jamk.fi

# Docker-ympäristön klusterointi

## Inmics Oy

Ville Anttila

Opinnäytetyö
Toukokuu 2016
Tekniikan ja liikenteen ala
Insinööri (AMK), tietotekniikan tutkinto-ohjelma

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

**Kuvailulehti**

| Tekijä(t) | Julkaisun laji | Päivämäärä |
|---|---|---|
| **Anttila Ville** | Opinnäytetyö, AMK | 22.05.2016 |
| | Sivumäärä<br>81 | Julkaisun kieli<br>Suomi |
| | | Verkkojulkaisulupa myönnetty: x |

Työn nimi
**Docker-ympäristön klusterointi**

Tutkinto-ohjelma
Tietotekniikan tutkinto-ohjelma

Työn ohjaaja(t)
Jarmo Viinikanoja
Mika Rantonen

Toimeksiantaja(t)
Inmics Oy
Perttu Kemiläinen


Tiivistelmä

Opinnäytetyön tavoitteena oli tutkia Docker-virtualisointitekniikkaa ja miten sen avulla voitiin isännöidä verkkosovellusta monessa eri palvelimessa. Docker on käyttöjärjestelmätasoinen virtualisointitekniikka, joka hyödyntää sovelluskontteja palveluiden käyttöönotossa ja ylläpidossa. Kontti sisältää kaikki tarvittavat komponentit siinä ajettavaa sovellusta varten. Kontti on eristetty sitä isännöivästä käyttöjärjestelmästä, eivätkä ne pysty suoraan kommunikoimaan keskenään ilman käyttäjää.

Omalle tietokoneelle luotiin testiympäristö, jossa käytettiin ilmaisia avoimeen lähdekoodiin perustuvia ohjelmia. Ympäristön avulla tutkittiin, miten palvelimia voitiin hallita yhtenä joukkona eli klusterina. Tämä tarkoittaa sitä, että jokaista palvelinta ja konttia voitiin hallita yhdeltä koneelta käsin. Klusteri muodostettiin Docker Swarm -työkalun avulla.

CoreOS-käyttöjärjestelmää hyödynnettiin Docker-konttien alustana. CoreOS on suunniteltu konesali- ja palvelinkäyttöön. Luodussa Swarm-klusterissa ajettiin Contriboard-verkkosovellusta. Verkkosovelluksen avulla tarkkailtiin, että ympäristö toimi oletetulla tavalla. Sen avulla käyttäjät pystyvät luomaan paperilappuja vastaavia tikettejä ja jakamaan niitä muiden kanssa. Contriboardin avulla saatiin tarkkailtua, kuinka paljon klusteri kesti käyttäjien luomaa verkkokuormaa ennen virhetilojen syntymisiä. Ympäristö kesti hyvin 250-500 käyttäjän luoman kuorman.

Toimeksiantaja halusi tietää, onko Docker valmis tuotantokäyttöön ja milloin sen lopullinen läpimurto laajemmille markkinoille voisi tapahtua. Opinnäytetyön aikana selvisi Dockerin olevan hyvä työkalu palveluiden ja sovelluksien hallintaan. Docker on jo valmis tuotantokäyttöön, ja monet suuryritykset käyttävät sitä jo omissa järjestelmissä. Docker Swarm sen sijaan on vielä keskeneräinen, koska se on ollut vasta vuoden julkaistuna. Tulosten ja teorianosuuden perusteella arviotiin, että Dockerin lopulliseen läpimurtoon kestää vielä kahdesta kolmeen vuotta.



Avainsanat (asiasanat)
Docker, Docker Swarm, klusteri, OS-tasoinen virtualisointi, virtualisointi, CoreOS, kuormantasaus, saatavuus


Muut tiedot

jamk.fi

**Description**

| Author(s)<br>**Anttila Ville** | Type of publication<br>Bachelor's thesis | Date<br>22.05.2016 |
|---|---|---|
| | | Language of publication:<br>Finnish |
| | Number of pages<br>81 | Permission for web publication: x |

Title of publication
**Clustering of Docker environment**

Degree programme
Information Technology

Supervisor(s)
Jarmo Viinikanoja
Mika Rantonen

Assigned by
Inmics Oy
Perttu Kemiläinen


Abstract

The objective of this bachelor's thesis was to research Docker virtualization technology, and how it is possible to host a service on multiple servers. Docker is an operation system level virtualization technology that uses a software Container to deploy and maintain the services. The Container contains all needed components that the application needs to run. It is isolated from the host machine and they cannot directly communicate between themselves without the user.

A test environment was created on a personal computer and only free open source programs were used on it. The environment was used to research how servers can be controlled as a cluster, which means that every server and container can be controlled from one machine. The cluster was created using Docker Swarm.

The operation system CoreOS was used to host the Docker containers. CoreOS is designed for data centers and server usage in mind. Contriboard web application was installed to the Swarm cluster to test that the environment worked as planned. With this application the user can create tickets like an ordinary paper note and share them to the other users. With Contriboard it was possible to get in-formation how much network load the environment can handle before failed states started to occur. The environment handled 250-500 users easily.

The assigner of the bachelor's thesis wanted to know if Docker is ready for production use and when will it make its breakthrough on major markets. While working on this bachelor's thesis it become clear that Docker is a good tool to control services and applications. Docker is ready for production use, and many big corporations already use Docker in their systems. Docker Swarm, on the other hand, is still in development because it has been available only for one year. Based on the results it was estimated that it will take two to three years before Docker will make its final breakthrough.



Keywords/tags (subjects)
Docker, Docker Swarm, cluster, Operating-system-level virtualization, virtualization, CoreOS, load balancing, high availability


Miscellaneous

# Sisältö

# Kuviot

# Taulukot

# Lyhenteet

| | |
|---|---|
| ACL | Access control list |
| AMD | Advanced Micro Devices |
| API | Application programming interface |
| CA | Certificate Authority |
| CLI | command-line interface |
| CPU | Central processing unit |
| DB | Database |
| DHCP | Dynamic Host Configuration Protocol |
| DNS | Domain Name System |
| ETCD | etc distributed |
| Eth | Ethernet |
| FTP | File Transfer Protocol) |
| HA | High Availability |
| HTTP | Hypertext Transfer Protocol |
| IP | Internet Protocol |
| LAN | Local Area Network |
| NAT | Network address translation |
| OS | Operating System |
| PEM | Privacy Enhanced Mail |
| PKI | Public Key Infrastructure |
| RAM | Random Access Memory |
| SMTP | Simple Mail Transfer Protocol |
| SSH | Secure Shell |
| TCP | Transmission Control Protocol |
| TLS | Transport Layer Security |
| UnionFS | Union File System |
| URL | Uniform Resource Identifier |
| VM | Virtual Machine |

# 1 Opinnäytetyön lähtökohdat

## 1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana oli Inmics Oy. Muut yritykset voivat ulkoistaa omia IT-ympäristöjään Inmicsille, yksittäisistä laitteista aina laajoihin järjestelmiin asti. Yritysasiakkaita Inmicsillä ovat esimerkiksi Panda, Tampereen Särkänniemi ja Vapo. (Tekniikka on menestyksen tukipilari n.d.)

Yrityksellä on tällä hetkellä töissä noin 120 henkilöä ja liikevaihtoa on yli 27 miljoonaa euroa (Liikevaihto ja henkilöstö n.d). Inmics Oy on perheyritys, jonka Jukka Autere perusti vuonna 1989. Hänen poikansa Tomi sekä Timo lähtivät mukaan yrityksen toimintaa 2000-luvun alkupuolella. (Perhe takaa kestävän kasvun n.d.)

## 1.2 Toimeksianto ja tavoitteet

Opinnäytetyön tavoitteena oli tutkia Docker-virtualisointitekniikkaa ja miten sen avulla voitiin isännöidä verkkosovellusta monessa eri palvelimessa. Toimeksiantaja halusi myös tietää, onko Docker valmis tuotantokäyttöön ja milloin sen lopullinen läpimurto laajemmille markkinoille voisi tapahtua.

Docker on käyttöjärjestelmätasoinen virtualisointitekniikka, joka hyödyntää sovelluskontteja palveluiden käyttöönotossa ja ylläpidossa. Kontti sisältää kaikki tarvittavat komponentit siinä ajettavaa sovellusta varten. Kontti on eristetty sitä isännöivästä käyttöjärjestelmästä, eivätkä ne pysty suoraan kommunikoimaan keskenään ilman käyttäjää.

Omalle tietokoneelle luotiin testiympäristö, jossa käytettiin ilmaisia avoimeen lähdekoodiin perustuvia ohjelmia. Ympäristön avulla tutkittiin, miten palvelimia voitiin hallita yhtenä joukkona eli klusterina. Tämä tarkoittaa sitä, että jokaista palvelinta ja konttia voitiin hallita yhdeltä koneelta käsin. Klusteri muodostettiin Docker Swarm -työkalun avulla.

CoreOS-käyttöjärjestelmää hyödynnettiin Docker-konttien alustana. Luodussa Swarm-klusterissa ajettiin Contriboard-verkkosovellusta. Verkkosovelluksen avulla

tarkkailtiin, että ympäristö toimi oletetulla tavalla. Sen avulla käyttäjät pystyvät luomaan paperilappuja vastaavia tikettejä ja jakamaan niitä muiden kanssa. Contriboardin avulla saatiin tarkkailtua, kuinka paljon klusteri kesti käyttäjien luomaa verkkokuormaa ennen virhetilojen syntymisiä.

# 2 Protokollat ja muut käsitteet

## 2.1 Linux-käyttöjärjestelmät

Linux on klooni Unix-käyttöjärjestelmästä, joka on ollut suosittu yritysympäristössä jo vuosikymmeniä. Linux koostuu käyttöjärjestelmän ytimestä eli kernelistä, mikä on Linuxin pääasiallinen ohjausohjelmisto. Monet kirjastot ja palvelut ovat riippuvaisia kernelin ominaisuuksista, joiden avulla käyttäjät voivat vuorovaikuttaa niihin. Linux-käyttöjärjestelmiä on saatavilla monelta eri jakajalta (distributor). Niiden ohjelmistot ja kernelin suunnittelut eroavat toisistaan. Linuxin jakelijoita ovat esimerkiksi Ubuntu, Fedora, CoreOS tai CentOS. Tällä hetkellä eri jakelijoita on satoja, ellei jo tuhansia. Linux on avoimeen lähdekoodiin perustuva (open source) käyttöjärjestelmä. Tämä tarkoittaa sitä, että kaikki Linuxin muodostavat tiedostot ovat vapaasti saatavilla. Tiedostoja voi muokata ja jakaa toisille käyttäjille vapaasti ilman rajoituksia. Jotkut jakelijat voivat vaatia maksua käyttöjärjestelmän käytöstä, mutta monia niistä voi ladata internetistä ilmaiseksi. (Smith 2012, 37.)

Koska Linux on klooni vanhasta Unix-käyttöjärjestelmästä, peri Linux monia eri ohjelmistoja Unixista, kuten esimerkiksi tärkeitä internetpalvelinohjelmia, tietokantoja, ohjelmointikieliä ja muuta sellaista. Linux skaalautuu hyvin ja sitä voidaan käyttää melkein kaikissa laitteissa. Linux toimii hyvin, myös vanhemmissa tietokoneissa. Linux on hyvä vaihtoehto koneille, jotka eivät voi enää hyödyntää uusimpia Microsoftin Windows-käyttöjärjestelmiä. Yleisesti yrityksien työasemissa käytetään Windows-käyttöjärjestelmää ja Linuxia hyödynnetään organisaatioiden verkkosovelluksien pyörittämiseen, verkkoliikenteen ohjaamiseen ja muihin kriittisiin taustaprosesseihin. (Smith 2012, 38.)

Monet nykyaikaset Linux-järjestelmät pystyvät siihen, mitä Unixin aikaan ei voitu toteuttaa. Näitä Linux-ominaisuuksia ovat esimerkiksi klusterointi (Clustering), virtualisointi (Virtualization), reaaliaikainen tietojenkäsittely (Real-time computing) ja verkkotallennusmahdollisuudet. Klusterin avulla voidaan määrittää useita järjestelmiä näkymään ulkomaailmaan yhtenä kokonaisuutena. Palveluita voidaan laittaa siirtymään edestakaisin klusterin solmujen välillä, jotta vikatilanteen sattuessa palvelut ovat saatavilla ilman pitkiä keskeytyksiä. Virtualisoinnin avulla voidaan jakaa tietojenkäsittelyn resursseja tehokkaammin palvelimissa ja työasemissa. Tämän tekniikan avulla voidaan ajaa useita käyttöjärjestelmiä yhdessä fyysisessä laitteessa. Reaaliaikaisen tiedonkäsittelyn avulla tärkeille prosesseille taataan nopeaa ja luotettavaa tarkkailua vikatilanteiden varalta. Linuxin avulla enää ei tarvitse tallentaa tietoa vain tietokoneen kovalevylle, vaan dataa voidaan levittää paikallisen- tai verkkotallennuksen avulla laajoiksi kokonaisuuksiksi. (Bresnahan & Negus 2012, 5.)

## 2.2 Virtualisointi

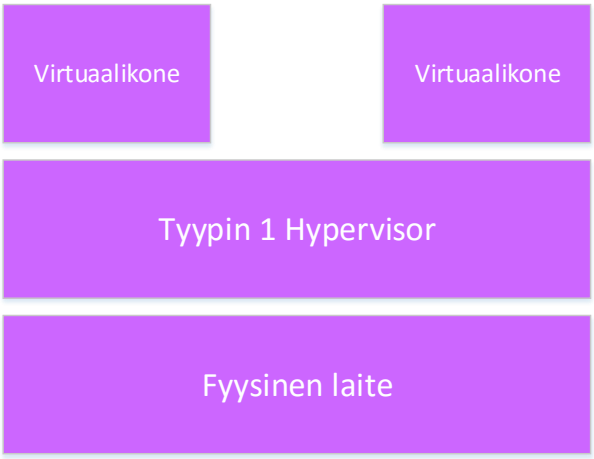
### 2.2.1 Yleistä

Virtualisoinnin avulla voidaan luoda useita virtuaalisia järjestelmiä fyysiseen laitteeseen. Virtualisointia voidaan hyödyntää tietokoneissa, käyttöjärjestelmissä, tallennuslaitteissa, ohjelmissa tai verkkoyhteyksissä. Eniten tätä tekniikkaa käytetään kuitenkin palvelimissa. Virtualisoinnin avulla voidaan ottaa paremmin laitteiden resurssit käyttöön kuin normaalissa käytössä, jossa ajetaan yhtä käyttöjärjestelmää ja sovellusta kerrallaan. Virtualisoinnin avulla ohjelmisto simuloi laitteistoa. Näin voidaan ajaa useita virtuaalisia järjestelmiä, käyttöjärjestelmiä tai ohjelmia yhdellä palvelimella. Virtuaalitietokone on eristetty ohjelmistosäiliö, jossa ajetaan käyttöjärjestelmää. Jokainen virtuaalikone on täysin itsenäinen. Näin voidaan ajaa useita virtuaalikoneita yhdessä fyysisessä isäntälaiteessa. Hypervisor on ohjelmistokerros, joka erottelee virtuaalikoneet isäntäkoneesta ja dynaamisesti määrittelee tarvittavat tietojenkäsittelyresurssit jokaiselle virtuaalikoneelle. (Virtualization n.d.)

### 2.2.2 Hypervisor

Ilman hypervisoria käyttöjärjestelmät keskustelevat suoraan laitteistolle tai kaikki virtuaalikoneet haluaisivat käyttää yhtä aikaa resursseja. Hypervisor hallitsee vuorovaikutusta jokaisen virtuaalikoneen ja niiden isäntälaitteen välillä. On olemassa kaksi hypervisor-luokkaa: Tyyppi 1 ja Tyyppi 2. (Portnoy 2012, 19-21.)

Tyypin 1 hypervisorin toimintamalli on esitetty kuviossa 1. Tyypin 1 hypervisoria ajetaan suoraan palvelimen laitteistolla ilman, että käyttöjärjestelmää on sen alla. Ilman välikättä tyypin 1 hypervisor voi suoraan keskustella laitteiston resurssien kanssa tehden tämän tehokkaammaksi tavaksi kuin tyypin 2 hypervisor. Tyypin 1 hypervisorissa kuluu vähemmän tietojenkäsittelykustannuksia, koska se ei tarvitse isäntäkäyttöjärjestelmään. Tämä tarkoittaa sitä, että voidaan ajaa enemmän virtuaalikoneita yhdellä isäntäkoneella. Tyypin 1 hypervisor on myös tietoturvallisempi kuin tyypin 2. Virtuaalikoneet voivat vahingoittaa itseään, mutta vikatilanteet tai tietoturvauhat eivät voi siirtyä virtuaalikoneen säiliön rajojen ulkopuolelle toisiin virtuaalikoneisiin tai isäntälaitteeseen. (Portnoy 2012, 21-22.)




Kuvio 1. Tyypin 1 hypervisor

Tyypin 2 hypervisorin toiminta on esitetty kuviossa 2. Tyypin 2 hypervisor ohjelmisto toimii tavallisessa käyttöjärjestelmässä. Yleisesti tyypin 2 hypervisor on helppo asentaa ja ottaa käyttöön, koska käyttöjärjestelmä hoitaa suuren osan konfiguraatiotyöstä. Tyypin 2 hypervisor ei ole niin tehokas kuin tyypin 1 hypervisor, koska siinä on

ylimääräinen kerros hypervisorin ja laitteiston välillä. Joka kerta kun virtuaalikoneet tekevät laitteiston kanssa jonkin toiminnon, se välitetään hypervisorille kuten tyypin 1 ympäristössä. Poikkeavasti tyypin 1 hypervisor antaa pyynnön käyttöjärjestelmälle, joka huolehtii käyttäjän komennoista. Käyttöjärjestelmä välittää tiedon hypervisorille ja siitä takaisin virtuaalikoneelle. Tämä lisää jokaiselle toiminnolle lisää kuormaa ja vie enemmän aikaa. Tyypin 2 hypervisor ei ole niin luotettava, koska siinä on enemmän vikapisteitä. Kaikki, mikä vaikuttaa isäntäkäyttöjärjestelmään, vaikuttaa myös hypervisoriin ja virtuaalikoneisiin. Esimerkiksi jos joudutaan käynnistämään isäntäkäyttöjärjestelmää, myös virtuaalikoneet joudutaan sammuttamaan. (Portnoy 2012, 23-22.)


Kuvio 2. Tyypin 2 Hypervisor

Hypervisor on muodostunut toimimaan laitteistojen käyttöjärjestelmänä. Ilman että se käsittelisi ohjelmisto- tai sovelluspyyntöjä, se palvelee kokonaisia palvelimia. Käskyn antaminen virtuaalikoneelle etenee seuraavalla tavalla: Virtuaalikoneella oleva ohjelmisto haluaa tehdä kovalevyltä lukutoiminnon. Pyyntö lähettää virtuaalikoneen käyttöjärjestelmälle. Virtuaalikone tekee lukutoiminnon näkemälleen levylle, ja hypervisor muokkaa tämän pyynnön oikean maailman fyysiseen vastaavaan tallennuspaikkaan. Kun vastaus tulee, hypervisor välittää tiedon takaisin virtuaalikoneen käyt-

töjärjestelmälle, joka vastaanottaa tiedostot, kuin se olisi tullut suoraan fyysisestä laiteesta. Hypervisor käsittelee myös kaikki käyttäjältä tulevat verkkoon liittyvät komennot ja prosessointityöt. Hypervisor tekee nämä suoritteet kaikille isäntäkoneessa oleville virtuaalikoneille. (Portnoy 2012, 25.)

Hypervisor ajoittaa resursseja prosesseille ja varmistaa, että kaikki resurssipyynnöt välitetään eteenpäin. On myös mahdollista priorisoida eri virtuaalikoneita, että tärkeimmät sovellukset saavat tarvittavan huomion ja ne eivät kärsisi liian vähistä resursseista. Virtuaalisessa infrastruktuurissa tämän tyyppinen hallinta ja resurssien jakaminen ovat kriittistä. Kaikki virtuaalikoneiden tarvitsemat resurssit on löydyttävä isäntälaitteesta. Jos jaetaan liikaa resursseja virtuaalikoneiden kesken, ne joutuvat kilpailemaan keskenään, ja tämä näkyy käyttäjälle toimintojen viiveenä. (Portnoy 2012, 25-26.)

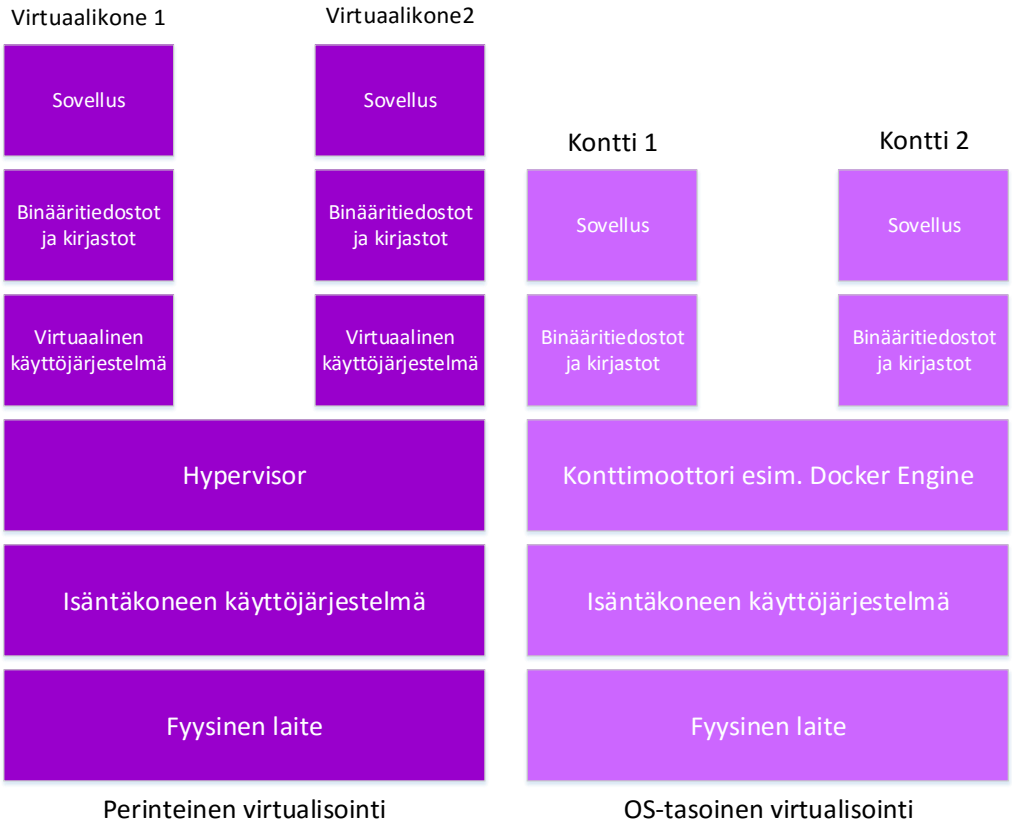
### 2.2.3 Virtuaalikone

Virtuaalikone eli VM (Virtual Machine) sisältää samoja ominaisuuksia kuin fyysinen palvelin. Virtuaalikone tukee käyttöjärjestelmiä, ja siihen on määritetty käytettäväksi laitteiston resursseja. Monia eri virtuaalikoneita voidaan ajaa samassa fyysisessä palvelimessa tai työasemassa. Nämä virtuaalikoneet voivat ajaa eri käyttöjärjestelmiä, jotka tukevat erilaisia sovelluksia. Virtuaalikoneet eivät ole mitään muuta, kun joukko tiedostoja, joiden avulla määritetään palvelin. (Portnoy 2012, 35.)

