockeria.

Tehtävät taas kuuluvat aina yhteen työhön (job) joka annetaan BorgMasterille suoritettavaksi RPC-rajapinnan kautta niitä varten luodussa BCL-kielisessä konfiguraatiotiedostossa joka muistuttaa Dockerfileä. Se kuvaa kaiken työn suoritukseen tarvittavan tiedon ja se tallennetaan Borgin Paxos-varastoon ja jonka suorituksen kulkua BorgMaster koko ajan päivittää.

Eräs Borgin ongelmallisuutta kuvaava ominaisuus on ollut mahdollisuus ajaa tehtäviä allokaatioiden ulkopuolella, jonka on kerrottu olleen syynä suureen määrään harmia: *"Borg also allows top-level application containers to run outside allocs; this has been a source of much inconvenience."*

Ehkä kuitenkin tärkein osa Borgia on sen sisäinen logiikka resurssien mahdollisimman tehokkaaseen käyttöön erilaisten keinojen avulla kuten tiiviimpi ohjelmien pakkaaminen (bin packing), kapasiteetin hyödyntäminen ajamalla lyhytaikaisia sarja-ajoja jo varatuilla mutta vähän käytetyillä resursseilla sekä optimoitu resurssien varaaminen sovittaen ajettavat tehtävät niille sopiville solmuille. Googlen mittakaavassa nämä tehokkuusparannukset tuovat jo huomattavia kustannusvähennyksiä.

Esimerkkejä palveluista joita Borgilla on ajettu ja yhä ajetaan on Googlen hakupalvelu, Google Docs ja Gmail. Borg on edelleen Googlen sisällä käytetyin klusterinhallintajärjestelmä ja sen monimutkaisuus mahdollistaa myös töiden monipuolisen konfiguroinnin mikä ei välttämättä Kuberneteksessa olisi mahdollista [43].

Ongelmiksi Borgissa on muodostunut esimerkiksi tehtävien ryhmittely pelkkien töiden avulla. Käyttäjä ei voi erotella tuotanto- ja testiympäristöä toisistaan muuten kuin luovilla nimeämisratkaisuilla. Tämä on johtanut taas työkalujen luontiin jotka parsivat nimistä laajemman palvelintopologian, mikä taas on tuottanut lisää monimutkaisuutta järjestelmään [45].

Toinen iso ongelma on ollut yhden IP-osoitteen rajoite laitetta kohti. Borgissa samalla laitteella olevat tehtävät jakavat kaikki saman IP-osoitteen joten ne myös jakavat porttiavaruuden. Näin ollen Borgin tulee allokoida ja käsitellä niitäkin resursseina joita tulee varata ja vapauttaa. Tehtäviä jotka käyttävät samaa porttia kuten 80 ei voida näin ollen ajaa samalla laitteella muuttamatta niiden konfiguraatiota.

Kolmas iso ongelma on ollut Borgin optimointi tehokäyttäjille tavallisten käyttäjien kustannuksella. Borg tukee laajan määrän eri konfiguraatio-asetuksia monille vaativille Googlen sisäisille palveluille joka tekee sen API:n kehityksestä vaikeaa. Ratkaisuna on ollut työkalujen luonti jotka automatisoivat ja hallinnoivat näitä käyttäjän puolesta.

Hyvinä puolina kuitenkin on huomattu kuinka käytännöllisiä allokaatiot ovat olleet, miten klusterin hallinta on enemmän kuin tehtävien hallintaa ja että introspektio, jälkihavainnointi, on hyvin tärkeää [43].

3.3 Omega

Ajan saatossa ja Borgin kasvaessa yhä monimutkaisemmaksi havaittiin sen käytössä joitain ongelmia joita Google halusi ratkaista. Borgissa ajettava kuorma oli hyvin vaihtelevaa, joten sen mahdollisimman tehokas aikatauluttaminen vaati monipuolisia ominaisuuksia Borgin monoliittiselta aikatauluttajalta. Tämä monimutkaisuus oli kuitenkin muodostumassa yhä vaikeammaksi hallita ja osana sen uudelleenkirjoitusta Google päätti tutkia uudenlaista tapaa lähestyä ongelmaa [38]. Tavoitteena oli myös parantaa Borgin ohjelmistolaatua sen koostuessa monesta eri ad hoc -tapaan liitetystä palvelusta [11].