Päätiedostot, jotka muodostavat virtuaalikoneen, ovat konfiguraatio- ja virtuaalilevykuvatiedostot. Konfiguraatiotiedosto määrittelee laiteresurssit, joita virtuaalikoneet käyttävät. Se matkii virtuaalista laitteistoa, jonka päälle luodaan haluttu virtuaalikone. Konfiguraatiotiedoston avulla listataan laitteet, jotka virtuaalikoneella löytyy, kuten esimerkiksi prosessori, muisti, tallennus- ja verkkoasemat. Ulkoapäin virtuaalikone on identtinen käyttäjälle fyysiseen laitteeseen verrattuna. Kaikki laitteet ovat geneerisiä eivätkä kuvasta mitään oikeaa laitetta, mitä käyttäjän järjestelmässä esiintyy. (Portnoy 2012, 35-36.)

### 2.2.4 Käyttöjärjestelmätasoinen virtualisointi

OS-tasoisessa (operating system) virtualisoinnissa kernel sallii monien toisistaan eristyksessä olevien käyttäjätilainstanssien (userspace instance) käytön. Näitä instansseja kutsutaan yleisesti konteiksi. Ne näyttävät ja tuntuvat ulkopuolisille käyttäjille oikeilta palvelimilta. Tätä virtualisointitekniikkaa käytetään yleisesti isännöintiympäristöihin kuten julkisiin pilvipalveluihin, joissa on hyödyllistä määrittää rajallisia määriä laiteresursseja suurien käyttäjämäärien kesken. Käyttöjärjestelmätasoinen virtualisointi ei sisällä ylimääräisiä resurssikustannuksia, koska virtuaaliympäristön ohjelmat käyttävät käyttöjärjestelmän komentorivejä. Eli niitä ei tarvitse ajaa emulaattoreilla tai virtuaalikoneilla kuten normaalissa virtualisoinnissa. Jotkut käyttöjärjestelmätasoiset virtualisointitavat sisältävät ominaisuuden siirtää kontteja dynaamisesti klusterin solmujen välillä. Käyttöjärjestelmätasoinen virtualisointi ei tarvitse laitteiston tukea, että se toimii tehokkaasti. Toisaalta se ei pysty ajamaan käyttöjärjestelmää kuten normaalissa virtualisoinnissa. Kuviosta 3 voidaan nähdä, miten perinteinen virtualisointi eroaa OS-tasoisesta virtualisoinnista. (Gonzalez & Krishnan 2015.)




Kuvio 3. Perinteisen virtualisoinnin ja OS-tasoisen virtualisoinnin vertailu

## 2.3 Klusteri

### 2.3.1 Yleistä

Klusteri on kokoelma toisiinsa yhdistetyistä tietokoneista, eli monta erillistä laitetta toimii yhtenä laskentaresurssina. Perinteiset klusterikomponentit koostuvat joukosta yksittäisiä tietokoneita, käyttöjärjestelmiä tai sovelluksia. Klusteroinnin avulla saadaan monia eri hyötyjä, kuten hyvä hintasuhde suurissa järjestelmissä, pystytään päivittämään systeemejä helposti, voidaan hyödyntää avoimeen lähdekoodiin perustuvia alustoja ja voidaan olla riippumattomia eri laitevalmistajista. Lisäksi klustereiden avulla saadaan palveluiden saatavuuteen varmuutta ja voidaan skaalata helposti ympäristöä suuremmaksi tai pienemmäksi tarpeen mukaan. (Buyya, Eskicioglu, Graham, Pourreza, Sommers & Yeo n.d, 2.)

### 2.3.2 Kuormantasaus

Kuormantasauksella tarkoitetaan sitä, että sisään tuleva verkkoliikenne jaetaan tehokkaasti klusterin laitteiden kesken. Moderneissa verkkosivuissa voi olla miljoonia samanaikaisia asiakaspyyntöjä, joihin klusterin on reagoitava. Kuormantasaaja toimii palvelimien etupuolella ja reitittää asiakaspyyntöjä kaikkien saatavilla olevien laitteiden kesken, jotta pyyntöihin voidaan vastata mahdollisimman nopeasti. Samalla hyödynnetään kaikki resurssit, joita klusterissa löytyy, mutta ei rasiteta yksittäisiä laitteita liikaa. Jos yksittäinen solmu menee vikatilaan klusterissa, kuormantasaaja ohjaa liikenteen muihin toimiviin palvelimiin. Näin taataan hyvä palveluiden saatavuus ja luotettavuus, kun lähetetään liikettä vain toimintakuntoisiin palvelimiin. Kun uusi laite lisätään klusteriin, kuormantasaaja alkaa siirtää verkkoliikennettä sinne automaattisesti. Eli näin voidaan lisätä tai poistaa palvelimia joustavasti tarpeen mukaan. (What is load balancing? n.d.)

### 2.3.3 Saatavuus

Termillä High Availability (HA) tarkoitetaan, että kriittisille resursseille taataan paras mahdollinen saatavuus. Tämä toteutetaan asentamalla klusterin palvelimiin ohjelmisto, joka huolehtii HA:sta. Tämä ohjelmisto tarkkailee klusterin solmujen saatavuutta. Jos yksi palvelin menee alas tai se joudutaan pysäyttämään, HA-ohjelma

käynnistää palvelun jossain muussa klusterin osassa. Tämä toimenpide yritetään tehdä mahdollisimman nopeasti, jotta palvelut olisivat käyttäjien saatavilla. HA takaa resurssien maksimaalisen saatavuuden, mutta sitä ei voi toteuttaa ilman katkoksia. Jos joku klusterin osa menee vikatilaan, siirretään palvelut toimiviin palvelimiin niin nopeasti, kun mahdollista. Tässä on kuitenkin pieni viive, ennen kun kaikki on täysin toimintakunnossa. (Vugt 2014.)

## 2.4 Verkkoprotokollat

Tässä kappaleessa käsitellään lyhyesti verkkoprotokollat, joita tulee esille testiympäristön yhteydessä.

Kaikki Internet kuljetusprotokollat käyttävät IP-protokollaa (Internet Protocol) tiedonsiirtoon lähetys- ja vastaanottopään välillä. IP on yhteydetön, eli se ei pysty tarkistamaan tuleeko kuljetettavat tiedot perille rikkoutuneena, tuplana tai epäjärjestyksessä. IP tarvitsee muita protokollia ja järjestelmiä, jotta voidaan luoda luotettava kuljetuspalvelu. IP-protokolla sisältää tarpeet osoitukseen, palveluntyypin valinnalle, tiedon paloittelulle sekä sen kasaamiseen. (RFC1122, 1989, 9.) IPv4 –osoitteet ovat määritetty neljän oktetin eli 32 bitin pituiseksi (RFC791, 1981, 6).

TCP (Transmission Control Protocol) on luotettava host-to-host protokolla pakettikytkentäisessä kommunikaatioverkossa. TCP pystyy vastaanottamaan yksinkertaisen ja epäluotettavan datagrammin toiselta protokollalta ja muuttamaan se luotettavaksi ja viansietäväksi. (RFC793, 1981, 1)

Jotta voidaan tunnistaa eri käsiteltävät tietoliikennevirrat, TCP luokittelee ne eri porttitunnisteilla. Portit eivät ole yksilöllisiä, koska jokainen TCP valikoi ne erikseen. Jotta voidaan luoda yksilöllisiä osoitteita yhteyksille pitää TCP:n yhteyteen määritellä IP –osoite. TCP:n avulla voidaan määritellä vapaasti portit. On kuitenkin olemassa tunnettuja portteja, joihin on määritetty tietty palvelu. (RFC1122, 1989, 82.)

DNS (Domain Name System) protokollan avulla voidaan nimetä resursseja niin että ne ovat käytettävissä eri isäntälaitteissa, tietoverkoissa, protokollissa ja internetissä. Käyttäjä siis pystyy kysymään tietyn isäntälaitteen osoitteen avulla sen toimialueen nimen. Näin käyttäjän ei tarvitse muistaa ulkoa esimerkiksi IP-osoitetta, vaan pääsee verkkosivun sivulle toimialueennimen avulla. (RFC1035, 1987, 3.)

# 3 Docker

## 3.1 Yleistä

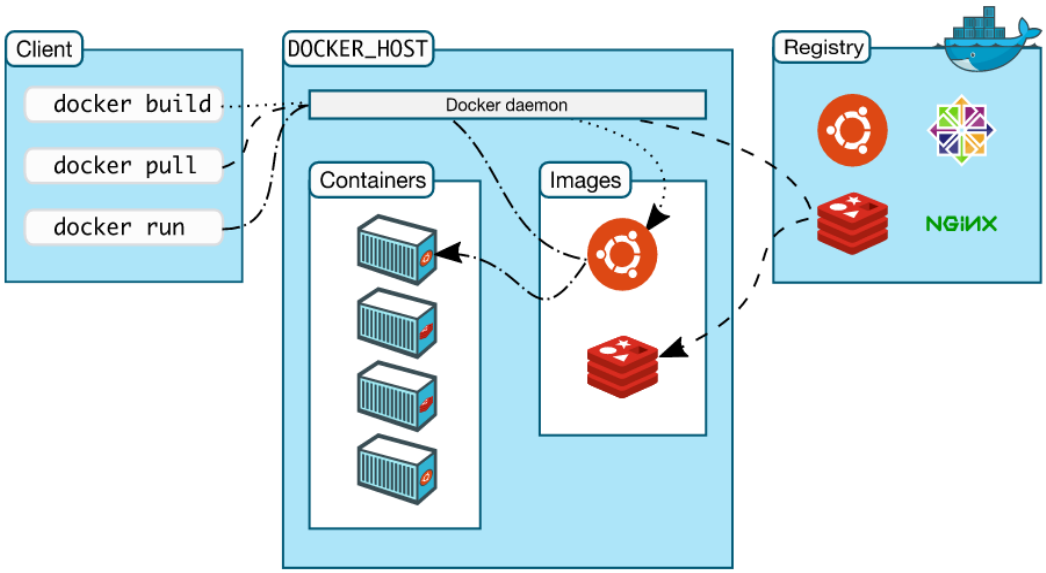
Solomon Hykes esitteli Dockerin ensimmäistä kertaa maailmalle esittelytilaisuudessa maalisuussa 2013. Tilaisuudessa neljäkymmentä henkilöä pääsi ensimmäistä kertaa kokeilemaan Dockeria. Muutaman viikon jälkeen ensiesittelystä oli yllättävän paljon kuhinaa julkisuudessa. Projekti meni nopeasti avoimeen lähdekoodiin perustuvaksi ja se julkaistiin yleisesti saataville GitHubissa. Seuraavien kuukausien aikana yhä enemmän ihmisiä alkoi kuulla Dockerista ja kuinka se tulisi mullistamaan ohjelmistojen rakentamisen, siirtämisen ja ajamisen tulevaisuudessa. Vuoden julkaisun jälkeen lähes jokainen tietotekniikan alalla tiesi Dockerista, ja ihmiset olivat todella innoissaan tästä uudesta järjestelmästä. (Kane & Matthias 2015, 1.)

Docker on avoin järjestelmä, jonka avulla voidaan kehittää, jakaa ja ajaa sovelluksia. Dockerilla pystytään luomaan sovelluksia, joita voidaan käyttää lähes kaikissa saatavilla olevissa alustoissa kuten tietokoneissa, pilvipalveluissa, konesalissa tai virtuaalikoneella. Se sisältää kaikki tarvittavat ominaisuudet ajettaville ohjelmille kuten koodin, ajonaikaiset kirjastot (runtime libraries) ja järjestelmätyökalut (system tools). Docker paketoi kaikki tarvittavat ohjelmistot, joita sovellus tarvitsee yhteen ohjelmistoyksikköön, jota kutsutaan Docker-levykuvaksi (Docker image). (Vohre 2016, 1.)

Sovellukset ajetaan Docker-konteissa, jotka ovat eristettyjä ympäristöjä ja ne sisältävät oman tiedostojärjestelmät ja ympäristömuuttujat. Docker-kontit ovat eristyksissä toisistaan ja sitä ympäröivästä käyttöjärjestelmästä. Tiedostoja voidaan kopioida isäntäkäyttöjärjestämästä konttiin, jos on tarvetta siihen. Sovellus voi vaatia, että muut ohjelmat, joita kehitetään, on linkitetty siihen, mikä antaa muiden konttien ympäristömuuttujat ja ohjelmat saataville sille. (Vohre 2016, 1.)

Docker käyttää client-server arkkitehtuuria, jonka toimintamalli nähdään kuviosta 4. Docker-asiakasohjelma (Docker client) keskustelee Docker-taustaprosessin (Daemon) kanssa, joka huolehtii konttien rakentamisesta, ajamisesta ja jakamisesta. Asiakasohjelma (Client) on Dockerin ensisijainen käyttöliittymä. Asiakasohjelmaa ja taustaprosessia voidaan ajaa samassa järjestelmässä. Toinen mahdollisuus on yhdistää Docker-asiakasohjelma etätaustaprosessiin. Docker-taustaprosessia ajetaan isäntäkoneessa,

ja käyttäjä ei suoraan pysty vuorovaikuttamaan siihen. Tämä toteutetaan Docker-asiakasohjelman kautta. Se hyväksyy komentoja käyttäjältä ja kommunikoi taustaprosessin kanssa edestakaisin. (Understand the architecture n.d.)




Kuvio 4. Dockerin client-server arkkitehtuuri (Understand the architecture n.d.)

Docker yhdistää sovelluksien ja vaadittujen käyttöjärjestelmä tiedostojärjestelmät yhteen standardoituun levykuvaformaattiin. Ennen piti tyypillisesti paketoida ohjelmistot ja niihin liittyvät kirjastot sekä palveluprosessit (Daemon). Ei pystytty kuitenkaan täysin takaamaan, että ajettavat ympäristöt olisivat täysin identtisiä toistensa kanssa. Dockerin avulla voidaan erotella sovellukset laitteistosta ilman, että uhrataan turhaan resursseja. Perinteisissä virtualisointitavoissa tyypillisesti erotellaan fyysisen laitteiston kerros ohjelmistosta ja tämä tekniikka vie turhaan resursseja. Hypervisor käsittelee vain VM-koneita ja niiden kerneliä. Tämä prosessi vie muutamia prosentteja järjestelmän resursseista jokaista VM-konetta kohden, mikä ei ole tietenkään enää isännöityjen ohjelmistojen käytössä. Kontti on käytännössä vain prosessi, joka puhuu suoraan käyttöjärjestelmän kernelille, ja tästä syystä se käyttää tehokkaasti laitteistoresursseja. (Kane & Matthias 2015, 4.)

Docker ei ole virtualisointialusta, pilvipalvelualusta tai konfiguraation hallintajärjestelmä. Kontti ei ole virtuaalikone perinteisessä mielessä. VM sisältää täydellisen käyttöjärjestelmän isäntäkäyttöjärjestelmän päällä. Kontit pyörivät samassa kernelissä isäntälaitteen kanssa, mistä syystä se käyttää vähemmän resursseja. Kontit ovat myös vähän samankaltaisia kuin pilvipalvelu. Kummatkin sallivat sovelluksien skaalauksen muuttuvan kysynnän mukaan. Docker ei ole kuitenkaan pilvipalvelualusta, koska se käsittelee sovelluksien käyttöönoton, ajamisen ja hallinnoinnin valmiiksi luodussa Docker-isäntäkoneessa. Sen avulla ei voi tehdä uusia isäntäinstansseja, joka on tyypillisesti käytetty pilvipalveluissa. Docker voi huomattavasti parantaa kykyä hallita ohjelmia ja niistä riippuvia osa-alueita. Se ei suoraan korvaa monia perinteisiä konfiguraatiohallintaohjelmia. Dockerfileä käytetään määrittelemään, miltä kontin tulisi näyttää sen luonnin yhteydessä. Se ei kuitenkaan hallinnoi sitä, kun se on jo käytössä. Tiedostoa ei voi myös käyttää hallinnoimaan Dockerin isäntäsysteemiä. (Kane & Matthias 2015, 5.)

## 3.2 Docker-levykuvat

Docker-levykuva (Docker image) on Dockerin kirjoitussuojattu rakennuskomponentti. Esimerkiksi levykuva voisi sisältää Ubuntu-käyttöjärjestelmän, joka sisältää Apache-palvelimen ja jonkin verkkosovelluksen. Docker tarjoaa yksinkertaisen tavan rakentaa uusia tai päivittää olemassa olevia levykuvia. On myös mahdollista ladata toisten käyttäjien tekemiä Docker-levykuvia. (Understand the architecture n.d.)

Jokainen levykuva koostuu useista kerroksista. Docker käyttää UnionFS-tiedostojärjestelmää (Union File System) yhdistääkseen kerrokset yhdeksi levykuvaksi. UnionFS sallii tiedostojen ja erillisten tiedostojärjestelmien hakemistojen (branch) laittamisen päällekkäin sekä yhtenäisen tiedostojärjestelmän muodostamisen. Yksi syy, miksi Docker on niin kevytrakenteinen, johtuu juuri näistä kerroksista. Jos tehdään muutoksia Docker-levykuvaan, uusi kerros muodostetaan. Kuten esimerkiksi, jos päivitetään sovellus uuteen versioon. Tästä johtuen ei korvata koko levykuvaa vaan muodostetaan uusi kerros tai päivitetään jo olemassa olevaa kerrosta. Eli käyttäjän ei tarvitse jakaa täysin uutta levykuvaa vaan sen päivitettyä versiota. Tehden Docker-levykuvasta nopeamman ja yksinkertaisemman kuin esimerkiksi verrattuna virtuaalikoneisiin. (Understand the architecture n.d.)

Jokainen Docker-levykuva aloitetaan jostain pohjalevykuvista kuten Ubuntu- tai Fedoran-levykuvista. On myös mahdollista käyttää omia levykuvia lähtökohtana uudelle Docker-levykuvalle. Esimerkiksi Apache-levykuvaa voidaan käyttää perustana kaikkiin verkkosovelluslevykuviin. Docker image muodostetaan näistä pohjalevykuvista käyttäen skriptiä nimeltään instructions eli ohjeistusta. Jokainen ohjeistus luo uuden kerroksen levykuvaan. Ohjeistuksessa voi olla toimintoja kuten komennon ajaminen, tiedoston lisääminen tai ympäristömuuttujan luominen. Nämä ohjeistukset ovat tallennettu tiedostoon nimeltään Dockerfile. Kun käyttäjä määrää uuden version luomisen levykuvasta, suoritetaan Dockerfile-ohjeistuksista komennot ja uusi versio muodostetaan. (Understand the architecture n.d.)

## 3.3 Docker-rekisterit

Rekisteri on Dockerin jakelukomponentti. Se (Docker registries) sisältää kaikki tarjolla olevat levykuvat. On olemassa yleisiä tai yksityisiä säiliöitä, johon voidaan lähettää tai sieltä voi ladata levykuvia. Yleiset Docker-rekisterit ovat saatavissa Docker Hub-palvelussa, joka sisältää suuren määrän jo olemassa olevia levykuvia käytettäväksi suoraan. Nämä levykuvat voivat olla itsetehtyjä tai toisten käyttäjien muodostamia. (Understand the architecture n.d.)

Kun luodaan uusi Docker-levykuva, käyttäjä voi siirtää sen julkiseen rekisteriin kuten Docker Hub-palveluun tai omaan palomuurin takaiseen rekisteriin. Docker-asiakasohjelmaa voidaan käyttää hakemaan julkaistuja levykuvia käytettäväksi omiin konteihin. Docker Hub tarjoaa julkisen ja yksityisen tallennusmahdollisuuden levykuville. Julkisia levykuvia pystyy jokainen hakemaan ja lataamaan. Yksityisiä rekisterejä voi nähdä vain käyttäjät kenelle on annettu oikeudet niiden käyttöön. (Understand the architecture n.d.)

## 3.4 Docker-kontit

Docker-kontit ovat Dockerin ajokomponentti, jota voidaan verrata hakemistoihin. Docker kontti sisältää kaiken tarvittavan, jotta voidaan ajaa jotain sovelluta tai palvelua. Jokainen kontti on luotu Docker-levykuvan avulla. Kontit ovat eristettyjä ja suojattuja sovellusalustoja. (Understand the architecture n.d.)

Kontti koostuu käyttöjärjestelmästä, käyttäjän lisäämistä tiedostoista ja meta-datasta. Levykuvat kertovat Dockerille mitä kontit tulevat sisältämään ja prosessin niiden ajamisesta, kun kontti käyttöönotetaan. Dockerin muodostaessa kontin levykuvasta, se lisää uuden kerroksen levykuvan päälle, jossa sovelluksia voidaan ajaa. (Understand the architecture n.d.)

Käytetään sitten Docker-binääriä tai ohjelmointirajapintaa, Docker CLI kertoo taustaprosessille, kun kontti otetaan käyttöön. Eli syöttäessä komento ”docker run” tapahtuu seuraavat toimenpiteet.

Aluksi haetaan levykuva joko lataamalla se paikallisesti tai ulkoverkosta. Jos levykuva on jo olemassa, Docker käyttää sitä. Luotavaan konttiin määritetään tiedostojärjestelmä ja muodostetaan read-write-kerros. Sitten allokoidaan verkkorajapinta ja määritetään IP-osoite, joiden avulla kontti keskustelee paikallisen isäntäkoneen kanssa. Lopuksi käynnistetään käyttäjän haluama sovellus. Lisäksi lokitiedostot alkavat seurata konttien toimintaa ja mahdollisia virhetilanteita. Nyt kontti on käytettävissä ja sitä voi hallita asiakasohjelman kautta. (Understand the architecture n.d.)