Tätä varten luotiin uusi konttienhallintajärjestelmä, Omega, jonka keskiössä oli halu kehittää täysin uusi aikatauluttaja tehostamaan palvelimien käyttöastetta. Toisin kuin Borg, jossa yksi monoliittinen aikatauluttaja huolehti kaikista muutoksista, Omegan aikatauluttaja perustui jaettuun tilaan, jolloin monella eri aikatauluttajalla oli mahdollisuus päästä tilaan samanaikaisesti käsiksi. Tämän avulla kaikkea logiikkaa ei tarvinnut lisätä yhteen vuorottajaan, vaan algoritmit voitiin jakaa eri vuorottajiin jotka puolestaan kilpailivat resurssien vuorottamisesta.

Mahdolliset yhteentörmäykset vuorottajien välillä tilaa päivittäessä hoidettiin optimistisella samanaikaisuudenhallinnalla (optimistic concurrency), jolloin kaikki vuorottajat pääsevät käsiksi kaikkiin resursseihin ja konfliktien sattuessa ne havaitaan jälkeenpäin ja muutokset perutaan. Pessimistinen hallinta lukitsisi resurssit jokaisella vuorottajalle yksi kerrallaan joka estää päällekkäiset muutokset, mutta hidastaa vuorottajien toimintaa.

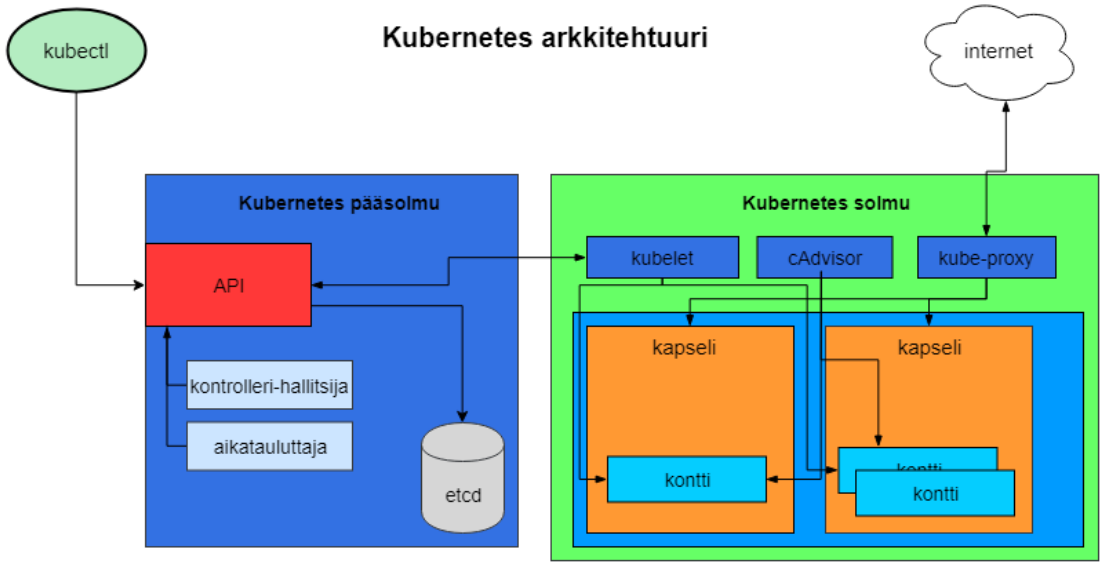
Tämä uusi aikataulutustapa on sittemmin tehty myös Borgiin ja sen pohjalta on myös rakennettu Googlen seuraava konttienhallintajärjestelmä Kubernetes [38].

3.4 Kubernetes

Toisin kuin aikaisemmat Googlen järjestelmät luotiin Kubernetes alun pitäen avoimen lähdekoodin projektiksi. Sen tarkoitus oli luoda kehittäjä- ja käyttäjäystävällisempi versio Borgista, joka korjaisi myös joitain Borgissa havaittuja puutteita. Se esimerkiksi pakottaa hyvin yhtenäisen rajapinnan toisin kuin Borg, jonka rajapinta oli hitaasti kasvanut vuosien aikana hyvin monimutkaiseksi [11].

Kuberneteksen kehitys aloitettiin vuonna 2014 ja annettiin Linux Foundationin hallinnoivan Cloud Native Computing Foundation:n alaisuuteen vuonna 2015 [27]. Kuberneteksessä ja Borgissa on hyvin paljon yhteistä ja sitä voidaan tavallaan katsoa kevyemmäksi versioksi Borgista [45].