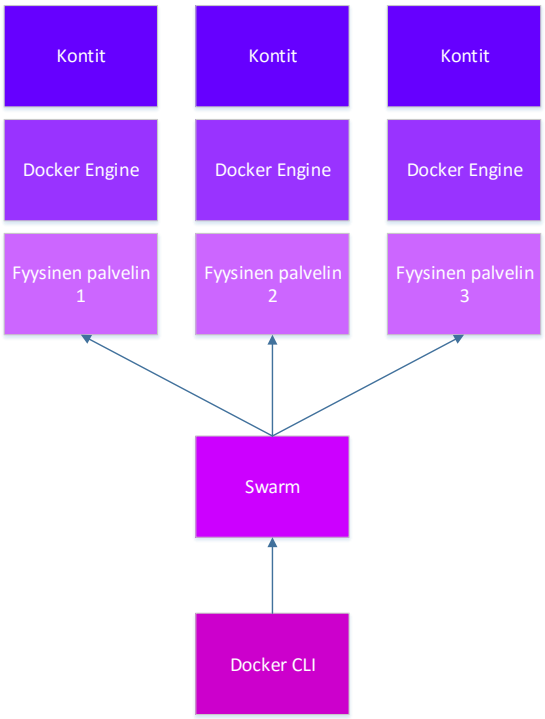
## 3.5 Dockerfile

Docker voi luoda uusia levykuvia käyttäen Dockerfile-tiedostoa hyväksi. Dockerfile on tekstipohjainen tiedosto, joka sisältää kaikki komennot uuden levykuvan muodostamiseen. Dockerfile sisältää ohjeet ohjelmien lataamiseen, niiden ajamiseen, porttien avaamiseen, kansioiden lisäämiseen ja ympäristömuuttujien luomiseen. ”Docker build”-komento rakentaa levykuvan Dockerfile-tiedostosta ja kontekstista. Dockerfile-määritelmän kontekstilla tarkoitetaan tiedostoja kohteissa PATH tai URL. PATH-kansio sijaitsee paikallisessa tiedostojärjestelmässä. URL ilmoittaa Git-tiedostovaraston (Git repository) sijainnin. PATH-kohde sisältää kaikkia alihakemistot, ja URL sisältää tiedostovarastot sekä niiden alimoduulit(submodule). (Dockerfile reference n.d.)

## 3.6 Docker Swarm

Docker Swarm tarjoaa klusterointiominaisuuden, jonka avulla voidaan muuttaa joukko Docker-kontteja näyttämään yhtenä virtuaalisena kokonaisuutena. Tällä re-

surssivarannolla voidaan ajaa sovelluksia käytännössä samanlaisesti kuin yhdessä laiteessa. Docker Swarm tarjoaa standardin Docker-ohjelmointirajapinnan, jonka avulla taustaprosessin kanssa keskustelevat työkalut voivat käyttää Swarmia skaalautumaan moniin eri isäntiin. Swarmin skaalausta on testattu tuhannella solmulla ja 50000 kontilla ilman suorituskyvyn heikkenemistä. Kuviossa 5 esitellään Swarmin toimintamalli (Docker Swarm n.d.)




Kuvio 5. Docker Swarm

Hyvä palvelun saatavuus luodaan Swarm Manager-ominaisuuden avulla. Näin voidaan luoda monia Swarm-isäntiä klusteriin ja määritellä käytänteitä johtajien valintoihin vikatilanteiden sattuessa. Manager tarjoaa testikäytössä olevan tuen konttien siirtämisestä, jos joku solmu kaatuu. Se kertoo myös virheilmoituksia, jos joku solmu ei pysty liittymään klusteriin. (Docker Swarm n.d.)

Swarm manageri huolehtii komentojen hyväksymisestä, jotka tulevat Swarm-klusterin sisälle. Lisäksi se aikatauluttaa klusterin resurssienkäyttöä. Jos Swarm-manageri

muuttuu tavoittamattomaksi, jotain klusterin toimintoja ei voi toteuttaa vikatilanteen aikana. Tämä ei ole sallittavaa suurissa tuotantoympäristöissä. (Plan for Swarm in production n.d.)

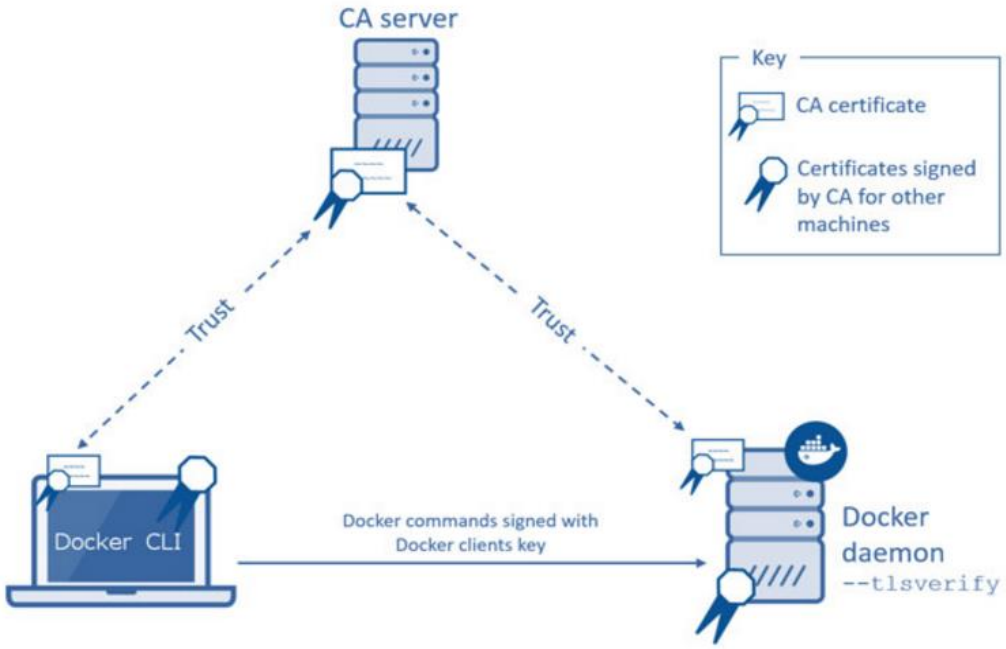
Managerit toimivat klusterissa aktiivisena tai passiivisena, joista yksi manageri on ensisijainen ja muut toissijaisia. Toissijaiset managerin toimivat valmiustilassa, eli niitä ajetaan ensisijaisen managerin taustalla. Toissijaiset managerit ottavat komentoja vastaan samanlaisesti kuin ensisijaiset. Sekundaariset managerit vain ohjaavat komennot suoraan ensisijaiselle managerille, missä komennot suoritetaan. Jos primaari Swarm-manageri kaatuu, uusi valitaan selvinneistä sekundaarisista managereista. (Plan for Swarm in production n.d.)

## 3.7 Transport Layer Security

Kaikki Docker Swarm klusterin laiteet täytyvät yhdistää taustaprosessinsa verkkoporttiin. Tämä luo tietoturvauhan, jonka takia Docker tukee Transport Layer Security (TLS) salausprotokollaa. TLS käyttää hyväksi jaetun avaimen infrastruktuuria eli PKI-tekniikka (public key infrastructure). PKI on yhdistelmä tietoturvajärjestelmiä ja menettelytapoja, mitä käytetään luomaan ja hallitsemaan digitaalisia sertifikaatteja. Nämä sertifikaatit ja infrastruktuurit suojaavat digitaalisia kommunikointeja käyttäen todennus- ja salausmenetelmiä. (Overview Swarm with TLS n.d.)

On mahdollista määritellä Dockerin komentorivirajapinta (command-line interface eli CLI) ja taustaprosessi käyttämään TLS-todennusta. Tämä tarkoittaa sitä, että kaikki kommunikointi komentorivin ja taustaprosessin kanssa pitää olla yhteydessä luotettuun digitaalisen sertifikaattiin. Komentorivirajapinta on tarjottava sertifikaatin taustaprosessille, jotta se alkaa hyväksyä tulevia käyttäjän komentoja. Taustaprosessin on myös luotettava CLI-rajapinnan sertifikaatteihin. Tähän tarkoitukseen pitää muodostaa yhteys johonkin kolmanteen osapuoleen. Tämä kolmas luotettu osallinen on Docker-ympäristössä CA-palvelin (Certificate Authority). CA hoitaa sertifikaattien tekemisen, ylläpidon ja poistamisen. Luottaminen osapuolien välillä muodostetaan asentaen CA pääsertifikaatin isäntälaitteeseen, jossa ajetaan Dockerin taustaprosessia. CLI pyytää tämän jälkeen omaa sertifikaattia CA-palvelimelta. Kun käyttäjä syöttää komennon, CLI-rajapinnan on lähetettävä sertifikaatin taustaprosessille ennen

kun toiminnot voidaan hyväksyä. Taustaprosessi tarkistaa sertifikaatin, ja koska se luottaa CA-palvelimeen, se voi automaattisesti luottaa kaikkiin sertifikaatteihin, joita palvelin on kirjoittanut. Jos kaikki on kunnossa, taustaprosessi hyväksyy käyttäjän komennon. Tämä TLS-toimintamalli voidaan hahmottaa paremmin kuviosta 6. (Overview Swarm with TLS n.d.)




Kuvio 6. TLS- salausprotokolla Dockerissa (Overview Swarm with TLS n.d.)

# 4 Muut käytetyt järjestelmät

## 4.1 CoreOS

CoreOS käyttöjärjestelmä on suunniteltu silmällä pitäen turvallisuutta, jatkuvuutta ja luotettavuutta. CoreOS ei asenna sovelluksia tai päivityksiä pakettimanagerilla, vaan tähän tarkoitukseen käytetään Linux-kontteja. Tarjottavaan palveluun liittyvä koodi ja kaikki muut käytettävät komponentit on sisällytetty konttiin, jota voidaan ajaa yhdessä tai useassa CoreOS-koneessa. (Using CoreOS n.d.)

CoreOS ei sisällä mitään ylimääräistä. Siinä ovat mukana vain tarvittavat komponentit kontteihin ja niiden sisällä pyöriviin sovelluksiin. Käytännössä CoreOS koostuu vain Linux kernelistä, prosessien käynnistämistä ja sammuttamista hallinnoivasta systemd-järjestelmästä. Kuten myös palveluista, joka avulla hallitaan käyttöjärjestelmän konfigurointia ja sen päivittämistä. CoreOS ei sisällä pakettimanageria, vaan se käyttää sovelluksien asennukseen kontteja hyväksi. Näin saadaan kyky luoda automaattisesti eristettyjä ympäristöjä ajamaan haluttuja prosesseja, joita voidaan vapaasti siirtää ja kloonata toisiin palvelimiin. Docker on yleisimmin käytetty konttijärjestelmä CoreOS:ssä. Erillisiä kontteja voidaan myös luoda käsin tai ladata valmiiksi tehtyjä levykuvia jotka ovat yhteensopivia Linux Containers-työkalujen kanssa. (CoreOS – a new approach to Linux-based server systems 2013.)

CoreOS-työkaluja voidaan käyttää automaattisesti tunnistamaan saatavilla olevia palveluita. Lisäksi voidaan käyttöönottaa joukko palvelimia yhdellä konfiguraatiolla sekä pystytään yhdistämään palvelimia yhdeksi klusteroiduksi järjestelmäksi. (CoreOS – a new approach to Linux-based server systems 2013.)

CoreOS käyttää klusterin hallintaan Etcd-järjestelmään (etc distributed), joka siirtää konfiguraatiotietoja eri laitteiden välillä. Tämä komponentti on myös käytössä palveluiden paikantamisessa järjestelmän sisällä. Konteissa ajettavia ohjelmia ajoitetaan ja hallitaan klusterissa käyttäen työkalua nimeltä fleet. Fleetin avulla voidaan hallita koko klusterin prosesseja yhdestä pisteestä käsin. Eli klusteri nähdään yhtenä kokonaisuutena eikä tarvita huolehtia yksittäisistä laitteista erikseen. (Elligwood 2014.)

## 4.2 Etc distributed eli Etcd

Etcd on avoimen lähdekoodiin perustuva työkalu, jonka avulla voidaan jakaa konfiguraatioita ja paikantaa palveluita CoreOS-klusterissa (Getting Started with etcd n.d). Etcd on lyhennetty termistä ”/etc distributed” eli toisin sanoen jaettu Linuxin etc-kansio. Etcd-työkalun avulla voidaan havainnoida klusterin laitteita yhdenmukaisen näkymän kautta. Järjestelmään tehtävät muutokset tulevat näkyviin Etcd:n avulla kaikkiin koneisiin, mutta ei pakolla samaan aikaan. (Corbet 2014.)

Etcd:tä ajetaan jokaisessa klusterin koneessa. Se hallitsee päälaitteen (master) uudelleen valitsemisen, jos tapahtuu verkon vikatilanteesta johtuva katkos alkuperäiseen

päälaitteeseen. Sovellukset joita ajetaan CoreOS klusterissa, voivat lukea ja kirjoittaa tietoa etcd-työkaluun. (Getting Started with etcd n.d.)

Tärkein etcd:n ominaisuuksista on organisoida koneita yhdeksi klusteriksi. CoreOS käyttää etcd:tä käsittelemään eri laitteissa ajettavien sovelluksien keskinäistä yhteystyötä klusterissa. Jotta voidaan linkittää joukko CoreOS-laitteita klusteriksi, etcd-instanssit pitää olla yhteydessä toisiinsa. (CoreOS Cluster Discovery. n.d.)

## 4.3 Vagrant

Vagrant on työkalu, jonka avulla voidaan luoda kokonaisia kehitysympäristöjä nopeasti. Vagrant keskittyy helppokäyttöisyyteen ja automatisoituun toimintaa. Näin säästetään ympäristöjen asennukseen käytettävää aikaa. (About Vagrant n.d.)

Vagrantin avulla voidaan määritellä kertakäytettäviä ympäristö, ilman että uhrataan mitään työkaluja, joita on totuttu käyttämään ennen järjestelmissä, jotka käyttöönotetaan Vagrantilla. Käyttäjän luodessa yhden Vagrantfile-tiedoston ja syöttää komennon ”vagrant up” koko ympäristö asennetaan ja määritellään toimintakuntoon automaattisesti. Jos toinen käyttäjä luo oman ympäristön käyttäen samaa konfiguraatiotiedostoa, muodostuu täysin samanlainen ympäristö. Eli jos joku saa tehtyä toimivan konfiguraatiotiedoston, muut voivat käyttää sitä ja saavat saman tuloksen.

Vagrant on myös hyvä tapa luoda kertakäyttöisiä järjestelmiä testaus käyttöön, koska samoja konfiguraatioita voidaan käyttää paikallisissa virtualisointialustoilla kuten Virtualbox tai pilvipalvelualustoilla. Lisäksi ympäristön voi tuhota komennolla ”vagrant destroy”, joka tapahtuu käytännössä samalla tavalla kuin luomisprosessi. On myös mahdollista luoda suoraan ympäristö, jonka avulla voidaan ajaa verkkosovellusta. (Why Vagrant n.d.)

## 4.4 VirtualBox

VirtualBox on virtualisointialusta, jonka avulla voidaan ajaa virtuaalikonetta isäntäkoneessa, ilman että sille annetaan suoraa pääsyä fyysiseen laitteistoon. VirtualBox-alustaa voidaan käyttää Windows-, Linux- ja muissa AMD/Intel pohjaisissa käyttöjärjestelmissä. Isäntä käyttöjärjestelmään tulee näkyviin ikkuna, jonka kautta käyttäjä

voi hallita VirtualBoxin avulla toimivaa virtuaalikonetta. Jotkut Linux versiot ovat optimoitu virtualisointi käyttöön, mutta kaikki käyttöjärjestelmät toimiva VirtualBoxissa yhtä hyvin. VirtualBoxin avulla voidaan luoda eristettyjä ympäristöjä, joissa on mahdollista ajaa uusia ohjelmia vahingoittamatta isäntäkäyttöjärjestelmää. Lisäksi voidaan ajaa vanhoja käyttöjärjestelmiä tai ohjelmia, joita ei voi uusissa järjestelmissä enää käyttää. Mitään erikoislaitteistoa ei tarvita pyörittämään virtuaalikoneita, mutta jotkut prosessorit ovat tehty virtualisointi käyttöä silmällä pitäen ja niiden avulla saa paremman suorituskyvyn. (VirtualBox n.d.)

## 4.5 Contriboard ja sen komponentit

Contriboard on tikettijärjestelmä, joka avulla halutaan korvata paperilappujen kirjoittamisen digitaalisella järjestelmällä. Omia tiketti pohjia voi jakaa muiden käyttäjien kanssa ja jokainen pystyy muokkaama samaa pohjaa. Contriboard mahdollistaa eri pohjien käyttämistä edistämään työntekoa. (Contriboard - Digitize ideas together while brainstorming n.d.)

Contriboard tarvitsee viisi eri komponenttia toimiakseen Teamboard-API, Teamboard-IO, Teamboard-Client, Redis ja MongoDB. Client on yksinkertainen SPA –sovellus (single-page-application). SPA tarkoittaa verkkosivustoa, mikä toiminta vastaa sulavaa työpöytäsovellusta enemmän kuin normaalia verkkosivustoa. API vastaan todennuksesta ja resursseista. IO –palvelu välittää tapahtumia API ja Client –komponenttien välillä. (Architecture n.d.)

Redis on avoimenlähdekoodin muistin tietokantapankki, mitä voidaan käyttää tietokantana, välimuistina tai viestien murtajana (Introduction to Redis, N.d). MongoDB on asiakirja tietokanta, minkä avulla saadaan hyvä suorituskyky, saatavuus ja automaattinen skaalaus (Introduction to MongoDB n.d).

## 4.6 HAproxy

HAproxy eli High Availability Proxy on avoimenlähdekoodin kuormantasausohjelmisto ja välityspalvelin ratkaisu. Haproxy avulla voidaan parantaa palvelinympäristön suorituskykyä ja luotettavuutta jakamalla verkkoliikennettä usean palvelimen välillä. Haproxy hyödyntää ACL (Access Control List) eli käytönvalvontalistoja, jotta voidaan

siirtää käyttäjiltä tulevaa liikennettä (frontend) palvelimien päähän (backend). On myös mahdollista asettaa toimimaan useampia HAproxy palvelimia, jotta saadaan palveluille parempi saatavuus sekä viansietoisuus. Haproxy tukee useita eri verkkoliikenteen jonotusmenetelmiä, jos käyttäjiltä tulee määriteltyä enemmän pyyntöjä yhtä aikaa. (Anicas 2014.)

### 4.7 Locust

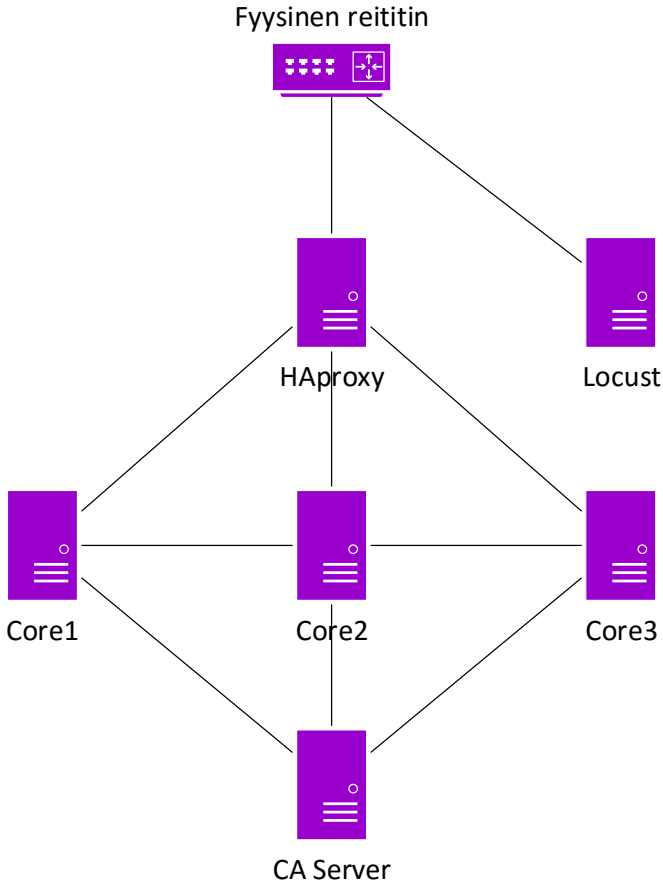
Locust on testaustyökalu, joka on tehty verkkosivujen kuorman keston tarkkailuun. Locustin avulla saadaan selville, kuinka monta yhtäaikaista käyttäjää luotu järjestelmä pystyy käsittelemään. Yhden laitteen avulla voidaan luoda jopa tuhansia keinotekoisia käyttäjiä kuormittamaan omaa verkkosivua. Locust käytetään verkkosivussa olevasta käyttäjärajapinnasta käsin. Siitä näkee kaiken tarvittavat tiedot reaaliajassa. (What is Locust n.d.)

Lisäksi Locust prosessi voidaan jakaa monien eri koneiden kesken, jos halutaan luoda vielä enemmän käyttäjiä. On mahdollista asettaa luotujen käyttäjien olemaan tekemään mitään. Näin niiden toimita saadaan realistisemman oloiseksi ja samalla käyttäjät päättävät mitä tekevät seuraavaksi. Tärkein arvo onkin mittausarvo RPS (Request per second) eli pyyntöjen määrä sekunnissa. (What is Locust n.d.)

## 5 Toteutus

### 5.1 Testiympäristön esittely

Docker ja Docker Swarmin ominaisuuksien esittelyä varten luotiin testiympäristö omalle koneelle. Testiympäristöä ajettiin Windows 10 -käyttöjärjestelmässä käyttäen VirtualBox -virtualisointialustaa. Vagrantin avulla luotiin neljä CoreOS- ja kaksi Ubuntu- virtuaalikonetta VirtualBox- alustaan. Testiympäristö on esittely kuviossa 7.




Kuvio 7. Testiympäristö

Ensimmäiseen Ubuntu -koneeseen asennettiin HAproxy, jonka kautta kaikki verkkoliikenne kulkee klusteriin ja liikenteen kuorma pystytään tasaamaan eri komponenttien kesken. Toiseen Ubuntu- virtuaalikoneeseen tuli Locust –työkalu, jota käytettiin luomaan liikennettä klusteriin ja Contriboard –komponentteihin. Näin saatiin todennettua, että Swarm -klusteri ja verkkosovellus toimivat odotetulla tavalla.

Jokaiseen CoreOS -koneeseen luotiin kaksi konttia, jossa ajettiin Docker Swarmin agent- ja master- komponentteja. Olisi ollut mahdollista käyttää vain yhtä Master -konttia, mutta vikatilanteen sattuessa pitäisi palvelu käynnistää uudestaan manuaalisesti. Tämän vuoksi jokaisessa Swarm -klusterin koneessa on master -kontti, ja ne pystyvät päättämään kuka niistä on ensisijainen master -laite. Neljäs CoreOS -koneista luovutettiin TLS -salausprotokollan CA –palvelimeksi, jota muut klusterin koneet käyttävät varmentamaan omat sertifikaatti tietonsa.

Docker Swarm klusterin solmuissa ajettiin konttien sisässä Contriboard –verkkosovelluksen komponentteja API, Client, IO, Redis ja MongoDB. Tällä sovelluksella voitiin havainnoida, että klusteri toimii oletetulla tavalla ja saadaan hyödyllistä tietoa ympäristöstä.