Kuberneteksen arkkitehtuuri on kuvattu kuvassa 4. Yksi Kubernetes klusteri koostuu yhdestä pääsolmusta (master node) joka on useimmiten replikoitu monelle eri datakeskukselle häiriöiden varalta. Tämän on hyvin samankaltainen Borgiin verrattuna ja pääsolmulle myös voidaan lähettää pyyntöjä joko sen HTTP- tai RPC-rajapinnan kautta tai suoraan kubectl-hallintasovelluksella.



Kuva 4: Yksinkertaistettu näkymä Kuberneteksen toiminnasta (Rock Mutchler 2017)

Pääsolmu hallitsee klusterin tilaa etcd[4] avain-arvo-varastossa ja varaa sekä hallitsee resursseja käyttäjän tekemän konfiguraation mukaisesti. Erona Borgiin ja Omegaan on konsensus-algoritmi, joka niissä on Paxos, Kuberneteksessä Raft.

Kuten Borgissa klusteri koostuu Kuberneteksessä solmuista (node) jotka vastaavat joko fyysisiä tai virtuaalisia palvelimia. Yksittäinen solmu sisältää ainakin Kubelet-hallintaprosessin sekä konttien konttienhallintaohjelman kuten Dockerin.

Solmun sisällä taas on joukko kapseleita (pod) jotka koostuvat yhdestä tai useammasta kontista. Yleinen käytäntö on pitää vain yhtä konttia per kapseli paitsi jos kyseessä on tiukasti toisiinsa sidottu palvelu kuten webpalvelin ja lokientallennussovellus.

Kapselit ovat Kuberneteksen atominen yksikkö ja ovat verrattavissa Borgin allokaatioihin. Ne ovat tarkoitettu olemaan hetkellisiä ja jatkuvasti muuttuvia joita Kubernetes hallinnoi käyttäjiensä puolesta kuten miten kapselit jaetaan eri solmuille. Saman kapselin sisällä kontit jakavat levyn ja verkkoavaruuden joten kontit voivat löytää toisensa `localhost`:in avulla. Erona Borgiin jokaisella kapselilla on oma uniikki IP-osoite joten kontit sen sisällä voivat käyttää portteja miten haluavat.

Kapseleita ei suositella luomaan suoraan vaan niitä varten luotujen kontrollerien avulla jotka kuvaavat kapselin ajamiseen tarvittavat tiedot. Esimerkiksi korkeamman tason Deployment-kontrolleri on normaali tapa kuvata yhtä kapselia ja määrittämään esimerkiksi kuinka monta kapselia pidetään jatkuvasti käynnissä ja miten niiden määrää muutetaan kuormituksen kasvaessa tai laskiessa. StatefulSet-kontrolleria suositellaan käytettävän kapseleihin jotka ylläpitävät tilaa kuten tietokannat ja joka takaa hallitun tilan siirron vanhoista versioista uusiin. Deploymentin lisäksi useimmat kapselit toteuttavat myös vähintäänkin ReplicaSet:in jonka avulla tietty

[4] `https://coreos.com/etcd/`

määrä kapseleita pidetään aina käynnissä niiden kaatuessa sekä skaalataan ylös tai alas kuorman muuttuessa.

Kapseleille voidaan myös antaa muita ominaisuuksia erilaisten abstraktioiden avulla. Service avaa kapselista portin ulos ulkopuoliseen internettiin ja sitä käyttävät kapselit valitaan selektorien (selector) avulla joilla on tietty nimike (label). Kuberneteksen konfiguraation kirjoittamiseen käytetään YAML-tiedostoja [1].

Optimoidun resurssienhallinnan lisäksi isoin etu Kuberneteksen kaltaisessa palvelimien hallintajärjestelmässä on riippumattomuus palvelininfrastruktuurista. Kubernetes-klusteri on mahdollista ajaa omilla palvelimilla yhtä lailla kuin pilvessä kuten AWS:ssä tai GCP:ssä ilman suurempaa konfiguraation uudelleenkirjoittamista [35].

Kubernetes voidaan nähdä esimerkkinä koreografiasta, jossa joukko itsenäisiä mikropalveluita ja pieniä kontrollisilmukoita luo yhteistyössä emergentin kokonaisuuden. Se on vastakohta keskittyyn orkestrointiin joka on helpompi luoda alussa, mutta joka myöhemmin muodostuu herkäksi ja kankeaksi erityisesti odottomattomien virheiden ja tilamuutosten takia [11].

3.5 Edut ja ongelmat

Laajamittaisella konttien hallinnoimisella on mahdollista ylläpitää valtavia määriä palvelimia joka muilla tavoin tuskin olisi mahdollista. Kuitenkin konttihallintajärjestelmissä on edelleen ongelmina muun muassa konfiguraatio ja riippuvuuksien hallinta.

Yhden suuren ympäristön konfiguraatio on vaikeaa ja vaatii nopeasti oman Turing-vahvan kielen. Ratkaisu on selkeästi erottaa data ja laskenta toisistaan jolloin staattinen informaatio on jossain formaatissa kuten JSON tai YAML ja laskenta tuotetaan jollain ohjelmointikielellä jolla on hyvin tunnettu semantiikka ja virheenkorjaustuki.

Uuden palvelun käynnistäminen usein vaatii monen muun eri palvelun käynnistämistä (monitorointi, tiedostovarasto, CI/CD) joiden keskinäisiä riippuvuuksia on useimmiten mahdoton pitää yllä automaattisesti. Näiden riippuvuuksien rekisteröinti ja hallitseminen tekee kehittämisestä vaikeaa ja usein johtaa siihen, että parhaita käytäntöjä ei noudateta. Tämä riippuvuuksien hallinta on yhä avoin ongelma konttienhallintajärjestelmien kanssa [11].

4 Suunnittelumalleista

Konttien standardiominaisuuksien takia sille on muodostunut erilaisia käyttötapoja joita voi luonnehtia suunnittelumalleiksi (design pattern). Yleisin tapa konteille on tietenkin luoda ne yksittäisinä kokonaisuuksina ilman erillisiä riippuvuuksia toisiin kontteihin kuten tietokantapalvelin tai monoliittinen applikaatiopalvelin.

4.1 Yhden solmun, usean kontin malleja

Sivuvaunu (sidecar) -mallissa on yksi pääkontti ja toinen pääkonttia tehostava kontti joka huolehtii tietystä tehtävästä pääkontin puolesta. Eräs esimerkki on webpalvelin ja sen lokeja levyltä hakeva ja tallentava lokitallentaja-palvelu. Toinen esimerkki on webpalvelin ja kontti, joka jatkuvasti synkronisoi webpalvelimen tarjoamaa sisältöä Git:in tai jonkin muun datalähteen kautta.

Hyötyinä kahden kontin käyttö yhden kontin sijaan on tässä tapauksessa resurssien tehokkaampi allokointi jolloin webpalvelin saa käyttöönsä isoimman osan resursseista ja lokitallentaja hyödyntää loppuja kun webpalvelin ei niitä tarvitse. Palvelua on myös helpompi käsitellä kahtena eri kokonaisuutena ja kehittää toisista erillään. Lokitallentajaa on tällöin myös mahdollista käyttää toisten konttien kanssa.

Eräs toinen malli on suurlähettiläs (ambassador) -malli jossa pääkontin kommunikointi johonkin toiseen palveluun on välitetty toisen kontin lävitse. Näin pyyntö voidaan esimerkiksi hajauttaa taustalla olevaan tietokantaklusteriin pääkontin siitä tietämättä helpottaen pääkontin sovelluksen kehittämistä.

Sovitin (adapter) -malli on samanlainen suurlähettiläs-malliin verrattuna mutta missä suurlähettiläs esittää yksinkertaistetun näkymän pääkontille, sovittimessa pääkontti esitetään ulkopuoliselle maailmalle yksinkertaistettuna näkymänä. Näin voidaan esimerkiksi muuntaa erilaiset monitorointi-rajapinnat samaksi yhtenäiseksi rajapinnaksi koko klusterin sisällä [12].

4.2 Monen solmun malleja

Siirryttyessä yhdellä palvelimella toimivista konttimalleista monen solmun malleihin, selventävät kontit niiden luontia yksinkertaisilla rajapinnoille, joita järjestelmän osat toteuttavat.

Yleinen ongelma hajautetuissa järjestelmissä on johtajan valinta monesta samanlaisesta kopiosta. Johtajan valinta (leader election) -mallissa voidaan erilaisten linkettävien kirjastojen käytön sijaan käyttää valitsija-konttia, jonka kanssa keskustelevat kopiot toteuttavat jonkin rajapinnan tietojen lähettämiseen valitsija-kontille. Valitsija-kontin voi luoda aiheeseen erikoistunut asiantuntija ja kehittäjät uudelleenkäyttää sitä välimättä siitä, millä ohjelmointikielellä se on toteutettu.