## 5.2 Asennus

### 5.2.1 Virtuaalikoneiden käyttöönotto

Testiympäristö pystytettiin käyttäen Vagrant -työkalua, jonka avulla voidaan käyttöönottaa useita virtuaalikoneita kerralla ilman, että jokaista tarvitsee erikseen manuaalisesti konfiguroida. Samalla voitiin käyttää CoreOS:n cloud-config -tiedostoa, jolla voitiin määrittää käyttöönotettavat koneet, eli ei tarvinnut manuaalisesti kirjoittaa jokaista komentoa erikseen virtuaalikoneille.

Windows- isäntäkoneeseen asennettiin Vagrant. Githubin kautta kopioitiin tarvittavat tiedostot CoreOS- käyttöjärjestelmään liittyen. Sen mukana tulevat myös kaikki Dockeriin liittyvät komponentit. Tiedostojen kloonaus Githubista tapahtuu ”git clone” –komennon avulla.

*$ git clone https://github.com/coreos/coreos-vagrant.git vagrant-docker-swarm*

Githubista kloonattu kansio sisälsi kaksi tärkeää tiedostoa; vagrantfile ja user-data. Vagrantfile-tiedoston avulla voitiin määrittää, kuinka monta CoreOS-konetta käyttöönotetaan kerrallaan. Esimerkiksi vagrantfile rivillä ”num_instances = 4” voitiin asentaa neljä virtuaalikonetta VirtualBoxiin.

Ladattu Vagrantfile on valmis skripti, joten se toimii ilman muutoksia. Sinne lisättiin muutama rivi helpottamaan klusterin ja verkkosovelluksen käyttöönottoa. Omasta Githubsta kloonattiin konfiguraatiotiedostoja jokaiseen CoreOS-laitteeseen. Tämän jälkeen skripti luo uusia kansiota koneisiin ja kopioi tiedostot oikeisiin kohteisiin.

Tiedoston ”50-vagrant1.network” avulla määriteltiin verkkoasetukset jokaiselle koneelle. ” Custom.conf”-tiedostolla asetetiin Dockerin klusteriin liittyvät parametrit. Tekstitiedostojen ”api.txt”, ”client.txt” ja ”io.txt” avulla konfiguroitiin Contriboard-verkkosovelluksen ympäristömuuttujat.

Käytetyt Vagrantfile-tiedostot löytyvät tämän dokumentin liitteistä 1 ja 2. Seuraavaksi on esitetty skriptiin itse lisätyt rivit.

*git clone https://github.com/VilleAnttila/coretest*

*sudo mkdir /etc/systemd/system/docker.service.d*

*sudo mkdir /home/core/envit*

*sudo cp /home/core/coretest/50-vagrant1.network /etc/systemd/network/50-vagrant1.network*

*sudo cp /home/core/coretest/custom.conf /etc/systemd/system/docker.service.d/custom.conf*

*sudo cp /home/core/coretest/api.txt /home/core/envit/api.txt*

*sudo cp /home/core/coretest/client.txt /home/core/envit/client.txt*

*sudo cp /home/core/coretest/io.txt /home/core/envit/io.txt*

User-data –tiedosto on toisin sanoen CoreOS- käyttöjärjestelmän cloud-config. Tätä samaa skriptitiedostoa voidaan käyttää, jos luodaan ympäristöä johonkin pilvipalveluun. Etcd:tä käytettiin, jotta klusterin laitteet löytävät toisensa ja pystyvät keskustelemaan keskenään. Etcd tarvitsee toimiakseen uniikin discovery tokenin. Tämä luodaan ”discovery.etcd.io/new?size\=4” -sivuston avulla. Kun edellä mainittu osoite syötetään Internet-selaimeen, avautuu sivusto, josta voitiin nähdä discovery token. Saatu token liitetään discovery riviin cloud-config-tiedostossa. Samassa skriptissä pakotettiin etcd käynnistymään ja määritettiin siihen liittyvät IP-osoitteet ja portit.

Seuraavaksi on esitetty käytetty cloud-config:

*#cloud-config*

*coreos:*

*etcd2:*

*discovery: https://discovery.etcd.io/888fd1e440faf680a7abb3fd934da6fd*

*advertise-client-urls: http://$public_ipv4:2379*

```
    initial-advertise-peer-urls: http://$public_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$public_ipv4:2380,http://$public_ipv4:7001
  units:
    - name: etcd2.service
      command: start
```

Nämä toimenpiteet tehtyä, voitiin Windows-komentorivin kautta kirjoittaa komento "vagrant up". Tämän jälkeen Vagrant automaattisesti asensi halutun määrä CoreOS-virtuaalikoneita VirtualBoxiin käyttäen määriteltyjä konfiguraatiotiedostoja. Asennuksen valmistuttua, oli VirtualBoxiin ilmestynyt käytettäväksi CoreOS-koneet.

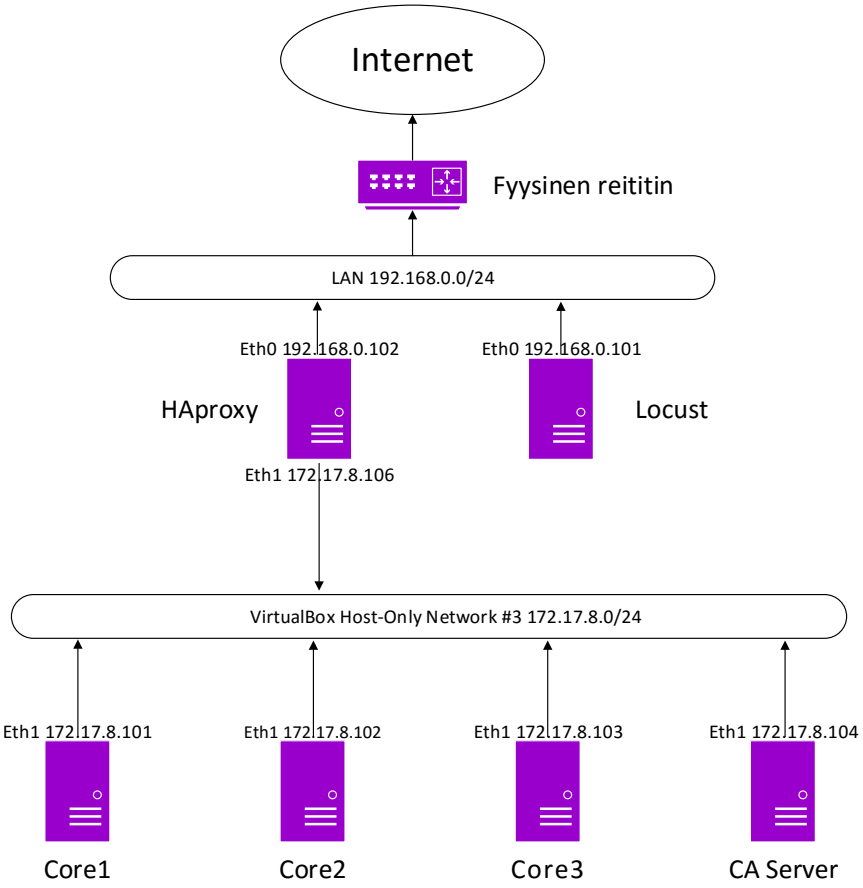
Tällä samalla kaavalla luotiin myös kaksi Ubuntu-virtuaalikonetta HAproxy- ja Locust-ohjelmien käyttöön. Käytetty Vagrantfile oli Ubuntun tapauksessa yksinkertaisempi verrattuna CoreOS-käyttöjärjestelmään liittyvään tiedostoon.

Vagrantfilessä määritettiin verkotus, joka esitellään kappaleessa 5.2.2 ja HAproxyn asennus, josta on laajempi selvitys kappaleessa 5.3.2.

### 5.2.2 Klusterin lähiverkon konfigurointi

Testiympäristössä oli käytössä kaksi lähiverkkoa 192.168.0.0 ja 172.17.8.0. Fyysisen koneen ja reitittimen lähiverkko oli 192.168.0.0. Toinen sisäverkko oli 172.17.8.0, jossa kaikki virtuaalikoneet sijaitsivat. Tämä verkko luotiin Vagrantin avulla automaattisesti, käyttäen VirtualBoxin ”Host-only”- verkkosovitinta. ”Host-only” –adapterin avulla virtuaalikoneet voivat keskustella toistensa kanssa.

Ubuntu toimii reitittimenä kahden verkon välillä. Ubuntu –virtuaalikoneeseen määritettiin kaksi verkkosovitinta VirtualBoxissa. ”Host-only” –adapteri suuntautui verkkoon 172.17.8.0 ja ”Bridged”- adapteri fyysisenkoneen verkkoon 192.168.0.0. Kuviossa 8 on esitetty ympäristön verkkotopologia.




Kuvio 8. Verkkotopologia

Jokaiseen CoreOS-laitteeseen luotiin Vagrantin avulla yksi rajapinta ”eth1”, jonka kautta laite pääsee verkkoon 172.17.8.0. Oli myös määritettävä DNS-palvelin, jotta verkkoyhteys saatiin muodostettua. Tähän tarkoitukseen käytettiin Googlen DNS-osoitetta 8.8.8.8.

CoreOS-laitteet tarvitsivat reitin verkkoon 192.168.0.0, jota kautta muodostettiin yhteys internettiin. CoreOS –käyttöjärjestelmän verkkokonfiguraatiotiedosto sijaitsee kohteessa /etc/systemd/network/50-vagrant1.network. Alla on esitetty Core1 -koneen esimerkki verkkokonfiguraatio.

*[Match]*

*Name=eth1*

*[Network]*

*DNS=8.8.8.8*

*Address=172.17.8.101/24*

*Gateway=172.17.8.106*

*[Route]*

*Gateway=172.17.8.106*

*Destination=192.168.0.0/24*

Ubuntuun tuli kaksi rajapintaa: ”eth0”, johon oli asetettu verkko 192.168.0.0 ja ”eth1” -rajapinta suuntautui verkkoon 172.17.8.0. Rajapinta ”eth0” toimi DHCP:n (Dynamic Host Configuration Protocol) kautta, eli se saa automaattisesti IP-osoitteen. ”Eth1”suuntautui klusterin verkkoon ja siihen määritettiin staattinen osoite 172.17.8.106. Klusterin muut laiteet käyttivät tätä osoitetta yhdyskäytävänä ulko-verkkoon. Ubuntun verkkokonfiguraatiotiedosto sijaitsee polussa */etc/network/interfaces.* Alla nähdään luodut konfiguraatiot edellä mainittuun kohteeseen.

*auto lo*

*iface lo inet loopback*

*auto eth0*

*iface eth0 inet dhcp*

*auto eth1*

*iface eth1 inet static*

*address 172.17.8.106*

*network 172.17.8.0*

*netmask 255.255.255.0*

*broadcast 172.17.8.255*

Jotta verkkoyhteys saadaan reitittymään rajapintojen välillä, pitää kohteeseen /etc/sysctl.conf lisätä rivi ”net.ipv4.ip_forward=1”. Lopuksi luotiin palomuuriin sääntöjä reititykseen liittyen.

*sudo iptables -A FORWARD -o eth0 -i eth1 -s 172.17.8.0/24 -m conntrack --ctstate NEW -j ACCEPT*

*sudo iptables -A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT*

*sudo iptables -t nat -F POSTROUTING*

*sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE*

*iptables-save > /etc/iptables.rules*

### 5.2.3 Klusterin muodostus

Docker Swarmin avulla yhdistettiin CoreOS-koneet yhdeksi klusteriksi. Tämä tarkoitti sitä, että Docker-kontteja voitiin hallita yhdeltä koneelta käsin. Etcd:n avulla Docker Swarm löytää klusteriin liitettävät laiteet. Klusterin muodostus lähtee liikkeelle juuri Etcd:n määritellystä. Konfiguraatiotiedosto sijaitsi kohteessa ”/etc/systemd/system/docker.service.d/custom.conf”, minkä avulla määritetiin klusterin mainostetut portit ja IP-osoitteet. Tänne tiedostoon lisättiin seuraavat rivit:

*[Service]*

*Environment="DOCKER_OPTS=-H=0.0.0.0:2376 -H unix:///var/run/docker.sock --insecure-registry 172.17.8.102:5000 --tlsverify --tlscacert=/home/core/.certs/ca.pem --tlscert=/home/core/.certs/cert.pem --tlskey=/home/core/.certs/key.pem" --cluster-advertise eth1:2376 --cluster-store etcd://127.0.0.1:2379"*

Tässä tiedostossa oli myös TLS-salausprotokollaan liittyviä rivejä, jonka käyttötarkoitus selviää tarkemmin kappaleessa 5.3.

Muutokset tulivat voimaan vasta kun, taustaprosessi ja Docker käynnistettiin uudestaan komennoilla ”systemctl daemon-reload” ja ”systemctl restart docker”.

Docker Swarm toimii siten, että yksi laitteista on aina ”Master” eli isäntä ja muut liittyvät klusteriin ”Agent”-nimikkeellä. Docker Swarmin voi luoda siten, että vain yksi kone sisältää master-kontin. Tässä menettelytavassa on ongelmana, jos master menee vikatilaan, koko klusteri lamaantuu. Tähän auttaa vain manuaalisesti luotu uusi Master tai korjata jo olemassa oleva kontti.

Tässä opinnäytetyössä luotiin HA-menetelmällä Swarm-klusteri. Eli jokaiselle klusterin laitteelle luotiin Master- ja Agent-kontti. Master-laitteista yksi toimii primäärinä ja muut sekundaarisina. Nyt jos master joutuu vikatilaan, joku sekundaarisista valitaan uudeksi primääriksi automaattisesti.

Master-kontti käynnistetiin ”docker run”-komennolla. Ohjeella ”-d” asetetiin kontti asennuksen jälkeen taustalle toimintaan. Ohje ”--name” antoi kontille nimen ja ”--net=host” ilmoitti kontin käyttävän samaa verkkoa isäntäkoneen kanssa. ”Swarm manage” avulla ilmoitettiin, että luotiin Swarm master.

Komentoon lisättiin myös vipu ”-experimental”, koska käytetiin viansieto ominaisuutta, joka oli vielä kokeellisessa vaiheessa. Vivulla ”-H :4000” asetetiin klusteri verkkoliikenne toimimaan TCP-portissa 4000.

Master-kontin sisälsi myös TLS-todennukseen liittyviä vipuja, joiden tarkempi toiminta kerrotaan seuraavassa kappaleessa. Ohjeella ”--replication” kerrottiin, että käytettiin useaa master-konttia klusterissa. Ohje ”--advertise” määrittää muille koneille klusterin mainostettavan IP-osoitteen ja portin. Lopuksi ”etcd://127.0.0.1:2379” kohdalla kerrottiin etcd:n IP-osoite ja portti, jonka avulla laitteet löysivät toisensa.

```
docker run -d --name master1 --net=host \
 -v /home/core/.certs:/home/core/.certs swarm \
 -experimental manage -H :4000  --tlsverify \
--tlscacert=/home/core/.certs/ca.pem \
 --tlscert=/home/core/.certs/cert.pem \
 --tlskey=/home/core/.certs/key.pem --replication \
--advertise 172.17.8.101:4000 etcd://127.0.0.1:2379
```

Swarm agentti käynnistettiin muuten samalla tavalla kuin master, mutta ohjeella ”--addr” ilmoitettiin palvelimen IP-osoite ja etcd-klusterin mainostettavan portin. Tässä pitää huomioida, ettei tässä sekoita porttia 2376, Swarm klusterin porttiin 4000 tai etcd store porttiin 2379.

*docker run -d --name agent1 --net=host swarm join --addr=172.17.8.101:2376 etcd://127.0.0.1:2379*

## 5.3 Transport Layer Security-salauksen asennus

Docker Swarmissa on tietoturvauhka, koska Docker-taustaprosessi sidotaan porttiin ja jos ollaan yhteydessä epäluotettavaan verkkoon kuten Internet, joutuu verkko uhanalle. Tämän takia otettiin käyttöön TLS-salausprotokolla Swarm klusteriin.

Yksi CoreOS- koneista varattiin CA- palvelimeksi. Ensimmäinen vaihe TLS:n konfiguraatiosta oli luoda CA- palvelimeen kaksi kansiota avainten luontiin. Tässä tapauksessa luotiin kansiot polkuihin ” /etc/docker/ssl ” ja ” ~/.docker ”. Tämän toimenpiteen jälkeen tuotettiin yksityinen avain kansioon ” ~/.docker ”, käyttäen ”openssl genrsa”-komentoa hyväksi.

*openssl genrsa -out core4-priv-key.pem 2048*

Luodusta avaimesta tehtiin ca.pem-tiedosto käyttäen seuraavalla komennolla.

*openssl req -x509 -new -nodes -key ~/.docker/core4-priv-key.pem -days 10000 -out ~/.docker/ca.pem -subj '/CN=docker-CA'*

Tämän jälkeen voitiin kopioida ca.pem kansioon /etc/docker/ssl. Avaintiedoston ca.pem lisäksi jokainen CoreOS -kone tarvitsee oman coreX-priv-key.pem ja coreX-cert.pem tiedostot. Merkki X ilmaisee koneen numeroa, esimerkiksi Core1-koneen tilanteessa tiedostonnimi oli muotoa core1-cert.pem. Nämä pem-tiedostot luotiin testiympäristössä kansioon ~/.docker/ seuraavilla komennoilla.

*openssl genrsa -out coreX-priv-key.pem 2048*

*openssl req -subj "/CN=coreX" -new -key coreX-priv-key.pem -out coreX.csr*

*openssl x509 -req -days 1825 -in coreX.csr -CA ca.pem -CAkey core4-priv-key.pem -CAcreateserial -out coreX-cert.pem -extensions v3_req -extfile ~/.docker/openssl.cnf*

*openssl rsa -in coreX-priv-key.pem -out coreX-priv-key.pem*

Kun jokaiseen laitteen oli luotu omat pem-tiedostot, voitiin ne kopioida SSH-yhteyden kautta SCP-komentoa käyttäen CoreOS -koneisiin.

*ssh core@172.17.8.10X 'mkdir -p /home/core/.certs'*

*scp ./ca.pem core@172.17.8.10X:/home/core/.certs/ca.pem*

*scp ./coreX-cert.pem core@172.17.8.10X:/home/core/.certs/cert.pem*

*scp ./core-priv-key.pem core@172.17.8.10X:/home/core/.certs/key.pem*

Näiden toimenpiteiden jälkeen, jos yritti syöttää komentoa klusteriin ilman TLS –salauksen käyttämistä tuli seuraavailmoitus:

*docker -H tcp://172.17.8.101:4000 info*

*Get http://172.17.8.101:4000/v1.22/info: malformed HTTP response "\x15\x03\x01\x00\x02\x02".*

** Are you trying to connect to a TLS-enabled daemon without TLS?*

Jotta Dockerin taustaprosessi hyväksyy käyttäjän komennot, oli syötettävä aluksi seuraavat tiedot.

*export DOCKER_HOST=172.17.8.101:4000*

*export DOCKER_TLS_VERIFY=1*

*export DOCKER_CERT_PATH=/home/core/.certs*

Docker Swarmin master-kontin ”docker run”-komennossa oli määritettävä sertifikaatti- ja avaintiedostojen sijainnit, ohjeilla tlscacer, tlscert ja tlskey. Lisäksi vivulla ”-v” eli volume asetettiin avain- ja sertifikaattitiedostojen looginen levy konttiin.

*docker run -d --name master1 --net=host* \

*-v /home/core/.certs:/home/core/.certs swarm* \

*-experimental manage -H :4000 --tlsverify* \

*--tlscacert=/home/core/.certs/ca.pem* \

*--tlscert=/home/core/.certs/cert.pem* \

*--tlskey=/home/core/.certs/key.pem --replication* \

*--advertise 172.17.8.101:4000 etcd://127.0.0.1:2379*

## 5.4 Contriboard

### 5.4.1 Lähtökohdat Contriboard-verkkosovelluksen asennukseen

Contriboard-verkkosovellusta käytettiin testiympäristön toiminaan todennukseen. Verkkosovelluksen komponentit jaettiin klusterin solmuihin ja niille annettiin oma uniikki TCP-portti, jotta liikenne saatiin erotettua toisistaan. Taulukossa 1 nähdään koneet ja niiden IP-osoitteet, johon Contriboard-kontit asennettiin. Lisäksi mainitaan TCP-portit, jotka liitettiin kontteihin.

Jokaiseen CoreOS-koneeseen asennettiin API-kontti, joka huolehtii resursseita ja todentamisesta Contriboard-ympäristössä. Näihin kontteihin liitettiin portti 9001. Core2-virtuaalipalvelimelle asennettiin Client-kontti, jonka avulla hallitaan käyttäjien toimintoja. Kontti liitettiin porttiin 80.

Jokaiselle koneelle lisättiin komponentit: IO, MongoDB ja Redis. IO- kontin avulla mahdollistetaan kommunikointi API- konttien ja Clientin välillä. IO toimi portissa 9002. MongoDB toimi klusterissa tietokantana, johon kaikki käyttäjien tiedot tallennetaan. Mongo asetetiin toimiman portissa 27017. Redis on muistin tietokantapankki tapahtumien hallintaan, joka käytti porttia 6379.

| | | | |
|---|---|---|---|
| Core1 | API1 | 172.17.8.101 | 9001 |
| Core1 | IO1 | 172.17.8.101 | 9002 |
| Core1 | Redis1 | 172.17.8.101 | 6379 |
| Core1 | MongoDB1 | 172.17.8.101 | 27017 |
| Core2 | Client | 172.17.8.102 | 80 |
| Core2 | API2 | 172.17.8.102 | 9001 |
| Core2 | IO2 | 172.17.8.102 | 9002 |
| Core2 | Redis2 | 172.17.8.102 | 6379 |
| Core2 | MongoDB2 | 172.17.8.102 | 27017 |
| Core3 | API3 | 172.17.8.103 | 9001 |
| Core3 | IO3 | 172.17.8.103 | 9002 |
| Core3 | Redis3 | 172.17.8.103 | 6379 |
| Core3 | MongoDB3 | 172.17.8.103 | 27017 |

Taulukko 1. Contriboard-kontit

### 5.4.2 Docker-rekisteri

Contriboard-ympäristön käyttöönottoa helpottamaan luotiin Docker-rekisteri. Tämän palvelun avulla pystyttiin jakaa paikallisesti levykuvia koneiden kesken. Näin konttien käyttöönotto helpottui, kun joka kerta levykuvia ulkoverkosta.

Ensimmäinen toimenpide on luoda kontti rekisteriä varten. Rekisteriin asetettiin TCP-portti 5000 vivulla ”-p”. Seuraavalla komennolla luotiin kyseinen kontti klusteriin:

*docker run -d -p 5000:5000 --name registry registry:2*

Koska klusterissa oli käytössä TLS-salaus, piti erikseen määritellä custom.conf tiedostoon ”--insecure-registry”, jos halutaan käyttää rekisteriä salaamattomana. Ilman tätä ohjetta, rekisteriin ei pysty siirtämään levykuvia. Tässä tapauksessa ei nähty tarpeelliseksi käyttää salausta, koska rekisteri toimi paikallisessa verkossa.

Kun edelliset toimenpiteet oli suoritettu, voitiin alkaa ”docker pull”-komennolla lataamaan levykuvia julkisista rekistereistä.

*docker pull n4sjamk/teamboard-api*

*docker pull n4sjamk/teamboard-client*

*docker pull n4sjamk/teamboard-io*

*docker pull mongo*

*docker pull redis*

Kun levykuvat oltiin saatu ladattua, pystyttiin komennon “docker images” avulla näkemään ladatut levykuvat. Tällä komennolla nähtiin myös levykuvien ID-nimet, jotka tarvittiin, kun leimataan levykuvat omaan rekisteriin käyttäen komentoa ”docker tag”

*docker tag 3be5ab65756a 172.17.8.102:5000/api*

*docker tag f4762f2de240 172.17.8.102:5000/client*

*docker tag cfa2b17ffd55 172.17.8.102:5000/io*

*docker tag ddad4160b92f 172.17.8.102:5000/mongo*

*docker tag 0f0e96f1f267 172.17.8.102:5000/redis*

Leimauksen jälkeen voitiin siirtää “docker push”-komennon avulla levykuvat omaan rekisteriin.

*docker push 172.17.8.102:5000/api*

*docker push 172.17.8.102:5000/client*

*docker push 172.17.8.102:5000/io*

*docker push 172.17.8.102:5000/mongo*

*docker push 172.17.8.102:5000/redis*

### 5.4.3 Contriboardin asennus CoreOS-laitteille

Jokaiselle CoreOS- koneelle luotiin envit- kansio ympäristömuuttujia varten. Tähän kansioon luotiin tekstitiedostot API-, Client- ja IO-konttien käyttöönottoa varten. Nämä toimenpiteet hoidettiin jokaiselle koneelle automaattisesti Vagrantin ja Githubin avulla.

NODE_ENV ilmoittaa ympäristömuuttujatiedostossa, että konttia käytetään tuotantoympäristössä ja kaikkia komponentteja ei löydy paikallisesti samasta koneesta. API tarvitsee tiedon Redis- ja MongoDB-konttien sijainnista. Redis oli sijoitettu jokaiseen laitteeseen, joten vain sen portti ilmoitettiin. "MONGODB_URL" määritti IP-osoitteen, jossa primääri MongoDB sijaitsi. API:n käyttämä portti asetetiin PORT- rivillä.

*NODE_ENV=production*

*MONGODB_URL=mongodb://172.17.8.101:27017/CB*

*REDIS_PORT=6379*

*PORT=9001*

*TOKEN_SECRET=coredocker*

Client tarvitsee tietoon ulkoverkon osoitteet, johon käyttäjät pystyvät yhdistymään. Lisäksi määriteltiin portti 80 IO- ja API-komponenteille.

*NODE_ENV=production*

*IO_URL=http://80.223.133.217*

*IO_PORT=80*

*API_URL=http://80.223.133.217/api*

*API_PORT=80*

IO-kontin ympäristömuuttujatiedostoon määritettiin API_URL kohtaan HAproxyn osoite. Rediksen IP-osoitetta ei tarvinnut ilmoittaa, koska sitä käytettiin paikallisesti jokaisessa laitteessa. Rediksen portti kuitenkin ilmoitettiin varmuuden vuoksi. Lisäksi kerrottiin IO:n oma portti.

*NODE_ENV=production*

*REDIS_PORT=6379*

*API_URL=http://172.17.8.106:80/api*

*PORT=9002*

Näiden toimenpiteiden jälkeen pystyttiin käynnistämään Docker-kontit. ”Docker run”- komennossa määritettiin aluksi ”–d ” vivulla, että kontti käynnistetään taustalle ja ”--name” vivulla asetettiin kontille nimi. Kontin käyttämiä laiteresursseja rajoitettiin vivulla ”–c ” ja ”–m”. C ohjeistuksella hallitaan prosessorinkäyttöä suhdeluvuilla. Kontteihin on asetettu oletuksena arvo 1024, mutta luku ei käytännössä tarkoita mitään, ellei luoda rinnalle toista konttia, jossa on kanssa ilmoitettu ”-c” ohjeistus. Jos 1024 arvo määritetään kahdelle kontille, tarkoittaa se sitä, että prosessoritehoja annetaan kummallekin 50%. Jos luvut olisi 1024 ja 512 niin toiselle kontille annettaisiin 75% tehoja ja toiselle 25%. Vipu ”–m” on sen sijaan helpompi ymmärtää, siinä ilmoitetaan kontille annettava RAM –muistion määrä.

Aiemmin luotu ympäristömuuttujatiedoston sijainti ilmoitettiin vivulla ”--env-file”. Tämän jälkeen kerrottiin missä portissa verkkoliikenne kulkee vivun ”–p ” avulla. Käskyllä ”--expose” asetettiin kontti kuuntelemaan tietyn portin liikennettä. Lopuksi kerrottiin Dockerille, mikä levykuva ladataan kontin käyttöön.

```
docker run -d --name api1 -c 1024 -m 512MB \
 --env-file=/home/core/envit/api.txt \
-p 9001:9001 --expose=9001 172.17.8.102:5000/api
```

Client- ja IO –kontit määritettiin samalla tavalla kuin API:n tapauksessa, mutta eri arvoilla ja porteilla.

Ohjeella ” -l 'com.docker.swarm.reschedule-policy=["on-node-failure"]' ” asetetaan kontti siirtymään toiseen koneeseen vikatilanteen sattuessa. Tämä on vielä kehitteillä oleva ominaisuus, joten Swarm master-kontin komennossa on ilmoitettava ”--experimental”, jotta viansietoisuusominaisuus toimii oikein.

```
docker run -d --name client -c 256 -m 256MB \
-l 'com.docker.swarm.reschedule-policy=["on-node-failure"]' \
 --env-file=/home/core/envit/client.txt \
-p 80:80 --expose=80 172.17.8.102:5000/client


docker run -d --name io1 -c 256 –m 256MB \
--env-file=/home/core/envit/io.txt \
 -p 9002:9002 --expose=9002 172.17.8.102:5000/io
```

Redis käynnistetään jokaiseen CoreOS-koneelle. Rediksen "docker run"-komento menee samalla kaavalla kuin edelliset. Eikä mitään uusia ohjeita tule tässä tapauksessa.

*docker run -d -p 6379:6379 --name redis1 172.17.8.102:5000/redis*

## 5.5 MongoDB asennus

MongoDB-tietokanta asennettiin jokaiseen klusterin CoreOS-koneelle. Tässä ympäristössä käytettiin Mongo-tietokannassa "Replication"-menetelmää eli tiedostot kopioidaan jokaisen solmun kesken. Yksi Mongo-konteista oli "Primary"-statuksella ja muut "Secondary". Jos yksi koneista tuhoutuisi täysin, tietokanta olisi kuitenkin turvassa muilla koneilla.

Ensisijaiseksi Mongo-solmuksi asetettiin Core01. API-kontit tarvitsevat tiedon ensisijaisen MongoDB:n sijainnista, jotta verkkosovellus toimisi oikein. "Replication"-menetelmällä ei jaeta kuormaa eri solmujen kesken, ne toimivat vain varmuuskopiona. On myös olemassa "Sharding"-menetelmä, jonka avulla jaetaan liikennettä komponenttien kesken. Tämän ympäristön tapauksessa "replication"-menetelmä koettiin riittäväksi, koska ympäristöön ei oleteta tulevan paljoa liikennettä.

"Docker run"-komennolla asennettiin Mongo jokaiseen CoreOS-koneeseen. Vivulla "–v" asetettiin tietokannan tiedostopolku kontille. Eri Mongo-tietokantoja hallittiin DNS-nimien avulla, joten ne piti määritellä kontin syöttämisen yhteydessä. Ohjeen

"smallfiles" avulla pienennetään käytettävien tiedostojen kokoa. "ReplSet" asetti käytettävän osion nimen.

```
docker run \
--name mongonode1 \
-v /home/core/mongo-files/data:/data/db \
--hostname="solmu1.testi.com" \
--add-host solmu1.testi.com:172.17.8.101 \
--add-host solmu2.testi.com:172.17.8.102 \
--add-host solmu3.testi.com:172.17.8.103 \
-p 27017:27017 -d 172.17.8.102:5000/mongo \
--smallfiles \
--replSet "rs0"
```

Kun kontit olivat toimintakunnossa, voitiin mennä Core01-koneessa sijaitsevan Mongo-konttiin sisällä "docker exec"-komennolla.

```
docker exec -it mongonode1 /bin/bash
```

Komentoriville on kirjoitettava "mongo" ennen kun päästiin hallitsemaan tietokantaa. Tämän jälkeen muodostettiin klusteri "rs.initiate"-komennolla. Muut solmut lisättiin klusteriin komennolla "rs.add".

```
rs.initiate()

rs.add("solmu2.testi.com")
rs.add("solmu3.testi.com")
```

## 5.6 HAproxy

Haproxy valittiin kuormantasaajaksi, koska siitä oli jo aiempaa kokemusta ja sen kilpailijoilla ei ole mitään dramaattisesti parempaa annettavaa tätä ympäristöä varten.

Testikäyttäjiltä tuli ympäristöön porttiin 80 merkittyä liikennettä, joka ohjattiin oikeisiin Contriboard – komponentteihin HAproxyn avulla. Samalla saatiin käyttöön hyödyllinen tilastosivusto, josta voitiin tarkkailla komponenttien tietoja reaaliajassa. Seuraavilla komennoilla asennettiin HAproxy Ubuntu –virtuaalikoneelle.

Pakettivarastoihin oli aluksi tehtävä päivityksiä, jotta varmistettiin kaikkien tarpeellisten tiedostojen ja hakemistojen olemassaolo.

*sudo add-apt-repository ppa:vbernat/haproxy-1.5*

*sudo apt-get update*

*sudo apt-get dist-upgrade*

*sudo apt-get install haproxy*

Jotta HAproxy saatiin toimimaan odotetulla tavalla, piti tiedostoon /etc/haproxy/haproxy.cfg lisätä muutama rivi konfiguraatioita. Tiedosto koostuu pääosin kuormanjakamisen määrittelystä ja jonotuksen hallinnasta.

Global –osion avulla ilmoitettiin, että Haproxy toimii taustalla ja maxconn –rivillä asetettiin suurin mahdollinen yhteyksien määrä. Jos pyyntöjen määrä ylittää ilmoitetun arvon, joutuu käyttäjien pyynnöt jonottamaan vuoroaan.

*global*

*daemon*

*maxconn 256*

Defaults –osiossa määriteltiin jokaisen muun sen jälkeen tulevat osion tiedot, ellei niissä kumota Defaults-oletusarvoja. Fullconn rivillä asetettiin kuinka monta yhteyttä saa yhtä aikaa olla käynnissä yhdessä backendissä. Maxconn määrittelee mikä on yhden palvelimen suurin sallittu yhteyksien määrä. Jos yhteyksien maksimi määrä ylittyy, liikenne siirretään jonoon, joka määritetään tarkemmin timeout –riveillä.

*defaults*

*mode http*

*fullconn 1024*

*maxconn 256*

*timeout queue 600s*

*timeout connect 5s*

*timeout client 600s*

*timeout server 600s*

Frontend –osiossa asetettiin ohjeet sisään tulevalle liikenteelle. Portista 80 vastaanotettiin liikenne ja se lähetettiin käytönvalvontalistoilla (ACL) eteenpäin backend-palvelimiin. Jos liikenne ei kuulu mihinkään ACL-listaan, se siirretään oletus backendiin eli Client-konttiin.

*frontend http*

*bind *:80*

*mode http*

*acl acl_api path_beg /api*

*acl acl_io path_beg /socket.io*

*use_backend API if acl_api*

*use_backend IO if acl_io*

*default_backend CLIENT*

“Listen stats”-osiolla luotiin parametrit tilastosivustolle, josta voitiin tarkkailla HAproxyyn liitettyjen komponenttien tietoja. Käytännössä tässä määritettiin URI-osoite, jossa tilastosivusto toimi ja sen käyttäjän kirjautumistiedot.

*listen stats :9000*

*mode http*

*stats enable*

*stats uri /haproxy_stats*

*stats auth adminr:Sudo66*

Kolmessa viimeisessä backend-osiossa määritettiin klusterissa olevien konttien sijainnit ja niiden välisen kuormatasauksen. Leastconn-kuormantasaus siirtää liikennettä solmuun, jossa on vähiten liikennettä. Server-rivillä check kohdassa ilmoitettiin, että palvelinta seurataan vikatilanteiden varalta.

```
backend API
        balance leastconn
        timeout server 600s
        timeout connect 5s
        server API1 172.17.8.101:9001 maxconn 128 check
        server API2 172.17.8.102:9001 maxconn 128 check
        server API3 172.17.8.103:9001 maxconn 128 check
```

Client ja IO-palvelimiin asetettiin source ja stick on source kuormantasaus käyttöön. Tämä tarkoitti sitä, että kuorma siirrettiin vain yhteen palvelimeen ja muut olivat varalla. Klusterissa oli kerrallaan käytössä vain yksi Client-kontti, mutta HAproxyyn määriteltiin kaikki palvelimet. Vikatilanteessa Client-kontti siirtyi automaattisesti Swarmin avulla toiseen laitteeseen. HAproxy huomasi tämän muutoksen näillä konfiguraatiolla. Kolme IO-konttia oli käytössä klusterissa koko ajan, mutta HAproxy lähetti liikennettä vain yhteen niistä source-kuormantasauksella.

```
backend CLIENT
        balance source
        hash-type consistent
        stick-table type ip size 1m expire 15m
        stick on src
        server CLIENT1 172.17.8.101:80 maxconn 256 check
        server CLIENT2 172.17.8.102:80 maxconn 256 check
        server CLIENT3 172.17.8.103:80 maxconn 256 check
```

```
backend IO
        balance source
        hash-type consistent
        stick-table type ip size 1m expire 15m
        stick on src
        server IO1 172.17.8.101:9002 maxconn 256 check
        server IO2 172.17.8.102:9002 maxconn 256 check
        server IO3 172.17.8.103:9002 maxconn 256 check
```

# 6 Toiminnan todennus

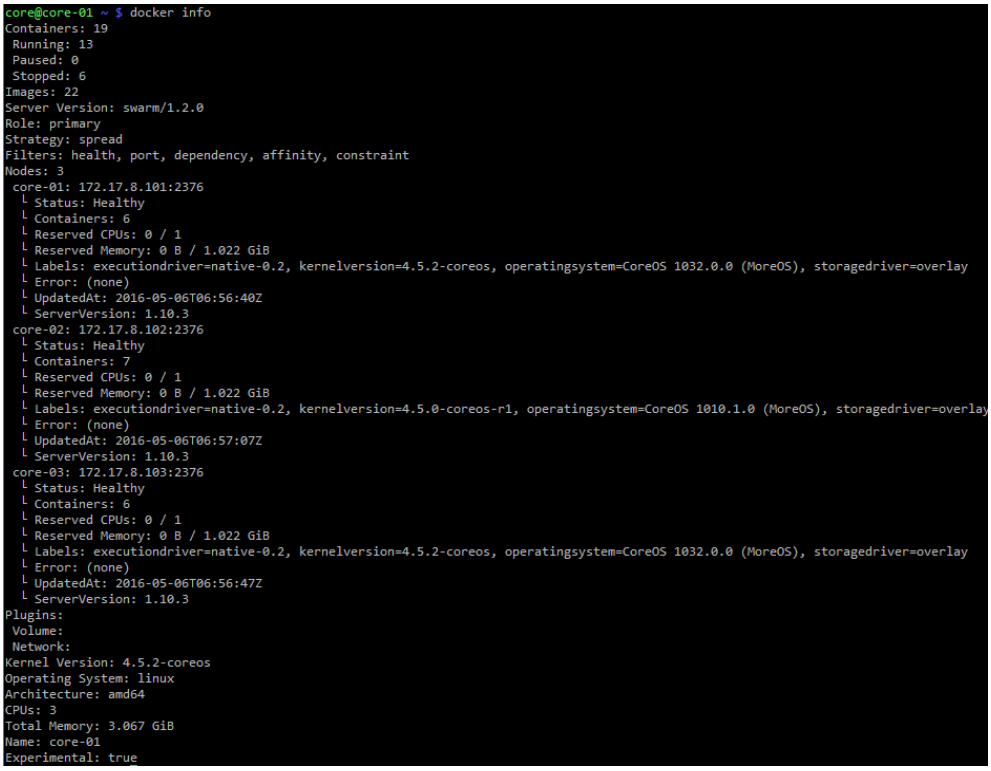
## 6.1 Docker Swarm

### 6.1.1 Hallinta

Jotta Docker Swarm- klusteria päästiin hallinnoimaan, piti aluksi kolmella “export”-komennolla ilmoittaa primaarin master-kontein IP-osoite ja TLS sertifikaatti-, sekä avaintietojen sijainnit. Jos seuraavia komentoja ei asetettu, Swarmia ei pystynyt käyttämään, koska klusteriin on konfiguriotu TLS-salaus käyttöön.

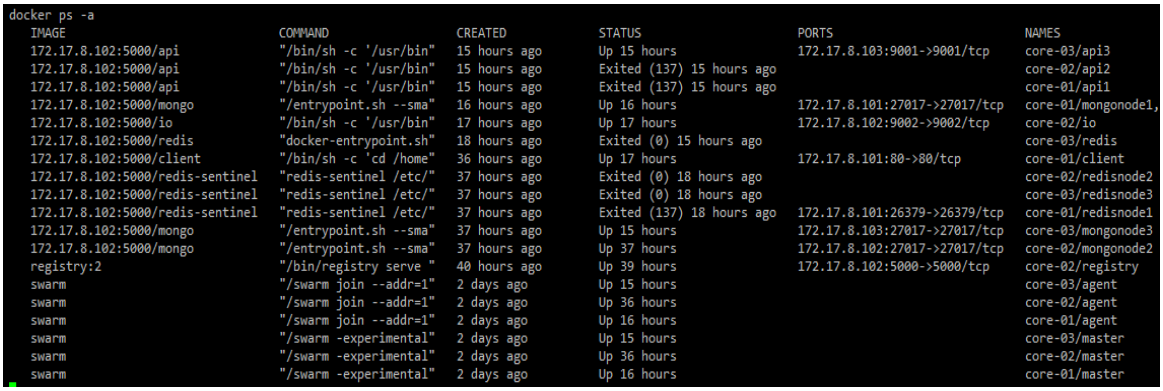
```
export DOCKER_HOST=172.17.8.101:4000
export DOCKER_TLS_VERIFY=1
export DOCKER_CERT_PATH=/home/core/.certs
```

Kun nämä komennot syötettiin, voitiin ”docker info”-komennon avulla saada tietoa klusterin laitteista. Tästä infosivusta voitiin nähdä esimerkiksi kaikki klusterissa olevat laitteet ja niissä sijaitsevien konttien määrä. Tämä komennon syöte nähdä kuviosta 9.




Kuvio 9. ”docker info”-komento

Kaikkien klusterissa olevien konttien tiedot pystyttiin näkemään syöttämällä komento ”docker ps -a”, jonka syöte voidaan nähdä kuviosta 10.




Kuvio 10. “Docker ps -a”-komento

### 6.1.2 Konttien luonti klusterin kautta

Jos kontteja luodaan Swarmin kautta, ne jaetaan automaattisesti eri laitteille. Tämän toiminnon testaukseen luotiin kuusi hello-world-konttia, jotta nähtiin mihin klusterin osiin ne menivät. Luodut kontit nähdään kuviossa 11.

```
core@core-01 ~ $ docker ps -n 6
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
f52a661e07dd        hello-world         "/hello"            19 seconds ago      Exited (0) 18 seconds ago                       core-03/test6
e0ca7a61cbb6        hello-world         "/hello"            23 seconds ago      Exited (0) 22 seconds ago                       core-01/test5
8a79a8285fc4        hello-world         "/hello"            27 seconds ago      Exited (0) 26 seconds ago                       core-02/test4
d1ace3c44184        hello-world         "/hello"            2 minutes ago       Exited (0) 2 minutes ago                        core-03/test3
fe62a09d02a5        hello-world         "/hello"            2 minutes ago       Exited (0) 2 minutes ago                        core-01/test2
0a8416498507        hello-world         "/hello"            2 minutes ago       Exited (0) 2 minutes ago                        core-03/test1
```

Kuvio 11. ”Docker ps -n 6”-komento

On olemassa kolme strategiaa, joidenka avulla kontteja sijoitetaan ympäri klusteria: spread, binpack ja random. Kuviosta 9 nähdään, että klusterissa käytetiin ”spread”-menetelmää. Spread jakaa uuden kontin solmuun, jossa on vähiten kontteja. BinPack asettaa kontin solmuun missä on eniten CPU- ja muistiresursseja käytössä. Random laittaa nimensäkin mukaa kontit satunnaiseen solmuun. (Docker Swarm strategies n.d.)

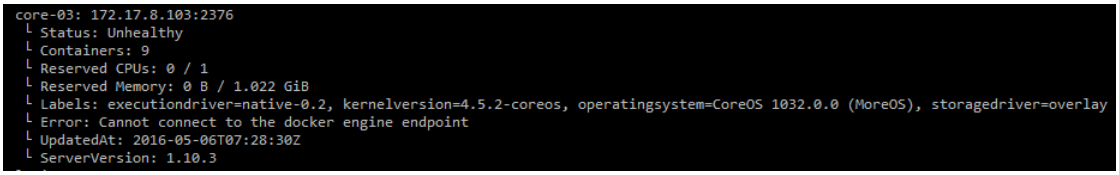
### 6.1.3 Viansieto

Swarm klusteri pystyy siirtämään kontteja toiseen solmuun, jos alkuperäiseen laitteeseen katkeaa yhteys. Tämän ominaisuuden testaukseen luotiin kolme hello-world-konttia Core-03-koneeseen, jotka nähdään kuviossa 12.

```
core@core-01 ~ $ docker ps -n 3
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
ada89cddf4ab        hello-world         "/hello"            12 seconds ago      Exited (0) 11 seconds ago                       core-03/test3
8ff854573e10        hello-world         "/hello"            15 seconds ago      Exited (0) 15 seconds ago                       core-03/test2
1cb62c269317        hello-world         "/hello"            19 seconds ago      Exited (0) 18 seconds ago                       core-03/test1
```

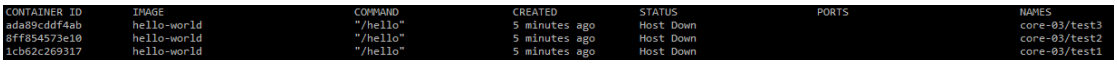
Kuvio 12. ”Docker ps -n 3”- komento

Jos Core-03 meni vikatilaan ”docker info ”-komennossa nähtiin koneen statuksena ”Unhealthy”, kuten kuviossa 13.