Työjono (work queue) on myös toinen yleinen malli hajautetuissa järjestelmissä ja siihen on monia eri kehysmalleja (framework) mutta jotka pakottavat kehittäjän tiettyyn ohjelmointikieleen tai alustaan. Konttien avulla kehittäjä voi luoda kontin, joka toteuttaa halutut kehysmallin vaatimat rajapinnat kuten `run()`- ja `mount()`-metodit. Tämän kontin tehtävänä on sitten vain ottaa vastaan syöte-tiedostoja tiedostojärjestelmästä ja muuntaa ne ulostulo-tiedostoksi. Kaiken muun työjonoon liittyvän voi valmis kehysmalli huolehtia eriyttäen vain prosessoivat osat kontteina erilleen.

Viimeisenä esiteltävänä mallina on hajoittaja-kerääjä (scatter/gatherer) -malli jossa

ulkopuoliset asiakasohjelmat (client) lähettävät pyyntöjä juurisolmulle (root node). Tämä juurisolmu taas lähettää pyyntöjä sille allokoiduille palvelimille, jotka suorittavat laskennan rinnakkaisesti. Jokainen pirstale (shard) palauttaa osittaisen datansa juurelle, joka kerää vastaukset yhdeksi vastaukseksi asiakkaalle. Tämä malli on yleinen hakukoneissa. Iso osa sen toteutusta koostuu geneerisistä osista jotka voidaan konteilla luoda tiettyjen rajapintojen toteuttaviksi osiksi, mikä yksinkertaistaa sen rakennetta [12].

5 Pilvipalvelujen tarjoajista

5.1 Johdanto

Konttipohjaisten ratkaisujen yleistyessä monet pilvipalvelujen tarjoajat keskittyvät yhä etenevissä määrin tarjoamaan niille räätälöityjä ratkaisuja. Googlen pilvipalvelu GCP (Google Cloud Platform) on yksi pisimpään näitä palveluita tarjonnut yhtiö (2015) [41] ja monet muut ovat seuranneet sitä kuten AWS (2017) [7], Azure (2017) [34] ja DigitalOcean (2018) [46].

5.2 AWS

AWS (Amazon Web Services) on markkinaosuudeltaan suurin pilvipalveluntarjoaja (Amazon 34%, Microsoft 11%, IBM 8% ja Google 5%)[5] joka selittyy AWS:n oltua monia muita aikaisemmin markkinnoilla että sen jatkuvalla uusien palveluiden innovoimisella [40]. AWS tarjoaa laajan valikoiman eri palveluita joista konttienhallintapalvelu on vain yksi osa.

ECS (EC2 Container Service) on alkuperäinen AWS:n konttipalvelu joka tuli markkinoille vuonna 2015 [2]. Se on verrattavissa ehkä enemmän Borgiin kuin Kubernetekseen ja se on täysin hallittu AWS:n puolesta ja integroitu muihin AWS:n palveluihin. Käyttäjän tarvitsee vain luoda tehtäviä, joiden sisällä kontit ajetaan ja joita ECS aikatauluttaa klusterilla ajettavaksi. ECS:n konfiguraation voi tehdä joko konsolista, komentoriviltä tai AWS:n oman CloudFormation-palvelun kautta JSON tai YAML-tiedoistoilla. Nämä tiedostot soveltuvat vain AWS:n palveluihin [4].

EKS (Elastic Kubernetes Service) on AWS:n uusi Kubernetes-palvelu joka vielä tätä tutkielmaa kirjoittaessa oli suljetussa testikäytössä eikä vapaasti kaikkien saatavilla. Se kuten Googlen palvelu on AWS:n omien palveluiden kanssa integroitu Kubernetes joka myös huolehtii automaattisesti pääsolmujen replikoinnista ja kestävyydestä. Verrattuna ECS:ään se on riippumaton AWS:stä ja sama Kubernetes klusteri tulisi olla mahdollista siirtää toiseen palveluntarjoajaan ilman suurempaa vaivaa [5].

AWS tarjoaa myös uudenlaista palvelitonta (serverless) kontti-palvelua nimeltä Fargate ECS:n tai EKS:n kanssa käytettäväksi. Fargaten avulla käyttäjä voi luoda

[5] `https://www.srgresearch.com/articles/leading-cloud-providers-continue-run-away-market`

kontteja joita ajetaan vain tarvittaessa. Näin ollen ne minimoivat hukkakäytön ja käyttäjä maksaa vain siitä ajasta, jolloin ne ovat käynnissä [6].