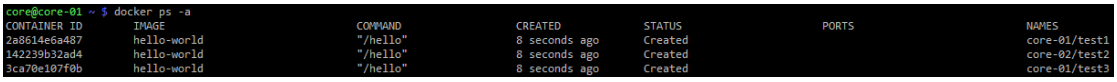

Kuvio 13. Core-03 vikatilassa

Myös ”docker ps”-komennon syötteessä tulee ilmoitus ”Host Down”, jos kone Core03-kone meni vikatilaan, jossa hello-world-kontit sijaitsivat. Tämä ilmoitus nähdään kuviossa 14.



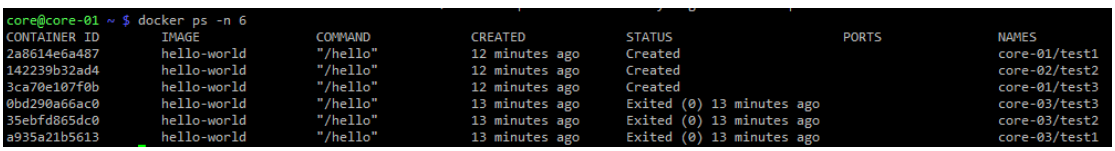

Kuvio 14. Core-03-laitteeseen ei yhteyttä

Tämän jälkeen kontit siirretään uuteen solmuun, kuten kuvion 15 kaappauksessa oli tapahtunut.




Kuvio 15. Core-03-kontit luotu toiseen koneeseen

Tämä on vielä kokeellinen ominaisuus ja ongelmana oli, kun Core-03-kone palasi verkkoon, oli olemassa kaksi kappaletta samaa konttia eri koneissa. Käytännössä Swarm kopio kontit uuteen koneeseen, mutta se ei osaa poistaa niitä siinä tapauksessa, kun alkuperäinen kone palaa klusteriin. Nämä uudelleen luodut kontit oli käytävä käsin poistamassa. Tämä heikkous on todennettu kuviossa 16.




Kuvio 16. Tilanne kun Core-03 palaa verkkoon

Docker Swarm master-kontit luotiin HA-menetelmällä, joten se kesti ensisijaisen master-kontin kaatumisen. Vikatilanteen sattuessa uusi primääri master valitaan pystyssä olevista koneista. Alla nähdään kaksi ilmoitusta master-kontin lokista, joissa voidaan havainnoida prosessi, kun primääri master kaatuu ja uusi johtaja valitaan klusteriin.

*time="2016-05-06T07:50:41Z" level=error msg="Flagging engine as unhealthy. Connect failed 3 times"*

*time="2016-05-06T07:50:48Z" level=info msg="Leader Election: Cluster leadership acquired"*

Kun uusi johtaja oli valittu, piti muistaa vaihtaa ”export”-komentoon uuden johtajan IP-osoite, jotta voitiin syöttää komentoja uudelleen klusteriin.

### 6.1.4 Docker -konttien toiminnan seuraus

Dockerissa on sisäänrakennettuja komentoja, joilla on helppo tarkkailla konttien toimintaa ja resurssien käyttöä.

”Docker exec”-komennolla pääsee suoraan kontin sisään ja antamaan sille komentoja. Tämä on käytännössä samanlailla kuin menisi SSH-komennolla jonkun virtuaalikoneen sisälle.

*docker exec -it <kontti> /bin/bash*

Komennolla ”docker stats” näkee reaaliajassa kaikkien konttien käyttämät järjestelmäresurssit. Jos ilmoitetaan tietyn kontin nimi, niin vain sen tietoja seurataan.

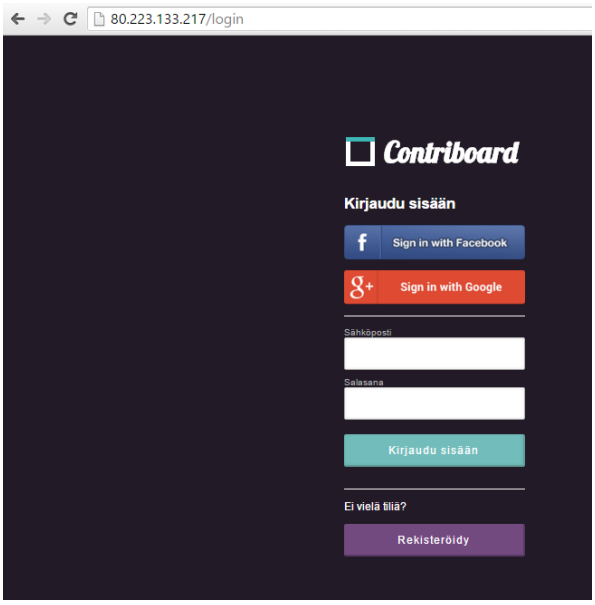
*docker stats <kontti>*

Komento ”docker logs” tulostaa kontin lokitiedot jos ne on tarjolla. Lokia voi myös seurata reaaliajassa jos komentoon lisätään vipu ”-f ” eli follow.

*docker logs <kontti>*

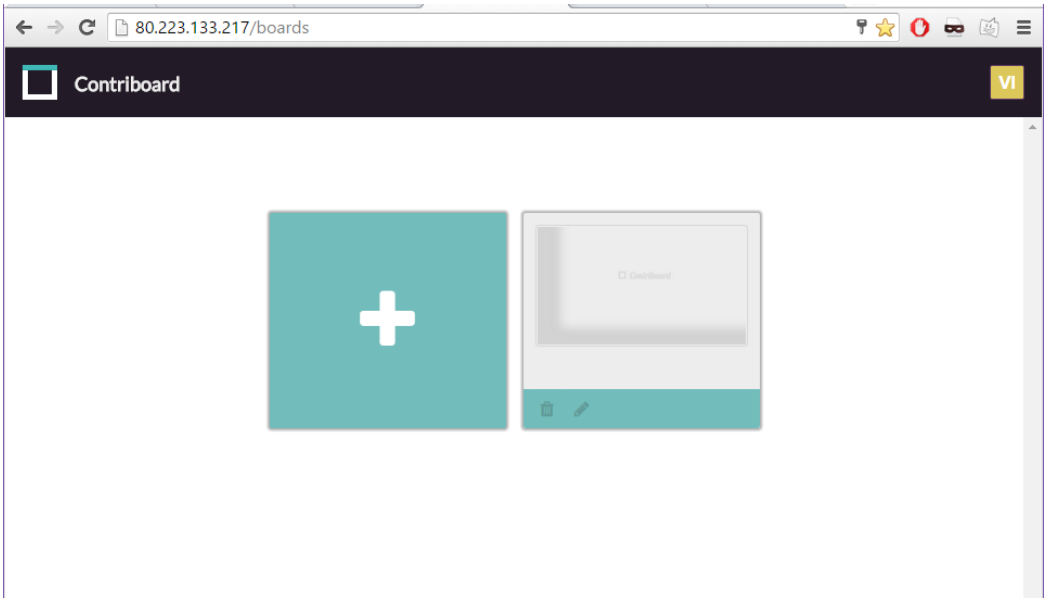
## 6.2 Contriboard –verkkosovelluksen käyttö

Kun testiympäristö oli saatu pystyyn, voitiin Contriboard-verkkosovellukseen mennä selaimen kautta käyttäen omaa ulkoverkon IP-osoitetta. Jos kaikki oli ympäristön puolella kunnossa, voitiin sivustolle rekisteröityä käyttämällä ”Rekisteröidy”-painiketta, joka nähdään kuviossa 17. Sähköpostin ja salasana syöttämisen jälkeen, voitiin kirjautua sisään palveluun.



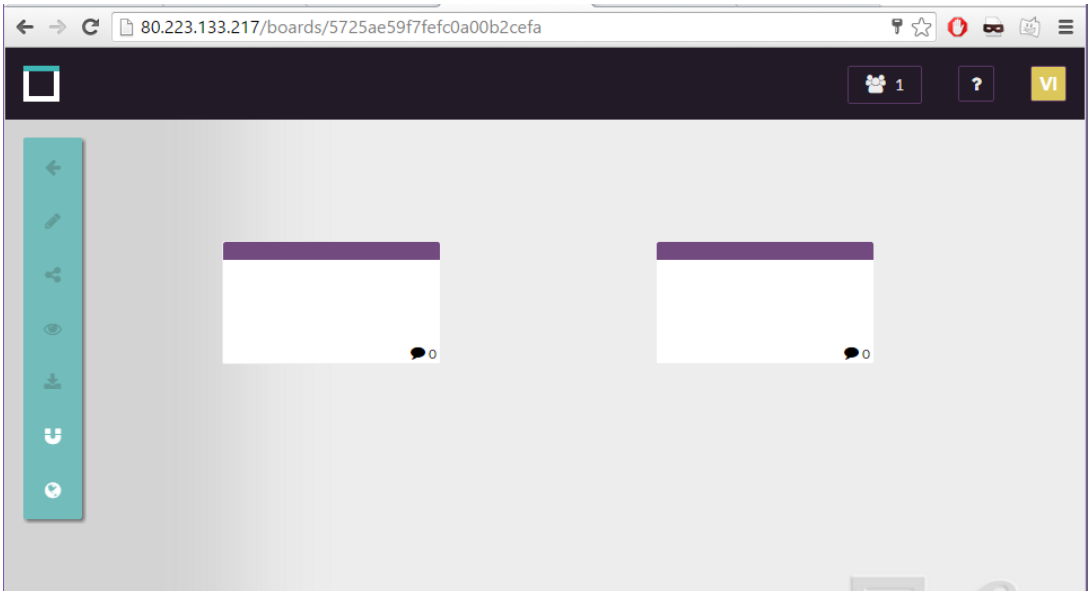

Kuvio 17. Contriboardiin rekisteröityminen

Uuden tikettilaudan sai luotua painamalla plusmerkiltä näyttävää kuvaketta, joka nähdään kuviosta 18. Lautoja on mahdollista luomaan useita ja niiden pohjien taustakuvia muuttamaan.




Kuvio 18. Laudan luonti

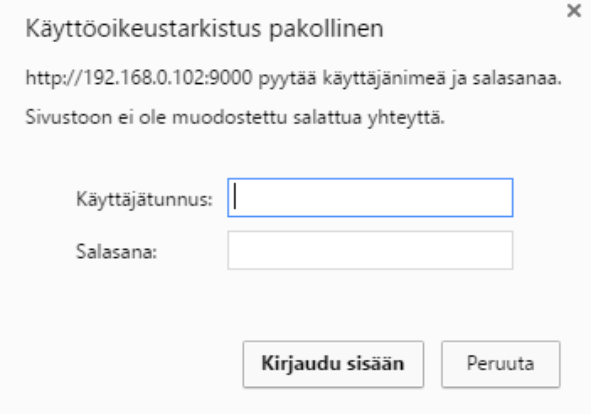
Tehtyyn lautaan pystyi luomaan paperilappuja vastaavia tikettejä ja kirjoittamaan niihin kommentteja. Jos verkkosovelluksessa olisi muita käyttäjiä voisi luotuja tauluja jakaa muiden kanssa. Tämä näkymä voidaan nähdä kuviosta 19.




Kuvio 19. Lappujen luonti

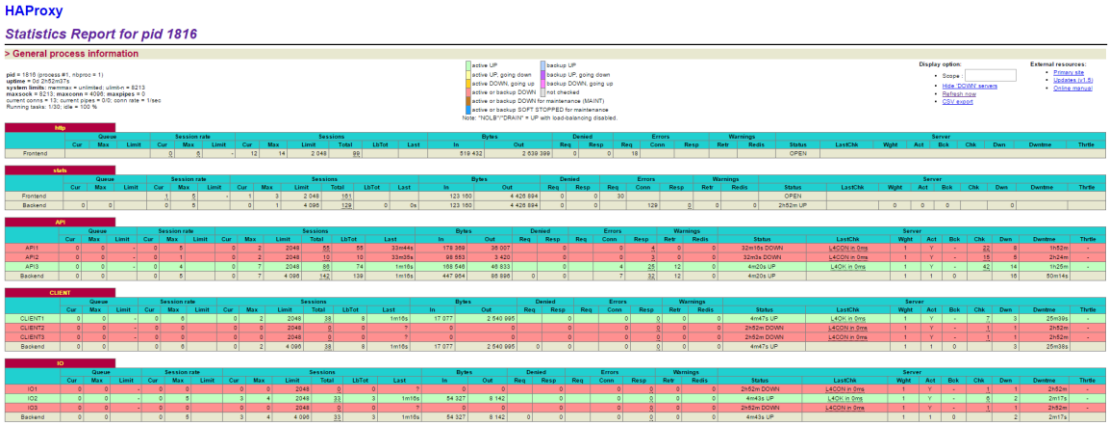
## 6.3 HAproxy –tilastosivusto

HAproxy konfiguraatiotiedostoon asetettiin ”Stats”-parametrit, joten tilastosivua pystyi tarkkailemaan Internet-selaimesta käsin. Sivustoon pääsi syöttämällä selaimeen osoitteen, jossa HAproxy sijaitsee. Tässä tapauksessa osoite oli http://192.168.0.102:9000/haproxy_stats. Sivusto kysyi tunnusta ja salasanaa, koska konfiguraatiotiedostoon oli määritelty kohta ” stats auth ”. Tähän kyselyyn syötettiin tunnus Admin ja salasana Sudo66. Kyselyruutu on taltioitu kuvioon 20.




Kuvio 20. Kirjautuminen tilastosivulle

Tilastosivulta voitiin tarkkailla jokaista backend-palvelinta, jos ne oli määritelty oikein. Jos backend yhteydessä HAproxy-palvelimeen on ne vihreällä pohjalla, mutta vikatilanteen sattuessa ne merkitään punaisella värillä. Kuviossa 21 nähdään Locustin tilastosivun ulkoasu.




Kuvio 21. Haproxy tilastosivu

## 6.4 Locust –työkalun asennus

Muodostettua testiympäristöä oli rasitettava liikenteengeneraattorilla, jotta saatiin selville, että verkkosovellus kestää realistista käyttäjämäärää. Tähän tarkoitukseen käytettiin Locust-ohjelmaa, joka asennettiin yhteen Ubuntu-virtuaalikoneeseen. Tämä laite sijoitettiin CoreOS-klusterin ulkopuolelle fyysisentietokoneen lähiverkkoon 192.168.0.0/24. Locust piti asentaa käyttäen ”pip”-komentoa, jonka avulla voidaan ladata Python-ohjelmia.

Seuraavilla komennoilla asennettiin tarvittavat tiedostot pip- ja Locust –ohjelmia varten.

*sudo apt-get install python-pip python-dev build-essential*

*sudo pip install --upgrade pip*

*pip install locustio*

Kun Locust oli saatu ladattua, piti Jyväskylän ammattikorkeakoulun N4S:n Githubista kloonata ”locust-tests-new.py”-pythontiedosto. Ilman tätä tiedostoa liikenteen-generoimista ei pystynyt aloittamaan Locustilla.

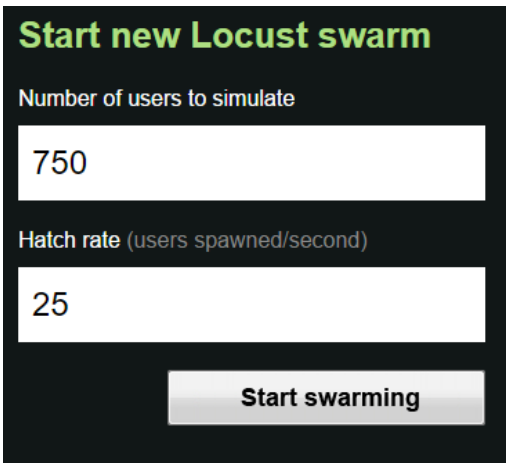
*git clone https://github.com/N4SJAMK/locust-contriboard*

Pythontiedosto kloonauksen jälkeen, voitiin Locust-liikennegenerointi laittaa käyntiin komennolla ”locust”. Vivulla ”-f ” määriteltiin pythontiedoston sijainti ja vivulla ”-H ” testattavan verkkosovelluksen IP-osoite, johon liikennettä tuotettiin.

*locust -f /home/core/locust-contriboard/locust-tests-new.py -H http://80.223.133.217/ TeamboardUser*

## 6.5 Liiketeen generoiminen klusteriin

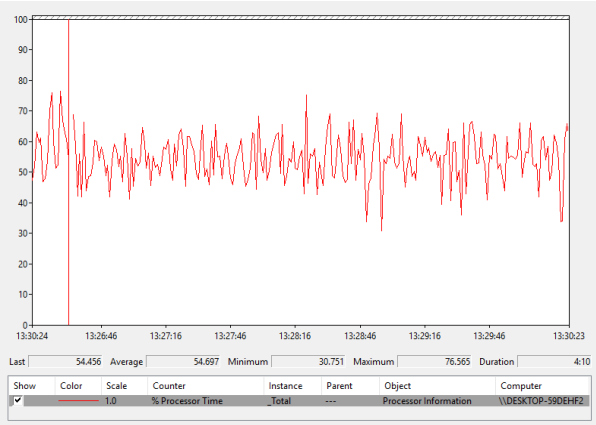
Locustin asennuksen jälkeen sitä voitiin käyttää Internet-selaimen kautta. Tämän ympäristön tapauksessa Locust-hallintasivulle päästiin osoitteella http://192.168.0.101:8089/. Ennen liikenteen tuottamista oli ilmoitettava simulaitavien käyttäjien määrä ja kuinka nopeasti niitä luotiin. Tämän jälkeen liikenteen generoiminen aloitetettiin painamalla ”Start swarming”-painiketta. Tämä aloitusikkuna nähdään kuviossa 22.




Kuvio 22. Käyttäjien määrän asettamien Locust-hallintasivun kautta

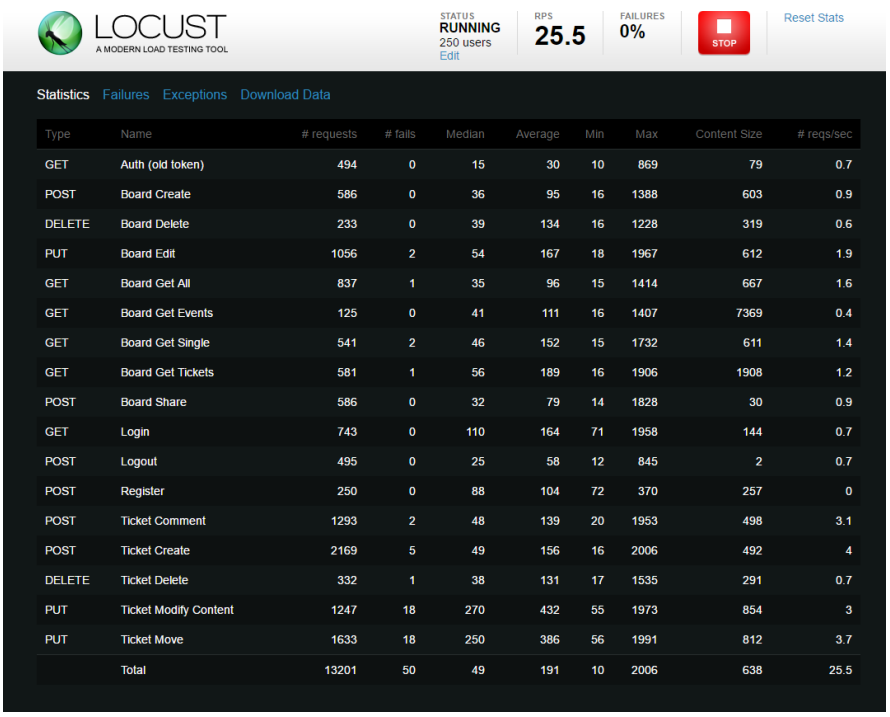
Locust-työkalun avulla rasitettiin ympäristöä ja tarkkailtiin kuinka monta käyttäjää pystyy käyttämään verkkosovellusta ilman ongelmia. Järjestelmää rasitettiin kolmella eri käyttäjä määrällä: 250, 500 ja 750. Jokaisesta kerrasta taltioitiin Locust-tilastot ja fyysisen koneen suoritteimenkäyttö. Suorittimen seuranta koostui 250 sekunnin jaksosta heti kun täysi käyttäjämäärä saatiin saavutettua.

250 käyttäjällä järjestelmä toimi odotetusti ja pakettientippumisia ei tapahtunut. Fyysisen koneen CPU-kuormitus liikkui 50-60% välillä, joka nähdään kuviosta 23.