ECR (Elastic Container Registry) on rekisteri Dockerkuvien tallentamiseen joka on syvästi integroitu ECS:n kanssa. Sen avulla käyttäjä voi pitää omaa täysin-hoidettua rekisteriään joka skaalautuu automaattisesti kuorman mukaan [3].

5.3 GCP

Google Cloud Platform (GCP) on Googlen pilvipalvelu jonka erikoisuutena ovat datan käsittelyyn erikoistuneet palvelut kuten TPU-palvelimet, BigTable ja MapReduce [22].

GCP myös tarjoaa kehittynyttä Kubernetes-palvelua johtuen suuresti siitä, että kyseessä on isolta osin Googlen luoma järjestelmä. Muut pilvipalveluntarjoajat ovat jo kuitenkin osin kuroneet kiinni Googlen etumatkaa ja tarjoavat melkein yhtä kattavia palveluita [30].

Kubernetes Engine (KDE) on Googlen tarjoama hoidettu Kubernetes-palvelu joka integroituu muiden GCP:n palveluiden kanssa ja joka hyötyy Googlen omasta konttijärjestelmien hallintaan liittyvästä osaamista [20].

5.4 Hintavertailu

Hintojen vertailu eri palveluntarjoajien välillä on vaikeaa erilaisten palveluiden ja ratkaisujen takia. Useimmat palvelut käyttävät Kubernetes-solmuinaan normaaleja virtuaalikoneita joita on mahdollista saada edulliseen hintaan pitkäkestoisilla sopimuksilla.

Cloud Spectator -nimisen pilvipalveluihin keskittyneen konsultointiyrityksen vuoden 2017 raportin mukaan sekä AWS että GCP ovat pelkillä virtuaalikoneiden tuntikustannuksilla mitattuna suhteellisen kalliita pienempiin tarjoajiiin verrattuna. Kuitenkin pitkäkestoisilla sopimuksilla niiden molempien hinnat ovat halvimmasta päästä varsinkin suurikokoisilla instansseille [15].

Virtuaalikoneiden lisäksi tavallinen Kubernetes-klusteri käyttää myös virtuaalilevyä joista esimerkkinä ovat vaikkapa GCP:n Compute Engine Persistent Disks ja AWS:n Elastic Block Storage (EBS) voluumit sekä Elastic File System (EFS). Raportin mukaan AWS on yksi halvimpia virtuaalilevyjen tarjoajia ja esimerkiksi GCP:tä hivenen halvempi [15].

Kuitenkin tavallinen suuri palvelinarkkitehtuuri käyttää monia eri palveluita joita hintavertailu ei ota huomioon.

Tutkimus- ja neuvonta-yhtiö Gartnerin käyttäjien antamissa arvioissa AWS ja GCP ovat molemmat saaneet saman arvosanan 4.4 [18].

6 Mielipiteitä konteista

Kontit ovat saaneet kritiikkiä ylimääräisen kompleksisuuden lisäämisestä sovelluksiin, jotka eivät välttämättä sitä tarvitse. Kontteja saatetaan käyttää mustan laatikon tavoin joiden käyttäytymistä ei kehittäjä täysin ymmärrä.[6]

Moderni palvelimienhallinta on kuitenkin hiljalleen keskittymässä virtualisointipohjaisiin ratkaisuihin eikä kehitykselle ole syytä nähdä loppua. Monet ongelmat joita suoraan rautatasolla toimivissa palvelimissa esiintyy katoavat siirryttäessä virtualisoituun ympäristöön.

Ovatko tulevaisuuden palvelimet sitten virtuaalikone- vai kontti-pohjaisia? Kontit kapseloivat ympäristönsä erittäin helposti käytettävään ja kehitettävään muotoon, joka helpottaa palvelimien kehittämistä ja jakamista. Tämä yhdistettynä tehokkuuden ja eristettävyyden parannuksiin luovat hyvin houkuttelevan tavan kehittää moderneja palvelinratkaisuja. Näkisin että tulevaisuuden palvelin tulee olemaan kuvattu konttikuvana.

Virtuaalikoneet eivät kuitenkaan kokonaan katoa vaikkakin ne eivät tavallisten kehittäjien työkuvaan liiemmin kuulu. Eristetyn ympäristön luominen palvelimille voidaan yhä toteuttaa vain virtuaalikoneella konttien turvallisuusongelmien takia joten pilvipalvelujentarjojat tuskin luopuvat niistä pääasiallisena toimintaympäristönä. Näkisin että ne toimivat parhaiten yhdessä täydentäen toisiaan, eivätkä ole toisiaan poissulkevia vaihtoehtoja.

Kehittäjien siis olisi parempi oppia tuntemaan kontit paremmin ja toivoisinkin, että tulevaisuudessa tullaan tarjoamaan esimerkiksi Helsingin Yliopistossa enemmän opetusta konteista.

7 Yhteenveto

Käyttöjärjestelmätason virtualisaatio on vanha keksintö joka on viimein saanut muotonsa kontiksi kutsuttuna toteutuksena. Sen avulla voidaan palvelimia kuvata yksinkertaisella ja standardilla tavalla helpottaen hajautettujen järjestelmien kehittämistä. Kontit tarjoavat virtuaalikoneisiin nähden kevyemmän virtuaalisaation vaikkakin ne kärsivät turvallisuusongelmista. Käyttämällä konttienhallintajärjestelmää voidaan kontti-pohjaisten palvelinten käyttöastetta parantaa tehokkaalla aikataulutuksella ja kuormaan sopivilla algoritmeilla. Avoimen lähdekoodin konttienhallintajärjestelmä kuten Kubernetes mahdollistaa myös palvelimien siirron pilvipalveluntarjoasta toiseen tai omille palvelimille vähäisellä vaivalla.

[6] `https://news.ycombinator.com/item?id=16445467`

Lähteet

1 T. K. Authors. Kubernetes documentation, 2018. https://kubernetes.io/docs, accessed at 12.3.2018.

2 AWS. Amazon EC2 Container Service is Now Generally Available. https://aws.amazon.com/about-aws/whats-new/2015/04/amazon-ec2-container-service-is-now-generally-available, accessed at 20.5.2018.

3 AWS. Amazon Elastic Container Registry. https://aws.amazon.com/ecr, accessed at 20.5.2018.

4 AWS. AWS ECS Overview. https://aws.amazon.com/ecs, accessed at 4.5.2018.

5 AWS. AWS EKS Overview. https://aws.amazon.com/eks, accessed at 4.5.2018.

6 AWS. AWS Fargate Overview. https://aws.amazon.com/fargate, accessed at 4.5.2018.

7 AWS. Introducing amazon elastic container service for kubernetes (preview). https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-amazon-elastic-container-service-for-kubernetes, accessed at 20.5.2018.

8 D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014.

9 J. Brodkin. What is a hypervisor? https://www.networkworld.com/article/3243262/virtualization/what-is-a-hypervisor.html, accessed at 18.5.2018.

10 J. Brodkin. With long history of virtualization behind it, ibm looks to the future, 2009. https://www.networkworld.com/article/2254433/virtualization/with-long-history-of-virtualization-behind-it--ibm-looks-to-the-future.html, accessed at 9.4.2018.

11 B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

12 B. Burns and D. Oppenheimer. Design patterns for container-based distributed systems. In *The 8th Usenix Workshop on Hot Topics in Cloud Computing (HotCloud '16)*, 2016.

13 E. Casalicchio and V. Perciballi. Measuring docker performance: What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 11–16, New York, NY, USA, 2017. ACM.

14 J. Che, C. Shi, Y. Yu, and W. Lin. A synthetical performance evaluation of openvz, xen and kvm. In *2010 IEEE Asia-Pacific Services Computing Conference*, pages 587–594, Dec 2010.

15 Cloud Spectator. Cloud iaas september 2017 edition price comparison. https://connect.cloudspectator.com/iaas-cloud-services-price-comparison-report, accessed at 20.5.2018.

16 CNCF. Cloud native computing foundation launches certified kubernetes program with 32 conformant distributions and platforms, 2017. https://www.cncf.io/announcement/2017/11/13/cloud-native-computing-foundation-launches-certified-kubernetes-program-32-co accessed at 12.3.2018.

17 G. Dunlap. The paravirtualization spectrum, part 1: The ends of the spectrum, 2012-10-23. https://blog.xenproject.org/2012/10/23/the-paravirtualization-spectrum-part-1-the-ends-of-the-spectrum, accessed at 16.5.2018.

18 Gartner. Compare amazon web services (aws) vs google in cloud infrastructure as a service. https://www.gartner.com/reviews/market/public-cloud-iaas/compare/amazon-web-services-vs-google, accessed at 20.5.2018.