Kuvio 23. 250 käyttäjän suorittimenkäyttö

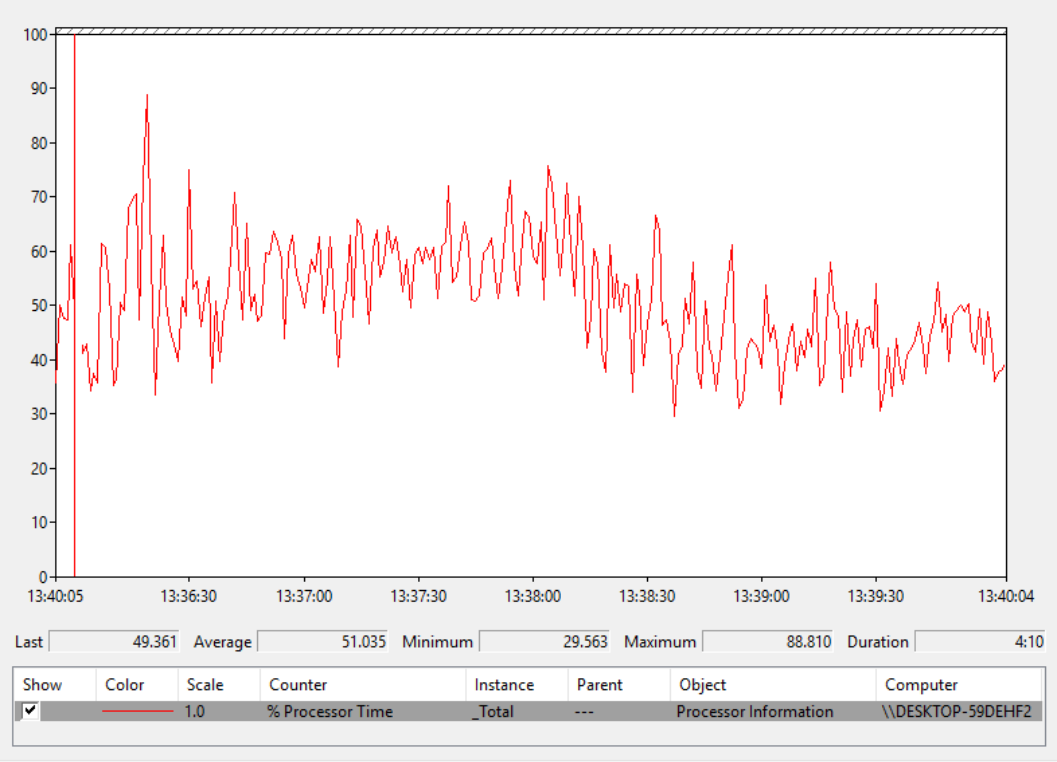
Komentoja tuli keskimäärin 20-30 sekunnissa, mutta virhetilanteita ei syntynyt. 250 käyttäjän tilastot nähdään kuviossa 24. Tällä käyttäjämäärällä Contriboardin käyttö oli sujuvaa ja komentoja syöttäessä viivettä ei havaittu.




| Type | Name | # requests | # fails | Median | Average | Min | Max | Content Size | # reqs/sec |
|---|---|---|---|---|---|---|---|---|---|
| GET | Auth (old token) | 494 | 0 | 15 | 30 | 10 | 869 | 79 | 0.7 |
| POST | Board Create | 586 | 0 | 36 | 95 | 16 | 1388 | 603 | 0.9 |
| DELETE | Board Delete | 233 | 0 | 39 | 134 | 16 | 1228 | 319 | 0.6 |
| PUT | Board Edit | 1056 | 2 | 54 | 167 | 18 | 1967 | 612 | 1.9 |
| GET | Board Get All | 837 | 1 | 35 | 96 | 15 | 1414 | 667 | 1.6 |
| GET | Board Get Events | 125 | 0 | 41 | 111 | 16 | 1407 | 7369 | 0.4 |
| GET | Board Get Single | 541 | 2 | 46 | 152 | 15 | 1732 | 611 | 1.4 |
| GET | Board Get Tickets | 581 | 1 | 56 | 189 | 16 | 1906 | 1908 | 1.2 |
| POST | Board Share | 586 | 0 | 32 | 79 | 14 | 1828 | 30 | 0.9 |
| GET | Login | 743 | 0 | 110 | 164 | 71 | 1958 | 144 | 0.7 |
| POST | Logout | 495 | 0 | 25 | 58 | 12 | 845 | 2 | 0.7 |
| POST | Register | 250 | 0 | 88 | 104 | 72 | 370 | 257 | 0 |
| POST | Ticket Comment | 1293 | 2 | 48 | 139 | 20 | 1953 | 498 | 3.1 |
| POST | Ticket Create | 2169 | 5 | 49 | 156 | 16 | 2006 | 492 | 4 |
| DELETE | Ticket Delete | 332 | 1 | 38 | 131 | 17 | 1535 | 291 | 0.7 |
| PUT | Ticket Modify Content | 1247 | 18 | 270 | 432 | 55 | 1973 | 854 | 3 |
| PUT | Ticket Move | 1633 | 18 | 250 | 386 | 56 | 1991 | 812 | 3.7 |
| | Total | 13201 | 50 | 49 | 191 | 10 | 2006 | 638 | 25.5 |

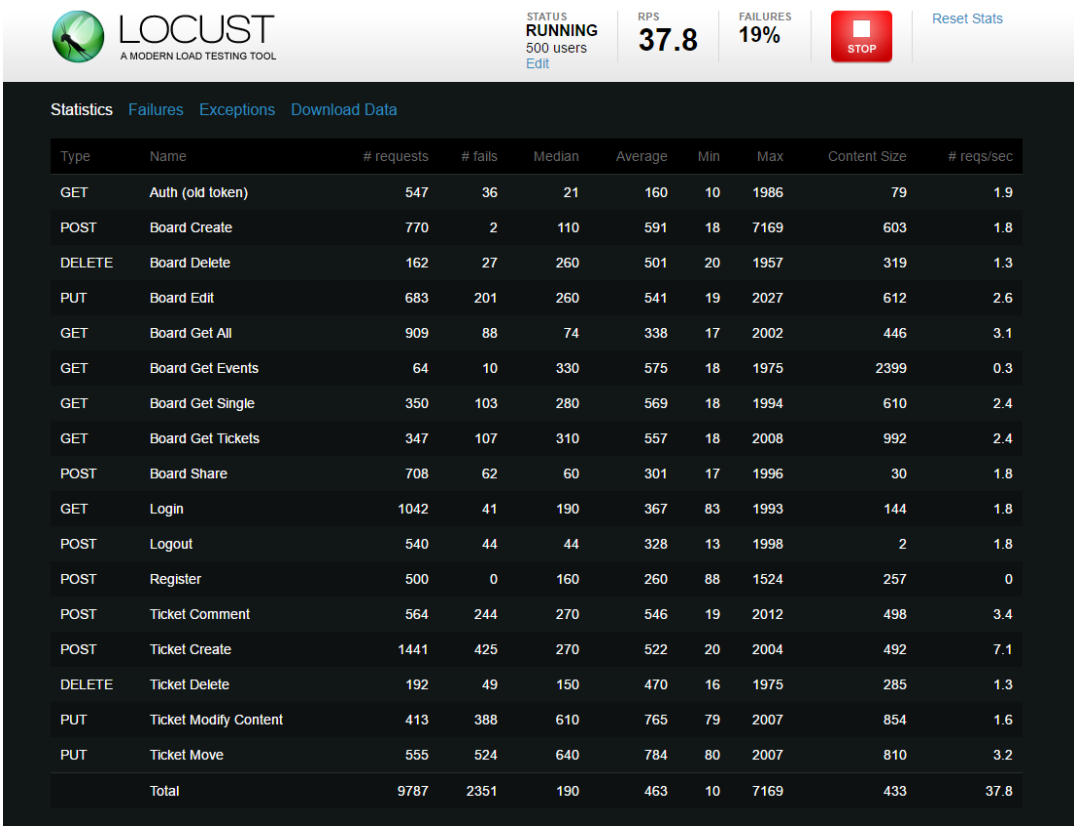
Kuvio 24. 250 Locust-tilastot

Viidelläsadalla käyttäjällä suorittimenkäyttö alkoi kasvaa verrattuna edelliseen mittaukseen, joka voidaan nähdä kuviosta 25. Keskimäärin käyttö oli edelleen noin 50% kohdilla, mutta mittauksen puolessa välissä alkoi paketteja tippua, koska järjestelmä ei pystynyt käsittelemään kaikkia sisään tulevia paketteja.




Kuvio 25. 500 käyttäjän suorittimenkäyttö

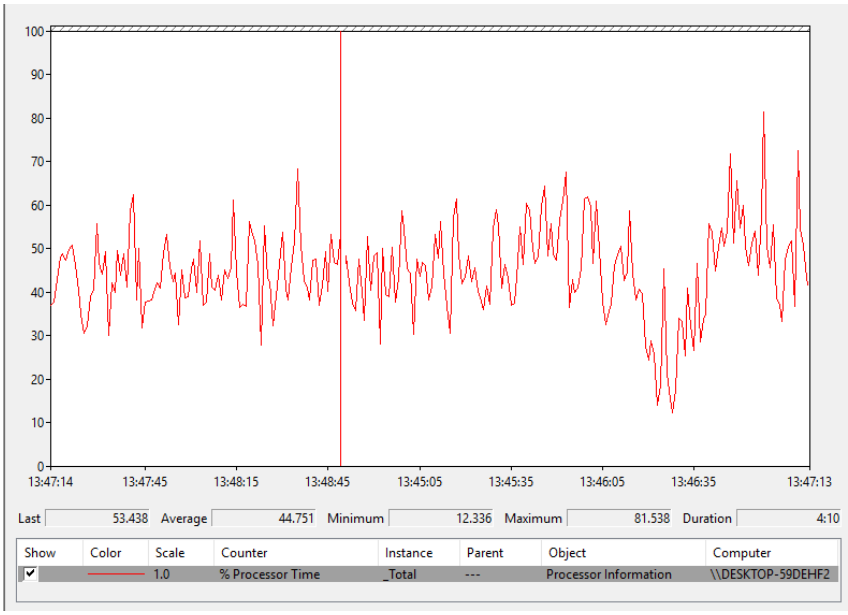
RPS liikkui 30-50 välillä kun virhetilanteita ei tullut, mutta pakettien tippuessa, myös RPS arvo laski. Virheiden määrä tässä mittauksessa oli noin 20%. CPU-käyttö ja Locust-tilastot 500 käyttäjällä nähdään kuviossa 26.




| Type | Name | # requests | # fails | Median | Average | Min | Max | Content Size | # reqs/sec |
|---|---|---|---|---|---|---|---|---|---|
| GET | Auth (old token) | 547 | 36 | 21 | 160 | 10 | 1986 | 79 | 1.9 |
| POST | Board Create | 770 | 2 | 110 | 591 | 18 | 7169 | 603 | 1.8 |
| DELETE | Board Delete | 162 | 27 | 260 | 501 | 20 | 1957 | 319 | 1.3 |
| PUT | Board Edit | 683 | 201 | 260 | 541 | 19 | 2027 | 612 | 2.6 |
| GET | Board Get All | 909 | 88 | 74 | 338 | 17 | 2002 | 446 | 3.1 |
| GET | Board Get Events | 64 | 10 | 330 | 575 | 18 | 1975 | 2399 | 0.3 |
| GET | Board Get Single | 350 | 103 | 280 | 569 | 18 | 1994 | 610 | 2.4 |
| GET | Board Get Tickets | 347 | 107 | 310 | 557 | 18 | 2008 | 992 | 2.4 |
| POST | Board Share | 708 | 62 | 60 | 301 | 17 | 1996 | 30 | 1.8 |
| GET | Login | 1042 | 41 | 190 | 367 | 83 | 1993 | 144 | 1.8 |
| POST | Logout | 540 | 44 | 44 | 328 | 13 | 1998 | 2 | 1.8 |
| POST | Register | 500 | 0 | 160 | 260 | 88 | 1524 | 257 | 0 |
| POST | Ticket Comment | 564 | 244 | 270 | 546 | 19 | 2012 | 498 | 3.4 |
| POST | Ticket Create | 1441 | 425 | 270 | 522 | 20 | 2004 | 492 | 7.1 |
| DELETE | Ticket Delete | 192 | 49 | 150 | 470 | 16 | 1975 | 285 | 1.3 |
| PUT | Ticket Modify Content | 413 | 388 | 610 | 765 | 79 | 2007 | 854 | 1.6 |
| PUT | Ticket Move | 555 | 524 | 640 | 784 | 80 | 2007 | 810 | 3.2 |
| | Total | 9787 | 2351 | 190 | 463 | 10 | 7169 | 433 | 37.8 |

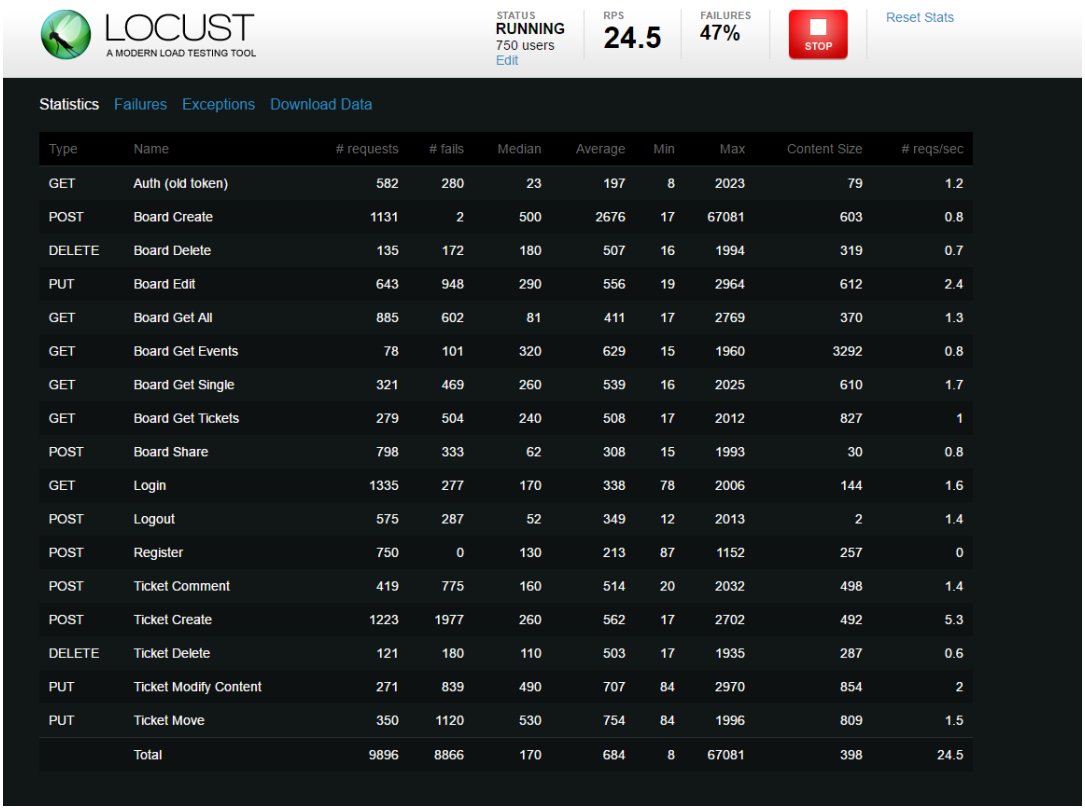
Kuvio 26. 500 Locust-tilastot

Viimeisessä mittauksessa tuotettiin 750 käyttäjää. Tätä mittausta järjestelmä ei enää pystynyt hallitsemaan tehokkaasti ja virheiden määrä oli jo 50 prosentissa. CPU-kuorma liikkui samoissa lukemissa kuin edellisissä mittauksissa ja jossain kohdissa se tippui jopa alemmalle tasolle. RPS arvo ei kasvanut edellisiin mittauksiin verrattuna. Tässä vaiheessa Contriboardia oli lähes mahdoton käyttää. Komentojen syötössä oli monen sekunnin viive tai ne hylättiin täysin. Tämän mittauksen suorittimenkäyttötulokset nähdään kuviosta 27.




Kuvio 27. 750 käyttäjän suorittimenkäyttö

Kuviosta 28 nähdään 750 käyttäjän mittauksen Locust-tilastot.




| Type | Name | # requests | # fails | Median | Average | Min | Max | Content Size | # reqs/sec |
|---|---|---|---|---|---|---|---|---|---|
| GET | Auth (old token) | 582 | 280 | 23 | 197 | 8 | 2023 | 79 | 1.2 |
| POST | Board Create | 1131 | 2 | 500 | 2676 | 17 | 67081 | 603 | 0.8 |
| DELETE | Board Delete | 135 | 172 | 180 | 507 | 16 | 1994 | 319 | 0.7 |
| PUT | Board Edit | 643 | 948 | 290 | 556 | 19 | 2964 | 612 | 2.4 |
| GET | Board Get All | 885 | 602 | 81 | 411 | 17 | 2769 | 370 | 1.3 |
| GET | Board Get Events | 78 | 101 | 320 | 629 | 15 | 1960 | 3292 | 0.8 |
| GET | Board Get Single | 321 | 469 | 260 | 539 | 16 | 2025 | 610 | 1.7 |
| GET | Board Get Tickets | 279 | 504 | 240 | 508 | 17 | 2012 | 827 | 1 |
| POST | Board Share | 798 | 333 | 62 | 308 | 15 | 1993 | 30 | 0.8 |
| GET | Login | 1335 | 277 | 170 | 338 | 78 | 2006 | 144 | 1.6 |
| POST | Logout | 575 | 287 | 52 | 349 | 12 | 2013 | 2 | 1.4 |
| POST | Register | 750 | 0 | 130 | 213 | 87 | 1152 | 257 | 0 |
| POST | Ticket Comment | 419 | 775 | 160 | 514 | 20 | 2032 | 498 | 1.4 |
| POST | Ticket Create | 1223 | 1977 | 260 | 562 | 17 | 2702 | 492 | 5.3 |
| DELETE | Ticket Delete | 121 | 180 | 110 | 503 | 17 | 1935 | 287 | 0.6 |
| PUT | Ticket Modify Content | 271 | 839 | 490 | 707 | 84 | 2970 | 854 | 2 |
| PUT | Ticket Move | 350 | 1120 | 530 | 754 | 84 | 1996 | 809 | 1.5 |
| | Total | 9896 | 8866 | 170 | 684 | 8 | 67081 | 398 | 24.5 |

Kuvio 28. 750 Locust-tilastot

Testiympäristö kesti hyvin 250-500 käyttäjämäärän, mutta jos liikenne kasvoi vielä tuosta, ongelmia alkoi esiintyä ja virheiden määrä kasvamaan. Tämän ympäristön tapauksessa nämä tulokset olivat riittäviä ja jos haluttaisiin käsitellä enemmän käyttäjiä, tulisi ympäristön siirtää pilvipalveluun tai konesaliin.

Ympäristö kuitenkin kesti liikenteen tuottamisen sinne, eikä tullut mitään muuta laajempaa virhetilaa, joka olisi täysin estänyt verkkosovelluksen käytön.

# 7 Pohdinta

## 7.1 Tulokset

Opinnäytetyön päätavoitteena oli tutkia Dockeria, joka osoittautui hyväksi työkaluksi verkkosovelluksen käyttöönotossa ja ylläpidossa. Docker oli ainut työkalu testiympäristössä, jonka kanssa ei joutunut tappelemaan ja se toimi odotetulla tavalla. Se myös ilmoitti hyvin, jos komentoa ei pystynyt suorittamaan ja syyn virhetilaan. Konttien lokit ja tilastot sai näkyviin kätevästi sisäänrakennetuilla komennoilla. Dockerin käyttö oli sujuvaa, koska konttien käyttöönotto, pysäyttäminen, käynnistäminen ja poistaminen tapahtui lähes viiveettä.

Docker Swarm oli sen sijaan pettymys, eikä tätä versiota voi vielä suositella tuotantokäyttöön. Klusterin luominen ja hallinta onnistuivat hyvin, mutta paljoa muita laajoja ominaisuuksia ei löydy. Viansieto olisi toteutettava jollain muulla systeemillä, kuten esimerkiksi CoreOS-käyttöjärjestelmän Fleetillä. Dockerin blogissa hehkutetaan Swarmin nopeudesta ja skaalaus mahdollisuuksista (Coleman 2016). Tämä on varmasti totta, mutta se silti ei poista pettymyksen tunnetta. Olisi ollut mielenkiintoista vertailla Fleet-ympäristöä Swarmiin, mutta tähän ei ollut tarpeeksi aikaa, että olisi kerennyt tekemään vielä toisen ympäristön nykyisen rinnalle.

CoreOS oli hyvä valinta käyttöjärjestelmäksi tukemaan Dockeria. Virtuaalikoneiden käyttöönoton jälkeen ei tarvinnut asentaa tai päivittää mitään Dockeriin liittyvää ja

pystyi suoraan alkaa luoda kontteja. CoreOS oli myös kevyt, eikä se vienyt turhaan resursseja. Tämä oli todella tärkeää, koska ympäristö ajettiin normaalissa tietokoneessa, jossa on hyvin rajalliset resurssit käytettävissä.

Vagrant oli hyödyllinen työkalu virtuaalikoneiden käyttöönotossa. Opinnäytetyön aikana joutui monta kertaa tuhoamaan ja luomaan virtuaalikoneita. Vagrant säästi huomattavasti aikaa, kun ei tarvinnut konfiguroida jokaista uutta konetta erikseen. Oli huoletonta aloittaa koko klusteri tyhjältä pöydältä. Eikä tarvinnut stressata, vaikka olisikin saanut pilattua koko järjestelmän huonolla konfiguraatiolla.

Tavoitteena oli käyttää ilmaiseksi saatavilla olevia järjestelmiä ja ohjelmia. Tähän tavoitteeseen päästiin ja kuka tahansa pystyy luomaan tämän ympäristön omalle koneelle käyttämättä penniäkään rahaa.

## 7.2 Työn prosessi

Opinnäytetyö muuttui prosessin aikana monta kertaa. Heti alussa oli ideana tehdä testiympäristö omalle koneelle liittyen virtualisointiin tai pilvipalveluihin. Lopullinen aihe alkoi muodostua, kun toimeksiantaja ehdotti, että opinnäytetyössä tutkittaisiin Dockeria. Tästä lähdettiin etenemään, mutta pelkästään Dockerista ei olisi saanut paljoa materiaalia kasattua. Tämän tueksi alettiin tutkia, miten voidaan luoda Dockerista klusteri ja skaalata tarjottavaa palvelua.

Dockerin rinnalle lisättiin CoreOS, jota muissa opinnäytteisissä ei ollut käsitelty käytännössä yhtään. Ennen kun lähdettiin työstämään omaa dokumentointia, tuli etsittyä muita samaan aihepiireihin liittyviä opinnäytetöitä ja tarkkaitua ettei aiheet turhan paljon mene päällekkäin. Dockerista on luotu jo monia muitakin opinnäytetöitä, muuta klusterointi, Docker Swarm ja CoreOS antoi uutta näkökulma tähän tuotoksen.

Alussa oli tarkoitus hoitaa koko klusterointi CoreOS työkaluilla fleet, flannel ja etcd. Tähän ei kuitenkaan päädytty, koska haluttiin pitää paremmin toimeksiantajan näkökulma opinnäytetyöhön. Tästä syystä klusterointi hoidettiin Docker Swarmin avulla, joka on Dockerin natiivi klusterointijärjestelmä. Fleetin avulla olisi saanut paremmin hoidettua klusterointi, koska se on paremmin varustettu kuin Swarm tällä hetkellä.

CoreOS oli kuitenkin hyvä pohja Dockerille, koska Docker oli jo valmiiksi asennettu siihen. Myös Etcd löytyi sisäänrakennettuna käyttöjärjestelmästä.

Aluksi oli tarkoitus tehdä itse todella yksinkertainen verkkosivu, jonka avulla olisi esitetty klusterin toimintakyky. Koulun ohjaajalta tuli ehdotus, että oman verkkosovelluksen sijasta käytettäisiin JAMK:kin N4S-projektin tekemää Contriboardia. Tämä oli hyödyllinen lisä tähän opinnäytetyöhön. Tämän verkkosovelluksen avulla pääsi tarkemmin sisälle, kuinka palvelua pystyy isännöimään omassa ympäristössä, koska sen toiminta vaati erillisiä kontteja useassa eri koneessa.

## 7.3 Käytetyt lähteet

Opinnäytetyön teoriaosuudessa haluttiin antaa hyvä tietopohja lukijalle tulevaa testiympäristön toteutuskappaleita varten. Lähteinä käytettiin käytännössä täysin englanninkielisiä materiaaleja. Tämä on sen verran tuore aihe, ettei hyvää suomenkielistä kirjallisuutta vielä löydy aiheesta. Dockeriin ja virtualisointiin liittyviin kappaleisiin käytettiin aiheeseen liittyvää kirjallisuutta ja Dockerin omilta sivulta löytyviä dokumentointeja. Parempia ja luotettavampia lähteitä ei näihin aihepiireihin pitäisi löytyä.

Muihin ei niin tärkeisiin aiheisiin, käytettiin lähteitä valmistajien sivulta tai verkkosivuilta kuten DigitalOcean. Tämä johtui myös siitä, ettei ollut saatavilla kirjallisuutta aihepiireihin liittyen. Jos johonkin aiheeseen löytyi hyvä kirja, niin sitä tuli käytettä, mutta jos ei ollut tarjolla niin käytettiin mahdollisimman luotettavaa sivustoa. Jotkut verkkolähteet ja kirjat olivat vuodelta 2014, joten ne saattoivat olla osaksi vanhentuneita, koska Docker oli silloin vasta juuri julkaistu. Teoriaa käsiteltiin mahdollisimman yleisellä tasolla, joten lähteiden vanhentuminen ei pitäisi vaikuttaa liian dramaattisesti.

## 7.4 Ongelmatilanteet

Contriboardin kanssa ilmeni ongelmia, kun alettiin tuottaa liikennettä siihen Locustilla. Mysteerisestä syystä kaikki API-kontit kaatuivat muutaman minuutin jälkeen, kun ne joutuivat rasituksen alle. Mitään virheilmoituksia ei tullut ja oli todella vaikeaa alkaa etsiä syytä ongelmaan. Tämän ongelman ratkaisemiseen meni useita päiviä ja

se viivästytti huomattavasti opinnäytetyön aikataulua. Koko ympäristö tuli tutkittua tarkasti läpi, ja oli tarkoitus siirtää koko ympäristö toiseen virtualisointialustaan. Tämä toimenpide olisi vienyt jo liikaa aikaa.

Lopulta selvisi, että Locustin skriptissä oli vanhentuneita rivejä, jotka hajottivat kontit. Locutin tilastoihin tuli ”Board export as image” kohtaan aina vain yksi pyyntö joka meni ”fails”-osioon. Aina kyseisen export-pyynnön tultua, sitä käsittelevä kontti kaatui. Tämä export- toiminto ei ollut enää tässä Contriboard versiossa käytössä. Kun Locust skriptistä poisti tuohon ominaisuuteen liittyvät rivit, alkoivat API-kontit toimia moitteitta.

Kun ongelmat oli saatu ratkaistua, järjestelmä alkoi kestää hyvin liikennettä. 200-500 yhtäaikaista käyttäjää tällä kokoonpanolla oli hyväksyttävä tulos. Jos liikenteen määrä tulisi kasvamaan tästä tulisi ympäristö siirtää konesaliin tai pilvipalveluun, jossa ei olisi näin rajoittuneet resurssit.

## 7.5 Kehitysideat

Testiympäristö saatiin lähelle tavoiteltua päämäärää. Ainoastaan viansiedossa ja käyttöönoton automatisoinnissa olisi vielä kehitettävää. Tavoitteena oli, että kaikki komponentit pystyisi automaattisesti kestämään vikatilanteen. Eikä palvelun saatavuus kärsisi, vaikka yksi koneita kaatuisi.

Tähän tavoitteeseen ei päästy. Aino tilanne mikä vaatisi ylläpitäjän toimenpiteitä, olisi jos Core01-kone kaatuisi. Tämä johtuu siitä, että sinne oli sijoitettu primääri Mongo-tietokanta ja API-kontit eivät ymmärrä, jos Mongon ensisijainenkone muuttuu. Kaikki tietokannan tiedot säilyvät muissa koneissa, koska tietokanta on replikoitu niihin. Vikatilanteen pystyy korjaamaan vain, jos laittaa Core01-koneen kuntoon tai luo API-kontit uudelleen vaihtaen ympäristömuuttujatiedostoon uuden primäärin Mongon IP-osoite.

Tämän ongelman ratkaisuun tarvittaisiin parempia ohjelmointitaitoja, koska pitäisi luoda skriptin, jossa API-kontille ilmoitetaan automaattisesti uuden primääri Mongo-tietokannan osoite. Tähän omat kyvyt eivät kuitenkaan riittäneet.

Järjestelmä kestää kuitenkin, jos Core02- tai Core03-koneiden kaatumisen. Tämä johtuu siitä, että kaikki Contriboard-komponentit on sijoitettu jokaiselle koneelle ja Client-kontti kopioidaan automaattisesti toiselle koneelle vikatilanteen sattuessa. HAproxy jakaa vain liikenteen uudelle koneelle, jos jompikumpi edellä mainituista koneesta kaatuu.

Toinen kriittinen vikatilanne olisi, jos HAproxy-kone kaatuisi. Tällöin liikenne koko klusteriin katkeaisi. Tähän auttaisi HAproxyn kahdentaminen toiselle koneelle. Tämä ominaisuus ei kuitenkaan suoraan liittynyt Dockeriin tai Swarmiin, joten päädyttiin rajaamaan tämä osuus pois.

Käyttöönoton automatisointia pitäisi vielä hioa, mutta tämä vaatii jo parempaa ohjelmointitaitoja. Nyt luotiin vain kansioita ja kopiotiin pilvestä konfiguraatiotiedostoja. Konfiguraatiopohjat saatiin siirrettyä koneille automaattisesti, mutta niihin piti käsin määritellä eri koneiden tarkat tiedot. Olisi hyvä taito, jos saisi kaikki asetukset kuntoon automatisoinnin avulla. Kun kerran konfiguraatiot saisi kuntoon, ei niitä tarvitsisi koskaan enää tehdä uudestaan, vain päivittää tarpeen mukaan.

Viimeinen hiottava osio olisi saada paremmin fyysisenkoneen resurssit käyttöön. Tällä hetkellä ympäristö pystyi hyödyntämään fyysisen koneen CPU-resursseja vain 50-70%. Tähän auttaisi jakamalla tietokanta palasiin ”sharding”-menetelmällä. Nyt kaikki kuorma tulee yhteen Mongo-konttiin ja muut toimivat vain varmuuskopiona. Myös HAproxyn jonotusasetuksia hiomalla voisi saada paremmin suorituskykyä käyttöön.

## 7.6 Johtopäätökset

Testiympäristö saatiin muodostettua lähelle toivottua päämäärää, mutta se on kuitenkin vielä kaukana siitä, että sitä voisi käyttää oikeasti tuotannossa tai yrityskäytössä. Syyt tähän on omat taidot, järjestelmien puutteelliset ominaisuudet ja ympäristön osittainen keskenräisyys.

Vaikka muodostetua ympäristöä ei voisi käyttää tuotannossa, voidaan koottuja tietoja pitää pohjana ammattimaisemmin tuotettuihin järjestelmiin. Jos joskus tulee tehtyä vastaavia töitä tai omalla ajalla luotua palveluita, tulee varmasti käytettyä Dockeria ja tätä opinnäytetyötä hyväksi.

Docker on elegantti ja helposti omaksutava työkalu palveluden käyttöönottoon ja ylläpitämiseen. Käytännössä mitään pahaa sanottavaa siitä ei löytynyt. Se oli ainoa ympäristössä käytetty komponentti, jonka kanssa ei ollut mitään ongelmia. Jos joku vikatilanne tuli verkkosovelluksessa, niin viimeisenä ajatteli että se voisi johtua Dockerista.

Nykyään on myös olemassa jo monia alustoja kuten Rancher, Marathon ja Kontena, joiden avulla voidaan helposti käyttöönotta skaalatun konttiympäristön tuotantoon. Eli käytännössä toteutetan sama mitä tässä opinnäytetyössä tehtiin, mutta ympäristö luodaan automaattisesti tarjottavan palvelun avulla. Näin ympäristön pystyttämiseen ei tarvita suurta tietotaitoa, vaan kehittäjät voivat keskittyä oman palvelun tuottamiseen ja ylläpitoon.

Docker on ollut vasta kolme vuotta markkinoilla, mutta se on jo kerryttänyt paljon suosiota. Suurimpia yritysasiakkaita Dockerilla on esimerkiksi BBC News, Spotify, ebay, Uber ja New York Times (Our Customers n.d). Dockerin suosio tulee varmasti kasvamaan tulevien vuosien aikana yrityskäytössä, ellei tule vielä jotain mullistavampaa järjestelmää markkinoille. Tästä huolimatta voi kestää kahdesta kolmeen vuoteen ennekuin Docker pääsee laajaan tietoisuuteen ja sen käyttö tuotanto ympäristöissä tulisi lisääntymään.

Dockerin markkinoiden dominoiminen ei kuitenkaan ole ollut viime vuosina täydellistä. Esimerkiksi vuonna 2015 käytiin tiukka taistelu konttihallintajärjestelmien kanssa, kun CoreOS ja Googlen Kubernetes tulivat kilpailemaan Dockerin rinnalle (Miller 2015). Tämä päättyi siihen kun CoreOS:n toimitusjohtaja Alex Polvi ja Dockerin teknologiajohtaja Solomon Hykes sopivat standaroidusta runC-konttijärjestelmästä (Kepes 2015). Tämä tarkoittaa sitä, ettei jokaisella valmistajalla ole täysin omaa koodia, vaan osan siitä jaetaan kaikkien kesken.

Näiden loppukappaleiden perusteella voidaan sanoa seuraavien vuosien olevan mielenkiintoisia Dockerin osalta. Uutisia tulee varmasti seurattua tästä eteenpäin näihin aihepiireihin liittyen. Suurin kysymys onkin tuleeko Docker dominoimaan markkinoita vai syrjäyttääkö sen joku uusi tulokas mullistavalla tekniikalla. Oma veikkaus on että kestää kaksi tai kolme vuotta ennenkun Docker tekee lopullisen läpimurtoinsa laajemille markkinoille. Eikä muu järjestelmä tule kaappaamaan sen

herruutta. Aika näyttää miten tässä tulee käymään ja toivottavasti joku toinen opinnytetyö tulee täydentämään tätä aihepiiriä tulevaisuuden tekniikoilla ja uusilla ratkaisuilla.

# Lähteet

About Vagrant. N.d. Tietoa Vagrantista. Viitattu 30.3.2016. https://www.vagrantup.com/about.html

Anicas, M. 2014. An Introduction to HAProxy and Load Balancing Concepts. DigitalOcean 13.05.2015. Viitattu 27.04.2016. https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts

Architecture. N.d. Tietoa Contriboardin komponenteistä. Viitattu 27.04.2016. https://github.com/N4SJAMK/teamboard-meta/wiki/about-architecture

Braden, R. 1989. Requirements for Internet Hosts -- Communication Layers. Viitattu 27.04.2016. https://tools.ietf.org/html/rfc1122

Bresnahan, C. & Negus, C. 2012. Linux Bible. uud.p. John Wiley & Sons.

Buyya, R., Eskicioglu, R., Graham, P., Pourreza, H., Sommers, F. & Yeo, C. N.d. Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers. Viitattu 16.3.2016. http://www.cloudbus.org/papers/ic_cluster.pdf

Coleman, M. 2016. Docker Swarm Exceeds Kubernetes Performance at Scale. Docker 09.03.2016. Viitattu 17.05.2016. https://blog.docker.com/2016/03/swarmweek-docker-swarm-exceeds-kubernetes-scale/

Contriboard - Digitize ideas together while brainstorming. N.d. Yleistietoa Contriboardista. Viitattu 27.04.2016. http://n4sjamk.github.io/contriboard/

Corbet, J. 2014. Etcd and fleet. LWN.com-sivusto 22.10.2014. Viitattu 30.3.2016. https://lwn.net/Articles/617452/

CoreOS Cluster Discovery. N.d. Tietoa palveluiden löytämisestä etcd:n avulla. Viitattu 30.3.2016. https://coreos.com/os/docs/latest/cluster-discovery.html

CoreOS – a new approach to Linux-based server systems. 2013. CoreOS ominaisuuksia. Viitattu 29.3.2016. http://itnews2day.com/2013/08/22/coreos-linux-based-server-systems/

Dockerfile reference. N.d. Tietoa Dockerfile-tiedostosta. Viitattu 16.3.2016 https://docs.docker.com/engine/reference/builder/

Docker Swarm strategies. N.d. Tietoa miten swarm jakaa kontteja eri klusterin solmuihin. Viitattu 6.5.2016 https://docs.docker.com/swarm/scheduler/strategy/

Docker Swarm. N.d. Yleistietoa Docker Swarmista. Viitattu 16.3.2016. https://www.docker.com/products/docker-swarm

Ellingwood, J. 2014. An Introduction to CoreOS System Components. DigitalOcean 5.9.2014. Viitattu 29.3.2016. https://www.digitalocean.com/community/tutorials/an-introduction-to-coreos-system-components

Getting Started with etcd. N.d. etcd-työkalun määrittely. Viitatttu 30.3.2016. https://coreos.com/etcd/docs/latest/getting-started-with-etcd.html

Gonzalez, J. & Krishnan, S. 2015. Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects. Apress.

Introduction to MongoDB. N.d. MongoDB:n esittely. https://docs.mongodb.org/manual/introduction/

Introduction to Redis. N.d. Rediksen esittely. Viitattu 27.04.2016. http://redis.io/topics/introduction

Kane, S. & Matthias, K. 2015. Docker: Up and Running. O'Reilly Media.

Kepes, B. 2015. The Container Format Wars Are Over--Docker Won. But CoreOS Didn't Necessarily Lose. Forbes 24.06.2015. Viitattu 17.05.2016. http://www.forbes.com/sites/benkepes/2015/06/24/the-container-format-wars-are-over-docker-won-but-coreos-didnt-necessarily-lose/#4021c12459bc

Liikevaihto ja henkilöstö. N.d. Inmicsin liikevaihto ja henkilöstömäärä. Viitattu 19.05.2016. http://www.inmics.fi/yritys/liikevaihto-ja-henkilosto/

Miller, P. 2015. Container Competitors Google, CoreOS, Joyent And Docker Join New Linux Club As Kubernetes Turns One. Forbes 21.07.2015. Viitattu 17.05.2016. http://www.forbes.com/sites/paulmiller/2015/07/21/container-competitors-google-coreos-joyent-and-docker-join-new-linux-club-as-kubernetes-turns-one/#2f95e6b024c2

Mockapetris, P. 1987. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. Viitattu 27.04.2016. https://www.ietf.org/rfc/rfc1035.txt

Our Customers. N.d. Dockerin asiakkaat listattuna. Viitattua 17.5.2016. https://www.docker.com/customers

Overview Swarm with TLS. N.d. TLS-protokollan käyttö Docker Swarmissa. Viitattu 31.3.2016. https://docs.docker.com/swarm/secure-swarm-tls/

Plan for Swarm in production. N.d. Swarmin käyttöönotto tuotantoon. Viitattu 16.3.2016. https://docs.docker.com/swarm/plan-for-production/

Perhe takaa kestävän kasvun. N.d. Yleistietoa Inmicsistä. Viitattu 3.4.2016. http://www.inmics.fi/yritys/perhe-takaa-kestavan-kasvun/

Portnoy, M. 2012. Virtualization Essentials. Sybex.

Postel, J. 1981. INTERNET PROTOCOL. Viitattu 27.04.2016. https://tools.ietf.org/html/rfc791

Postel, J. 1981. TRANSMISSION CONTROL PROTOCOL. Viitattu 27.04.2016. https://www.ietf.org/rfc/rfc793.txt

Smith, R. 2012. Linux Essentials. Sybex

Tekniikka on menestyksen tukipilari. N.d. Yleistietoa Inmicsistä. Viitattu 3.4.2016. http://www.inmics.fi/yritys/

Understand the architecture. N.d. Tietoa Dockerista. Viitattu 16.3.2016. https://docs.docker.com/v1.8/introduction/understanding-docker/

Using CoreOS. N.d. Yleistietoa CoreOS-käyttöjärjestelmästä. Viitattu 29.3.2016. https://coreos.com/using-coreos/

VirtualBox. N.d. Tietoa VirtualBox-alustasta ja sen asentamisesta sekä konfirmoimisesta. Viitattu 30.3.2016. https://help.ubuntu.com/community/VirtualBox

Virtualization. N.d. Perustietoa virtualisoinnista. Viitattu 16.3.2016. https://www.vmware.com/virtualization/how-it-works.html

Vohre, D. 2016. Pro Docker. Apress.

Vugt, S. 2014. Pro Linux High Availability Clustering. Apress.

WHAT IS LOAD BALANCING?. N.d. Tietoa kuormantasauksesta. Viitattu 16.3.2016. https://www.nginx.com/resources/glossary/load-balancing/

What is Locust. N.d. Tietoa Locust- työkalusta. Viitattu 27.04.2016. http://docs.locust.io/en/latest/what-is-locust.html

Why Vagrant. N.d. Tietoa Vagrantista. Viitattu 30.3.2016. https://www.vagrantup.com/docs/why-vagrant/

# Liitteet

## Liite 1. Ubuntu Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Boostrap Script
$script = <<SCRIPT


  sudo add-apt-repository ppa:vbernat/haproxy-1.5
                sudo apt-get update
                sudo apt-get dist-upgrade
                sudo apt-get install haproxy

  sudo cp /vagrant/haproxy.cfg /etc/haproxy/haproxy.cfg
  sudo cp /vagrant/sysctl.conf /etc/sysctl.conf
  sudo cp /vagrant/interfaces /etc/network/interfaces

  sudo iptables -A FORWARD -o eth0 -i eth1 -s 172.17.8.0/24 -m conntrack --ctstate
NEW -j ACCEPT
  sudo iptables -A FORWARD -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
  sudo iptables -t nat -F POSTROUTING
  sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
  iptables-save > /etc/iptables.rules

  sudo /etc/init.d/networking restart
  sudo sysctl -p /etc/sysctl.conf
  sudo service haproxy restart

SCRIPT
```

```
Vagrant.configure(2) do |config|

  config.vm.box = "ubuntu/trusty32"

  config.vm.box_check_update = true

  config.vm.provision "shell", inline: $script

config.vm.network "forwarded_port", guest: 80, host: 8080

config.vm.network "public_network",
  auto_config: false

config.vm.network "private_network", ip: "172.17.8.106",
  auto_config: false

end
```

## Liite 2. CoreOS Vagtantfile

```
# -*- mode: ruby -*-
# # vi: set ft=ruby :

require 'fileutils'

Vagrant.require_version ">= 1.6.0"

CLOUD_CONFIG_PATH = File.join(File.dirname(__FILE__), "user-data")
```

```
CONFIG = File.join(File.dirname(__FILE__), "config.rb")

# Defaults for config options defined in CONFIG
$num_instances = 1
$instance_name_prefix = "core"
$update_channel = "alpha"
$image_version = "current"
$enable_serial_logging = false
$share_home = false
$vm_gui = false
$vm_memory = 1024
$vm_cpus = 1
$shared_folders = {}
$forwarded_ports = {}

# Attempt to apply the deprecated environment variable NUM_INSTANCES to
# $num_instances while allowing config.rb to override it
if ENV["NUM_INSTANCES"].to_i > 0 && ENV["NUM_INSTANCES"]
  $num_instances = ENV["NUM_INSTANCES"].to_i
end

if File.exist?(CONFIG)
  require CONFIG
end

# Use old vb_xxx config variables when set
def vm_gui
  $vb_gui.nil? ? $vm_gui : $vb_gui
end

def vm_memory
  $vb_memory.nil? ? $vm_memory : $vb_memory
```

```
end

def vm_cpus
 $vb_cpus.nil? ? $vm_cpus : $vb_cpus
end

Vagrant.configure("2") do |config|
 # always use Vagrants insecure key
 config.ssh.insert_key = false

 config.vm.box = "coreos-%s" % $update_channel
 if $image_version != "current"
    config.vm.box_version = $image_version
 end
 config.vm.box_url = "https://storage.googleapis.com/%s.release.core-
os.net/amd64-usr/%s/coreos_production_vagrant.json" % [$update_channel, $im-
age_version]

 ["vmware_fusion", "vmware_workstation"].each do |vmware|
  config.vm.provider vmware do |v, override|
    override.vm.box_url = "https://storage.googleapis.com/%s.release.core-
os.net/amd64-usr/%s/coreos_production_vagrant_vmware_fusion.json" % [$up-
date_channel, $image_version]
  end
 end

 config.vm.provider :virtualbox do |v|
  # On VirtualBox, we don't have guest additions or a functional vboxsf
  # in CoreOS, so tell Vagrant that so it can be smarter.
  v.check_guest_additions = false
  v.functional_vboxsf     = false
 end
```

```
# plugin conflict
if Vagrant.has_plugin?("vagrant-vbguest") then
  config.vbguest.auto_update = false
end

(1..$num_instances).each do |i|
  config.vm.define vm_name = "%s-%02d" % [$instance_name_prefix, i] do |config|
    config.vm.hostname = vm_name

    if $enable_serial_logging
      logdir = File.join(File.dirname(__FILE__), "log")
      FileUtils.mkdir_p(logdir)

      serialFile = File.join(logdir, "%s-serial.txt" % vm_name)
      FileUtils.touch(serialFile)

      ["vmware_fusion", "vmware_workstation"].each do |vmware|
        config.vm.provider vmware do |v, override|
          v.vmx["serial0.present"] = "TRUE"
          v.vmx["serial0.fileType"] = "file"
          v.vmx["serial0.fileName"] = serialFile
          v.vmx["serial0.tryNoRxLoss"] = "FALSE"
        end
      end

      config.vm.provider :virtualbox do |vb, override|
        vb.customize ["modifyvm", :id, "--uart1", "0x3F8", "4"]
        vb.customize ["modifyvm", :id, "--uartmode1", serialFile]
      end
    end

    if $expose_docker_tcp
```

```
    config.vm.network "forwarded_port", guest: 2375, host: ($expose_docker_tcp +
i - 1), auto_correct: true
  end

  $forwarded_ports.each do |guest, host|
    config.vm.network "forwarded_port", guest: guest, host: host, auto_correct:
true
  end

  ["vmware_fusion", "vmware_workstation"].each do |vmware|
    config.vm.provider vmware do |v|
      v.gui = vm_gui
      v.vmx['memsize'] = vm_memory
      v.vmx['numvcpus'] = vm_cpus
    end
  end

  config.vm.provider :virtualbox do |vb|
    vb.gui = vm_gui
    vb.memory = vm_memory
    vb.cpus = vm_cpus
  end

  ip = "172.17.8.#{i+100}"
  config.vm.network :private_network, ip: ip

  # Uncomment below to enable NFS for sharing the host machine into the coreos-
vagrant VM.
  #config.vm.synced_folder ".", "/home/core/share", id: "core", :nfs => true,
:mount_options => ['nolock,vers=3,udp']
  $shared_folders.each_with_index do |(host_folder, guest_folder), index|
    config.vm.synced_folder host_folder.to_s, guest_folder.to_s, id: "core-
share%02d" % index, nfs: true, mount_options: ['nolock,vers=3,udp']
```

```
    end


    if $share_home
      config.vm.synced_folder ENV['HOME'], ENV['HOME'], id: "home", :nfs => true,
:mount_options => ['nolock,vers=3,udp']
    end


    if File.exist?(CLOUD_CONFIG_PATH)
      config.vm.provision :file, :source => "#{CLOUD_CONFIG_PATH}", :destination =>
"/tmp/vagrantfile-user-data"
      config.vm.provision :shell, :inline => "mv /tmp/vagrantfile-user-data /var/lib/co-
reos-vagrant/", :privileged => true
    end


  end
 end
end
```

## Liite 3. Cloud-config

```
#cloud-config
---
coreos:
 etcd2:
  discovery: https://discovery.etcd.io/3f94830822e72dc4361439f2649146c5
  advertise-client-urls: http://$public_ipv4:2379
  initial-advertise-peer-urls: http://$public_ipv4:2380
  listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
  listen-peer-urls: http://$public_ipv4:2380,http://$public_ipv4:7001
 units:
 - name: etcd2.service
  command: start
```