19 J. Gomes, I. C. Plasencia, E. Bagnaschi, M. David, L. Alves, J. Martins, J. M. Pina, Á. L. García, and P. O. Fernández. Enabling rootless linux containers in multi-user environments: the udocker tool. *CoRR*, abs/1711.01758, 2017.

20 Google. Kubernetes engine - reliable, efficient, and secured way to run kubernetes clusters. https://cloud.google.com/kubernetes-engine, accessed at 20.5.2018.

21 Google. Kubernetes engine concepts. https://cloud.google.com/kubernetes-engine/docs/concepts, accessed at 20.5.2018.

22 Google. What makes google cloud platform different? https://cloud.google.com/free/docs/what-makes-google-cloud-platform-different, accessed at 20.5.2018.

23 X. He and J. Alves-Foss. A lightweight virtual machine monitor for security analysis on intel64 architecture. *J. Comput. Sci. Coll.*, 27(1):155–162, Oct. 2011.

24 S. Hykes. Introducing runc: A lightweight universal container runtime. https://blog.docker.com/2015/06/runc/, accessed at 20.5.2018.

25 S. Hykes. runc: The little container engine that could. https://opensource.com/life/16/8/runc-little-container-engine-could, accessed at 20.5.2018.

26 A. M. Joy. Performance comparison between linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346, March 2015.

27 F. Lardinois. As kubernetes hits 1.0, google donates technology to newly formed cloud native computing foundation.

28 P. Lessard. Linux process containment - a practical look at chroot and user mode. https://www.sans.org/reading-room/whitepapers/linux/linux-process-containment-practical-chroot-user-mode-1073, accessed at 20.5.2018.

29 Y. Li, R. West, and E. Missimer. A virtualized separation kernel for mixed criticality systems. *SIGPLAN Not.*, 49(7):201–212, Mar. 2014.

30 K. Marko. AWS' managed Kubernetes service faces tough competition in GKE. https://searchitoperations.techtarget.com/tip/AWS-managed-Kubernetes-service-faces-tough-competition-in-GKE, accessed at 26.5.2018.

31 S. McCarty. A practical introduction to container terminology, 2018. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction, accessed at 12.3.2018.

32 D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

33 R. Meshenberg. Microservices at netflix scale: Principles, tradeoffs & lessons learned. https://www.youtube.com/watch?v=57UK46qfBLY, accessed at 20.5.2018.

34 G. Monroy. Introducing aks (managed kubernetes) and azure container registry improvements. https://azure.microsoft.com/en-us/blog/introducing-azure-container-service-aks-managed-kubernetes-and-azure-containe accessed at 20.5.2018.

35 T. P. Morgan. Google wants kubernetes to rule the world. https://www.nextplatform.com/2016/11/08/google-wants-kubernetes-rule-world, accessed at 20.5.2018.

36 M. Nestor. Infographic: Lxd machine containers from ubuntu linux. http://linux.softpedia.com/blog/infographic-lxd-machine-containers-from-ubuntu-linux-492602.shtml, accessed at 20.5.2018.

37 Oracle. 3.13. configuring and using chroot jails, 2013. https://docs.oracle.com/html/E36387_02/ol_cj_sec.html, accessed at 25.5.2018.

38 M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

39 P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 1:1–1:13, New York, NY, USA, 2016. ACM.

40 Smartgov. Why amazon is leading the cloud. `http://smartgovcommunity.com/2017/02/why-amazon-is-leading-the-government-cloud`, accessed at 20.5.2018.

41 B. Stevens. Google cloud platform live: Introducing container engine, cloud networking and much more. `https://cloudplatform.googleblog.com/2014/11/google-cloud-platform-live-introducing-container-engine-cloud-networking-and-`
`html`, accessed at 20.5.2018.

42 G. Torres. Everything you need to know about the intel virtualization technology. `https://www.hardwaresecrets.com/everything-you-need-to-know-about-the-intel-virtualization-technology/2/`, accessed at 20.5.2018.

43 A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

44 C. White. Linux isolation basics. `https://www.engineyard.com/blog/linux-containers-isolation`, accessed at 20.5.2018.

45 J. Wilkes. Cluster management at google with borg. `https://www.youtube.com/watch?v=0W49z8hVn0k`, accessed at 20.5.2018.

46 J. Wilson. Simplify container orchestration: Announcing early access to digitalocean kubernetes. `https://blog.digitalocean.com/introducing-digitalocean-kubernetes/`, accessed at 20.5.2018.